Tivoli® Directory Integrator

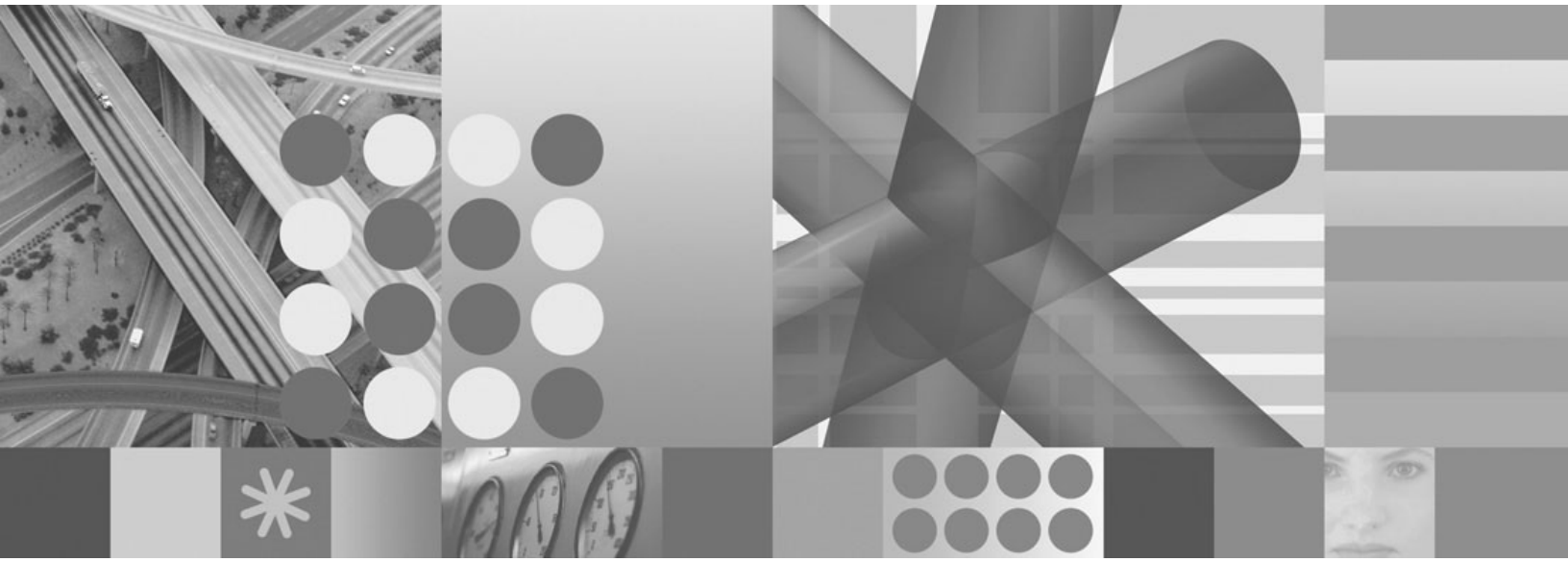
IBM

IBM Tivoli Directory Integrator 6.1.1: Users Guide

SC32-2568-01

Tivoli® Directory Integrator

IBM



IBM Tivoli Directory Integrator 6.1.1: Users Guide

SC32-2568-01

Note

Note: Before using this information and the product it supports, read the general information under Appendix F, "Notices," on page 247.

Second Edition (February 2007)

This edition applies to version 6.1.1 of the IBM Tivoli Directory Integrator and to all subsequent releases and modifications until otherwise indicated in new editions.

Preface

This document contains the information that you need to develop solutions using components that are part of the IBM® Tivoli® Directory Integrator.

Who should read this book

This book is intended for those responsible for the development, installation and administration of solutions with the IBM Tivoli Directory Integrator.

Tivoli Directory Integrator components are designed for network administrators who are responsible for maintaining user directories and other resources. This document assumes that you have practical experience installing and using both IBM Tivoli Directory Integrator and IBM Directory Server.

The reader should be familiar with the concepts and the administration of the systems that the developed solution will connect to. Depending on the solution, these could include, but are not limited to, one or more of the following products, systems and concepts:

- IBM Directory Server
- IBM Tivoli Identity Manager
- IBM Java™ Runtime Environment (JRE) or Sun Java Runtime Environment
- Microsoft® Active Directory
- PC and UNIX® operating systems
- Security management
- Internet protocols, including HTTP, HTTPS and TCP/IP
- Lightweight Directory Access Protocol (LDAP) and directory services
- A supported user registry
- Authentication and authorization
- SAP/R3

Publications

Read the descriptions of the IBM Tivoli Directory Integrator library and the related publications to determine which publications you might find helpful. After you determine the publications you need, refer to the instructions for accessing publications online.

IBM Tivoli Directory Integrator library

The publications in the IBM Tivoli Directory Integrator library are:

IBM Tivoli Directory Integrator 6.1.1: Getting Started
A brief tutorial and introduction to IBM Tivoli Directory Integrator 6.1.1.

IBM Tivoli Directory Integrator 6.1.1: Administrator Guide
Includes complete information for installing the IBM Tivoli Directory Integrator. Includes information about migrating from a previous version of IBM Tivoli Directory Integrator. Includes information about configuring the logging functionality of IBM Tivoli Directory Integrator. Also includes information about the security model underlying the Remote Server API.

IBM Tivoli Directory Integrator 6.1.1: Users Guide
Contains information about using the IBM Tivoli Directory Integrator 6.1.1 tool. Contains instructions for designing solutions using the IBM Tivoli Directory Integrator tool (**ibmditk**) or running the ready-made solutions from the command line (**ibmdisrv**). Also provides information about interfaces, concepts and AssemblyLine/EventHandler creation and management. Includes examples to create interaction and hands-on learning of IBM Tivoli Directory Integrator 6.1.1.

IBM Tivoli Directory Integrator 6.1.1: Reference Guide
Contains detailed information about the individual components of IBM Tivoli Directory Integrator 6.1.1 AssemblyLine (Connectors, EventHandlers, Parsers, Plug-ins, and so forth).

IBM Tivoli Directory Integrator 6.1.1: Problem Determination Guide
Provides information about IBM Tivoli Directory Integrator 6.1.1 tools, resources, and techniques that can aid in the identification and resolution of problems.

IBM Tivoli Directory Integrator 6.1.1: Messages Guide
Provides a list of all informational, warning and error messages associated with the IBM Tivoli Directory Integrator 6.1.1.

IBM Tivoli Directory Integrator 6.1.1: Password Synchronization Plug-ins Guide
Includes complete information for installing and configuring each of the five IBM Password Synchronization Plug-ins: Windows Password Synchronizer, Sun ONE Directory Server Password Synchronizer, IBM Directory Server Password Synchronizer, Domino Password Synchronizer and Password Synchronizer for UNIX and Linux®. Also provides configuration instructions for the LDAP Password Store and MQe Password Store.

IBM Tivoli Directory Integrator 6.1.1: Release Notes
Describes new features and late-breaking information about IBM Tivoli Directory Integrator 6.1.1 that did not get included in the documentation.

Related publications

Information related to the IBM Tivoli Directory Integrator is available in the following publications:

- IBM Tivoli Directory Integrator 6.1.1 uses the JNDI client from Sun Microsystems. For information about the JNDI client, refer to the *Java Naming and Directory Interface™ 1.2.1 Specification* on the Sun Microsystems Web site at http://java.sun.com/products/jndi/1.2/javadoc/index.html.
- The Tivoli Software Library provides a variety of Tivoli publications such as white papers, datasheets, demonstrations, redbooks, and announcement letters. The Tivoli Software Library is available on the Web at: http://www.ibm.com/software/tivoli/library/
- The *Tivoli Software Glossary* includes definitions for many of the technical terms related to Tivoli software. The *Tivoli Software Glossary* is available on the World-Wide Web, in English only, athttp://publib.boulder.ibm.com/tividd/glossary/tivoliglossarymst.htm

Accessing publications online

The publications for this product are available online in Portable Document Format (PDF) or Hypertext Markup Language (HTML) format, or both in the Tivoli software library: http://www.ibm.com/software/tivoli/library.

To locate product publications in the library, click the **Product manuals** link on the left side of the Library page. Then, locate and click the name of the product on the Tivoli software information center page.

Information is organized by product and includes READMEs, installation guides, user's guides, administrator's guides, and developer's references as necessary.

Note: To ensure proper printing of PDF publications, select the **Fit to page** check box in the Adobe Acrobat Print window (which is available when you click **File->Print**).

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully. With TDI 6.1.1, you can use assistive technologies to hear and navigate the interface. After installation you also can use the keyboard instead of the mouse to operate all features of the graphical user interface.

Accessibility features

The following list includes the major accessibility features in TDI 6.1.1:

- Supports keyboard-only operation.
- Supports interfaces commonly used by screen readers.
- Discerns keys as tactually separate, and does not activate keys just by touching them.
- Avoids the use of color as the only way to communicate status and information.
- Provides accessible documentation.

Keyboard navigation

This product uses standard MicrosoftWindows® navigation keys for common Windows actions such as access to the File menu, copy, paste, and delete. Actions that are unique to TDI use TDI keyboard shortcuts. Keyboard shortcuts have been provided wherever needed for all actions.

Interface Information

The following points include accessibility features of the TDI 6.1.1 user interface and documentation:

- Steps for changing fonts, colors, and contrast settings in the Config Editor (CE):
  1. Type **Alt-F** to access the CE **File** menu. Using the downward arrow, select **Edit Preferences** and press **Enter**.
  2. Under the **Appearance** tab, select **Theme** settings to change the font.
  3. Under **Theme Colors**, select the colors for the CE, and by selecting colors, you can also change the contrast.
- The TDI 6.1.1 Information Center and its related publications are accessibility-enabled for the JAWS screen reader and the IBM Home Page Reader. You can operate all documentation features using the keyboard instead of the mouse.

Vendor software

The IBM Tivoli Directory Integrator installer uses the FLEXnet Publisher Installation Module (FNPIM).

Related accessibility information

Visit the *IBM Accessibility Center* at http://www.ibm.com/able for more information about IBM's commitment to accessibility.

Contacting IBM Software support

Before contacting IBM Tivoli Software support with a problem, refer to IBM System Management and Tivoli software Web site at:

http://www.ibm.com/software/sysmgmt/products/support/

If you need additional help, contact software support by using the methods described in the *IBM Software Support Handbook* at the following Web site:

http://techsupport.services.ibm.com/guides/handbook.html

The guide provides the following information:

- Registration and eligibility requirements for receiving support
- Telephone numbers and e-mail addresses, depending on the country in which you are located
- A list of information you must gather before contacting customer support

Contents

Chapter 1. Introduction

Examples complementing this manual are in the *install_directory*/examples directory in the IBM Tivoli Directory Integrator.

General concepts

The following list contains a discussion of some of the general concepts that can be found in the IBM Tivoli Directory Integrator documentation:

Program components and interface

IBM Tivoli Directory Integrator includes IBM Java version 1.5.

IBM Tivoli Directory Integrator (TDI) consists of these three programs:

- The Config Editor (CE)
- The Server
- Administration and Monitoring Console (AMC)

TDI is a tool for creating and enforcing rules for how data flows between systems. These rules are stored in XML documents called *Configs* that you write, test and maintain using the Config Editor (CE). You deploy a Config by starting a TDI Server and pointing it at one or more Configs to run. Configs can be assigned to a Server at startup, or dispatched to it using the API.

Each individual rule in TDI is called an *AssemblyLine* (AL), and each AL is built by connecting together a series of components that handle the various services, data stores, transports, APIs and formats covered by the rule.

The Config Editor

The Config Editor is a graphical user interface that you can start by initiating the *ibmditk* batch file or script, which sets up the Java Virtual Machine (JVM) environment parameters and then starts the Config Editor.

This tool is used to build your integration solution Configs, and enables you to work with multiple Configs at the same time. Configs are stored as highly structured XML documents and can be encrypted.

The Config Editor is normally used to edit local Config files, but if a remote TDI server is set up correctly (see *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*), you can also use the Config Editor to edit files that reside on a remote computer (see "Remote" on page 129). Remote editing is particularly useful when you are working on a system where you do not have access to a local graphical screen.

The Config Editor also offers the option of starting AssemblyLines and of stepping through them. See "Debugging" on page 154. Even though both the local and the remote CE can start ALs, do not use these editing tools in a production environment. See the Administration and Monitoring Console (AMC) if you need a tool for monitoring your ALs. See AMC .

In no case does saving a configfile (remote or local) make the Server reload the file. Old Config files that are running continue to run until stopped and restarted.

- See Chapter 3, "The Config Editor," on page 117 for more information about the IBM Tivoli Directory Integrator Config Editor.

The Server

When you have a Config, you can then deploy your solution with the IBM Tivoli Directory Integrator Server batch file or script, *ibmdisrv*, which sets up the JVM environment and then launches the Server.

In addition to commands placed in the Config itself, there are a number of command-line options for controlling server behavior and its handling of the Config. Once a TDI Server is running, you can manage it with AMC or the Command-line Interface (CLI), or directly through calls to the TDI API.

- See Chapter 6, "TDI command-line options," on page 215.

Administration and Monitoring Console (AMC)

The IBM Tivoli Directory Integrator AMC is a Web-based tool for administering and monitoring TDI solutions. AMC communicates to TDI servers over the Remote Server API.

See the "Administration and Monitoring Console" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide* for more information.

IBM API

Full technical documentation ("JavaDocs") of all available internal Java objects (which can be reached from script language) is found in the installation package (*root_directory*\docs\api\index.html file). It can be launched in a browser by selecting **Help>Low Level API** in the Config Editor.

Note that you can specify which browser TDI should use under **File > Edit Preferences > Misc Settings**.

Script objects

Script objects are available everywhere inside the AssemblyLines (provided their context is valid). Some major objects are described in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Chapter 2. IBM Tivoli Directory Integrator concepts

An IBM Tivoli Directory Integrator Config describes the various rules for how data is to be transformed and transferred. Each rule is called an AssemblyLine (AL), and each AL includes one or more components that have been configured and linked together to form a solution. These components can be used to access systems and services, like *Connectors* and *Functions*, or components that control how data flows down an AssemblyLine, such as Branches, Loops and Switches. There is also an AttMap component for changing data attributes, and a place to drop in your own script logic called a Script component. Finally, both Connectors and Functions can use Parsers to turn incoming byte streams into structured data (for example, parsing XML) or the other way around (for example, writing XML).

Note: Additionally, a pre-6.1 Config can contain components called EventHandlers (EHs). EHs reside outside AssemblyLines and are used to service incoming events, launch AssemblyLines and then dispatch this event data to them. This is considerably less efficient than having ALs that do their own event-handling. With this version of IBM Tivoli Directory Integrator, the transition is complete from the concept of EventHandlers to event-handling directly in AssemblyLines. This is done using Connectors in Server or Iterator mode. Although EventHandlers are still handled by the Server, and can still be edited in the Config Editor for pre-6.1 Configs, future versions will eventually drop this support; so therefore best practice is to replace EventHandlers in TDI solutions as quickly as possible.

TDI components are described in the following sections, along with other relevant concepts.

The Entry object (TDI data model)

One of the cornerstones of understanding TDI is knowing how data is stored and transported within the system. This is done using an object called an *Entry*. The Entry object can be thought of as a "Java bucket" that can hold any number of Attributes (none, one or many).

Attributes are also bucket-like objects in TDI. Each Attribute can contain zero or more *values*, these being the actual data values that are read from (and written to) connected systems. Attribute *values* are Java objects as well – strings, integers and timestamps; whatever is needed to match the native type of this data value – and a single Attribute can readily hold values of different types. However, the values of a single Attribute will tend to be of the same type in most data sources.

Although this *Entry-Attribute-value* paradigm matches nicely to the concept of Lightweight Directory Access Protocol (LDAP) directory entries, this is also how rows in databases is represented inside TDI, as are records in files, IBM Lotus® Notes® documents and HTTP pages received over the network. All data – from any source that TDI works with – are stored internally as Entry objects with Attributes and their values.

There are a handful of Entry objects that are created and maintained by TDI. The most visible instance is called the **Work Entry**, and it serves as the main data carrier in an AssemblyLine (AL). This is the bucket used to transport data down the AL, passing from one component to the next.

The Work Entry is available for use in scripting through the pre-registered variable 'work', giving you direct access to the Attributes being handled by an AssemblyLine (and their Values). Furthermore, all Attributes carried by the Work Entry are displayed in the Config Editor in the **Work Entry** window under the Data Flow component list of an AssemblyLine.

The AssemblyLine

An AssemblyLine is a set of components strung together to move and transform data. It is the unit-of-work in TDI and typically represents a flow of information from one or more data sources to one or more targets. Data to be processed is fed into the AL one *Entry* at a time, where these Entries carry Attributes with *values* coming from directory entries, database rows, e-mails, *Notes* documents, records or similar data objects. Each Entry carries *Attributes* that hold the data values read from fields or columns in the source system. These Attributes are renamed, reformatted or computed as processing flows from one component to the next in the AL. New information can be "joined" from other sources and all or parts of the transformed data can be written to target stores or sent to target systems as desired.

It is important to keep in mind that the AssemblyLine is designed and optimized for working with one item at a time. However, if you want to do multiple updates or multiple deletes (for example, processing more than a single item at the time) then you must write AssemblyLine scripts to do this. If necessary, this kind of processing can be implemented using your choice of script languages, Java libraries and standard IBM Tivoli Directory Integrator functionality (such as pooling the data to a sorted data store, for example with the JDBC Connector, and then reading it back and processing it with a second AssemblyLine).

The AssemblyLine has a **Data Flow** tab in the Config Editor (CE). This is where the list of components that make up this AL are kept. The components list is divided into two sections: **Feeds** and **Flow**. **Feeds** holds Connectors that are used to 'feed' the AL with a stream of data entries. For example, a FileSystem Connector that is reading and parsing a CSV file, or an HTTP Server Connector servicing connections with browser clients. The **Flow** section holds Connector and other components that are used to process and aggregate data coming from Feeds.

Note: For an introduction to how an AssemblyLine operates, see *IBM Tivoli Directory Integrator 6.1.1: Getting Started*.

And now an important note on the naming of your Config elements (AssemblyLines, Connectors, Functions, Attributes and so forth). All components in an AL are automatically registered as script variables. So if you have a Connector called **ReadHRdump** then you can access it directly from script using the **ReadHRdump** variable. As a result, you will want to name your AL components as you would script variables: Use alphanumeric characters only, do not start the name with a number, and do not use special national characters (for example, å, ä), separators (apart from underscore '_'), white space, and so forth.

There is always an alternative method for accessing an AL component - for example, the task.getConnector() function - but a conscious naming convention is always advisable.

Starting an AssemblyLine in TDI is a fairly costly operation, as it involves the creation of a new Java thread and usually sets up connections to one or more data sources. Consider carefully if your solution design could be made to work with

fewer, rather than more, distinct AssemblyLines, where each AssemblyLine does more work; for example by using Branches or Switches to define multiple operations handled by a single AL. Note that each operation can still be implemented as a separate AssemblyLine, but these can be embedded "hot-and-ready" into a single AL that dispatches work to them by using the AL Connector or AL Function. This also allows you to leverage features like Global Connector Pools to manage resource usage and boost performance and scalability.

Note: Messages for all TDI components (Connectors, EventHandlers, Parsers, and so forth) are formatted in TMSXML. TMSXML formatting facilitates Problem Determination Serviceability. In TDI 6.1, the TMSXML format messages are automatically converted to properties files during the build process. Regardless, TDI 6.1 components can be dropped into TDI 6.0 build. See the *IBM Tivoli Directory Integrator 6.1.1: Messages Guide* for more information.

AssemblyLines can include the following components:

Connectors

Connectors are used to abstract away the details of some system or store, giving you the same set of access features. This lets you work with a broad range of disparate technologies and formats in a consistent and predictable way. A typical AL has one Connector providing input and at least one Connector writing out data. A rich Connector library is one of the strengths of IBM Tivoli Directory Integrator, but you can of course write your own Connectors (just look at the Script Connector or Script Parser to see how easy this is).

Each Connector is designed for a specific protocol, API or transport and handles marshalling data between the native type of the connected system and Java objects. Unlike the other components, Connectors have a **Mode** setting that determines how this Connector accesses its connected system.

Note: AssemblyLines can consist of as many, or as few, Connectors as required to solve the specific Data Flow. There is no limitation in the system. However, best practice is to keep an AssemblyLine as simple as possible in order to maximize maintainability.

When used in an AL, Connectors provide an "Initialize" option to control when the component is set up (for example, connections made, resources bound, and so forth). By default, all Connectors initialize when the AssemblyLine starts up (AL Startup Phase).

Connectors in Server or Iterator mode appear in the **Feeds** section—although Iterators can also be used in the **Flow**. A Feeds Connector is responsible for 'feeding' the Flow section components with a new Work Entry for each cycle that the AL makes. The Work Entry is passed from component to component in the Flow section (following any Branching logic you've implemented) until the end of the Flow is reached. At this point, end-of-cycle behavior kicks in and, for example, the Iterator gets the next Entry from its source and passes it to the Flow section for a new cycle.

While an Iterator in the Feeds section will actually drive the Flow, an Iterator in the Flow section will simply get the next Entry and offer its data Attributes for Input Mapping into the Work Entry.

Note: You can put a Connector in Iterator Mode into the Flow Section. As such, the Iterator works in the same way as it would in the Feeds: it

is initialized (including building its result set with the selectEntries call) during AL startup and retrieves one Entry (getNextEntry) on each cycle of the AL. However, an Iterator in the Flow section does not drive the AL itself, as it would do in Feeds.

Feeds section behavior is different for Server and Iterator modes: An Iterator expects to be the first component running in your AL, and it will only read its next Entry if the Work Entry does not already exist. If the AL is passed an Initial Work Entry, then Iterators do not read any data for this first cycle. It also means that Iterators run in a series, with the second one beginning to return Entries once the first one has reached end-of-data and returned nothing (*null*).

Server mode, on the other hand, causes the Connector to launch a Server *listener* thread (for example, to an IP port or event-notification callback) and then pass control to the next Feeds Connector.

When you call an AL (locally or remotely) using the AssemblyLine Function component (AL FC), if you use the manual cycle mode then only the Flow section components are used each time the FC executes the call.

There is a **Connectors** folder in the Config Browser where you can maintain your library of configured Connectors. This is also where Connector Pools are defined.

Functions (Function components, FCs)

A Function is a component much like a Connector, except that it does not have a mode setting. Whereas Connectors provide standard access verbs for connected systems (Lookup, Delete, Update, and so forth), Functions on the other hand only perform a single operation, like pushing data through a Parser, dispatching work to another AssemblyLine or making a Web service call. Functions can appear anywhere in the Flow section of an AL. The "Functions" library folder in the Config Browser can be used to manage your library of Function Components.

Like Connectors, Functions in AssemblyLines provide an **Initialize** option to decide when this component starts . By default, Functions initialize during AL startup.

Scripts (Script components, SCs)

A Script is a block of JavaScript™ that can be placed in an AssemblyLine just like a Connector or Function. Unlike the other types of AL components, the Script does not have any predefined behavior or mode; you can implement behavior and mode with your script. A library of Scripts can be stored under the **scripts** library folder in the Config Browser.

AttributeMaps (AttMaps)

AttMaps are free-standing collections of individual Attribute Maps (simple, Advanced and Expressions). These can be copied to and from other Attribute Maps or any Input or Output Map in a Connector or Function, individually or as a whole. To copy individual Attribute Mappings, select them in the map, right-click and choose **Copy**. Then right-click in another Attribute Map and select Paste. To copy an entire Input or Output Map, click **Copy to library**. You can drag any AttributeMap from the library folder in the Config Browser and onto the **Inherit from** area for an Input or Output Map in order to inherit from this map.

AttMap components, support dragging to Input or Output Maps on Connectors and Functions. AttMap components also provide this reciprocal feature: you can use the**Copy to Resources** button for all Input and Output Maps. The **Copy to Resources** button allows you to copy these Input or Output Maps on Connectors and Functions as AttMaps to Resources.

Note: A simply mapped Attribute can be used for either Input or Output Maps. However, JavaScript or Expression-based Attribute Mappings will probably contain explicit references to Entry objects like work or conn. As such, you must make sure that they fit the context in which you want to use them.

You can also right-click any Attribute in a map and copy it, allowing you to paste the entire Attribute Map definition for this item to any other Map or Schema (like for an AL Operation).

Branch Components

Branch components affect the order in which the other components (Connectors, Scripts, Functions, AttributeMaps and other Branch components) are executed. Branch components come in three varieties:

- Simple (also called just Branch)
- Loops (Branches that loop)
- Switches (Branches that share the same expression)

Branches can appear anywhere in the Flow section, but there is no library folder for them in the Config Browser.

The same script call is used to exit any type of Branch: system.exitBranch(). See “Exiting a Branch (or Loop or the AL Flow)” on page 18 for more details.

The three Branch component types are:

Branch

While each type of Branch lets you define alternate routes for AssemblyLine processing, this simplest form lets determine the action in case: "If this situation occurs, then take this action." You define what "situation occurs" means by setting up Conditions that must be met; for example, by comparing data values or checking the result of some operation. If the conditions are true, the components attached under this Branch are executed.

The Branch provides an interface that allows you to define Conditions based on any data in the TDI Server: Attribute values, parameters settings, externally accessible properties, and any information available using JavaScript (like operating system calls for disk or memory usage). Multiple Conditions are ANDed or ORed, depending on the **Match Any** check box setting.

This simplest form of Branch component also supports three subtype settings, IF, ELSE-IF and ELSE, that you can select.

The available options are:

IF Can appear anywhere within the AssemblyLine Flow, the IF Branch provides an alternative track for processing to follow if Conditions are true. Once the components under the Branch are executed, control passes to the first component *after* this Branch. If you do not want this to

happen, you must either add an ELSE or ELSE IF Branch, or exit the Branch with a scripted call to system.exitBranch().

ELSE-IF
Identical to the IF Branch, except it can only appear immediately following an IF or ELSE-IF Branch.

ELSE Only allowed immediately after an IF or ELSE-IF Branch, the ELSE Branch has no Conditions. Its components are processed only if no preceding IF or ELSE-IF Branch was true.

As mentioned above, you can prematurely exit a Branch by means of scripting, as described in "Exiting a Branch (or Loop or the AL Flow)" on page 18.

Loop

The Loop component provides functionality for adding cyclic logic within an AssemblyLine. Loops can be configured for three modes of operation: based on Conditions (like a simple Branch), based on a Connector (in Iterator or Lookup mode) or based on the values of an Attribute:

Conditional
Just as with a simple Branch, you can define Conditions that control Loop behavior. The Loop will continue to cycle as long as the Conditions are met, and will stop as soon as they fail. The details pane for Loops is the same as for the simple Branches described in the previous section.

Connector
This method lets you set up a Connector in either *Iterator* or *Lookup* mode, and will cycle through your Loop flow for each Entry returned. The details pane of this type of Loop contains the Connector tabs necessary to configure it, connect and discover attributes and set up the Input Map.

Note that you have a parameter called **Init Options** with which you can instruct the AL to either:

- **Do Nothing** which means that the Connector will not be prepared in any way between AL cycles.
- **Initialize and Select/Lookup** causing the Connector to be re-initialized for each AL cycle.
- **Select/Lookup Only** keeps the Connector initialized, but redoes either the Iterator select or the Lookup, depending on the Mode setting.

Note also that there is a **Connector Parameters** tab, which functions similarly to an Output Map in that you can select which Connector parameters are to be set from **work** Attribute values.

This brings us to the topic of how Looping with an Iterator differs from doing so based on Lookup mode. Both options perform *searches* that create a result set returned for looping. For Iterator mode, the result set is controlled exclusively by the parameter settings of this component.

Lookup mode, on the other hand, uses Link Criteria to define search or match rules. Since it frees you from having to code Hooks like "On No Match" or "On Multiple Found", this is the preferred way of doing searches that may not always return one (and just one) matched Entry.

Attribute Value

By selecting any Attribute available in the work Entry, the Loop flow will be executed for each of its values. Each value is passed into the Loop in a new Work Entry attribute named in the second parameter. This option allows you to easily work with multi-valued attributes, like group membership lists or e-mail.

You can prematurely exit a Loop by means of scripting, as described in "Exiting a Branch (or Loop or the AL Flow)" on page 18.

Switch

Unlike expressions used in the Conditions of Branches and Loops, the Switch expression can result in more values than just *true* or *false*. For example, you could Switch on the value of an Attribute, or the operation requested when this AL was called from another AssemblyLine or process. Under the Switch component, you add a Case for each constant value of the Switch expression that you want to handle. So for example, if you set up the Switch to use the delta operation code in the Work Entry, your Cases would be for values like "add", "delete" and "modify."

In an AL Switch-Case construct, multiple cases can be active at the same time. TDI checks each case, just as it would a series of standard IF Branches. The following example shows how multiple cases work:

```
work.setAttribute("test","abc");

Switch work.test
  Case startsWith("a"): this is true
  Case contains ("bc"): this is true
  Case length=3: this is true
```

The two Switch work.test expressions that are true will trigger Switch execution.

You can prematurely exit a Switch Case by means of scripting, as described in "Exiting a Branch (or Loop or the AL Flow)" on page 18.

In addition to the above components that can appear alone in an AssemblyLine, there is also a **Parser** component. Parsers are used by Connectors and Functions to interpret the structure in a byte stream, or to write a structure to one.

AssemblyLine flow and Hooks

AssemblyLines provide built-in automated behavior that helps you rapidly build and deploy your data flows. This automated behavior is detailed in the TDI Flow Diagrams in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*. In addition, Connectors and Functions have their own behaviors (Connector behavior depending on the mode setting) and these are also shown in the Flow Diagrams.

Throughout these built-in logic flows are numerous *waypoints* where you can add your own scripted logic to extend built-in behavior, or to override it completely. These waypoints are called "Hooks" and are available for customizing under the Hooks tab of all Connectors and Functions, as well as of the AssemblyLine itself.

You can enable and disable Hooks according to whether a particular Hook is applicable to the AssemblyLine you are running. When you disable a Hook, you do not break its inheritance in the connector of which it is a part.

When an AssemblyLine is launched, it goes through three phases: Startup, Data flow and Shutdown. During Startup, Prolog Hooks are available for reconfiguring components before they are initialized. In the Data flow phase, each Work Entry fed into the AssemblyLine is passed down the Flow components for processing.

Finally, during Shutdown, Epilog Hooks can be used to carry out end-of-job work, like checking and reporting on error status, or storing state data for the next time the AL is started.

Startup Phase

At this point the Server is instructed to load and run an AssemblyLine. The Server uses the blueprint stored in the Config to set up the AL. If a TaskCallBlock (TCB) is passed into the AssemblyLine, then its contents are evaluated (which can result in changes to AL component parameters). At this point, Prolog Hook flows are initiated.

Global Prologs

First, if any "Global" Prologs are defined then these are evaluated. Global Prologs are Scripts under the library folder in the Config Browser that have been included in the AssemblyLine. This is typically done in the **Config** tab of the AssemblyLine by selecting the Scripts to run at AL startup. After all Global Prologs are finished, the AL Prolog Hooks are called.

AL Prolog Hooks (Before Initialize)

First the AssemblyLine Prolog - Before Initialize Hooks is invoked. After this, all Connectors and Functions configured to Initialize "at Startup" go through their initialization phase, which also invokes their Prolog Hooks, as seen in the next point.

Connector/Function Initialization

The initialization sequence is performed for each Connector and Function with Initialization set to "at Startup." These are started in turn as defined by their order in the AssemblyLine. For each Connector or Function the flow is as follows:

1. The component's **Prolog – Before Initialize** Hook is called.
2. The component is started (for example, connecting to its underlying data source, target or API).
3. For Connectors in Iterator mode, the **Prolog – Before Selection** Hook is processed, and the Connector performs the entry selection (triggers a data source specific call, like performing an SQL SELECT or an LDAP search).
4. For Iterators, the **Prolog – After Selection** Hook is evaluated.
5. The **Prolog – After Initialize** Hook is called, ending the sequence.

If initialization fails for a Connector, then AssemblyLine flow passes to the **Prolog – On Error** Hook where you can deal with this error.

The Reconnect feature allows you to configure a Connector to automatically attempt to re-establish its connection if an error occurs during setup or data access. These settings are found under the Connector's **Connection Failure** tab.

Note: Script Connectors (Connectors implemented using JavaScript) are evaluated at this stage so that required Connector functions are registered and initialization code is executed.

AL Prolog Hooks (After Initialize)
The **AssemblyLine Prolog - After Initialize** Hook is executed. Completion of this Hook signals the end of Start up Phase, and the beginning of Data flow Phase.

Data flow Phase

AssemblyLine Start of Cycle Hook
This Hook is invoked at the start of every cycle before Feeds or Flow components.

AssemblyLine Cycle
Control is passed to the first component in the flow, typically a Server or Iterator mode Connector in the Feeds section.

If you have one or more Iterators in the AssemblyLine, then the first one starts the cycle by retrieving the next Entry from its result set and mapping Attributes into the Work Entry. The resulting Work Entry is passed to Flow section components, starting at the top of the list as seen in the CE.

For Server mode Connectors, a listener process is launched that waits for incoming client connections. When a connection request is detected, the Connector clones itself, accepts the connection and then switches itself to Iterator mode in order to feed data from the client into the Flow section for processing. Either way, you get an Iterator driving Work Entries to the Flow components. (Meanwhile, the original Server mode Connector waits for additional incoming connection requests.)

If your AssemblyLine neither has an Iterator Mode or Server Mode Connector, then you have a one-shot AssemblyLine typically used to process an Initial Work Entry (IWE) fed by another calling process.

End-of-Cycle
When the last Flow component is executed, one of three things can happen: If the current Work Entry came from an Iterator, control is passed back to the Iterator to get the next Entry from its source; In the case of a Server mode Connector, a reply is made to the client; for an AL that is called in manual cycle mode, the thread is passed back to the caller so that results can be accessed. There is no specific Hook at this point, although this can be added to your AssemblyLine by inserting a Script at the end.

End-of-Data
End-of-data is an Iterator mode Hook that is called when the end of the input data set is reached. At this point, control is either passed to the next Feeds Connector, or the AssemblyLine goes into Shutdown Phase.

Shutdown Phase
At this point, AL processing has either completed normally, or aborted due to an error.

AssemblyLine Epilog - Before Close Hook
The AssemblyLine Hook called **Epilog – Before Close** is processed.

Connectors/Function Close flow
Each Connectors' and Functions' Epilog Hooks are called in the order that they appear in the Config Editor:

1. The **Before Close** Hook.
2. The close operation is carried out (for example, closing a connection or release an API callback).
3. The **After Close** Hook.

AssemblyLine Epilog - After Close Hook
Finally, the AssemblyLine **Epilog – After Close** Hook is run.

Server mode Connector Setup
When a Connector in Server mode starts, it goes into event listening mode. Once an event is received, it clones the AL and resumes waiting for more events. In the clone, meanwhile, the Connector switches itself to Iterator mode and passes control to the next component in the *Feeds* list. This process allows you to have multiple Server mode Connectors active and feeding data into the flow at the same time—one example would be to have several HTTP Server Connectors in Server mode listening to different ports, but feeding the same AL. Although Server mode Connectors are part of an AssemblyLine configuration, they run as separate processes (threads).

There is an additional set of Hooks that is evaluated for Connectors in this mode. The Hooks specific to Server mode functionality for dealing with incoming notifications/connections are:

Before Accepting connection
This Hook is called before the Connector goes into listening mode.

After Accepting connection
Once a connection is received, this Hook is invoked. Note that no data is available at this time. In order to examine incoming event information, use the Iterator Hooks like **After GetNext** or **GetNext Successful**.

Error on Accepting connection
This Hook is executed if an error occurs in any of the Server mode Hooks, or received from the data source during event listening.

- As mentioned previously, if you have more than one Connector in Iterator mode (see "Connector modes" on page 26), these Connectors are stacked in the order in which they are displayed in the configuration (top to bottom). For example, if you have two Iterators, **a** and **b**, then **a** is called until it returns no more entries before the AssemblyLine switches to **b**.
- If you have no Connectors in Iterator mode, and no **Initial Work Entry** (IWE) is provided to the AssemblyLine when it is started; for example, by a calling an

EventHandler, and if no **work** Entry is created in an AssemblyLine or Connector **Prolog** Hook, then the AssemblyLine still performs a single pass.

Finally, there is a **Shutdown Request** Hook where you can put code that is processed if the AssemblyLine is closed down properly due to a external request to shut down (that is, the AssemblyLine does not crash), enabling you to make it perform a graceful shutdown.

Special functions are available from the **system** object to skip or retry the current **work** entry, as well as to skip over a Connector, and so forth. See "Controlling the flow of an AssemblyLine" on page 53 for more details.

Improve termination and cleanup for critical errors

There are a number of TDI methods that allow catching and handling of TDI internal errors, as well as errors occurring in TDI Connectors, Parsers, Function Components, EventHandlers. These methods comprise:

- Error Hooks, in which JavaScript code can be written to handle an error - this method is accessible to TDI users
- Java try-catch-finally blocks, which make sure that a minor failure does not break the TDI server as well as that all errors are handled appropriately – such blocks are already put in place in the core TDI server classes.

The new JVM shutdown Hook feature improves the reliability of the TDI server. Java shutdown Hooks allow a piece of code to perform some processing after Control-C is pressed, or when the JVM is shutting down for some other reason, even System.exit.

Users can specify an external program to be started when the JVM is shutting down. This external program is started from within the JVM shutdown Hook. This external program is configured using an optional property in the global.properties or solution.properties file:

```
jvm.shutdown.hook=<external application executable>
```

Shell scripts and batch files can also be specified as the value of this property.

When the JVM shutdown Hook is called, nothing can be done to prevent the JVM termination. However, with the execution of an external program it is possible to perform customizable operations: for example, sending a message that the TDI server has terminated, carrying out clean up operations, or even restarting a new TDI server if so desired.

Custom exit/return codes

A new method is available for the com.ibm.di.server.RSInterface interface: shutdownServer(int aExitCode). This new method is available in JavaScript through a "main" variable, which references RSInterface and offers functions like runAL(). The specified aExitCode is then passed to the process that started TDI, allowing the caller to react to different exit codes; for example, in the script or batch file used to launch TDI.

Access using TDI API calls: The Server API Session interfaces are extended to provide the shutDownServer(int aExitCode) method. The invocation of this method will result in termination of the TDI Server with the supplied exit code.

A new method shutDownServer(Integer aExitCode) is available for DIServer MBean so that it can be accessed from the Java Management Extensions (JMX) context as well.

The com.ibm.di.server.RSInterface has a new method to terminate the TDI Server with a specific exit code:

```
/**
 * Set the shutdown request flag and specify an exit code
 */
public void shutdownServer (int aExitCode);
```

New methods are also available for the following Server API Interfaces:

com.ibm.di.api.local.Session

```
/**
 * Shuts down the TDI Server with the specified exit code.
 * @throws DIException if an error occurs while shutting down the server.
 */
public void shutDownServer (int aExitCode)
    throws DIException, RemoteException;
```

com.ibm.di.api.remote.Session

```
 /**      * Shuts down the TDI Server with the specified exit code.      *
@throws DIException if an error occurs while shutting down the server.
  */     public void shutDownServer (int aExitCode)         throws DIException,
RemoteException;
```

com.ibm.di.api.jmx.mbeans.DIServerMBean

```
/**      * Shuts down the TDI Server with the specified exit code.      *
@throws DIException if an error occurs while shutting down the server.
  */     public void shutDownServer (Integer aExitCode)         throws DIException;
```

Server API Notification Enhancements

The Server API has features improvements in several areas, including authentication, locking of Config files, custom notifications, Server shutdown events and documentation. These improvements are described in the following sections.

Server API Script Object: In order to make script access to the API easier, a new session variable references a local session to the TDI Server.

See the "Server API" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

Remote Config Editing: Loading a remote Config for editing is differs from opening a remote Config in TDI 6.0. Configurations loaded for editing are not started in the usual way on the TDI Server. You have two options for loading a configuration for edit:

- Either the configuration is only loaded for editing and cannot be started at all.
- Or the configuration is loaded for editing and a temporary Config Instance is started on the Server so that the configuration can be tested while being edited.

See “Remote” on page 129 for more information.

TDI Config Folder: A new TDI Server property api.config.folder is available in Solution or Global Properties for specifying a folder on the local disk. This folder (and its sub-folders) is where Configs that can be browsed and loaded using the API are stored.

Note: Implementing your own functionality and accessing it from the Server API—both local and remote—is simplified. You can drop your own JAR file in the TDI classpath and access it from the Remote Server API without having to deal with RMI.

See the "Server API" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

Load for Editing: TDI 6.1 does not allow modification of the Config object of an active Config Instance. Server API users can still get the Config object for an active Config Instance, but the following calls for setting the Config object and saving it on the disk throws an exception when executed on a normal running Config Instance.

See the "Server API" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

Configuration Locking: The Server API internally tracks all configurations loaded for editing. When another Server API user requests a configuration already loaded for editing, the method call will fail with an exception.

A new Server API call is available for checking whether a configuration is currently loaded for editing (locked). The lock on a configuration is released when you loaded the configuration for editing and you resave it cancel the update.

See the "Server API" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

Load for editing with temporary Config instance: This is a special version of the "load for edit" mechanism that causes a temporary Config Instance to be started as well. This allows for testing of the Config and its AssemblyLines while they are being changed, providing valuable functionality to tools like the TDI Config Editor.

See the "Server API" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

New Server API event for configuration update: A new Server API event "di.ci.file.updated" starts whenever a configuration that is locked is saved on the TDI Server.

This feature allows Server API clients to get notified on changes in Configs they are using, for example to reload them in order to run the latest version.

See the "Server API" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

New API calls: All configurations are identified through the relative file path of the configuration file according to the TDI Server configuration Config folder. All paths specified as parameters are relative to the Config folder itself (so "./" references the folder specified by the "api.config.folder" property).

See the "Server API" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information about the new calls that have been added to the local and remote Server API Session objects, as well as to the JMX interfaces.

Server shutdown event: A new Server API event notification is available to signal Server shutdown events. This event is available to Server API clients and JMX clients, both in local and remote context, and is of type `di.server.stop` for both the Server API and JMX notification layers.

See the "Server API" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

Custom server API event notifications: Server API functionality allows sending custom, user-defined event notifications.

See the "Server API" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

Authentication: In addition to previously supported authentication based on an SSL channel with client side authentication, SSL supports an option for using custom authentication. Furthermore, SSL client authentication can be switched off.

See the "Security and TDI" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide* for more information about authentication.

SSL support enhancements: TCP-based components all support SSL (when applicable).

Remote Config Editor SSL enhancement:

In TDI 6.0, the Remote Config Editor windows contained a check box labeled **SSL**. Users of the Remote Config Editor were supposed to specify whether SSL is enabled on the TDI Server. Actually the SSL option was not used for creating the connection, but instead used Remote CE functionality—running remote AssemblyLines and EventHandlers, as well as receiving their logs.

In TDI 6.1, the **SSL** option is added to the Config Editor windows. A method is added to the Server API Session Interface:

```
public boolean isSSLon();
```

This method returns **true** if the current Session is over SSL.

The Remote Config Editor uses this Server API method instead of forcing you to provide the **SSL** parameter, thus reducing confusion and improving usability.

Furthermore, the Remote Config dialog supports the authentication options, providing parameters for setting user name and password.

Server API authentication exception: When custom authentication is used and the script you specified indicates that the authentication has failed, an exception of type com.ibm.di.api.exceptions.AuthenticationException is thrown (by the createSession, password) method.

See the "Security and TDI" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide* for more information about authentication.

Server API JMX layer does not support custom authentication: The remote JMX layer of the Server API does not support the custom authentication mechanism. It will ignore the api.custom.authentication property. See the "Security and TDI" chapter of the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide* for more information about authentication.

Exiting a Branch (or Loop or the AL Flow)

Note: ELSE-IF and ELSE logic for Branches had been added. New keywords for the system.exitBranch() call have been added for Loops and Branches. Loops have been updated with a new continue method, as well as new keywords. The following new methods are available in the system object :

```
void continueLoop() throws com.ibm.di.exceptions.ContinueLoopException
void continueLoop (String name) throws com.ibm.di.exceptions.ContinueLoopException
```

Both of these methods throw the following exception:

```
com.ibm.di.exceptions.ContinueLoopException.
```

This exception causes the LOOP to continue. In case a loop name is provided, the program flow is transferred to the LOOP component with that name. If you want to exit a Branch, Loop, or Switch, or even built-in Branches like the AL Flow section, you use the `system.exitBranch()` method from a place where you can script, for example, a Hook, or even a Script Component. Calling `system.exitBranch()` with no parameters (or with an empty string) will cause the containing Branch to exit, and flow continues with the first component after the Branch.

You can also provide the method with a string parameter containing either:

One of the reserved keywords: ″Branch″, ″Loop″, ″Flow″, ″Cycle″ or ″AssemblyLine″ (case insensitive)

This will break the first Branch of this type, tracing backwards up the AssemblyLine. So if your script code is in a Branch within a Loop, and you execute the call `system.exitBranch("Loop")`, you will exit both the Branch and the Loop containing it. Using the reserved word ″Flow″ causes the flow to exit the Flow section of the AssemblyLine, continuing either to the response behavior in the case of a Server Mode Connector or to an active Iterator to read in the next Entry, or to AL shutdown (Epilogs, ...). The ″Cycle″ keyword passes control to the end of the current AL cycle, and does not invoke response behavior in Server Mode Connectors, while the "AssemblyLine" keyword will cause the AL to stop and shutdown.

All other values used in the exitBranch() call cause a break out of the branch/loop having the specified name. So, for example, the call exitBranch(“IF_LookupOk”) sends the flow after the containing Branch or Loop called “IF_LookupOk”. Note that unlike system.skipTo(), which will pass control to any named AL component, exitBranch() will cause processing to continue after the specified Loop/Branch.

The name of a Branch or Loop (case sensitive)

If you pass the name of a Branch or Loop in which your script call is nested, then control will pass to the component following it in the AL. If no Branch or Loop with this name is found (tracing backwards from the point of the call) then an error results.

Starting an AssemblyLine in the Config Editor – ibmditk

When you launch an AssemblyLine, it typically has a Feeds Connector operating in Iterator or Server mode, or you are passing in an Initial Work Entry (IWE) when the AL is started. If you don’t have an Iterator, no Connector in Server mode and no IWE is provided, then the AssemblyLine has no data feed and will not have a Work Entry to process. This does not prevent you from having other components that read, write or manipulate data, but they cannot be dependent on a Work Entry being passed in from outside the Flow section.

To launch the debugger, perform one of the following actions:

- Right click **AssemblyLine** and select **Debug** (**Step on Break** or **Step**).
- Select **Step (Pause)** and click **Run**.
- Type the **Alt+R** keyboard shortcut.

This causes the Config Editor to launch a separate instance of the TDI Server and connect to it using the API. The Config Editor then pipes over the Config and instructs the Server to run the selected AssemblyLine. In addition, the Server is told to send log output to the console, so that the Config Editor can capture this and display it in an Execute screen.

You can also run and test your AssemblyLines using the Debugger. This powerful tool is available using the **Run in debug mode** button or with the **Alt-D** keyboard shortcut.

Starting an AssemblyLine from another AL or script

Refer to "AssemblyLine parameter passing" for more information.

Accessing AL components inside the AssemblyLine

Each AL component is available as a pre-registered script variable with the name you chose for the component.

Note that you can dynamically load components with scripted calls to functions like `system.getConnector()`, although this is not for inexperienced users.[1]

AssemblyLine parameter passing

There are three ways for data to get into an AssemblyLine:

- Generating your own initial entry inside the AssemblyLine (for example, in the a Prolog script).
- Fed from one or more Iterators.
- Starting the AssemblyLine with parameters from another AssemblyLine using the AL Connector or AL Function Component, or using an API call.

If you want to start an AssemblyLine with parameters from another AssemblyLine or EventHandler, then you have a couple of options:

- Use the **Task Call Block** (TCB), which is the preferred method. The TCB is detailed below.
- Provide an Initial Work Entry directly.

  Note: These two options are provided for backward compatibility.

Task Call Block (TCB)

Basic Use: The Task Call Block (TCB) is a special kind of Entry object (like the Work Entry) used by a caller to set a number of parameters for an AssemblyLine. The TCB can provide you with a list of input or output parameters specified by an AssemblyLine (including operation codes defined in the AssemblyLine's **Operations** tab), as well as enabling the caller to set parameters for the AssemblyLine's Connectors. The TCB consists of the following logical sections:

- The Initial Work Entry passed to the AssemblyLine: `tcb.setInitalWorkEntry()`
- The Connector parameters: `tcb.setConnectorParameter()`

1. The Connector object you get from this call is a **Connector Interface** object, and is the data source specific part of an AssemblyLine Connector. When you change the *type* of any Connector, you are actually swapping out its data source intelligence (the Connector Interface) which provides the functionality for accessing data on a specific system, service or data store. Most of the functionality of an AssemblyLine Connector, including the Attribute Maps, Link Criteria and Hooks, is provided by the IBM Tivoli Directory Integrator kernel and is kept intact when you switch Connector types.

- The input or output mapping rules for the AssemblyLine (set in the Config Editor under the **Operations** tab)
- An optional user-provided *accumulator* object that receives all work entries from the AssemblyLine: `tcb.setAccumulator()`

For example, starting an AssemblyLine with an Initial Work Entry and setting the **filePath** parameter of a Connector called **MyInput** to `"d:\myinput.txt"` is accomplished with the following code:

```
var tcb = system.newTCB();                      // Create a new TCB
var myIWE = system.newEntry();                  // Create a new Entry object
myIWE.setAttribute("name","John Doe");          // Add an attribute to myIWE
tcb.setInitialWorkEntry ( myIWE )               // Set the IWE and parameter
                  // (below);
// Note that since this is a JavaScript string, we must "escape" the forward slash
//      or use a backslash (Windows syntax)
tcb.setConnectorParameter ( "MyInput", "filePath", "d:\myinput.txt" );

var al = main.startAL ( "MyAssemblyLine", tcb )  // Start the AL with the tcb;
al.join();                                       // Wait for AL to finish
```

Using an accumulator: As noted previously, you can also pass in an accumulator object to an AssemblyLine with the TCB. An accumulator can be any one of the following classes or interfaces:

java.util.Collection
: All work entries are cloned and added to the collection (for example, ArrayList, Vector, and so forth).

com.ibm.di.connector.ConnectorInterface (Connector Interface)
: The putEntry() method for this Connector Interface is called with the **work** Entry at the end of each AssemblyLine cycle.

com.ibm.di.parser.ParserInterface (Parser)
: The writeEntry() method is called for this Parser with the **work** Entry at the end of each AssemblyLine cycle.

com.ibm.di.server.AssemblyLine Component (AssemblyLine Connector)
: The add() method is called for this AssemblyLine Connector with the Work Entry at the end of each AssemblyLine cycle.

If the accumulator is not one of these classes or interfaces, an exception is thrown.

For example, to accumulate all work entries of an AssemblyLine into an XML file you can use the following script:

```
var parser = system.getParser ( "example_name.XML" ); // Get a Parser
// Set it up to write to file
parser.setOutputStream ( new java.io.FileOutputStream ( "d:/accum.xml" ));
parser.initParser();                                 // Initialize it.
tcb.setAccumulator ( parser );                       // Set Parser to tcb

var al = main.startAL ( "MyAssemblyLine", tcb );     // Start AL with tcb
al.join();                                           // Wait for AL to finish

parser.closeParser();    // Close the parser - this flushes and
          // closes the output file
```

Of course, you can configure a Connector instead of programming the Parser manually as in the following script:

```
var connector = system.getConnector("myFileSysConnWithXMLParser");
tcb.setAccumulator ( connector );

var al = main.startAL( "MyAssemblyLine", tcb);
al.join();

connector.terminate();
```

Typically, you initialize the TCB and then the TCB is used by the AssemblyLine. If the AssemblyLine has an Operations specification, the TCB remaps input attributes to the Initial Work Entry as expected by the AssemblyLine, and likewise for setting the result object. This is done so that the external call interface to an AssemblyLine

can remain the same even though the internal Work Entry names change in the AssemblyLine. Once the TCB is passed to an AssemblyLine, you must not expect anything more from the TCB. Use the AssemblyLine's `getResult()` and `getStats()` to retrieve the result object and statistics.

The TCB result mapping is performed before the Epilog so you can still access the final result before the AssemblyLine caller gets to it.

Disabling AssemblyLine components: In TDI version 6.0, each of the components (Connectors, Function Components, and so forth) hosted by an AssemblyLine are created and initialized when AssemblyLines initialize. In TDI version 6.1, you can specify that certain AssemblyLine components must not be created or initialized on AssemblyLine initialization. This is done by disabling those components using the Task Call Block (TCB).

AssemblyLine components are enabled by default.

In order to enable or disable a component in the AssemblyLine, you must call the `com.ibm.di.server. TaskCallBlock.setComponentEnabled(String name, boolean enabled)` method on the TaskCallBlock (TCB) object of the AssemblyLine. The name argument of the method specifies the name of the component to be enabled or disabled. The enabled argument of the method specifies whether the component is to be enabled or disabled.

The actual enabling or disabling of the AssemblyLine components happens in the `com.ibm.di.server. TaskCallBlock.applyALSettings(AssemblyLineConfig alc)` method. This method is invoked upon AssemblyLine initialization. As the initialization of the AssemblyLine progresses, the components which have been marked as Disabled do not get created/initialized.

If a LOOP component is disabled then all components contained in that LOOP will also be disabled.

Even if a component is disabled from the Config Editor GUI, it can be enabled using this feature:

```
com.ibm.di.server.TaskCallBlock.setComponentEnabled(String name, boolean enabled)
```

Providing an Initial Work Entry (IWE)

This is an alternative way of passing parameters using a TCB and is supported for backward compatibility reasons.

When an AssemblyLine is started with the **system.startAL()** call by an EventHandler or from a script, the AssemblyLine can still be passed parameters by setting attribute or property values in the Initial Work Entry (which is accessed through the *work* variable). It is then your job to apply these values to set Connector parameters (for example, in the AssemblyLine **Prolog – Init** Hook using the *connectorName*.`setParam()` function.)

Note: You must clear the **work** Entry with the `task.setWork(null)` call, otherwise Iterators in the AssemblyLine pass through on the first cycle.

You can examine the result of the AssemblyLine (which is the **work** Entry when the AssemblyLine stops) by using the `getResult()` function. See also "runtime provided Connector" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Below is an example of passing in a Connector parameter value with an IWE:

```
var entry = system.newEntry();
entry.setAttribute ("userNameForLookup", "John Doe");

// Here we start the AssemblyLine
var al = main.startAL ( "EmailLookupAL", entry );

// wait for al to finish
al.join();

var result = al.getResult();

// assume al sets the mail attribute in its working entry
task.logmsg ("Returned email = " + result.getString("mail"));
```

Sandbox

IBM Tivoli Directory Integrator includes a Sandbox feature which enables you to record the operation of one or more Connectors in an AssemblyLine for later replay without the necessary data sources being available. This feature uses the System Store facility. See "System Store" on page 84 for more information about System Store.

Before you can record or replay an AssemblyLine, you must first tell IBM Tivoli Directory Integrator where to store the AssemblyLine recording data. This is done in the **Sandbox** tab of the AssemblyLine Details window. At the top of this window is a field labeled **Database** where you can enter the directory path for the system to use.

The Sandbox facility is not supported in AssemblyLines using Checkpoint/Restart, containing a Connector in Server mode, or an Iterator Connector with Delta enabled. The server will abort the running of the AssemblyLine when or if this is discovered.

Recording AssemblyLine input

Once you have configured the User Store Database path to use, you must then select the Connectors to be recorded. This is also done in the AssemblyLine **Sandbox** tab. This tab presents you with the list of AssemblyLine Connectors, each line offering you **Record Enabled** and **Playback Enabled** selections. In order to record the operation of a Connector, simply select **Record Enabled** for that Connector.

In order to start AssemblyLine Recording, select **Record** from the list next to the **Run** button at the top of the AssemblyLine Details window. When you run the AssemblyLine, all data access operations are recorded for selected Connectors.

In order to run an AssemblyLine in Record mode from the command line, start the server with the **-qr** switch.

When in **Record Mode**, the AssemblyLine saves all content that it receives from the Connector Interfaces (both attributes and properties), as well as the TCB if this is passed to the AssemblyLine. Only input Connectors (Connectors in Iterator, Lookup or Call/Reply mode) actually have their data recorded. Connectors doing output (Update, Add, Delta, or Delete mode) have their data ignored during recording, although error messages thrown by the Connector Interface are logged and replayed later during playback.

Sandbox playback of AssemblyLine recordings

When an AssemblyLine is in Sandbox mode, all the Connectors set for playback are said to be in *virtual mode*. This means that their Connector Interface operations

(for example, `getNext()`, `findEntry()`, and so forth) are not actually called. Instead, these operations are simulated during playback.

In order to run an AssemblyLine in **Playback Mode**, you must select the Connectors to be run in virtual mode by the corresponding **Playback Enabled** check box in the AssemblyLine **Sandbox** tab.

Note: Not all recorded Connectors need to be enabled for playback (you can enable them to access live data sources), although this might affect the results of the playback operation.

To run an AssemblyLine from the command line, start the server with the **-q2** switch. A sandboxed AssemblyLine runs with input (including its Initial Work Entry) coming from a recorded set of data. For example, if you have a Java Messaging Service (JMS) Connector in your sandboxed AssemblyLine, the JMS Connector retrieves input from the previously recorded data and is never actually initialized.

When recording an AssemblyLine, the server creates a CloudScape database in the specified **Database** directory using the AssemblyLine name as the database name. This database contains tables for each Connector in the AssemblyLine. A sandboxed AssemblyLine can have one or more of its virtual Connectors replaced by renaming the recorded Connector and then adding a new one with its original name.

Connectors

Connectors are used to access and update information sources. A Connector's job is to level the playing field so that you do not have to deal with the technical details of working with various data stores, systems, services or transports. As such, each type of Connector is designed to use a specific protocol or API, handling the details of data source access so that you can concentrate on the data manipulations and relationships, as well as custom processing like filtering and consistency control.

There are basically two categories of Connectors:

- The first category is where both the transport and the structure of data content is known to the Connector (that is, the schema of the data source can be queried or detected using a well known API such as JDBC or LDAP).
- The second category is where the transport mechanism is known, but not the content structuring. This category requires a Parser (see “Parsers” on page 47 or "Parsers" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*) to interpret or generate the content structure in order for the AssemblyLine to function properly.

Notes:

1.

When you select a Connector for your AssemblyLine, a dialogue box is displayed enabling you to choose the type of Connector you want to *inherit* from. Inheritance is an important concept when working with IBM Tivoli Directory Integrator because all the components you include in solutions inherit some or all of their characteristics from another component—either from one of the basic types, or from your library of pre-configured components (the Connectors, Parsers, EventHandlers and Function Component folders in the Config Browser).

Connector prefixes (as seen in the Config Editor) and their meanings are:

system:/Connectors/ibmdi
: These Connectors are the standard Connectors included with IBM Tivoli Directory Integrator 6.1.1.

/Connectors/
: These are Connectors that you have pre-configured in the "Connectors" library folder of the current Config.

myInclude:/Connectors/
: These are Connectors that you can include from another configuration file (see “Include/Namespaces” on page 109).

@myConnector
: This is a very special case where, instead of inheriting a Connector, you are *reusing the connection* of another Connector in the same AssemblyLine. The name **myConnector** used as an example is replaced by the name of the Connector you wish to reuse.

The list of all Connectors included with the IBM Tivoli Directory Integrator can be found in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*. But you can also write your own Connector in JavaScript (see "Script Connector," *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*) or even Java (see "Implementing your own Components", *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.)

The mode setting of a Connector determines what role this component plays in the AL. Each Connector supports only a subset of modes that are suited for its connected system. For example, the File System Connector supports only a single output mode, AddOnly, and not Update, Delete or CallReply. When you use a Connector, you must first consult the documentation for this component for a list of supported modes (see "Connector Interfaces" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*).

Connectors in Iterator or Server mode can be used in the **Feeds** section of the AssemblyLine (the component list in the AL **Data Flow** tab). Connectors in other modes show up in the **Flow** section (although Iterators can also participate as Flow section components).

Connector Schema

Connectors can read and write data. The Connector Schema describes what the Connector expects to find when reading and writing the data. If an input field is required, the AssemblyLine fails if it does not show up in the Connector Interface.

The schema can be seen in the **Input Map** or **Output Map** tab (depending on the Connector’s Mode) of the Connector and certain behavior can be customized there, although this is dependent on the data source itself. From either Attribute Map tab (Input or Output) you can discover the schema of the connected data source by pressing **Quick discovery** (at the top of the Attribute Map itself) or by clicking the **Connect** and **Read Next** buttons found above the Connector Schema. For those systems that support it, there is a "Query schema" button as well. Even though you cannot discover Attributes, you can define the list of expected Attributes yourself. See “Setting up the Attribute Map” on page 164.

How Connectors share data (the work Entry)

Data is passed between Connectors using a storage object called the **work** Entry. This storage object can be accessed from your scripts by using the *work* variable,

and you can remove attributes, add new ones and change attribute values. For an AssemblyLine to operate, the **work** Entry must be populated from some source. This is often done with a Connector set to Iterator mode. In this case, the Iterator drives data to the AssemblyLine. You can also pass an Initial Work Entry (IWE) to an AssemblyLine when it is started, and you can even create your own **work** Entry in a Prolog script, for example:

```
init_work = system.newEntry(); // Create a new Entry object
init_work.setAttribute("uid", "cchateauvieux"); // populate it
task.setWork(init_work); // make it known as work to the Connectors
```

Note: An IWE in the Prolog can be regarded as fed from an invisible one-pass Iterator. See "Multiple Iterators in an AssemblyLine" on page 27 for the side effect this has on the behavior of any Iterators in the AssemblyLine.

The **work** Entry can be populated by a Connector in Lookup mode as well, although Link Criteria must be set up so that it is not dependent on attributes in the **work** Entry if none is available.

Another source of **work** Entries is a Connector in Server mode.

Connector modes

The mode of an AssemblyLine Connector defines what role that Connector plays in the data flow, and controls how the automated behavior of the AssemblyLine drives the Component for you. Connectors can be set to one of these standard modes:

- Iterator
- Lookup
- AddOnly
- Update
- Delete
- CallReply
- Server
- Delta

These modes are discussed in the sections below. For a detailed description of Connector mode behavior, as well as that of the AssemblyLine in general, see "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Iterator mode

Connectors in Iterator mode are called Iterator Connector and are used to scan a data source and extract its data. The Connector in Iterator mode actually iterates through the data source entries, reads their attribute values, and delivers each Entry to the AssemblyLine Flow section components. A Connector in Iterator mode is referred to as an Iterator.

AssemblyLines (except those called with an IWE; see "AssemblyLine parameter passing" on page 20) typically contain at least one Connector in Iterator mode. Iterators (Connectors in Iterator mode) supply the AssemblyLine with data by building Work Entries and passing these to the AL Flow section.

Flow section components are powered in order, starting at the top of the Flow list. When Flow processing completes, control is passed back to the Iterator in order to retrieve the next Entry.

Multiple Iterators in an AssemblyLine: If you have more than one Connector in Iterator mode, these Connectors are stacked in the order in which they appear in the Config (and the Connector List in the Config Editor, in the **Feeds** section) and are processed one at a time. So, if you are using two Iterators, the first one reads from its data source, passing the resulting Work Entry to the first non-Iterator, until it reaches the end of its data set. When the first Iterator has exhausted its input source, the second Iterator starts reading in data.

An initial **work** Entry is treated as coming from an invisible Iterator processed before any other Iterators. This means an IWE is passed to the first non-Iterator in the AssemblyLine, skipping all Iterators during the first cycle. This behavior is visible on the AssemblyLine Flow page in "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Assume you have an AssemblyLine with two Iterators, **a** preceding **b**. The first Iterator, **a**, is used until **a** returns no more entries. Then the AssemblyLine switches to **b** (ignoring **a**). If an Initial Work Entry (IWE) is passed to this AssemblyLine, then both Iterators are ignored for the first cycle, after which the AssemblyLine starts calling **a**.

Sometimes the IWE is used to pass configuration parameters into an AssemblyLine, but not data. However, the presence of an IWE causes Iterators in the AssemblyLine to be skipped during the first cycle. If you do not want this to happen, you must empty out the **work** Entry object by calling the `task.setWork(null)` function in a Prolog script. This causes the first Iterator to operate normally.

Using the Iterator mode: The most common pattern for using a Connector in Iterator mode is:

1. Add a Connector in Iterator mode to your AssemblyLine:
   a. Right-click the **Connectors** folder in the Config Browser, or select **Object->New Connector ...**.
   b. Name your new Connector.
   c. Select your Connector type.
   d. Set the mode of your new Connector to Iterator.
2. Click **Config ...** for your Connector in the IBM Tivoli Directory Integrator Config Editor and set the connection parameters for this Connector in the **Connection** sub-tab. Some Connectors require you to configure a Parser as well in the **Parser** sub-tab.
3. Click **Input Map** on the Connector configuration window to discover or define the schema for this data source:
   a. Click **Connect to the data source** to start the Connector.
   b. Click **Read next entry** to get the next entry from the data source and examine it to discover attributes.
   c. Click **Discover the schema of the datasource** if this is supported by the underlying data source.
   d. You can add additional attributes, or remove existing ones, using **Add** and **Remove**.
4. Finally, select Attributes from the **Connector Attributes** list and then drag them into your Input Map, or add these by hand with the **Add** and **Remove** buttons. The Input Map controls which Attributes are brought into your AL for processing, as well as any transformations you specify.

These mapped Attributes are retrieved from the data source and are passed to the Connectors in the **Flow** section in your AssemblyLine.

Lookup mode

Lookup mode enables you to join data from different data sources using the relationship between attributes in these systems. A Connector in Lookup mode is often referred to as a Lookup Connector.

In order to set up a Lookup Connector you must tell the Connector how you define a match between data already in the AssemblyLine and that found in the connected system. This is called the Connector's Link Criteria, and each Lookup Connector has an associated **Link Criteria** tab where you define the rules for finding matching entries.

Using the Lookup mode: The most common pattern for using a Connector in Lookup mode is:

1. Add a Connector in Lookup mode to your AssemblyLine:
   a. Right-click the **Connectors** folder in the Config Browser, or select **Object->New Connector ...**.
   b. Name your new Connector.
   c. Select your Connector type.
   d. Set the mode of your new Connector to Lookup.
2. Click **Config ...** for your Connector in the IBM Tivoli Directory Integrator Config Editor and set the connection parameters for this Connector in the **Connection** sub-tab. Some Connectors require you to configure a Parser as well in the **Parser** sub-tab.
3. Click **Input Map** on the Connector configuration window to discover or define the schema for this data source:
   a. Click **Connect to the data source** to start the Connector.
   b. Click **Read next entry** to get the next entry from the data source and examine it to discover attributes.
   c. Use the **Discover the schema of the datasource** button, if this is supported by the underlying data source.
   d. You can add additional attributes, or remove existing ones, using **Add** and **Remove**.
4. Select Attributes from the **Connector Attributes** list and then drag them into your Input Map, or add these by hand with the **Add** and **Remove** buttons. The Input Map controls which Attributes are brought into your AL for processing, as well as any transformations you specify.
5. Click **Link Criteria** on the Connector configuration window and set up the rules for attribute matching. Here you have a couple of choices:
   a. Click **Add new Link Criteria** and select an attribute from the connected system, the matching operator (for example, Equals, Begins With, and so forth) and then the **work** Entry attribute to be matched. When the Connector performs the Lookup, it creates the underlying API or protocol syntax based on the Link Criteria you have specified, keeping your solution independent of the type of system used. You can add multiple Link Criteria, which are connected by the Boolean operator AND, together to build the search call.
   b. You can also select **Build criteria with custom script** , which opens a script editor window where you can create your own search string, passing this back to the Connector using the **ret.filter** object. For example:

```
ret.filter = "uid=" + work.getString("uid");
```

Note that Expressions can also be used to dynamically specify the Attribute or Value to use for any Link Criteria. See "Expressions" on page 110 for information. Also see "Link Criteria" on page 50 for more details about Link Criteria.

The attributes that you read (and compute) in the Input Map are available to other downstream Connectors and script logic using the **work** Entry object.

AddOnly mode

Connectors in AddOnly mode (AddOnly Connectors) are used for adding new data entries to a data source. This Connector mode requires almost no configuration. Set the connection parameters and then select the attributes to write from the **work** Entry.

Using the AddOnly mode: The most common and simple pattern for using a Connector in AddOnly mode is:

1. Add a Connector in AddOnly mode to your AssemblyLine:
   a. Right-click the **Connectors** folder in the Config Browser, or select **Object->New Connector ...**.
   b. Name your new Connector.
   c. Select your Connector type.
   d. Set the mode of your new Connector to AddOnly.
2. Click **Config ...** for your Connector in the IBM Tivoli Directory Integrator Config Editor and set the connection parameters for this Connector in the **Connection** sub-tab. Some Connectors require you to configure a Parser as well in the **Parser** sub-tab.
3. Optionally, and if data is available to read from the connected system, you can click **Input Map** on the Connector configuration window to discover and define the schema for this data source:
   a. Click **Connect to the data source** to start the Connector.
   b. Click **Read next entry** to get the next entry from the data source and examine it to discover attributes.
   c. Use the **Discover the schema of the datasource** button, if this is supported by the underlying data source.
   d. Add additional attributes, or remove existing ones, using the **Add** and **Remove** buttons.
4. In the **Output Map** tab you can select attributes from the **Work Entry** window and then drag them into your Output Map. You can also add and remove attributes with the **Add** and **Remove** buttons.

Entries with the Attributes you have selected are added in the data source during AssemblyLine's execution.

Update mode

Connectors in Update mode (Update Connectors) are used for adding and modifying data in a data source. For each Entry passed from the AssemblyLine, the Update Connector tries to locate a matching Entry from the data source to modify with the Entry's attributes values received. If no match is found, the Update mode Connector will add a new Entry.

As with Lookup Connectors, you must tell the Connector how you define a match between data already in the AssemblyLine and that found in the connected system. This is called the Connector's Link Criteria, and each Update Connector™ has an

associated **Link Criteria** tab where you define the rules for finding matching entries. If no such Entry is found, a new Entry is added to the data source. However, if a matching Entry is found, it is modified. If more than one entry matches the Link Criteria, the **Multiple Entries Found** Hook is called. Furthermore, the Output Map can be configured to specify which attributes are to be used during an Add or Modify operation.

When doing a Modify operation, only those attributes that are marked as Modify (Mod) in the Output Map are changed in the data source. If the Entry passed from the AssemblyLine does not have a value for one attribute, the Null Behavior for that attribute becomes significant. If it is set to **Delete**, the attribute does not exist in the modifying entry, thus the attribute cannot be changed in the data source. If it is set to **NULL**, the attribute exists in the modifying entry, but with a null value, which means that the attribute is deleted in the data source.

An important feature that Update Connectors offer is the **Compute Changes** option. When turned on, the Connector first checks the new values against the old ones and updates only if and where needed. Thus you can skip unnecessary updates which can be valuable if the update operation is a heavy one for the particular data source you are updating.

Using the Update mode: The most common and simple pattern for using a Connector in Update mode is:

1. Add a Connector in Update mode to your AssemblyLine:
   a. Right-click the **Connectors** folder in the Config Browser, or select **Object->New Connector ...**.
   b. Name your new Connector.
   c. Select your Connector type.
   d. Set the mode of your new Connector to Update.
2. Click **Config ...** for your Connector in the IBM Tivoli Directory Integrator Config Editor and set the connection parameters for this Connector in the **Connection** sub-tab. Some Connectors require you to configure a Parser as well in the **Parser** sub-tab.
3. Click **Input Map** of the Connector configuration window to discover and define the schema for this data source:
   a. Click **Connect to the data source** to start the Connector.
   b. Click **Read next entry** to get the next entry from the data source and examine it to discover attributes.
   c. Use the **Discover the schema of the datasource** button, if this is supported by the underlying data source.
   d. You can add additional attributes, or remove existing ones, using the **Add** and **Remove** buttons.
   e. Add additional attributes, or remove existing ones, using the **Add** and **Remove** buttons.
4. In the **Output Map** tab you can select attributes from the **Work Entry** window and then drag them into your Output Map. You can also add and remove attributes with the **Add** and **Remove** buttons.
5. Click **Link Criteria** tab of the Connector configuration window and set up the rules for attribute matching. Here you have a couple of choices:
   a. Click **Add new Link Criteria** and select an attribute from the connected system, the matching operator (for example, Equals, Begins With, and so forth) and then the **work** Entry attribute to be matched. When the

Connector performs the Lookup (which is part of the Update behavior), it creates the underlying API or protocol syntax based on the Link Criteria you have specified, keeping your solution independent of the type of system used. You can add multiple Link Criteria, which are connected by the Boolean operator AND, together to build the search call.

b. You can select **Build criteria with custom script** , which opens a script editor window where you can create your own search string, passing this back to the Connector using the **ret.filter** object. For example:

```
ret.filter = "uid=" + work.getString("uid");
```

See "Link Criteria" on page 50 for more information about Link Criteria.

Attributes you have selected for the Entries are updated in the data source during AssemblyLine's execution.

Note: In Update mode, multiple entries can be updated. See "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Delete mode

Connectors in Delete mode (Delete Connectors) are used for deleting data from a data source. For each Entry passed to the Delete Connector, it tries to locate matching data in the connected system. If a single matching Entry is found, it is deleted, otherwise the **On No Match** Hook is called if none were found, or the **On Multiple Entries** Hook is more than a single match was found. As with Lookup and Update modes, Delete mode requires you to define rules for finding the matching Entry for deletion. This is configured in the Connector's Link Criteria tab.

Using the Delete mode: The most common and simple pattern for using a Connector in Delete mode is:

1. Add a Connector in Delete mode to your AssemblyLine:
   a. Right-click the **Connectors** folder in the Config Browser, or select **Object->New Connector ...**.
   b. Name your new Connector.
   c. Select your Connector type.
   d. Set the mode of your new Connector to Delete.
2. Click **Config ...** for your Connector in the IBM Tivoli Directory Integrator Config Editor and set the connection parameters for this Connector in the **Connection** sub-tab. Some Connectors require you to configure a Parser as well in the **Parser** sub-tab.

   Note: Server Mode Connectors based on TCP protocols have a new parameter called Connection Backlog that controls the queue length for incoming connections.

   Note: The Response section is removed from the AssemblyLine component list, leaving only Feeds and Flow. Instead, the Output Map and Response Hooks are part of the Server Mode Connector itself on screen. The execution of response behavior is still done after Flow section processing is finished.

   Note that system.skipEntry() will continue to skip response behavior, and as such, no reply will be made by a Server Mode Connector. If you would prefer to simply skip the remaining Flow section components and yet send the response, use the `system.exitBranch("Flow");` call instead.

3. Click **Input Map** on the Connector configuration window to discover and define the schema for this data source:
   a. Click **Connect to the data source** to start the Connector.
   b. Click **Read next entry** to get the next entry from the data source and examine it to discover attributes.
   c. Use the **Discover the schema of the datasource** button, if this is supported by the underlying data source.
   d. You can add additional attributes, or remove existing ones, using the **Add** and **Remove** buttons.
4. In the **Input Map** tab you can select attributes from the Connector Attribute list and then drag them into your Input Map. You can also add and remove attributes with the **Add** and **Remove** buttons.

   Note: The Input Map is used in Delete mode for reading the matching entry found in the data source into the **conn** Entry object, which can then be used in your scripts (for example, to determine if the entry actually is to be deleted).
5. Click **Link Criteria** on the Connector configuration window and set up the rules for attribute matching. Here you have a couple of choices:
   a. Click **Add new Link Criteria** and select an attribute from the connected system, the matching operator (for example, Equals, Begins With, and so forth) and then the **work** Entry attribute to be matched. When the Connector performs the Lookup, it creates the underlying API or protocol syntax based on the Link Criteria you have specified, keeping your solution independent of the type of system used. You can add multiple Link Criteria, which are connected by the Boolean operator AND, together to build the search call.
   b. You can select **Build criteria with custom script** , which opens a script editor window where you can create your own search string, passing this back to the Connector by way of the **ret.filter** object. For example:

      ```
      ret.filter = "uid=" + work.getString("uid");
      ```

   See "Link Criteria" on page 50 for more information about Link Criteria.

CallReply mode

CallReply mode is used to make requests to data source services (such as Web services) that require you to send input parameters and receive a reply with return values. Unlike the other modes, CallReply gives access to both Input and Output AttributeMaps.

Using the CallReply mode: The most common and simple pattern for using a Connector in CallReply mode is:

1. Add a Connector in CallReply mode to your AssemblyLine:
   a. Right-click the **Connectors** folder in the Config Browser, or select **Object->New Connector ...**.
   b. Name your new Connector.
   c. Select your Connector type.
   d. Set the mode of your new Connector to CallReply.
2. Click **Config ...** for your Connector in the IBM Tivoli Directory Integrator Config Editor and set the connection parameters for this Connector in the **Connection** sub-tab. Some Connectors require you to configure a Parser as well in the **Parser** sub-tab.

3. In the **Output Map** tab you can select attributes from the Connector Attribute list and then drag them into your Output Map. You can also add and remove attributes with the **Add** and **Remove** buttons. These are sent as input parameters to the service call.
4. In the **Input Map** tab you can select attributes from the **Connector Attribute** list and then drag them into your Input Map. You can also add and remove attributes with the **Add** and **Remove** buttons. These are expected as return parameters from the service call.

Server mode

The Server mode, available in a select number of Connectors is meant to provide functionality of waiting for an incoming event, dispatch a thread dealing with the event, and send a reply back to the originator.

Currently, all Server mode connectors are connection-based. That is, a (master) AssemblyLine (AL) which such a Connector heads waits for an incoming network channel, session, or connection of some kind (TCP, HTTP, LDAP, Web Services, SNMP); when a connection is initiated the Server Mode connector clones the AL it is part of, and resumes waiting for the next event (a new connection initiation). In the cloned *worker* AL, meanwhile, the Server Mode Connector places itself in *Iterator* Mode, and starts reading data from the connection. The data obtained from the connection is then fed to the rest of the AL in normal Iterator fashion, including following the standard Iterator Hook flow, reading the event entries one at a time and passing them to the other Flow components for processing until there is no more data to read. At the end of each cycle (often there will only be one) the AL headed by the Server Mode connector sends a reply back to the client—unless you decide to skip the reply phase with, for example, `system.skipEntry()`.

Once the AL it feeds is complete (that is, the data source is exhausted) that thread terminates; that is, at this time, the *worker* AL is cleared away, and if necessary, the Pool Manager is informed that this AL instance is available again.

The original ("master") Server Mode connector is still actively listening for more connection initiations.

Note: Under certain rare conditions (when you issue more than 5 client requests to the server in parallel, primarily on the zOS operating system), SNMP clients exit, giving bad Protocol Data Unit (PDU) exceptions. Importantly, in a real-life situation an agent (SNMP Server Connector) is rarely queried intensely by multiple managers (SNMP Connector). The SNMP Connector has an undocumented configuration parameter named "snmpWalkTimeout". You can override the default for this parameter, which is 5000 ms. The parameter is not accessible using the TDI GUI. You can set the override value for this parameter using Javascript. Type the value you want for the snmpWalkTimeout parameter in the following format:

```
thisConnector.connector.setParam("snmpWalkTimeout", "100000");
```

Server Mode and the ALPool: The process of creating a clone AL can be optimized by using the ALPool. On event detection, the Server mode Connector either proceeds with the Flow section of this AL; or if an ALPool is configured for this AL, then it contacts the Pool Manager process to request an available AL instance to handle this event.

When an AssemblyLine with a Server mode connector uses the ALPool, the ALPool will execute AL instances from beginning to end. Before the AL instance in

the ALPool closes the Flow connectors, the ALPool retrieves those connectors into a pooled connector set that will be reused in the next AL instance created by the ALPool (ALPool uses tcb.setRuntimeConnector method).

There are two system properties that govern the behavior of connector pooling:

`com.ibm.di.server.connectorpooltimeout`
: This property defines the timeout in seconds before a pooled connector set is released.

Table 1. Value table

< 0	Disable Connector pooling
0	Timeout disabled; pool connectors never timeout
> 0	Number of seconds before pooled connectors timeout

`com.ibm.di.server.connectorpoolexclude`
: This property defines the Connector types that are excluded from pooling. If a Connector's class name appears in this comma separated list it is not included in the Connector pool set.

When a new AssemblyLine (AL) instance is created by the ALPool, it will look for an available pooled connector set, which, if present, is provided to the new AL Instance as runtime-provided connectors. This ensures proper flow of the AL in general in terms of Hook execution, and so forth

Connectors are never shared. They are only assigned to a single AL instance when used.

Using the Server mode: The most common pattern for using a Connector in Server mode is:

1. Add a Connector in Server mode to your AssemblyLine:
   a. Right-click the **Connectors** folder in the Config Browser, or select **Object->New Connector ...**.
   b. Name your new Connector.
   c. Select your Connector type.
   d. Set the mode of your new Connector to Server.
2. Click **Config ...** for your Connector in the IBM Tivoli Directory Integrator Config Editor and set the connection parameters for this Connector in the **Connection** sub-tab. Some Connectors require you to configure a Parser as well in the **Parser** sub-tab.
3. Click **Input Map** of the Connector configuration window to discover and define the schema for this data source:
   a. Click **Connect to the data source** to start the Connector.
   b. Click **Read next entry** to get the next entry from the data source and examine it to discover attributes.
   c. Use the **Discover the schema of the datasource** button, if this is supported by the underlying data source.
   d. You can add additional attributes, or remove existing ones, using the **Add** and **Remove** buttons.

   Note: Due to the nature of Connectors in Server mode, using the **Connect to the data source** button will in most cases not work; therefore, you will in most cases need to use an alternative method (using the **Discover the**

schema of the datasource button, or even setting up Attribute Maps completely by hand with the **Add** and **Remove** buttons).

These mapped Attributes are retrieved from the data source and are passed to the Connectors in the **Flow** section in your AssemblyLine.

Server mode Connectors are special in that they usually need to return some information to the client that connects to it.

In the **Output Map** tab, you can select attributes from the **Work Entry** window and then drag them into your Connector Attributes list, after which you assign those Attributes to Attributes in the Output Schema. You can also add and remove attributes with the **Add** and **Remove** buttons. The data mapped out in this manner is sent back to the client, for each cycle of the AL (though as mentioned before, in a typical request or response way of operation there will often be only one cycle.)

Delta mode

The Delta mode is designed to simplify the application of changes to data by providing incremental modifications to the connected system, based on delta operation codes set by either the Iterator Delta Engine feature (**Delta** tab for Iterators), or Change Detection Connectors like the Intrusion Detection System (IDS), LDAP, Active Directory (AD) or Exchange Changelog Connectors, or the ones for RDBMS and Lotus/Domino Changes; or by parsing this delta information with the Lightweight Directory Interchange Format (LDIF) or Directory Services Markup Language (DSML) Parsers.

In previous TDI versions (pre-6.1), snapshots written to the Delta Store (a feature of the System Store) during Delta Engine processing were committed immediately. As a result, the Delta Engine would consider a changed entry as handled even though processing the AL Flow section failed. This limitation is addressed through the ″Commit″ parameter on the Connector Delta tab. The setting of this parameter controls when the Delta Engine commits snapshots taken of incoming data to the System Store.

Delta mode is only available for LDAP and JDBC Connectors.

Note: A Connector in Delta mode needs to be paired with another Connector that provides Delta information, otherwise the Delta mode has no delta operation codes to work with.

The Delta features in TDI are designed to facilitate synchronization solutions. You can look at the Delta capabilities of the system as divided into two sections: *Detection* and *Application*.

Delta Detection: IBM Tivoli Directory Integrator provides a number of change (delta) detection mechanisms and tools:

Delta Engine
This is a feature available to Connectors in Iterator mode. If enabled from the Iterator's Delta tab, the Delta Engine feature uses the System Store to take a snapshot of data being iterated. Then on successive runs, each Entry iterated is compared with the snapshot database to see what has changed.

Change Detection Connector
These components leverage information in the connected system to detect changes, and are either used in Iterator or Server mode, depending on the Connector. For example, Iterator mode is used for many of the Change

Detection Connectors, like those for LDAP, Exchange and ActiveDirectory Changelog, as well as the RDBMS and Notes/Domino Change Connectors.

The Change Detection Connectors have been reworked to make them behave in a common way, as well as to provide the same parameter labels for common settings. The Connectors are:

- IBM Directory Server (TDS) Changelog
- AD Changelog v2 (Active Directory)
- Domino® Change Detection
- Netscape Changelog (openLDAP, SunOne, iPlanet, and so forth)
- RDBMS Changelog (DB2®, Oracle, SQL Server, and so forth)
- zOS Changelog

See the "Connectors" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information about these Connectors.

The Delta Store feature, based on the System Store, reports specific changes all the way down to the individual values of attributes. This fine degree of change detection is also available when parsing LDIF files. Others components are limited to simply reporting if an entire Entry is added, modified or deleted.

This delta information is stored in the Work Entry object, and depending on the Change Detection component/feature used can be stored as an *Entry-Level operation code*, at the *Attribute-Level* or even at the *Attribute Value-Level*.

As an example, set up a FileSystem Connector with the Delta Store feature enabled. Have it iterate over a simple XML document that you can easily modify in a text editor. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-3333</ValueTag>
        </Telephone>
        <Birthdate>1958-12-24</Birthdate>
        <Title>Full-Time TDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
    </Entry>
</DocRoot>
```

Be sure to use the special map-all Attribute Map character, the asterisk (*). This is the only Attribute you need in your map to ensure that all Attributes returned are mapped in to the Work Entry object.

Now add a Script Component with the following code:

```
// Get the names of all Attributes in work as a String array
var attName = work.getAttributeNames();
// Print the Entry-level delta op code
task.logmsg(" Entry (" +
    work.getString( "FullName" ) + ") : " +
    work.getOperation() );
// Loop through all the Attributes in work
for (i = 0; i < attName.length; i++) {
  // Grab an Attribute and print the Attribute-level op code
 att = work.getAttribute( attName[ i ] );
 task.logmsg("    Att ( " + attName[i] + ") : " + att.getOperation() );
  // Now loop through all the Attribute's values and print their op codes
 for (j = 0; j < att.size(); j++) {
  task.logmsg( "      Val (" +
```

```
                    att.getValue( j ) + ") : " +
                    att.getValueOperation( j ) );
 }
}
```

The first time you run this AL, your Script Component code will create this log output:

```
12:46:31   Entry (John Doe) : add
12:46:31      Att ( Telephone) : replace
12:46:31        Val (111-1111) :
12:46:31        Val (222-2222) :
12:46:31        Val (333-3333) :
12:46:31      Att ( Birthdate) : replace
12:46:31        Val (1958-12-24) :
12:46:31      Att ( Title) : replace
12:46:31        Val (Full-Time TDI Specialist) :
12:46:31      Att ( uid) : replace
12:46:31        Val (jdoe) :
12:46:31      Att ( FullName) : replace
12:46:31        Val (John Doe) :
```

Since this Entry was not found in the previously empty Delta Store, it is tagged at the Entry-level as *new*. Furthermore, each of its Attributes has a *replace* code, meaning that all values have changed (which makes sense because the Delta is telling us that this is new data).

Make the following changes to your XML file:

1. Change the last *Telephone* number value to 333-3334.
2. Delete *Birthdate*.
3. Add a new *Address* Attribute.

Your resulting Config should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-3334</ValueTag>
        </Telephone>
        <Title>Full-Time TDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
        <Address>123 Willowby Lane</Address>
    </Entry>
</DocRoot>
```

Run your AL again. This time your log output should look like this:

```
13:53:22   Entry (John Doe) : modify
13:53:22      Att ( Telephone) : modify
13:53:22        Val (111-1111) : unchanged
13:53:22        Val (222-2222) : unchanged
13:53:22        Val (333-3334) : add
13:53:22        Val (333-3333) : delete
13:53:22      Att ( Birthdate) : delete
13:53:22        Val (1958-12-24) : delete
13:53:22      Att ( uid) : unchanged
13:53:22        Val (jdoe) : unchanged
13:53:22      Att ( Title) : unchanged
13:53:22        Val (Full-Time TDI Specialist) : unchanged
13:53:22      Att ( Address) : add
13:53:22        Val (123 Willowby Lane) : add
13:53:22      Att ( FullName) : unchanged
13:53:22        Val (John Doe) : unchanged
```

Now the Entry is tagged as *modify* and the Attributes reflect what the modifications for each of them. As you can see, the *Birthdate* Attribute is marked as *delete* and *Address* as *add*. That's the reason you used the special map-all character

for our Input Map. If you had mapped only the Attributes that existed in the first version of this XML document, we would not have retrieved Address when it appeared in the input.

Note especially the last two value entries under the *Telephone* Attribute, marked as *modify*. The change to one of this Attribute's value resulted in two Delta items: a value *delete* and then an *add*.

When you wanted to build a data synchronization AssemblyLine in previous versions of TDI, you needed to script in order to handle flow control. Although you could be receiving *adds*, *modifies* and *deletes* from your change component or feature, a Connector could only be set to one of the two required output modes: Update or Delete. So either you had two Connectors pointing to the same target system and you put script in the Before Execute Hook of each to ignore the Entry if its operation code did not match the mode of this component; or you could have a single Connector (either Update or Delete mode) in *Passive* state, and then control its execution from script code where you checked the operation code. This still meant that even though you knew what had changed in the case of a modified Entry, our Update Mode Connector would still read in the original data before writing the changes back to the data source. This can lead to unwanted network or datasource traffic when you are only changing a single value in a multi-valued group related Attribute containing thousands of values.

Enter the Connector *Delta* mode.

Delta Application (Connector Delta Mode): The Delta mode is designed to simplify the application of delta information (read: make the actual changes) in a number of ways.

Firstly, Delta mode handles all types of deltas: adds, modifies and deletes. This reduces the number of data synchronization ALs to two Connectors: One Delta Detection Connector in the *Feeds* section to pick up the changes, and a second one in Delta mode to apply these changes to a target system.

Furthermore, Delta mode will apply the delta information at the lowest level supported by the target system itself. This is done by first checking the Connector Interface to see what level of incremental modification is supported by the data source[2]. If you are working with an LDAP directory, then Delta mode will perform Attribute value adds and deletes. In the context of a traditional RDBMS (JDBC), then doing a delete and then an add of a column value does not make sense, so this is handled as a value replacement for that Attribute.

This is dealt with automatically by the Delta mode for those data sources that support this functionality[3]. If the data source offers optimized calls to handle incremental modifications, and these are supported by the Connector Interface, then Delta mode will use these. On the other hand, if the connected system does not offer "intelligent" delta update mechanisms, Delta mode will simulate these as much as possible, performing pre-update lookups (like Update mode), change computations and subsequent application of the detected changes.

2. Note that the only Connectors that support incremental modifications are the LDAP and JDBC Connectors, since LDAP directories provide this functionality.

3. Of course, you can control these built-in behaviors through configuration parameters and Hook code.

Component states

The state of a component determines its level of participation in the operation of the AssemblyLine. In general terms, an AssemblyLine performs two levels of component handling:

- Powering up the component at the start of AssemblyLine operation (Start up Phase) and closing it down when the AssemblyLine completes (Shutdown).
- Driving the component during AssemblyLine operation.

You can change the state of a component by right-clicking on it in the AssemblyLine component list (AL **Data Flow** tab) and choosing the Enabled option, or by changing the **State** in the **Details** window for that component. You can change the state of AL components using the TCB (See "Disabling AssemblyLine components" on page 22).

Enabled state

Enabled is the normal component state. In Enabled state, a component is initialized during AL Start up, used during cycling and then closed when the AL shuts down. All component types (Connectors, Functions, Scripts, AttributeMaps and Branches) can be set to Enabled state .

Passive state

Only Connectors can be set to Passive state . Passive Connectors (Connectors in Passive state) are started and closed just like Enabled Connectors. However, they are not driven by the AssemblyLine automated behavior. However, Connectors in Passive state can be invoked by script code from any of the control points for scripting provided by IBM Tivoli Directory Integrator. For example, if you have a Passive Connector in your AssemblyLine called **myErrorConnector** then you could invoke its `add()` operation with the following script code:

```
var err = system.newEntry(); // Create new Entry object
err.merge(work); // Merge in attributes in the work Entry
// This next line sets an attribute called Error
err.setAttribute ( "Error", "Operation failed" );
myErrorConnector.add( err ) // Add new err Entry;
```

Disabled state

All component types can be set to Disabled state . In Disabled state, the component is neither initialized (and closed) nor operated during normal AssemblyLine activation. If you want to use it in your scripts, then you must initialize it yourself.

The name of a disabled Connector is registered but pointing at null, so you can write the following conditional code:

```
if (myConnector != null)
 myConnector.connector.aMethod();
```

to handle the situation where you plan on setting **myConnector** to Disabled state.

This state is often used during troubleshooting in order to simplify the solution while debugging, helping to localize any problems.

Adapters

In addition to the standard modes, you can define custom Connector modes by writing a TDI *Adapter*. An Adapter is an AssemblyLine that is "wrapped" by an AL Connector so that it exposes the operations defined for the AL as Connector mode settings. Adapters are easy to distribute to other TDI developers, and just as simple to use as traditional Connectors.

Adapters enable developers to leverage the entire TDI framework when creating a custom Connector with potentially complex business logic and custom operations to be offered to the TDI development community.

A number of new features are described in the following sections that in combination make Adapters possible.

The following case describes the high level flow of activities involved in implementing and using a TDI Adapter:

1. Anne develops the Adapter AssemblyLine (for example, to access a custom developed ERP system) that implements the Connector modes to be supported (such as Iterator and Delete), as well as custom modes as required.
2. Anne publishes the AL into a package that can be distributed to Pete, another TDI developer, as a stand-alone file.
3. Pete copies the package into his TDI development environment—either into the "adapters" folder so that it shows up as a Connector, or into his Resources library area so that he can drag the AssemblyLine into his solutions. Resources are ideal for maintaining a personal library of components, script and AssemblyLines for re-use across multiple TDI Configs.
4. Pete uses Anne's Adapter in his AL just like any other TDI Connector by using an AL Connector to call the Adapter.
5. Anne can improve her Adapter and publish new versions by repeating the steps above.

Features that enable implementation of a TDI Adapter

This section discusses new features that are used to create and use Adapters. Most of these features are not developed specifically for the Adapter concept, so they have many cases for non-Adapter use as well.

AL operations: Any number of operations can be defined in an AssemblyLine (AL). They are similar to the call-return schema of the AL, but any number of operations can be created. When ALs are called or executed through the API, from script, from the AL Function Component, or from the AL Connector, an operation can be specified along with the required attributes for that operation. At run time, the AL will know what operation is invoked, and the flow inside the AL can be adjusted accordingly.

These operations provide the entry points for the AL Connector to view and treat the Adapter as a Connector. The entry points are the same as when developing a Connector in Java or JavaScript, and are described in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*. For example, if the Adapter developer wants only to implement the Lookup mode, then it's only necessary to implement the `findEntry` operation.

Switch/case component: The Switch component is a new AL component that is similar to switch constructs in traditional development languages. The Switch component is a variant of the If-ElseIf-ElseIf component. Within the Switch component, a number of Cases are defined that contain the AL components to be executed when the Switch statement matches the value of the case. There is a new Switch component providing functionality for implementing AL Operations.

One benefit of the Switch component is that it can automatically populate the case statements based on the AL operations that have been defined. This way you can easily ensure that code is implemented for all of the operations that have been defined.

A Switch can be based on one of the following mechanisms:

- The value of a Work Entry Attribute.
- The AL Operation specified when this AssemblyLine was invoked.
- The delta operation code set in the Work Entry (for example, add, modify, delete, and so forth).
- The default case of the Switch component
- A user-defined Expression

Each method returns some value that is then used to match against Case components set up under the Switch. A Case is always based on an Expression. Also, note that only Cases that are matched are executed, so you do not have to exit the Switch manually (as is the case for Switch-case in many 3GL programming languages).

The TDI Configuration Editor provides a default case for the Switch component, precluding the need for unnecessary scripting. Make sure you place the default switch case statement last, because the Switch component executes the default switch case only if none of the preceding switch cases is executed. When enabled, the default case statement works with the following switch types:

- Work attribute
- Operation
- Delta entry
- User-defined switch

The default case is not applicable to AssemblyLine operations.

Flexible connector initialization: Formerly, all Connectors in an AssemblyLine were initialized during the AL initialization phase. Dynamic configuration of a Connector usually required termination of a connection, modification of the connection parameters, and then a re-establishment of the connection—all through script. The alternative was to establish and use the Connector solely through scripting.

In TDI 6.1, Connectors can optionally initialize:

On demand
: The Connection is not established at AL initialization, but as control passes for the first time to this Connector during the AssemblyLine execution phase (Flow). This means that a complex AL will only initialize the Connectors that are actually used.

Every time
: The Connector initializes every time as control is passed to it. This is useful when the connection parameters (such as a file name or LDAP credentials) are part of the information passed into each call to the Adapter. A separate benefit is that this capability is also helpful when using pooled Connectors, as “every time” will result in acquiring a Connector instance from the pool at run time, and then released after use. In essence, this implements a shared or reuse Connector pool across ALs.

When Config has changed
: The Connection is re-initialized if the configuration parameters have changed since the previous initialization of the Connector. With the new parameter substitution feature, Connectors can be dynamically configured much more easily than before. This facilitates changing the connection parameters of a Connector —and forcing re-connection – from both inside

the AL as well as outside. For example, another AL, or a command-line modification of properties can result in an AL automatically re-connecting to its targets with much less effort than before.

Using an Iterator in Flow: Iterators have previously only been used in the Feed section to drive the entire AL cycle, or within the Flow to power a Loop component. An Iterator can be placed in the Flow itself to facilitate implementing Iterator Mode in an Adapter. To understand this, a short review of how the Iterator works is in order. First `selectEntries()` of a Connector (or for Adapters, the selectEntries operation as described below in " The use of operations in a TDI Adapter"), is called to create the result set, then `getNextEntry()` is called to read from the result set until it's empty. By limiting iterators to the Feed section, it would be very impractical to implement a `getNextEntry` operation that returned the next record from an Iterator Connector in your Adapter. With an Iterator in the Flow, your `getNextEntry` operation could use an Iterator Connector.

Packaging an Adapter for consumption: The following section describes packaging an Adapter for consumption using the following case: Helping Anne package her Adapter component and publish it for others to consume.

The concept of Adapters is to allow you to publish an AL as though it were a Connector. AL Publishing makes this powerful feature easy to use.

When the Adapter is developed, the **Publish** command in the TDI development environment creates a Package of Anne's AL. Publishing an AL means resolving all inheritance and dependencies between the Adapter AL and the rest of Anne's development environment.

The Package consists of a standard, stand-alone Config XML file that only contains the Adapter code that can be sent to other TDI developers for inclusion in their resource library. The package can be saved anywhere, but the default location is the packages directory in Anne's TDI solution directory. When the Adapter is published, it shows up in the Adapter section of the resources library, in the Connector list as an available Connector, and can be queried from the AL Connector.

The difference between a Package and an Adapter is that Adapters are Packages that are intended to be used as Connectors. Other Packages simply contain AssemblyLines that can be called with the AL FC or other mechanism to run an AL.

Improved GUI to facilitate use of re-usable components: This section describes the following case: Extending the development environment so that Pete can receive an Adapter package and store it in his local environment.

The **Resources** tab is part of the TDI development environment. It can contain any TDI components, such as Connectors, Parsers, and even ALs. It's a place to keep re-usable Connectors, Attribute maps, Loops, script components. Previously, shared components across Configs had to be implemented with include commands. They are visible in the Config Editor and can easily be used across Configs.

Adapters have their own tab in Resources. Adapters need to be located either in the jars directory tree (for maintainability reasons it's suggested to create an `/adapter` subdirectory under the `jars` subdirectory) or in the `adapter` subdirectory in your TDI `solution` subdirectory.

In addition to being a repository for Adapters, **Resources** addresses a long wished for capability to have an easy-to-use mechanism to save components and code that can be used across TDI projects.

Using an Adapter in your AssemblyLine: This section describes the following case: Providing the interfacing mechanism so that Pete can use Anne's Adapter in his own AssemblyLines as any other Connector.

There are a number of mechanisms available when calling an AL from another AL. However, when an AL is developed as an Adapter, then the primary mechanism to use is the AssemblyLine Connector. The AssemblyLine Connector existed in TDI 6.0, but it is greatly improved to deal with Adapter-style ALs. In 6.0, it could only be used to iterate on the output of another AL. In TDI 6.1, when the AL Connector is used in an AssemblyLine, it is configured by specifying what Adapter it should call. The target Adapter is then inspected for operations, and that determines which Connector modes are made available.

As a convenience feature, TDI automatically wraps all Adapters so that they look like Connectors in the Connector list. Pete can therefore choose to insert an Adapter directly from his Connector list, or can insert an AL Connector and then specify the desired Adapter to call.

The configuration of the Adapter is done in the usual Connector Config window. All parameters displayed here are defined in the schema of the reserved operation `$initialization` of the Adapter. This provides a mechanism for sending configuration parameters to the Adapter for dynamic configuration of its Connectors.

The flexible initialization of Connectors feature is helpful in that it can be used both in the AL that calls the Adapter, as well as in the Adapter itself. Using **on demand** or **every time** initialization of the Adapter, the calling AL can use information retrieved during its execution phase to configure the Adapter, rather than having to pre-configure this through more static mechanisms as was the case before TDI 6.1.

The use of operations in a TDI Adapter

To implement the TDI Connector modes, AL operations must be created in the Adapter that correspond to the Connector primitives that any Connector has to implement, just as if it was implemented in Java or JavaScript. The AL Connector will automatically determine what modes are available by inspecting the operations that have been defined in the Adapter. It's only necessary to implement the operations that correspond to the modes that you want the Adapter to expose. Operations that do not correspond to any in the table below are exposed as additional Adapter modes, and executed in call or reply mode by the AL Connector.

Mapping Adapter operations to Connector modes: The following table shows the methods that you have to consider when implementing a TDI Connector in Java or JavaScript. Refer to the "Implementing your own components" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*, to fully understand the relationships between these methods and the Connector modes that they implement. For example, to implement Lookup mode in an Adapter, only the `findEntry` operation needs be defined.

Table 2. Adapter operations and Connector modes

	Iterator	Lookup	Addonly	Update	Delete	Delta	CallReply	Server
initialize	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)
querySchema	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)
selectEntries	(X)							
getNextEntry	X							X
findEntry		X		X	X	(X)		
modEntry				X		X		
putEntry			X	X		X		(X)
deleteEntry					X	X		
queryReply							X	
getnextclient								X
terminate	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)

Notes on table:

- (X) means optional. Will be called if they exist.
- Delta mode is handled in a special manner.
- Server mode is not currently supported in Adapters.
- The operation names are case sensitive.

Implementing code in the Adapter for each operation: The Adapter AL is called by the AL Connector according to the rules for the modes that are exposed – through the operations illustrated in the table above. The Adapter must check for what operation is invoked and execute the corresponding code in the AL. The Switch Component is well-suited for this purpose, but the If-Else Components can be used as well by creating conditions using the `op-entry.$operation` attribute, which will be set each time the AL is called with an operation. This attribute may of course be used in a script as well.

Op-entry is a new Entry object. Like the work object, it's created by the AssemblyLine, but it doesn't get cleared every time the AL cycles. It's used to store attributes that the AL needs throughout its life cycle.

Adapter configuration through the $initialization operation: All attributes that are defined in the schema of the `$initialization` operation of the Adapter are displayed in the configuration window of the AL Connector that calls the Adapter. These attributes are passed to the Adapter at initialization time so that the Adapter can perform the necessary preparation and connection to the target systems.

The attributes defined in the `$initialization` schema are available to the Adapter throughout its life cycle as attributes in the op-entry attribute.

Connectors in the Adapter can be configured with these attributes by using expressions in the Connector parameter fields. For example, if the Adapter has defined an `ou` attribute in its `$initialization` schema, then you see `ou` as one of the configuration parameters in the AL Connector. The Adapter could then define a search base in an LDAP Connector as:

```
cn=...,ou={op-entry.ou}
```

These attributes will be available at the initialization time of the Adapter, which by default is the same time that the calling AL is initialized, unless one of the mechanisms described above in “Flexible connector initialization” on page 41is used.

Understanding the link criteria: The Link Criteria defined in the AL Connector is passed into the Adapter through the search object (SearchCriteria) in an op-entry object.

Extracting the individual criteria objects can be done with the following script code:

```
search = Task.getOpEntry().getObject("search");
criteria = search.getCriteria(0); /* index ranges from 0 to search.size() */
name  = criteria.name;    /* target attribute */
match = criteria.match;    /* expression (less, greater, equal.. */
value = criteria.value;   /* value to test the target attribute against through
 the expression */
negate = criteria.negate;    /* Boolean flag */
```

Each criteria object contains the attributes: name, match, value, and negate (Boolean).

The search object provides convenience methods to create LDAP, Domino and SQL search strings based on its link criteria. Refer to the Javadocs for further information.

Attribute Mapping: When the Adapter operations are called through the AL Connector, the Work Entry is populated with the attributes in the output map of the AL Connector. On return, the AL Connector expects returned attributes either in the Work, or in the `conn` objects.

From the calling AL into the Adapter: The Adapter must use script methods such as:

```
email = work.getString("email");
```

to extract the value of the e-mail attribute so that it can be used in further Attribute Mapping inside the Adapter. A practical suggestion is to insert an AL-level AttMap component early in the Adapter to extract the desired attributes from conn and make them visible in the work object for easy reference in the rest of the Adapter.

Return data from the Adapter to the calling AL: The modes Iterator, Lookup and CallReply, return data to the calling AssemblyLine and populate the output map of the AL Connector. The simplest way to return attributes to the calling AL is through the work object. Any attributes left in the work object at the end of the Adapter cycle will be passed back to the input map of the calling AssemblyLine Connector. It’s therefore important to remove temporary work attributes at the end of the Adapter so that they don’t get returned as well.

```
work.removeAttribute("attributeName");
```

The Adapter indicates end-of-data by returning an empty conn object in the work object. An empty work object is not sufficient since that is merely interpreted as an empty record by the AL Connector. To indicate end-of-data by the Iterator work.newAttribute(“conn”);

Lookup mode: Lookup mode may return multiple records. If it is necessary to return more than one record, the Adapter must create the Entry attribute conn in the Work Entry that can contain zero, one, or more values of type Entry. Further on in this section there’s some script code to illustrate how this can be achieved in an Adapter.

The example is a mix of JavaScript and pseudo-code to illustrate the part of implementing the findEntry operation (that implements lookup mode) where attributes are mapped into a structure that can be returned to the AL Connector in the calling AL.

The example illustrates two ways to return multiple records to the calling AL. The example is simpler because work is cleared for each Iterator cycle, and all the values in the work object are therefore a result of the Iterator's output map, and can therefore be added to acc (acc is shorthand for accumulator) in a single operation. An important note is that getClone() needs to be used to ensure that the value of the attributes are copied into acc.

```
acc = system.newEntry().newAttribute("conn");
```

Loop on Iterator (that returns attributes a,b,c from target into work):

```
{

 /* all of the below would be located in a script
 component inside the Loop component */
 temp = system.newEntry();
 temp.setAttribute(work.getAttribute("a"));
 temp.addAttribute(work.getAttribute("b"));
 temp.addAttribute(work.getAttribute("c"));
 acc.addValue(temp) ;

}
work.setAttribute("conn", acc);
```

Clear work for each Iterator cycle:

```
work.removeAllAttributes();
acc = system.newEntry().newAttribute("conn");
Loop on iterator (that returns attributes a,b,c from
target into work)
{
acc.addValue(work.getClone());
 work.removeAllAttributes();
}
work.setAttribute("conn", acc);
```

Status indication: A good practice is to return the attribute recordsProcessed to indicate how many records were deleted, modified, or otherwise processed. This attribute can be passed back to the calling AL as in the work object. To indicate an error situation where the AL Connector should invoke one of the error Hooks in the calling AL, the Adapter needs to throw an exception. Refer to 237for more details on this.

Implementing Query Schema: A user of the Adapter will want to discover the schema of the Adapter. This is typically done when configuring the AL Connector where there are buttons to connect to and to query schema. If the Adapter implements a static schema, then the simple solution is to create a querySchema operation in the Adapter, and define the schema there. The schema defined in the querySchema operation will be common for all standard Connector modes. Specific schemas can be defined for any non-standard modes. For example, if the Adapter implements an "AddUser" operation, then it can have its own schema defined.

Delta mode: Delta mode is handled somewhat differently from other modes because there are two different cases for handling delta data— meaning an Entry that is tagged for change at the Entry, attribute or value level.

1. If the target system (or implemented in the Adapter) supports change-based modification; for example, LDAP allows individual values to be updated in a specific attribute in a specific entry without supplying any of the other values of the attribute. These systems are defined as "Delta savvy" and indicate that the Adapter can deal with a tagged Entry.

2. For other systems, Delta mode can be simulated by performing either delete, add, or a sequence of find and then applying the proper changes to the record before writing the entire record back with modify. This is something that the AL Connector can do by using the basic Adapter primitives, but the Adapter needs to indicate that this is desired functionality.

To enable Delta behavior in an Adapter, first the Delta operation needs to be defined. The next step is to create an attribute deltaSavvy in the Delta schema. Without the deltaSavvy attribute, the AL Connector simulates the Delta mode as described above. With the deltaSavvy attribute in place, the AL Connector does not call findEntry first, but rather calls the modEntry operation directly where it's the Adapter's job to inspect the attributes for tags and apply the appropriate commands against the target system.

Error handling: Throw an exception in your Adapter code to let the calling AssemblyLine drop you into error Hooks of the AL Connector, such as:

```
throw new java.lang.Exception ("error message");
```

Parsers

Parsers are used in conjunction with a byte stream component (for example, File System Connector) to interpret or generate the structure of content being read or written.

When the byte stream you are trying to parse is not in harmony with the chosen Parser, you get a `sun.io.MalformedInputException`. For example, the error message can show up when using the **Input Map** tab to browse a file.

The IBM Tivoli Directory Integrator Config Editor provides three places where you can select Parsers:

1. In the **Parser** tab of a byte stream Connector
2. In an Apply Parser Action (in the **Action Map** tab of an EventHandler)
3. From your own scripts (for example, Hooks, script components and Custom Script Actions in EventHandlers)

For more information about individual Parsers, see "Parsers" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Character Encoding conversion

Java2 uses Unicode as its internal Character Encoding. When you work with strings and characters in AssemblyLines and Connectors, they are always assumed to be in Unicode. Most Connectors provide some means of Character Encoding conversion. When you read from text files on the local system, Java2 has already established a default Character Encoding conversion which is dependent on the platform you are running. More information about character sets can be found in Appendix D, "Double byte character sets in IBM Tivoli Directory Integrator," on page 229.

The TDI Server has the **-n** command-line option, which specifies the character set of Config files it will use when writing new ones; it also embeds this character set designator in the file so that it can correctly interpret the file when reading it back in later.

However, occasionally you read or write data from or to text files in which information is encoded in different Character Encodings. For example, Connectors

that require a Parser usually accept a **characterSet** parameter in the Parser configuration. This parameter must be set to one of the accepted conversion tables as specified by the IANA Charset Registry (http://www.iana.org/assignments/character-sets).

Availability

Refer to the IANA Charset Registry (http://www.iana.org/assignments/character-sets).

Function components (FC)

Several new function components have been created. A Function Component is an AssemblyLine wrapper around some function or discreet operation, allowing it to be dropped into an AL as well as instantiated or invoked from script. The idea behind FCs is to allow complex components (for example, the Web Services EventHandler) to be split into smaller logical units and then strung together as needed; as well as to provide more visual "helper" objects where custom scripting was necessary before. FCs also offer the functionality previously provided by EventHandler **Actions** (for example, launching ALs, invoking Parsers, and so forth). As with all TDI components, users can easily create their own Scripted FCs, turning custom logic into a library of reusable AL components.

FCs are similar to Connectors in CallReply mode in that they have both Input and Output maps[4]. The Output Map is used to pass parameters to the FC, while the Input Map lets you retrieve and manipulate return data.

Also like Connectors, FCs have state s which can be set to *Active*, *Passive* or *Disabled*. State behavior is identical with that of Connectors. Since FCs are registered as script variables (beans) when the AL starts up, you can access them directly from your script using the name given them in the AL.

```
myFunction.callreply( work )
```

The above example is invoking the AssemblyLine Function called *myFunction*. Note that calling the AssemblyLine Function method `callreply()` will cause Attribute Maps and the normal FC Hook flow to be executed.

Note: When an FC receives no data in return, it invokes the mandatory **No answer returned** Hook.

Like the other components, FCs have a library folder in the Config Browser where you can configure and manage your FC library. You can then either drag the FCs into ALs or select them from the list that appears when you click **Add Component** under the AL Connector List.

Like the other components, FCs have an Interface part (like the Connector Interface or Parser Interface, in the case of FCs called the *Function Interface*) that implements the function logic. When an FC is dropped into an AL, it is wrapped in an *AssemblyLine Function* object which provides the generic functionality necessary for the AL to manage and execute it. For more information on the Config Browser, see Config Browser.

4. The Function Interface method called when the FC is executed during AL cycling is actually `callreply()`.

Java function component

This Function Component lets you open a .jar file, browse and select a method, then populates the schemas for Input and Output Maps with the required parameters.

See the "Function Components" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

SDOtoXML and XMLtoSDO function components

TDI 6.0 featured the Castor JavaToXML and Castor XMLToJava Function Components, which serialized Java bean objects to XML and parsed XML into Java bean objects respectively. These two components (in fact, the entire Castor library and all related components) have been removed in TDI 6.1 and replaced with greater functionality provided by the Eclipse Modelling Framework (EMF) library.

As a result, two new FCs have been added: the SDOtoXML and XMLtoSDO Function Components, both of which leverage Service Data Objects (SDOs) to let you work with arbitrarily complex XML documents.

For details regarding the SDOtoXML and XMLtoSDO Function Components, refer to the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Secure remote command-line function component

The new Secure Remote command-line FC (or RCL FC for short) enables command-line system calls to be executed on remote systems using any of the following protocols: Remote Shell (RSH), Remote Execute (REXEC), Secure Socket Shell (SSH) or Windows. The RCL FC uses the Reparable Exchange Activity (RXA) toolkit to connect to remote systems, execute the commands and return the results. The returned output can then be parsed, to be consumed one value at a time and detect any problems with the executed command.

SendEMail function component

This new component uses the javax.mail package to offer a simple method for connecting to SMTP servers and sending e-mails. It can send e-mails to multiple recipients and includes an option to attach multiple files with different MIME types as well.

See the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information about this Function Component.

Common base event (CBE) function component

The CBEGeneratorFC is used for creating CBE event objects. The FC allows you to select between either generating a CBE Java object, which can then be passed to a Comment Event Infrastructure (CEI) Server using the TEC Web services, or an XML document that adheres to the Hyades CBE Logging format and that can then be used to write a CBE-style log file.

See the "Function Components" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information about this Function Component.

AssemblyLine connector and function component

Both the AL Connector and the AL FC support AL Operations. See “Operations” on page 145 for more information.

The Function Interface

There is also a `system.getFunction()` method to instantiate Function Interfaces (the underlying function itself) that works similarly to the `system.getConnector()` function. In fact, using an FC from script is very similar to working with a Connector Interface.

```
// Get FC (Function Interface) named "customFC"
var fc = system.getFunction( "customFC" );
// Initialize the FC
fc.initialize( null );
// Invoke the function, passing in the object needed (work Entry in this case)
// This FC returns an Entry object when the call is successful
var retEntry = fc.perform( work );

// Close the FC
fc.terminate();
```

As with Connector Interface functions, making the `perform()` call above will not execute Attribute Maps or Hook flows.

As seen in the above example, FCs have an interface of three main methods[5]:

initialize(obj)
: Initializes the Function Component. The *obj* parameter is the parameter block as described by your FCs **Config** dialog.

terminate()
: Closes down the FC, releasing resources and so forth

perform(obj)
: This calls the function itself, making the FC do its work. Here again the *obj* parameter is whatever you have defined as necessary input parameters for your function. For example, this can be an Entry object, or a string command, as described below under the AssemblyLine Function Component.

The individual Function Components which are part of IBM Tivoli Directory Integrator are described in detail in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Link Criteria

The Link Criteria is used to tell a Connector in Update, Lookup and Delete modes how you define a match between data attributes in the AssemblyLine and those found in the connected system. The Link Criteria is accessible in the Config Editor through the **Link Criteria**, which is only enabled for Update, Lookup and Delete Connector modes.

There are two types of Link Criteria, **Simple** and **Advanced**.

Simple Link Criteria

For each simple Link Criteria, specify the **Connector Attribute** (those attributes defined in the Connector Schema), the **Operator** to use (for example, Contains, Equals, and so forth), and the **Value** to use with the operation. The value you use can be entered directly, or it can refer to the value of an attribute in the Work Entry that is available at this point in the AssemblyLine flow. When the Connector performs the Lookup operation (for Lookup, Update and Delete modes) it converts

5. You can optionally implement the `getUI( fc)` method, returning a JComponent that describes the Config window of your FC, allowing you to create a dynamic Config interface for your FC. Note that the other TDI components also provide this feature.

the Link Criteria to the data source-specific call, enabling you to keep your solution independent of the underlying technology.

If you want to build a Link Criteria using the value of an attribute in the work Entry, simply use the name of the attribute in the **Value** field of the Link Criteria, preceded by the dollar sign (**$**). So, if you want to match the attribute named **cn** with an attribute in the **work** Entry called **FullName**, your Link Criteria is specified as:

```
cn EQUALS $FullName
```

If you want to find a specific person directly, set the Link Criteria with a literal constant value:

```
cn EQUALS Joe Smith
```

Note: The dollar sign ($) matches the first value of a multi-valued attribute only. If you want to match an attribute in the data source with any of the multiple values stored in a **work** Entry attribute instead, then use the *at* symbol (**@**). For example:

```
dn EQUALS @members
```

This example tries to match the **dn** attribute in the connected system to any one of the values of the multi-valued attribute in the **work** Entry named **members**

A Connector can have multiple Link Criteria defined, and these are normally connected together (by use of the Boolean operator AND) to find the match.

However, if you click **Match Any** , just one of the Link Criteria needs to match, the equivalent of an OR operation.

Note that the name of the**Attribute** to match can be specified as an Expression (See “Expressions” on page 110 for details). The possible formats for the **Value** field of a Simple Link Criteria are:

A text string
Mapped to a constant with that value.

$*Name* Corresponds to *work.getString("Name")*, that is the first value of the attribute *Name*.

@*Name*
One of the values of the multi-valued attribute *Name*.

A TDI Expression
Described in detail here: “Expressions” on page 110.

Advanced Link Criteria

You can also create your own custom search criteria by checking the **Build criteria with custom script** check box. This presents you with a script editor to write your own Link Criteria expression. Not all Connectors support the Advanced Link Criteria, and the Connector documentation states whether Advanced Link Criteria is supported. See "Connectors" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

The search expression that you build must comply with the syntax expected by the underlying system. In order to pass your search expression to the Connector, you must populate the **ret.filter** object with your string expression.

A simple JavaScript example for an SQL Connector is:

```
ret.filter = " ID LIKE '" + work.getString("Name") + "'";
```

This custom Link Criteria assumes an example where the data source has an attribute called **ID** (typically a column name) that we want to match with the **Name** attribute in the **work** Entry.

Notes:

1. The first part of the SQL expression, `Select * from Table Where`, is provided by the IBM Tivoli Directory Integrator.
2. Single quotation marks have been added because **work.getString()** returns a string, while SQL syntax asks for single quotation marks around strings constants.
3. The special syntax with **$** and **@** is not used here.

Link Criteria errors

The most common error you get when using Link Criteria is:

```
ERROR> AssemblyLine x failed because
No criteria can be built from input (no link criteria specified)
```

This error occurs when you have a Link Criteria that refers to an attribute that cannot be found during the Lookup. For example, with the following Link Criteria:

```
uid equals $w_uid
```

Link Criteria setup fails if the **w_uid** is not present in the **work** entry. This might be because it is not read from the input sources (for example, not in an Input Map, or missing from the input source) or is removed from the **work** Entry in a script. In other words, the function call `work.getAttribute("w_uid")` returns **null**.

One way to avoid this is to write code in the **Before Execute** Hook of the Lookup, Delete or Update mode Connector that skips its operation when the Link Criteria cannot be resolved due to missing attributes. For example:

```
if (work.getAttribute("w_uid") == null)
  system.ignoreEntry();
```

Your business logic might require other processing, such as a `skipEntry()` call instead of `ignoreEntry()`, which causes the AssemblyLine to stop processing the current entry and begin from the top on a new iteration. The ignoreEntry() function simply skips the current Connector and continues with the rest of the AssemblyLine.

EventHandlers

The concept of EventHandlers is replaced by Server Connectors in IBM Tivoli Directory Integrator 6.1.1. In place of EventHandlers, you can use a suitable Connector in “Server mode” on page 33 to simplify the task of event handling.

Note: In pre-6.1 Configs that employ them, you can still use EventHandlers.

Scripting

IBM Tivoli Directory Integrator provides its users with a highly-flexible engine that can be customized both from the user interface controls of IBM Tivoli Directory Integrator Config Editor, as well as through scripting of custom logic. While the user interface controls provide a means of controlling the data flow at a higher level, scripting provides users with the ability to control almost any aspect of the

data flow at any level (including overriding standard IBM Tivoli Directory Integrator processing). Special functions are available in the **system** object to reiterate on an AssemblyLine Entry, skip a Connector and start new AssemblyLines or EventHandlers.

IBM Tivoli Directory Integrator provides support for JavaScript as a scripting language.

Attention: Support for any scripting language other than JavaScript is withdrawn from IBM Tivoli Directory Integrator. This means that support for scripting languages like JScript and VBScript is no longer in effect, and any of your solutions depending on those languages to interface with Windows objects should be converted to use the COMProxy Object, which provides the glue between the JavaScript language and COM Objects on Windows. Refer to the Javadocs for more information about the COMProxy Object.

See "Script languages" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information on how scripting is supported in IBM Tivoli Directory Integrator.

Controlling the flow of an AssemblyLine

Hooks are programmable waypoints in the built-in automated behavior of IBM Tivoli Directory Integrator where you can impose your own logic. Hooks are found in EventHandlers, AssemblyLines and Connectors. For example, if you want to skip or restart parts of the AssemblyLine entirely, you typically do this from within a Hook in a Connector:

Note: The constructs below can be used to exit a Branch Component or Loop, too.

system.ignoreEntry()
Ignore the current Connector and continue processing your existing data **entry** with the next Connector.

system.skipEntry()
Skip (drop) the **entry** completely, (aborting the current cycle), return control to the start of the AssemblyLine and get the next **entry** from the current Iterator.

system.exitFlow()
Drop further processing of the current Entry, execute end-of-cycle logic (for example, save the Iterator State Key (if the Connector is configured for this)), return control to the start of the AssemblyLine and get the next **entry** from the current Iterator.

system.restartEntry()
Restart from the beginning of the AssemblyLine, forcing the current Iterator to reuse the current **entry**.

system.skipTo(String name)
Skip to the named Connector.

system.abortAssemblyLine(String reason)
Abort the entire AssemblyLine with the specified error message.

Note: If you put any code in an **Error** Hook and do not terminate the current AssemblyLine or EventHandler, then processing continues regardless of how you got to the **Error** Hook. This means that even syntax errors in your script are ignored. So be sure to check the **error** object if you want to know what caused the error.

The methods described in the previous list can be regarded as goto-statements, in that no further code in this Hook is run. For example:

```
system.skipEntry(); // Causes the flow to change
// This next line is never executed.
task.logmsg("This will never be reached");
```

When scripting is needed

Scripting is necessary when you need to add custom processing to your AssemblyLine or EventHandler. Examples of where scripting can be helpful include:

- The value of an output attribute needs to be calculated based on one or more input attributes (attribute manipulation or computation).
- You want to process only entries that match a particular set of criteria (data filtering).
- Invalid data values need to be reported or corrected (data consistency or validity checking).
- You want to override the update operation of the Connector you are using (flow control).
- You want to run some initializing procedures before your AssemblyLine starts (initialization).

Each of these cases mentioned (and many others not mentioned) require scripting.

Integrating scripting into your solution

As already explained, you use script whenever you need custom processing in your integration solution. Best practices with IBM Tivoli Directory Integrator divide this custom processing into two categories: attribute transformation and flow control.

Note: This is convention, and not a limitation or rule enforced by the system. The need for custom data processing inevitably comes at some identifiable point in the flow of data (for example, before any processing begins, before processing a particular entry, after a failure, and so forth), so by placing your code as close to this point as possible, you can make solutions that are easier to understand and maintain.

The logical place to do the attribute transformations is in your **Attribute Maps**, both Input and Output. If you need to compute a new attribute that is required by scripted logic or other Connectors downstream in the AssemblyLine, best practice is to do this in an **Input Map** if possible. Alternatively, if you must transform attributes for the sake of a single output source, then you can avoid cluttering the **work** Entry object with output-specific transformations by putting these in the relevant Connector's **Output Map**.

The other category of custom logic, flow control, is best implemented in the Hooks that are invoked at that point in the automated workflow where the logic is needed. These control points are easily accessed from the IBM Tivoli Directory Integrator Config Editor. Implementing custom processing is simply a matter of identifying the correct control point and adding your script in the appropriate edit window.

AssemblyLine **Script Components** also provide you with a place to create your own custom processing, and then enable you to reposition your code within the AssemblyLine. Although Script Components are frequently used during test and debugging, they can also serve an important role in a production Config. Just

remember to name your components clearly and to include some documentation in the script itself to explain (to others, as well as yourself when you have to revisit your code some time later) why you implemented this logic in a Script Component, and not in an **Attribute Map** or **Hook**.

And while it is important to both correctly identify the appropriate control point where you input your script, it is equally important to limit the scope of your script to cover just the single goal associated with the control point. If you keep your units of logic independent of each other, then there is a greater chance that they are reusable and less of a chance that they might break when components are reordered or reused in other contexts. One way to build reusable code is by creating your own functions in your **Script Library** (or a **Prolog** Hook) to implement frequently used logic, instead of copying and pasting the same code into multiple places.

To sum up some of the best practices that you want to keep in mind while building solutions:

- Do attribute manipulation in Attribute Maps.
- Put flow control (filtering, validation, branching, and so forth) in Hooks, and where necessary, AssemblyLine script components.
- Use the automated behavior as much as possible (for example, AssemblyLine workflow and Connector modes).
- Simplify your solution by keeping AssemblyLines short and focused.
- Put often used logic in functions (for example, Script Library).
- Think reuse.

It is worth mentioning again that although the methods outlined previously are best practices, you might encounter situations where you have to deviate from established convention. If this is the case, your solution documentation is vital for maintaining and enhancing your work over time.

How scripting affects execution

The IBM Tivoli Directory Integrator engine exposes a number of classes and objects that can be accessed, read and modified from user-created scripts in an EventHandler or AssemblyLine. These objects represent the state of the EventHandler or AssemblyLine and the whole IBM Tivoli Directory Integrator environment at any moment. By modifying any of these objects, you modify the IBM Tivoli Directory Integrator environment and thus affect the execution of the integration process.

Note: Changes can be applied to either instances of a component, AssemblyLine or EventHandler. Changes can also be made to operational parameters (such as system or Java parameters). Changes can also be made to the Config. In this case, new instances of Config objects reflect these Config changes.

For more information about the global objects, see the Javadocs included as part of the IBM Tivoli Directory Integrator product (select **Help > Low Level API** in the Config Editor).

A description of all classes and instances available can be found in the installation package.

By understanding the classes and interfaces exposed, you can better understand the elements of the IBM Tivoli Directory Integrator engine as well as the language it speaks.

Using variables

It is important to distinguish between the standard container object for data being processed (the Entry object) and other generic variables and data types provided to you by your chosen scripting language, as well as those that you create yourself. Your creativity and the capabilities of the scripting languages are your only restrictions in terms of what can be placed in the scripting windows. However, when you manipulate data in the context of the data flow, you must be aware of and use the structure of the Entry object.

Entry objects carry attributes, which are themselves the container for data values. Attribute values are themselves objects (java.lang.String, java.util.Date and more complex structures). An attribute value can even be another Entry object with its own set of attributes and values. It is the job of IBM Tivoli Directory Integrator to understand how data is stored in the connected system, as well as how to convert these native types to (and from) the system's own data representation (Java objects).

If you open the **Input Map** tab for a Connector, click **Connect to the data source** , and then click **Read the next entry**. You see the attributes discovered in the first object found in the data source. The column labelled **Java Class** shows you how the Connector is converting native data types to the system's internal Java representation. Click **Discover the schema of the data source** to display the full definition of attributes defined for the object you are looking at (for some data sources). The column named **Native Syntax** displays the data types used by the data source to represent these values.

If you know the attribute value's class, you can successfully access and interpret this value. For example, if a `java.lang.String` attribute contains a floating point value that you want to use as a floating point, you must first manually transform this value (by means of the scripting language) to some numeric data type.

When creating variables or processes not directly related to the data flowing in the integration process and the global objects available, the following principle applies: You can declare and use any variables (objects) enabled by the scripting language you choose. The purpose of these variables is to help you achieve the specific goal associated with the control point in which you script. The variables must serve only as temporary buffers and not attempt to affect the state of the IBM Tivoli Directory Integrator environment.

Control points for scripting

Scripting in an AssemblyLine

AssemblyLine Hooks: These Hooks are found in the **Hooks** tab of the AssemblyLine. These Hooks are all executed only once per AssemblyLine run, or, in the case of **Shutdown Request**, whenever the AssemblyLine is told to shutdown by some external process. However, if you start your AssemblyLine multiple times (for example, by using an EventHandler), then you start the Hooks multiple times as well.

Script Component

You can add Script Components to your AssemblyLine in addition to Connectors by clicking the **Add script component** button under the **Data Flow** tab in the AssemblyLine. The Script Component is started once for each entry processed by the AssemblyLine (such as a Connector) and can be placed anywhere in the AssemblyLine.

Note: Iterators are still processed first, even if you place your Script Component before them in the AssemblyLine.

Scripting in a Connector

Input Map and Output Map

Custom Attribute Mapping is performed in these tabs. When the attribute is selected, you must select the **Advanced Mapping** check box, and input your script in the edit window. Remember that after you have done all processing necessary, you must assign the result value achieved to **ret.value**, for example:

```
   ...
ret.value = myResultValue;
```

Connector Hooks

Hooks give you the means to respond to certain events that occur, and override the basic functionality of a Connector. You have access to the global IBM Tivoli Directory Integrator objects when scripting Hooks, although some of the standard objects might not be available in every Hook. For details on temporary object availability, see ″AssemblyLine and Connector mode flowcharts″ in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*. You also have full control over the IBM Tivoli Directory Integrator environment, the AssemblyLine, the Connector, entries and attributes. Hooks give you a diversity of control points for customizing the process flow. See “Hooks” on page 69.

Setting internal parameters by scripting

It is possible to set Internal Parameters for a Connector using the following script:

```
myConnector.setParam ( "filePath", "examples/scripting/sample.csv" );
```

This is typically something you do in the Prolog, but it can be very useful while the AssemblyLine is running as well (provided that you stop and reinitialize the Connector).

```
myConnector.terminate()
myConnector.setParam ( "filePath", "examples/scripting/sample.csv" );
myConnector.initialize(null);
```

Scripting in a Parser

Scripting in a Parser actually refers to implementing your own Parser by scripting. A description of this process is included in ″Script Parser″ in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Scripting in TDI

Ready-to-use, TDI provides the tools to quickly snap together the framework of an integration solution. However, for all but the most trivial migration jobs, you will need to customize and extend the built-in behavior of TDI by writing JavaScript.

IBM Tivoli Directory Integrator is pure Java. Whenever you issue a command to TDI, work with components and objects, or manipulate data in your flow, you are working with Java objects. TDI 6.1 uses IBM Java version 1.5.

Your customization on the other hand is done in JavaScript, and this marriage of two ostensibly similar, yet fundamentally different, programming languages warrants closer examination.

If you have experience with JavaScript, great; if you don't, then you will be getting some here in these examples. However, this manual does not teach JavaScript itself – merely its application in TDI. You will need to secure your JavaScript reference materials elsewhere.

There are a number of commercially available reference guides to JavaScript, as well as documentation, tutorials and examples on the net. Note however that much of the JavaScript content out on the Web is related to beautifying and automating HTML content. You need only concern yourself with the core language itself, as it is described at the following link:

http://devedge-temp.mozilla.org/library/manuals/2000/JavaScript/1.5/guide/index.html

There is also a handy link on this site for downloading the reference in HTML format for installation locally. An excellent handbook on JavaScript is *"The Definitive JavaScript Guide"*, 4th Edition by David Flanagan (O'Reilly).

You will also want the Javadocs for Java as well, since all TDI objects, as well as the data values inside your solution, are in the form of Java objects. These documents are located online at this URL:

http://java.sun.com/j2se/1.4.2/docs/api/index.html

The J2SE documentation itself can be found here:

http://java.sun.com/j2se/1.4.2/docs/index.html

JSengine Integrated into TDI

The integration of the IBM JSengine with TDI requires changes to the com.ibm.di.script.ScriptEngine. That Java class is used by all components where a script engine is required.

Note: Rhino is no longer available as a ScriptEngine for TDI. The only requirement that may cause some compatibility issues is the replacement of the Rhino JavaScript engine with the IBM JSengine. Great care is taken to minimize possible compatibility issues, and the IBM JSengine is being updated as issues are found.

The use of the IBM JSengine is optimized in that ScriptEngine saves the compiled expressions derived from evaluating a JavaScript code segment. Hence, ScriptEngine provides similar functionality to Rhino for enhancing execution speed. The TDI ScriptEngine (which wraps the JSengine inside TDI) works in the same way as it did with Rhino, so your AssemblyLines won't notice any difference.

Advantages of the IBM JSengine include:

- Better platform for debugging features. Many of the new Debugger/Stepper features are possible because of the JSengine.
- Improved eval() method also allows registration of functions, not just variables as it did in Rhino.

- Improved error messages .

Improved error messages: The IBM JSengine JavaScript engine (jsengine) provides better clear and precise error messages (where possible). To facilitate finding syntax errors, the error message and the exception from the IBMJS engine are displayed. It also shows the line and column numbers in the script. An example of JSengine information displayed: If your script had the following code in its third line:

```
this is an error;
```

Then error information is displayed.

```
com.ibm.jscript.parser.ParseException: Syntax error at line 3, column 6; Invalid 'is'.
```

The first error encountered in the above line was "is", because "this" is a reserved word in JavaScript.

Library Loader enhancements: In addition to providing a manual method for you to specify a single additional directory to pick up libraries that are loaded by TDI, specific files and directories of jar files can be configured. In addition, the organization of the jars directory changes to provide better organization of class files.

New User Function allows you to add jar files dynamically: A new method to UserFunctions, **loadJarFile(String)** is added. The method allows you to add jar files dynamically while running TDI. The String parameter can be either a single jar file or a directory containing jar files (or new directories, which are searched recursively). For example, `system.loadJarFile(c:\\myjardirectory)`.

Custom specification of JAR files: The previous version provided a property, “com.ibm.di.loader.userjars”, for specifying a single directory containing jar files. This property is extended in TDI 6.1 to allow solution builders to specify more than one directory or jar file, separated by the Java property ″path.separator.″ The “path.separator” is ″:″ on UNIX/Linux platforms and ″;″ on Windows platforms. Directories are searched recursively by the TDILoader for jar files containing classes and resources. The TDI loader behaves otherwise as it always has, and only files with a ″.zip″ or ″.jar″ extension are searched.

A new method to UserFunctions, loadJarFile(String) has also been added. The method allows a user to add jar files dynamically while running TDI. The String parameter can be either a single jar file or a directory containing jar files (or new directories, which are searched recursively). For example,

```
system.loadJarFile("c:\\myjardirectory");
```

Note: JAR files added using the com.ibm.di.loader.userjars or the loadJarFile() method, cannot contain idi.inf files (the information in the idi.inf files is not used, and is only read during initial loading at system startup). JAR files containing such information must be placed in the jars subdirectory. This is because of the way TDI constructs the system namespace today.

Restructuring of the TDI jars subdirectory: The TDI `jars` subdirectory is restructured. The JAR files in this directory have been moved into subdirectories to organize them better and the TDILoader is updated to understand the directory changes.

The options specify the new directory structure under the `jars` subdirectory and determines which JAR files go under each category:

connectors
Contains all TDI Connector JAR files.

eventhandlers
Contains all TDI EventHandler JAR files.

functions
Contains all TDI Function Component JAR files.

parser Contains all TDI Parser JAR files.

plugin
JAR files needed for the TDI server to communicate with the plug-ins.

common
The core server JAR files.

patches
Used to hold patch JAR files for the support stream.

3rdparty\IBM
Contains all of the JAR files TDI needs from other IBM products.

3rdparty\others
Contains all of the JAR files TDI needs from non-IBM products.

ce Contains the JAR files that are needed by the CE and not the server.

The TDILoader is updated to first search the subdirectories in the following order in every directory:

1. patches
2. common
3. connectors
4. eventhandlers
5. functions
6. parsers
7. plug-ins

When looking for a class, TDI uses the first instance found. Now that the order is specified in IBM Tivoli Directory Integrator, you can drop fixed components and other jars into the patches subdirectory , overriding installed classes. This makes it easy to test patches without corrupting an installation (or having to make backup copies of .jar files first). Furthermore, this makes it easy to see if an installation has any patches installed, since they are all in the same place.

Script editor

TDI provides a simple script editor which offers common features like copy, cut and paste, undo and redo and free-text searching. If you already have a preferred script editor, you can configure TDI to use it under the **Editor Settings** tab of the **File > Edit Preferences** dialog. This will allow you to use the **Launch External Editor** button found in the Toolbar of the internal script editor.

TDI Internal Data Model (Entries, Attributes and Values)

When TDI components access information from connected systems, they convert the data from system-specific types to an internal representation using Java objects. On output, components convert the other way, going from this internal data model to the native types of the target system. This same internal representation is used when you wish to pass data to and from AssemblyLines. It is therefore vital that you understand how the TDI internal data model works.

Looking in detail at when a data value is received by a component, a corresponding TDI *Attribute* object is created using the name of the attribute being read. The data value itself (or values, if it is a multi-valued attribute) are converted to appropriate Java objects—like `Java.lang.String` or `java.sql.Timestamp` —and assigned to the Attribute. If you take a look in the TDI Javadocs, you will see that the Attribute object provides a number of useful methods, like `getValue()`, `addValue()` and `size()`. This allows you to create, enumerate and manipulate the values of an Attribute directly from script. You can also instantiate new Attribute objects as needed, as shown in this example for advanced mapping the objectClass attribute of a directory:

```
var oc = system.newAttribute( "objectClass" );

oc.addValue( "top" );
oc.addValue( "person" );
oc.addValue( "organizationalPerson" );
oc.addValue( "inetOrgPerson" );

ret.value = oc;
```

TDI provides some shortcuts and convenience features when working in JavaScript, so the above specific advanced mapping can be simply coded as follows:

```
ret.value = [ "top", "person", "organizationalPerson", "inetOrgPerson" ];
```

The advanced mapping feature supports JavaScript arrays for passing multiple attribute values.

Attributes themselves are collected in a data storage object called an *Entry* object. The Entry is the primary data carrier object in the system and TDI gives you access to important Entry objects by registering them as script variables. A prime example is the *Work Entry* object in the AssemblyLine, used to pass data between AL components (as well as between AssemblyLines). This Entry object is local to each AssemblyLine and available as the script variable *work*.

Looking at the Javadocs, you will see that the Entry object offers various functions for working with Entries and their Attributes and values, including `getAttributeNames()`, `getAttribute()` and `addAttribute()`. If we wanted to create and add an Attribute to the AssemblyLine Work Entry, we could use the following script (for example, in a Hook or a Script Component):

```
var oc = system.newAttribute( "objectClass" );

oc.addValue( "top" );
oc.addValue( "organizationalUnit" )

work.addAttribute( oc );
```

Note that in this case we do **not** have the option of using a JavaScript array to set the value:

```
oc.addValue( ["top", "organizationalUnit"] );  // Does not work like Advanced Mapping
```

This code will result in the oc attribute getting a single value, which in turn is an array of strings.

Entry objects can also contain *properties*. Properties are data containers like Attributes, except that they are only single valued. While Attributes are used to store data content, properties hold parametric information, allowing you to keep this information separated. Properties do not show up for Attribute Map selection or in the Work Entry list, but can be accessed much like Attributes from script. Entry functions like `getProperty()` and `setProperty()` are used for this, and these

work directly with Property values (which can be any type of Java Object, just like Attribute values). There is no intermediate Property object as there is when we work with Attributes.

So this is how the TDI internal data model looks: An Entry containing zero or more Attributes, each with zero or more values—a flat schema.

This is one of the strengths of TDI: simplifying and harmonizing data representations and schema. It also represents a challenge when you need to handle information with a more complex structure. However, since an Attribute value can be any type of Java object, including another Entry object (with its own Attributes and values), TDI allows you to work with hierarchically structured data. The TDI CE visually supports only a flat schema, so you must deal with the hierarchy yourself. For example, if you are reading in a complex XML document, you could either ″flatten″ the data before input (for example, using XSLT), or you could read the document and then work directly with the resulting Document Object Model tree in your script.

The Script Component

The Script Component (SC) is a user-defined Hook that you can drop any place in the AssemblyLine **Data Flow** list, alongside your Connectors and Function Components, causing the script code within to be executed for each cycle at this point in the AL workflow. Unlike Hooks, Script Components are easily moved around in the AssemblyLine flow, making them very powerful tools for everything from debugging to prototyping and implementing your own flow logic.

For example, if you want to test and debug only part of an AssemblyLine, you could put the following code in an SC to limit and control AL flow.

```
task.dumpEntry( work );
system.skipEntry();
```

This SC then displays the contents of the `work` object (writing it to log output) and then skip the rest of the AL for this cycle. By moving the SC up and down the component list, you control how much of the AL is actually executed. If you swap out the `system.skipEntry()` call with `system.skipTo("ALComponentName")`, you directly pass control to a specific AL Component.

You can also use SCs to drive other components. A typical scenario when doing directory or database synchronization is having to handle both updated and deleted information. Since Connectors powered by the built-in AL workflow can only operate in one mode at a time (Update or Delete) you will need to extend this logic a bit with your own code. One method is to add two Connectors, one in Update mode and one set to Delete, and then put code in each Connector's Before Execute Hook to tell it to skip change operations that it should not handle. For example, in the Update Connector's *Before Execute* Hook you would write something like this:

```
// The LDAP change log contains an attribute called "changeType"
if (work.getString("changeType").equals("delete"))
 system.ignoreEntry();
```

This will cause your Update mode Connector to skip deleted Entries. You would have complementary code in the *Before Execute* Hook of your Delete mode Connector, skipping everything but deletes.

However, if you are synchronizing updates to multiple targets, this would require you to have two Connectors per data source. Another approach is to have a single

Connector in Passive state that you power from script. As an example, let's say you have a Passive AL Connector called *synchIDS*. You can then add an SC with the following code to drive it[6]:

```
if (work.getString("changeType").equals("delete"))
 synchIDS.deleteEntry( work )
else
   synchIDS.update( work );
```

As long as you label your SC clearly, indicating that it is powering Passive Connectors, this approach will result in shorter AssemblyLines that will be easier to read and maintain. This is an example of having to choose between two best practices: keeping the AL short, and using built-in logic versus custom script. However, in this case the goals of legibility and simplicity are best served by writing a little script.

The Script Component also comes in handy when you want to test logic and script code. Just create an AssemblyLine with a single Script Component in the **Data Flow** list, put in your code and run it.

Java + Script ≠ JavaScript

JavaScript is **not** Java. It may look like Java, but it is actually just close enough to really cause confusion. JavaScript was originally called *Live!Script* back when Netscape first created it. Although there is broad support for JavaScript, you will learn that *dialects* exist; for example, Microsoft's version, called *JScript*. There is a standard definition, which is known as *ECMAScript*, and you can find its specification at this URL:

http://www.ecma-international.org/

Although the syntax is similar, Java and JavaScript deal with data and data types differently. This is one of the main sources of errors when working with JavaScript.

Data Representation: Java supports something called *primitives*, which are simple values like signed integers, decimal values and single bytes. Primitives do not provide any functionality, only non-complex data content. You can use them in computations and expressions, assign them to variables and pass them around in function calls. Java also provides a wealth of objects that not only carry data content (even highly complex data), but also provide intelligence in the form of object functions, also known as methods.

When you make the script call to `task.logmsg( "Hello, World" )`, you are calling the task object's `logmsg()` method.

Many Java primitives have corresponding objects. One example is integers, which can used in their primitive form *(int)* or by manipulating java.lang.Integer objects.

JavaScript does not understand the concept of primitives. Instead, all data is represented as JavaScript objects. Furthermore, JavaScript has only a handful of native objects as compared to Java's rich data vocabulary. So while Java differentiates between non-fractional numeric objects and their decimal counterparts – even distinguishing between signed and unsigned types, as well as offering similar objects for different levels of precision – JavaScript lumps all numeric values into a single object type called a *Number*.

6. Since Passive Connectors are not powered by the AL logic, it does not matter where they appear in the **Data Flow** list.

As a result, you can get seemingly erroneous results when comparing numeric values in JavaScript:

```
if (myVal == 3) {
 // do something here if myVal equals 3
}
```

If *myVal* was set by an arithmetic operation, or references a Java decimal object, the object's value could be 3.00001 or 2.99999. Although this is very close to 3, it will not pass the above equivalency test. To avoid this particular problem, you can convert the operand to a Java Integer object to ensure a signed, non-fractional value. Then your Boolean expression will behave as expected.

```
if (java.lang.Integer( myVal ) == 3) { ...
```

Or you can make sure that your variables reference appropriate Java objects to begin with. In general, you will want to be conscious of the types of objects you are using.

Ambiguous Function Calls: Java also provides a primitive type called a *char*, which can contain a single character value. A collection of characters could be represented in Java as an array of character primitives, or it could be handled as a java.lang.String object. As mentioned before, JavaScript does not savvy primitives. Character data must be dealt with using the JavaScript String object. Even if you specify a single character in JavaScript ("a"), this is considered a *string*.

Now consider that when you call a Java function from your script, this is matched up to the actual method using the name of the function as well as the number and types of the parameters used in the call. This matching is carried out by the LiveConnect extension to JavaScript. LiveConnect does its best to figure out which function signature you are referring to, which is no small task given that Java and JavaScript represent parameter types in different ways. But JavaScript and LiveConnect do some behind-the-scenes conversions for you, trying to match up Java and JavaScript data types.

A problem arises when you have multiple versions of a single method, each taking a different set of parameters. In particular, if two functions have the same number of parameters, but of different types, AND if these types are not differentiated in JavaScript. Let's take a look at an example script that will be performing an MD5 encryption of a string.

```
// Create MessageDigest object for MD5 hash
var md = new java.security.MessageDigest.getInstance( "MD5" );

// Get the EID attribute value as byte array.
var ab = java.lang.String( "message to encrypt" ).getBytes();

md.update( ab );

var retHash = md.digest();
```

The above `update()` call will fail with an evaluation exception stating that the function call is ambiguous. This is because the `MessageDigest` object has multiple versions of this function with similar signatures: one accepting a single byte and one expecting a byte array (`byte[]`). We can get around this if we can find another variant of the same method taking a different number of parameters, thus giving it a uniquely identifiable signature. Fortunately, `MessageDigest` has what we are looking for: a version of `update()` that takes a byte array plus a couple of numeric values (offset and length parameters). So we can change our code to use this call instead:

```
md.update( ab, 0, ab.length );
```

Finally, you can always specify the exact signature of the Java method you want to use by quoting it inside brackets after the object:

```
md["update(byte[])"](ab);
```

Here we are calling for the version of the `update()` function declared with a single byte array parameter.

Char/String data in Java vs. JavaScript Strings: Both Java and JavaScript provide a String object. Although these two types of String objects behave in a similar fashion and offer a number of analogous functions, they differ in significant ways. For example, each object type provides a different mechanism for returning the length of a string. With Java Strings you use the `length()` method. JavaScript Strings on the other hand have a `length` *variable*.

```
var jStr_1 = new java.lang.String( "Hello, World" ); // Java String
task.logmsg( "the length of jStr_1 is " + jStr_1.length() );

var jsStr_A = "Hello, World"; // JavaScript String
task.logmsg( "the length of jsStr_A is " + jsStr_A.length );
```

This subtle difference can lead to baffling syntax errors. Trying to call `jsStr_A.length()` will result in a runtime error, since this object has no `length()` method.

Even more confounding mistakes can occur with string comparisons.

```
var jsStr_A = "Hello, World"; // JavaScript String
var jsStr_B = "Hello, World"; // JavaScript String

if ( jsStr_A == jsStr_B )
 task.logmsg( "TRUE" )
else
 task.logmsg( "FALSE" );
```

As expected, you will get a result of ″TRUE″ from the above snippet. However, things work a little differently with Java Strings.

```
var jStr_1 = java.lang.String( "Hello, World" ); // Java String
var jStr_2 = java.lang.String( "Hello, World" ); // Java String

if ( jStr_1 == jStr_2 )
 task.logmsg( "TRUE" )
else
 task.logmsg( "FALSE" );
```

This will result in ″FALSE″, since the equivalency operator above will be comparing to see if both variables reference the *same object* in memory, instead of matching their values. To compare Java String values, you must use the appropriate String method:

```
if ( jStr_1.equals( jStr_2 ) ) ...
```

But wait, there's more. This next snippet of code will get you a result of ″TRUE″:

```
var jsStr_A = "Hello, World"; // JavaScript String
var jStr_1 = java.lang.String( "Hello, World" ); // Java String

if ( jsStr_A == jStr_1 )
 task.logmsg( "TRUE" )
else
 task.logmsg( "FALSE" );
```

Since JavaScript cannot operate on an unknown type like a Java String object, it first converts `jStr_1` to an equivalent JavaScript String in order to perform the evaluation.

In summary, be aware of the types of objects you are working with. And remember that TDI functions always return Java Objects. Keeping these factors in mind will help minimize errors in your script code.

Variable scope and naming: JavaScript is a relatively informal language and does not require you to define variables before assigning values to them. Neither does it enforce strict type checking, or complain when a variable is redefined. This makes JavaScript fast and easy to work with, but can quickly lead to illegible code and confounding errors, especially since you can create variables that overwrite built-in.

One bug that can defy debugging is when you declare a variable with the same name as a built-in one, like `work`, `conn` and `current`, so you will need to familiarize yourself with the reserved names used by TDI.

Another common problem occurs when you create new variables that redefine existing ones, perhaps used in included Configs or Script Libraries. These mistakes can be avoided if you are conscious about the naming of variables and their *scope*. Scope defines the sphere of influence of a variable, and in TDI we talk about global variables — those which are available in all Hooks, Script Components and Attribute Maps — and those that are local to a function.

To get a better understanding of scope you must first understand that every AL has its own Script Engine, and therefore runs in its own script context. Any variable not specifically defined as locally scoped inside a function declaration is global for that Script Engine. So the following code will create a global variable:

```
myVar = "Know thyself";
```

This variable will be available from this point on for the life the AL. Making this variable local requires two steps: using the `var` keyword when declaring the variable, and putting the declaration inside a function:

```
function myFunc() {
 var myVar = "Know thyself";
}
```

Now `myVar` as defined above will cease to exist after the closing curly brace. Note that placing the variable inside a function is not enough; you have to use `var` as well to indicate that you are declaring a new, local, variable.

```
var glbVar = "This variable has global scope";
glbVar2 = "Another global variable";

function myFunc() {
    var lclVar = "Locally scoped within this block";
    glbVar3 = "This is global, since "var" was not used";
};
```

As long as you declare a local variable within a function then you can call it what you like. As soon as it goes out of scope, any previous type and value are restored

Even though the `var` keyword is not required for defining global variables, it is best practice to do so. And it is recommended that you define your variables at the top of your script, including enough comments to give the reader an understanding of what they will be used for. This approach not only improves the legibility of your code, it also forces you to make conscious decisions about variable naming and scope. [7]

7. Function naming works a little differently. Programming languages like Java identify a function by the combination of its name plus the number and types of parameters. JavaScript just uses the name. So if you have multiple definitions of the same function, JavaScript will only ″remember″ the last one — regardless of whether that definition has a different number of parameters.

Accessing your own Java classes

You can access your own Java classes from inside the IBM Tivoli Directory Integrator framework as long as the these are *public* classes and methods. These libraries must be packaged into a .jar or .zip file, and then be placed in the IBM Tivoli Directory Integrator jars directory, preferably in your own subdirectory. You can also use the CLASSPATH environment variable or the Java runtime environment extension folder, but both these methods are discouraged. These methods let you call IBM Tivoli Directory Integrator classes from within your own classes only if the loader happens to load the IBM Tivoli Directory Integrator classes before your own.

If you are running the server from the Config Editor, you must restart the Config Editor before it knows about new classes in the jars directory and subdirectories.

After putting the jar files in the jars subdirectory, you can create an instance of the class to refer to within the IBM Tivoli Directory Integrator. Note that the Java FC allows you to open jar files, browse objects contained in them as well as their methods. Once you have chosen the function to call, the FC prepares your Input and Output schema to match the parameters needed by the Java function.

Instantiating the classes using the Config Editor

Use the Java library folder of the Config Browser to declare your classes. This works only if your class has a no-argument constructor (usually but not always the default constructor).

When adding a class object simply click **Add** and specify two parameters: the script object name (the name of the scripting variable that is an instance of your Java class), and the Java class name. For example, you can have a Script Object Name **mycls** while the Java Class might be **my.java.classname**. The **mycls** object will available for any AssemblyLines, defined before the Global Prologs execute. For more information on the Config Browser, see Config Browser

Note:

Note that this causes your object to be instantiated for every AssemblyLine run. If this is not desirable, and if you prefer to instantiate on demand, then see the next section.

Runtime instantiation of the classes

If you want to instantiate your class at a specific point of execution or for the classes without no-argument constructors, you need to instantiate the class during run time. For example:

```
crytoLib = new com.acme.myCryptoLib();
```

Scripting in JavaScript

Instantiating a Java class

Assuming you want to use the standard java.io.FileReader, use the following script:

```
var javafile = new java.io.FileReader( "myfile" );
```

The same technique is used to instantiate your own objects:

```
var myfile = new my.FileReader("myfile");
```

Using binary values in scripting

Binary values can be retrieved from Attributes by using the Entry's getObject() function. The binary Attribute value itself is returned as a byte array. Here is a JavaScript example:

```
var x = conn.getObject("objectGUID");
for ( i = 0; i < x.length; i++ )
{
   task.logmsg ("GUID[" + i + "]: " + x[i]);
}
```

This example writes some numbers varying between -128 and 127 into the log file. You might want to do something else with your data. If you have read a password from a Connector, which stored it as a ByteArray, you can convert it to a string with this code:

```
password = system.arrayToString(conn.getObject("userpassword"));
```

Using date values in scripting

When we talk about using dates, we are referring to instances of java.util.Date. With any of the available scripting languages, you can implement your own mechanism for handling dates; however, this is not a common practice.

The IBM Tivoli Directory Integrator scripting engine provides you with a mechanism for parsing dates. The **system** object has a **parseDate(date, format)** method accessible at any time.

Note: When you get an instance of java.util.Date, you can use the standard Java libraries and classes to extend your processing.

Here is a simple JavaScript example that handles dates. This code can be placed and started from any scripting control point:

```
var string_date1 = "07.09.1978";
var date1 = system.parseDate(string_date1, "dd.MM.yyyy");

var string_date2 = "1977.02.01";
var date2 = system.parseDate(string_date2, "yyyy.dd.MM");

task.logmsg(date1 + " is before " + date2 + ": " +
  date1.before(date2));
```

The script code first parses two date values (in different formats) into java.util.Date. It then uses the standard java.util.Date **.before()** method to determine whether the first date instance comes before the second one. The output of this script is then printed to the log file.

Using floating point values in scripting

The following examples demonstrate how floating point values can be used within the scripting code you create. All of these examples are implemented in JavaScript. While the same examples might be repeated using several other scripting languages, the syntax might be different. The following simple script assigns floating point values to two variables in order to find their average. This code can be started from any scripting control point. The log file output is " r = 3.85 ".

```
var a = 5.5;
var b = 2.2;
var r = (a + b) / 2;
task.logmsg("r = " + r);
```

The next example extends this simple script. Consider that in your input Connector is a multiple values attribute called "**Marks**" containing string values (java.lang.String) representing floating point values (a common situation). This attribute is mapped to an attribute in your output Connector called

"**AverageMark**", which holds the average value of the "**Marks**" attribute's values. The following code is used in the Advanced Mapping of the "**AverageMark**" attribute:

```
// First return the values of the "Marks" attribute
var values = work.getAttribute("Marks").getValues();

// Zero out counter and sum variables
var sum = 0;
var count = 0;

// Loop through the values, counting and summing them
for (i=0; i<values.length; i++)
{
   // use the Double() function to convert value to number
   sum = sum + new Number(java.lang.Double(values[i]));
   count++;
}

// If count > 0, compute the average
var average = (count > 0) ? (sum / count) : 0;

// Return the computed average
ret.value = average;
```

The central call in this example is the `java.lang.Double(values[i])` used to convert the currently indexed value of "**Marks**" into a numeric value that can then be used in the average computation.

Examples

Go to the examples/scripting subdirectory of your IBM Tivoli Directory Integrator installation.

Hooks

Those IBM Tivoli Directory Integrator objects that provide built-in workflows, such as AssemblyLines, Connectors and Functions, also offer Hooks throughout these flows that enable you to extend, change or completely override the automated behavior. Hooks enable you to customize the behavior of your integration solution to an arbitrary depth, including behaviors such as:

1. Override the basic data access for a Connector (see "Override Hooks" on page 70).
2. Respond to errors received from data sources. For example, an Iterator Connector has an **Iterator Error** Hook that can be programmed to log or send mail about input errors.
3. Modify the flow of an AssemblyLine (such as skipping a Connector, or restarting the cycle).

Hooks are usually called in the AssemblyLine process as part of the flow control. If for some reason you want to call a Hook, use the following JavaScript command:

```
myConnector.trigger("hookName");
```

where *myConnector* is the name you gave your Connector and *hookName* is the internal name of the Hook. The internal name of the Hook is not to be confused with the name of the Hook as seen in the Config Editor.

There is a special variable called *thisConnector* that is available in Connector Hooks, and which is always a reference to the current Connector. This facilitates the use of generic scripts, because you do not need to know the name of the Connector.

Enabling or disabling Hooks

By default, all Hooks are disabled. As soon as you add some code to a particular Hook, the Config Editor marks that Hook as enabled. If you do not want a Hook

to be used, you can clear the **Enabled** flag for that Hook. This enables you to keep the code in your Hook in case you later want to use the code again.

A Hook is enabled if it is marked as enabled, even if it contains no code. This can be important in error Hooks, where IBM Tivoli Directory Integrator can treat the error as handled based on the mere presence of an enabled Hook. Enabled (and not inherited) Hooks containing code have their name in **black boldface**.

Hooks can be inherited from the parent component that they appear in, and in this case the Hook will be listed in *blue cursive text*.

Note that if you edit an inherited Hook, the script code becomes a local copy that overrides the inherited logic. To restore inheritance you need to select the Hook and click **Delete Hook** button at the top of the **Hooks** list.

Override Hooks

For every Connector mode there is a mode-specific **Override** Hook. If this Hook is enabled, first the **Before Execute** Hook is called (if it is enabled), and then the **Override** *mode-specific operation* Hook, before the flow control continues with the next Connector in the AssemblyLine. No other Hooks are called, except **Error** Hooks if necessary.

For a Connector in Update mode, there are also **Override Add** and **Override Modify** Hooks. After the Connector has decided whether it performs a Modify or Add operation (based on whether there already exists a matching Entry in the data source), the appropriate Hook is called if it is enabled. After the Hook has finished, the flow control continues to the **After Update** and **Update Successful** Hooks.

Error Hooks

Error Hooks enable you to respond to errors generated by data sources, the operating system and the IBM Tivoli Directory Integrator. For example, an Iterator has an **Iterator Error** Hook that is invoked if an error occurs during Iterator operation (for example, reading the next Entry). Refer to ″TDI Hook Flow diagrams″ in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* to see when (and which) error Hooks are called.

In an **Error** Hook, you typically want to check the **error** Entry object to see what caused the process to pass through the **Error** Hook. The **error** object is an object of type **Entry** (just like **work** and **conn**) and it contains a number of attributes that describe the error. These include the following error attributes:

status The error status (for example, **fail**).

connectorname
The name of the Connector where the error occurred.

operation
The internal name of the operation that was being performed.

exception
The full exception text.

message
The error message.

class The type of error (exception), which often gives you a clue as to which system originated the error.

When a failure occurs in a Connector, the following events occur:

1. The error counter is increased by one.
2. The **On Error** Hook specific to the mode of the Connector is called (***Connector_mode* Error**).
3. If that Hook is not enabled, the Connector's **Default On Error** Hook is called.
4. If no Hook is enabled to catch the error, the AssemblyLine is terminated and the error is displayed in the log.
5. If the error Hook is enabled, then AssemblyLine continues with the next Connector. If you want another behavior, you must program it in the Hook with one of the methods described in "The AssemblyLine" on page 6.

List of Hooks

Connectors

These Hooks are common for all Connectors. See "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* to see the when Hooks are invoked.

Prolog - Before Initialize (`before_initialize`)
Called before initialization of the Connector is attempted.

Prolog - After Initialize (`after_initialize`)
Called after the Connector is initialized.

Prolog - On Error (`initialize_fail`)
Called when the Connector initialization fails. If no Hook is enabled to catch the error, the AssemblyLine aborts. If a Hook is enabled, it must take the proper action, which might be to abort the AssemblyLine.

Prolog - On Connection Error (`connect_init`)
Called when the Connector initialization fails and before auto-reconnect functionality engages (if this feature is enabled for initialization).

Before Execute (`before_execute`)
Called before every Connector start.

Default On Success (`default_ok`)
Called when a Connector operation succeeds, after mode-specific success Hooks.

Default On Error (`default_fail`)
Called when an error occurs during a Connector operation, unless the mode-specific fail Hook is enabled, in which case the mode-specific Hook is invoked first.

Data Flow On Connection Error(`on_connection_failure`)
Called when a Connector operation fails and before auto-reconnect functionality engages (if this feature is enabled for DataFlow operation).

Epilog - Before Close (`before_close`)
Called before the Connector is closed.

Epilog - After Close (`after_close`)
Called after the Connector is closed.

Epilog - On Error (`close_fail`)
Called when the Connector fails to close after the AssemblyLine has finished.

These Hooks are available for Connectors in **Iterator** mode:

Prolog - Before Selection (`before_selectEntries`)
Called before the Connector selects entries as part of the initialization.

Prolog - After Selection (`after_selectEntries`)
Called after the Connector has selected entries in the initialization.

Override GetNext (`override_getnext`)
Enables you to override the mode-specific operation and Hook flow.

Iterator - Before GetNext (`before_getnext`)
Called before the Connector attempts to get the next item.

Iterator - After GetNext (`after_getnext`)
Called after a GetNext is successfully performed on the Connector, but before Attribute Mapping is done. The **conn** Entry object is available for inspecting or changing the attributes retrieved from the Connector.

GetNext Successful (`get_ok`)
Called before **Default Success**, this is the mode-specific success Hook for Iterator mode.

Iterator Error (`get_fail`)
Called before **Default On Error**, this is the mode-specific error Hook for Iterator mode.

End-of-data
Called when the Connector reaches the end of the data it is iterating through.

These Hooks are available for Connectors in **AddOnly** mode:

Note: TDI no longer throws an ignoreEntryException when the attribute map is explicitly empty, as it used to do. Instead, flow goes to the Before Add Hook where Attributes can be added to **conn** by way of script. If **conn** is still empty after the **Before Add** Hook, then the new **On No Add** Hook (internal name: abandon_add) is called. At the same time, TDI logs a warning about an explicitly empty map (again, only if **conn** remains empty). The **On No Add** Hook is available only in **Update Mode**.

Override Add (`override_add`)
Enables you to override the mode-specific operation and Hook flow.

AddOnly - Before Add (`before_add`)
Called before an add operation is attempted.

Note: This Hook is shared between AddOnly and Update modes.

AddOnly - After Add (`after_add`)
Called after an entry is successfully added.

Note: This Hook is shared between AddOnly and Update modes.

Add Successful (`add_ok`)
Called before **Default Success**, this is the mode-specific success Hook for Add mode.

AddOnly Error (`add_fail`)
Called before **Default On Error**, this is the mode-specific error Hook for Add mode.

These Hooks are available for Connectors in **Delete** mode:

Override Delete (override_delete)
Enables you to override the mode-specific operation and Hook flow.

Delete - Before Lookup (before_lookup)
Called before initial lookup is attempted.

Note: This Hook is common between all the modes that perform a lookup: Lookup, Delete and Update.

Delete - On No Match (delete_nomatch)
When no entry matches the Link Criteria, the Connector calls this Hook if it is enabled, otherwise the **On Error** Hooks are called.

Delete - After Lookup (after_lookup)
Called after an entry is found, but before Attribute Mapping is done. The **conn** Entry object is available for inspecting or changing the attributes retrieved from the Connector.

Note: This Hook is shared between Lookup, Delete and Update modes.

Delete - On Multiple Entries (delete_multiple)
When more than one entry matches the Link Criteria, the Connector calls this Hook if it is enabled, otherwise the **On Error** Hooks are called. See "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for a description on how to select a particular entry in order to continue with the operation.

Delete - Before Delete (before_delete)
Called before delete is attempted. At this point, the **conn** Entry object is available for inspecting or changing the attributes retrieved from the Connector by the initial lookup. That is why Delete mode is in **Input Map**.

Delete - After Delete (after_delete)
Called after an entry was deleted.

Delete Successful (delete_ok)
Called before **Default Success**, this is the mode-specific success Hook for Delete mode.

Delete Error (delete_fail)
Called before **Default On Error**, this the mode-specific error Hook for Delete mode.

These Hooks are available for Connectors in **Lookup** mode:

Override Lookup (override_lookup)
Enables you to override the mode-specific operation and Hook flow.

Lookup - Before Lookup (before_lookup)
Called before lookup is attempted.

Note: This Hook is common between all the modes that perform a lookup: Lookup, Delete and Update.

Lookup - On No Match (lookup_nomatch)
When no entry matches the Link Criteria, the Connector calls this Hook if it is enabled, otherwise the **On Error** Hook is called.

Lookup - On Multiple Entries (lookup_multiple)
When more than one entry matches the Link Criteria, the Connector calls this Hook if it is enabled, otherwise the **On Error** Hook are called. See "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory*

Integrator 6.1.1: Reference Guide for a description on how to select a particular entry in order to continue with the operation.

Lookup - After Lookup (`after_lookup`)
Called after an entry is found, but before Attribute Mapping is done. The **conn** Entry object is available for inspecting or changing the attributes retrieved from the Connector.

Note: This Hook is shared between Lookup, Delete and Update modes.

Lookup Successful (`lookup_ok`)
Called before **Default Success**, this is the mode-specific success Hook for Lookup mode.

Lookup Error (`lookup_fail`)
Called before **Default On Error**, this is the mode-specific error Hook for Lookup mode.

These Hooks are available for Connectors in **Update** mode:

Before Update (`before_update`)
Called before the initial lookup is attempted (in order to determine whether to perform an add or modify operation).

Update - Before Lookup (`before_lookup`)
Called before lookup is attempted.

Note: This Hook is common between all the modes that perform a lookup: Lookup, Delete and Update.

Update - On Multiple Entries (`update_multiple`)
When more than one entry matches the Link Criteria, the Connector calls this Hook if it is enabled, otherwise the **On Error** Hooks are called. See "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for a description on how to select a particular entry in order to continue with the operation.

Update - After Lookup (`after_lookup`)
Called after an entry is found, but before Attribute Mapping is done. The **conn** Entry object is available for inspecting or changing the attributes retrieved from the Connector.

Note: This Hook is shared between Lookup, Delete and Update modes.

Update - On Modify - Override Modify (`override_modify`)
Enables you to override the modify operation and Hook flow.

Update - On Modify - Before Modify (`before_modify`)
Called before a modify operation is attempted.

Update - On Modify - On Compute Changes - On No Changes (`modify_nochange`)
Called when an Update-mode Connector (with the Compute Changes flag set) reports no changes to update.

Update - On Modify - On Compute Changes - Before Applying Changes (`modify_apply`)
Called immediately before a modification is performed. This Hook is called only when the Compute Changes flag is set. If no changes are detected, the **On No Changes** Hook is called instead.

Update - On Modify - After Modify (`after_modify`)
Called after an entry was modified.

Update - On Add - Override Add (`override_add`)
Enables you to override the add operation and Hook flow.

Note: This Hook is shared between Update and AddOnly modes.

Update - On Add - Before Add (`before_add`)
Called before an add operation is attempted.

Note: This Hook is shared between Update and AddOnly modes.

Update - On Add - After Add (`after_add`)
Called after an entry was successfully added.

Note: This Hook is shared between Update and AddOnly modes.

Update - On Add - On No Add
Called if the conn Entry is empty after the Output Map has completed. After this Hook is called, nothing is added, and the Connector logs an ignore.

Update - On Add - After Update (`after_update`)
Called after successful update (add or modify).

Update - Update Successful (`update_ok`)
Called before **Default Success**, this is the mode-specific success Hook for Update mode.

Update Error (`update_fail`)
called before **Default On Error**, this is the mode-specific error Hook for Update mode.

These Hooks are available for Connectors in CallReply mode:

Override CallReply (`override_callreply`)
Enables you to override the mode-specific operation and Hook flow.

CallReply - Before CallReply (`before_call`)
Called before the service or system call is made.

CallReply - After CallReply (`after_reply`)
Called after a reply is received from the called service or system.

CallReply Succesfull (`callreply_ok`)
Called before **Default Success**, this is the mode-specific success Hook for CallReply mode.

CallReply Error (`callreply_fail`)
Called before **Default On Error**, this is the mode-specific error Hook for CallReply mode.

No Answer Returned
Invoked if the function call receives no reply in return (based on timeout settings) after the call is made. Note that this is a mandatory Hook for CallReplymode.

Connectors in Server mode have five sets of Hooks: Server, as well as two **Data Flow** sets (Iterator and Reply), plus Prologs and Epilogs. Prologs and Epilogs are standard for all modes, although they behave somewhat differently here. That

leaves Server, **Data Flow** (Iterator) and **Data Flow** (Reply), which represent the three different tasks carried out by this mode:

1. The Connector first acts as a ″Server″, listening on some resource (like an IP port) for incoming client connections. When a connection arrives and is accepted, the Server mode Connector clones itself in Iterator mode, attaches itself to the Flow section[8] and feeds it the stream of data coming from the client. In other words, the Server mode Connector starts iterating on the client data and driving AL cycles -- just like a typical AssemblyLine works.

   Server functionality is tied to the Server Hooks.
2. The Iterator cycles on the client data. Each time the Flow section completes, the Iterator switches momentarily to Server Response.

   The Prolog, **Data Flow** (Iterator) and Epilog Hooks are active when the Connector is in Iterator mode.
3. Server Response sends a reply to the client, switches back to Iterator and resumes cycling on client data.

   Note that the Iterator Hooks are described under that mode.

Before Accepting connection (`before_getnextclient`)
Called before the Connector goes into listening mode.

After Accepting connection (`after_getnextclient`)
Once a connection is received, this Hook is invoked. Note that the no data is available at this time. In order to examine incoming event information, use the Iterator Hooks like **After GetNext** or **GetNext Successful**.

Error on Accepting connection (`getnextclient_fail`)
Executed if an error occurs in any of the Server mode Hooks, or received from the data source during event listening.

As mentioned above, the Reply Hooks apply to Server Response behavior:

Before Execute
Called before any Server Response behavior begins. Note that this is not the same as the Before Execute Hook in Iterator mode.

Override Reply
Enables you to implement your own **Response** logic.

Before Reply
Called before the Output Map and the actual call to the Connector Interface's `replyEntry()` method.

After Reply
Executed after the response is made.

Reply Success
Iinvoked if the Server Response behavior (reply to the client) has completed without unhandled errors.

Reply Error
Called if an error occurs during Server Response behavior (reply to the client), or in any of the other Reply Hooks.

8. You can define a pool of Flow sections that you want hot-and-ready to process incoming client data. This is done in the Config tab of the AssemblyLine.

Function Components

These Hooks are common for all Function Components. The tables below are here to let you know the internal name of Hooks (if you want to use the trigger() method).

Prolog - Before Initialize (`prologinit`)
Executes code just before the Function is initialized.

Prolog - After Initialize
Executed after the Function initializes.

Prolog - On Error
Flow ends up in this Hook if an error occurs during the Function initialization phase.

Prolog - On Connection Error
As with Connectors, this Hook is called if an exception matching the rules that define connection-type errors.

Dataflow - Before Execute
Called during each AL cycle before any other action is taken by this component.

Epilog -Before close
Called just before the Function closes down.

Epilog - After close
Executes code just after the Function has closed.

Epilog - On Error
Called if the Function encounters an error while closing.

AssemblyLines

These Hooks are common for all AssemblyLines, and are found in the **Hooks** tab of each AssemblyLine. They are all executed only once per AssemblyLine run, unless you happen to start your AssemblyLine multiple times (for example, by using an EventHandler).

Prolog - Before Init. (`prologinit`)
Runs the Prolog before Connectors initialize, so it is a convenient spot for modifying the Connector parameters. For example, if at runtime you know the filename to use in a File Connector, you can pass this to the Connector with the following script:

```
myConnector.connector.setParam("filePath",myFileNameVariable);
```

Prolog (`prolog`)
Code started after all Connectors have initialized successfully. This is a good place to declare global variables you need through the execution of the AssemblyLine (for example, for counters, average values, and so forth).

On Start of Cycle
Invoked at the start of each AL cycle, the first time after all the Prolog Hooks (and Connector initialization) have completed.

Note: If you get here not for the first time but after the AssemblyLine has at least executed one cycle, the *work* entry is not yet reset at this point; it will still reflect the state in which it was at the end of the previous cycle.

Dataflow

Note: This is not a Hook.
Here the AssemblyLine drives its Connectors and Script Components. See "AssemblyLine and Connector mode flowcharts" and "Script Connector" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Epilog (`epilog`)
Code is executed once for each run of the AssemblyLine before all Connectors perform their close Hooks and operations.

Epilog - After close (`epilog2`)
Code is executed once for each run of the AssemblyLine after all Connectors have finished their tasks. A common pattern is to release all resources allocated in the **Prolog** section here and save.

On Success
Called if the AL completes without errors/exceptions (or at least, with any errors that have not been dealt with by your solution).

On Failure
Called if the AL stops because of an error/exception. Note that this will not happen if a Connector Error Hook catches the exception.

Shutdown Request (`shutdown`)
This Hook is started when the AssemblyLine is instructed to stop (for example, from the AMC Console, or from another server). This script enables you to add clean-up code for terminating the AssemblyLine gracefully.

Note: If the AssemblyLine gets a shutdown request and this script is empty, then this results in an error (although this error can be handled in the Epilogs).

Server Hooks

Server Hooks allow you to write JavaScript code to respond to events and errors that occur at the server level. Unlike AssemblyLine and component Hooks, Server Hooks are stored in separate script files. These files are kept in the *serverhooks* folder in the current solution directory and must contain specifically named script functions. The TDI server and configuration instances provide a method for TDI components to invoke custom Server-level Hooks. A Server Hook is a function name that is defined in a script file. Function implementations are provided by simply dropping script files in the "serverhooks" directory of the solution directory.

In addition to these Hooks being called by the Server when specific events occur, they can also be invoked from your scripts. Calls to these Hooks are synchronized to avoid potential multi-threading issues.

Upon startup, IBM Tivoli Directory Integrator loads and executes all user scripts in the *serverhooks* subdirectory. Scripts may or may not contain function declarations. A script that has no function declarations is executed once at startup before any configuration instances are started. Code that defines standard TDI Server Hook functions are prefixed with "TDI_"., and these are executed at various points during operation.

All TDI Server Hook functions have the following JavaScript signature:

```
/**
   * @param main The configuration instance invoking the function
  * @param source The component invoking the function
```

```
 * @param user Arbitrary parameter information from the source
 */
function TDI_functionName(main, source, user) {
}
```

The “main” and “source” parameters always provide access to the Config Instance and calling component, respectively. The “user” parameter is used for different purposes in the various Hook functions.

The following standard function names are invoked by various TDI components:

Function Name	**Called by (source)**	**User Parameter and Expected Value**
TDI_ALStarted	Config Instance	Called when an AssemblyLine is started. user = The AssemblyLine that started *return value ignored*
TDI_ALStopped	Config Instance	Called when an AssemblyLine stopped. user = The AssemblyLine that stopped *return value ignored*
TDI_ConfigStarted	Server	Called when a Config instance started. user = The configuration instance *return value ignored*
TDI_ConfigStopped	Server	Called after a Config instanced stopped. user = The configuration instance *return value ignored*
TDI_Shutdown	Server/Config Instance	Called immediately before the TDI server is terminating the JVM (for example, System.exit()). user = Exit status (integer) *return value ignored*

Access to TDI Server Hook functions is provided through the `main.invokeServerHook()` method. This function is synchronized to prevent more than one thread executing a Hook at a time. All calls are invoked synchronously so the caller will wait for the function to return. As a result, care should be taken not to spend too much time in a server Hook.

As mentioned previously, scripts are defined and made available by creating files in the “serverhooks” subdirectory of the solution directory. Scripts that contain sensitive information should be encrypted with the Server-API before adding it to the directory. The serverapi/cryptoutils tool is available for encrypting script files.

Note that TDI automatically tries to decrypt files with extension .jse, hence encrypted files should preferably have that extension.

Furthermore, the files in the serverhooks directory are loaded and executed after first sorting the file names using case-sensitive sort with the standard collating sequence for the platform. All files in the top-level directory are loaded before files in any subdirectories are processed.

Some examples Server Hook use are:

- A custom object that you always want loaded in TDI for use for your own scripting could be instantiated from a JavaScript snippet hooked into the server Hook on TDI startup. This gives you more control than simply referring to the class under the Java Libraries folder in the Config Browser.
- One or more custom ALs that you start creates an audit log for these events or propagates these events to other systems using some transport (SNMP, HTTP, JMS, and so forth).
- A corporate security policy you implement that is invoked every time a Config is loaded or AL started.

Calling Server Hooks from script

The `com.ibm.di.server.RS` class (script variable "main") has a method for invoking Server Hooks:

```
/**
 * Invokes a server hook.
 *
 * @param name The name of the hook (also the filename)
 * @param caller The object invoking the hook
 * @param userInfo Arbitrary information to the hook from the caller
 */

public Object invokeServerHook(
 String name,
 Object caller,
 Object userInfo) throws Exception;
```

This call can return a Java Object (any type), so even though TDI ignores this during Server Hook execution, you can make use of returned values in your own scripted calls.

Deltas

Attention: The Delta Engine introduces a new local repository for storing "snapshots" of data in order to compute changes during the synchronization process. The data source that is being scanned for changes becomes the master in a master-slave relationship, and it is then vital that all changes made to the slave (for example, the Delta store) be made using the Delta mechanism, and not by directly manipulating the underlying database table. Otherwise, the Delta snapshot information that IBM Tivoli Directory Integrator maintains becomes inconsistent, and the Delta Engine fails.

Whenever possible, enable the **Compute Changes** functionality (in the Update mode of the Connector) instead of the Delta feature. This feature also limits writes made to the data source.

Still, there might be times when your solution calls for reacting to changes in real time. However, IBM Tivoli Directory Integratorcannot detect changes in some data sources. Often this because there is no support (or published interface) for change detection in the target system or source (for example, a flat file). Or it might be that there is no Change Connector available for this system. This is where the

Delta feature can be used so that IBM Tivoli Directory Integratorcan determine which Entries are new, modified or deleted.

Note: You can configure Delta settings on Connectors in Iterator mode only.

The Delta mechanism knows whether Entries or Attributes have been added, changed or deleted by keeping a local copy of each entry in a persistent store called a **Delta Table**, which is part of the System Store. Each time the AssemblyLine is run, the Delta mechanism compares the data being iterated with its copy in the Delta Table. You can configure the Iterator's Delta settings to control which types of changes you want passed into the AssemblyLine. All other changes are ignored.

Unique attribute

In order for the Delta mechanism to be able to uniquely identify each entry, you must specify a unique attribute to use as the Delta key. This is done in the Connector's **Delta** tab by entering (or selecting) an attribute name in the **Unique Attribute Name** parameter. This attribute must be found in the Input Map of your Iterator, and can either be an attribute read from the connected system or a computed attribute (Advanced Attribute Map). Each attribute must have a value that can be represented as a string (that is, it must implement the `toString()` function), and the resulting string value cannot be null or blank. If a designated unique attribute has more than one value, then an error is thrown.

You can also specify multiple attributes by separating them with a plus sign (**+**):

```
LastName+FirstName+BirthDate
```

At least one of the attributes specified as a unique attribute must contain a value. When several attributes are specified, their string values are concatenated into one string, which then becomes the unique Delta identifier. Attributes with no values (for example, **null**) are skipped when the Delta key is built for an entry.

Delta flags

The Delta Engine feature (available for any Connector in Iterator mode) enables you to control how entries are returned in the AssemblyLine. If an entry is not returned as a result of one of the flag settings, the Delta Engine silently skips the entry and continues with the next entry from the Connector. A skipped entry is always updated in the Delta Store. See "Delta Store" on page 87.

Return Unchanged
If flag is TRUE then unchanged entries are returned to the AssemblyLine.

Read Deleted
If this flag is TRUE then all the deleted entries from the delta db are returned to the AssemblyLine. Note that delete-tagged Entries are not removed from the Delta snapshot db unless you also enable the Remove Deleted flag.

Remove Deleted
If select, then deleted entries are removed from the AssemblyLine after they are read.

Return Unchanged
If flag is TRUE, then unchanged entries are returned to the AssemblyLine.

Currently, only the LDAP Connector can handle incremental modifications automatically, using its Delta mode. However, you can add this functionality to

your solution with other Connectors by examining the operation codes listed above and driving the modify operation from script.

Deltas and compute changes

Computing changes from a data source is a straightforward process. When comparing two entries (the entry read from the data source and the Delta copy), the returned entry has an operation code set to **add**, **modify** or **delete**. You can then control the behavior of your AssemblyLine Connectors based on this operation code.

For example, suppose you want your AssemblyLine to handle all three types of operations. One way to build this solution is to use two Connectors configured to work with the same target system. Set one to **Delete** mode and the other to **Update**. In the **Before Execute** Hook of the Update Connector, add the following code:

```
if (work.getOperation() ==  "delete")
   system.ignoreEntry();
```

This causes the Update Connector to skip over all **work** Entries marked for **delete** by the Delta feature. In the **Before Execute** Hook of the Delete Connector you put the reciprocal code:

```
if (!work.getOperation() ==  "delete")
   system.ignoreEntry();
```

Now the Delete Connector ignores all **work** Entries not tagged for **delete**.

Another way to implement this same functionality is to have a single Connector in **Update** or **Delete** mode. The actual mode of the Connector is not important, because you use script to drive it. The Connector must be set to **Passive** state, so it is initialized, but not operated by the AssemblyLine. Then you add a **Script Component** to your AssemblyLine with the following script (this example assumes your Connector is called `myTargetConnector`):

```
if (work.getOperation() ==  "delete")
   myTargetConnector.deleteEntry(work)
else
   myTargetConnector.update(work);
```

Both methods perform the same actions, including invoking all the relevant Connector Hooks.

Computed Changes

This is a additional change-control feature available for Connectors in Update mode (using a check box in the Connector Detail pane). **Compute Changes** is used to make the Connector check that the existing record in the connected system is actually different from the one about to be written before actually performing the modify operation. Description of how this option is used can be found in “Connectors” on page 24, in **Update mode**.

Examples

Go to the *root_directory*/examples/deltas directory of your IBM Tivoli Directory Integrator installation.

Delta process

The process of computing changes in a data source involves the use of a local store in which the Delta process stores information about each entry read from a

Connector. Each time the delta process is run, a sequence counter increments. Each entry read from the Connector is stored in a Delta table along with the current sequence number. The purpose of the sequence number is to be able to detect entries no longer part of the source data set. This is accomplished by comparing the sequence numbers after a completed iteration over the source data set. For more information about Delta storage, see "Delta Store" on page 87.

Delete Entry

After a completed iteration of the data source, any entries in the Delta Table with a sequence number lower than the current sequence number is considered to be a deleted entry. This is true only if the iteration completes successfully.

Modify Entry

When an entry is read from the Connector, the delta process looks up its corresponding entry in the Delta Table using the unique attribute's value. If a match is found, the Delta process compares each attribute (and each attribute's values) to determine if there have been modifications to the entry.

Add Entry

If a match is not found in the Delta Table, the entry is added to the Delta Table and treated as a new entry.

Unchanged Entry

If an entry from the Connector matches an entry in the Delta Table, the entry is treated as an unchanged entry.

Delta Table structures

The tables used by the Delta store consist of the Delta Systable (DS) that contains information about each Delta Table currently in use in the delta store. The Delta Table (DT) contains information about each entry that is read and processed by the Delta function in a Connector.

Delta Systable

The Delta Systable (DS) contains information about each Delta Table (DT) in the System Store (CloudScape database). The purpose of the DS is to maintain the sequence counter for each DT. The structure for the DS is as follows:

Column	Type	Description
ID	Varchar	The DT identifier
SequenceID	Int	The sequence ID from the last run
Version	Int	The DS version (1)

Delta Table

Each Connector that requests a Delta store needs to specify a unique Delta identifier to be associated with the Connector. This identifier is also used as the name of the Delta Table in the System Store. The Delta Table structure is as follows:

Column	Type	Description
ID	Varchar	The unique value identifying an entry
SequenceID	Int	The sequence number for the entry

Version	JavaObject	The serialized Entry object

System Store

The System Store addresses the various needs of IBM Tivoli Directory Integrator for persistent storage and by default uses the DB2Java (CloudScape) RDBMS as its underlying storage technology. Other relational databases, like IBM DB2, can be used to hold the System Store. The System Store can be shared by multiple instances of IBM Tivoli Directory Integrator servers if the CloudScape database runs in networked mode, or if a multi-user relational database system is used. If CloudScape runs embedded in an IBM Tivoli Directory Integrator server, it cannot be shared simultaneously with other servers.

Note:
The database bundled with TDI, and used by default to provide System Store functionality, is upgraded to the latest version, Derby version 10.1. As a result, the existing Cloudscape™ database must be converted to the newest db level. This is handled by the installer for the standard "Cloudscape" directory (default System Store setup), which results in a new "TDISysStore" directory/database after completed conversion. The conversion utility – which is found under the "tools\CSMigration" folder of the installation directory – can also be run manually for any other databases that you need to convert. To migrate, use the following command:

```
migrateCS <oldCSdirectoryDB> <newCSdirectoryDB>
```

Note that the new Cloudscape directory/db must be different from the old one.

The system store implements three types of persistent stores for IBM Tivoli Directory Integrator components:

- The User Property Store
- The Delta Tables
- The Checkpoint/Restart Tables

Each store offers its own set of features and built-in behavior, as well as a callable interface that users can access from their scripts, for example, to persist their own data and state information.

Under the **Store** menu is a choice called **Manage System Store** or **View System Store**. Selecting this opens the **System Store**. Double-click **System Store** . Here you can change the name of the directory where CloudScape maintains the system database for the System Store.

If you click **Open** , the System Store Browser displays three items, one for each type of persistent store. You can use this window to examine the contents of the System Store. Although you cannot change values directly from this screen, you can delete tables with the **Delete Table** button. You can also set up a JDBC Connector to access any of these tables, although changing data in these System Store tables must be avoided, as this can cause your solution to malfunction.

Attention: If you are running CloudScape embedded in IBM Tivoli Directory Integrator as opposed to running it in networked mode as a server, then be sure to **Close** the database again before trying to test or run your Config. Because the Config Editor starts up a separate instance of the IBM Tivoli Directory Integrator Server, running in its own JVM, the System Store is not available to this Server. Closing the System Store Details window also closes your connection to the database.

Note: Although the Sandbox feature also uses the System Store technology, you specify a new database directory for each AssemblyLine.

Configure RDBMS database servers as System Store

You can configure Oracle, MS SQL Server and DB2 servers as System Store databases. The following sections contain the necessary configuration settings:

Oracle

JDBC connection parameters:

Note: Place `ojdbc14.jar` in the `<TDI_HOME>/jars` subdirectory.

```
com.ibm.di.store.database=jdbc:oracle:thin:@itdidev.in.ibm.com:1521:itimdb
com.ibm.di.store.jdbc.driver=oracle.jdbc.OracleDriver
com.ibm.di.store.jdbc.urlprefix=jdbc:oracle:thin:
com.ibm.di.store.jdbc.user=SYSTEM
{protect}-com.ibm.di.store.jdbc.password=password
```

Create table statements:

```
com.ibm.di.store.create.delta.systable=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH)
 NOT NULL, SEQUENCEID int, VERSION int);ALTER TABLE {0} ADD CONSTRAINT IDI_CS_{UNIQUE} PRIMARY KEY (ID)
com.ibm.di.store.create.delta.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, SEQUENCEID int, ENTRY BLOB );
 ALTER TABLE {0} ADD CONSTRAINT IDI_DS_{UNIQUE} Primary Key (ID)
com.ibm.di.store.create.property.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ENTRY BLOB );
 ALTER TABLE {0} ADD CONSTRAINT IDI_PS_{UNIQUE} Primary Key (ID)
com.ibm.di.store.create.checkpoint.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ALSTATE BLOB, ENTRY BLOB,
 TCB BLOB );ALTER TABLE {0} ADD CONSTRAINT IDI_CR_{UNIQUE} Primary Key (ID)
com.ibm.di.store.create.sandbox.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ENTRY BLOB )
com.ibm.di.store.create.recal.conops=CREATE TABLE {0} (METHOD varchar(VARCHAR_LENGTH), RESULT BLOB, ERROR BLOB)
```

MS SQL Server

JDBC connection parameters:

Note: The DATA TYPE for MS SQL is IMAGE.

```
com.ibm.di.store.database=jdbc:Microsoft:sqlserver://localhost:1433;DatabaseName=master;selectMethod=cursor;
com.ibm.di.store.jdbc.driver=com.microsoft.jdbc.sqlserver.SQLServerDriver
com.ibm.di.store.jdbc.user=sa
com.ibm.di.store.jdbc.password=passw0rd
```

The above connection parameters are used with Microsoft JDBC jars:

- `Msutil.jar`
- `MsBase.jar`
- `MSsqlserver.jar`

You must download these jar files and copy them into the `<TDI_HOME>/jars` directory.

JDBC connection parameters (For JSQLConnect driver):

```
com.ibm.di.store.database= jdbc:JSQLConnect://itdiderver/database=reqpro
com.ibm.di.store.jdbc.driver= com.jnetdirect.jsql.JSQLDriver
com.ibm.di.store.jdbc.urlprefix= jdbc:JSQLConnect:
com.ibm.di.store.jdbc.user=administrator {protect}-
com.ibm.di.store.jdbc.password=password
```

These connection parameters are used with JSQLConnect drivers. You must download the JSQLConnect.jar file and copy it into the `<TDI_HOME>/jars` subdirectory.

Note: The DATA TYPE for MS SQL is IMAGE .

Create table statements:

```
com.ibm.di.store.create.delta.systable=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL,
 SEQUENCEID int, VERSION int);ALTER TABLE {0} ADD CONSTRAINT IDI_MYCONSTRAINT_{UNIQUE} PRIMARY KEY (ID)
com.ibm.di.store.create.delta.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL,
 SEQUENCEID int, ENTRY IMAGE );ALTER TABLE {0} ADD CONSTRAINT IDI_DS_{UNIQUE} Primary Key (ID)
com.ibm.di.store.create.property.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH)
 NOT NULL, ENTRY IMAGE );ALTER TABLE {0} ADD CONSTRAINT IDI_PS_{UNIQUE} Primary Key (ID)
com.ibm.di.store.create.checkpoint.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH)
 NOT NULL, ALSTATE IMAGE, ENTRY IMAGE, TCB IMAGE );ALTER TABLE {0} ADD CONSTRAINT IDI_CR_{UNIQUE} Primary Key (ID)
com.ibm.di.store.create.sandbox.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ENTRY IMAGE)
com.ibm.di.store.create.recal.conops=CREATE TABLE {0} (METHOD varchar(VARCHAR_LENGTH), RESULT IMAGE, ERROR IMAGE)
```

DB2

Configuration of DB2 as System Store for z/OS

JDBC connection parameters:

```
com.ibm.di.store.database=jdbc:db2:net://localhost:50000/ididb
com.ibm.di.store.jdbc.driver=com.ibm.db2.jcc.DB2Driver
com.ibm.di.store.jdbc.urlprefix= jdbc:db2:net:
com.ibm.di.store.jdbc.user=db2admin
{protect}-com.ibm.di.store.jdbc.password=db2admin
```

Where *idib* in the database URL is the DSN for a DB2 instance.

The above connection parameters are used with the `db2jcc_license_cisuz.jar` DB2 jar.

Create table statements:

Note: Tablespace and Indexes must be unique.

```
com.ibm.di.store.create.delta.systable=CREATE TABLESPACE TS1DSYS LOCKSIZE ROW BUFFERPOOL BP32K;CREATE TABLE {0}
 (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, SEQUENCEID int, VERSION int) IN TS1DSYS;CREATE UNIQUE INDEX DSTIX1 ON {0}
 (ID ASC);ALTER TABLE {0} ADD CONSTRAINT IDI_DT_{UNIQUE} PRIMARY KEY (ID)
com.ibm.di.store.create.delta.store=CREATE TABLESPACE TS1DST LOCKSIZE ROW BUFFERPOOL BP32K;CREATE TABLE {0}
 (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, SEQUENCEID int, ENTRY BLOB) IN TS1DST; CREATE UNIQUE INDEX DSIX1 ON
 {0} (ID ASC); ALTER TABLE {0} ADD CONSTRAINT IDI_DS_{UNIQUE} Primary Key (ID);CREATE LOB TABLESPACE DSENT11
 BUFFERPOOL BP32K LOCKSIZE LOB;CREATE AUX TABLE TBDSEN1 IN DSENT11 STORES {0} COLUMN ENTRY;CREATE INDEX
 IXEN1 ON TBDSEN1
com.ibm.di.store.create.property.store=CREATE TABLESPACE PS3DST LOCKSIZE ROW BUFFERPOOL BP32K;CREATE TABLE {0}
 (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ENTRY BLOB) IN PS3DST;CREATE UNIQUE INDEX PSIX3 ON {0} (ID ASC);ALTER TABLE
 {0} ADD CONSTRAINT IDI_PS_{UNIQUE} Primary Key (ID);CREATE LOB TABLESPACE PSENT31 BUFFERPOOL
 BP32K LOCKSIZE LOB;CREATE AUX TABLE TBPSEN3 IN PSENT31 STORES {0} COLUMN ENTRY;CREATE INDEX PSIXEN3 ON TBPSEN3
com.ibm.di.store.create.checkpoint.store=CREATE TABLESPACE TSPACE1 LOCKSIZE ROW BUFFERPOOL BP32K;CREATE TABLE {0}
(ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ALSTATE BLOB, ENTRY BLOB, TCB BLOB) IN TSPACE1;CREATE UNIQUE INDEX INDX1 ON
{0} (ID ASC);ALTER TABLE {0} ADD CONSTRAINT IDI_CR_{UNIQUE} Primary Key (ID);CREATE LOB TABLESPACE LOBSP11
 BUFFERPOOL BP32K LOCKSIZE LOB;CREATE AUX TABLE ALSTATE1 IN LOBSP11 STORES {0} COLUMN ALSTATE;CREATE INDEX IXALST1
 ON ALSTATE1;CREATE LOB TABLESPACE LOBSP12 BUFFERPOOL BP32K LOCKSIZE LOB;CREATE AUX TABLE ENTRY1 IN LOBSP12 STORES {0}
 COLUMN ENTRY;CREATE INDEX IXENT1 ON ENTRY1;CREATE LOB TABLESPACE LOBSP13 BUFFERPOOL BP32K LOCKSIZE LOB;CREATE AUX
 TABLE TCB1 IN LOBSP13 STORES {0} COLUMN TCB;CREATE INDEX IXTCB1 ON TCB1;
com.ibm.di.store.create.sandbox.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ENTRY BLOB)
#com.ibm.di.store.create.recal.conops=CREATE TABLESPACE IM{UNIQUE} LOCKSIZE ROW BUFFERPOOL
 BP32K;CREATE TABLE {0} (METHOD VARCHAR(VARCHAR_LENGTH), RESULT BLOB, ERROR BLOB) IN IM{UNIQUE};CREATE
 LOB TABLESPACE LB{UNIQUE} BUFFERPOOL BP32K LOCKSIZE LOB;CREATE AUX TABLE AT{UNIQUE} IN LB{UNIQUE} STORES
 {0} COLUMN RESULT;CREATE INDEX IX{UNIQUE} ON AT{UNIQUE};CREATE LOB TABLESPACE LS{UNIQUE} BUFFERPOOL
 BP32K LOCKSIZE LOB;CREATE AUX TABLE AE{UNIQUE} IN LS{UNIQUE} STORES {0} COLUMN ERROR;CREATE INDEX IN{UNIQUE} ON AE{UNIQUE}
```

Configuration of DB2 as System Store for other OS (other than z/OS)

JDBC connection parameters:

```
com.ibm.di.store.database=jdbc:db2:net://localhost:50000/ididb

com.ibm.di.store.jdbc.driver=com.ibm.db2.jcc.DB2Driver
com.ibm.di.store.jdbc.urlprefix= jdbc:db2:net:
com.ibm.di.store.jdbc.user=db2admin {protect}-
com.ibm.di.store.jdbc.password=db2admin
```

Where `idib` is the DSN for a DB2 instance.

Create table statements:

```
com.ibm.di.store.create.delta.systable=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, SEQUENCEID int, VERSION int);
 ALTER TABLE {0} ADD CONSTRAINT IDI_MYCONSTRAINT_{UNIQUE} PRIMARY KEY (ID)
com.ibm.di.store.create.delta.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, SEQUENCEID int, ENTRY BLOB );
 ALTER TABLE {0} ADD CONSTRAINT IDI_DS_{UNIQUE} Primary Key (ID)
com.ibm.di.store.create.property.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ENTRY BLOB ) ;
 ALTER TABLE {0} ADD CONSTRAINT IDI_PS_{UNIQUE} Primary Key (ID)
```

```
com.ibm.di.store.create.checkpoint.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ALSTATE BLOB,
 ENTRY BLOB, TCB BLOB );ALTER TABLE {0} ADD CONSTRAINT IDI_CR_{UNIQUE} Primary Key (ID)
com.ibm.di.store.create.sandbox.store=CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ENTRY BLOB )
```

User Property Store

The User Property Store is a System Store table used for maintaining serialized Java objects associated with a key value. This is where persistent component parameters and properties (such as the **Iterator State Store**) are maintained, as well as data you store.

For example, when you set the **Iterator State Store** parameter for the Active Directory Changelog Connector, you are specifying the key value that the Connector uses to save and restore Iterator state. If you want the Iteration to start with the first (or last) change entry, simply delete the **Iterator State Store** entry in the User Property Store (that is, click **Delete** next to the parameter).

You can persist your own objects with the following **system** calls:

system.setPersistentObject(*keyValue***,obj)**
: Saves the object **obj** in the User Property Store using the specified *keyValue*. The object is returned if it was saved successfully, otherwise the function returns **null**.

system.getPersistentObject(*keyValue***)**
: Returns the object with the specified *keyValue* from the User Property Store. If the *keyValue* is not found, then the function returns **null**.

system.deletePersistentObject(*keyValue***)**
: Deletes the object with the specified *keyValue* in the User Property Store. This function returns the object that was deleted, or **null** if the *keyValue* was not found.

These methods access the default User Property Store.

Note: However, you can create and use your own stores using the **Store Factory** .

If you view the User Property Store from the System Store window, note that it has the following table definition:

Key The unique key (512 chars)

Entry The object associated with key

Note: Any object to be persisted in the User Property store must be serializable.

Delta Store

The Delta Store is found under the **Delta Tables** folder in the System Store Browser. Each table represents one **Delta Store** parameter setting (in the **Delta** tab of an Iterator). There are a number of classes and methods for working directly with the Delta Store, although this is not recommended. For more information on the Delta feature, see the section entitled “Deltas and compute changes” on page 82.

Checkpoint/Restart Store

The Checkpoint/Restart store consists of a number of classes that aid the AssemblyLine and other components to implement Checkpoint/Restart. See“Checkpoint/Restart” on page 97.

Store Factory methods

The following examples are of methods that can be used with the Store Factory:

public static PropertyStore getDefaultPropertyStore () throws Exception;
Returns the default property store.

public static PropertyStore getPropertyStore (String table) throws Exception;
Returns the Property Store identified by name. Only one instance of a given name is present at one time.

@param name
The property store name.

@return
The property store object associated with name.

public static String getSystemDatabaseURL ();
Returns the System Store JDBC URL.

public static Connection getConnection () throws Exception;
Returns a connection object to the default database.

public static Connection getConnection (String database) throws Exception;
Returns a connection object to the named database with AutoCommit set to **TRUE.**.

@param database
The database name.

public static Connection getConnection (String database, boolean autoCommit) throws Exception;
Returns a connection object to the named database.

@param database
The database name.

@param autocommit
The auto-commit flag.

@return
A connection object to the named database.

public static boolean dropTable (Connection connection, String table);
Drops a table in the database associated with connection.

@param connection
The connection object obtained by getConnection()

@param table
The table to drop

public static void verifyTable (Connection connection, String table, Vector sql) throws Exception;
Verifies that a table is accessible in the database.

@param connection
The connection object obtained by getConnection(). If NULL, a connection to the default table is obtained.

@param table
The table name to verify.

@param sql
A vector of SQL statements to create the table if it does not exist.

public static Exception dropTable (String tableName);
Drops a table in the default database.

@param tableName
The name of the table to drop

public static byte[] serializeObject (Object obj) throws Exception;
Serializes an object to a byte array.

@param obj
The object to serialize

@return
The byte array containing the serialized object

public static Object deserializeObject (byte[] array) throws Exception;
Deserializes a byte array into a Java object.

@param array
The byte array with the serialized Java object

@return
The resurrected Java object

Property Store methods

The following examples are methods that you can use with the Property Store:

public Object setProperty (String key, Object obj) throws Exception;
Adds or updates a value in the property store. If an update is performed the old value is returned.

@param key
The unique identifier

@param obj
The value

@return
The old value in case of an update

public Object getProperty (String key) throws Exception;
Returns a value in the property store.

@param key
The unique identifier

@return
Value in the store or **NULL** if not found

public Object removeProperty (String key) throws Exception;
Removes a value in the property store.

@param key
The unique identifier to remove

@return
The old value or **NULL** if key is not in the table

UserFunctions (system object) methods

The UserFunctions class (for example, the system object) has additional methods defined to get/set objects in the System Property Store:

public Object getPersistentObject (String key) throws Exception;
This method retrieves a named object from the default system property store.

@param key
The unique key

public Object setPersistentObject (String key, Object value) throws Exception;
This method stores a named object in the default system property store.

@param key
The unique key

@param value
The object to store (must be Java serializable)

@return
The old object if any

public Object removePersistentObject (String key) throws Exception;
This method removes a named object in the default system property store.

@param key
The unique key

@return
The old object if any

Property Store

The Properties framework provides a common interface for managing and using all TDI-related properties. It builds on Connector technology, allowing you to read and write properties to a broad range of systems and data stores (not just files as in previous versions).

See “Properties” on page 181 for more information.

Inheritance

All Connectors can inherit configuration parameters from other Connectors located in your Connector Library, or from system Connectors included as part of IBM Tivoli Directory Integrator The inheritance is recursive (for example, **myldap** can inherit from **corpldap** in your Connector library, which can then inherit from com.ibm.di.connector.LDAPConnector, the standard LDAP Connector included with IBM Tivoli Directory Integrator.

Connection parameters, Parsers, schema, attribute maps, Link Criteria, Hooks, and Delta settings from other Connectors can all be inherited. To get an overview of a Connector's inheritance settings, click **Inheritance** for that Connector. Optionally, you can view and change inheritance settings for any tab (**Input Map**, **Delta**, and so forth) by clicking on the **Inherit from:** box at the bottom right hand corner of the tab. See "Adding the Input Connector" in *IBM Tivoli Directory Integrator 6.1.1: Getting Started* for more information.

Inheriting from a Connector in the Connector library enables inheritance of every part of that Connector. The sole exception is the Delta settings. The Delta settings are not inheritable.

Attribute Mapping

An Attribute Map is a set of rules that define how the Attributes of an Entry are copied, transformed and/or computed. For example, an Attribute Map component (AttMap) specifies a list of Attributes that are to be stored in the Work Entry. Attribute Maps are also available for Connectors and Functions as Input and Output Maps. To understand how these work, first consider that a Connector or Function component itself is divided into two parts:

- A generic part provided by the Server kernel that allows the component to be 'clicked' into an AssemblyLine and that provides consistent and predictable behavior regardless of the platform or technology being used. This is the AL component wrapper that holds customizable settings like Hooks, Input/Output Maps, Link Criteria and Connection Failure handling.
- A technology-specific part that understands the technical details of the data source or function being accessed. This is known as a Component Interface, and it can be exchanged for a different Interface without affecting the customization of the AL component. For Connectors this is the Connector Interface, while Functions have Function Interfaces[9].

Each Interface has its own Entry object (called the conn Entry) that is used as a cache for read and write operations. Whenever data is read in by a component, it is marshalled from native types to Java objects and stored in the conn Entry. In order to bring these cached values into the Work Entry for processing, an Input Attribute Map is provided (called an "Input Map"). Conversely, if a component Interface is to write out information to an API or data source, you are provided with an Output Map for taking data in the Work Entry and copying or transforming it to the conn Entry. An output component can only write data found in its conn Entry.

The conn Entry is a temporary object that only exists during Attribute Mapping, and is therefore of limited availability in scripting. So when a Connector reads in and parses data, this information is stored in **conn** and must be transferred to the Work Entry, otherwise data read by the component Interface will not be available to the AssemblyLine. As mentioned above, you must transfer Attributes from the Work Entry to **conn** in order for an output Connector to be able to write this information to the data source. The conn Entry is only available for scripting in Attribute Mapping code, and in the Hooks that follow the Attribute Map (see "Conn object" on page 94). You can also refer to the "Hook Flow Diagrams" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information on when the temporary system script variables are available.

Depending on the mode the Connector is in, the Attribute Mapping is done in the **Input Map** tab or the **Output Map** tab of the Connector.

For example, a Connector Interface can retrieve the attributes *frstnm* (first name) and *lstnm* (last name). The Input Map takes these and concatenates the values into a single *FullName* Attribute, making only this available to other Connectors within the AssemblyLine.

On the AssemblyLine **Config ...** tab, there is an option to automatically map all attributes. Selecting this option causes all attributes to be mapped for each Connector with no Attribute Map configured. Specifically, if this is an input

9. Whenever you choose to script a component (like the Script Connector or Script Function) you only implement the Interface methods needed by the modes your component will support.

Connector (Iterator, Lookup, CallReply, or Delete), all attributes are copied from the conn Entry to the Work Entry. In the case of an output Connector (AddOnly, Update, or CallReply), all attributes are copied the other way: from **work** to **conn**.

Automatic Attribute Mapping is an AssemblyLine global value and affects all Connectors *without an explicit attribute map*. If even a single Attribute is specified in an Input or Output Map, then the Auto-map functionality is disabled for this map.

You can specify automatic mapping of all Attributes by using the special attribute name '*' (the wildcard map). The wildcard map can be combined with individual mappings for specific attributes (even adding new ones).

Finally, note that both the name of the Attribute to map, as well as the mapping itself can be specified using an Expression. See"Expressions" on page 110 for more information.

Null Behavior

Occasionally, the system tries to map an attribute that is missing. For example, if an optional telephone number is not present in the input data source, or an attribute in an Output Map is removed from the **work** Entry. Different data sources treat missing values in different ways (NULL value, empty string) and the IBM Tivoli Directory Integrator provides a way of mapping missing attributes as well. This feature is called **Null Behavior** and with it you can define both what a "Null value" is as well as how it is to be handled.

The JDBC Connector has the `jdbcExposeNullValues` parameter setting, enabling you to map NULL values to missing Attributes (see "JDBC Connector" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*).

Null behavior can be specified at a number of levels: system, Config, AssemblyLine, AttributeMap and Attribute. However, since Null Behavior is highly data source-specific, it makes most sense to set this system property at the Attribute Maplevel (for example, for all Attributes handled by a Connector's Input/Output Maps). These possible levels are described here in more detail:

System level
Specifying system level Null Behavior is done in the global.properties file by setting the `rsadmin.attribute.nullBehavior` and `rsadmin.attribute.nullDefinition` properties one of the values listed later in this section.

Config level
This overrides System level Null Behavior, and is configured by setting the `rsadmin.attribute.nullBehavior` and/or `rsadmin.attribute.nullDefinition` properties in either a Property Store to one of the following values listed later in this section.

AssemblyLine level
AssemblyLine Null Behavior is specified in by pressing the Null Value Behavior button in the AL **Config** tab.

Attribute Map level
Defining Null Behavior for all the Attributes in a map is done by pressing the Null button above the Attribute Map list.

Attribute level
Null Behavior can be configured for a specific attribute by clicking on it in an Attribute Map and selecting the desired behavior from the Attribute Details pane.

Null Behavior supports five different settings for defining a "null value", as shown below (typically defined in the Solution or Global-Property Store). Each setting shows the actual property value in parenthesis. Note that these definitions are listed in inclusive order, so that the second case also includes the first one; the third one includes the first two; and so forth:

Attribute is missing (`AbsentAttribute`)
This situation arises when the Attribute referenced as the source of value(s) in an Attribute Map is missing.

Attribute with no values (`EmptyAttribute`)
This is the case when the Attribute used as the source of value(s) in an Attribute is found, but has no values. The previous case is also checked for.

Attribute contains an empty string value (`EmptyString`)
The Attribute is found, but has only a single string value.

Value (`value`)
The Attribute contains a specified value. For AssemblyLine, Attribute Map and Attribute-level Null value definition, this value is set in the **Value** field of the Null Behavior Dialog. Here you can specify multiple attribute values if desired by placing values on separate lines. If you use **rsadmin.attribute.nullDefinition** for system and Config level setting then you must also set the **rsadmin.attribute.nullDefinitionValue** property.

Note: Several enhancements have been made to the HTTP Server connector. TCP-based components, like the HTTP Server Connector, have a switch in their Config screens for returning TCP headers as Attribute values. When this flag is unchecked, TCP headers are stored as properties in the returned Entry object.

Default Behavior (`Default Behavior`)
This setting indicates that the Null value definition must be inherited from a higher level. For example, an Attribute inherits its Null value definition from the Attribute Map setting, which in turn inherits from the AssemblyLine.

Note: Config level Null Behavior overrides any system level settings. Furthermore, the Default Behavior setting at system level is the same as specifying **delete**, while at Config level this is equivalent to **value**.

The Null Behavior feature also lets you define the action to be taken in case a "null value" is detected:

Empty String (`empty string`)
Missing attributes are mapped with a single value which has an empty String value ("").

Null (`null`)
Missing attributes are mapped with no values, meaning that the `att.getValue()` call returns **null**.

Delete (`delete`)
The attribute is removed from the map.

Value (`value`)

Missing attributes are mapped with a specified value. For AssemblyLine, Attribute Map and Attribute-level Null Behavior, the values are set in the **Value** edit of the Null Behavior Dialog. Here you can specify multiple attribute values if desired by placing values on separate lines. If you use **rsadmin.attribute.nullBehavior** for system and Config level settings then you must also set the **rsadmin.attribute.nullBehaviorValue** property.

Default Behavior (`Default Behavior`)

This setting indicates that Null Behavior must be inherited from a higher level. For example, Attribute level inherits from the AttributeMap, which in turn inherits from the AssemblyLine setting.

Note: Config level Null Behavior overrides any system level settings. Furthermore, the Default Behavior setting at system level is the same as specifying **delete**, while at Config level this is equivalent to **value**.

These options are usually set in the configuration file's Java Properties section: See "Preferences (Java properties)" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for their name and value.

Conn object

The **conn** Entry object is used by the component Interface for all data source access. Input Maps are used to move data from the conn Entry to the Work Entry. Output Maps copy/transform values in the other direction: from work to conn. The conn Entry is only available during Attribute Mapping and the Hooks that follow in the mode specific flow (see the "Hook Flow Diagrams" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*). Below is a list of the Hooks that appear near Attribute Mapping, and which therefore have access to the **conn** object.

Iterator

After the Connector has read and parsed the data into the **conn** object, Attribute Mapping is done. The **After GetNext** Hook is called just before Attribute Mapping is done, and the **GetNext Successful** Hook is called immediately afterwards, so both have access to **conn**.

Lookup

After the Connector has located the data and put it into the **conn** object, Attribute Mapping is done. The **After Lookup** Hook is called before Attribute Mapping is done, and the **Lookup Successful** Hook is called afterwards, so both have access to **conn**.

Update

Note: TDI no longer throws an ignoreEntryException when the attribute map is explicitly empty, as it used to do. Instead, flow goes to the **Before Applying Changes** Hook where Attributes can be added to **conn** using script. Note that this Hook will only be called if **Compute Changes** is turned off; or if there are changes that need to be written. If **conn** is still empty after the **Before Applying Changes** Hook, then the **On No Changes** Hook is called. At the same time, TDI logs a warning about an explicitly empty map (again, only if **conn** remains empty).

The first thing the Connector does is try to locate the Entry to be updated in order to determine whether an Add or Modify operation is needed:

- If no Entry is found, Attribute Mapping is done for the resulting add. Then empty Attributes are removed, and so are Attributes not marked as Add in the Attribute Map. Then the **Before Add** Hook is called, and the Entry is added to the source. On completion, the **After Add** Hook is called. Both Hooks have access to **conn**.
- If more than one matching Entry is found, the **Multiple Entries Found** Hook is called. If this Hook does not exist, the update fails. The **conn** object is not available in this Hook, but the **work** object is.
- If one matching Entry is found, Attribute Mapping is done. The **current** object refers to the object found and which is to be updated, and **conn** refers to the object containing the new values. The **work** object is also available. Then the **After Lookup** and **Before Modify** Hooks are called, and attributes not marked as **Mod** in the Attribute Map are removed. If **Compute Changes** is enabled, and there are changes, the **Before Applying Changes** Hook is called. Then the data source is updated with the **conn** Entry, and the **After Modify** Hook is called, or the **Modify No Changes Hook** is called. The **current** object is available in all these Hooks.

AddOnly
After the Attribute Mapping, empty Attributes are removed. Then the Hook **Before Add** is called, and the Entry is added to the source. Afterwards the Hook **After Add** is called. Both Hooks have access to **conn**.

Delete Input Attribute Mapping takes place, and **conn** is available and contains the entry to be deleted in the **After Lookup**, **Before Delete** and **After Delete** Hooks.

CallReply
No Attribute Mapping takes place, and therefore no **conn** object is available. Both Hooks have access to **conn**.

Important Config and system objects

All thread owners (AssemblyLines and EventHandlers) are accessed in scripts through the **task** object. See "The task object", *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

The **task** object is actually the thread that starts and executes the AssemblyLine or EventHandler, and gives you access to the log for this process (for example, the `task.logmsg()` method), enables you to access configuration parameters (like Null Behavior settings), and provides the debug commands, `debugBreak()` and `debugMsg()`.

In addition, Connectors in the AssemblyLine are automatically declared as script variables with the names given them in the AssemblyLine configuration. Hence, you must name Connectors such that they are valid variable names in the selected script language.

In addition, you have the primary thread, owned by the Config Editor or the Server, depending on which system is started. This object is available through the **main** variable, and provides methods for writing to the system log, starting AssemblyLines and EventHandlers, and accessing any part of the Config. For more information, see "Main object", *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*. This object has methods for manipulating AssemblyLines and querying status information.

Controlling the number of threads

Note: This section is for advanced users.

As stated previously, AssemblyLines and EventHandlers are both threads. Certain EventHandlers (such as TCP and HTTP) might even start a new thread for every new connection they receive. Branching out numerous threads by letting AssemblyLines and EventHandlers start a number of other threads can cause you to run out of memory.

Using global system properties

The global system property `com.ibm.di.server.maxThreadsRunning` can be used to reduce the maximum number of threads started by the server. This property can either be set in the global.properties file or in the **Java Properties** for a single Config. Thread control is enforced by delaying the start of new threads once the limit is reached. This approach helps avoid deadlocks where an AssemblyLine cannot finish before a branched (forked) one has completed. However, you can still get more threads started than you want. If you want better control over threading in your solution, you will need to handle this with script.

Using scripting

If you want to completely avoid starting too many threads, an easy way is to wait for AssemblyLines to finish (with the `join()` method) before you start a new one. For example:

```
// Here we start the AssemblyLine itself
var al = main.startAL ( "myAssemblyLine", entry );
// wait for al to finish
al.join();
var result = al.getResult();
```

does just that. If you omit the `al.join()` call, the main thread (AssemblyLine or EventHandler) continues without waiting for the started AssemblyLine to complete.

Another way of limiting the number of threads you have is to use the function `java.lang.Thread.activeCount()` (and optionally `java.lang.Thread.sleep()`).

The `activeCount()` method can be used to determine the total number of threads in use by the current thread *owner* (**task**).

For example, if the EventHandler checks how many running threads it owns, it can sleep before starting new ones. Here is the JavaScript code for the EventHandler to sleep 1000 milliseconds if more than 20 *child* threads are still running.

Note: The Config Editor and Server use some threads, so your `activeCount` starts around 4.

```
while (java.lang.Thread.activeCount() > 20)
   java.lang.Thread.sleep(1000);
main.startAL ( "AssemblyLine", entry );
```

If you wanted to start AssemblyLines asynchronously until your thread count is too high, you can write a script similar to this:

```
var al = main.startAL ("myAssemblyLne");
main.logmsg ("Number of threads: " + java.lang.Thread.activeCount());
if (java.lang.Thread.activeCount() > 20 )
   al.join();
```

Checkpoint/Restart

Note: The packaged Checkpoint/Restart functionality described in this section is deprecated; it will be removed in future versions of TDI. We encourage you to design other ways to implement robustness into your solutions.

IBM Tivoli Directory Integrator enables you to checkpoint the operation of AssemblyLines and restart them from the point where they were interrupted by either a controlled or uncontrolled shutdown.

Checkpoint/Restart is not supported in AssemblyLines containing a Connector in Server mode, an Iterator Connector with Delta enabled, an AssemblyLine using the Sandbox facility, or a conditional component like a Branch or Loop. The server will abort the AssemblyLine when or if this is discovered.

This Checkpoint/Restart framework stores state information and other parameters at various points during AssemblyLine execution, enabling the server to reinstate the running environment of the AssemblyLine so that it can be restarted in a controlled way. This can be on the original server, but potentially can also be on a different machine.

The ability to restart an AssemblyLine is one of the building blocks for failover functionality. Note that IBM Tivoli Directory Integrator is not a system that provides general failover functionality straight out-of-the-box. Rather, it has a framework that provides generic building blocks for this kind of functionality, and can in this way reduce the amount of hand-coding that might otherwise be required. Be aware, though, that the framework does not implement full checkpoint and restart functionality at the click of a mouse. Some thought as to how it is applied to the business problem at hand is essential.

Note: Enabling Checkpoint/Restart can degrade performance. Large amounts of information must be securely written to a repository, and this comes at a price. Occasionally, you might be better off not using the Checkpoint/Restart framework at all, as reprocessing a certain amount of entries is less costly than the performance hit on the first attempt. However, sometimes reprocessing entries is not an option.

Saving and storing AssemblyLine state information

The rationale behind the functionality is simple. Something went wrong, and you want to continue where you stopped. There are at least two reasons for this:

Performance
: You don't want to redo things you have already done. This is only true for AssemblyLines that are very slow due to the Connectors they invoke. AssemblyLines that normally process high volumes of data can experience serious loss of performance when Checkpoint/Restart is enabled.

Business implications
: Resetting a telephone-number once might not be an issue, but ordering a ticket multiple times certainly is.

You want to avoid doing work that has already been done. However, if the AssemblyLine, during its first run, caused side-effects (for example, building some information store outside IBM Tivoli Directory Integrator during Hook processing), then at restart, that information store is not available. The IBM Tivoli Directory Integrator Checkpoint/Restart framework can only restore state and environment under its direct control.

The IBM Tivoli Directory Integrator Checkpoint/Restart framework saves the following information in a designated store. The directory path for this store is specified in the **Checkpoint** tab of the AssemblyLine. When the AssemblyLine is run, the "System Store" on page 84 framework creates and maintains this database for you, storing the following state objects:

- Initial Work Entry (IWE)
- TCB
- Position in the AssemblyLine (which step, and which Connector)
- **work** Entry object at any given step
- For an Iterator, the position in the input set (cursor)
- Likewise, for a File System Connector in AddOnly mode, the point at which the last entry was successfully written
- Other state information for the Connector as required to be able to reinstate it

Note: Whatever is saved during AssemblyLine processing is highly configurable. Do not assume that in order for an AssemblyLine to be restartable, every element and every step must be saved. It is possible that a good design enables a restart even without some of these elements. Saving less rather than more impacts performance in a positive way. You specify what to save in the AssemblyLine **Checkpoint** tab.

In the IBM Tivoli Directory Integrator Config Editor, you find a master switch for the whole AssemblyLine called **Enable Checkpoint**. This must be on for any Checkpoint/Restart functionality and any recording of information to take place. Also, in order to be able to distinguish between different runs of the AssemblyLine you must specify an Identifier, which can be used in a subsequent Restart. (If nothing is specified, a default in the form of `IDI_CP_`*`AssemblyLine-name`* is used.)

Note: If you use the "Commit on end of AL cycle" option (an AL global option in the AL tab) you should not be using Checkpoint/Restart, and you should switch off the commit "after every DB operation" for each and every Connector in the AssemblyLine. Otherwise unpredictable results (like half-updated databases) may occur. Conversely, if you choose to use CheckPoint/Restart, you should enable Autocommit for each and every Connector, and **not** use "Commit on end of AL cycle".

Then, for each Connector in the AssemblyLine, there is an **Enabled** check box; this must be **on** for the CPR framework to consider recording any information about this particular Connector at all. It also causes Connectors in Iterator mode to save the **work** Entry as it hands off control to the next Connector.

Checking the **work** Entry check box instructs the Checkpoint/Restart framework to record the contents and state of the **work** Entry before this particular Connector does its work in the AssemblyLine. By enabling **Connector Restart Info**, you ensure that the Checkpoint/Restart records any information required for this connector to be able to make a meaningful resumption of its processing during an AssemblyLine restart. An example of this is the position in an input file.

Notes:

1. EventHandlers do not know anything about Checkpoint/Restart, and enabling AssemblyLines for Checkpoint/Restart which are started from EventHandlers can be difficult to get right in terms of synchronization during a possible Restart.

2. Make sure that in your implemented solution you cannot end up in a situation where multiple of the same AssemblyLine get started using the same Identifier. There is nothing in the framework that stops you from doing this, and if multiple AssemblyLines using the same Identifier do execute at the same time they will also both use the same tables in the System Store for bookkeeping, with unpredictable results.

   A common situation where this might occur is if you start an AssemblyLine (using Checkpoint/Restart) from an EventHandler, and the EventHandler does not wait for the AssemblyLine to complete before returning to the main event loop. In such a case there is a high chance that when the Config file is rerun after an abnormal termination and EventHandlers restart, they may start multiple instances of an AssemblyLine, all with the same Identifier. They would all enter Restart processing using the same Restart information, with very strange results.
3. It is good practice to start such an AssemblyLine with an empty Initial Work Entry (IWE) once when an EventHandler starts up. If there was an instance of the AssemblyLine left in restartable mode, it completes its work; otherwise gets the null IWE and do nothing (make sure your AssemblyLine does not abort on an empty work entry).

It follows that the Checkpoint/Restart framework can restore state and environment only under its direct control and only if it was saved in the first place. Anything you builds up outside the Checkpoint/Restart framework might not be available at restart. If continued AssemblyLine processing is predicated upon such external information being in the exact same state, then it is up to you to ensure that this external information is restored before a successful attempt at a restart can be made.

One example of an AssemblyLine that might survive a restart fairly well is one that updates a JDBC database. Transactions executed by a JDBC Connector in AddOnly or Update mode are secured by the integrity mechanisms of the underlying database, so all information stored there might still be available when the AssemblyLine is restarted (provided no other processes manipulate or change that data). Even data stored by direct JDBC calls in Hooks might survive this scenario.

However, anything stored by a JavaScript call to files in the underlying operating system is likely to be lost when the AssemblyLine terminates unexpectedly, and therefore are not available at restart.

Notes:

1. When IBM Tivoli Directory Integrator writes to files in a Checkpoint/Restart-enabled AssemblyLine, it takes care to flush output to files where and when appropriate. A user writing directly to files needs to take the same precautions.
2. Whether you enabled saving the **work** entry or not, the **work** entry is automatically saved for Iterators.

```
If (Checkpoint/Restart enabled AND ConnectorIsIterator)
 alwaysSaveWorkEntry().
```

Limitations

Because of the way some Connectors and Parsers are implemented, there are some limitations on what the Checkpoint/Restart framework can safeguard.

For Connectors in AddOnly mode, any data stored in memory is lost if the AssemblyLine terminates unexpectedly. For example, a File System Connector

using the simple XML Parser writes the entire XML document when the Connector closes at the end of the AssemblyLine process. If the AssemblyLine is interrupted before this point and the Connector close does not take place, nothing is found in the output. Therefore, any Connector operating on a dataset using the XML parser (and its derivatives, DSML and SOAP) in AddOnly mode, and with the Connector Restart Info Save flag enabled, fails.

Note: With care, you can work around this limitation. However, the framework does not do that out of the box.

Another example of where state information is somewhat difficult to maintain correctly is a JMS connector. Distinguish between reading in Auto-acknowledge mode or not.

Conceptually, by using Checkpoint/Restart, and with Auto-acknowledge set to **on**, your task can become much easier as the framework is supposed to save the **work** object and acknowledge receipt of the message in one AssemblyLine step.

However, because of certain limitations in the way Checkpoint/Restart is implemented in the AssemblyLine flow process, there currently exists a window between where a message is received from the JMS queue, and where the applicable contents of this message, mapped into the work entry, is committed to the Checkpoint/Restart System Store. If an interrupt happens in this window, then with Checkpoint/Restart enabled and Auto-acknowledge **on**, on restart this message is lost. With Checkpoint/Restart enabled and Auto-acknowledge **off**, the message is retrieved once more from the queue, which in most cases is a more desirable behavior. Still, for proper queue processing, messages must be acknowledged at some point in this latter mode. To keep any other windows of failure as small as possible this is best done by inserting a Script component just after the JMS connector in the AssemblyLine which does just one thing: tell the JMS bus (that is, acknowledge) receipt of the message, because from here on we are certain that we have what we need of the message present in the work entry, properly safeguarded by the Checkpoint/Restart framework.

The following table presents the possible combinations of Connectors, any Parsers (if enabled and required by the Connector) and their operation Modes, not only specifically in the context of the Checkpoint/Restart framework but even in the general IBM Tivoli Directory Integrator context:

Connector	Parser	Mode						Remarks
		I	L	C	A	U	D	
File System								
	CSV	Y	X	X	Y	X	X	
	XML	Y	X	X	N	X	X	
	Fixed	Y	X	X	Y	X	X	
	DSML (v1)	Y	X	X	N	X	X	
	HTTP	Y	X	X	Y	X	X	
	LDIF	Y	X	X	Y	X	X	
	Line Reader	Y	X	X	Y	X	X	
	SOAP	Y	X	X	N	X	X	
	Simple	Y	X	X	Y	X	X	

	Script Parser	Y	X	X	–	X	X	Any writing to output files must be flushed explicitly.
URL Connector								
	CSV	–	X	X	N	X	X	
	XML	–	X	X	N	X	X	
	Fixed	–	X	X	N	X	X	
	DSML (v1)	–	X	X	N	X	X	
	HTTP	–	X	X	N	X	X	
	LDIF	–	X	X	N	X	X	
	Line Reader	–	X	X	N	X	X	
	SOAP	–	X	X	N	X	X	
	Simple	–	X	X	N	X	X	
	Script Parser	–	X	X	N	X	X	
HTTP Client								
	CSV	–	–	–	–	X	X	
	XML	–	–	–	–	X	X	
	Fixed	–	–	–	–	X	X	
	DSML (v1)	–	–	–	–	X	X	
	HTTP	–	–	–	–	X	X	
	LDIF	–	–	–	–	X	X	
	Line Reader	–	–	–	–	X	X	
	SOAP	–	–	–	–	X	X	
	Simple	–	–	–	–	X	X	
	Script Parser	–	–	–	–	X	X	
HTTPServer								
	CSV	Y	X	X	N	X	X	The HTTP Server Connector is fundamentally incompatible in a restartable AssemblyLine.
	XML	Y	X	X	N	X	X	
	Fixed	Y	X	X	N	X	X	
	DSML (v1)	Y	X	X	N	X	X	
	HTTP	Y	X	X	N	X	X	
	LDIF	Y	X	X	N	X	X	
	Line Reader	Y	X	X	N	X	X	
	SOAP	Y	X	X	N	X	X	
	Simple	Y	X	X	N	X	X	
	Script Parser	Y	X	X	N	X	X	
FTP Client								
	CSV	Y	X	X	N	X	X	
	XML	Y	X	X	N	X	X	

	Fixed	Y	X	X	N	X	X	
	DSML (v1)	Y	X	X	N	X	X	
	HTTP	Y	X	X	N	X	X	
	LDIF	Y	X	X	N	X	X	
	Line Reader	Y	X	X	N	X	X	
	SOAP	Y	X	X	N	X	X	
	Simple	Y	X	X	N	X	X	
	Script Parser	Y	X	X	N	X	X	
TCP								
	CSV	Y	X	X	–	X	X	
	XML	Y	X	X	–	X	X	
	Fixed	Y	X	X	–	X	X	
	DSML (v1)	Y	X	X	–	X	X	
	HTTP	Y	X	X	–	X	X	
	LDIF	Y	X	X	–	X	X	
	Line Reader	Y	X	X	–	X	X	
	SOAP	Y	X	X	–	X	X	
	Simple	Y	X	X	–	X	X	
	Script Parser	Y	X	X	–	X	X	
Memory Stream								
	CSV	N	X	X	N	X	X	
	XML	N	X	X	N	X	X	
	Fixed	N	X	X	N	X	X	
	DSML (v1)	N	X	X	N	X	X	
	HTTP	N	X	X	N	X	X	
	LDIF	N	X	X	N	X	X	
	Line Reader	N	X	X	N	X	X	
	SOAP	N	X	X	N	X	X	
	Simple	N	X	X	N	X	X	
	Script Parser	N	X	X	N	X	X	
Script Connector		–	–	–	–	–	–	Any Checkpoint/Restart logic is implemented by user.
SNMP		Y	X	X	Y	X	X	
Notes		Y	Y	X	Y	Y	Y	
Domino Users Connector		Y	Y	X	Y	Y	Y	
LDAP		Y	Y	X	Y	Y	Y	
JNDI		Y	Y	X	Y	Y	Y	
Mailbox		N	–	X	X	X	–	
JDBC		Y	Y	X	Y	Y	Y	
JMS								
	CSV	Y	Y	X	Y	X	X	

	XML	Y	Y	X	Y	X	X	
	Fixed	Y	Y	X	Y	X	X	
	DSML (v1)	Y	Y	X	Y	X	X	
	HTTP	Y	Y	X	Y	X	X	
	LDIF	Y	Y	X	Y	X	X	
	Line Reader	Y	Y	X	Y	X	X	
	SOAP	Y	Y	X	Y	X	X	
	Simple	Y	Y	X	Y	X	X	
	Script Parser	Y	Y	X	Y	X	X	
IBM-MQ								
	CSV	Y	Y	X	Y	X	X	
	XML	Y	Y	X	Y	X	X	
	Fixed	Y	Y	X	Y	X	X	
	DSML (v1)	Y	Y	X	Y	X	X	
	HTTP	Y	Y	X	Y	X	X	
	LDIF	Y	Y	X	Y	X	X	
	Line Reader	Y	Y	X	Y	X	X	
	SOAP	Y	Y	X	Y	X	X	
	Simple	Y	Y	X	Y	X	X	
	Script Parser	Y	Y	X	Y	X	X	
IBM Directory Changelog		Y	Y	X	X	X	X	
Active Directory Changelog		Y	Y	X	X	X	X	
Netscape Changelog		Y	Y	X	X	X	X	
Exchange Changelog		Y	Y	X	X	X	X	
BTree Object DB		Y	Y	X	Y	Y	Y	
Command line								
	CSV	Y	X	X	Y	X	X	
	XML	Y	X	X	Y	X	X	
	Fixed	Y	X	X	Y	X	X	
	DSML (v1)	Y	X	X	Y	X	X	
	HTTP	Y	X	X	Y	X	X	
	LDIF	Y	X	X	Y	X	X	
	Line Reader	Y	X	X	Y	X	X	
	SOAP	Y	X	X	Y	X	X	
	Simple	Y	X	X	Y	X	X	
	Script Parser	Y	X	X	Y	X	X	

Here is the legend for this table:

Y Connector or Parser is enabled for Checkpoint/Restart.

N Connector or Parser is incompatible with Checkpoint/Restart.

– No special considerations or not available to Checkpoint/Restart. Use with care.

X Illegal mode for this Connector, or illegal combination of mode and Parser.

I Iterator mode.

L Lookup mode.

C CallReply mode.

A AddOnly mode.

U Update mode.

D Delete mode.

There are two more Modes: **Server** and **Delta**. However, AssemblyLines containing Connectors in any of those two modes are not compatible with Checkpoint/Restart.

Note: You must deal with any variables declared and used in the scripting environment (Hooks, Attribute Mapping, and so forth) for Checkpoint/Restart purposes.

Restart implications

When an AssemblyLine starts normally, it runs defined Prologs, both before Connectors are initialized and after. The Connectors themselves run various Hooks grouped under the **Prolog** tab, notably, the **Before Initialize** Hook and **After Initialize** Hook.

During a restart, it is imperative that you do not do anything in these Hooks that might contradict an operation being resumed halfway through. For example, a File System Connector doing maintenance tasks such as deleting files, setting up temporary directories, and so forth. In other words, in order to make an AssemblyLine safe and suitable for a restartable environment, some modifications need to be made to the various scripted elements, both at the AssemblyLine level (the Prologs and Epilogs) as well as the individual Connectors Hooks and Script Components.

In order to be able to tell whether a restart is taking place, a Boolean method named `task.isRestarting()` can be invoked. It returns **true** during restart processing. Once the AssemblyLine has resumed fully, the method returns **false**. This enables any user code to take special precautions during a restart, and in the previous example the File System Connector can skip the maintenance tasks as it wants to continue with the partial results obtained so far before it was interrupted.

Restart actions

The following steps take place when an AssemblyLine is started:

1. An AssemblyLine is started (restarted).
2. The Server task verifies there is valid Checkpoint/Restart data for the Checkpoint Identifier specified for this AssemblyLine.
3. If the data is valid and indicates that a restart is necessary, the AssemblyLine Prolog is run, and Connectors are initialized and told that they are restarting. Then **After Initialization** Prologs are executed.
4. Various data elements of the interrupted AssemblyLine are restored:
   a. **work** entry as it existed at the point of interruption.
   b. Connector-specific information for all Connectors.

c. If the AssemblyLine had one or more Iterators, the number of times the active Iterator had executed is restored. In other words, the position in the input source is restored.

5. The AssemblyLine resumes the main loop from just past the last step that was executed and saved in the Restart info. It can cause the AssemblyLine to jump straight to the Epilogs or directly to one of the Connectors in the AssemblyLine, not necessarily the first one.

The matter of restoring the position in the input source in the case of an Iterator requires some more explanation.

Iterator Connectors

Connectors in Iterator mode generally fall into one of two categories:

- Those that operate on a fixed set of data (for example, a file)
- Those that iterate on a dynamically created result set (for example, SQL SELECT, ldapsearch, and so forth)

For this purpose, an FTP Client Connector is an example of one that operates on a fixed set of data. Even though the Connector reads through a network connection, the data set it operates on is still a fixed set that sits on a file system somewhere. It just is not based locally.

The Iterators in the first category are repositioned by the Checkpoint/Restart framework at restart by simply reading (and discarding) as many entries from the input source as the interrupted AssemblyLine had (any Hooks with the exception of the **Override GetNext** Hook are bypassed). When the AssemblyLine resumes operations properly and gets to the Iterator, the correct entry is retrieved.

Note: This holds true, of course, if the input set or file has not changed from the previous run to the one that is restarting. If it has, the reposition operation will result in an entirely different position than expected.

However, Iterators in the second category do not attempt to read as many entries as before. Those Connectors that save their own state information are instructed to reposition themselves. Other Connectors, such as the JMS Connector, simply use the first entry presented on the newly-restored connection. The JMS Connector works with other message queues, in addition to the already supported IBM MQ. Although support for IBM MQ is included, you can plug in other message queues by supplying your own JMS initiator class. See the "Connectors" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information about this Connector.

Note: Again, this will work correctly only if the input set has not changed from the previous run.

Here is an overview of action taken on restart for the various Connectors in Iterator mode (provided Restart info is saved):

Connector in Iterator mode	Restart Action
File System	Read and Discard
URL Connector	Read and Discard
HTTP Client	No Action
HTTP Server	No Action (incompatible with Checkpoint/Restart)

FTP Client	Read and Discard
TCP	No Action
Memory Stream	Read and Discard
Script Connector	Read and Discard
SNMP	No Action
Notes	Read and Discard
Domino Users Connector	Read and Discard
LDAP	Read and Discard
JNDI	Read and Discard
Mailbox	Read and Discard
JDBC	Read and Discard
JMS	No Action
IBM-MQ	No Action
IBM Directory Changelog	Restart from last Change Number
Active Directory Changelog	Restart from last Change Number
Netscape Changelog	Restart from last Change Number
Exchange Changelog	Restart from last Change Number
BTree Object DB	Read and Discard
Command line	Read and Discard

Note: In many cases, depending on the Iterator driving it, a restart of an AssemblyLine might not make much sense. For example, even though a Memory Stream Connector might read and discard from the source, this source was a memory object that was not saved and therefore has not survived the restart. It returns end-of-data immediately. If you have such components in your AssemblyLine, then by design the AssemblyLine is not suitable for Checkpoint/Restart.

AddOnly Connectors

Connectors in AddOnly mode fall into two categories:

- Those that operate on a fixed set of data (for example, a file)
- Those that iterate on a network connection and therefore operate on a per-entry basis

Similar to Iterators, AddOnly Connectors attempt, if possible, to restore their position in the AssemblyLine when the AssemblyLine was running, so that after a restart there is a seamless continuation of output.

In order to understand the limitations of Connectors in AddOnly mode, there needs to be distinction between Connectors writing databases (writing single entries or full datasets). The following table describes this:

Connector in AddOnly mode	Output Quantum
File System	Data set
URL Connector	Data set
HTTP Client	Entry
HTTP Server	Entry

FTP Client	Data set
TCP	Data set
Memory Stream	Data set
Script Connector	Entry
SNMP	Entry
Notes	Entry
Domino Users Connector	Entry
LDAP	Entry
JNDI	Entry
Mailbox	na
JDBC	Entry
JMS	Entry
IBM-MQ	Entry
IBM Directory Changelog	na
Active Directory Changelog	na
Netscape Changelog	na
Exchange Changelog	na
BTree Object DB	Entry
Command line	Data set

Any of the Connectors in AddOnly mode that operate on a data set in the previous table are incompatible with the XML parser for restart purposes. If **Save Connector Restart Info** is enabled for such a combination of Connector/Mode/Parser, the AssemblyLine fails.

For a File System Connector in particular, IBM Tivoli Directory Integrator attempts to seek to the last reliable position in the output file (truncating any half-written data that might end up in the file) and append new output from the resumed AssemblyLine there.

In concept, the FTP Client Connector is similar, except the file is on a remote system. However, the current implementation of this Connector does not enable the same behavior. Any output to a file on a remote system through an FTP connection is probably lost when the connection is terminated abnormally on the previous run. In this case, on restart the FTP Client Connector starts writing a new file (an FTP append mode is not supported).

The FTP Connector supports IP version 6.

The Config

The **Config** is a complete description of the systems, data flows and events that define your integration solution. A Config is stored as a highly structured XML document, and can be encrypted if desired. The AssemblyLines, Connectors, Functions, Scripts, Parsers and Attribute Maps, along with Server and GUI preferences and so forth are all stored in the Config, and edited and maintained using the **Config Editor** (ibmditk).

Usually you have everything in one Config. However, there are situations where you want to isolate parts of it:

- Usernames and passwords in configured components (or any parameter information) that you want stored (and even encrypted) elsewhere
- Shared Components that you want to reuse in several Configs

Configs are stored on a file system, and are either edited with the Config Editor locally (that is, the file system and the Config Editor are on the same system) or remote (in which the Config Editor is in constant dialog with a remote process, an instantiation of the IBM Tivoli Directory Integrator server in daemon mode). This latter concept, called the Remote Config Editor, provides support for platforms with no native Config Editor. It can be used to read/write/execute a configuration on a remote server, and is an extension to the local Config Editor. The basic idea is to provide a uniform interface for both remote and local Config files.

Remote Configs

There are two options for loading a remote configuration for editing:

- The configuration is loaded only for editing and cannot be started at all.
- The configuration is loaded for editing and a temporary Config Instance is started on the Server so that the configuration can be tested while being edited.

In both cases, the configuration loaded for editing is independent of the Configs loaded and running (normally) on the Server. As a result, you could have the same Config both running on the Server and independently opened for editing (with or without a temporary Config Instance). In other words, you cannot edit a Config instance that is running on the Server. Furthermore, when you save the Config back to disk on the TDI Server, this operation does not affect an active instance of the same configuration on the TDI Server. Instead, there are API calls (like those used by AMC) to perform a ″reload″ operation, bringing in the last version of the Config from the disk.

See “Remote” on page 129 for a description of the new Remote Configuration interface.

Parameter substitution with Expressions

A large number of windows in the TDI Config Editor provide input fields where parameters to various components are configured. For example, the Config tabs for Connectors or Functions. You have the choice of entering a specific value or clicking on the parameter label. This brings up the Parameter Information Dialog which provides an Expression field for either entering a TDI Expression, or from the list of **Properties**. The parameter substitution mechanism allows you to exploit Expressions to build parameter values that can even change over time. See “Expressions” on page 110 for more information.

User-defined Property Stores

These are stored in external files, which can be individually encrypted. They are configured by clicking the **Properties** folder in the Config Browser and adding, changing or removing Property Stores from this list. See “Properties” on page 186 for more information.

Note: Property Stores are intended for externalizing certain configuration parameters such as username, password, filename and so forth. If you want to save more general parameters and make them available for your AssemblyLine, see “Config ...” on page 142.

Advanced parameter substitution

In addition to External Properties, there is a more powerful parameter substitution mechanism in TDI called Expressions which builds on top of the services provided by the standard Java java.text.MessageFormat class. The MessageFormat class provides powerful substitution and formatting capabilities, and greatly simplifies how dynamic parameter values are set.

The configuration of these parameter substitutions is accomplished by means of the parameter substitution editor.

Include/Namespaces

A Config can include items from Configs. These included components belong to another namespace, but can be used within the context of the including Config as though they were stored there. See “Includes” on page 185 for more information.

To create a new namespace, select **Object–>New Include ...** and choose a name (for example, **myInclude**) for your namespace (**system** is reserved). Link the name you choose to a file in the table, and you can then later refer to myInclude:/Connectors/myConnector to use a myConnector from the Connector library of your include.

Securing Configs, passwords and sensitive data

TDI saves configuration information in an XML file (Config file) which contains clear text for all configuration values. This often includes sensitive information like passwords. TDI not only supports encryption of the entire configuration file, but also protecting individual values or settings. You can also be set up to automatically handle any component parameters tagged as passwords. Values entered into these configuration fields are delegated to the specified Password Property Store . The parameter itself is then set with an Expression that references the newly created password property. So, as passwords are entered or changed in the password field, the actual value is never visible or stored in the Config itself.

You can also set up TDI to handle any component parameters tagged as passwords. Values entered into these configuration fields are delegated to the specified Password Property Store. The parameter itself is then set with an Expression that references the newly created password property. So, as passwords are entered or changed in the password field; the actual value is never visible or stored in the Config itself.

Note: Changes will not be made to existing Configs. If you want previously defined Config password parameters stored in the Password Store as well, then you must define a Password Store and re-enter the passwords themselves.

Default and user-defined parameter protection

The password protection mechanism is directly related to the configuration windows offered to you. The configuration windows, or forms, contain descriptions of each parameter and its syntax. One type of parameter syntax is the “password” type, which causes the CE to use the special password edit field for user intput. Whenever the value for a password syntax component parameter is changed, the value entered is saved to the designated Password Store. If no such Property Store is configured, then password values are still saved in clear text in the configuration file. See the "Security and TDI" chapter in the*IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*. The new `setProtectedParameter(name,value)` method in the `com.ibm.di.config.interfaces.BaseConfiguration` interface will

query the associated MetamergeConfig object for the default password store. If one is configured, a unique property name is generated the first time a call to `setProptectedParameter` is called. This key is used as the key in the password store. The same property name is written to the configuration file as a standard property reference. When the value is later retrieved, standard property resolution takes place to retrieve the actual value from the password store. Hence, there is not an accompanying `getProtectedParameter(name)` for retrieval of protected parameters. The type of protection of the value is not defined by this feature since the protection mechanism may vary from connector to connector.

New methods in the API

The following methods have been added to the `com.ibm.di.entry.Attribute` and `com.ibm.di.entry.AttributeInterface` classes:

```
public void setProtected(boolean protect)
```

If the parameter is true, try to protect the Attribute values by not dumping them in log files:

```
public boolean getProtected()
```

Returns true if the values should not be dumped in log files.

The following method was added to the `com.ibm.di.entry.Entry` class:

```
public void setAttribute (Object name, Object value, boolean protect)
```

Where:

- *name* is the attribute name
- *value* is the attribute value. If this parameter is null, then the attribute is removed.
- If the *protect* parameter is true, do not dump the Attribute values in log files

The following method was added to `com.ibm.di.server.TaskCallBlock`:

```
public void setConnectorParameter (String connectorName, Object parameterName,
 Object parameterValue, boolean protect)
```

If the *protect* parameter is true, do not write the value of the parameter in log files

Methods that have been modified to not dump protected Attribute values:

- `Log.dumpEntry( Entry e )` This will also affect for example, the dump() and dumpEntry() methods in AssemblyLine (task) and RS (main).
- `Attribute.toString()` - Which also affects Entry.toString()
- `Attribute.toDeltaString()` - Which also affects Entry.toDeltaString()
- `TaskCallBlock.setConnectorParameters (AssemblyLine task)`

Also modified `Entry.mergeAttributeValue(Object p1, AttributeInterface p2)` - If either Attribute is protected, the merged Attribute is protected.

`Entry.merge (Entry e, boolean mergevalues)` - Make sure merged Attributes are protected if either of the old Attributes is.

Expressions

TDI boasts an Expressions feature that allows you to compute parameters and other settings at run time, making your solutions dynamically configurable. This feature expands on the Properties handling found in previous versions.

In addition to support for simple External Properties references (fully backwards compatible), Expressions provide more power in manipulating AssemblyLine and component configuration settings during AL/component initialization and execution. Expressions can also be used for Attribute maps, as well as for Conditions and Link Criteria, alleviating much of the scripting previously required to build dynamically configured solutions. TDI provides an Expressions editor to facility building these expressions.

The Expressions feature is built on top of the services provided by the standard Java java.text.MessageFormat class. The MessageFormat class provides powerful substitution and formatting capabilities. Here is a link to an online page outlining this class and its features: http://java.sun.com/j2se/1.4.2/docs/api/java/text/MessageFormat.html

In addition to features described in the above class, TDI provides a number of runtime objects that can be used in expressions—although the availability of some objects will depend on runtime state (for example, whether conn or current defined, or the error Entry): The Expressions syntax provides a shorthand notation for accessing the information in these objects, like Attributes in a named Entry object, or a specific parameter of a component.

TDI Reference	Value	Avaliability

work.*attrname[.index]*	The *work* entry in the current AssemblyLine.
	
	The optional *index* refers to the n'th value of the attribute. Otherwise the first value is used.
	
	This Advanced Attribute Map:
	`ret.value = work.getString ("givenName") + " " + work.getString("sn");`
	
	can be expressed simply as:
	`{work.givenName} {work.sn}`
		AssemblyLine

conn.*attrname[.index]*	The conn entry in the current AssemblyLine The optional *index* refers to the *n*'th value of the attribute. Otherwise the first value is used.	AssemblyLine during Attribute Mapping
current.*attrname[.index]*	The *current* entry in the current AssemblyLine The optional *index* refers to the *n*'th value of the attribute. Otherwise the first value is used.	AssemblyLine during Attribute Mapping for Modify

TDI Reference	Value	Avaliability
config.*param*	The configuration object of the "scoped" component/AL: Furthermore, if "config" is used in the parameter of a Connector, Parser or Function, then it refers to the Config object of that component's *Interface* (for example, JDBC Connector, or XML Parser) *param* is the name of the parameter itself, as if you were to make a call to getParam() or setParam(). For example, for the JDBC Connector you could make the following reference: `{config.jdbcSource]`	AssemblyLine EventHandler Connector Parser Function Component

alcomponent.*name.param*	The component Interface parameter value of a named AssemblyLine component.
	
	name is the name of the AssemblyLine component
	
	param is the parameter name of the *name* object
	
	So, the following Expression:
	`{alcomponent.DB2conn.jdbcSource}`
	
	is equivalent to the following scripted call:
	`DB2conn.connector.getParam ("jdbcSource");`
		AssemblyLine

property[:storename].name property[:storename/ bidi].name	A TDI-Properties reference. The optional storename targets a specific Property Store. If no storename is specified, then the default store is used. *name* is the property name *bidi* will, when present, cause setting the parameter value to forward the call to the referenced Property Store. When *bidi* is present no other substitution patterns or text is allowed.	Always.

TDI Reference	Value	Avaliability
JavaScript<<EOF script code ... // Must contain "return"EOF	*Embedded* script code used to generate a value for the Expression. This script must return a value. The "EOF" text shown is an arbitrary string that terminates the JavaScript snippet. The JavaScript is collected up until a single line with the EOF string is encountered (or no EOF is flag is set – see the note below). Note that embedded JavaScript is evaluated using the AssemblyLine's script engine instance, so you have access to all variables otherwise present for scripting. **Note:** There is a short-hand form of adding JavaScript that works for input fields that do not support multiple lines (like Link Criteria or the names of Attributes in maps) and can therefore not have the necessary EOF line: `{JavaScript return work.getString ("givenName") + " " + work.getString("surName")}`	Always

Embedded JavaScript in Expressions has access to the AssemblyLine's script engine. As a result, even script variables defined elsewhere in the AssemblyLine can be accessed. Note that if you reference a variable or object that is not one of those specifically listed in the tables shown in this section, the Expression evaluator will check with the AL's script engine to see if it is defined there.

Expressions in component parameters

When used for a component parameter, the following objects are of special interest:

Table 3.

Object	Value
config	The component's Interface configuration object.
mc	The MetamergeConfig object of the Config instance (`config.getMetamergeConfig()`)
work	The Work Entry of the AssemblyLine.
task	The AssemblyLine object.

As an example, take a JDBC Connector with the Table Name parameter set to "Accounts." You could then click on the **SQL Select** parameter label and then enter this into the **Expression** field at the bottom of the dialog:

```
select * from {config.jdbcTable}
```

This will take the Table Name parameter and create the following SQL Select statement:

```
select * from Accounts
```

Or you could get more advanced, and try something like this for the SQL Select parameter:

```
SELECT {JavaScript<<EOF

 var str = new Array();
 str[0] = "A";
 str[1] = "B";
 return str.join(",");
EOF

} FROM {property:mystore.tablename} WHERE A = '{work.uniqueID}'
```

The embedded JavaScript will return the value "A,B" which is then used to complete the rest of the Expression. If you have a Property Store called "mystore" with a "tablename" property set to "Accounts", and there a "uniqueID" Attribute in the Work Entry with the value "42", the final result will be:

```
SELECT A,B FROM Accounts WHERE A = '42'
```

This evaluated result is not displayed onscreen in the CE. Simply entering curly braces will not cause Expression evaluation to be done for the parameter value. Instead you have three choices when tying Expressions to parameters:

1. Press Ctrl+E to open the Expressions Editor dialog while in the parameter input field.
2. Click on the Parameter label and enter the Expression directly into the field labeled "Expression" at the bottom of the Parameter Information Dialog.
3. Type the special preamble, @SUBSTITUTE, manually into the parameter input field, followed by the Expression. For example:

   ```
   @SUBSTITUTEhttp://{property.myProperties:HTTP.Host}/
   ```

   Note: This last method of entering expressions directly is not recommended; rather use the Expression Editor instead.

Expressions in LinkCriteria

Expressions in Link Criteria provide a similar list of pre-defined objects. Again, note that you also have access to any other objects or variables currently defined in the AssemblyLine's script engine.

Table 4.

Object	Value
config	The component's Interface configuration object.
mc	The MetamergeConfig object of the Config instance (config.getMetamergeConfig())
work	The Work Entry of the AssemblyLine
task	The component itself, or a named component.
alcomponent	The Connector/Function component

So, for example, let's say that you want to set up the Link Criteria for a Connector so that the Attribute to use in the match is determined at run time. In addition to standard data Attributes in the work Entry, there is also a "matchAtt" Attribute with the string value "uid". In this case, the following Expressions used in Link Criteria:

```
{work.matchAtt}  EQUALS {work.uid}
```

Is equivalent to this:

```
uid  EQUALS $uid
```

Expressions in Branches, Loops and Switch/Case

The list of Expression objects here is similar to that for Link Criteria:

Table 5.

Object	Value
config	The component's Interface configuration object.
mc	The MetamergeConfig object of the Config instance (config.getMetamergeConfig())
work	The Work Entry of the AssemblyLine
task	The AssemblyLine
alcomponent	The Connector/Function component

You have can use Expressions for both the Attribute Name and the Operand of a Condition. You can also use Expressions to configure Switch and Case components.

Scripting with Expressions

You can also use Expression directly from JavaScript code. Here is an example that builds an expression using the new ParameterSubstitution class:

```
var ps = new com.ibm.di.util.ParameterSubstitution("{work.FullName} -> {work.uid}");

map = new java.util.HashMap();

map.put("mc", main.getMetamergeConfig());
map.put("work", work);

task.logmsg(ps.substitute(map));
```

Resulting in the following log messages when run for several iterations in the test AssemblyLine:

```
14:35:29  Patty S Duggan -> duggan
14:35:29  Nicholas P Butler -> butler
14:35:29  Henri T Deutch -> deutch
14:35:29  Ivan L Rodriguez -> rodriguez
14:35:29  Akhbar S Kahn -> sahmad
14:35:29  Manoj M Gupta -> gupta
```

Chapter 3. The Config Editor

Config Editor Interface

This section describes the menu items and windows of the IBM Tivoli Directory Integrator Graphical User Interface (GUI): the Config Editor.How to work with the concepts the Config Editor visualizes is explained in "Using the Config Editor" on page 124.

Main window

When you start the IBM Tivoli Directory Integrator Config Editor, either from your system's launch interface or from the command line with the *ibmditk* command you will see the Main window.

In the default layout, using the *Cards* layout, the left navigation pane (called the Config Browser) provides a tree view of the current configuration, as well as all the current AssemblyLines, EventHandlers, Connectors, and so forth.

The buttons in the button bar at the top of the main window, from left to right, are:

Create a new configuration root
Enables you to create a new configuration.

Open an existing configuration
Enables you to open one or more configuration files you have already created and saved.

Save selected configuration
Enables you to save your configuration without closing it.

Show/Hide the Config Browser
This closes the left navigation pane, also known as the Config Browser.

Previous tab/window
Enables you to navigate through already open panes, for example, AssemblyLines, EventHandlers, Connectors, and so forth. Greyed out if there are no open panes to show.

Next tab/window
Enables you to navigate through already open panes, for example, AssemblyLines, EventHandlers, Connectors, and so forth. Greyed out if there are no open panes to show.

Help for this window
This brings up the help page (i.e. this page).

Solution Directory

IBM Tivoli Directory Integrator allows you to specify the Solution Directory that it is to use. This affects relative paths through your Config, and is also where a number of special property files and sub-directories are expected by the Server. You will be asked to set this value during installation. The options presented are:

Do not specify. Use current working directory at startup time
If this selection is made, no Solution Directory will be set. As a result, the base path will be wherever you happen to start the system from. Since the

Server (and CE) will expect the presence of property files and support sub-directories, startup may fail depending on the current directory path.

Use a TDI subdirectory under my home directory
A TDI subdirectory will be created under your home directory if it does not already exist. This is the default setting and the preferred method for organizing your TDI solutions.

Use the Install Directory
This is the backwards-compatibility choice, since previous versions of TDI assumed that relative paths in your Configs where based from the installation directory.

Select a directory to use
This choice allows you to specify any directory where you have write access.

Note that if you select any other directory than the installation directory, property files and necessary sub-folders will be created and maintained there for you.

The Solution Directory can be specified when you start either the CE or Server with the **–s** parameter. If the CE is started with the **–s** parameter, it checks to see if this directory contains the necessary solution files and directories. If any of these are missing (for example, this is the first time you are using a new Solutions Directory) then the system prompts you to confirm that the specified directory is correct, terminating if you answer ″No″. If you confirm the Solution Directory, TDI copies in the necessary solution info from the TDI installation directory and uses it for this session. The Server will not make these preparations for you, and will instead fail with error messages if you attempt to use an unprepared Solutions Directory.

Note: Prevent conflicts between your global.properties file and your solution.properties file. Whenever you need to edit properties, make sure you are doing so in the right place. Use the global.properties file to store settings that affect all users and all solutions that share the same TDI installation. Use solution.properties for personalizing the server configuration for a specific solution or user. If you have installed TDI with a solutions directory, then you have a solution.properties file in this folder that overrides settings in your etc/global.properties file in your installation directory. You must make sure that there are no statements in the solution.properties file that override your global.properties file settings.

Files in the Solution Directory include:

`solution.properties`
This is usually a copy of the `global.properties` file, tailored for your own solution-specific needs. If the Solution Directory does not contain this file, a copy of `global.properties` is placed there by the Config Editor (unless the Solution Directory is equal to the installation directory).

`log4j.properties`
A copy of the default log strategy configuration file log4j.properties. Refer to “Logging and debugging” on page 156 for more information.

`idisrv.sth`
A ″stash″ file for your solution. Refer to the "Security and TDI" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

`serverapi`
A directory containing configuration files related to the Server API; refer to the "Server API" chapter in *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

Java Libraries

IBM Tivoli Directory Integrator enables you to layer new functionality on top of the server by including your own Java libraries.

This is done by selecting the JavaLibraries folder in the Config browser. The system presents you with a list of included libraries.

JavaLibraries can only be created and removed by using the toolbar in the Details Pane title area.

Once you have created a new Java Library entry, simply click the grid field that you want to edit and press **F2**. Alternatively, you can double-click the field to enter edit mode. Fill out the following fields:

Name This is the name of the object that is made available to your scripts, and which provides you access to the library's functions.

Value Here you enter the name of the Java class that is tied to this new script object.

Note: In order to be able to include new Java libraries, you must place the library's jar files in either the `.jars` subdirectory of the IBM Tivoli Directory Integrator installation directory, or in an existing or new subdirectory under `.jars`.

For more information, see "Java Libraries" on page 183

Java Properties

The Java Properties is a section of the configuration file where you specify a list of properties and their associated values. The properties are added to the Java runtime properties so you can change or add Java-specific or other third party-specific properties here. Also, you find IBM Tivoli Directory Integrator properties that fine tune the way IBM Tivoli Directory Integrator Server (ibmdisrv) and Config Editor (ibmditk) behaves.

These values are part of your configuration file and might be different from file to file. You cannot include these values. If you want these values to be global for your installation, you can set them in the global.properties file.

Your user preferences (for the Config Editor) are stored in the `.ibmdi` file that you find in your home directory. This file is created for you the first time you start ibmditk.

For more information, see "Preferences" on page 185

Includes

A Config can be set up to include configuration elements from other Configs. This makes it possible to set up central libraries of components, scripts and other Config elements that can be stored on corporate servers, but still available to integration specialists working through the organization.

You create and manage your Includes as you do other Config items like Connectors and AssemblyLines.

The only available option at this point is the Config Driver parameter. Select the Configuration Storage Architecture (CSA) driver that you must use to reach this Config.

There are only two choices at present:

- Pre-5.1.1 legacy .cfg format
- XML

Note: The .cfg format is supported as read-only and for legacy reasons. It is recommended that you use the XML format for storing Config information to file.

Once you have chosen the Config driver, you are presented with more parameters. You can set the filename or URL to the Config file, as well as the decryption password (if one is in use). You can have as many Includes as you want in your Config. These can be stored in various ways. They can be accessed through the different Config drivers.

For more information, see "Includes" on page 185

Properties

The Properties window lets you manage both the standard Property Stores (Global, User and Java), as well as the System Property Store and any user-defined Property Stores that you create for your solution. User-defined Property Stores are typically used to store sensitive information outside your Config in a secure format, but still keep it configurable.

Think of these user-defined Properties as global system variables that can be used throughout your solution. Of course you can access Properties from your scripts, enabling you to make your code data-driven, changing its functionality based on the value of one or more of these properties. However, the one powerful use for Properties is as parameter values in the configuration of components, like Connectors, Parsers and Functions, or as part of Expressions that can be used to define everything from Link Criteria to Attribute Maps.

TDI 6.1 introduces the TDI Properties framework, providing a common interface for managing and using all TDI-related properties. It builds on Connector technology, allowing you to read and write properties to a broad range of systems and data stores (not just files as in previous versions).

For more information, see "Properties" on page 186 and "Expressions" on page 110

Accessing properties from JavaScript

In addition to the existing UserFunctions (the "system" object) functions like getExternalProperty() and getJavaProperty, there are new methods for working with the Properties framework that manages all types of Properties: getTDIProperty() and setTDIProperty(). Both functions come in two flavors, one which names the property in question and another which allows you to specify a specific Property Store as well.

System Store

The System Store provides persistent storage of Entry objects and other state information for IBM Tivoli Directory Integrator solutions. The default version of the system store uses CloudScape as its underlying storage technology; other database systems like IBM DB2 can be used as well as databases accessible through a JDBC driver.

The System Store can be shared by multiple of IBM Tivoli Directory Integrator servers if the CloudScape database runs as a server, or if another multi-user database system is used. If CloudScape runs embedded in an IBM Tivoli Directory Integrator server, it cannot be shared simultaneously with other servers.

The System Store implements three types of storage for TDI components:

The Property Store
The Property Store is a simple keyed table with associated Java objects designed for use by user script code. The Java object must be serializable. A default property store is made available through additional system object methods and by direct access to the Store Factory.

Delta Store
The Delta Store holds the keyed tables that are used by selected Connectors in Iterator mode, with Delta support enabled. Delta Objects are only stored in the Delta Store if a storage method other than Btree Objects is selected.

Checkpoint/Restart Store
The Checkpoint/Restart Store consists of a number of classes that aid the AssemblyLine and other components to implement Checkpoint/Restart.

For more information, see "System Store" on page 84

Preferences

The behavior of the Config Editor can be tailored to a certain extent, by means of a number of settings in the **File->Edit Preferences ...** dialog. Sub-windows in the Preferences dialog are:

- "File Settings"
- "Editor Settings" on page 122
- "Appearance" on page 122
- "Misc Settings" on page 123

The settings are described in more detail below.

File Settings

This window has the following parameters for you to set:

Max Recently Used Files
The maximum number of recently used files to display.

Current Recently Used Files
You can edit the files you want to display in the **File->Recent** option from the main menu.

Auto-load Files
You can list files that load automatically when you open IBM Tivoli Directory Integrator.

Editor Settings

In this window you can change the settings of the internal script editor in the Config Editor, or point to an external editor. Options are:

Editor Font
The font used by the internal script editor.

Max undo levels
The maximum number of Undo operations you can perform on the internal edit buffer. A very high number may cause memory problems; the default is 100.

External editor
The command line to invoke an external editor executable to use instead of the internal editor.

Note: For UNIX systems, the following restriction exists:

Make sure that an editor window is displayed. The editor can be, for example, "*xterm -e v*i" or *emacs* or anything else that displays a window.

Wait & paste from external editor
Checking this box (which is on by default) causes the external editor to be run synchronously — that is, the Config Editor waits for the external editor to finish. After that, the results of the external edit session are pasted back into the script buffer you were editing. Conversely, if this box is not checked, the external editor is started asynchronously with the current contents of the script buffer, but the results of the external editor are not applied back.

Confirm before update
Waits for an OK before updating local buffer (only when this option is checked).

Appearance

The Appearance tab has four sub-tabs. Under **General**, there are the following parameters that you can set:

View Type
Defines the look of the Interface. Options are **Tabbed**, **Cards**, and **Frames**.

Tabbed Browser
When checked, will show different Config Files as stacked tabs instead of serially in the Config Browser.

Show Toolbar
When checked, enables the toolbar to be shown.

Show Status bar
When checked, enables the status bar to be shown.

Meta Look
Provides a legacy look to the Interface, providing a white and orange ribbon at the top of the Interface.

Under the **Look & Feel** sub-tab, you will find the following parameters that you can set:

Look & Feel
Provides a specific style to the Interface. Options are platform-dependent, on Windows they are **Metal**, **CDE/Motif**, and **Windows**.

Metal Theme
Specifies a Theme for the Metal Look & Feel option (see previous). Currently, the only choice is **com.ibm.di.admin.ProductTheme**.

The **Theme Settings** sub-tab has the following parameters you can set:

Menu Font
Change the font that appears in menus. This text is provided by the product.

UserText Font
Change the font of text that you provide.

Controls Font
Changes the font of names of menu options.

SubControls Font
Changes the font of sub-menu options.

WindowTitle Font
Changes the font of window titles.

Note: To determine the combination that appeals to you the most, you will need to experiment with these settings.

The **Theme Colors** sub-tab has the following parameters you can set:

Primary 1 - 3
Optional primary colors for the Interface.

Secondary 1 - 3
Optional secondary colors for the Interface.

Default colors
Select to enable the default colors that are part of the Interface.

Note: To determine the combination that appeals to you the most, you will need to experiment with these settings.

Misc Settings

The **Miscellaneous settings** tab has the following parameters you can set:

ExecuteTask Lines
Number of lines displayed in execute task window (the window that opens when you run an AssemblyLine from the Config Editor).

Internet browser
Choose the browser you want to use to display help files.

Resources

A simpler method for sharing AssemblyLines and components is provided by the TDI Resource library in the Config Editor. **Resources** appear on the **Resources** tab located just below the Config Browser.

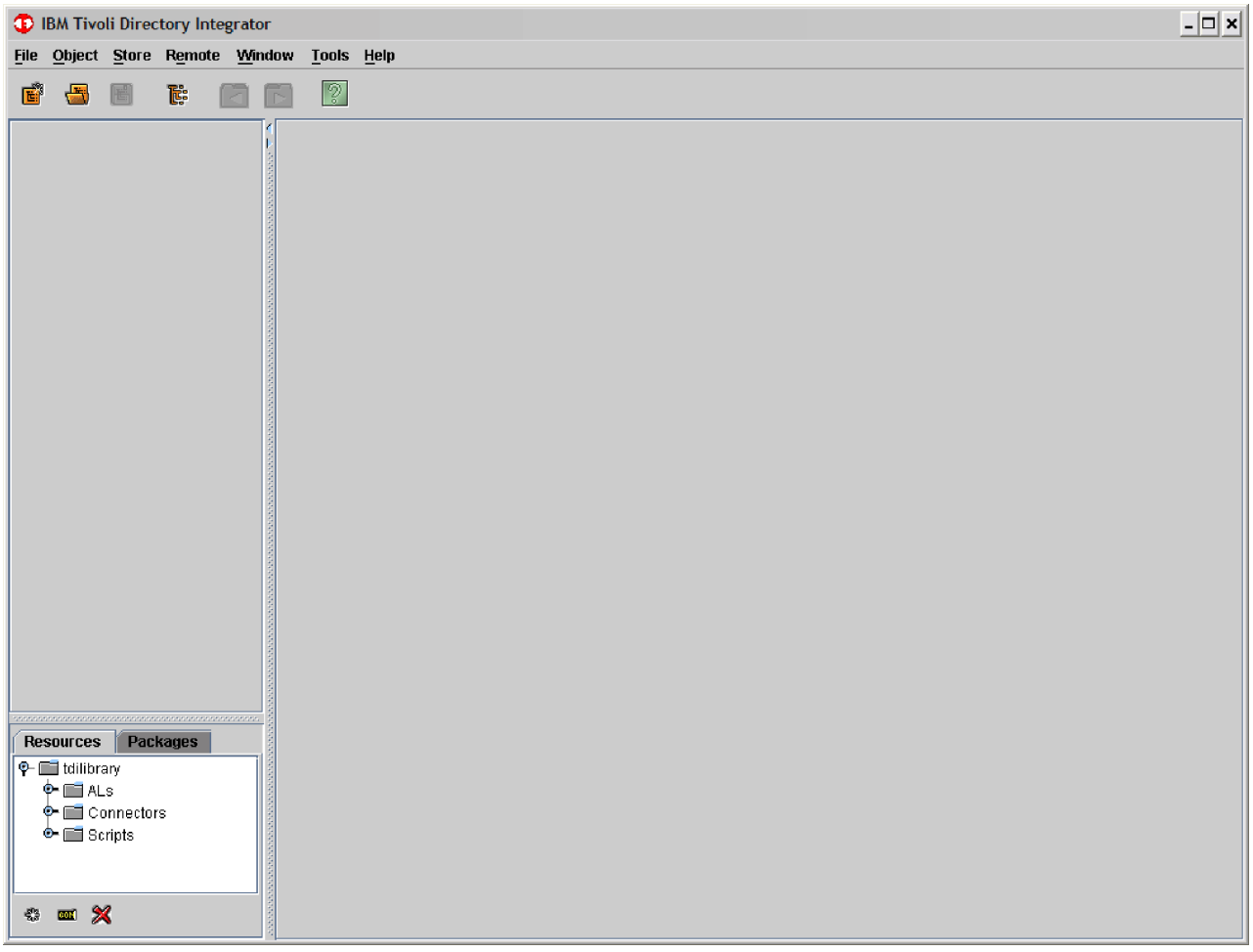
The resources in the **Resources** tab reflect a directory structure located at the path specified in the Solution or Global Property by the `com.ibm.di.admin.library.dir`

property. By default, this is set to "<*User Home*>/tidlibrary", but can be reassigned to point to any other folder, including shared space on a network disk.

ALs and components can be dragged between the Config Browser and Resources. However this works the same as dragging between multiple Configs open in the Config Editor: you only get the items you drag. So if you drag an AssemblyLine to Resources (or to another open Config), it will not bring along any **Resources** components that are being inherited from. The answer to this dilemma is the new AssemblyLine Package Publishing feature. See "Packaging, Resources and reports" on page 136 for more information.

Using the Config Editor

Open an item in the Config Browser by clicking it. If the item is a folder, it reveals the items enclosed, otherwise it opens the item. If there are no items in the folder, then you must only click once on the folder (double-clicking a standard folder causes you to try to rename the folder, which you cannot do). Folders are distinguished from items by the open/close widget. You can also open an item by selecting **Object–>Open** item selection from the Main window menu, or right-click a folder (for example, **Connectors**) and select **New Connector ...**.

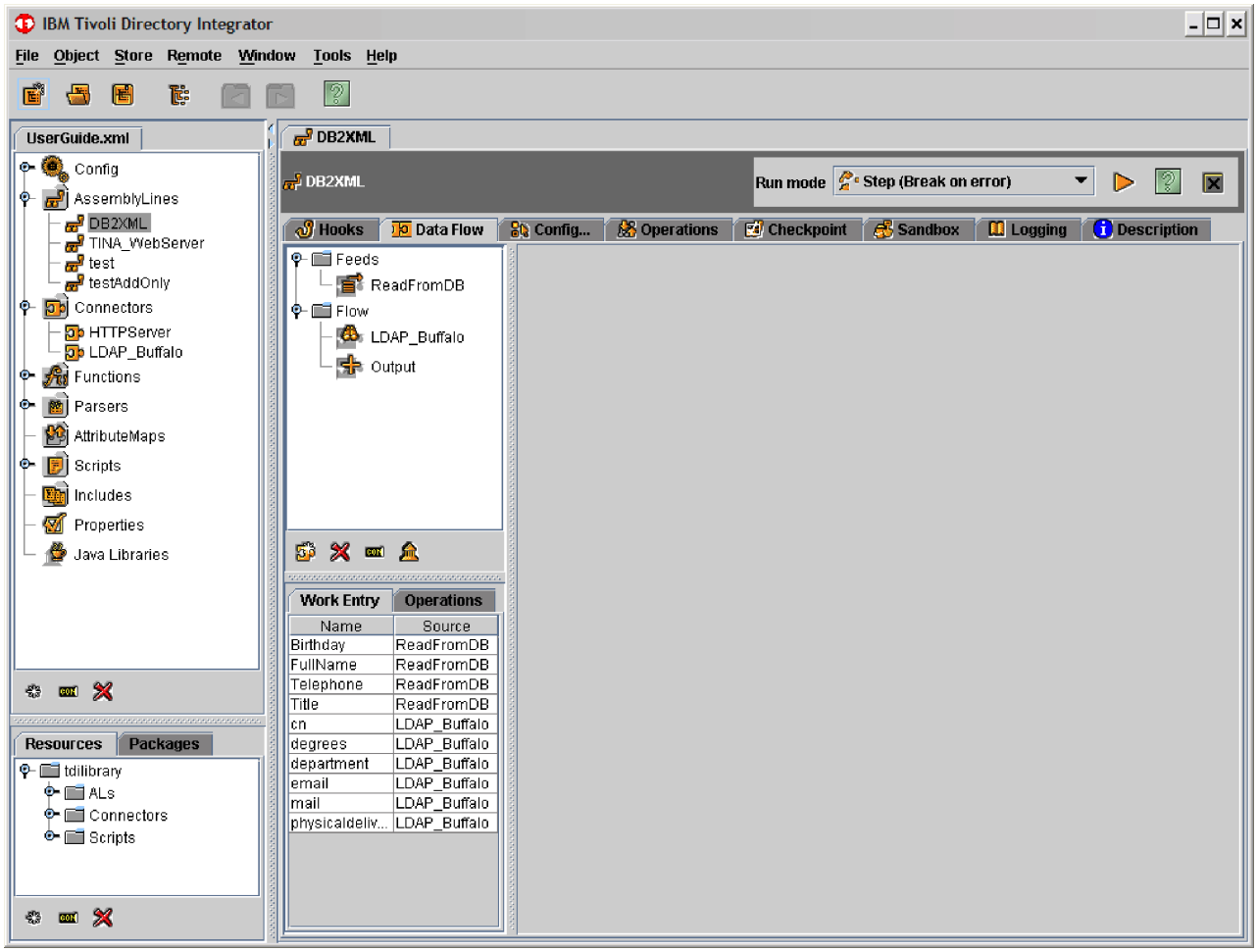


You can rename an object in the Config Browser by selecting it and pressing **F2**. Alternatively, you can double-click the item. This allows the item to be renamed.

Note: You cannot rename standard folders.

If you open an item in the Config Browser, then the Details Pane is filled with information for that object. This is called a **Details Window**.

Each Details Window has its own toolbar providing quick access to relevant features for the type of object displayed (in fact, IBM Tivoli Directory Integrator has a number of toolbars, each tied to a particular window or List Control). Common for all Details Window toolbars is the **Close** button, which is displayed at the right side of the toolbar.



List controls

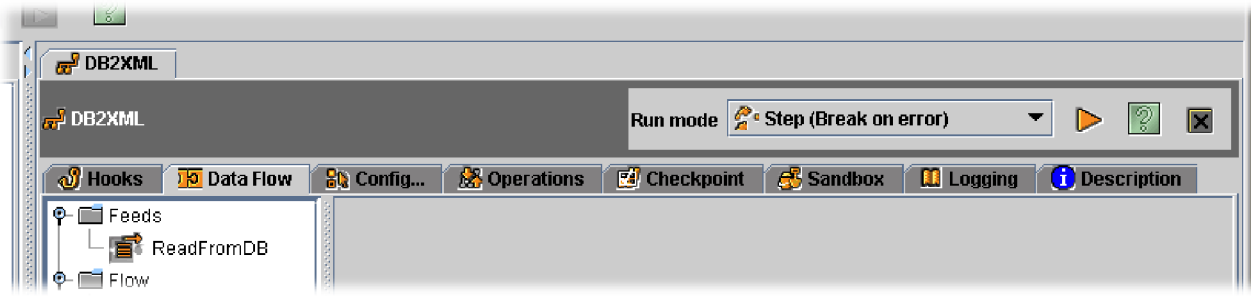
This image shows AssemblyLine details. In the left part of the Details Pane you sometimes find a List Control. Each List Control has a toolbar associated with it, usually found below or above the List Control itself.

In order to view the details of an item in a list, click the item to select it. Most List Controls support multiple selections (**Ctrl** + mouse-click, **Shift** + mouse-click or **Shift** + arrow keys).

If you want to rename an item in a List Control, you can enter edit mode by pressing **F2**. Some List Controls, such as those for AttributeMaps, enable you to double-click the item to edit, or simply to start typing.

Tab controls

Along the top of the various sections of information in the Details Pane are a set of tab controls, as in the following screenshot:



Keyboard controls

IBM Tivoli Directory Integrator has several **Look & Feel** options:

- **Metal**
- **CDE/Motif**
- **Windows**
- **Windows Classic**

Perform these steps to access the **Look & Feel** option:

1. Click **File>Edit Preferences**.
2. Click **Appearance** tab.
3. Click **Look & Feel** tab.
4. Click **Look & Feel** box.
5. Select a **Look & Feel**.
6. Click **OK**.

To see the keymaps for these **Look & Feel** options, go to the following URLs:

Metal http://java.sun.com/j2se/1.4/docs/api/javax/swing/doc-files/Key-Metal.html

CDE/Motif
http://java.sun.com/j2se/1.4/docs/api/javax/swing/doc-files/Key-Motif.html

Windows
http://java.sun.com/j2se/1.4/docs/api/javax/swing/doc-files/Key-Win32.html

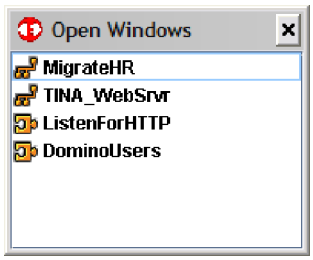
Note: In some fields where the **Tab** key itself is a legal character, use **Ctrl+Tab** to shift focus.

Moving between details windows

The Details Window tabs are displayed as you open objects in the Config Browser. You can change focus from one object to another by doing any of the following actions:

- Select another object in the Config Browser.
- Click the desired Details window tab.

- Use the **Previous Tab/Window** and **Next Tab/Window** buttons in the Main window toolbar.
- Open the Window List dialog (click **Window>Show/Hide Windows list** from the Main Menu).



Select the desired window in order to make it active in the Details Pane.

Main menu selections

File

Most of the options in the **File** menu are also available in the toolbar in the "Main window" on page 117. The options in this menu operate on local Config files only; all actions related to remote configurations are located under the "Remote" on page 129 menu

New ...
Creates a new Config. Shortcut: **Alt-F N**

Open ...
Opens an existing Config. Shortcut: **Alt-F O**

Close Closes selected Config.

Save Saves selected Config locally. Has no effect on running ALs. Shortcut: **Ctrl-S** and **Alt-F N**. See the "Remote" on page 129 menu for remote saving.

Save As ...
As Save but saves the Config file under a different name. Shortcut: **Alt-F A**. Note that the newly-saved Config is kept open and available in the Config Browser.

Save All
As Save, but for all loaded config-files.

Recent
Opens a recently accessed Config.

Edit Preferences
For setting user and system preferences.

Quit Exits IBM Tivoli Directory Integrator.

When editing Config files on a Remote server, you need to make sure that the communication between the local Config Editor and the Remote server is set up correctly; see the "Remote Server" chapter in *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide* for more information. A remote server must be running on the specified IP address and port, and the path entered must exist and must be writable. The Config Editor will then create a Config file with default folders and

save it on the remote server. It also maintains a temporary local copy of the file, and loads it in the Config Editor in the format <IP>_<port>_<filename>.

Object

Most of the options on the Object menu are also available as context-sensitive options on the objects in the Config Browser. Right-click on an object to see the context-sensitive menu.

New AssemblyLine ...
Creates new AssemblyLine in **AssemblyLines** folder.

New EventHandler ...
Creates new EventHandler in **EventHandlers** folder.

New Connector ...
Creates new Connector in **Connectors** folder.

New Parser ...
Creates new Parser in **Parsers** folder.

New Script ...
Creates new Script in **Scripts** folder.

New Include ...
Creates new Include (file reference) in **Includes** folder.

New Function ...
Creates new Function Component in the **Functions** folder.

New Attribute Map ...
Creates a new, independent Attribute Map in the **AttributeMaps** folder.

New external property file ...
Creates a new entry for an external properties file in the ExternalProperties folder.

Open Item
Opens selected object (Connector, AssemblyLine, and so forth).

Delete Item
Deletes selected items.

Rename
Allows you to rename the selected item.

Clone Copies the selected item, and prompts for the name of the new item. This is a convenient way to copy an item in the same place in the hierarchy, as opposed to a copy-and-paste operation.

Config Report
Initiates an HTML-format report, based upon an XSL StyleSheet, about the selected object. You will be asked to provide the StyleSheet, or choose from a pre-defined template.

You can also use standard Copy, Cut and Paste operations from the Object menu to manipulate Config elements; see "Copying elements between open Configs (or folders)" on page 133.

Store

Manage System Stores
This selection allows you to define one or more set of **System Store** settings, which can then be referenced and used by Configs. See "System Store" on page 84.

View System Store
Allows you to open, view and delete System Store tables.

Network Server Settings
For configuring and managing the settings for System Stores that use networked of Cloudscape.

Remote

The Remote menu lets you edit a Config file that is located on a remote system with a TDI remote server running on it.

A remote server can be started by executing the *ibmdisrv* script on the remote system with the **-d** option. The interface and functionality of the remote server prevents simultaneous access to Configs.

All actions related to remote configurations are located under the **Remote** menu.

Note: Refer to the "Remote Server" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide* for more information as to how to configure a Remote Server.

The Remote menu lets you edit a Config file that is located on a remote system with a TDI remote server running on it. The interface and functionality of the remote server prevents simultaneous access to Configs. You can start a remote server by executing the ibmdisrv script on the remote system with the -d option. Editing files on a remote server does not affect running AssemblyLines on that server.

New Remote
Creates a new configuration on a Remote TDI Server, immediately locks that configuration for editing and starts a temporary Config Instance on the Server. This will prompt for the IP address, port and relative path of the Config to created on the Remote Server.

Open Remote
Loads a configuration for editing on a Remote TDI Server, locks that configuration and starts a temporary Config Instance on the Server. This will prompt for the IP address and port of the remote server, and a password if the Config you select is password protected.

Save Saves the configuration on the Remote Server, but keeps the exclusive lock on the configuration. Does not affect running AssemblyLines on the Remote Server.

Save Local Copy
Saves a copy of the remote configuration on the local Config Editor machine. The configuration is not saved on the Remote Server, the lock is not released and the Config Instance on the Remote Server is not stopped. In the Config Editor, you continues to work with the remote configuration and not with the locally saved copy.

Save and Close
Saves the configuration on the Remote Server, releases the lock on the configuration and stops the temporary Config Instance on the Server.

Close Closes the remote configuration in the Config Editor, releases the lock on the configuration on the Remote Server and stops the temporary Config Instance.

Window

Next Window
Displays next tabbed window in Details Pane.

Previous Window
Displays previous tabbed window in Details Pane.

Close Closes current tabbed window in Details Pane.

Close All
Closes all tabbed windows in Details Pane.

Show/Hide Window list
Toggles visibility of the window list.

Tools

Key Manager
Starts the certificate tool IKeyman.

Edit Solution Properties
Launch an editor window with the currently active Solution Properties file. If your Solution Directory is the installation directory, this will be the `global.properties` file.

The Tools menu is configurable. By default, it contains the entry for the IKeyman tool, but by modifying the properties file `ibmditk-plugins.properties` (located in the installation directory) you can add items to the Tools menu.

In this file, the **ToolsMenu** block contains a list of comma separated names. Each name in this list has four properties suffixed **.label**, **.tooltip**, **.params** and **.javaclass** that defines the Label, Tooltip, action parameters and the Java class that performs an action when the menu item is activated. The format is:

```
ToolsMenu=Sample
Sample.label=MenuItemLabel
Sample.tooltip=Menu Item tooltip / description
Sample.javaclass=javax.swing.Action class
Sample.params.<parameters required by the action class>
```

The default value for **.javaclass** is `com.ibm.di.action.ShellAction`, which executes a command line on the local system. The ShellAction class uses the `commandline` parameter as its execution path. The sample below defines a menu item labeled **IBM** that opens http://www.ibm.com in a browser (on Windows):

```
ToolsMenu=GotoIBM
GotoIBM.label:IBM Home Page
GotoIBM.tooltip:Goto http://www.ibm.com
GotoIBM.params.commandline:explorer.exe http://www.ibm.com
```

Help

Online Help is not contained within the IBM Tivoli Directory Integrator binaries, but is retrieved from an IBM Eclipse-based Infocenter[10], either running locally on your workstation or on a server somewhere on the network. See the section on "Installing local Help" under the installation instructions in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide* for more information.

Help Initial online help information about the Main window.

About IBM Tivoli Directory Integrator
Displays version information.

10. The help system is powered by Eclipse technology. (http://www.eclipse.org)

About IBM Tivoli Directory Integrator Components
Displays version information about all the Components that are part of the IBM Tivoli Directory Integrator.

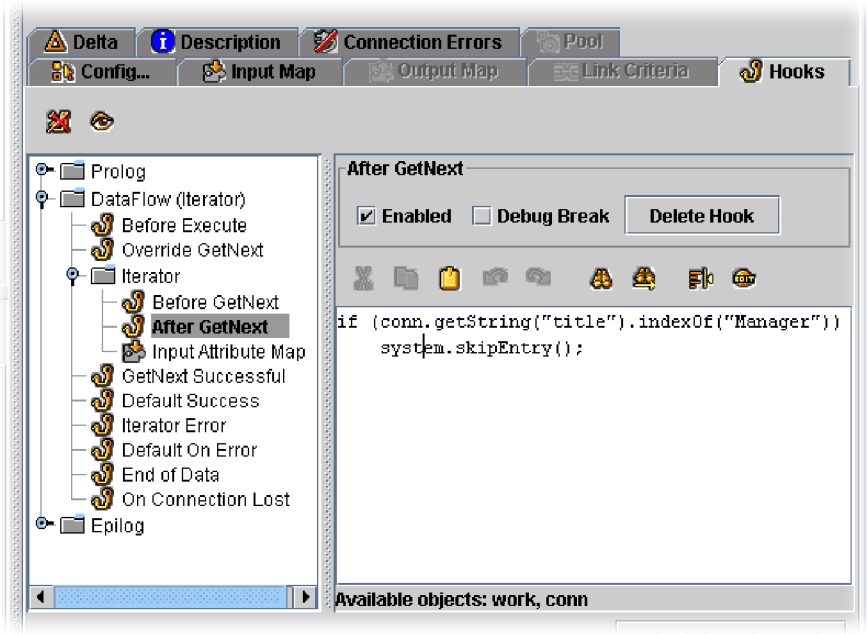
About Remote IBM Tivoli Directory Integrator Components
Displays version information about all the Components that are part of the Remote IBM Tivoli Directory Integrator Server instance you specify.

Low Level API
Launches a Web browser with the IBM Tivoli Directory Integrator Javadocs documentation main page.

Newsgroup
Launches your default Network News Transfer Protocol (NNTP) news reader to the TDI community newsgroup.

Script editor windows

Script Editor windows are found any place where you can enter script, for example, Prologs/Epilogs, Hooks, Advanced Attribute Mapping, and so forth.



This image is from an AssemblyLine **After Getnext** (in an Iterator connector).

Script editor windows operate as simple text editors, with support for the standard **Cut** (**Ctrl+X**), **Copy** (**Ctrl+C**) and **Paste** (**Ctrl+V**) shortcuts. These commands are also available as buttons of the Script Editor toolbar.

The next two buttons are for **Undo** and **Redo**, followed by another set of buttons for **Find** and **Find Again**. The **Toggle line wrap in editor** button toggles line-wrap in the editor.

The **Edit buffer with external editor** button launches an external editor for script editing. The choice of external editor is set in the **Edit–> Preferences** selection from the Main Menu (the **Editor** tab).

Note: The tabs **Theme Settings** and **Theme Colors** are available if the **Look & Feel** tab has **Metal** selected as the theme. Perform the following steps:

1. Click **File–>Edit Preferences**.
2. Click **Appearance** tab.
3. Click **Look & Feel** tab.
4. Click **Look & Feel** box.
5. Select **Metal**.
6. Click **OK**.
7. Go back to the **Appearances** tab. You can change the theme and color settings in the **Theme Settings** and **Theme Colors** tabs.

Currently, the only theme available is the **Product Theme**.

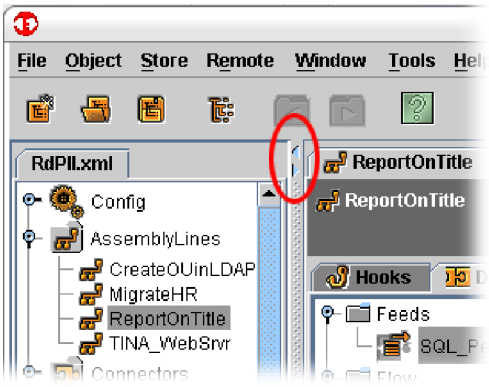
Configurations (Config)

A **Config** is a description of an Integration implementation that both the Config Editor and the server read and work with. Configs are shown in the Config Browser at the left side of the IBM Tivoli Directory Integrator Config Editor.

When you create a new Config, IBM Tivoli Directory Integrator creates a default set of folders. These folders cannot be deleted or renamed. Furthermore, you are not enabled to create other folders at this level. Instead, you are able to create sub-folders to these main Config divisions, enabling you to organize your Connectors, Parsers, AssemblyLines and other Config items into logical groups.

Notes:

1. This part of the window can be hidden or shown by clicking the **Config browser** button in the main toolbar. You can also click the arrows that appear at the top of the divider bar.



2. The Config Browser is a hierarchical tree-view browser, so you might not be seeing all elements in a Config. For more information, see "Config folder management" on page 134.

Creating a new Config

- Go to **File–>New ...**.
- Click **Create a new configuration root** button in the main toolbar

Each of these methods results in the Select Configuration Driver dialog where you can choose to have your Config stored in XML format. For IBM Tivoli Directory Integrator legacy 4.x format configs cannot be created, but they can be read.

Once you have made this selection, you must then enter the filepath for this new Config.

You can choose to encrypt the Config file by entering an encryption password.

Note: If you enter the filename of an existing file, you will be asked to confirm if you want to overwrite the old file.

Opening an existing Config

- Go to **File–>Open ...**.
- Go to **File–>Recent** (only available for recently opened Configs).
- Click **Open an existing configuration** button in the main toolbar.

Choose the Config you want to open. The Config is displayed in the left navigation window.

Saving a Config

- Go to **File>Save**.
- Click **Save selected configuration** button in the main toolbar.

This saves the currently open Config.

Renaming a Config

- Go to **File>Save As ...**

Give the Config a new name in the **Save As ...** window. This creates a copy of your original Config, but with the new name. Both the new and original Configs are present in the Config Browser.

Closing a Config

- Menu selection **File>Close**.
- Click **File Close** in the main toolbar.

Copying elements between open Configs (or folders)

Perform the following steps:

1. Select the element you want to copy.
2. Menu selection **Object>Copy** (or the keystroke combination **Ctrl+C**)
3. Select where you want the element copied.
4. Menu selection **Object>Paste** (or the keystroke combination **Ctrl+V**).

Notes:

1. If you copy an AssemblyLine that inherits library components (Connectors, Parsers, and so forth), then these components lose their link to the library components, becoming fully instantiated instead.
2. You can move an item by using **Object>Cut** (**Ctrl+X**) to cut the selected item instead of copying it.

Config folder management

IBM Tivoli Directory Integrator comes with a recommended set of folders that match the Configuration menu selections of versions prior to 5.2 (for example, AssemblyLines, Connectors, Parsers, and so forth). You cannot rename these standard folders, nor can you delete them.

Config Folder

The first standard folder in a Config, shown in the TDI CE Config Browser is a folder called ″Config″. This folder has three items under it: **Logging**, **AutoStart** and **Tombstones**.

Logging

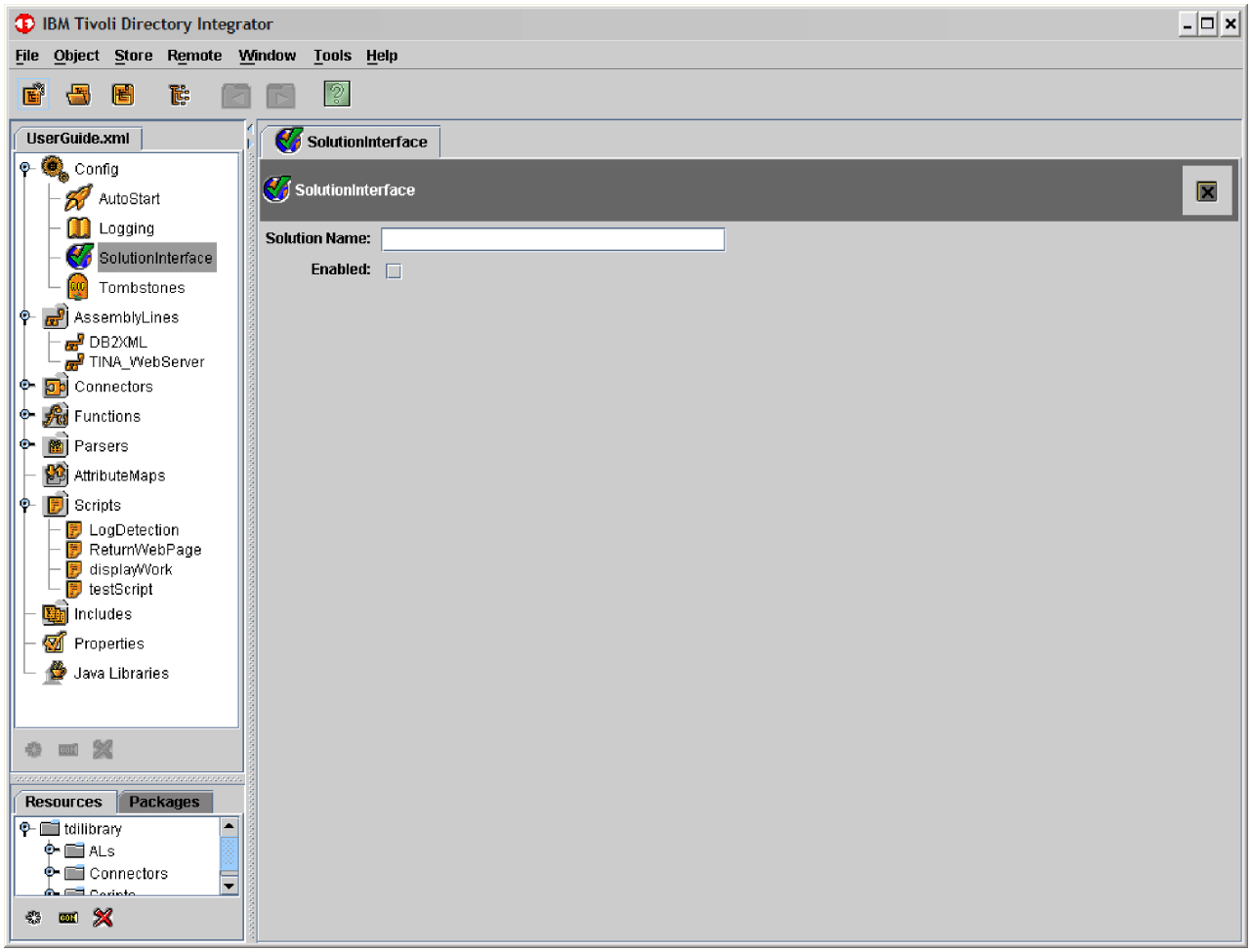
This is where you define the Config-level log appenders to be used *in addition* to any specified for an AssemblyLine (or EventHandler).

AutoStart

You drag in the ALs (or EventHandlers) that you want automatically launched whenever the Server starts this Config (unless suppressed by starting the Server with the **-D** command-line option)[11]. There is also a button row above the AutoStart list for adding and removing items.

Solution Interface

11. This function is in addition to the Auto Start Service check box found in the EventHandler Config tab.



The Solution Interface allows you to define a simple, memorable, name for a Config, and allows you to refer to the Config using this shorthand solution name in AMC/ActionManager and with the TDI Commandline Utility.

Tombstones

The new Tombstone Manager creates a tombstone for each AssemblyLine as it terminates. This marker stores a timestamp, as well as the AL exit status and other information. In this section, you define Tombstone options at the Config level. Tombstones and the Tombstone Manager are described in detail in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

Creating new elements (AssemblyLines, Connectors, Parsers, and so forth)

Select the folder under which you want to create the new element and then perform one of the following actions:

- Right-click an element, and select **New** *element*.
- Go to **Object–>New** *element*.
- Select the folder where you want to create the item and click **Insert new object** in the Config Browser toolbar.

Where *element* is the type of element you want to add, for example, an AssemblyLine, Connector, Parser, EventHandler, and so forth.

When creating a new AssemblyLine or a Script, all that is required as input in the ensuing dialog box, is a valid name. When creating other elements, you must provide a valid name as well as choose a template from the list provided.

Deleting existing folders or Config elements (AssemblyLines, Connectors, Parsers, and so forth)

Select the folder (or other element) that you want to delete and then perform one of the following actions:

- Right-click on an item and select **Delete Item**.
- Click **Delete selected object(s)** in the Config Browser toolbar.

Note: There is no undo for this operation. If you delete a folder that contains other Config elements, then these are deleted as well.

Navigating the left navigation window

You can either:

- Click the **Open/Collapse** icon next to a folder.
- Use the left and right arrow keys.

Show the details of an element in the Config Browser

Click an element in the Config Browser.

This action results in the details of the selected element being displayed in the Details Pane. For more information on the Config Browser, see

Renaming a folder or Config element

- Select an element and click **Rename selected object** in the Config Browser toolbar, or press **F2**.
- Right-click on an item and select **Rename**.

Packaging, Resources and reports

When you publish an AssemblyLine, TDI resolves all dependencies to inherited Library components, as well as processing the operations defined for this AL. Publishing is done by right-clicking on an AL in the Config Browser and then select **Publish**, which brings you to the Package Information window.

You can create self-contained solutions, called packages, that can be easily reused by other TDI users. A package hides all the component configurations and code that is necessary for a solution to function and exposes only the useful AssemblyLines configurations for users of the package. These AssemblyLines are used by way of the AssemblyLine FC, which also provides the end-user user interface for the AssemblyLine.

Resources is a place where developers keep common code and configurations to be used in TDI solutions. It is a repository of independent objects that can be reused in solutions. When a configuration object is copied from a Config into **Resources**, the dependencies of the source object are resolved before the source object is copied, so that the copied object has no inheritance other than those from the system namespace and other packages. Together with the packaging feature, **Resources** deliver a new development and deployment environment that greatly simplifies sharing of solutions and code.

Packaging

Packaging is the process of creating a self contained configuration file based on one or more AssemblyLines. During this process, all objects in the configuration

referenced by the AssemblyLines are resolved in order to create a configuration file that has no external dependencies. You select which AssemblyLines to package and TDI resolves all objects referenced by those AssemblyLines. The exception to this is when objects reference objects in the system or package namespace.

Note: You can publish a package as a solution. The package should contain the AssemblyLines, properties, and HealthAL that make up the solution. You should make your solution accessible from the Remote Server API so that the TDI Administration and Monitoring Console (or any other server API client) can provide users with information about better maintenance, administration, or configuration of solution features.

When creating a package, you are prompted to fill out a form that specifies the restrictions under which the package operates.

To create a package, perform the following steps:

1. Select the AssemblyLine or AssemblyLines you want to package.
2. Click **File**>**Publish**.
3. Enter the following parameters:

   Package ID
   A unique identifier that distinguishes this package from others.

   Description
   A description of the package and what it does.

   Author
   The author of the package

   Version
   The user-defined version of this package

   Date The issue date of the package.

   Help URL
   The URL for a help/contact Web page.

   Operations
   Displays the Operations published by this AssemblyLine.

   Resource Usage
   A generated list of resources used by the package, including the name and version of components and other packages.
4. Click **Save**.

Resources

Resources is a directory in your home directory where individual configuration objects are stored as separate files. Each file contains one single configuration object. Resources are always available in the Config Editor on the Resources tab below the Config Browser.

You can drag and drop configuration objects between the Resources area and your Configs, and this is always a copy operation. When a configuration object is dragged from a Config and into Resources, the dependencies of the source object are resolved before it is copied so that the copied object has no inheritance other than those from the system namespace and other packages.

AssemblyLines cannot be executed directly from Resources; instead they must first be copied into a config. When an item is copied from Resources into a Config, the object is copied verbatim to the configuration file (for example, no flattening or preprocessing on the item is done).

Objects in Resources can only inherit from the system namespace. Note that the system namespace always targets the TDI version used by the CE. Hence, running different versions of the CE may cause unwanted results.

Although an object in Resources can have Property references, the Resource object itself does not have TDI Properties available as this is a feature provided through a Config. As such, Property references in Resource objects will be resolved by the target Config into which the Property references are copied.

Config and AssemblyLine Reports

AssemblyLine and Config reports provide a formatted view of the selected Config or AssemblyLine. You can customize Reports with the use of stylesheet transfroms.

To create a report:

1. In the tree view of the current configuration, right-click the Config or AssemblyLine for which you want to create a report.
2. Select **Config Report.**
3. Select a report template or browse for a specific report. Report templates are located in the *<TDI_INSTALLDIR>*/XSL/ConfigReports directory and have the extension .xsl. The following report templates are provided:
   - ALOverview.xsl
   - Inheritance.xsl
   - ALTable.xsl
4. Click **Open**.

An alternate method for creating a Config report requires that you:

Right-clicking on an AssemblyLine or component in the Config Browser (or on the "AssemblyLines" folder itself). You will get a option called **Config Report**. Selecting **Config Report brings** up a File Browser dialog where you can choose which Config Report to run.

AssemblyLines

Managing AssemblyLines

AssemblyLines are created, deleted and renamed through the Config Browser (see "Config folder management" on page 134).

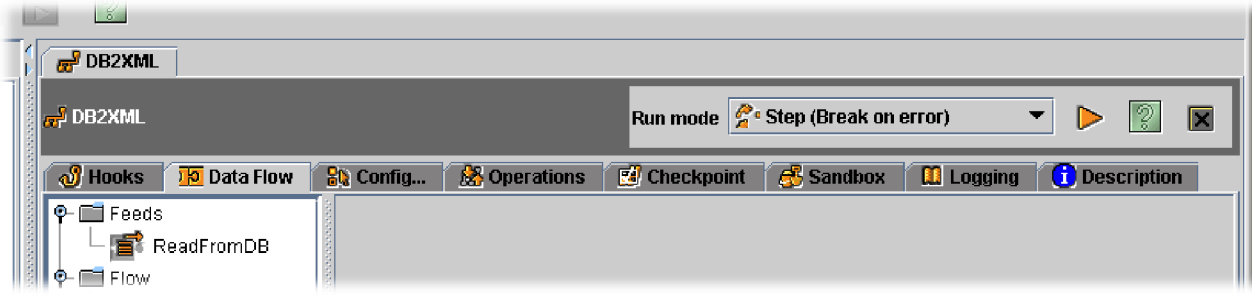
AssemblyLine configuration

Once an AssemblyLine is opened in the Details Pane, then this is where you configure and access its various features.

Below the title area are a set of Tab controls that provide you access to the following AssemblyLine features:

- **Hooks**
- **Data Flow**

- **Config ...**
- **Operations**
- **Checkpoint**
- **Sandbox**
- **Logging**
- **Description**



Hooks

This tab provides access to the various AssemblyLine Hooks (see "Hooks" on page 171). These include:

Prologs (Before Connectors Initialized and After Connectors Initialized)
This Hook, called **Prolog – Before Initialize**, is started before Connectors are initialized, and is especially useful for configuring Connectors based on parameters passed into the AssemblyLine. There is also another Hook named**Prolog – After Initialize** that executes after Connectors are initialized where you can check the status of opened connections, or fine-tune parameters.

On Start of Cycle
This Prolog is run each time the AssemblyLine starts at the beginning. For example, when the AL first begins running, or each time the Flow is cycled and control is passed back to the Feeds component (Iterator or Server mode Connector).

Epilog – Before Close
This Hook is executed when the AssemblyLine has come to an end (without errors).

Epilog – After Close
Script that is entered here is executed after the AssemblyLine stops (either successfully or in error). All Connectors will have closed down at this stage.

On Success
This Hook is executed when the AssemblyLine has terminated successfully.

On Failure
This Hook is executed when the AssemblyLine has terminated unsuccessfully, and the error condition has not been dealt with earlier.

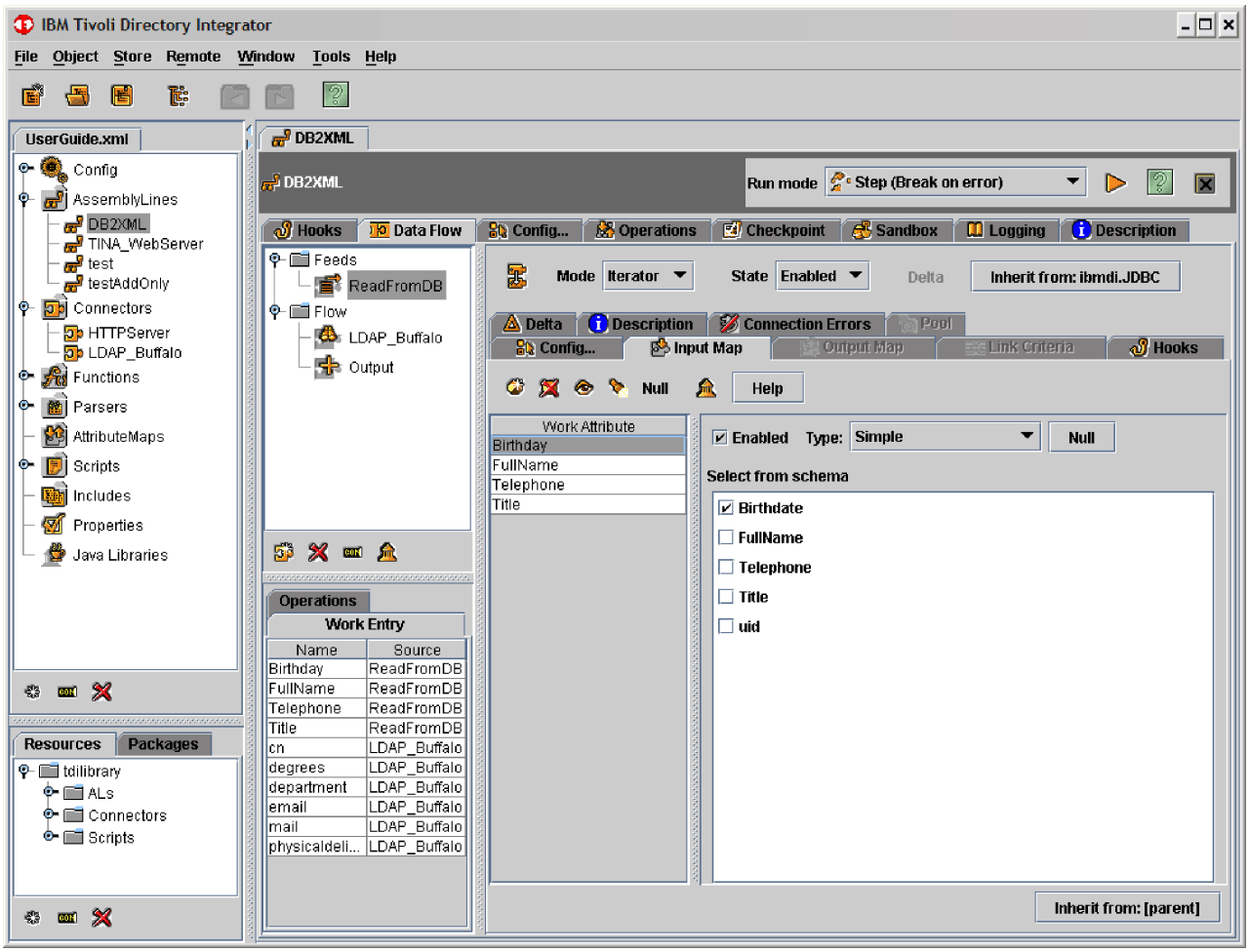
Shutdown Request
This tab lets you enter script to be started when the AssemblyLine is instructed to stop (for example, from the AMC Console, or from another server). This script enables you to add clean-up code for terminating the AssemblyLine in a controlled fashion.

For more information about the different Hooks available, see “List of Hooks” on page 71.

All of these Hooks present you with a Script Editor Window (described in “Script editor windows” on page 131).

Data Flow

Here is where you find the Connector List Control, along with their details.



Data Flow tab: The **Data Flow** tab is where Connectors and other AssemblyLine components are managed. At the top left part of this window is the Connectors List. The Connectors List consists of three sections:

Feeds The "Feeds" section houses those connectors that create *work* entries. Connectors in Iterator or Server mode end up in this section. At any given time, only one Connector in this section will create an Entry and pass it to the Flow section; it will not be touched by any other Connector in the Feeds section.

Flow Once a *work* entry has sprung from the Feed, the "Flow" section is where you would place Connectors to operate on the Entry. The Connector at the top starts first; once it has completed its processing the second Connector starts , and so on. Which Connectors are executed can be controlled by means of Flow Components: Branches and Loops.

You can change the order of the various Connectors in the respective sections by dragging and dropping them as you like. Multiple Connectors in Server mode can be active in the ″Feed″ section, and any one of them can at any given time serve up an Entry and pass it down a cloned AssemblyLine for one or more cycles, depending on the data retrieved from the connection. In the case of Iterator Connectors in the ″Feed″ section, only one of them will be active: when the AssemblyLine starts for the first time, the topmost Iterator Connector will execute and pass an Entry into the ″Flow″ section, until such time its source is exhausted; thereafter, control will pass to the next Iterator Connector.

When the last Iterator Connector has exhausted its datasource, then once the AssemblyLine has finished processing the Entry in the Flow section it will terminate. If a Connector in Server mode was configured, the "master" AL this Connector was a part of, still remains active listening for more connection initiations.

Note: The order of Connectors in this list is significant, as they are started from the top down, with Feeds components firing first (Feed components include Connectors in Iterator or Server mode) followed by the Flow section components. In addition, TDI uses the topmost Iterator in the Feeds section to read from its data source, continuing to get new entries from the Iterator until it reaches end-of-data. On the next iteration, the second Iterator in the list is started, and so on. The order of Connectors in Server Mode is not relevant (i.e., they run asynchronously and listen for events), aside from the fact that this is the order in which they are initialized when the AssemblyLine starts.

Connectors in Server mode typically open up a communications port, and then assign an instance of itself in a pseudo-Iterator mode to the port, waiting for things to happen.

The buttons below the Connectors list are:

Add Component
Here you can choose to add an AssemblyLine Connector. The AssemblyLine in these terms indicates that these components are tied to a specific AssemblyLine. You can also add a new AssemblyLine Connector by dragging one from the Config Browser.

Notes:

1. Trying to drag any component other than the Connector component in the **Data Flow** tab of the AssemblyLine gives you the following error message:

   ```
   You may only drop a Connector in an AssemblyLine
   ```

2. The Mode of the Connector determines in which section the Connector will be placed. Iterators and Server Mode Connectors will end up in the ″Feeds″ section, all others will be placed in the ″Flow″ section.

Delete selected object(s)
Removes the AssemblyLine component.

Rename selected object
Renames the AssemblyLine component.

Copy this object to the standard library
The selected component in the flow window is copied to the appropriate place in the configuration tree. This means that an AssemblyLine

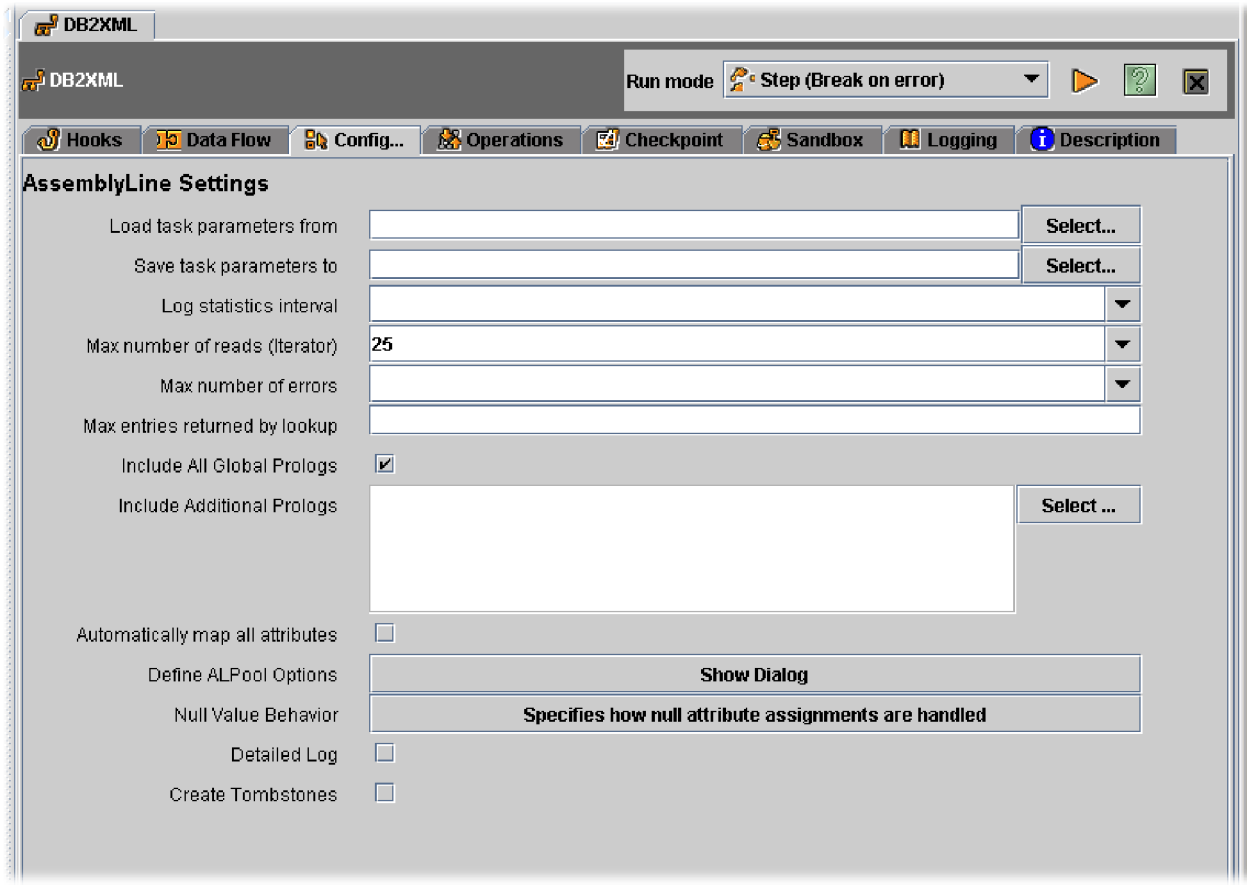
Connector is copied to the list of library Connectors under the *Connectors* folder, a Parser is copied to the *Parsers* folder and so forth

When creating a new AssemblyLine component, you have the choice of either adding an AssemblyLine Connector or a Script Block.

Just below the Connector list is a window that shows you the contents of the **Work Entry**, as well as the name of the Connector that is mapping each Attribute into the flow.

Config ...

This tab contains a number of parameters that you can use to set the operation properties of this AssemblyLine, as well as control the behavior of Iterators and attributes that are missing during Attribute Mapping.



These parameters are:

Load task parameters from

Enter the name or path of a file where you want to read parameters that were saved by this AssemblyLine (or a different one). This is one way of passing operational data between AssemblyLines, or different iterations of the same AssemblyLine.

Note: This parameter is deprecated, and this functionality will be removed in future versions of TDI. Use the facilities provided by the Properties framework instead.

Save task parameters to
Same as the **Load task parameters from** parameter, except this parameter specifies which file User specified parameters are written to after completing the AssemblyLine.

Note: This parameter is deprecated, and this functionality will be removed in future versions of TDI. Use the facilities provided by the Properties framework instead.

Log statistics interval
Here you can set a count value for how often you want the IBM Tivoli Directory Integrator server to display a progress count. This is useful for providing visual metrics during the processing of large quantities of data.

Max number of reads (Iterator)
This parameter limits the number of entries that any Iterators in this AssemblyLine passes down the data flow.

Note: If you are filtering data during Iterator input (for example, in the After GetNext Hook) such that some input objects are being skipped, the Iterator continues to read data until it has passed the specified number of entries to the next Connector, or reached the end of the input data set.

Max number of errors
Specify the error tolerance level of this AssemblyLine. If the number of errors exceeds the value entered in this parameter, then AssemblyLine terminates.

Max duplicate entries returned
Whenever a Connector in this AssemblyLine looks up information in a data source (for example, Delete, Update and Lookup mode), then this parameter limits the number of entries that are returned. If no value is specified, a default value of 10 is implied; specify 0 to return an unlimited number of entries (in fact, as many as memory will allow).

Note: This value is the total number of entries returned, so limiting this parameter to 1 (one) can cause unexpected results for Delete and Update operations. This is due to the fact that if the Connector is unable to isolate a single entry, then the results of the Delete or Update might result in multiple entries being deleted or updated, or none at all (depending on the functionality of the data source). If you limit the Max duplicate parameter to a value of 1 then you cannot detect these situations in your solution.

Include All Global Prologs
If this check box is enabled, then all Script Libraries that have the Auto include check box checked are evaluated at the startup of this AssemblyLine.

Include Additional Prologs
Here you can select which Script Libraries are to be included as additional Prologs for this AssemblyLine.

Note: You can include a Script Library with this parameter even if the Script Library is not marked for Auto include.

Automatically map all attributes
A convenience parameter that enables you to quickly set up mapping to

and from data sources that have compatible schema. If this parameter is enabled, then it works the same as if you had set up an any map (a single Attribute named *) for each AssemblyLine Connector.

Define ALPool Options
Clicking this button opens the **Define ALPool Options** dialog. ALPool stands for AssemblyLine Pool. The ensuing dialog box will present you with two input fields and a check box:

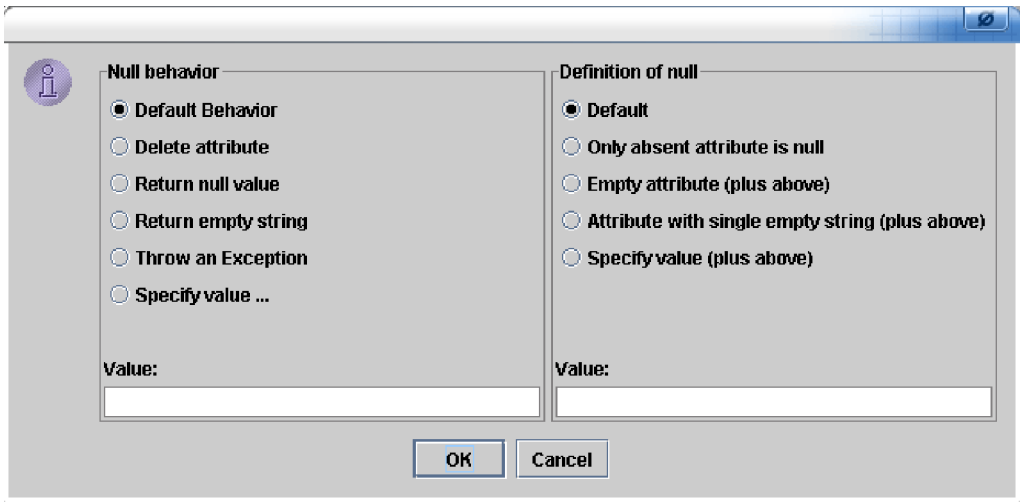
Number of prepared
Defines the minimum number of identical threads to be started by TDI, pre-loaded with the execution logic of this particular AssemblyLine. These threads are held in a pool, ready to be assigned to an actually executing AssemblyLine when a *work* entry is ready for processing.

Maximum concurrent
Defines the maximum number of threads that can exist for this particular AssemblyLine, even if there are more *work* entries created by the Entry Feed components, ready for processing.

Ignore delayed initialization settings
Ignores the "Initialization" settings for all components, and instead initializes them at AL startup. This setting can prevent a connection timing out due to excessive initialization time required by "delayed" components.

If at some stage the number of idle threads (i.e., threads not currently assigned to process *work*) is higher than **Number of prepared** , the excess will be trimmed.

Null Value Behavior
This form lets you decide how Attributes that are listed in an Attribute Map, but which are not found during mapping, are to be handled. Both what constitutes a Null Value, and how to handle a Null Value in Attribute Mapping can be defined.



You can define the *Null Behavior*:

- **Default Behavior** tells IBM Tivoli Directory Integrator to use the behavior setting defined by the global attributes `rsadmin.attribute.nullBehavior`,

rsadmin.attribute.nullBehaviorValue, rsadmin.attribute.nullDefinition and rsadmin.attribute.nullDefinitionValue, as follows:

- If rsadmin.attribute.nullBehavior is not defined anywhere, the behavior is "Delete".
- If rsadmin.attribute.nullBehavior is defined as "value", then the value defined for rsadmin.attribute.nullBehaviorValue is taken - this attribute must have a value in this case.
- If rsadmin.attribute.nullDefinition is not defined anywhere, only absent attributes count as null values.
- If rsadmin.attribute.nullDefinition is defined as "value", then the value defined for rsadmin.attribute.nullDefinitionValue is taken - this attribute must have a value in this case.

- **Delete Attribute** means that the Attribute is removed during mapping.
- **Return Null Value** means the Attribute is returned with no value (null value).
- **Return Empty String** means that the Attribute is returned with an empty string value.
- **Throw an Exception** means when an Attribute is missing in the Data source, throw an Exception.
- **Specify Value ...** lets you set a value to be returned each time this Attribute is missing.

In addition, you can define what exactly constitutes a NULL value under *Null Definition* (note that the list is cumulative):

- **Default** means IBM Tivoli Directory Integrator uses the setting in **Edit–>Preferences**.
- **Only absent attribute is null** means (missing) attributes are defined as null, no other comparisons are made.
- **Empty attribute** means empty attributes (no values) are considered null (as well as absent ones).
- **Attribute with single empty string** means an attribute containing nothing but an empty string are considered null (in addition to absent and empty Attributes).
- **Specify value** means an Attribute containing a user-specified value (in the box below) are considered to be null, in addition to all of the above.

Detailed log

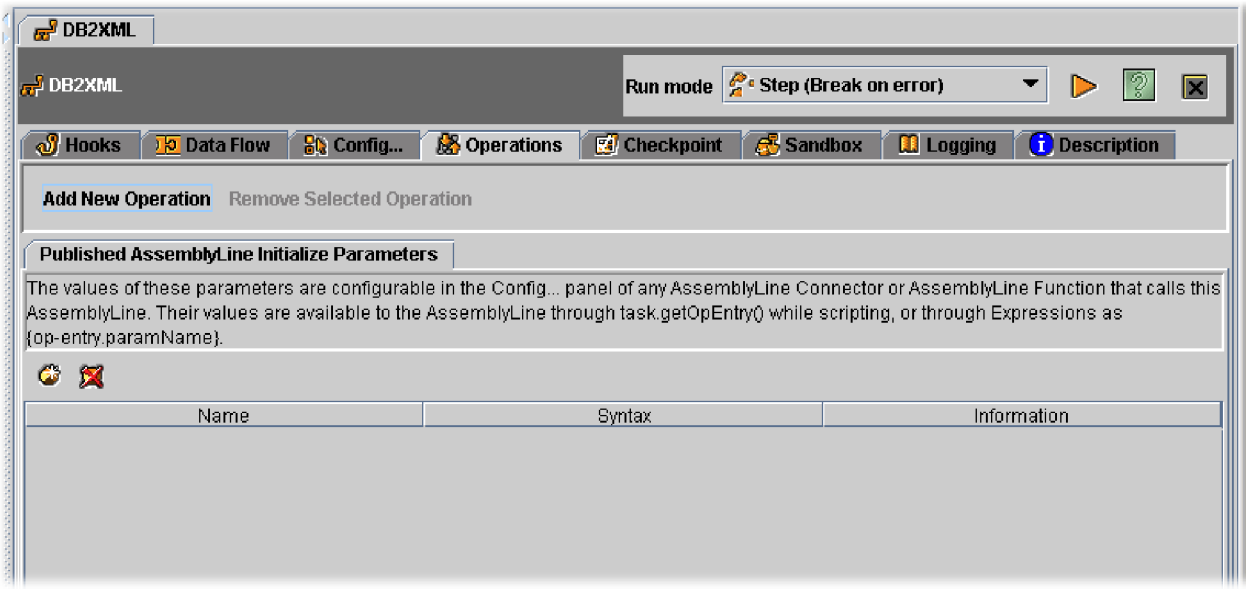
Causes IBM Tivoli Directory Integrator to log detailed operational messages that are helpful when troubleshooting or debugging an AssemblyLine.

Create Tombstones

Causes Tombstones to be created for the AssemblyLine. Tombstones and the Tombstone Manager are described in detail in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

Operations

AssemblyLine Operations allow you to implement any number of distinct functions to be performed by an AL. Each operation has an associated set of Input and Output Maps for defining both parameter values passed in when an Operation is called, as well as Attributes returned after the called AL Operation is finished.



The Operations tab is where you add and remove operations for an AssemblyLine. Select an Operation in the list in order to edit the Input and Output Maps associated with it.

Note:

> If no operations are defined, then TDI considers the AL to have a single "default" operation, allowing existing Configs to run without modification. However, if you define any AL Operations and still wish to run this AssemblyLine directly (for example, not using a call from another AL), or if you wish to define Input and Output Attribute parameters, then you must create an Operation called "Default" yourself.

There is one pre-defined tab in the Operation window (as shown in the screenshot above) and it is for defining initialization parameters for the AssemblyLine. Here you can create Attributes that can be applied to configuration parameters, for example, in the Config tab of components, as well as for Delta Engine settings and Logging.You can create these Attributes by assigning an Expression to the desired parameter that accesses the named initialization Attribute using the **Operation Entry** (op-entry). The Operation Entry is an Entry object that contains information about the Operation called, as well as any Input (or initialization) Attributes sent by the caller.

As an example, say that you have specified an Initialization Attribute called "useFilePath". You can then specify the following Expression as the File Path parameter of a FileSystem Connector: `{op-entry.useFilePath}`. Note that you cannot assign Expressions to parameters by simply entering the text into the parameter field. Instead, use the CTRL-E shortcut to open the Expression Editor, or click on the parameter label (for example, "File Path" for the above example) and then write the Expression into the Expression field of the resulting Parameter Dialog.

Each Operation that you create has its own Input and Output Schemas where you can define call/return Attributes for this Operation. In addition, you can set up Attribute Maps to determine how these parameter Attributes will be mapped to and from the Work Entry.

In addition to these initialization Attributes, each Operation that you create has its own Input and Output Schemas where you can define call or return Attributes for this Operation. In addition, you set up Attribute Maps to determine how these parameter Attributes will be mapped to and from the Work Entry.

At the top of these Attribute Maps is the standard mapping toolbar:

Add a new Attribute to the Attribute map
Adds an Attribute to an Entry list.

Remove selected Attribute from Attribute Map
Removes an Attribute from an Entry list.

Switch views
This button toggles between the Schema view, Detail View and the List view.

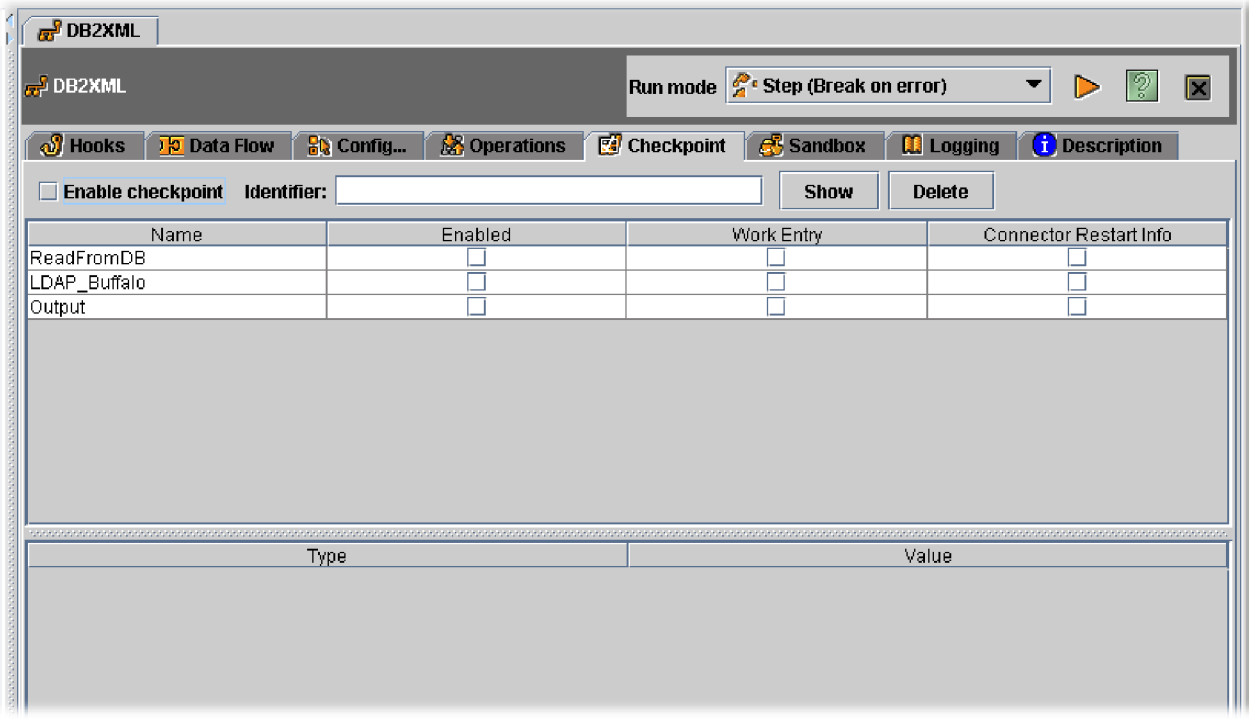
Perform quick discovery
Pre-fills the Schema according to the Work Entry as designed in the AssemblyLine.

Null Defines how to handle Null values passed in the Initial Work Entry.

There can actually be more Attributes in the Work Entry in both cases, but this is the minimum requirements list.

Checkpoint

For more information about the **Checkpoint** functionality, see "Checkpoint/Restart" on page 97.



This tab has the following parameters:

Enable Checkpoint
The master switch for the whole AssemblyLine. This must be checked for any Checkpoint/Restart functionality and any recording of information to

take place. Also, in order to be able to distinguish between different runs of the AssemblyLine, you must specify an **Identifier**.

Identifier
Identifies the AssemblyLine for subsequent Restart. If nothing is specified, a default in the form of IDI_CP_*AssemblyLine_name* is used.

The following values are available for the Checkpoint result:

Name Name of the Connector in the AssemblyLine.

Enabled
This must be checked for the Checkpoint/Restart framework to consider recording any information about this particular Connector at all. It also causes a Connector in Iterator mode to save the Work Entry just before the AssemblyLine is finished processing and moves on to the next Connector.

Work Entry
Checking the **Work Entry** check box instructs the Checkpoint/Restart framework to record the contents and state of the Work Entry before this particular Connector does its work in the AssemblyLine.

Connector Restart Info
By enabling the **Connector Restart Info** check box, you ensure that the Checkpoint/Restart records any information required for this Connector to be able to make a meaningful resumption of its processing during an AssemblyLine restart. An example of this is the position in an input file.

AssemblyLine operations

Traditionally, AssemblyLines have performed a single function. Although they could contain branches for dealing with specific situations or data values, an AL was traditionally designed to move data in one direction from a set of data sources to another set of targets. As a result, when you used the 6.0 Web services components to expose an AL as a service, multiple ALs were necessary to provide additional services.

TDI version 6.1 introduces the concept of AssemblyLine Operations, allowing you to implement any number of distinct functions to be performed by an AL. Each Operation has an associated set of Input and Output Maps for defining both parameter values passed in when an Operation is called, as well as Attributes returned after the called AL Operation is finished. This extends and replaces the single set of Call and Return Attribute Maps found for AssemblyLines in previous versions.

Once you have defined Operations for an AssemblyLine, the new Switch-Case constructs let you easily implement the logic in the AL to deal with them. Furthermore, both the AL Function (FC) and the AL Connector support AL Operation calls.

AssemblyLines with Operations can be published as “Adapters”, using the AL Publishing feature. These Adapters show up as Connectors and can easily be added to other AssemblyLines or Config Connector Libraries. If you drop the improved Web Service Receiver Server Connector into an AssemblyLine, it can generate the WSDL for the AL based on its Operations and the associated Attributes.

And, of course, AL Operations are accessible through API calls. As a result, the new Command-line Interface for TDI and the Administration and Monitoring

Console both offer features for calling specific AL operations, and for passing Attribute values between the calling AL and the called AL.

Defining AL operations: Operations are defined in the **Operations** tab of an AssemblyLine.

Calling AL operations: In addition to API calls, there are a number of Components for running and controlling AssemblyLines. These all support AL Operations, and are described in the following sections.

AssemblyLine function component: This component offers a Query parameter for retrieving Operations from the configured AL. See the AL Function Component section in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

AssemblyLine connector: The AssemblyLine-caller Connector, called the AL Connector, allows you to exploit an AssemblyLine as a single component in another AL. Furthermore, it enables the development of custom Connector Modes that represent the various operations implemented by the called AL.

See the AL Connector section in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information.

Using operations from JavaScript: Specifying the operation to call, as well as the Attribute values to pass into the called AL, are both done in the Task Call Block (TCB).

```
// Set up a new TCB and choose the "findUser" Operation.
//
var myTCB = system.newTCB();
myTCB.setOperation("findUser");

// Now I have to pass in the "uid" Input Attribute
// defined in the example screen shown earlier in this section.
// The value I want to use is in a work Entry Attribute called "userID".
//
myTCB.setOperationInitParam("uid", work.getString("userID"));

// Now I can call the AL Operation.
//
var al = main.startAL( "WS", myTCB );
al.join(); // Wait for it to complete
var resEntry = al.getResult(); // Return the resulting Attribute
```

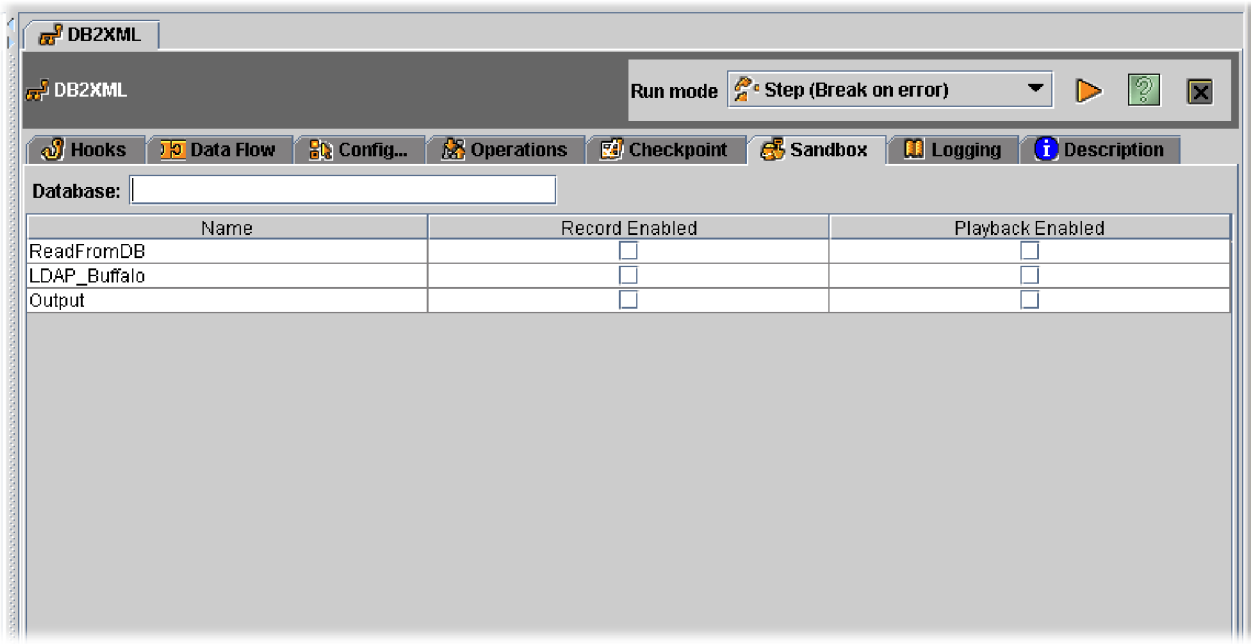
A more efficient way of calling AL Operations is by using the AssemblyLine's Manual Mode feature. If you start the AL in Manual Mode, then it start all components and returns without actually doing any processing:

```
myTCB.setRunMode("manual");
var al = main.startAL( "WS", myTCB );
```

Now you can make calls to different AL Operations each time you cycle the AssemblyLine manually with the executeCycle() method.

Sandbox

For more information about **Sandbox** functionality, see "Sandbox" on page 23.



This tab has the following parameters:

Database
URL of a database to hold Sandbox information.

For each of the Connectors in the AssemblyLine, you can set options:

Name
Name of the Connector in the AssemblyLine.

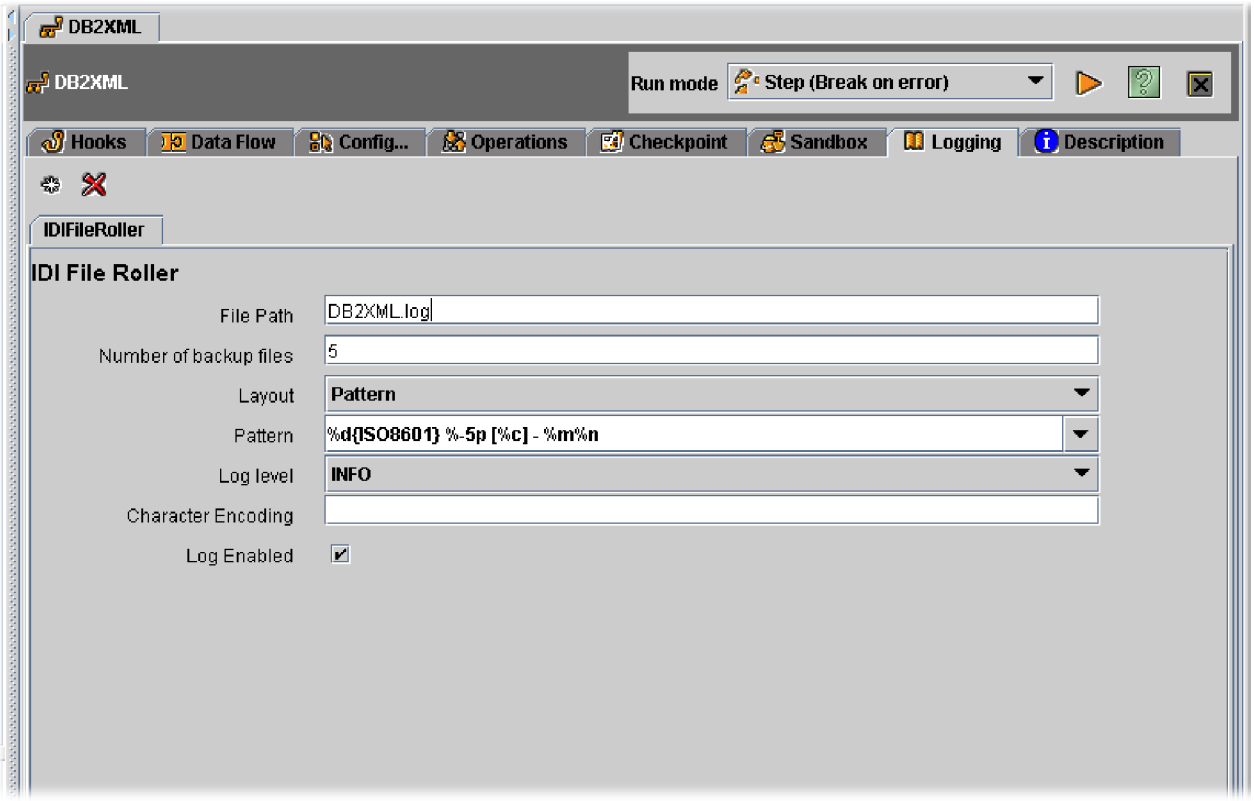
Record Enabled
If checked, record the actions of this Connector in the database.

Playback Enabled
If checked, instead of actually performing any work, retrieve the results of what this Connector did during an earlier run, and feed this into the AssemblyLine.

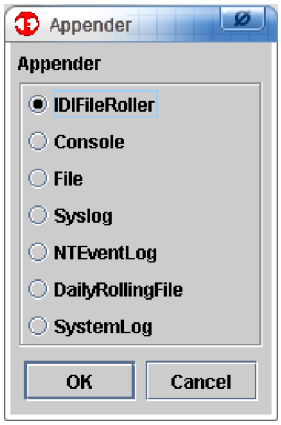
Logging

For more complete information about Logging, see "Logging and debugging" in *IBM Tivoli Directory Integrator 6.1.1: Problem Determination Guide*.

Only the parameters that describe how messages are logged are described here.



All log configuration windows operate in the same way. For each one you can set up one or more log schemes. These are active at the same time, in addition to whatever defaults are set in the log4j.properties and executetask.properties files.



For a list of the possible logging schemes, see “Logging” on page 192.

Log Levels: Log levels can be

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

DEBUG, INFO, WARN, ERROR and FATAL have increasing levels of message filtration.

Note that the IBM Tivoli Directory Integrator logmsg() (called from script, for example, `task.logmsg("Hello");`) calls log at INFO level if nothing else is specified. This means that setting loglevel to WARN or lower silences your logmsg calls as well as all Detailed Log settings.

Description

This is a free-form text field, where you can document (aspects of) your AssemblyLine. Whatever is in this field, is not considered by IBM Tivoli Directory Integrator anywhere. Additionally, individual Components have their own Description field.

Logging and problem determination enhancements

Character encoding for all file appenders

All windows showing loggers that write to encoding enabled streams have an extra parameter that lets you define the character encoding for the stream.

Custom appender support

Log4j allows logging requests to print to multiple destinations. In log4j speak an output destination is called an appender. TDI Server supports several appenders (IDIFileRoller Appender, Console Appender, File Appender, Syslog Appender, NTEventLog Appender, DailyRollingFile Appender, SystemLog Appender). The current TDI logging environment cannot be extended with additional appenders. The TDI Server allows you to add your own custom Appenders. Custom Appenders are defined with a system property in the global.properties/ solution.properties file:

```
custom.appender.<CustomAppenderName>=<CustomAppenderClass> [CustomFormUserInterfaceClass]
```

where:

- *CustomAppenderName* is the custom Appender name used in the Config Editor.
- *CustomAppenderClass* is the custom Appender Java class implementation. All custom Appenders must implement com.ibm.di.log.CustomAppenderInterface interface.
- *CustomFormUserInterfaceClass* is an optional parameter that specifies custom GUI implementation for that Appender. The custom user interface class must implement `com.ibm.di.admin.ui.CustomAppenderUIInterface`.

Here is an example for a custom Appender definition in global.properties:

```
custom.appender.customLog=com.ibm.di.log.CustomLogAppender com.ibm.di.admin.ui.CustomLogFormUI
```

For more information, see the "Logging and Debugging" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

Log4j logs folder

The default location of the logs generated by log4j changes to the “logs” folder. The change is made in both the log4j.properties file and the ce-log4j.properties file. Both the ibmdi.log and the ibmditk.log are by default placed under the “logs” directory. The log4j.appender.Default.file property in both log4j.properties file and ce-log4j.properties file is modified to change the default folder to the logs folder and then place the corresponding files under the log folder.

Miscellaneous problem determination enhancements

In order to improve the serviceability of TDI, the following enhancements have been made to 6.1:

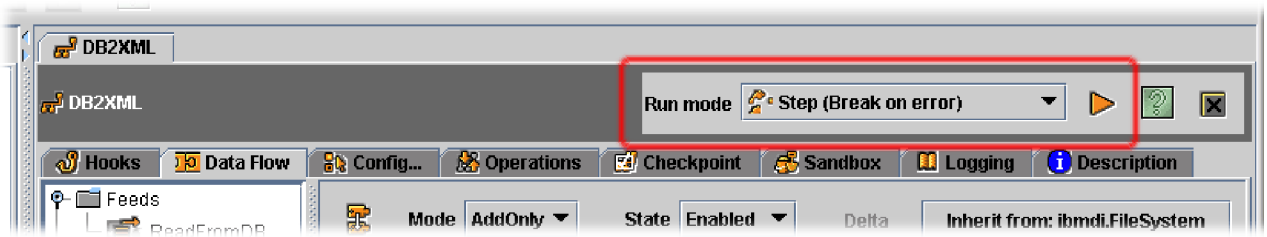
- IBM Tivoli Directory Integrator parsers include trace information.
- Log, message and trace files are stored in the following subdirectories of the common TDI installation directory:
  - Trace files are stored in CTGDI/logs
  - FFDC data are stored in CTGDI/FFDC/<*date*>
  - Message logs are stored in CTGDI/logs
- Message log file names have prefix **msg** and suffix **.log**.
- Problem Determination scripts – collect.bat (sh) and logcmd.bat (sh) – are provided in the CTGDI/scripts directory.
- Properties file jlog.properties is placed under the Tivoli_Common_Dir/CTGDI/etc directory.

An example of the defaults set for the logs and traces:

```
jlog.snapmemory.className=com.tivoli.log.SnapMemoryHandler
jlog.snapmemory.description=Memory handler used to trace to memory
jlog.snapmemory.queueCapacity=10000
jlog.snapmemory.dumpEvents=true
jlog.snapmemory.snapFile=trace.log
jlog.snapmemory.baseDir=$Tivoli_common_dir$/CTGDI/FFDC/
jlog.snapmemory.userSnapFile=userTrace.log
jlog.snapmemory.userSnapDir=$Tivoli_common_dir$/CTGDI/FFDC/user/
jlog.snapmemory.triggerFilter=jlog.levelflt
jlog.snapmemory.msgIds=*E
jlog.snapmemory.msgIDRepeatTime=10000
jlog.snapmemory.maxFiles=10
jlog.snapmemory.maxFileSize=1000000
```

Testing AssemblyLines

At the top of the AssemblyLine Details window is a toolbar.



Here you find buttons to access the following operations:

Run Mode

Here you select how this AssemblyLine is to be run. The options are:

Step (Break on error)

Step (Break on error) is the default run mode and will cause the AssemblyLine to execute normally until an error occurs, at which time the AssemblyLine Stepper/Debugger window appears, allowing you to examine data and debug the exception or error.

Step (Paused)

In this mode the AssemblyLine is initialized and then pauses in the **Stepper/Debugger**. From here you can step through AL execution, set breakpoints and watch/modify data during processing.

Standard (Run to completion)
Standard (Run to completion) is the fastest run mode and does not invoke the AssemblyLine Stepper/Debugger.

Standard (Record)
Standard (Record) allows you to record input data from Connectors during execution.

Standard (Playback)
Standard (Playback) executes a recorded AssemblyLine run.

Step (Playback)
Step (Playback) allows you to debug a recorded AssemblyLine run.

Run **Run** starts execution of the current AssemblyLine. Note that you can also run an AssemblyLine directly from the Config Editor by first selecting the AssemblyLine, and then clicking **Run** in the Main toolbar.

Help for this window
This help button opens a help page in the online documentation for the specific tab currently open in the Details window.

Close this window
Closes the AssemblyLine Details window.

Debugging

The IBM Tivoli Directory Integrator Debugger

The TDI Debugger allows you to step through an AssemblyLine, analyzing the AssemblyLine for errors at each step. AssemblyLine steps are defined by a Watch List and breakpoints.

Watch List: The Watch List displays any variables or expressions that you want to watch as your scripts execute. Whenever one of these expressions changes, its value is updated in the Watch List. Watch list items are evaluated at each breakpoint.

To add an expression to the Watch List:

1. Click **Insert** on the **Watch List** tab.
2. Perform one of the following actions:
   - Select a variable from the **Add watch variable(s)** list.
   - Enter an variable expression in the **Expression** field.
3. Click **OK**.

Breakpoints: Breakpoints are the core elements of the Debugger. They enable you to select where the TDI server will interrupt its execution of the AL for developer intervention. There is a default breakpoint at the beginning of an AL, meaning that the Debugger always breaks before executing any of the AL code.

There are four types of Breakpoints:

Static Static breakpoints are set by checking the corresponding check box in the AssemblyLine Flow window as described above, or by selecting Debug Break found in the Hooks Details windows.

Conditional
A Static breakpoint can be made **conditional** by selecting it in the Breakpoints tab (beside the Watch List) and entering a snippet of JavaScript

that evaluates to true or false in the Expressions column. Then this breakpoint will only cause the AL to pause if the JavaScript statement is true; for example, work.getString(″cn″).startsWith(″S″).

Temporary
Right-click any breakpoint in the list at the left of the Debugger window, select **Run** and **Break Here**. A temporary breakpoint is created and AL processing starts. Note that this breakpoint disappears as soon as the AssemblyLine pauses again, regardless of which breakpoint it actually stops at.

Programmatic
You can trigger a breakpoint that pauses AL execution by calling the task.debugBreak(msg) method from script (for example, in a Hook or Attribute Map). The Programmatic breakpoint also displays the string message passed to the function.

To define a breakpoint, perform one of the following actions:

- Select the check box next to a possible breakpoint in the Debug Breaks tree view.
- Select **Show all possible breakpoints** on the breakpoints tab, and select the desired breakpoints from the list.

Selecting a breakpoint in the Debug Breaks tree view selects that breakpoint on the Breakpoints tab, and vice versa.

Debugging an AssemblyLine

TDI offers an AssemblyLine debugging tool called the AL Stepper. The AL Stepper allows you to:

1. Define breakpoints for AssemblyLines.
2. Pause AssemblyLine processing at the defined breakpoints to examine the AssemblyLine for errors.

The AL Stepper is part of the Config Editor.

To debug an AssemblyLine, select the **Step (Paused) Run** mode and press AssemblyLine Run, or press the Alt-D keyboard shortcut. Running the Debugger starts the TDI Server. The server loads the Config, prepares to run the AL, and then returns control to the Debugger. At this stage you can select breakpoints and Watch List expressions, or you can begin stepping through your AL. Stepping options are made available through buttons, menus or right-click options:

Step Into
Continues execution and stops at the next breakable point.

Step Over
Continues execution until the next static breakpoint or at the next component in the AssemblyLine.

Continue
Executes until the next static or conditionally true Breakpoint.

Continue Until here
This right-click option causes execution to continue until the next static Breakpoint or the selected Hook is reached.

Once the selected task is started, the Debugger pauses processing at specified breakpoints. Whenever execution is paused, you can use the **Evaluate Command**

input field at the top of the Output Display to display information or run script. You can also examine Entry, Attribute and script variable values by adding these to the **Watch List**.

Breakpoints can be set in enabled Hooks by enabling **Debug Break** or by selecting the check box next to the desired Hook in the Debugger Breaks list. If a breakpoint is enabled in the Config Editor, the **Debug Breaks** list shows you where you are when execution pauses.

You can also set breakpoints in your scripts by using the following method:

```
task.debugBreak ( "my message" );
```

This code stops processing here and outputs the text "my message" to the log.

Note: The **Debug Break** pane cannot determine where this breakpoint is, so you must rely on the messages you write to the log.

There is another commonly used debug method:

```
task.debugMsg ( "my message" );
```

This function simply outputs a message without pausing task execution.

These methods are only evaluated when the debug-process is run. See the Javadocs for more debug-methods.

AL execution can be interrupted at any time by clicking the **Stop current debug session** button.

Logging and debugging

IBM Tivoli Directory Integrator relies on *log4j* as its logging engine. Log4j is a very flexible framework that lets you send your log output to a variety of different destinations, such as files, the Windows EventLog, UNIX Syslog or a combination of these. It is highly configurable and supports many different types of *log appenders*.

The log scheme for the IBM Tivoli Directory Integrator Server (ibmdisrv) is described by the file log4j.properties, while the console window you get when running from the Config Editor (ibmditk) is governed by the parameters set in the executetask.properties file (both found in the root directory of your IBM Tivoli Directory Integrator installation.

In addition to these system-level log parameters, you can specify the log configuration for a specific Config by adding, changing and deleting Logging Appenders under the **Config>Logging** folder in the Config Browser. Finally, AssemblyLines and EventHandlers provide a **Logging** tab that lets you tailor logging for this specific task.

In order to augment the IBM Tivoli Directory Integrator built-in logging, you can create your own log messages by adding script code in your AssemblyLine. This page describes this part of the logging process.

Logging functions (like those for debugging) are mainly provided by the **task** object.

Using the CE for remote debugging (Enable remote debugging): The Configuration Editor (CE) allows you to launch debugging remotely. From the CE,

connect to a remote TDI server. Once connected to the remote server, you can run an AssemblyLine in debug mode. You can launch remote debugging on any server supported by the TDI CE. To start the remote debugging:

1. Copy the Config files that contain the AssemblyLines you want to debug to the Configs folder so that you can access these configs remotely.
2. On your remote server, start the remote CE and Run an AssemblyLine (from the remote CE).
3. Use the remote CE to open the Config file you want.
4. Use the **Run Debugger** option to run the AL from the remote CE.

Dumping the content of an Entry object: One handy function enables you to examine the contents of any Entry object (such as **work**, **conn** and **event**):

```
task.dumpEntry(entry)
```

This call dumps the specified Entry object to the log, showing you all Attributes, Properties and operation codes stored there.

Dumping the content of an Attribute:

Dumping single value Attribute: Suppose an attribute (with the name **attr**) is the single-value attribute that you want to examine. This can be done using the task.logmsg() function:

```
task.logmsg("Dumping single value attribute:" +  attr.getName());
task.logmsg("Value  = " + attr.getValue());
```

Note: The `.getValue()` function returns only the first value of an attribute.

Dumping multiple values Attribute: Suppose an attribute (with the name **attr**) is a multiple-valued attribute and you want to display them all. In this case, you must iterate through these values:

```
var values =  attr.getValues(); // get all attribute values in new array variable
// write out the attribute name
task.logmsg ("Dumping multiple values Attribute:" +  attr.getName());

for (i=0;  i<values.length; i++)
{
  // write out each value
  task.logmsg("Value " + i + " -> " +  values[i]);
}
```

If you don't know whether the attribute is single- or multi-valued you can always use the method show previous. In fact, if you do this often, you want to create a function in your Script Library that you can reuse as needed:

```
function dumpAttribute(tsk, att)
{
   tsk.logmsg("------- Attribute: " + att.getName());

   for (var i=0; i<att.size(); i++)
      tsk.logmsg("           Value: " + att.getValue(i));
}
```

Now you can display any attribute by simply calling your function:

```
dumpAttribute(task, myAtt)
```

Dumping the state of a Connector: You can, at any time, dump the state of any of the Connectors involved in your integration process. This is the same information that is displayed in the log output of an AssemblyLine powering this Connector. The following example script displays all the information available for the Connector called `myConn`. Of course, you can use an arbitrary subset of the Connector's parameters listed below according to your needs.

```
var status = myConn.getStats();
task.logmsg("Dumping myConn status:");

task.logmsg("Number of add operations performed: " + status.numAdd());
task.logmsg("Number of delete operations performed: " +
 status.numDelete());
task.logmsg("Number of errors: " + status.numErrors());
task.logmsg("Number of get operations performed: " +  status.numGet());
task.logmsg("Number of entries ignored: " + status.numIgnored());
task.logmsg("Number of lookup operations performed: " +
 status.numLookup());
task.logmsg("Number of modify operations performed: " +
 status.numModify());
task.logmsg("Number of no-change entries: " + status.numNoChange());
task.logmsg("Number of entries skipped: " + status.numSkipped());
```

Dumping arbitrary log messages: The `.logmsg()` method lets you include any text you want to the log output. This means you can indicate in the log or console any state of the custom logic of your AssemblyLines.

```
task.logmsg("Enter your own log message here");
```

The `.logmsg()` method optionally takes a log level parameter which you can use to override the default INFO level. The legal values for log level are: ″FATAL″, ″ERROR″, ″WARN″, ″INFO″, ″DEBUG″, corresponding to the log levels available for log Appenders. Any unrecognized value is treated as ″DEBUG″.

Debugging a Script Connector (or other object) where task not available: The *task* variable gives you access to the currently executing task *(thread owner)*. However, there are situations where the *task* variable is not available to your script, for example, inside a scripted Component (such as a Script Connector or Parser). This is an important object for many script operations, so you want to set up your own *task* variable with the following code:

```
task = java.lang.Thread.currentThread()
```

Similarly, the **main** can always be accessed through:

```
main = connector.getRSInterface()
```

Working with AssemblyLine files before processing is completed

Files are closed only when the AssemblyLine ends. However, you can force a Connector to close and reinitialize. The following snippet of code creates a new file each time it is started. The filenames are file1.xml, file2.xml and so forth (assuming the variable *iteration* was initialized to 0).

```
iteration++;
// close the file associated with the Connector named xml
xml.connector.terminate();
// Associate a new filename to the Connector parameter filePath
xml.connector.setParam("filePath","c:/tmp/file" + iteration + ".xml");
// reinitialize the Connector
xml.connector.initialize(null);
```

This code can be put wherever you want, even within the Connector itself.

If you have opened the AddOnly Connector in **append** mode, the setParam() is not necessary, but for **output** mode the sequence terminate() and initialize() can cause you to lose previous work.

AssemblyLine Reports

To create a formatted view of an AssemblyLine, see “Config and AssemblyLine Reports” on page 138

Connectors

Connector management

Connectors can be created two ways:

- Directly in the Config (also known as Library Connectors. See "Config folder management" on page 134).
- In an AssemblyLine (also called AssemblyLine Connectors. For more info on how to add Connectors to an AssemblyLine, see "Data Flow tab" on page 140).

Or, if you want to add a pre-configured Connector in your Config to an AssemblyLine, simply drag the Connector from the Config Browser to the AssemblyLine's Connector List (in the **Data Flow** tab). Note that the order in which the Connectors are listed is significant.

Using Connectors in AssemblyLines (AssemblyLine Connectors)

When you create a new Connector, you must enter the following settings to complete its setup:

Set Mode
If you create an AssemblyLine Connector (click **Add new Connector** in the **Data Flow** tab), then mode is set in the **Add Connector** dialog. However, if you create a Connector directly in the Config Browser, you can set the mode of the Connector at the top of the Connector Details window.

State You can choose between **Enabled**, **Passive** and **Disabled**. The default is Enabled.

Set Type
When you create an AssemblyLine Connector, you can select the type directly in the **Add Connector** dialog. If you create the Connector directly in the Config Browser (or if you want to change the type of Connector), use the Inheritance dialog which is displayed when you click **Inherit from: ...** at the top of the Connector Details window.

Note: If you want to connect a Parser to a Connector, then this is also done through the Inheritance configuration dialog.

Config...
On this tab you set the parameters that are specific to the type of Connector selected. At the top of this tab is the **Initialize** menu for controlling when this component is initialized; the default is **At Startup**.

Input/Output Map (Attribute Map)
On this tab you set up the Attribute Map for this Connector.

Note: Only one of these tabs is active at any time, depending on the Mode of the Connector (with the exception of CallReply Mode which uses both).

Delta This tab is for setting up the Delta feature that is available for Connectors in Iterator mode.

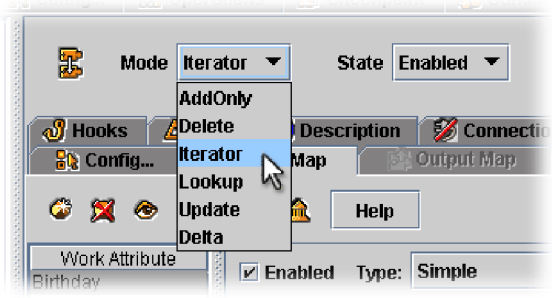
Pool This tab is disabled unless the feature is inherited from a Connector in your Library that is set up for Global Connector Pooling. Then you can enable or disable the participation of this connector in this global pool, as Exhausted Pool behavior (*Wait* or *Fail*).

Description
This tab consists of a free-form text field where you can enter documentation regarding this particular Component.

Setting up a Connector

The Connector Details pane provides two selection boxes for setting or changing the mode or state of a Connector.

Most IBM Tivoli Directory Integrator components are bi-directional, and in the case of Connectors this means that they can perform both read and write operations on the connected data source. However, each Connector in an AssemblyLine must be assigned a specific role for that particular data flow. This is done by setting the Mode of the Connector. For more information about Connector modes, see "Connector modes" on page 26.



Connector modes are (only the ones valid for this particular Connector will be shown):

AddOnly
This output mode tells IBM Tivoli Directory Integrator that this Connector is adding new information only to the source, for example, writing to a text file. AddOnly mode provides access to the Output Attribute Map.

Delete A Connector in Delete mode uses its Link Criteria to find a specific entry in the data source and then delete it. Delete mode provides access to the Input Attribute Map, enabling you to examine the data to be deleted, making the decision at the Attribute level.

Iterator
Iterator mode means that the Connector gets a view into the connected source and then returns one entry at a time for processing in the AssemblyLine. For example, a JDBC Connector which is linked to an SQL database first fires off a SELECT statement, and then returns all the records in the result set, one at a time. Iterator mode provides access to the Input AttributeMap.

Note: One of the key characteristics of IBM Tivoli Directory Integrator is that it processes information one entry at a time.

Lookup
Lookup mode is used to find data in the connected data source that matches some attributes in the data that is in the flow already. The Connector then aggregates (also called a join) this information into the flow. Lookup mode provides access to the Input Attribute Map.

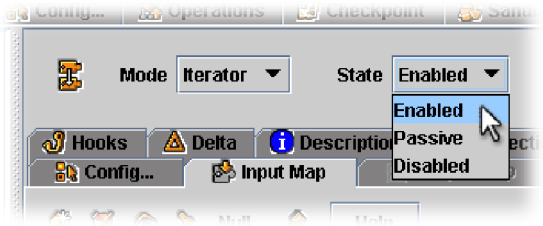
Update
In Lookup mode, the Connector first performs a find operation (just as in Delete and Lookup modes). If matching data is found, then the Connector performs a modify operation on this entry. If a match is not found, then the entry is added to the data source. Update mode provides access to the Output Attribute Map.

CallReply
CallReply mode is used to make requests to data source services (such as Web services) which require you to send input parameters and then return the information you request. Unlike the other modes, CallReply provides access to both Input and Output AttributeMaps.

Server This mode is used to subscribe to certain events from client systems. Once a notification from a client system is received, the Server mode connector generates a clone of itself as an Iterator, and feeds received Entries into the AssemblyLine. When there are no more entries from the client, the clone thread dies (or is returned to the AssemblyLine Pool) and the Server mode Connector becomes dormant again. See also “Server mode” on page 33.

Delta This mode is a combination of Update mode and Delete mode. It is dependent on delta information generated by either the Iterator Delta Store feature (Delta tab for Iterators), or Change Detection Connectors like the IDS/LDAP/AD/Exchange Changelog Connectors, or the ones for RDBMS and Lotus/Domino Changes.

In addition to these modes, Connectors also have three states:



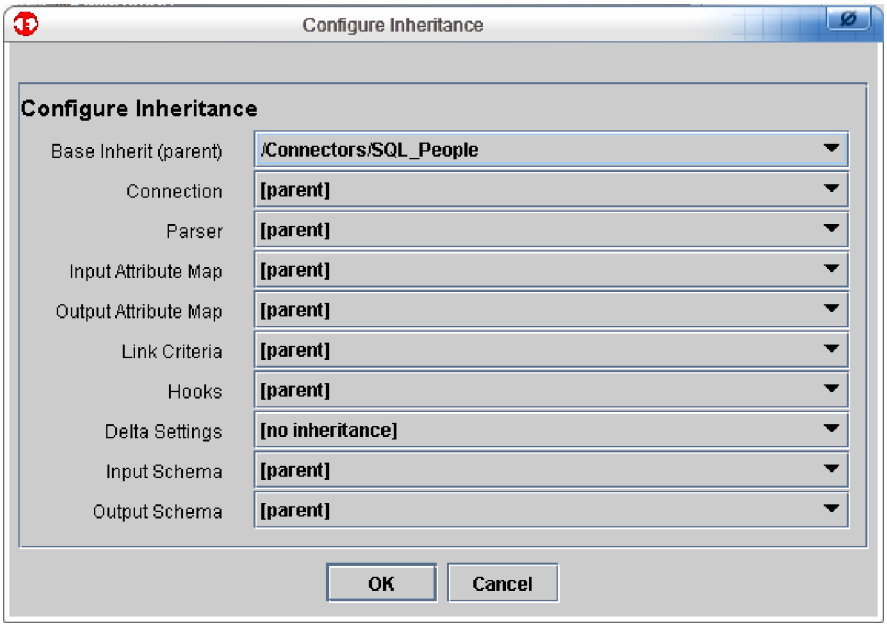
These states have the following meanings:

Enabled
This is the standard (and default) mode for all Connectors and means that the Connector is initialized and operating as usual.

Passive
If you set a Connector to Passive mode, then it is initialized at AssemblyLine startup, but is not started during AssemblyLine operation. Instead, you can invoke the Connector from your scripts.

Disabled
In Disabled mode, a Connector is neither initialized nor run during AssemblyLine execution. This mode is often used during troubleshooting in order to simplify the solution while debugging, helping you localize any problems.

In order to set or change the type of an inherited Connector, use the **Inheritance configuration dialog** button, which results in the following dialog:



Here is where you can set or modify the following inheritance values:

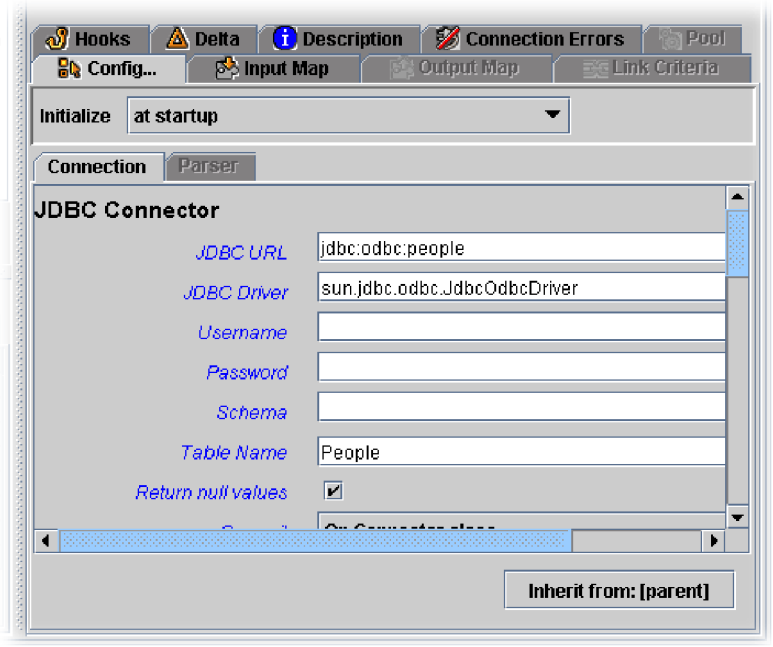
- Inheritance (base), that is, parent; for example, which basic template or library object from which parameters are to be inherited.
- Connection to the data source itself. This is for situations where the data source only supports a limited number of connections (for example, an IP port, or a seat license issue for a database). In this way, several Connectors in an AssemblyLine can share the same connection.
- Parser to be associated with this Connector.
- AttributeMaps (input and output).
- Link Criteria.
- Hooks.
- Delta Settings.
- Schema (Data Source Schema), which is available for building AttributeMaps.

Even though you have set inheritance for an item in the previous list, you can modify all or part of the inherited values. One prime exception is with AttributeMaps. You cannot remove any Attributes from an inherited map, although you can disable them.

Note: If you set an Inheritance parameter to [parent] then this means you want to inherit this value from whatever is set as the Base Inherit (parent) in the topmost list. By setting [parent] for all these parameters, except for the Base Inherit, then you can base this Connector completely on another component.

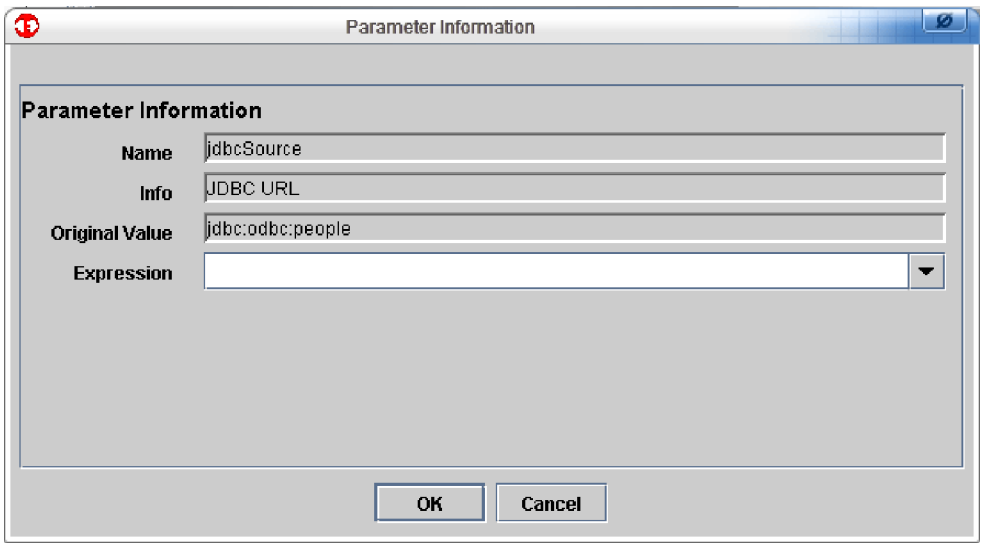
Configuring a Connector

Configure a Connector in the **Config ...** tab, where you find a set of parameters specific to the type of Connector you are working with.



Although the actual content and layout of this dialog differs from Connector to Connector, each type includes the **Detailed Log** check box. If this is selected, then IBM Tivoli Directory Integrator includes detailed operational messages in the log file which are a valuable aid when troubleshooting or debugging an IBM Tivoli Directory Integrator solution.

Note: Any of these parameters can be set directly from your scripts by using the `setParam()` call. In order to discover the name of any parameter, simply click on the label in the Configuration dialog (this feature is actually available in most dialogs in IBM Tivoli Directory Integrator.) You are presented with a Parameter Information dialog:



The first field here is the actual internal name of the parameter (to be used in the setParam() call).

Next is a description or help text about the meaning of the parameter.

The third field is what the value of the parameter would be if inherited from the parent object—if it was not overridden in the dialog box.

The last field allows you to assign an External Property to this parameter. The list lets you choose from currently defined External Properties.

Note: Each parameter has a unique Parameter Information dialog.

Setting up the Attribute Map

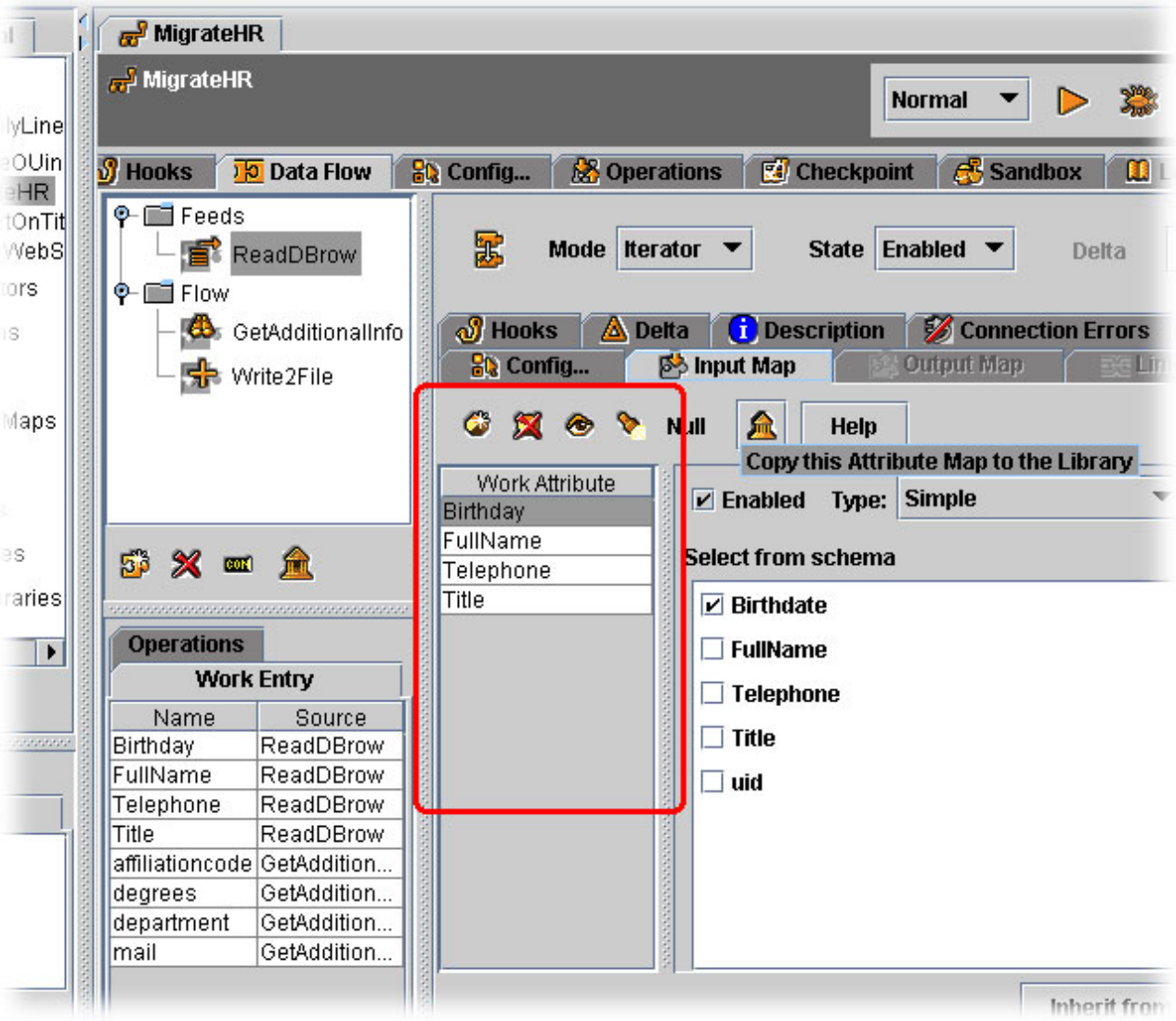
The Attribute Map is available under the **Data Flow** tab, and is a specification of which attributes are to be moved between the data flow and the Connector.

A Connector has two maps, one for **Input** and one for **Output**. However, usually only one of these maps is available at any one time, depending on the mode of the Connector:

- **Input Map** for Iterator or Lookup modes.
- **Output Map** for AddOnly, Update or Delete modes.
- **CallReply** mode: both.

Note: CallReply is the exception. It has both Input Map and Output Map available.

For more details on Connector modes, see "Connector modes" on page 26.



The Attribute Map list is a List Control (described in "List controls" on page 125). You can manipulate the list in a number of ways:

- **Add a new attribute to the Attribute Map** button in the Attribute Map toolbar.
- **Remove selected attribute from the Attribute Map** button in the Attribute Map toolbar.
- Drag one or more Attributes from the Connector Schema window.
- Drag one or more Attributes from the Work Entry window.

Note: If you want to drag an Attribute into the Attribute Map, you must drop the Attribute onto an existing item or the title bar.

The remaining buttons in the Attribute Map toolbar are:

Switch between List, Detail and Schema view
This button switches the display between the selection list, the Attribute Map details or the Schema view window.

Whenever you select an Attribute in the Work Attribute list, this replaces the Connector Schema window with the Attribute Map Settings window.

Use this button to bring the Schema window back again. You can also do this by pressing **Ctrl** while clicking an Attribute (**Ctrl**+click *Attribute*).

Perform quick discovery of schema

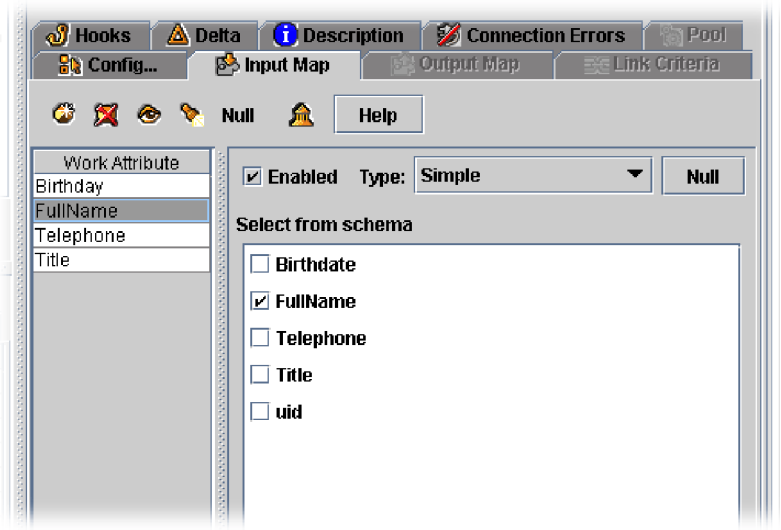
Clicking this button results in the following actions:

1. Connector connects to the data source.
2. Connector performs a Next operation.
3. Connector closes the Connection.

Note: This might not be sufficient to discover all the necessary Attributes in the data source. For example, for an LDAP directory, the first entry read (by the GetNext operation) might not contain all the desired Attributes (it might not even be of the correct class).

Null This brings up the Null Value behavior configuration dialog, and the definitions you make here become the default for this particular Input Map for this Connector. The choices here are similar to, but take precedence to those that are set for an AssemblyLine.

Whenever you select an Attribute in the Attribute Map, then the Connector Schema window is replaced by the Attribute Map Settings window.



At the top of this window are two check boxes, and another Null Value button:

Enabled

If you deselect this box, then this Attribute is not mapped during Attribute Mapping.

Type This menu allows you to specify how this Attribute map is to be carried out. The options are:

- Simple - Simple type means that the mapping is done by copying values from the selected Attributes in the list.
- Advanced (JavaScript) - Advanced mapping, on the other hand is done by executing JavaScript, where values can be computed. For example:

```
ret.value = work.getString("FirstName") + "." + work.getString("LastName") + "@acme.com";
```

- Expression - Allows you to define a TDI expression for evaluation. For example:

```
{work.FirstName}.{work.LastName}@acme.com
```

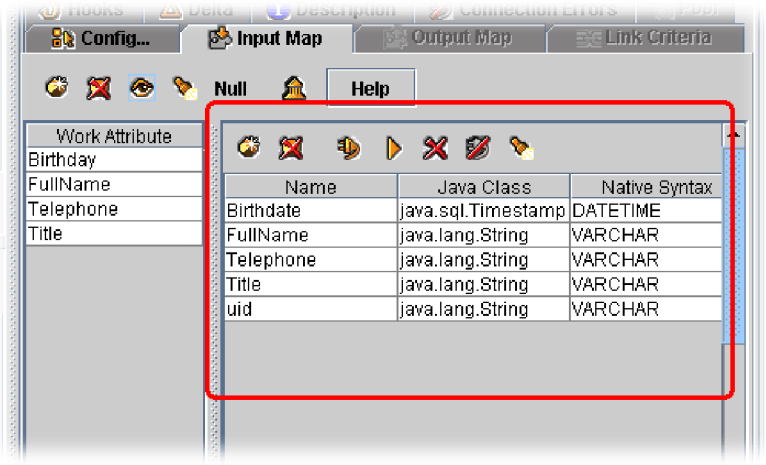
Null This button brings up the Null Value behavior button, similar to the one discussed before for the AssemblyLine level and the Connector level. The button here configures the behavior for the currently selected Attribute in the Work Attribute list, and this overrides what is set for the Input Map, above.

Note: This is the most detailed setting of Null Value behavior, at the single Connector Attribute level.

Below this line is the Connector Schema checklist. Here you can select any number of items that you want returned as the value of the Attribute currently selected in the Attribute Map list.

Note: You can switch back to the Connector Schema Window again by clicking on the **Toggle View** button described previously, or by pressing **Ctrl** while clicking an Attribute in the Attribute Map (**Ctrl**)+click *Attribute*).

Discovering datasource schema: The **Input Map** tab shows you a List Control where you can create the datasource schema by hand, or have the Connector discover it for you:



The buttons at the top of this list provide you access to the following functions:

Add an attribute to the schema
Adds a new attribute to the schema.

Remove selected attribute
Removes a selected attribute from the schema.

Connect to the data source
Connects to the data source.

Read the next entry
This button reads the next entry and populates the list with the attributes found in that object.

Note: This does not erase previously listed attributes. To do this, you must select those attributes and click **Remove selected attribute**.

Remove all schema attributes
Removes all schema attributes.

Close the connection to the data source
Closes the connection.

Discover the schema of the data source
Pressing this button requests the schema of the data source. For example, with an LDAP Connector you get the full list of Attributes defined for the objectClasses associated with the currently displayed Entry. If you are connected to an RDBMS, then the JDBC Connector retrieves the Columns definitions for the currently selected Table or view. This function might not be supported by all data sources. If that happens, then the only option is to **Connect** and use the **Read the next entry** button.

In addition, you can edit directly into some of the columns of this grid list. Schema list fields have the following meanings:

Name This is the name of the Attribute that is displayed in the *conn* object (local storage object for the Connector's Data Source Adaptor) when this data is read. It is also the name the Connector uses during write operations.

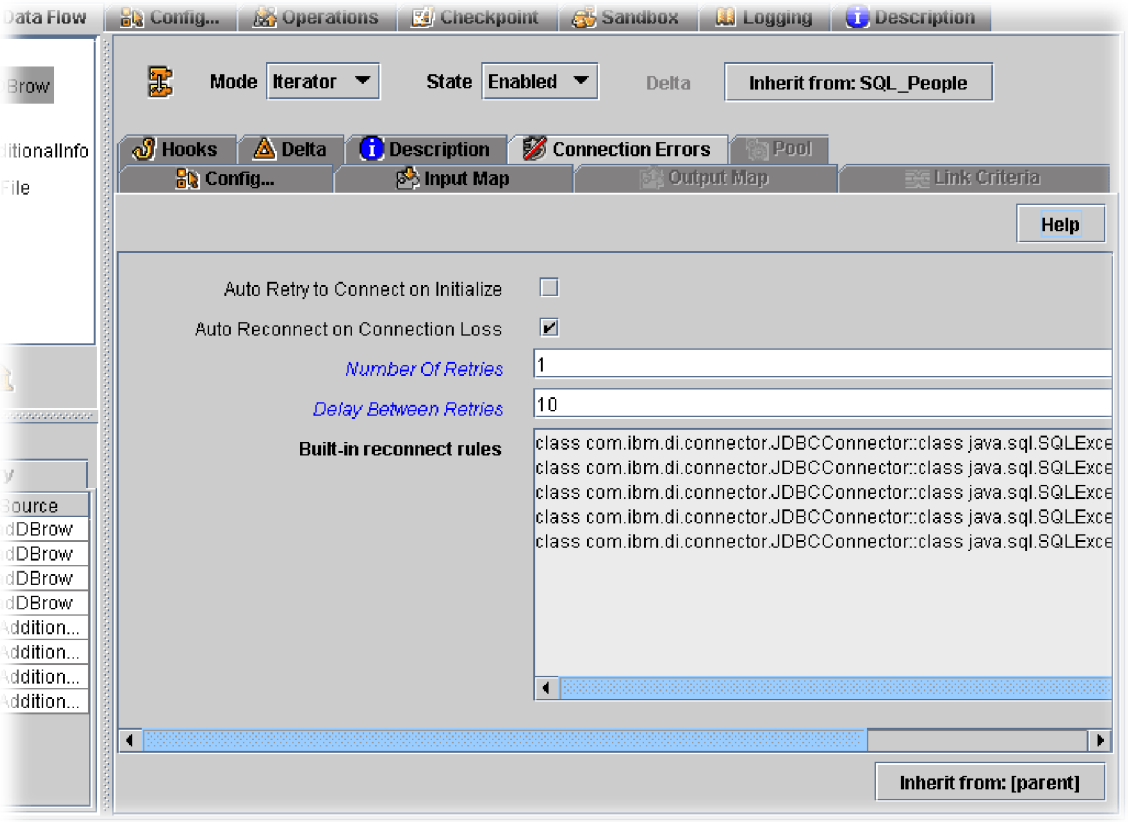
Java class
This is the type of Java object that IBM Tivoli Directory Integrator represents this Attribute as. This field is read-only.

Native syntax
This column specifies the data source-specific type of this Attribute. This field is read-only.

Sample
Here you see a sample value for this Attribute, read from the data source after you pressed the **Read the next entry** button.

Connection Errors

You can make solutions built with IBM Tivoli Directory Integrator a little more robust by telling a Connector to automatically attempt to reconnect when the connection fails. You do this in the Connector's **Connection Errors** sub-tab:



The parameters are:

Auto retry to connect on initialize
If enabled, specifies that reconnect behavior also occurs for initialization errors. Check to enable.

Auto reconnect on connection loss
The master switch for the reconnect functionality for this Connector. Check to enable.

Number of retries
The successive number of times the Connector will try to re-establish the Connection, once it fails. The default is 1. When the number of retries is exceeded, an exception is thrown. Note that after a successful reconnect, the counter we use to track retries is reset to 0.

Delay between retries
The number of seconds to wait (in seconds) between successive retry attempts. The default is 10 seconds.

Built-in reconnect rules
Displays the regular Expressions used for detecting connection exceptions (and triggering Auto-Reconnect functionality). Contents will vary from Connector to Connector, and can be extended by editing the `etc/reconnect.rules` text file.

Inherit from
Click this button to configure inheritance for the reconnect object.

Connector Pooling

TDI version 6.0 introduced the concept of AssemblyLine pooling as a feature of the new Connector Server Mode. As of TDI 6.1, you can also define global pools of Connectors that can be shared between AssemblyLines in the same Config.

Connector pooling allows you to increase overall performance and scalability. The pool represents a set of configured, instantiated and initialized Connectors associated with a Config Instance that AssemblyLines from that Config Instance can use. When an AssemblyLine is using a Connector from the pool, it will not initialize the Connector because it is already initialized. You can also specify various pool parameters like minimum and maximum pool size.

Notes:

1. Server Connectors are not usually candidates for Connector pooling.
2. The following Connector Hooks will not be executed when a Connector is created and initialized in the Connector Pool:
   - Before Initialize
   - After Initialize
   - Before Close
   - After Close

To define a pool:

1. Click **Pool** on the configuration window of a connector.
2. Enter the following parameters:

   Enable Pooling
   If **Enable Pooling** is selected, a pool is created for the Connector. All other parameters on the **Pool** tab are taken into account only when **Enable Pooling** is selected. If **Enable Pooling** is not selected, a pool is not created for the Connector and AssemblyLines can only use the configuration of this Connector.

   Min Pool Size
   Specifies the minimum number of Connector this pool will hold. A default value of 0 is assumed when this field is empty.

   Max Pool Size
   Specifies the maximum number of Connector that this pool can hold. This field is required when “Enable Pooling” is checked.

   Purge Interval
   Specifies the time interval in seconds on which the Connector Pool will be regularly reduced to its minimum size. If a value of 0 (zero) is specified, the Pool will never be reduced automatically. A default value of 0 is assumed when this field is empty.

   Number of Attempts to Initialize
   Specifies how many attempts to initialize the Connector will be made if errors occur during initialization. A default value of 1 is assumed when this field is empty.

   Sleep Interval between Initialize Attempts
   Specifies the sleep interval in seconds between consecutive attempts to initialize the Connector. A default value of 0 is assumed when this field is empty.

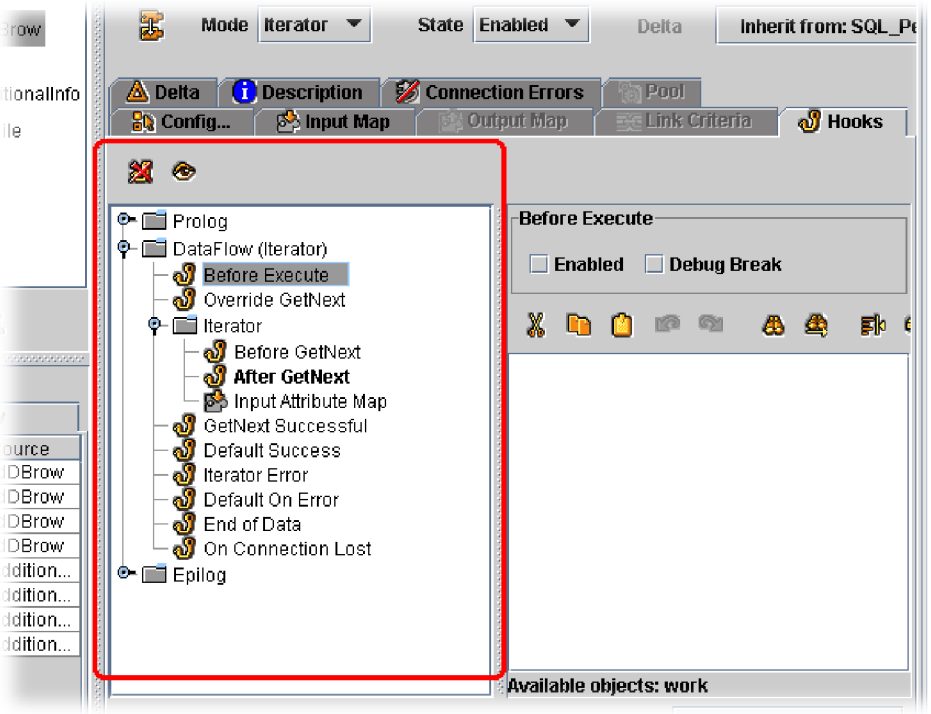
Enhance connector initialization failure handling

Although the Reconnect feature was introduced in TDI version 6.0, it did not handle initial Connector initialization failures as desired. In IBM Tivoli Directory Integrator, the Reconnect feature includes a check box in the **Connection Errors** tab. The check box tells the TDI server to use the Reconnect parameters for initial connection failure in addition to being able to use the Reconnect feature for connection loss during AssemblyLine operation. In addition, the actual exceptions that should be handled by Connection Loss behavior are customizable. By default, rules for defining which errors trigger this behavior are coded into Connectors themselves.

In addition, you can define additional rules to complement or override built-in ones.

Hooks

The **Hooks** tab presents you with a list of waypoints in the execution of a Connector where you can add your own scripts to be started. Here you have access to the Connector Hooks, Hooks that are common to all connectors, and Hooks that are specific to their own mode settings.



The Hooks toolbar contains the following buttons:

Delete selected Hook
Clicking this button with a Hook selected deletes the underlying script.

Toggle between editor and Hook tree-view
This swaps the display between the tree-view and the Hook editor view, giving you more window space in which to write and edit your scripts.

Above the Script Editor window for Hooks are two check boxes:

Enabled
When you enter a script in a Hook, this box is automatically checked. By disabling a Hook, you tell IBM Tivoli Directory Integrator not to start it. This also works for inherited Hooks.

Debug Break
If you check this box, then when you run this AssemblyLine in the debugger, IBM Tivoli Directory Integrator stops execution at this point.

Hooks are divided into three sections:

Hooks Here you can add JavaScript to augment or override the built-in behaviors of your Connector (based on its Mode setting).

Data Flow (*connector_mode***)**
This section contains some common Hooks, such as Before Execute and Override *mode_operation* (such as **Override Lookup** for a Connector in Lookup mode, or **Override GetNext** for Iterator mode). There are also **Default Success** and **Default On Error** Hooks. In addition, there are several mode-specific Hooks.

Connection Errors
This tab lets you enable and configure the automatic reconection feature.

In most cases, the order of the Hooks in this tree-view represents the order in which they are evaluated.

Hooks can be inherited from other Connectors. However, you can still override the inherited scripts by entering some script in the Script Editor window.

If you want to remove this overriding script and revert back to inherited script behavior, simply delete the Hook by selecting it and click **Delete selected Hook**.

Delta

This tab is only available for Connectors in Iterator mode, and is where you set up the parameters for the Delta feature. See "Deltas" on page 80.

Enabling Delta causes the Connector to create and maintain a local repository of all data iterated, enabling it to determine if data is changed, added or deleted (for example, gone missing) between each run.

The Delta tab has the following parameters:

Enable Delta
This tab is for setting up the Delta Engine feature that is available for Connectors in Iterator mode.

Pool This tab is disabled unless inherited from a Connector in your Library that is set up for Global Connector Pooling. Then you can enable or disable this Connector's participation in this global pool, as Exhausted Pool behavior (Wait or Fail).

Unique Attribute Name
Specify a unique attribute to use as the Delta key.

Delta Driver
Specify the Delta storage type (database). Currently you can choose between CloudScape and BTree. The usage of BTree is deprecated, and may disappear in future versions of ITDI.

Delta Store
The name of the table in the System Store into which the Delta objects are stored. For BTree Delta files, the file path of the Delta database.

Read Deleted
If selected, then deleted entries are returned to the AssemblyLine. Deleted entries are only returned when iteration of the data source has completed successfully.

Remove Deleted
If selected, then deleted entries are removed from the AssemblyLine after they are read.

Return Unchanged
If selected, then entries that are unchanged are returned to the AssemblyLine, along with new and modified entries.

Commit
From the menu, select the criteria for when information is recorded in the Delta database. The options are:

Faster Algorithm
If enabled, do not write unchanged entries to delta data base, instead remember keys in memory.

After every database operation
Snapshots are committed to the System Store immediately as they are computed during the iteration. This is the default (and backwards compatible) option.

On end of AL cycle
Wait with the commit until the AL completes the current cycle. This is the recommended setting.

On Connector close
The commit is delayed until the AL is finished and Connectors closed. Although delaying commit until the end of AL processing can boost performance, it can also result in situations where some changes have been successfully propagated, and yet the Delta Engine snapshots do not reflect this.

No autocommit
Commit of snapshot must be handled manually through script with the commitDeltaState() method of the Connector in Iterator Mode:

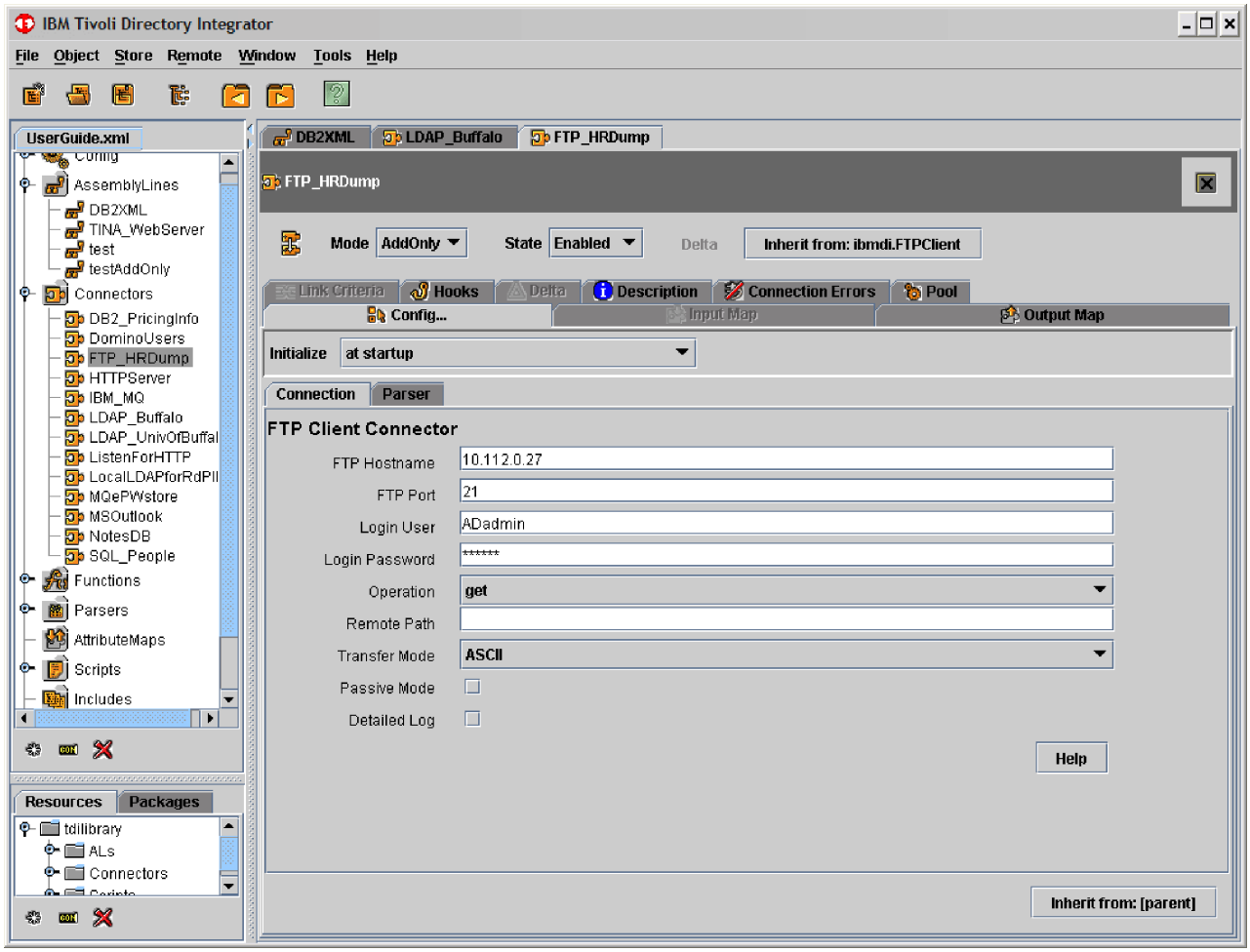
```
myIterator.commitDeltaStore();
```

Description
This free-form text field is similar to its counterpart in the AssemblyLine, except it is meant as a documentation field for this particular Component only.

Library Connectors
In addition to adding Connectors to AssemblyLines, you can also create Connectors directly in the Config Browser. These are available for use in AssemblyLines, and changes made to these library components are reflected in the AssemblyLines using them (unless you override the inherited properties).

The same features outlined in “Using Connectors in AssemblyLines (AssemblyLine Connectors)” on page 159 for AssemblyLine Connectors are available when working with Library Connectors. The main difference is that the Connector

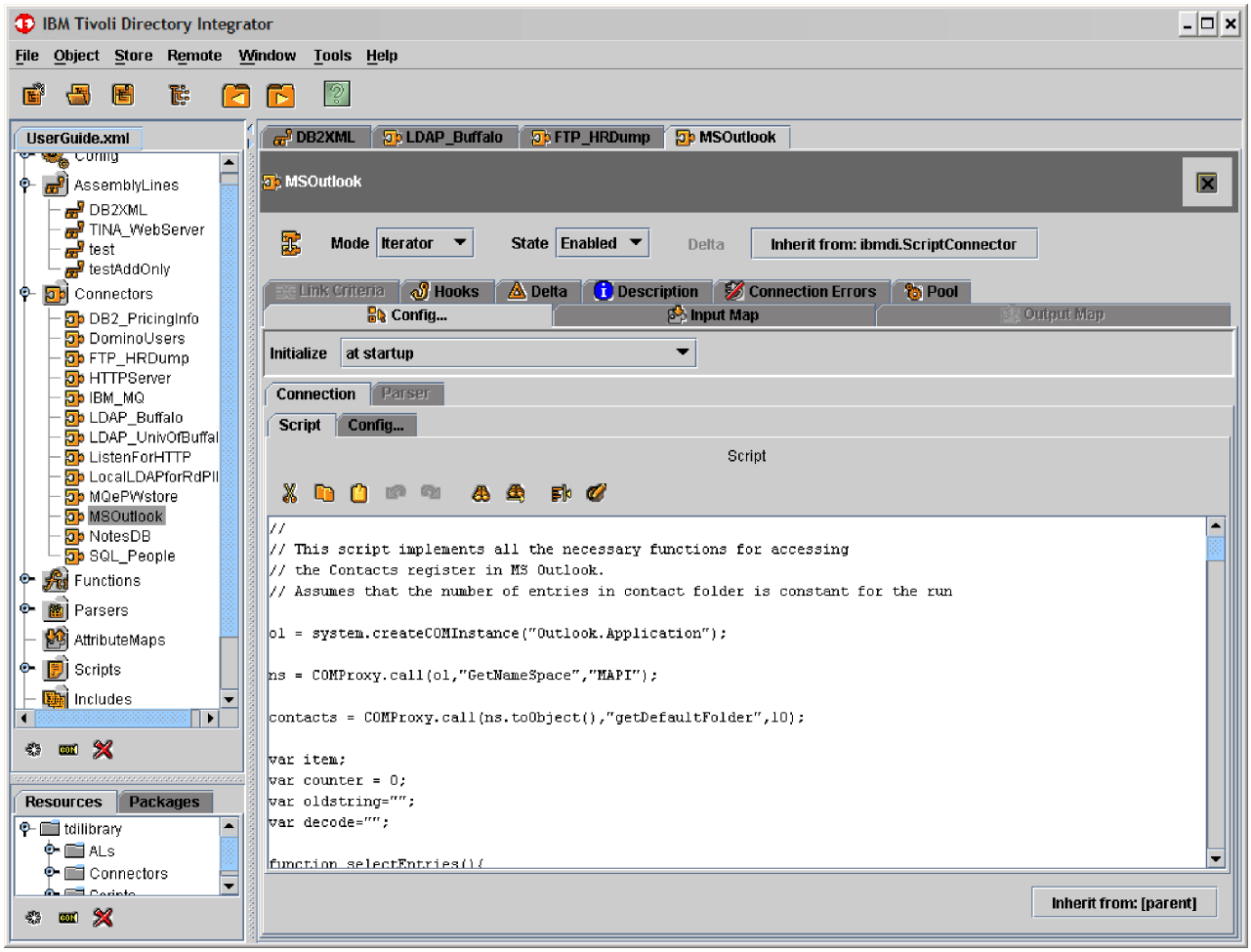
Details window for a Library Connector fills the entire Details Pane (as there is no AssemblyLine information to display).



Scripted Connectors

It is possible to implement a Connector using one of the supported script languages. This is done by creating a Connector (either in the Config Browser or directly in an AssemblyLine) and choosing the Script Connector type.

When you then open the **Config ...** tab of this Connector, you see the following screen:



There are two tabs in this window:

Script Write the script for your Connector. Some sample template code (essentially the methods you should provide) is already provided for you.

Config...
Here you can define the script language to use, and whether to include external script file or files from the Script Library.

Note: Even if you have selected one script language for use in an AssemblyLine, you can still use a component (such as a Connector or Parser) that is written in a different language.

Function components (FC)

As discussed under the Concepts section "Function components (FC)" on page 48, Function components (FCs) are AssemblyLine components that implement some function. Examples of FC functions include driving data through a Parser, invoking some external service (like Web services), or accessing classes and methods in a JAR file. You can create FCs directly in the Config (also known as Library Functions. See "Config folder management" on page 134), or you can create FCs in an AssemblyLine (also called AssemblyLine Functions. For more info on how to add Functions to an AssemblyLine, see the **Data Flow** tab). To add a pre-configured Function in your Config Library to an AssemblyLine, drag the

Function from the Config Browser to the component of the AssemblyLine (in the **Data Flow** tab). Note that the order in which the components are listed is significant.

Setting up a Function component

When you set up an FC, you must make or verify the following settings:

State You can choose **Enabled**, **Passive** or **Disabled**. The default is **Enabled**.

Set Type
When you create an FC, you can select the type directly in the **Add Function** window. To change the type of Function later, use **Inherit from** at the top of the **Function Details** window.

Config
As with Connectors, the **Config** tab is where the Function component displays its configuration parameters. Note that you control the **Initialize** setting of the FC with the menu at the top of the tab.

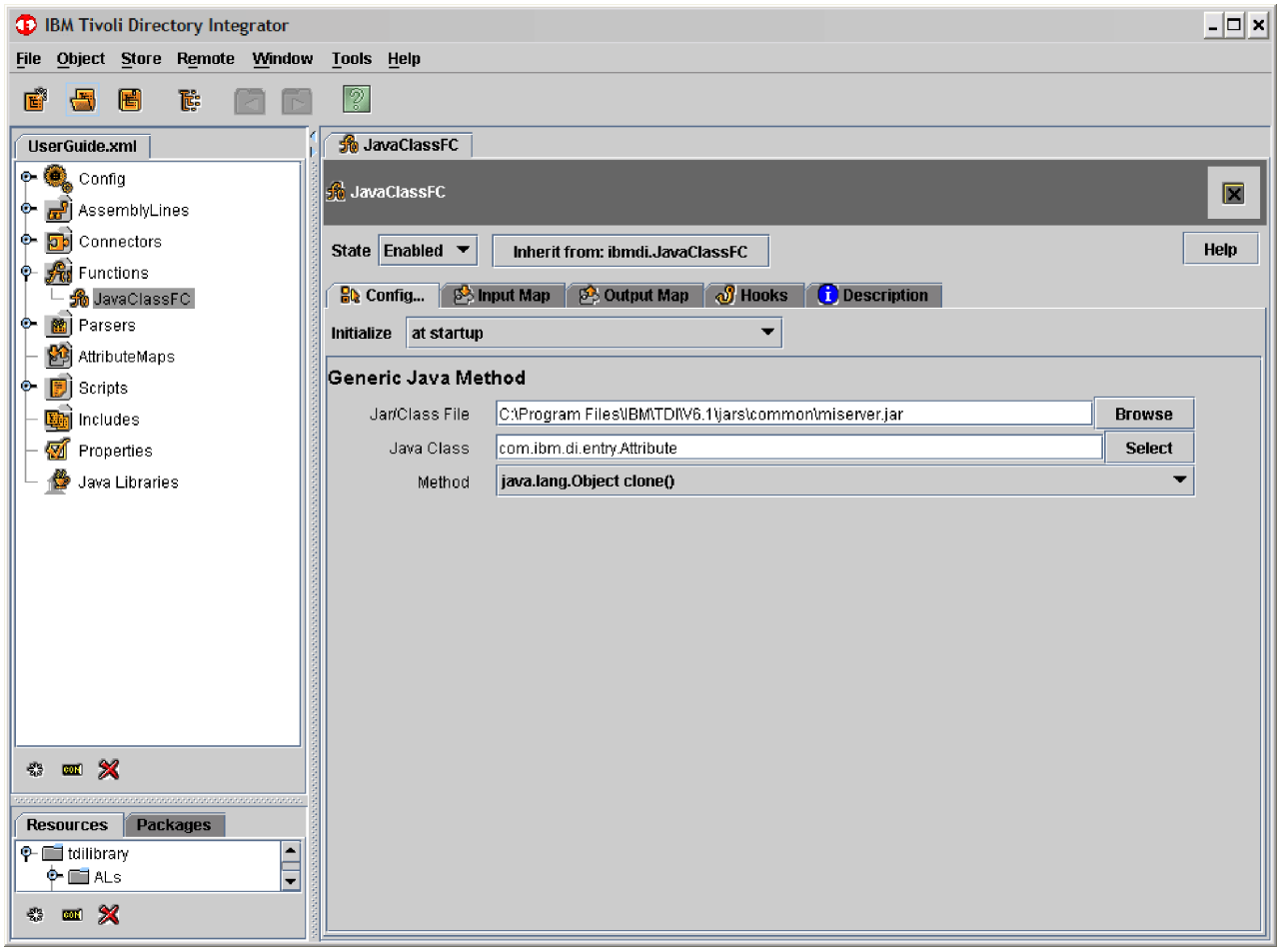
Input Map
In the Input Map you specify the Attributes returned by the FC.

Output Map
For defining those Attributes that are sent to the FC when it is invoked.

Hooks Function Hooks are available in this tab.

Description
This tab consists of a free-form text field where you can enter documentation regarding this particular Function.

Config
The Config Pane for an FC looks like this:



These states have the following meanings:

Enabled

Enabled is the standard (and default) mode for all Functions and means that the Function is initialized and operating as usual.

Passive

If you set a Function to **Passive** mode, then it is initialized at AssemblyLine startup, but is not started during AssemblyLine operation. Instead, you can invoke the Function from your scripts.

Disabled

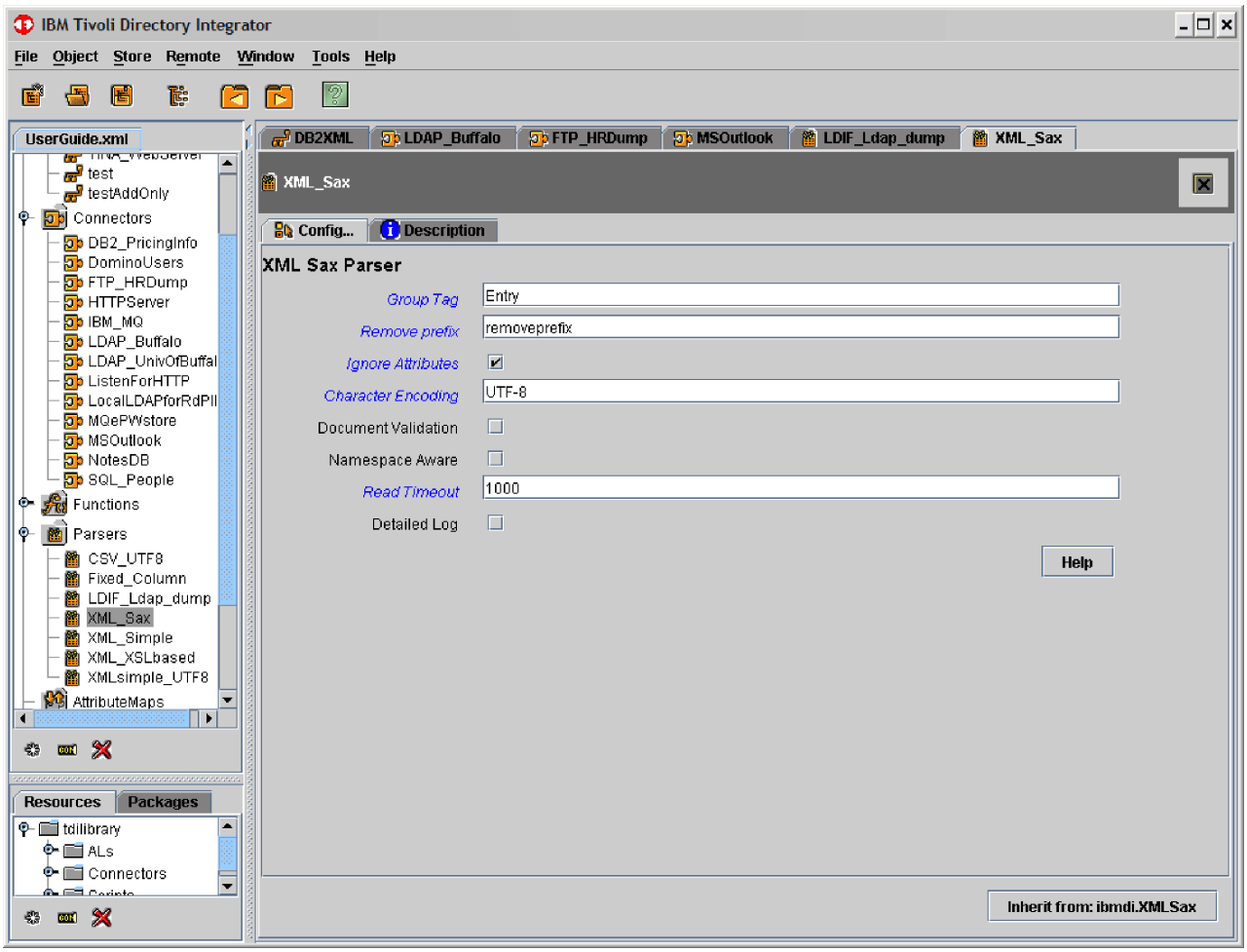
In **Disabled** mode, a Function is neither initialized nor run during AssemblyLine execution. **Disabled** mode is often used during troubleshooting in order to simplify the solution while debugging, helping you localize any problems. Note that a disabled component is not registered as a script variable either.

Parsers

Parsers are used in conjunction with Connectors and Functions to interpret or generate the content that travels over the Connector's byte stream. When the bytestream you are trying to parse is not in harmony with the chosen Parser, you can get an exception like `sun.io.MalformedInputException`. For example, the error message can show up when using the Input Map tab to browse a file.

You can open a new Parser like you open any other AssemblyLine element. See "Config folder management" on page 134 for details.

When you open a new Parser, you get the following Detail window:



This Detail window varies with each Parser.

DSML v2

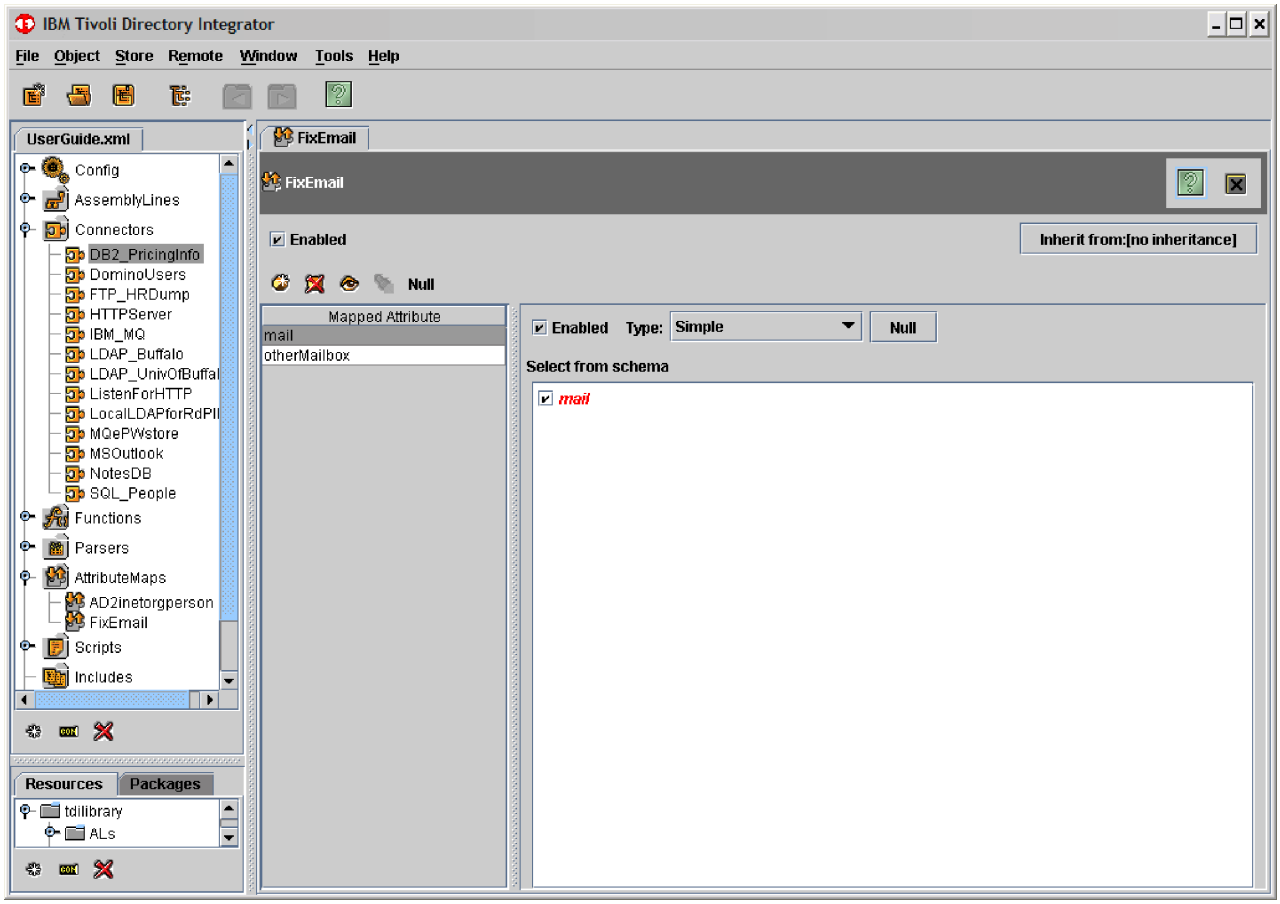
The DSML v2 library that is used by TDI is from IBM Tivoli Directory Server. Using this XML format in the TDS makes TDI compliant with the DSML v2 standard. This means that in addition to supporting the DSML operations Search, Modify, Add, Delete, ModifyDN and Compare, both Auth and Extended operations are also handled. Furthermore, the DSMLv2 Parser supports the optional "requestID" DSMLv2 attribute and the optional "control" DSMLv2 elements.

See the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information about DSMLv2 Components.

Attribute Map components

AttMaps are Attribute Maps that you can drop directly into the AssemblyLine component list, or on the **Inherit from** button of an Input or Output Map in another component. Note that you can save Input and Output Maps to the AttMap Library using the **Copy to library button** at the top of the Attribute Map.

The Attribute details panel looks like this:



You can define Null Behavior for missing attributes by using one of the following methods:

- Simple, Advanced (JavaScript) and Expression mapping types.
- Individual attribute map items that you copy or paste.

See "Attribute Mapping" on page 91 and "Null Behavior" on page 92.

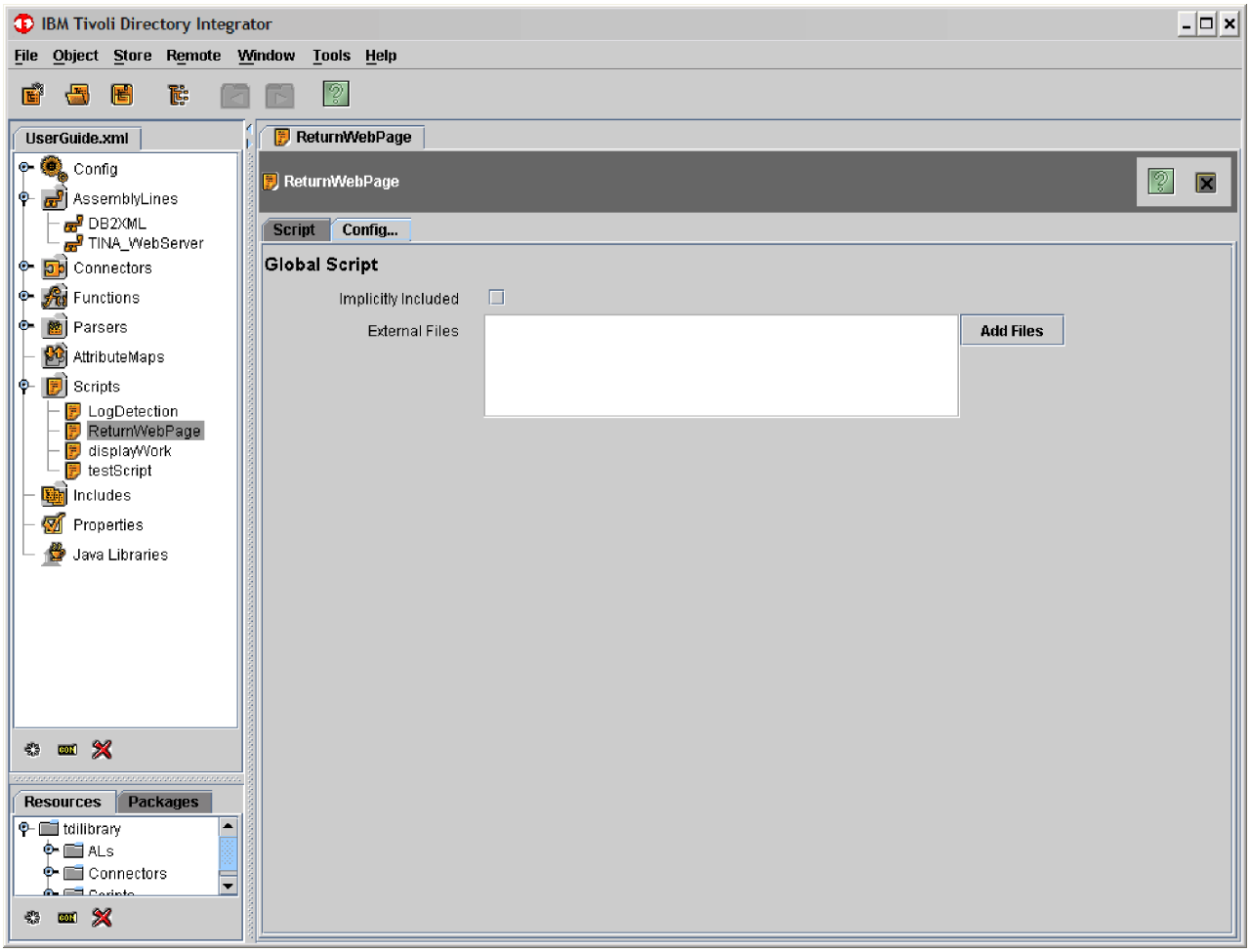
Script Library

Note: This is a library of variables, code snippets and functions you can access from scripts within your AssemblyLine. They can be included as Global Prologs (for processing before the Prolog Hooks of the AL) or dropped into an AL as Script components (SCs).

Scripts that you create in the Config Browser become available for inclusion in your AssemblyLines and EventHandlers as Prologs (for example, scripts which are started at the startup of that item). See “Scripting” on page 52 for more about the concept of scripting in TDI.

You manage your Script Library items as you do any other item in the left navigation bar. See “Config folder management” on page 134 for more details.

The Config Pane for a Script Library item looks like this:



Here you can select if this Script is to be **Implicitly Included** in all AssemblyLines and EventHandlers. If this check box is set, then the script becomes a prolog for every AssemblyLine that has its **Include All Global Prologs** flag enabled in the **Config ...** tab.

If the **Implicitly Included** check box is not enabled, then the script must be explicitly included by an AssemblyLine using the **Include Additional Prologs** parameter (under the **Config ...** tab).

Properties

Properties can be used to control the configuration of components. Together with the new TDI Expressions feature, you can compute these values using Attribute values, Java Properties and even JavaScript code. You can also use Properties and Expressions with Attribute Maps, Link Criteria and Conditions. Properties are fully accessible from script and from the API.

To access properties fromthe Config Editor, click **Properties** in the Config Browser. This opens the **Properties** tab.

The following property-types are available by default:

Solution
Solution Properties arestored in the `solution.properties` file.

Global
Global Properties are stored in the `global.properties` file.

System
System Properties are also available by default, and are kept in the System Store, giving you access to user-defined properties without having to set up an External Properties file.

Java Java Properties, which are only kept in memory (not stored anywhere), provide access to configuration settings of the JVM itself.

At the top of the **Properties** window are two lists, both of which can be set to reference one of the defined Property Stores:

Default store
The Default store is the one that is searched first, as well as where new property values created using API calls go. The designated Default Store takes precedence over the sorting-order of the Property Store list . On reading a property, the first Property Store that has a presence for the property is used. Conversely, when writing a property, the first Property Store that has a presence for the property will be used to write it back. This ensures that the location of the property remains consistent when reading and writing properties . Of course, this behavior can be overridden by explicitly naming the store to be used in the API call.

Password store
TDI will automatically store password parameters for Connectors, removing this sensitive information from the Config file and into an encrypted store.

There is a row of buttons at the bottom of the Property Stores list for creating new Stores based on Connector-technology:

Insert new object
Click to insert a new Store

Delete selected object(s)
Click to delete the select store or stores

Move object upwards
Click to move the selected store up one position in the list.

Move object downwards
Click to move the selected store down one position in the list.

All Properties have **Configuration** and **Editor** tabs. To edit a store, select the store you want to edit from the Properties List, and then use the **Configuration** and **Editor** tabs:

Configuration

This tab defines the layout (Schema) and behavior of this Property Store.

Configuration parameters:

Key Attribute
Defines the schema of the Property Store

Value Attribute
Defines the schema of the Property Store

Initial Load
This switch is relevant only when local caching is selected and the Connector supports Iterator mode. If selected, then the Connector iterates once through the entire store to build a complete local in-memory cache, instead of doing searches when individual properties are accessed.

Cache Timeout
This parameter is set to the number of seconds that a property can be cached before another call to the connector is made to refresh the contents.

Read-Only
Controls whether this Store can be written to or not using API calls. (Note that the file holding your Password Store should never be Read-Only).

Name Filters
Used to control which properties this store is to provide access to. The store could contain additional properties, but only those that conform to the Name Filters are visible using the Property Manager calls.

Editor

Above the property list is a toolbar with the following features:

Reload contents
Loads properties into the editor

Commit changes
Saves properties to the associated data store.

Note: Java Properties are not saved.

Insert new object
Inserts a new property

Delete selected object(s)
Deletes the selected property or properties.

Search
Searches through the list of properties based on text you enter.

Note: Remember that you must press the **Commit changes** button to have TDI persist any changes you make on this tab.

Encrypt individual properties: To encrypt an individual property, select the **Protect** check box next to the desired Property on the **Editor** tab.

Note: Properties are encrypted with the Server Key, meaning that encrypted properties cannot be shared among different server instances with different server keys

Connector tab

This tab is only available for user-defined Property Stores.

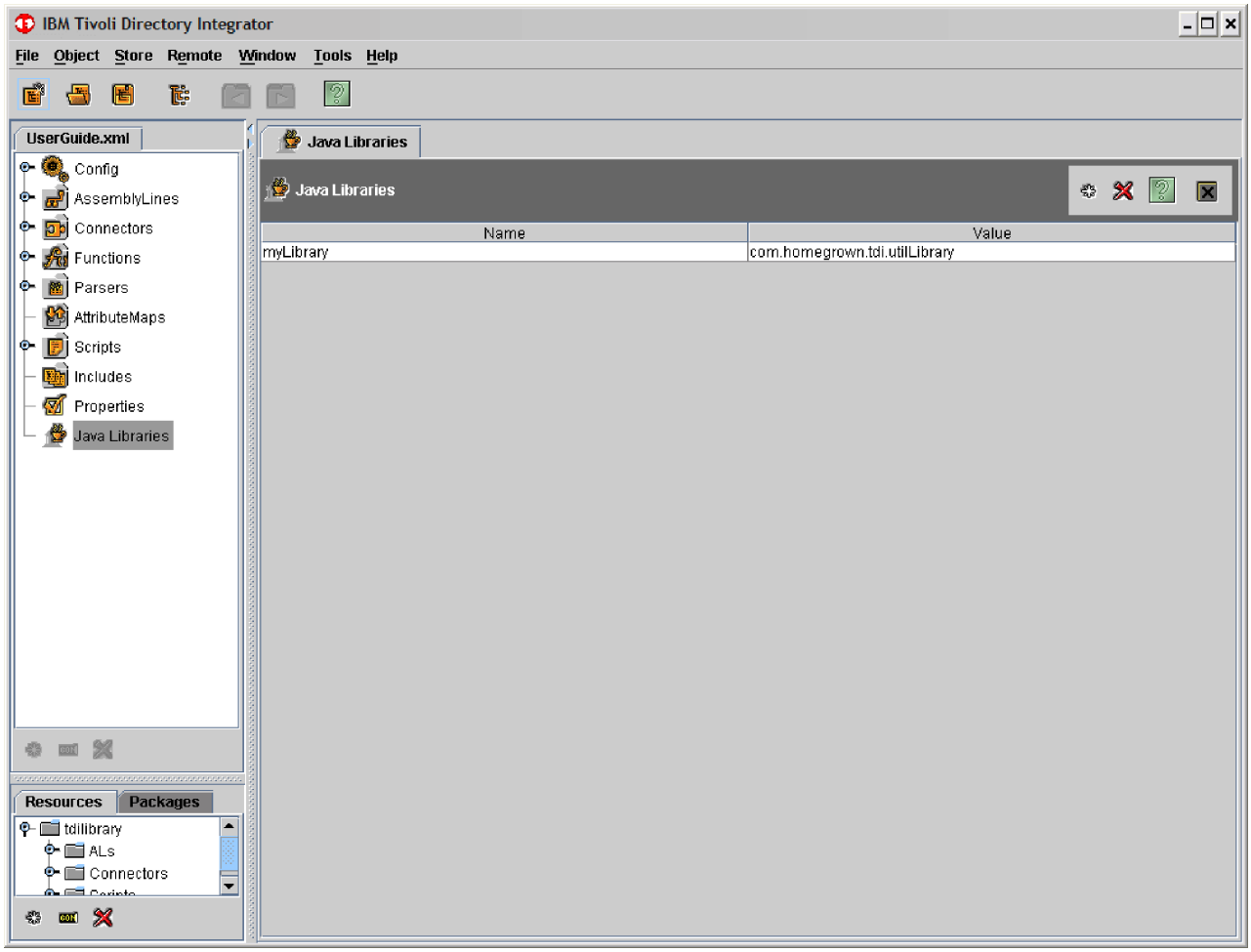
The **Inherit from:** button at the bottom-right part of the Connector tab lets you control which type of Connector to use for this Property Store. The default setting is the ibmdi.Properties Connector, which reads and writes the standard External Properties format used in previous versions.

The rest of the parameters on the tab are dependent on the Connector being used.

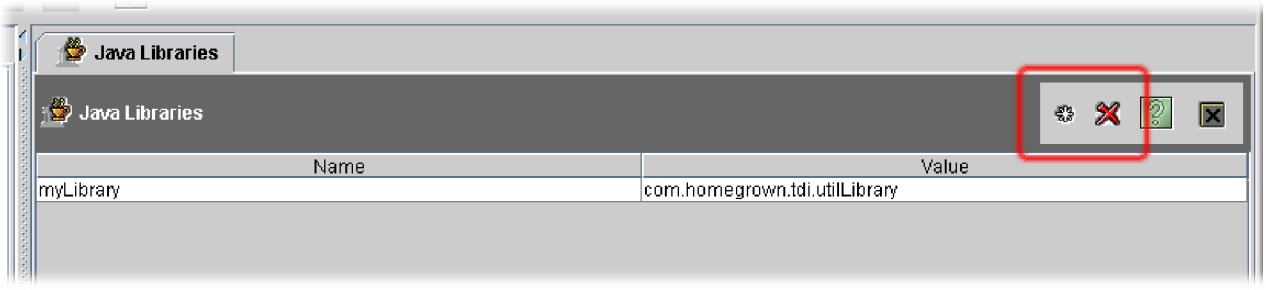
Java Libraries

IBM Tivoli Directory Integrator enables you to layer new functionality on top of the Server by including your own Java libraries.

This is facilitated by opening the JavaLibraries folder in the Config Browser. The system presents you with a list of included libraries.



JavaLibraries can be created and removed by using the toolbar in the Details Pane title area only.



Once you have created a new Java Library entry, simply click in the grid field that you want to edit and press **F2**. Or you can also double-click the field to enter edit mode. There are two fields here for you to enter:

Name This is the name of the object that is made available to your scripts, and which provides you access to the library's functions.

Value Here you enter the name of the Java class that is tied to this new script object.

Note: In order to be able to include new Java libraries, you must place the library's .jar files in either the `jars` subdirectory of the IBM Tivoli Directory Integrator installation directory, or in an existing or new subdirectory under `jars`.

Preferences

Your user preferences (for the Config Editor) are stored in the .ibmdi file that you find in your home directory. This file is created for you the first time you start ibmditk. It defines the following field:

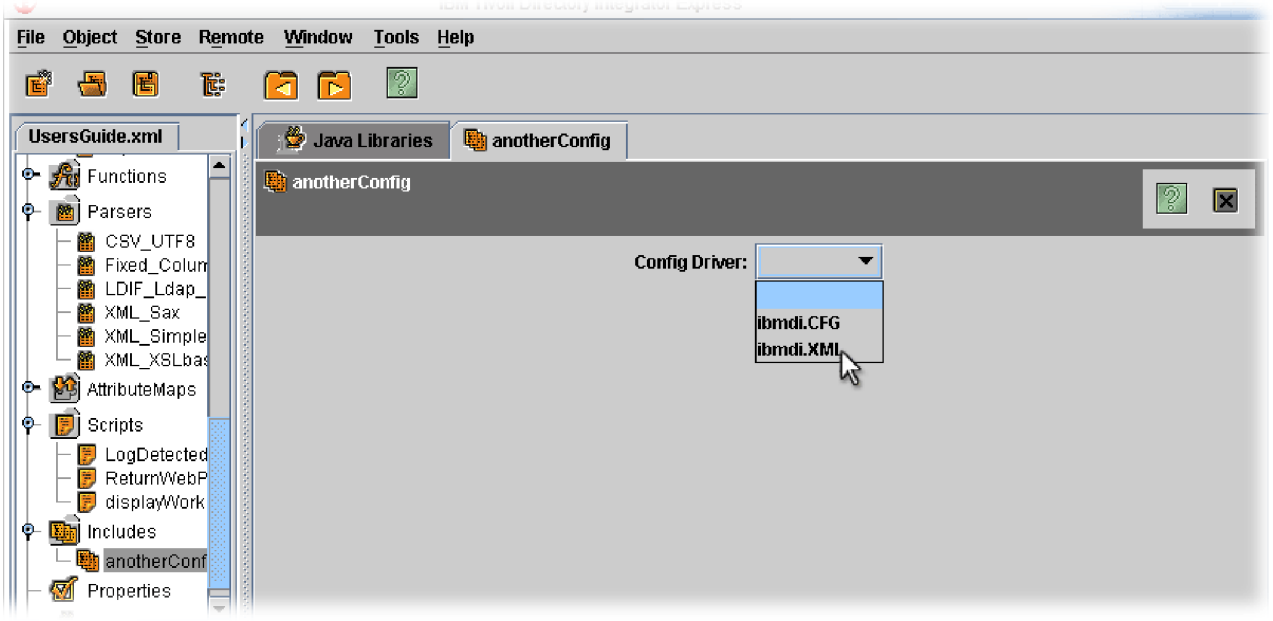
window.*
The look of your dialog boxes, for example size, position, where the split is, and so forth.

This dialog box is available through **File–>Edit Preferences**. It contains options that are valid for the current Config Editor session.

Includes

A Config can be set up to include configuration elements from other Configs. This makes it possible to set up central libraries of components, scripts and other Config elements that can be stored on corporate servers, but still available to integration specialists working through the organization.

You create and manage your Includes as you do other Config items such as Connectors and AssemblyLines. See "Config folder management" on page 134 for details on how to do this. Once you have added an Include, you see the following screen:



The only available option at this point is the Config Driver parameter. Select the Configuration Storage Architecture (CSA) driver that you must use to reach this Config.

There are only two choices at present: IBM Tivoli Directory Integrator 5.1.1 legacy .cfg format, and XML.

Note: The .cfg format is supported as read-only and for legacy reasons. It is recommended that you use the XML format for storing the Config information to file.

Once you have chosen the Config driver, you are presented with more parameters. You can set the filename or URL to the Config file, as well as the decryption password (if one is in use). You can have as many Includes as you want in your Config, and these can be stored in various ways, accessed through the different Config drivers.

Parameter Substitution

Parameter Substitution is a feature that enables you to store sensitive information outside your Config in a secure format, but still keep it configurable.

Think of the Parameter Substitution concept as global system variables that can be used throughout your solution. Of course you can access these Parameters from your scripts, enabling you to make your code data-driven, changing its functionality based on the value of one or more of these Parameters. However, the most powerful use of Parameter Substitution is as parameter values in the configuration of components, such as Connectors, Parsers and EventHandlers.

There are two types of Parameter Substitution in TDI:

- Properties
- Advanced parameter substitution

Properties

Properties are simple `keyword:value` pairs of parameters kept outside your Configs, stored instead in External Properties files. This enables you to keep confidential information like passwords outside of your Config files.

For information about setting up and managing Properties, see "Properties" on page 181.

Advanced Parameter Substitution with Expressions

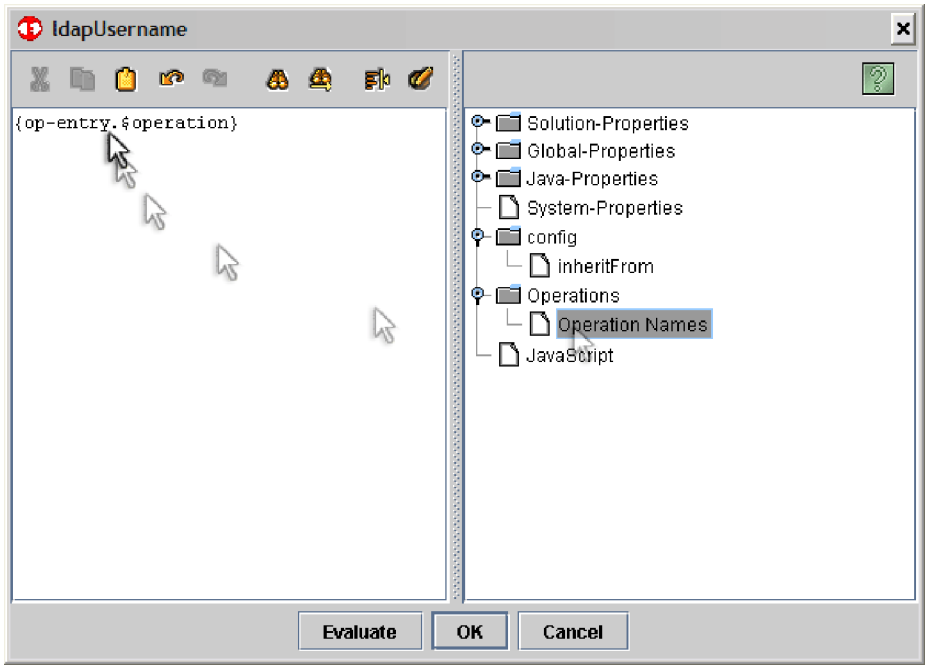
Contrary to the simple, static `keyword:value` concept implemented in Properties, the Advanced Parameter Substitution facility (which leverages TDI Expressions) provides extensive capabilities to evaluate and re-format parameter values at runtime. It is allows you to use Property settings as well as data and Config settings.

Expresions Editor

Throughout the configuration screens in TDI, selection lists offer a Parameter Substitution selection that brings up the Parameter Substitution Editor. The selection lists are:

1. The *Value* column in the Branch Component configuration conditions window
2. The *Value* column in the Loop Component configuration conditional-loop window
3. The *Value* column in the LinkCriteria window
4. The *External Property* menu in the Parameter Information dialog that is shown when a component parameter label is clicked.

Once invoked, the Parameter Substitution Editor window appears:



Expression formatting

TDI Expressions leverage the java.text.MessageFormat class, part of the Java engine under which TDI runs. Specifically, the substitution pattern reuses the syntax and layout of java.text.MessageFormat. Refer to the Java API for a complete description. In general though, MessageFormat uses placeholders in the format string to refer to the values passed as parameters. An example copied from the JavaDocs resembles the following lines:

```
Object[] arguments = {
     new Integer(7),
     new Date(System.currentTimeMillis()),
     "a disturbance in the Force"
 };

 String result = MessageFormat.format(
     "At {1,time} on {1,date}, there was {2} on planet {0,number,integer}.",
     arguments);

 output: At 12:30 PM on Jul 3, 2053, there was a disturbance
           in the Force on planet 7.
```

The format string uses curly braces to enclose substitution and format elements:

```
FormatElement:
          { ArgumentIndex }
          { ArgumentIndex , FormatType }
          { ArgumentIndex , FormatType , FormatStyle }

ArgumentIndex: integer
```

TDI Expressions will use this format with the following additional syntax:

```
TDIFormatElement:
         { TDIReference }
         { TDIReference, FormatType }
         { TDIReference, FormatType , FormatStyle }

TDIReference: string
```

The TDI Expressions feature will parse the format string and replace *TDIReference* occurrences with a sequentially increasing *ArgumentIndex*. For each *TDIReference*, the value it refers to is added to the argument list used in the call to

MessageFormat. Hence, the following procedure shows the primary differences between MessageFormat in its original form and Expressions:

1. Each *TDIReference* is replaced by the next sequential value of *v* (where *v* starts with zero).
2. The object/value referenced by *TDIReference* is added to the argument list at the position yielded by *v*.
3. ArgumentIndex references refer to the n'th *TDIReference* starting from zero. (for example, {2}) refers to the third *TDIReference* occurrence, {3} to the fourth etc).
4. You can mix {*ArgumentIndex*} and {*TDIReference*} (see examples section) in the same pattern.

Depending on the context in which parameter substitution is performed, there are a number of runtime objects that can be referenced. The table below shows a complete list of recognized references.

Table 6. References

TDIReference	Value	Availability
work.*attrname[.index]*	The *work* entry in the current AssemblyLine. The optional *index* refers to the n'th value (starting at zero) of the attribute. Otherwise the first value is used.	AssemblyLine
conn.*attrname[.index]*	The *conn* entry in the current AssemblyLine The optional *index* refers to the n'th value of the attribute. Otherwise the first value is used.	AssemblyLine during Attribute Mapping
current.*attrname[.index]*	The current entry in the current AssemblyLine The optional index refers to the n'th value of the attribute. Otherwise the first value is used.	AssemblyLine during Attribute Mapping for Modify
config.*param*	The configuration object containing the parameter being expanded. *param* refers to another parameter in the same configuration object.	AssemblyLine EventHandler Connector Parser Function Component
alcomponent.*name.param*	The connection parameter value of another AssemblyLine component. *name* is the name of the AssemblyLine component *param* is the parameter name of the name object **Note:** This is a shortcut for `alcomponent.connector.getParam(name)`, `alcomponent.function.getParam(name)` and so forth	AssemblyLine

Table 6. References (continued)

TDIReference	Value	Availability
property[:storename] .name property [:storename/bidi].name	A TDI-properties reference. The optional storename targets a specific property store. The default store is used in case this parameter is absent. *name* is the property name *bidi* will, when present, cause setting the parameter value to forward the call to the referenced property store. When *bidi* is present no other substitution patterns or text is allowed.	Always

JavaScript<<*EOF* script code &ellipsis; *EOF*	Arbitrary script code to generate a value.
	
	The *EOF* is an arbitrary string that terminates the JavaScript snippet. The JavaScript is collected up until a single line with the EOF string is encountered.
	
	The JavaScript snippet is automatically wrapped in a function body with a single parameter that holds all expanded values as well as all the implicitly available objects (for example,task, work etc). Expanded values are named by its generated ArgumentIndex (for example, “0”, “1” etc).
	
	function x(params)
	
	Example:
	`var conn = params.get("conn");`
	`var p0 = params.get("0");`
		Always

Examples

All examples show the actual substitution string without the trigger prefix/suffix strings. In a configuration file (the XML representation of the Config), these strings would be enclosed in *@PROPERTY{<substitution-string>}* - also for the simple External Properties format.

Simple: Previous versions use a single substitution mechanism where a prefix/suffix triggers substitution based on a value from the External Properties.

Table 7. Simple example

Input String	external-property-name
Translated String	{0}
Parameter Array	[0] = extprop.getParameter(“external-property-name”)

Multiple Substitutions: This example shows basic substitutions with multiple arguments.

Table 8. Multiple example

Input String	uid={work.uid},{work.ldapSearchBase}
Translated String	uid={0},{1}

Table 8. Multiple example (continued)

Parameter Array	[0] = work.getObject(“uid”); [1] = work.getObject(“ldapSearchBase”);

An example with mixed references:

Table 9. Mixed reference example

Input String	cn={work.sn} or sn={0}
Translated String	cn={0} or sn={0}
Parameter Array	[0] = work.getObject(“sn”);

JavaScript Substitutions: This example shows complex substitutions using JavaScript where the returned value is the “sn” attribute which is conditionally pulled from *conn* or *work*.

Table 10. JavaScript example

Input String	cn={JavaScript<<END
	if (params.get(″conn″) != null)
	return params.get(″conn″).getString(″sn″);
	else
	return params.get(″work″).getString(″sn″);
	END
	} or sn = {JavaScript<<END
	if (params.get(″conn″) != null)
	return params.get(″conn″).getString(″sn″);
	else
	return params.get(″work″).getString(″sn″);
	END
	}

Translated String	cn={0} or sn={1}

Generated JavaScript	function x0(params) {
	if (params.get(″conn″) != null)
	return params.get(″conn″).getString(″sn″);
	else
	return params.get(″work″).getString(″sn″);
	}
	
	function x1(params) { function body above }

Parameter Array	[0] = x0(params) [1] = x1(params)

Descriptive Example: Here is a detailed description of the steps performed using the following substitution pattern:

```
SELECT {JavaScript<<EOF
 var str = new Array();
 str[0] = "A";
 str[1] = "B";
 return str.join(",");
EOF
} FROM {property:mystore.tablename} WHERE A = '{work.uniqueID}'
```

Assuming there is a TDI property store name mystore that has a property named tablename with the value of “PEOPLE”, and a Work Entrywith an attribute named uniqueID with a value of “uid123”, the expansion would result in the following three steps:

1. Constructed argument array:

   ```
   args[0] = ″A,B″;  ←
   args[1] = ″PEOPLE″; ← TDIProperties.getProperty(″mystore″, ″tablename″);
   args[2] = ″uid123″; ← work.getObject(″uniqueID″);
   ```

2. Converted substitution string:

   ```
   SELECT {0} FROM {1} WHERE A = '{2}'
   ```

3. Final result after calling MessageFormat(string, args):

   ```
   SELECT A,B FROM PEOPLE WHERE A = 'uid123'
   ```

The above example did not use any of the available MessageFormat formatting capabilities but these are of course also available.

Schema

The “schema” for the pattern substation depends on the context in which substitution is performed. If you call it manually, then the available objects (schema) are defined by you. The configuration driver will perform automatic substitution when it encounters a trigger string. The following tables list the available objects in each standard context. These objects are also the ones passed to any embedded JavaScript function.

Note that pattern keywords are expanded automatically by TDI-PS using the provided objects. There is no one-to-one relation between the pattern keywords and the provided objects.

Component Configuration:

Table 11.

Object	Value
config	The component’s “connection” configuration object.
mc	The MetamergeConfig object of the Config instance (config.getMetamergeConfig())
work	The Work Entryof the AssemblyLine
task	The AssemblyLine object

LinkCriteria:

Table 12.

Object	Value
config	The component’s “connection” configuration object.
mc	The MetamergeConfig object of the Config instance (config.getMetamergeConfig())
work	The Work Entryof the AssemblyLine
conn	The conn entry of the AssemblyLine
task	The AssemblyLine
alcomponent	The Connector component

Branch & Loop Components:

Table 13.

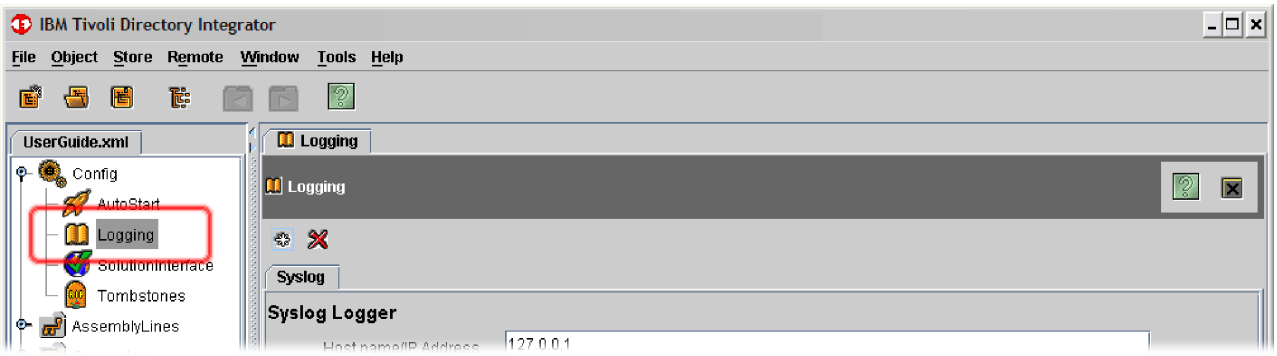
Object	Value
config	The component's "connection" configuration object.
mc	The MetamergeConfig object of the Config instance (config.getMetamergeConfig())
work	The Work Entryof the AssemblyLine
task	The AssemblyLine
alcomponent	The Connector/Function component

Logging

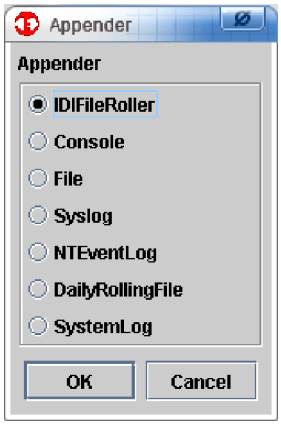
For more information about Logging, see "Logging and debugging" in *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide* as well as "Logging and debugging" on page 156 in this guide.

Note: TDI solution builders need a way to protect sensitive data, such as passwords, from being printed in clear text when tracing on the solution is needed. In IBM Tivoli Directory Integrator, some of the methods dealing with the Attribute class say whether an attribute is protected or not.

Logging that is global to the entire Config file can be configured by opening up the **Config** folder in the Config Browser, and selecting the **Logging** item. Only the parameters that describe how messages are logged are described here.



All log configuration windows operate in the same way. For each one you can set up one or more log schemes. These are active at the same time, in addition to whatever defaults are set in the `log4j.properties` and `executetask.properties` files.



The possible log schemes are as follows:

Note: All log windows showing loggers that write to encoding enabled streams contain the **Character encoding** field.

IDIFileRoller

Sometimes, you want to log to file but keep a limited number of files, as they can fill your disks. IDIFileRoller generates a new file for each run of the Server. The system saves only the specified number of previous logs. If your log is called mylog.txt, and you ask for 2 generations, then after 3 runs you have a mylog.txt (last run) as well as the files mylog.txt.1 and mylog.txt.2, where mylog.txt.2 is the oldest log. From this point, you do not get more files, only newer versions with the same name. Keep two generations of backup files.

IDIFileRoller has the following parameters:

File Path

The name of the file to log to. The path is relative to where you installed IBM Tivoli Directory Integrator. The special macro {0} used in filenames is replaced by the name of the Server. Similarly, {1} used in filenames is replaced by a unique identifier generated by the system for you. The {1} macro has no relevance for the special case where you use IDIFileRoller, but is important where you want unique file names.

Number of backup files

If your File Path was mylog.txt, and you select 2 backup-files, the two previous runs have their files renamed to mylog.txt.1 and mylog.txt.2 when you run a third time.

Layout

Determines the format of the log message. Options are:

- Pattern (used if you want to customize the way the messages are logged)
- Simple (format containing just the loglevel and the message)
- HTML (creates an HTML file containing some (relative) time info, thread info, loglevel, category, and message)
- XML (similar to HTML, but generates an XML file (using namespace-prefix log4j))

Pattern
Only used when **Layout** is **Pattern**. See ″Creating your own log strategies″ in *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

Log level
Severity level of the log messages. Options are (from maximum to minimum information):

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

Log Enabled
Click to enable the use of this appender.

Console
Logs to the console (standard output). This is in the window where you started the server (ibmdisrv) or the execute task-window in the Config Editor (ibmditk). **Console** has the following parameters:

Layout
See **IDIFileRoller**, previous.

Pattern
See **IDIFileRoller**, previous.

Log level
See **IDIFileRoller**, previous.

Log Enabled
See **IDIFileRoller**, previous.

File Logs to a file. **File** has the following parameters:

File Path
See **IDIFileRoller**, previous.

Append to file
Click to append log information to file. If this is not checked, the file is overwritten.

Layout
See **IDIFileRoller**, previous.

Pattern
See **IDIFileRoller**, previous.

Log level
See **IDIFileRoller**, previous.

Log Enabled
See **IDIFileRoller**, previous.

Syslog
Enables IBM Tivoli Directory Integrator to log to Unix Syslog. **Syslog** has the following parameters:

Host name/IP Address
Host to log to.

Syslog Facility
Legal facilities found in the list. Must be supported by the host you are logging to.

Print Facility String
If set, the printed message includes the facility name of the application.

Layout
See **IDIFileRoller**, previous.

Pattern
See **IDIFileRoller**, previous.

Log level
See **IDIFileRoller**, previous.

Log Enabled
See **IDIFileRoller**, previous.

NTEventLog
Enables applications to log using the Windows NT® EventHandler (on Windows platforms). **NTEventLog** has the following parameters:

Layout
See **IDIFileRoller**, previous.

Pattern
See **IDIFileRoller**, previous.

Log level
See **IDIFileRoller**, previous.

Log Enabled
See **IDIFileRoller**, previous.

DailyRollingFile
DailyRollingFile saves old files with a datestamp in their names. It usually is used with the **Append to file** parameter set to **true**. **DailyRollingFile** has the following parameters:

File Path
See **IDIFileRoller**, previous.

Append to file
Create new file or append to existing file, depending on whether this is checked. You usually want this on when using the DailyRollingFile.

Date Pattern
How often the file is rotated. Use the list to choose resolution from minutes to months. For example, if the File Path is set to example.log and the DatePattern set to `'.'yyyy-MM-dd`, on 2003-10-31 at midnight, the logging file example.log is copied to example.log.2003-10-31. Logging for 2003-11-01 continues in example.log until it rolls over the next day.

Layout
See **IDIFileRoller**, previous.

Pattern
See **IDIFileRoller**, previous.

Log level
See **IDIFileRoller**, previous.

Log Enabled
See **IDIFileRoller**, previous.

SystemLog
This Appender creates log files in a catalog hierarchy under *<TDI_installation_directory>*/system_logs. For each Config File, there will be a corresponding directory with logfiles named *AL_xxx* or *EH_xxx*, where xxx is the name of the AssemblyLine or EventHandler being run.

If you want to view AssemblyLines logs using the Administration and Monitoring Console, you must add a SystemLog Appender to the AssemblyLine.

This Appender has the following parameters:

Pattern
Specifies the format of the log as defined by LOG4J. The default value is:

```
"%d{ISO8601} %-5p [%c] - %m%n"
```

Additional values available in the field are:

```
"%d{HH:mm:ss} %p [%t] - %m%n"
"%p [%t] %c %d{HH:mm:ss,SSS} - %m%n"
```

Log level
See **IDIFileRoller**, previous.

Log Enabled
See **IDIFileRoller**, previous.

Log Levels

Log levels can be

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

DEBUG, INFO, WARN, ERROR and FATAL have increasing levels of message filtration.

Note that the IBM Tivoli Directory Integrator logmsg() (i.e., `task.logmsg("Hello")`) calls log at INFO level if nothing else is specified. This means that setting loglevel to WARN or lower silences your logmsg calls as well as all Detailed Log settings.

Parameter labels in the Connector and Parser windows

If you leave the cursor on the label, a tooltip pops up, showing a short description.

Blue, cursive label text indicates that the value of this parameter is inherited. When you override an inherited parameter, the text label turns black.

If the value is derived from an External Property, the name of the parameter is in black boldface; however, if this External Property is missing somehow, the name will be cursive, bold, and in red.

If you click on the label, you get some information about the parameter, including the internal name of the parameter. This can be useful when you want to set Connector parameters through scripting.

Chapter 4. Web Services Suite

IBM Tivoli Directory Integrator contains a number of components collectively referred to as the TDI Web Services Suite; its aim is to provide an environment where Web Services (WS) can be both consumed and published. *Consuming* WS means that work flows in TDI can reach out to existing Web Services, while *publishing* WS means that TDI work flows can be called by other systems through the WS mechanisms.

The Web Services suite consists of the following components:

- WSReceiverServerConnector
- AxisEasyWSServerConnector
- AxisJavaToSoap FC
- WrapSoap FC
- InvokeSoapWS FC
- AxisSoapToJava FC
- AxisEasyInvokeSoapWS FC
- ComplexTypesGenerator FC

Detailed information about each of these components is available in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

TDI WS Suite philosophy

The philosophy of the TDI WS Suite is to provide a generic architecture for Web service interaction with extensibility and customization points that allows you to handle (by coding, scripting or using specific components) any incompatibility issues that may arise. No general assumptions are made in the architecture itself about Web service style, data encoding, data types, and so forth – these can be addressed by specific components that may or may not be used in a particular solution.

Components and tools

TDI provides different mechanisms (including Axis binding logic) for mapping XML data to Java data and vice versa—these mechanisms facilitate or provide out-of-the box solutions of the serialization problem described below.

TDI also provides simplified components that are easy to configure, and which you can use out-of-the box in non-complex Web service scenarios.

This is the first set of TDI Web service components. The components introduced below and described in detail in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* provide the means to build a complete (both client-side and server-side) Web service solution in the modular TDI Web service architecture.

AxisJavaToSoap FC

This component can be used both as a Web service client and a Web service server. This component receives an Entry or a Java object and produces the SOAP request (when on the client) or response (when on the

server) message. It provides the whole SOAP message, as well as separately the SOAP Header and the SOAP Body to facilitate processing and customization.

The component supports both RPC and Document style. A parameter (*Mode*) specifies whether the FC is used on the client or on the server side.

WrapSoap FC

This component generates a complete SOAP message given the SOAP Body and optionally a SOAP Header. This is useful when you customize the content of the SOAP Body or create it completely on your own.

The component accepts the contents of the SOAP Body, the SOAP Header, and attributes for the SOAP Envelope, Header and Body XML elements (usually namespace declarations) and will create the complete SOAP message.

This is actually a helper FC that saves you from error-prone processing of string or DOM objects to wrap your SOAP data into a complete SOAP message.

InvokeSoapWS FC

This FC uses a WSDL and a corresponding operation to be invoked. When called, it requires a complete SOAP request message, and the remote operation is called with this message. The output message is returned, but no XML-Java binding is performed—it only returns the SOAP response message.

To facilitate further custom XML-Java binding the FC provides on output the entire SOAP response message, as well as the SOAP Header and the SOAP Body.

AxisSoapToJava FC

This component can be used both as a Web service client and a Web service server. This FC uses Axis' mechanism for parsing SOAP response (when on the client) or SOAP request (when on the server) to Java objects—as a complementary component to the *AxisJavaToSoap* FC.

It is given a SOAP response/request message and returns the parsed Java objects either as standalone Java object(s) or capsulated in an Entry object.

This component supports both RPC and Document style.

AxisEasyInvokeSoapWS FC

This is a ″simple″ Web service invocation component. When using this FC you loose the capability to Hook custom processing, i.e. you are tied to the processing and binding provided by Axis, but you gain simplicity of setup and use.

ComplexTypesGenerator FC

This FC generates JAR files which contain Java classes which implement the complex types (used by the SOAP operations in a WSDL file) and their serialization/de-serialization to/from XML.

The following steps are performed:

1. Generate Java source files
2. Compile these Java source files
3. Pack the Java class files into a JAR file

The JAR files generated by this FC are used by the *AxisJavaToSoap*, *AxisSoapToJava*, *AxisEasyInvokeSoapWS* and *AxisEasyWSServerConnector* in order to serialize/de-serialize complex types.

This FC displays a ″Generate complex types″ button in its UI. Click this button to generate the JAR file. This FC is only used during design time, and ignored at run time by the AssemblyLine.

Note: In order to be able to use this FC, you need to have a Java SDK installed; its location must either be specified as a parameter to the FC or present in your execution path.

WSReceiverServerConnector

This connector processes Web service requests by first passing the SOAP request to the AssemblyLine it is embedded into, and later retrieving the result of the AssemblyLine and sending the response back to the Web service client.

The Connector provides the entire SOAP message, as well as separately the SOAP Header and the SOAP Body to facilitate processing and customization. The Connector also supplies the value of the ″soapAction″ HTTP header as well as other SOAP/HTTP request details.

The Connector supports both RPC and Document style.

AxisEasyWSServerConnector

This is a ″simplified″ Web service server Connector. It internally instantiates, configures and uses the *AxisSoapToJava* and *AxisJavaToSoap* FCs.

When using this Connector you forgo the possibility of hooking custom processing before parsing the SOAP request and after serializing the SOAP response, i.e. you are tied to the processing and binding provided by Axis, but you gain simplicity of setup and use.

Usage and scenarios

The above FCs and Server Connectors provide the functionality of Axis in the modular TDI Web Service architecture. This gives you extension points where you can customize or override certain parts of the Axis Web service processing.

The TDI Web Services suite of components attempts to combine and provide both the ″simple″ and ″advanced″ way of invoking and providing Web services with Axis-based code. There are two independent aspects of ″simple″ and ″advanced″ use:

Simple or Complex Types

When you want to use complex types you need to generate custom Java classes with the *ComplexTypesGenerator* FC, make them available to TDI (put them on the TDI classpath) and use these classes in the business logic of your process that is before and after the Web service call (when on the client) and during the handling of the Web service request (when on the server).

But when no complex types are used, no class generation will be necessary – you will not fill in the Complex Types FCs' and Connector's parameters. As a result when the Web service is simple (no complex types) – the usage is simple, when the Web service is complex – the usage is complex.

Simple or Customized workflow

The second aspect is the possibility of customizing the workflow which in turn leads to heavier setup and use – configuration of multiple FCs, scripting to tie them together.

The below functionality attempts to enable both simple and advanced/customized use.

- The *AxisJavaToSoap*, *InvokeSoapWS* and *AxisSoapToJava* components can be used in that order to implement a full-featured Web service client.
- The *AxisSoapToJava* and *AxisJavaToSoap* components can be used on the server to expose a piece of functionality as a Web service.
- The *WrapSoap* FC facilitates the creation of the SOAP request/response message before it is passed to *InvokeSoapWS*/*AxisSoapToJava* when you have performed custom processing of the SOAP data or have completely overridden the Axis binding mechanism.
- The *ComplexTypesGenerator* generates complex types from WSDL.

Using the WS Suite

Simple usage

The *AxisEasyInvokeSoapWS* FC that provides easy access (single component, single Config screen, no scripting) to each service that Axis can talk to.

The *AxisEasyWSServerConnector* uses the *AxisSoapToJava* and *AxisJavaToSoap* FC functionality. A limitation of this Connector is that it is tied to the Axis data binding code and no other data binding code can be used. Another limitation of the *AxisEasyWSServerConnector* is that it can only be configured to handle one SOAP operation—it cannot service several SOAP operation requests. But apart from that the customer is free to use whatever components he likes in the AssemblyLine that contains the Connector.

Advanced usage

Below we will describe three scenarios using a WebService Connector in Server mode that can be modeled with the above components.

Simple operation

The *WSReceiverServerConnector* is embedded in an AssemblyLine. The AssemblyLine contains the *AxisSoapToJava* and *AxisJavaToSoap* FCs. In between these two components there are an arbitrary number of other components (Connectors, Script Components, Branch Components, other FCs) which implement the actual logic of the Web service. In this way after the Work Entry goes past the *AxisSoapToJava* FC it contains the Java representation of the SOAP request message. Thus the Work Entry is ready to be processed by the core logic configured for this AssemblyLine. After this processing is complete the *AxisJavaToSoap* FC converts the result to a SOAP message, which the Connector returns to the Web service client. In this scenario the *WSReceiverServerConnector* provides only one WSDL operation to its clients.

Handling several WSDL operations

It is possible to configure the *WSReceiverServerConnector* to handle several WSDL operations and not just one. The *AxisSoapToJava* FC requires the name of the WSDL operation to be passed as a FC parameter. That is why we need to put in a Script Component in the AssemblyLine instead of the *AxisSoapToJava* FC and inside this script component to script, you must

create an instance of the *AxisSoapToJava* FC and set its WSDL operation. Next, invoke the functionality of the FC and store the result in the Work Entry as would the *AxisSoapToJava* FC directly in the AssemblyLine. In this way it is possible to script your logic for choosing what to execute in response to Web service requests for different WSDL operations.

Mapping AssemblyLines to WSDL operations

If you want to map AssemblyLines to WSDL operations, you could use the following configuration: A *WSReceiverServerConnector* is configured in an AssemblyLine.

If you know the set of operations to invoke beforehand, the AssemblyLine will initialize a number of *AxisSoapToJava* and *AxisJavaToSoap* components with predefined configurations. A Script Component in the AssemblyLine will analyze the operation to invoke and will parse the SOAP request with the correct *AxisSoapToJava* FC. Then the corresponding AssemblyLine will be started by the core AssemblyLine (directly or through an AssemblyLine Connector) and the result will be passed to the correct *AxisJavaToSoap* FC.

If you do not know the set of operations to invoke beforehand, then instead of using a set of pre-configured FCs the AssemblyLine will instantiate and configure an *AxisSoapToJava* and an *AxisJavaToSoap* FC for each event.

WS Suite Considerations

The most important issue to understand when building or accessing WS is that **SOAP and WSDL are not enough for a successful invocation of a Web service**. These standards define the core framework for WS communication but this is not enough—there are elements of the real invocation process that are either not specified by the standards or the standards provide freedom to the implementations to define and use their own extensions of the standards. As a result, when two applications successfully communicate with each other using WSDL and SOAP there is **always** a direct or indirect agreement about one or more items not covered by the standards and employed by these applications. It is these holes left by the standards that lead to interoperability problems.

A very good example of a core communication element for which the implementations need to make assumptions is data encoding, or how a particular data model is serialized into XML. In fact, SOAP defines a message format, but it does not enforce data encoding. There are two types of data encoding allowed by the standards: *encoded* and *literal*.

Encoded means that a certain data model is serialized using a certain set of rules (i.e. they can say that a hashtable is written in XML using this XML structure with these XML elements and these XML attributes). WSDL however cannot describe these rules, it will only provide an identifier for the encoding that the two parties must recognize and have their own logic (which is not formally specified by the standards) for this encoding. So what if the WSDL specifies an encoding that the Web service client cannot recognize—then the client surely cannot talk to this service. The bad thing is that even when both sides claim to support a given encoding there is still no guarantee that they will be able to talk to each other. This is so because the encoding rules are not 100% complete and unambiguous themselves and also because there is no formal mechanism to verify whether a piece of XML is a valid encoded XML.

Let's use "*SOAP Section 5 Encoding*" as an example, the most popular set of encoding rules and actually a synonym of encoded usage. *SOAP Section 5 Encoding* does not force the use of data types, it defines the abstract concept of nodes and edges that allow serialization of a graph data model. So an implementation of *SOAP Section 5 Encoding* cannot claim to marshal that data into native types of a programming language without having another implicit assumption being made.

The other option for data encoding – *Literal encoding* simply means that the parties do not agree about data encoding but they agree about the exact format of the XML data, usually specified through an XML Schema. The XML Schema provides a formal mechanism for validation and automatic marshalling but again the parties need an agreement that a specific format is mapped to a specific native language data type/structure. The XML Schema specifies data types but even this specification cannot guarantee that these types can be used flawlessly. For example, the XML Schema specifies that any number of digits after seconds can be coded into a `dateTime` data type. However, `java.util.Date` supports precision up to the nearest millisecond, while the .NET `Date` data type can represent a nanosecond—this could potentially cause interoperability problems. There are similar issues with floating point numbers and other data types.

WS Provisioning and WS Trust

In order for the TDI Web Service components to work with Web services which employ complex data types Java classes which represent these complex data types must be present in the TDI classpath. These Java classes define public getter and setter methods for their properties/attributes, which can in turn also be instances of such Java classes. In this way a Java representation of a complete hierarchy of complex data types can be created.

The Apache Axis library (on which the TDI Web Service components are based) provides a tool called *WSDL2Java*, which given a WSDL Web service definition can generate the corresponding Java classes. This tool is used to create the Java class files for the Web services standard WS-Provisioning and WS-Trust specifications using the corresponding WSDL files. After the Java class files are generated, they will be compiled and packaged in two JAR files to be dropped in the TDI classpath.

This enables TDI to be used as both a Web service client and a Web service provider for both WS-Provisioning and WS-Trust applications.

WS-Provisioning and WS-Trust support is based on the IBM Tivoli Directory Integrator Web Service components.

TDI supports WS-Provisioning specification version 0.7. This version of the WS-Provisioning specification along with its schema and WSDL Web service definition can be downloaded from http://www.ibm.com/developerworks/webservices/library/ws-provis/.

TDI also supports a related Web service definition, called WS-Trust - in particular the specification labeled "February 2005". The WS-Trust specification along with its schema and WSDL Web service definition can be downloaded from http://www.ibm.com/developerworks/library/specification/ws-trust/.

The WS-Provisioning package "wsprov.jar" JAR file contains the compiled Java classes which implement the data types used by WS-Provisioning. These Java

classes and the Web Services components will help you to develop or connect to a WS-Provisioning implementation. There are three Java packages in "wsprov.jar":

api._1._0.provisioning.ws.names.ibm
Implements types defined in the `api.xsd` schema, which is part of the WS-Provisioning specification

core._1._0.provisioning.ws.names.ibm
Implements types defined in the `core.xsd` schema, which is part of the WS-Provisioning specification

notify._1._0.provisioning.ws.names.ibm
Implements types defined in the "notification.xsd" schema, which is part of the WS-Provisioning specification

Mapping Java class names to WS-Provisioning XSD types

The name of each Java class matches the name of a WS-Provisioning schema type. For example, the `api._1._0.provisioning.ws.names.ibm.ListTargetsRequestType` Java class implements the type `ListTargetsRequestType` defined in the `urn:ibm:names:ws:provisioning:0.1:api` namespace.

wsprov.jar file contents

The wsprov.jar file contains Java classes generated using the Axis WSDL2Java tool. There were errors in some of the Java files generated by the WSDL2Java tool – these files were edited manually to fix these errors and then all sources were compiled and packed into the wsprov.jar file. The wsprov.jar file is located in the *<TDI_install_folder>*\jars\common" folder.

WS-Provisioning examples

1. Create the input for the `listTargets` operation in Javascript:

```
var listTargetsInput = new Packages.api._1._0.provisioning.ws.names
 .ibm.ListTargetsRequestType();
```

When serialized this "listTargetsInput" variable will look like this:

```
<ListTargetsRequest xsi:type="ns1:ListTargetsRequestType"
xmlns="urn:ibm:names:ws:provisioning:0.1:api"
xmlns:ns1="urn:ibm:names:ws:provisioning:0.1:api"/>
```

2. Create the output for the `listsTargets` operation in JavaScript:

```
var set = new
Packages.core._1._0.provisioning.ws.names.ibm.ProvisioningTargetSetType();
var provTargetType = new
Packages.core._1._0.provisioning.ws.names.ibm.ProvisioningTargetType();
var provTargets =
java.lang.reflect.Array.newInstance(provTargetType.getClass(), 1);
var id = new
Packages.core._1._0.provisioning.ws.names.ibm.ProvisioningIdentifierType();
id.setName("WindowsNTServerAccount");
provTargets[0] = provTargetType;
provTargets[0].setIdentifier(id);
var provTargetSchema = new
Packages.core._1._0.provisioning.ws.names.ibm.ProvisioningTargetSchema();
var schema = java.lang.reflect.Array.newInstance(provTargetSchema.getClass(),
1);
schema[0] = provTargetSchema;
schema[0].setLocation("http://myhost.com/myschema.xsd");
provTargets[0].setSchema(schema);
set.setProvisioningTarget(provTargets);
var response = new
Packages.api._1._0.provisioning.ws.names.ibm.ListTargetsResponseType();
response.setTargets(set);

var result = new
Packages.core._1._0.provisioning.ws.names.ibm.ProvisioningIteratedResultType(
);
result.setSize(1);
var status = new
Packages.core._1._0.provisioning.ws.names.ibm.ProvisioningRequestStatusType()
;
status.setCode(Packages.core._1._0.provisioning.ws.names.ibm.ProvisioningStat
usCode.success);
result.setStatus(status);
response.setResult(result);
```

When serialized, the response will be similar to the following lines:

```
<ns2:arg0 xsi:type="ns1:ListTargetsResponseType"
xmlns:ns1="urn:ibm:names:ws:provisioning:0.1:api"
xmlns:ns2="urn:ibm:names:ws:provisioning:0.1:psp">
 <ns1:targets xsi:type="ns3:ProvisioningTargetSetType"
xmlns:ns3="urn:ibm:names:ws:provisioning:0.1:core">
  <ns3:ProvisioningTarget xsi:type="ns3:ProvisioningTargetType">
`   <ns3:identifier xsi:type="ns3:ProvisioningIdentifierType"
name="WindowsNTServerAccount"/>
   <ns3:schema xsi:type="ns3:ProvisioningTargetSchema"
 location="http://myhost.com/myschema.xsd"/>
   </ns3:ProvisioningTarget>
  </ns1:targets>
  <ns1:result xsi:type="ns4:ProvisioningIteratedResultType"
 remaining="0" size="1"
 xmlns:ns4="urn:ibm:names:ws:provisioning:0.1:core">
  <ns4:status xsi:type="ns4:ProvisioningRequestStatusType">
   <ns4:code xsi:type="ns4:ProvisioningStatusCode">
  <ns4:value xsi:type="xsd:string">success</ns4:value>
  </ns4:code>
   </ns4:status>
  </ns1:result>
 </ns2:arg0>
```

Chapter 5. TDI Examples

In order to work with examples complementing this manual, you can go to the *root_directory*/examples directory in the installation directories.

Scripted Outlook Connector using COMProxy

The Connector described here is a re-implementation in JavaScript of the Outlook Connector written in VBScript, in the previous example. You can find the code in the *installation_directory/examples/MSOutlook* directory.

This example shows how you can manipulate your Outlook Contacts using COMProxy. It is an example of an `ibmdi.scriptconnector`, and shows how you can create a script connector that supports add, iterate, update, lookup, and delete modes.

COMProxy allows you to call COM Automation components from Java. Java Native Interface (JNI) makes native calls into the COM and Win32 libraries. COMProxy makes use of Object Linking and Embedding (OLE) Automation under the wraps (also knows as late binding) to make calls to COM objects/interfaces. COMProxy also supports Moniker URLs. To obtain a handle to a COMProxy instance use the `system.createCOMInstance( )` method.

Note: Full documentation for the COMProxy is available in the Javadocs under the `com.ibm.di.automation` package

The script code is provided below if you would like to create your own script connector and input this data. The file *msoutlook.xml* is a TDI Config file with the Connector already entered for you. If you open `msoutlook.xml` you will find a scriptconnector called **msoutlook** that contains this script information (″config″->″script″).

You could also copy the MSOutlook.jar file to the *installation_directory/jars/connectors* directory, after which it will appear in your list of available connectors to inherit from.

Example code

```
//
// This script implements all the necessary functions for accessing
// the Contacts register in MS Outlook.
// Assumes that the number of entries in contact folder is constant for the run

ol = system.createCOMInstance("Outlook.Application");

ns = COMProxy.call(ol,"GetNameSpace","MAPI");

contacts = COMProxy.call(ns.toObject(),"getDefaultFolder",10);

var item;
var counter = 0;
var oldstring="";

var decode="";

var outlookEntry = system.newEntry();

function selectEntries(){
 counter = 0;
}

function getNextEntry (){
 ol = system.createCOMInstance("Outlook.Application");

 ns = COMProxy.call(ol,"GetNameSpace","MAPI");

 contacts = COMProxy.call(ns.toObject(),"getDefaultFolder",10);
```

```
 items = COMProxy.call(contacts.toObject(),"Items");

 count = COMProxy.get(items.toObject(),"count");

 counter++;
 if(counter > count){
  result.setStatus(0);
  result.setMessage("End of input");
 }else{
  item = COMProxy.call(items.toObject(),"Item",counter);
  populateEntry();
 }
}

function findEntry (){
 flt = "[" + search.getFirstCriteriaName() + "] = " + search.getFirstCriteriaValue();
 items = COMProxy.call(contacts.toObject(),"Items");
 item = COMProxy.call(items.toObject(),"Find",flt);
 if (item == null){
  result.setStatus(0)
  result.setMessage("Not found" + "--->["+ flt + "]");
 }
 else
  populateEntry();
}

function modEntry (){
    populateItem();
    COMProxy.call(item.toObject(),"Save");
}

function deleteEntry (){
 COMProxy.call(item.toObject(),"Delete");
}

function putEntry (){
 items = COMProxy.call(contacts.toObject(),"Items");
 item = COMProxy.get(items.toObject(),"Add");
 if(item==null){
  result.setStatus(2)
  result.setMessage("Unabled to create item");
  return;
 }
 oldString = entry.getString("FullName");
 COMProxy.put(item.toObject(),"FileAs",oldString);
 populateItem();
 COMProxy.call(item.toObject(),"Save");
}

function populateEntry (){
 entry.setAttribute("FileAs", COMProxy.get(item.toObject(),"FileAs"));
 entry.setAttribute("FullName", COMProxy.get(item.toObject(),"FullName"));
 entry.setAttribute("Email1Address", COMProxy.get(item.toObject(),"Email1Address"));
 entry.setAttribute("Birthday", COMProxy.get(item.toObject(),"Birthday"));

 entry.setAttribute("BusinessAddress", COMProxy.get(item.toObject(),"BusinessAddress"));
 entry.setAttribute("BusinessTelephoneNumber", COMProxy.get(item.toObject(),"BusinessTelephoneNumber"));
 entry.setAttribute("BusinessFaxNumber", COMProxy.get(item.toObject(),"BusinessFaxNumber"));
 entry.setAttribute("CompanyName", COMProxy.get(item.toObject(),"CompanyName"));
 entry.setAttribute("JobTitle", COMProxy.get(item.toObject(),"JobTitle"));

 entry.setAttribute("HomeAddress", COMProxy.get(item.toObject(),"HomeAddress"));
 entry.setAttribute("HomeTelephoneNumber", COMProxy.get(item.toObject(),"HomeTelephoneNumber"));
 entry.setAttribute("HomeFaxNumber", COMProxy.get(item.toObject(),"HomeFaxNumber"));

 entry.setAttribute("MobileTelephoneNumber", COMProxy.get(item.toObject(),"MobileTelephoneNumber"));

 entry.setAttribute("Categories", COMProxy.get(item.toObject(),"Categories"));
 entry.setAttribute("LastModificationTime", COMProxy.get(item.toObject(),"LastModificationTime"));
        outlookEntry = entry.clone(entry);
}

function populateItem (){
 outlookEntry.merge(entry);
        COMProxy.put(item.toObject(),"FileAs", outlookEntry.getString("FileAs"));
 COMProxy.put(item.toObject(),"FullName", outlookEntry.getString("FullName"));
 COMProxy.put(item.toObject(),"Email1Address", outlookEntry.getString("Email1Address"));
 COMProxy.put(item.toObject(),"BusinessAddress", outlookEntry.getString("BusinessAddress"));
 COMProxy.put(item.toObject(),"BusinessTelephoneNumber", outlookEntry.getString("BusinessTelephoneNumber"));
 COMProxy.put(item.toObject(),"BusinessFaxNumber",outlookEntry.getString("BusinessFaxNumber"));
 COMProxy.put(item.toObject(),"JobTitle", outlookEntry.getString("JobTitle")) ;
 COMProxy.put(item.toObject(),"CompanyName", outlookEntry.getString("CompanyName")) ;
 COMProxy.put(item.toObject(),"HomeAddress",outlookEntry.getString("HomeAddress")) ;
 COMProxy.put(item.toObject(),"HomeTelephoneNumber", outlookEntry.getString("HomeTelephoneNumber")) ;
 COMProxy.put(item.toObject(),"HomeFaxNumber", outlookEntry.getString("HomeFaxNumber")) ;
 COMProxy.put(item.toObject(),"Categories", outlookEntry.getString("Categories"));
 if (outlookEntry.getString("Birthday")!=null && !outlookEntry.getString("Birthday").equals(" "))
  COMProxy.put(item.toObject(),"Birthday", outlookEntry.getString("Birthday"));
}
```

See also

"Script Connector" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

JavaScript Connector

This example shows how to write a Connector using a script language. The example uses JavaScript and shows the objects available and how they are used.

This is about as simple a connector as you can create. It supports Iterate mode only. All it does is count from 0 - 100. When the counter reaches 100, a return message is sent ″End of input″.

Example code

```
//
// Place intialiazation code before function declarations.
//
var counter = 0;

function selectEntries()
{
}

function getNextEntry ()
{
 if (counter > 100) {
  result.setStatus (1);
  result.setMessage ("End of input");
  return;
 }

 entry.setAttribute ("counter", counter);
 counter++;
}

function modEntry ()
{
}

function deleteEntry ()
{
}

function findEntry ()
{
}

function putEntry ()
{
}
```

See also

″Script Connector″ in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

JavaScript Parser

This example shows how to write a Parser using a script language. The example uses JavaScript and shows the objects available and how they are used.

Example code

```
//
// This is a simple Parser that returns one line at a time from
// the input stream.
//

var counter = 0;

function writeEntry ()
{
 var names = entry.getAttributeNames();
 for (i = 0; i < names.length; i++) {
  out.write (name[i], entry.getString(name[i]));
  out.write (13);
  out.write (10);
 }
}
```

```
function readEntry ()
{
 var str = inp.readLine();

 if (str == null) {
  result.setStatus (0);
  result.setMessage ("End of input");
  return;
 }

 counter++;

 entry.setAttribute ("line", str);
 result.setStatus (1);
}
```

See also

"Script Parser" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Publishing a TDI AssemblyLine into a Service Component Architecture (SCA) infrastructure

SCA is a service oriented component model for defining and invoking business services. An example is provided that illustrates how to start a TDI AssemblyLine using the SCA infrastructure. The example consists of two SCA modules wired together using SCA binding. The TDI Module exposes the Start AssemblyLine functionality using the TDI Server API, and the Sample Module consumes this service to start AssemblyLines. This example uses Websphere Integration Developer 6.0 to develop the application, and Websphere Process Server 6.0 to host the example. During TDI installation, the examples are placed under `<TDI_install>/examples/sca`.

Components of the SCA example

The business objects (BOs) for this example include:

Server Info

Attributes:

- serverIP (String)
- configName (String)
- alName (String)

This BO is used to transmit the details of the server from the UI to the TDI SCA module.

Execution Result Attributes:

- result1 (String)
- result2 (String)

This BO transmits the result of invoking the TDI SCA module back to the user interface.

The following are the Interfaces for the example:**TDI_Invoke**

- Operation name — startAL
- Input — Execution Result
- Output — ServerInfo

This interface exposes the TDI service.

```
TDI_Invoke interface -
  public interface TDI_Invoke {  public static final String PORTTYPE_NAME = "{http://TDI_Common/TDI_Invoke}TDI_Invoke";
  public  DataObject startAL(DataObject serverInfo)
```

User_Input

- Operation name — getuserInput
- Input — ExecutionResult
- Output — ServerInfo

This is the interface through which the Java Server Page (JSP) application invokes the Sample Module and passes information about the AssemblyLine that is to be started.**TDI_Module**

The TDI_Module implements the TDI_Invoke interface defined above. The Assembly Line Editor of the Websphere Integration Developer (WID) can be used to associate the interface with the module. The TDI Server API is used to start Assembly Lines on the TDI Server. These are relevant snippets of code from the Java implementation of the startAL service:

```
public DataObject startAL(DataObject serverInfo) {
//Create the ExecutionResult Business Object.
 ServiceManager serviceManager = new ServiceManager();
 BOFactory bof = (BOFactory)
serviceManager.locateService("com/ibm/websphere/bo/BOFactory");
execStatus = bof.create("http://TDI_Common", "ExceutionResult"); //

// Create a TDI Session object . serverIP, configPath and assemblyLine name are attributes of serverInfo object passed to this method.
 SessionFactory sf = (SessionFactory) Naming.lookup("rmi://" + serverIP + ":" + portNo + "/SessionFactory");
 session = sf.createSession();

//Try to obtain ConfigInstance object (ie .if this config has already been started on the server)
//else start the Config on the server.
String configID = com.ibm.di.api.syslog.LogUtils.getCleanConfigId(configPath);
configInstance = session.getConfigInstance(configID);
if (configInstance == null) {
 configInstance = session.startConfigInstance(configPath);
}

//Start the AssemblyLine , set status in ExecutionResult Business Object and return BO
 assemblyLine = configInstance.startAssemblyLine(assemLine);
 execStatus.set("result1", "AssemblyLine Started");
 return execStatus;
}
```

Sample Call Module

The Sample Call Module is a sample SCA module that calls the TDI module. It also contains a JSP application that is bundled with the Sample SCA module, using a Stand Alone Reference. These are relevant snippets of code from the SampleCallImpl.java class:

```
Locate the TDI service and call Start AL method
TDI_Invoke invokeTDI = (TDI_Invoke) locateService_TDI_InvokePartner();
execStatus =(DataObject) invokeTDI.startAL(serverInput);
```

The code for locateService_TDI_InvokePartner() is provided below. "TDI_InvokePartner" is the name given to the SCA binding in the AssemblyLine editor.

```
public TDI_Invoke locateService_TDI_InvokePartner() {
 return (TDI_Invoke) ServiceManager.INSTANCE.locateService("TDI_InvokePartner");
}
```

Each of the above modules is created independently and then connected to the other modules using the Websphere Integration Developer's Assembly Editor.

Running the example

1. Import the TDI_SCA_ProjectInterchange.zip file from <tdi_installation>/examples/sca into the Websphere Integration Developer workspace.
2. Import the following TDI jars as utility jars into the TDI_ModuleApp. When importing these jars as utility jars to the TDI_Module project, specify the option to copy all jars to the EAR file.
   - derby.jar
   - derbyclient.jar
   - derbynet.jar
   - derbytools.jar

- diserverapi.jar
- diserverapirmi.jar
- icu4j_3_6.jar
- log4j-1.2.8.jar
- miconfig.jar
- miserver.jar
- mmconfig.jar
- tdiresource.jar

3. Refresh the list of jars being referenced in the TDI_ModuleEJB project (the J2EE perspective of WID).
4. Perform a complete build and resolve any errors.
5. Deploy the SampleCallModuleApp and TDI_ModuleApp projects on the Process Server.
6. Start the TDI server on default port 1099.
7. Run the Sample JSP to get the user input page and click to invoke TDI Service.

A javascript window indicates whether the AssemblyLine has started or not.

Extending the SCA Example

The example shows how to run an AssemblyLine using a SCA TDI module. The SCA TDI module can be extended to include any functionality that is exposed by the Server API. To extend the SCA TDI module, modify the TDI_Invoke interface to include the new functionality. Next update the Java implementation in the TDI_Module component. For more information on the Server API see "Appendix C" in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Writing a new Connector Interface

There are generally two ways of writing new Connectors. The first way is to write a script that implements a set of functions using your favorite scripting language. The second way is to write the Connector Interface using Java.

Script-based Connector

Read the documentation for the "Script Connector" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* and also take a look at "Scripted Outlook Connector using COMProxy" on page 207. The example provides the necessary information to roll your own Connector.

ACT Connectors

An ACT connector allows TDI users to send CBE messages to ACT, and ACT triggers can invoke TDI ALs through TDI event notifications. TDI ships with an ACT example in `<TDI installation folder>/examples/ACT`.

Java-based Connector

Learning by example is probably the best way to learn new things. Here, knowledge to the Java programming language is really useful. Source code for a Connector that can read directories and return filenames contained in those directories is provided in the *install_directory/examples/connector_java* directory, and the process of transforming this example into a Connector that TDI can used is described in the appendix, "Implementing your own Components" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Copying directories with the LDAP Connector

Usually, the best way to do this is to export using LDIF from one server, and then importing it to the other. However, if you want to do this using IBM Tivoli Directory Integrator to get some more control, read on.

You can do the copying by having a very simple AssemblyLine containing two Connectors:

1. An Iterator Connector (reading the source directory) using a one level scope
2. An AddOnly Connector (updating the target directory)

Naturally a recursion must be introduced to copy the entire tree. We start the same AssemblyLine over and over, but the search base can be set to whatever DN has just been inserted in the target directory:

```
for all entries returned in current level
 add entry in target system
 if (success)
 start same AssemblyLine with search base level set to current's entry DN
```

The starting of AssemblyLines can be made parallel to make the processing faster. Of course the number of threads can explode but it is possible to control it. Because the Config Editor sets the *-w* parameter, you must start IBM Tivoli Directory Integrator from the command line.

ibmdisrv -rDumpDir -cDumpDirectory.cfg

You must use these options for code customizing:

- The AddOnly Connector (*on_success hook:* starts a new AssemblyLine (with an initial entry that the Prolog picks up).
- In the AssemblyLine Prolog (*Before Connectors Initialized*).

Chapter 6. TDI command-line options

Command-line options must have their value followed immediately after the option. Do not use a space between the option and the value. There are options for:

- "Config Editor"
- "Server"
- "Command-line Interface (CLI)" on page 218

Config Editor

```
ibmditk -v
```

Show the Config Editor and components versions, and exits.

```
ibmditk [-s solutiondir] [-?] [filename]
```

-s Specifies the working directory where the solution is located. All relative file references in TDI and in your Configs and so forth will be relative to this location. Must be the first parameter specified.

-? Prints a *usage* message, showing all options in brief.

filename
You can have 0 or more filenames that contain legal configuration files.

Server

The following command-line options are for the IBM Tivoli Directory Integrator Server (`ibmdisrv [options]`):

Example:

```
ibmdisrv -c"C:\demos\rs.xml" -r"Access2LDAP" -l"c:\metamerge\mydemo.log"
```

Notes:

1. There is no space between the option letter and the value. Use quotes to save against possible spaces or commas in the values.
2. The Windows Shell executive allows a maximum of nine (9) arguments, from the list below. There aren't any limitations on other platforms.

-s <dir>
Specifies the working directory where the solution is located. All relative file references in TDI and in your Configs and so forth will be relative to this location. Must be the first parameter specified.

-f <extProp1=file1, extProp2=file2>

-c <file...>
Configuration file(s). If you don't specify this option, the items in the Autostart folder will be loaded and started (unless suppressed by specifying **–D**). Wildcards, as in `*.xml`, are allowed too.

-n <encoding>
Encoding to be used to write Config files. This must be a valid character set identifier valid in Java2; refer to the IANA Charset Registry

(http://www.iana.org/assignments/character-sets) for the full list of values. Note that Java2 only supports a subset of those.

-r <al...>
List of AssemblyLine names to start. To start AssemblyLine **a** and **b**, use the command **-r a b**. Other syntaxes are supported as well: **-ra,b**; **-ra -rb**.

Note: If you use includes and namespaces, the AssemblyLine can be myNamespace:/AssemblyLines/alName (assuming namespace **myNamespace** and AssemblyLine name **alName**).

-t <eh...>
List of EventHandler names to start. To start EventHandler **a** and **b**, use the command **-t a b**. Other syntaxes are supported as well: **-ta,b**; **-ta -tb**.

-T<name>
Enable JLOG-style tracing to file `trace<name>.log`, in directory `<Tivoli_Common_Dir>/TDI/logs/`. Default is trace to memory (from which it can be retrieved by the traceback routines of JFFDC in case of an unhandled exception.)

-D Flag to disable startup of EventHandlers and/or items in the Autostart folder.

-w If -r (or -t) is specified then this flag causes IBM Tivoli Directory Integrator to wait for each AssemblyLine's EventHandler to complete before starting the next. If this flag is not specified then IBM Tivoli Directory Integrator starts all AssemblyLines specified by the -r parameter in parallel. When the last AssemblyLine and explicitly started EventHandler has finished, the server stops.

Note: The server stops when it has no active threads. However, we have experienced that with Perl, the Perl task is counted as an active thread. Use **-w** to force IBM Tivoli Directory Integrator to stop after the last AssemblyLine has finished.

-e Specifying this option causes the Server to run in Secure mode. Using the master password specific to this server, it will decrypt and encrypt all Config files as well as the Server API Registry.

-v Show version information and exit. This is logged in the log file only.

-P <password>
Password if configuration file(s) is/are encrypted.

-p Dump Java properties on startup. Note that you still must provide a configuration file, which is read before Java properties are dumped.

-d Start a "daemon", or *Config Instance* on this machine. The server starts one thread for each Config specified, plus one extra thread. None of the threads will terminate.

-Z Instruct the AssemblyLine to act as if the Checkpoint Table was empty. All ALs given on the command line clears their checkpoint tables.

-q Takes 1 argument, mode. Mode=1 means run in record mode, mode=2 means run in playback mode.

-l <file>
Log file (default console output). Does very little as few messages go to the console. To change the log file for most of the logging, change log4j.properties.

-R Disables the Remote API, regardless of the setting in `global.properties`.

-W All Configs are started in the same thread, and they do not terminate but rather wait forever.

-S This option is for internal use for communication between the Config Editor and a Server only; it is used to pass Config Files between them. Do not use this option yourselves.

-f <*extProp1***=***file1***,** *extProp2***=***file2***>**
Where *extProp* is the name of the external Property Store. *file* specifies from where to read the properties. This option specifies a user-defined, external Property Store that can be entered when starting a TDI server. This new optional command-line parameter [-f] can be used with the "ibmdisrv" server startup scripts. *extProp* is the name of the external Property Store. *file* specifies from where to read the properties. When the [-f] option is used to specify a properties file from the command line, the server changes the Property Store configuration in memory only, i.e. the server does not make this change permanent by changing the TDI Config file on disk – this change is valid for the current run of the TDI server.

If any property files are specified at the command line, they are valid only for the Config Instances specified with the [-c] command-line option (which are loaded on TDI server startup). The property files specified at the command line do not have any impact on Config Instances which have not been explicitly named with the [-c] command-line option (these can be Config Instances loaded by remote Server API client for example).

If a Property Store whose name is specified with the [-f] command-line switch cannot be found in a Config Instance, an error message is logged in the server log (ibmdi.log in the Install-directory). When a Property Store name is specified more than once with the [-f] command-line switch then there are two effects: (1) a warning message is logged, and (2) the file specified last will take effect. This feature is implemented in the com.ibm.di.server.RS Java class (referenced by way of the main variable when scripting). After the reload() method is called, the MetamergeConfig object is loaded, and for each Property Store specified on the command line, the corresponding PropertyStoreConfig object is updated.

Note:

Although Copy/Paste of Config objects (ALs, Connectors, FCs, and so forth) are fully supported. You can easily copy ALs and components and then paste them into another Config. You can also exchange ALs and components using IM chats, e-mails and text files, because the copy-buffer is filled with the TDI Config XML definition of the selected item. This makes passing stuff around simple and easy, and is a great tool for support and online assistance (for example, ICT/NotesBuddy, forums, ...).

Note:
Make sure you select the entire <MetamergeConfig> node in your copy command, including the start and end tags.

-i This options specifies that the TDI server ignores any properties from the `global.properties` file, and reads only the `solution.properties` file. This option can be used when the `global.properties` file is unreadable - for example, when the encoding the TDI server is started with is different from the encoding of `global.properties`.

-? Prints a *usage* message, showing all options in brief.

When IBM Tivoli Directory Integrator ends it returns one of the following exit codes:

0 User started IBM Tivoli Directory Integrator with -v parameter (show info and exit)

1

- Cannot open log file (-l parameter)
- Cannot open Config file
- Stopped by admin request

2 Exit after auto-run. When you start IBM Tivoli Directory Integrator specifying -w IBM Tivoli Directory Integrator runs the AssemblyLines specified by the -r parameter and then exits.

Note: AssemblyLines run from the Config Editor are started in a different way and will not exit with status 2.

9 License expired or invalid (obsolete).

Command-line Interface (CLI)

TDI 6.1.1 offers a stand-alone command-line tool for accessing and controlling a local or remote TDI Server. This tool allows you to connect to a running Server and then start and stop Configs and ALs, reload Configs, query status, read and set properties, send events and shut down the Server. This program is located in the <*TDInstallation*>/bin folder and is called **tdisrvctl**. It offers a complete usage display if you fire it up without parameters (or with invalid ones).

The CLI is described in more detail in *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

Note: The pathname for the Config instance that you enter when using **tdisrvctl** commands should exactly match the path you get when you type **tdisrvctl -op status**. Pathnames are case-sensitive for all commands that have to query the Config instance. For example, you cannot get valid commands typing c:\rs.xml when the path given by **tdisrvctl -op status** is **C:\rs.xml**

Some typical examples of usage are:

- `tdisrvctl –op status`

  which results in a listing of loaded Configs, as well as the status of all ALs and EHs.
- `tdisrvctl –op report –c "c:/tdiwork/MyConfig.xml"`

  which produces a textual report of the Config specified.

Note: The default host and port are used (localhost:1099). Also, for the “op report” example above, note that you must enter the full path to the Config, except if this Config is located in the TDI Configs folder (specified by the api.config.folder Global/Solution property). You can request the list of Configs stored in the Configs folder with the following command

```
tdisrvctl –op report –l
```

This will return a list of Config files found for that Server.

Appendix A. Enhancements and changes in 6.1.1

Introduction

This chapter is an outline of significant changes and new features of this release. This outline of new and changed features gives you a quick overview on how changes in this version have been designed to improve IBM Tivoli Directory Integrator 6.1.1.

New features

Platform support for i5/OS and HP Itanium

The TDI server Administration and Monitoring Console (AMC) and Command Line Interface (CLI) can run on i5 and HP Itanium platforms.

New debugging run modes

You are returned into the debugger if AssemblyLine (AL) execution fails. This allows the you to inspect the status of the AL dynamically. The traditional “fire and forget” mechanism is still available.

Remote debugging

The AL stepper and debugger can now be used against remote TDI servers.

ITM integration

An example is provided that illustrates how TDI can be integrated with ITM using the JMX management interface. TDI servers and TDI configs can be monitored by ITM, and events can be sent from TDI into the ITM infrastructure.

Enable sharing of Iterator connection

A connector in iterator mode can now be reused throughout the AL in other modes as well. Reuse of connectors is possible in previous TDI versions as well, but the behavior has not been predictable when reusing iterators.

ACT engine

The Autonomic Active Correlation Technology server is now embedded in TDI. An ACT connector allows TDI users to send CBE messages to ACT, and ACT triggers can invoke TDI ALs through TDI event notifications.

RAC connector

TDI can now communicate with the Autonomic Remote Agent Controller. Data messages conforming to the CBE format can be sent to and received from RAC.

GLA connector

The Autonomic Generic Log Adapter is embedded in TDI and enables TDI to use the existing parsers and sensors that GLA supports. GLA sends CBE messages into TDI, and those CBEs can be used for any TDI purpose, either restructured into other formats or sent on to RAC, CEI, or other CBE compliant targets.

Integration with SCA

An example is provided that illustrates how to publish a TDI AssemblyLine into an SCA infrastructure. A small Java application uses the TDI server API to communicate with the AssemblyLine, and an SCA descriptor is created that defines the bindings into the AL.

Added solution information into Config

The TDI developer can now add information to the config that describes which ALs and properties are to be displayed with AMC, thereby reducing the effort for the AMC admin to define this in AMC.

Enhancements

Delta engine performance improved

A new algorithm is implemented that increases the performance for data sources that contain non-changed data as well as changed data. The speed of dealing with non-changed data has increased by several factors.

Default case for Switch

The switch component now supports a default case that always is executed.

Maintain Hook inheritance when Enabled flag changed

Domino Change Detection Connector

Harmonize the "assured delivery" mode Add option to return.

Documentation of JMS interface to Sonic MQ

An example is provided that describes how to configure the JMS connectors to work with Sonic.

Allow aliases for Config instance name

Config instance names have until now been long strings that included the name of the config xml file. Interfacing from AMC and commandline is now much more user friendly.

Re-introduce Castor “Java to XML” and “XML to Java” Function Components (FCs)

The powerful XML manipulation components were removed in TDI 6.1 due to Open Source liability reasons. They are now back in TDI after a complete review.

All DDL used to create SystemStore tables is externalized

The SQL Data Definition Language calls (DDL) needed to set up specific TDI SystemStore tables is now externalized to the Global/System properties, allowing you to customize these to fit any JDBC-compliant RDBMS that they wish to use.

Administration and Monitoring Console (AMC) improvements

- New welcome window with quick links to common functions
- Quick-discovery and creation of config views
- Support for TDI 6.0 Servers

The AMC admin can quickly create default config views or choose to use the information in the (also new) solution interface.

Wherever AssemblyLines can be started in AMC, it is now possible to specify any operation – if defined in the target AL – as well as input data according to the schema defined for the operation.

Support for AL operations: Wherever AssemblyLines can be started in AMC, it is now possible to specify any operation — if defined in the target AL — as well as input data according to the schema defined for the operation.

GUI improvements: Several additions to the monitoring window provide a better overview of the TDI servers monitored. Automatic refresh is added to some windows.

Better arrangement of actions in Action Monitor (AM): The management of AM actions is completely reworked.

Action manager improvements

AM rules are suspended (Any rule triggered by "API termination" will be triggered) when AM loses connection with a TDI server, and the rules are enabled when the connection to the TDI server is re-established.

New SAP change connector supports IDoc/ALE

This connector enables TDI to receive changes from SAP as they happen in SAP for any type of business object supported by SAP.

New TAM connector

This Connector enables TDI to read and write TAM account information directly.

Remote command line FC extended

The remote command line component can also connect to i5 and z/OS servers.

zLDAP now supported as password store for password synch

The TDI password-catcher plug-ins can now be configured to store this information in the directory server running on z/OS.

Appendix B. Using CloudScape database

These are some of the commands used for the CloudScape database:

- `start -p portnumber [-ld]`
- `shutdown [-h host][-p portnumber]`
- `dbstart databaseDirectory [-b bootPassword][-ld][-ea encryptionAlgorithm] [-ep encryptionProvider] [-u user password][-h host][-p portnumber]`
- `dbshutdown databaseDirectory [-h host][-p portnumber]`
- `testconnection [-d databaseDirectory] [-u user password] [-h host] [-p portnumber]`
- `sysinfo [-h host][-p portnumber]`
- `conpool min max [-d databaseDirectory][-h host][-p portnumber]`
- `logconnections {on|off}[-h host][-p portnumber]`
- `maxthreads max[-h host][-p portnumber]`
- `timeslice milliseconds[-h host][-p portnumber]`
- `trace {on|off} [-s session id][-h host][-p portnumber]`
- `tracedirectory traceDirectory[-h host][-p portnumber]`

Embedded CloudScape

The CloudScape database engine by default runs inside IBM Tivoli Directory Integrator (embedded), as opposed to running in its own JVM (networked mode). When you run CloudScape in networked mode, multiple instances of IBM Tivoli Directory Integrator can access the same database server, because all database requests are sent to a single CloudScape server. In embedded mode there is no way for another process to get access to the CloudScape database engine that runs inside IBM Tivoli Directory Integrator

In networked mode you can also start multiple databases and have a single CloudScape server serve multiple instances of IBM Tivoli Directory Integrator.

Overriding the CloudScape defaults

Modify the global.properties file, and edit the section that specifically says `Location of the database (networked mode)`. Comment out the preceding section that says `Location of the database (embedded mode)` to override the CloudScape creation, usage and shutdown defaults.

The internal Store Factory in IBM Tivoli Directory Integrator that manages the database connections detects that CloudScape is not running in embedded mode, and does not shutdown any database you open from the Config Editor or the server.

Much more detailed information about the usage of CloudScape and the System Store is available in the *System Store* chapter in the*IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

Appendix C. Increasing the memory available to the Virtual Machine

While processing large sets of data, especially when using memory-based components like the XML Parser, you may run out of memory in the Java Virtual Machine even though you have more memory available in your machine. In such a case it may help to increase the heap size in IBM Tivoli Directory Integrator

Windows platforms

Edit *ibmdisrv.bat* in the IBM Tivoli Directory Integrator install directory to adjust the existing -Xms16m option to -Xms254m -Xmx1024m in the next to last line of the file (i.e. the line that invokes java).

Note: -Xms is the initial heap size in bytes and -Xmx is the maximum heap size in bytes. You can set these values according to your needs.

This will have no effect if you are trying to run an AssemblyLine with a memory problem from the Config Editor (ibmditk), as the Config Editor starts a new instance of the JVM to run the AssemblyLine; with default parameters. In order to accommodate this situation, you need to do the following:

1. Edit the global.properties or solution.properties file to alter the settings of com.ibm.di.javacmd to refer to a batch file. (for example, com.ibm.di.javacmd=c:\Program Files\IBM\TDI\V6.1\myjava.bat)
2. Create a command file (the aforementioned c:\Program Files\IBM\TDI\V6.1\myjava.bat) containing the appropriate Java invocation command, like `"javaw" -Xms254m -Xmx1024M %*`

Now the CE will use the modified JVM invocation, with increased heap size.

Unix/Linux platforms

Edit *ibmdisrv* in the IBM Tivoli Directory Integrator install directory to adjust the existing -Xms16m option to -Xms254m -Xmx1024m in the last line of the file (i.e. the line that invokes java).

Note: -Xms is the initial heap size in bytes and -Xmx is the maximum heap size in bytes. You can set these values according to your needs.

This will have no effect if you are trying to run an AssemblyLine with a memory problem from the Config Editor (ibmditk), as the Config Editor starts a new instance of the JVM to run the AssemblyLine; with default parameters. In order to accommodate this situation, you need to do the following:

1. Edit the global.properties or solution.properties file to alter the settings of com.ibm.di.javacmd to refer to a batch file. (for example, com.ibm.di.javacmd=/opt/IBM/TDI/V6.1/myjava.bat)
2. Create a command file (the aforementioned /opt/IBM/TDI/V6.1/myjava.bat) containing the appropriate Java invocation command, like `"java" -Xms254m -Xmx1024M $*`

Now the CE will use the modified JVM invocation, with increased heap size.

Appendix D. Double byte character sets in IBM Tivoli Directory Integrator

IBM Tivoli Directory Integrator is written in Java which in turn supports Unicode (double byte) character sets. However, external components such as drivers might not support the set. The prevalent scripting engine used in IBM Tivoli Directory Integrator IBM JSEngine (which implements the JavaScript language) is known to support double byte character sets correctly.

Some files, when UTF-8, UTF-16 or UTF-32 encoded, may contain a Byte Order Marker (BOM) at the beginning of the file. A BOM is the encoding of the character 0xFEFF. This can be used as a signature for the encoding used. The TDI File Connector does not recognize a BOM.

If you try to read a file with a BOM, you should add this code to for example, the 'Before Selection' Hook of the connector:

```
var bom = thisConnector.connector.getParser().getReader().read(); // skip the BOM = 65279
```

This code will read and skip the BOM, assuming that you have specified the correct character set for the parser.

Some care must be taken with the http protocol; see *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*, section about character sets encoding in the description of the HTTP Parser for more details.

Scripting languages supported by WSH (Windows Scripting Host), like JScript and VBScript might cause problems when using Unicode character sets.

Appendix E. Dictionary of terms

IBM Tivoli Directory Integrator terms

Action Manager (AM)
Action Manager is a stand-alone Java application used to configure failure-response behavior for TDI 6.1.1 solutions. AM executes *rules* defined with AMC v.3. An AM rule consists of one or more *triggers* that define a "failure" situation – such as the termination of an AL that should not stop running, or if an AL has not been executed within a given time period, and so forth Furthermore, each rule also defines *actions* to be carried out in case of this "failure". Actions include operations like sending events or e-mail, starting ALs (locally or remotely) and changing configuration settings. Action Manager requires TDI 6.1.1 and AMC v.3.

Accumulator
A special object that can be set in a Task Call Block (TCB) for use when starting another AssemblyLine either via a scripted call, or a component like the AssemblyLine Connector or the AssemblyLine FC. The Accumulator is either a collection of Work Entry objects handled by the called AL, or it is a component that is called to output each Entry. Accumulator handling is done at the end of each AssemblyLine Cycle.

Adapter
Adapter is a word is used in many contexts and with different meanings. A *TDI Adapter* refers to an AssemblyLine that is "packaged" as a single Connector. Creating a TDI Adapter requires setting up an AssemblyLine that is written to perform (and expose) one or more business related tasks. Each task is defined as an AssemblyLine Operation (for example, 'EnableAccount', or 'ReturnGroupMembers'). This AL can then be *published* for sharing, and can be leveraged by the AssemblyLine Connector which offers mode settings reflecting these operations[12].

AL Shorthand for AssemblyLine.

Administration and Monitoring Console (AMC)
AMC is a browser-based console for managing and monitoring TDI solutions. AMC version 3, which is part of the *TDI 6.1* and later releases, runs on the WebSphere Application Server (enterprise and express versions), as well as Tomcat. Each AMC version is designed to work with a specific release of TDI and is incompatible with other versions. AMC v.3 is designed for *TDI 6.1*, AMC v.2 works with *TDI 6.0* and AMC v.1 runs with *TDI 5.2*.

API Application Program Interface. A way of programmatically (local or networked) calling another application, as opposed to using a command-line or a shell script.

Appender
Appender is a log4j term (a third party Java library) for a module that directs log-messages to a certain device or repository. In IBM Tivoli Directory Integrator you control logging for your AssemblyLines by creating and configuring *Appenders*, either under the Logging tab of a

12. AL Operations are also accessible via the AssemblyLine FC.

specific AL, or under Config -> Logging in the Config Browser to control how all AssemblyLines in the Config do their logging.

AssemblyLine (AL)
The basic *unit-of-work* in a TDI solution. Each AL runs as a JVM thread in the Server and is made up of a series of AssemblyLine components (one or more Connectors, Functions, Scripts, Attribute Maps and Branches) linked together and driven by the built-in workflow of the AssemblyLine.

AssemblyLine Component
This term denotes an TDI component used to construct AssemblyLines. The possible Components are:

- Connectors
- Function Components
- Script Component
- Attribute Map Component
- Branches (including Loops and Switches)

The components list in an AssemblyLine is divided into two sections: *Feeds* where the Work Entry for each AL cycle is created from input data by a Connector in Iterator or Server mode, and the *Flow* section that holds the Connectors (in any mode except Server), Functions, Attribute Maps and Scripts providing the additional data access and processing.

AssemblyLine Operation
A business task that is implemented by an AssemblyLine and published via its Operations tab. Each Operation can have its own Input and Output Attributes Maps for defining the parameters expected when this Operation is invoked (Input Map), as well as those returned (Output Map). This is also called the *Schema* of the Operation.

AssemblyLine Phases
An AssemblyLine goes through three phases:

Initialization
At this point the TDI Server uses the "blueprint" for the AssemblyLine in the Config to create the various components as well as set up the AL environment, including processing the TCB, starting the ALs script engine and invoking the AssemblyLine's Prolog Hooks. All components that are configured for Initialization At Startup are initialized at this point causing their Prolog Hooks to get run as well.

Cycling
Now the AL workflow drives each of its components in turn, starting each cycle by invoking the On Start of Cycle Hook. Then the currently active *Feeds* Connector reads in data, creates the Work Entry and passes it to the *Flow* section. The Work Entry is passed from component to component until the end of *Flow* is reached, at which time control is returned to the start of the AssemblyLine again[13]. Cycling continues until an unhandled error occurs or there is no more data available (for example, the Iterator reaches End-of-Data).

13. If the current cycle was fed by a Server mode Connector, then the reply is created by the Server mode Connector's Output Map and sent to the client.

Shutdown
When cycling stops then the AssemblyLine goes into Shutdown phase: Epilog Hooks are called and all initialized components are closed down (which flushes output buffers and executes their Epilog Hooks as well). Finally the AssemblyLine closes down its environment and its thread terminates.

AssemblyLine Pool
Actually a collection of AL *Flow* sections that can be configured to allow a Server mode Connector to service more clients. Available for ALs that use Server mode Connectors and set up in the AssemblyLine's Config tab.

Attribute
Part of the TDI Entry data model. Attributes are carried by Entry objects (Java "buckets", like the Work Entry) and they can hold zero or more *values*. These *values* are the actual data values read from, or written to connected systems, and are represented in TDI as Java objects.

Attribute Map (AttMap)
An Attribute Map is a list of rules (individual Attribute Mapping instructions) for creating or modifying Attributes in an Entry object typically based on the values of Attributes found in another Entry object. Components like Functions and Connectors have an Input Map for taking data read into local cache (the conn Entry) and use this to define Attributes in the Work Entry. These components also have an Output Map that takes Attributes carried by the AssemblyLine (in its Work Entry) and use this to set up the conn Entry that will be used by the component's output operation. Attribute Map components use the Work Entry as both the source and target of the mappings.

Attributes can be mapped in one of three ways: Simply (copying values between Attributes), Advanced (using a snippet of JavaScript) or with a TDI Expression.

Attribute Map component
A free-standing list of individual Attribute Mappings that take values from the Work Entry and use them to create and update other Attributes in the Work Entry. They can be tied to Connector and Functions to define their Input or Output Maps. Note that Input and Output Maps can be copied to the library as AttMap components for reuse.

Best Practices
Recommended methodology and techniques for working with TDI. These include the ABCs: Automation, Brevity and Clarity:

Automation
Use the automated features of TDI in preference to your own custom scripted logic whenever possible – for example, using Branches/Loops instead of extensive scripting in Hooks. Not only will this make your solution easier to read and maintain (and step through with the AL Debugger!), but your solution will benefit directly as built-in logic is strengthened and extended with each new release.

Brevity
Keep your AssemblyLines as short and simple as possible, as well as your script snippets. Break complex logic into simpler patterns that can be tested individually and reused in other solutions.

Clarity
Choose legibility over elegance. Write solutions for others to read and maintain.

Branches
A construct used to control the flow of logic in an AssemblyLine. TDI 6.1.1 provides three types of Branches:

- Simple Branches (IF, ELSE-IF and ELSE)
- Loops (Connector-based, Attribute-based or Conditional)
- Switches (for example, switching on the Work Entry delta operation code, or the Operation an AL is called with).

CBE Common Base Event. A term used in the Common Base Infrastructure. See "Common Base Event" in the chapter about the CBE Generator Function Component in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

CEI The IBM Common Event Infrastructure. See "The Common Event Infrastructure", in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Change Detection Connector (CDC)
A Connector that returns changes made in the connected system. Typically, a CDC can be configured to return only a subset of Entries: new, modified, deleted, unchanged or a combination of these. Some CDC's provide only the changed Attributes in the case of a modified Entry, while others return them all. Change Detection Connectors also tag the data with special *delta operation codes* to indicate what has changed, and how.[14].

CloudScape (Derby)
A free, Java-based Relational Database, akin to IBM DB2, that is bundled with TDI as the default repository for the System Store.

CLI Command-line Interface, such as the `tdisrvctl` utility.

Components
The architecture of IBM Tivoli Directory Integrator is divided into two parts: generic functionality and technology-specific features. Generic functionality is provided by the TDI *kernel* which provides automated behaviors to simplify building integration solutions. The kernel also lets you extend or override these behaviors as desired, as well as doing the housekeeping for your solution: logging/tracing, Hooks for error handling, API and CLI access, and so forth Technology-specific "intelligence" is handled by helper objects called *components*, such as Connectors, Functions, Branches, Scripts and Attribute Map components. Components provide a consistent and predictable way to access heterogeneous systems and platforms, and the kernel lets you "click" together components to build AssemblyLines.

Compute Changes
A special feature of the Connector Update mode that instructs the Connector to compare the Attributes about to be written to the connected system with those that exist in this data source already – in other words, it compare the value of each Attribute in the conn Entry (the result of the Output Map) with the corresponding ones found during the Update mode *lookup* operation (which is stored in the current Entry).

14. For LDAP there is also a special kind of modify operation where the directory entry has beeen moved in the tree: *modrdn*, i.e. a "renamed" entry.

Config or Config File
A collection of AssemblyLines and components that comprise a solution. A Config is stored in XML format, typically in a Config file and is written, tested and maintained using the Config Editor.

Config Browser
This is the tree-view window at the top left-hand part of the Config Editor screen. It gives you access to Config-wide settings, the AssemblyLines and components that make up the Config, as well as Properties, *included* Configs and custom Java libraries that are to be loaded and made available to your scripts.

Config Editor (CE)
The graphical development environment used to write, test and maintain Configs. Configs are stored in XML format and are deployed by assigning them to one or more IBM Tivoli Directory Integrator Servers to execute.

Config Instance
A copy of a TDI Config that is running on a Server. Typically loaded only once on a given Server, TDI allows you to start the same Config multiple times if desired. Each running copy is given its own context and can be accessed individually through the API.

Config View
This term is used in the context of AMC to describe how a particular Config appears in the management screens of AMC. A Config View is a selection of the AssemblyLines and properties that are to be visible onscreen (user/role based), providing solution-oriented Config administration and management. Config Views can be combined to define a Monitoring View in AMC.

conn Entry
This is the local Entry object maintained by a Connector or Function. The conn Entry is used as a local cache for read and write operations, and data is moved between this cache and the AssemblyLine's Work Entry via Attribute Maps (specifically, Input and Output Maps).

Connector
One of the component types available in TDI to build AssemblyLines. Connectors are used to abstract away the technical details of a specific data store, API, protocol or transport, providing a common methodology for accessing diverse technologies and platforms.

Unlike the other components, Connectors can perform different tasks based on their *mode* setting (for example, Iterate, Delete, Server and Lookup). Modes are provided by the AssemblyLine component part of the Connector. However, the list of modes supported is dependent on the Connector Interface.

Connector Interface
When a component is used in an AssemblyLine, a distinction must be made between the *Connector Interface* (CI), containing the "intelligence" for working with a connected system (for example, LDAP, JDBC, Notes, and so forth), and the *AssemblyLine Connector*. [15]. This latter object is the "AL wrapper" that allows the CI to be plugged into an AssemblyLine and provides them with a consistent set of generic features, like input or output maps, Link Criteria, Hooks and the Delta Engine. See ″Objects″ in *IBM*

15. Functions are similar to Connectors in that they are divided into two parts: the Function Interface and the AssemblyLine Function. Unlike Connectors, Functions have no mode setting.

Tivoli Directory Integrator 6.1.1: Reference Guide for more information. See also "Connectors" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Connector Pool
Unlike the AssemblyLine Pool feature available to ALs using Server mode Connectors, a Connector Pool is a global collection of pre-initialized Connectors that can be used in multiple ALs. Note that the Connector Initialization setting "Initialize and terminate every time it is used" means that no AssemblyLine gains exclusive rights to a pooled Connector, giving you detailed control over resources used by your solution.

current Entry
This Entry object is local to a Connector Interface (just like the conn Entry) and contains the Attributes read in from a *lookup* operation (for example, as carried out by Lookup, Update and Delete modes). It is used to provide the Compute Changes feature.

Delta Engine
Available for Connectors in Iterator mode, the Delta Engine provides functionality for detecting changes in data sources that do not offer any changelog or change notification features. See Delta Operation Codes, as well as "Deltas and compute changes" in *IBM Tivoli Directory Integrator 6.1.1: Users Guide* for more information.

Delta mode (for Connectors)
This Connector mode is used to the apply changes specified with delta operation codes in the Work Entry, and to do so as efficiently as possible by performing incremental modifications. Note that Delta mode is only available for the LDAP and JDBC Connectors, and will not work with Entries without a valid delta operation code. See "Deltas and compute changes" in *IBM Tivoli Directory Integrator 6.1.1: Users Guide*.

Delta Operation Codes
These are special values assigned to Entries, Attributes and their values to reflect change information detected in some data source. An Entry that has delta codes assigned is called a *Delta Entry*, and these are only returned by a limited set of components: Change Detection Connectors, the Delta Engine and the DSML and LDIF Parsers[16]. Delta Operation Codes can be queried and used in Branch Conditions or your own JavaScript code, and are used by Delta mode to apply all types of changes to target systems as efficiently as possible.

See also "Deltas and compute changes" in *IBM Tivoli Directory Integrator 6.1.1: Users Guide*.

Derby CloudScape v10, also known as Apache Derby is a small footprint relational database implemented entirely in Java. Cloudscape is shipped as the default system store for TDI.

Distinguished Name (DN)
An LDAP term that refers to the fully qualified name of an object in the directory, representing the *path* from the root to this node in the directory information tree (DIT). It is usually written in a format known as the User Friendly Name (UFN). The dn is a sequence of *relative distinguished names* (RDNs) separated by a single comma (,).

Entry An Entry is a TDI object used to carry data, and forms the core of the TDI Entry model. The Entry object can be thought of as a "Java bucket" that can

16. Note that these Parsers only return Delta Entries if the DSML or LDIF entries read contain change information.

hold any number of Attributes, which in turn carry the actual data values read from, or written to connected systems. Each Entry corresponds to a single row in a database table/view, a record from a file or an entry in a directory (or similar unit of data), and there are a number of named Entry objects available in the system. The Work Entry and conn Entry are the most commonly used ones, but there is also a current Entry available in some Connector modes, an error Entry that contains the details of the last exception that occurred, and an Operation Entry (Op-Entry) for accessing details of an AL operation.

Epilog A set of Hooks that, if enabled, are run during the AssemblyLine Shutdown phase. Note that the shutdown of components occurs between the two AL Epilog Hooks, which means that the Epilog Hooks of these components are all completed before the AssemblyLine Epilog - After Close Hook is called.

Error Entry
An Entry object that is created by an AssemblyLine during initialization, and contains Attributes like "status", "connectorname" (applies for all types of components) and "exception"[17]. See also Error Handling.

Error Handling
Error Handling in TDI is based on the concept of *exceptions*. Exceptions are a feature of a programming language, like Java, C and C++, that lets you build error handling like a wall around your program. It also lets you fortify smaller parts within any wall, so you can add specific handling where necessary. TDI leverages the power of exception handling so that you can design the error handling in your solution the same way.

First you have the AssemblyLine's On Failure Hook which is called if the AL stops due to an unhandled exception[18]. This is the outer line of defense[19]. The next level is a component, given that it provides Error Hooks. Connectors actually provide two levels of handling: the mode-specific Error Hook, as well as the Default On Error (same goes for Success Hooks as well).

Finally, in your JavaScript code you can do exception handling yourself. Use the `try-catch` statement, for example:

```
try {
      myObj = someFunctionCallThatCanThrowAnException();
} catch ( excptThrown ) {
      task.logmsg("**ERROR - The call failed: " + excptThrown );
}
```

ERP Enterprise Resource Planning, usually indicates a software suite of programs that aims to manage enterprise resources, usually after heavy customization by the software vendor.

EventHandler
EventHandlers are components that reside outside of AssemblyLines, but that were used in older versions of TDI to "listen" for a specific event, and then dispatch this event data to one or more ALs. Each event (like a

17. The "exception" Attribute holds the actual Java exception object, in the case of an error – in which case the "status" Attribute would also be changed from a value of "ok" to "error" and "message" would contain the error text.

18. An "unhandled" error is one that is *caught* in an enabled Error Hook (no actual script code is necessary). If you wish to escalate an error to the next level of error handling logic, you need to re-`throw` the exception:

    ```
    throw error.getObject("exception");
    ```

19. If you want to share this logic (or that in any Hook) between AssemblyLines, implement it as a function stored inScript and then include them as a Global Prolog for the AL.

received DSML message, or a new changelog entry) resulted in a new AssemblyLine being launched, including the setting up and breaking down of all connections—which was quite resource-intensive. The functionality provided by EventHandlers is now handled using Connectors in Iterator or Server mode.

Note: EventHandlers are deprecated as of the TDI 6.1.1 release, although they are still supported for pre-6.1 Configs. Use Connectors in Server and Iterator Modes instead of EventHandlers. From TDI 6.1 on, if you create a new Config, no folder called "Event Handlers" will be visible in the Config Browser window of the Config Editor.

For Configs created prior to TDI 6.1, if you open such a Config, the Config Browser will contain the “EventHandler” library folder, and the TDI Server still supports their operation.

Exception
See Error Handling.

External Properties
A type of Property Store that uses a flat file for storing configuration settings (like passwords and other component parameter settings) outside the Config itself.

Feeds This is the first section of an AssemblyLine and can only hold Iterator and Server mode Connectors. The Feeds section is where the Work Entry is created from data retrieved from a connected system or client. The Feeds section is like a built-in Loop that drives the Flow section components list, once for each Entry read.

Flow This is the second (and usually the main) section of an AssemblyLine and holds a list of components; any type, except Connectors in Server mode. The Flow section receives a Work Entry from the currently active Feeds Connector and passes it from component to component for processing.

Function component (FC)
One of the component types available in TDI to build AssemblyLines. Functions are used to abstract away the technical details of a specific service or method call. Typical examples are the AssemblyLine FC used to execute ALs and the Java Class FC that lets you browse jar files and call class methods. Unlike Connectors, FCs do not have mode settings.

Global Prolog
This is a Script component that is defined in the "Scripts" library folder of the Config Browser, and which is configured to be executed when an AssemblyLine starts up. The simplest way to do this is to select which Scripts to use with the "Include Addition Prologs - Select" button. Note that Global Prologs are executed before the AssemblyLine's own Prolog Hooks.

GUI (`ibmditk` or `ibmditk.bat`)
The term "TDI GUI" is sometimes used to refer to the Config Editor.

Hook This is a *waypoint* in the built-in workflow of the AssemblyLine, or of a Connector or Function, where you can customize behavior by writing JavaScript. In a Connector, the Hooks available are also dependent on the mode setting.

HTML
Hyper**T**ext **M**arkup **L**anguage. a more or less standardized way of

describing and formatting a page of text on the WordWide Web. Different manufacturers' interpretations of the standard are often the cause of Web Browser's different renderings of a given page.

HTTP **H**yper**T**ext **T**ransfer **P**rotocol. The protocol in use for the WorldWide Web, another protocol on top of TCP.

IEHS IBM Eclipse Help System. Used to host the TDI documentation locally. The documentation hosted by IBM in the Documentation Library also uses IEHS.

Initial Work Entry (IWE)
This is an Entry that is passed into an AssemblyLine by the process that called it (for example, an AssemblyLine Connector or Function, or by using script calls like `main.startAL()`. Note that the presence of an IWE will cause any Iterators in the Flow section to skip on this cycle.

Iterator
A Connector mode[20] that first creates a data result set (for example, by issuing a SQL SELECT statement, a LDAP search operation, opening a file for input, and so forth) and then returns one Entry at a time to the AL for processing. Iterators can reside in the AssemblyLine Feeds section where they drive data to Flow components. If they are placed in the Flow section then they still retrieve the next Entry from their result set for each AL cycle, but they do not *drive* AL cycling in this case.

IU Installation Unit. A term specific to Solution Install (SI). Each major component of the product is broken into separate IUs - for ease of maintenance, installation and updates.

Java Virtual Machine or JVM
The JVM IBM Tivoli Directory Integrator runs inside what is known as a Java Virtual Machine. It has its own memory management and is in most respects a Machine within the Machine.

Javadocs
A set of low-level API documentation, embedded in the product's source code and extracted by means of a special process during the product's build. In IBM Tivoli Directory Integrator the Javadocs can be viewed by selecting **Help>Low Level API** from the Config Editor.

JavaScript
The language you can use to fine tune the behavior of your AssemblyLines. TDI 6.1.1 uses the IBM JSEngine.

JMS **J**ava **M**essaging **S**ervice. A standard protocol used to perform guaranteed delivery of messages between two systems.

JNDI **J**ava **N**aming and **D**irectory **I**nterface. See "JNDI Connector", in the *IBM Tivoli Directory Integrator 6.1.1: Reference Guide*.

Link Criteria
Link Criteria represent the matching rules defined for a Connector in Update, Lookup or Delete, and they must result in a single entry match in the connected system; otherwise either an Not Found or Multiple Found exception occurs. Note that a Lookup Connector tied to a Loop is an efficient way of dealing with lookup operations where no match (or multiple matches) are expected.

20. Connectors running in Iterator mode are often referred to as "Iterators".

LDAP **L**ightweight **D**irectory **A**ccess **P**rotocol. An easier way of accessing (using TCP) a name services directory than the older Directory Access Protocol. Used in for example querying the IBM Directory Server.

Memory Queue (MemQ)
The MemQ is a TDI object that lets you pass any type of Java object (like Entries) between AssemblyLines running on the same Server. This feature is usually accessed through the MemQueue Connector (or the deprecated Memory Queue FC). See also System Queue for more on how to pass data between running ALs.

Message Prefix
All error messages and Info messages in IBM Tivoli Directory Integrator are prefixed with a unique Message Prefix. The prefix assigned to TDI is **CTGDI**.

Mode Connectors have a mode setting that determines how this component will participate in AssemblyLine processing. In addition to the custom modes (implemented through Adapters) there is a set of standard modes:

- Iterator
- AddOnly
- Lookup
- Update
- Delete
- CallReply
- Server
- Delta

Dependent on the features provided by the underlying system or functionality built into the Connector, the list of modes supported by the different Connectors will vary. See "Connectors" in *IBM Tivoli Directory Integrator 6.1.1: Reference Guide* for more information about Connector modes.

Null Value Behavior
This term refers to how TDI will deal with Attribute Mappings that result in "null" values. Null Behavior configuration can be done for a Server by setting Global/Solution properties. These Server-level settings can be overridden for an Attribute Map by pressing the **Null** button in the button bar at the top of the map; or for a specific Attribute via the **Null** button in the Details Window for its mapping.

TDI lets you both configure what constitutes a "null" value situation (for example, missing values, empty string or a specific value) as well as how to handle this.

Op-Entry (Operation Entry)
An entry which contains information about the Operation for the currently executing AL. An Op-Entry persists its value over successive cycles for the same AL run and is available for scripting via the `task.getOpEntry()` method.

Parameter Substitution
A way of specifying patterns based on Java MessageFormat class - for simpler/quicker editing. Available in various places in TDI.

Parser TDI components used to interpret or generate the structure for a byte stream. Parsers are used by attaching them to a Connector that

reads/writes byte streams, or to a Function component like the Parser FC which is used to parse data in the Work Entry.

Persistent Object Store
See System Store.

Persistent Parameter Store
See Property Store.

Prolog A set of Hooks that, if enabled, are run during the AssemblyLine Initialization phase. You can also define Global Prologs: Scripts that are run before either of the AL Prolog Hooks. Note that the "At Startup" initialization of components occurs between the two AL Prolog Hooks, which means that the Prolog Hooks of these components are all completed before the AssemblyLine Prolog - After Initialization Hook is called. See also Epilog.

Properties
This term refers to values maintained in a Property Store and used to configure AssemblyLine and Component settings at run-time[21].

Property Store
This is a feature for reading and writing all types of properties. This includes:

- Java-Properties, which are settings of the JVM.
- Global-Properties, TDI Server settings that are kept in a file called `global.properties` the "etc" folder of your installation directory.
- Solution-Properties, which typically override Global-Properties and are found in a file in your solution directory called `solution.properties`.
- System-Properties, for keeping custom property settings (uses the System Store).

In addition, you can define your own Property Stores using a Connector. The Property Store feature also lets you designate one of your Property-Stores as a *Password Store*, giving you automatic protection of sensitive configuration details.

Raw Connector
Deprecated term; this is now called the Connector Interface and refers to the part of an AL Connector that contains the logic needed to access a specific API, protocol or transport.

Relative Distinguished Name (RDN™)
In LDAP terms the name of an object that is unique relative to its siblings. RDNs have the form *attribute name=attribute value*.

```
cn=John Doe
```

Resource Library
A simple method for sharing AssemblyLines and components between Configs. In the Config Editor, the “Resources” navigator appears just below the Config Browser.

RMI **R**emote **M**ethod **I**nvocation; a way of making procedure or method calls on a remote system using a network communication channel. In TDI, used by the Remote API functionality.

21. Note that an Entry object can also hold *properties* (in addition to Attributes and delta operation codes) and these can be accessed via the `getProperty()` and `setProperty()` methods of the Entry class.

Sandbox
The feature of the IBM Tivoli Directory Integrator that enables you to record AssemblyLine operations for later playback without any of the data sources being present. See "Sandbox" in *IBM Tivoli Directory Integrator 6.1.1: Users Guide*.

SAP Used to stand for "Systeme, Anwendungen, Produkte" (Systems, Applications, Products) but today, the abbreviation just stands for itself. A large, German provider of an integrated suite of ERP applications. Mostly known for its R/3 distributed ERP software suite, but also known for its mainframe-based R/2 software.

Script component (SC)
A Script is a block of JavaScript that is stored as a single component in TDI. The scripts library appears in the <tdi_installation> folder.Config Browser[22], Scripts can be dropped anywhere in the Flow section of an AssemblyLine.

Script Engine
The component that interprets the Java scripts written inside a TDI Config. The IBM jsEngine is used by TDI 6.1.1, which replaces Rhino from the previous releases.

Schema
The word "Schema", unfortunately, can mean different although related things, depending on context. In a relational database context, a schema is the collection of tables and objects a user has defined and owns (including content); and each table in a schema is described by a Data Definition. In an LDAP context, the Schema is the actual layout of the LDAP database, with its attributes and objects.

In addition, Connectors and Functions can have Input and Output schemas that represent the data model discovered in a connected system. Furthermore, an AssemblyLine Operation can have an Input and Output schema as well.

In a product like TDI, which with equal ease can access both relational databases as well as LDAP databases, the word Schema can therefore mean different things, depending on where it is used.

Script Connector
A Script Connector is a Connector where you write the *Interface* functionality yourself: It is empty in the sense that, in contrast to an already-existing Connector, the Script Connector does not have the base methods `getNextEntry()`, `findEntry()` and so forth implemented. Not to be confused with the Script Component.

Server (`ibmdisrv` or `ibmdisrv.bat`)
This is the part of the TDI product that is used to deploy and execute Configs.

Server (mode)
This is a Connector mode used for providing a request/response service (like an HTTP server). This mode also provides an AssemblyLine Pool feature to enable support for more connections/traffic.

Solution Directory
The directory in which you store your Config files, CloudScape databases,

22. In order to be used as Global Prologs (which are executed at the very start of Assemblyline Initialization) the Script must be in the scripts library folder and selected for inclusion in the Config tab of an AssemblyLine.

properties files, keystores and so forth. The solution directory is selected when you install TDI, and the filepaths used in your solution can be relative to this folder. The solution directory can be explicitly specified when you start the Config Editor or Server using the **-s** commandline option. Note that the counterpart of `global.properties` is kept in this folder and called `solution.properties`—unless, of course, your solution directory is the same as your installation directory.

SI Solution Installer. A common IBM utility for installation of many IBM products. The TDI installer is one such product.

SSL Secure Socket Layer; a protocol used in Internet communications to encrypt data such that if someone where to eavesdrop on the packets going back and forth he would not be able to see what the packets contain. The protocol was invented by Netscape; and you can see if a Web page uses the SSL protocol to talk to the Web server if it has the 'https//' prefix instead of 'http'. SSL is by no means limited to Web pages; in fact, TDI uses it (if configured that way) to talk between different TDI Servers and AssemblyLines if network access is called for.

State Defines the *level of participation* for an AssemblyLine component. It can be in either *Enabled* State, which means it will participate in AL processing, or *Disabled* in which case the component is not used in any way.

Connectors and Functions can be set to a third State: *Passive*. Passive State causes the component to be initialized and closed during the Assemblyline Initialization and Shutdown phases, but never used during AL cycling. However, you can drive these components manually through script calls.

System Queue
A built-in queue infrastructure to facilitate the guaranteed delivery of messages between AssemblyLines, even running on different TDI Servers. By default, the System Queue uses the bundled MQe (WebSphere MQ Everyplace), but can be configured to leverage other JMS-compliant messaging systems. TDI provides a SystemQueue Connector to help you leverage this feature.

New with TDI 6.1, System Queue is a built-in queue infrastructure to facilitate communications between ALs, in much the same way that the MemQ feature (and components) do. The big difference is that the System Queue enables data transfer across multiple TDI Servers. By default, the System Queue uses the bundled MQe (WebSphere® MQ Everyplace®), but can be configured to leverage other JMS-compliant queuing systems.

For more information about the System Queue and how to enable it, see the "System Queue" chapter in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

System Store
Called the Persistent Object Store, or POS in older TDI versions, the System Store is a relational database used to store state information, like Delta Tables (used by the Delta Engine) or Iterator state for Change Detection Connectors. It also provides the User Property Store which is accessible through the `system.setPersistentObject()`, `system.getPersistentObject()` and `system.deletePersistentObject()` methods. In the current implementation, the IBM DB2 for Java product (also known as **CloudScape**) is used. See http://www-3.ibm.com/software/data/cloudscape for more details.

Task By convention, all threads (AssemblyLines, EventHandlers and so forth) are referred to as *tasks* and are accessible from script code via the pre-registered **task** variable.

Task Call Block
A Java structure used to pass parameters to and from AssemblyLines. Often referred to by its abbreviation: **TCB**.

TCP **T**ransmission **C**ontrol **P**rotocol, a level 4 (transmission integrity) protocol usually seen in combination with its layer 3 (routing) Internet Protocol as in TCP/IP. A stack of protocols designed to achieve a standardized way of communicating across a network, be it local (as in on the premises) or over long distances. Originally invented and specified by DARPA, the (US) Defense Advanced Research Projects Agency. Successor to ARPANET, which was a network of a (small) number of universities and the US Department of Desfense, the civil side of which was managed by the Stanford Research Institute (SRI). TCP is related to UDP.

TDI Unofficial monicker for this product, IBM **T**ivoli **D**irectory **I**ntegrator.

TMS XML
Tivoli Message Standard XML. A Tivoli standardized way of formatting messages. Each message is prefixed by a unique TMS code, which can be looked up in the Message Guide for explanation and user response. If the code ends in ″E″ - it indicates an Error, ″W″ indicates a warning and ″I″ indicates an Information message. All Tivoli messages issued by TDI start with this product's unique identifier, which is "CTGDI".

Tombstone
A record or trace showing that an AssemblyLine has terminated. Configured through the Tombstone Manager in the CE. The trace includes a timestamp and the AL exit status. The new Tombstone Manager creates a tombstone for each AssemblyLine as it terminates.

TWiki TWiki as a piece of software is a flexible and easy to use enterprise collaboration system. Its structure is similar to the WikiPedia, except that is not linked into that. It is rather meant as an independent community resource for a group of people with common interest. There is one for IBM Tivoli Directory Integrator as well, at http://www.tdi-users.org.

Note: The TWiki site is a volunteer effort, and is not an official Tivoli support forum. If you need immediate assistance contact your local Tivoli support organization.

Update
One of the standard Connector modes. Update mode causes the Connector to first perform a lookup for the entry you want to update[23], and if found it modifies this entry. If no match is found then a new entry is added instead. See also Computed Changes.

UDP **U**ser **D**atagram **P**rotocol. A protocol used on top of the Internet Protocol (IP) which, unlike TCP does **not** guarantee that the packet of data sent with it reaches the other end. Also see TCP.

URL **U**nified **R**esource **L**ocator. A way of defining where a resource is, be it a fileserver or a HTML page on the WordlWide Web.

---

23. Data is read into both the `conn` and `current` Entry objects. After the Output Map, the contents of `conn` are now the Attributes to be written. The original entry data is still available in `current`.

User Property Store
See Property Stores in the *IBM Tivoli Directory Integrator 6.1.1: Users Guide*.

Value (data values and types)
See Entries, and Attribute.

WikiPedia
A Web-based world-wide encyclopedia, where (registered) users can add articles or pictures, edit them, browse them, search for applicable content, and so forth For TDI there is one that similar in functionality but not linked into the WikiPedia, a "TWiki" at http://www.tdi-users.org. The TWiki is a groupware product.

Work Entry
An Entry object that is used by the AssemblyLine to carry data from component to component[24]. This object can be accessed in script code via the pre-defined variable `work`. The Work Entry is typically built by a Server or Iterator mode Connector in the Feeds section before being passed to the AL Flow section. You can also have an Initial Work Entry (IWE) passed in if the AL was called from another process; or you can create it in the Prolog by using `task.setWork()`:

```
init_work = system.newEntry(); // Create a new Entry object
init_work.setAttribute("uid", "cchateauvieux"); // populate it
task.setWork(init_work); // make it known as "work" to the Connectors
```

Note that an Iterator in the Feeds section will not return any data if the Work Entry is already defined at this point in the AL. So if an IWE is passed into an AssemblyLine, any Iterators in the Feeds section will simply pass control to the next component in line. It is also the reason why multiple Iterators in the Feeds section run sequentially, one starting up when the previous one reaches End-of-Data.

XML The **X**tensible **M**arkup **L**anguage. A general purpose markup language (See also HTML) for *creating* special-purpose markup languages, and also capable of describing many types of data IBM Tivoli Directory Integrator uses XML to store Config files.

24. Note that the "Work Entry" window shown in the Config Editor is actually a list of all Attributes that appear in Input Maps or in the Loop Attribute field of Loops in the AssemblyLine.

Appendix F. Notices

This information was developed for products and services offered in the U.S.A. IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department MU5A46
11301 Burnet Road
Austin, TX 78758
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Third-Party Statements

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2006 International Business Machines Corporation and others

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM	Tivoli	AIX®	Lotus
Notes	pSeries®	DB2	WebSphere
OS/390®	Domino	iNotes™	CloudScape

Java, JavaScript and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Microsoft, Windows and Windows NT are registered trademarks of Microsoft Corporation.

Intel® is a trademark of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the U.S., other countries, or both.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

Other company, product, and service names may be trademarks or service marks of others.

Printed in USA

SC32-2568-01