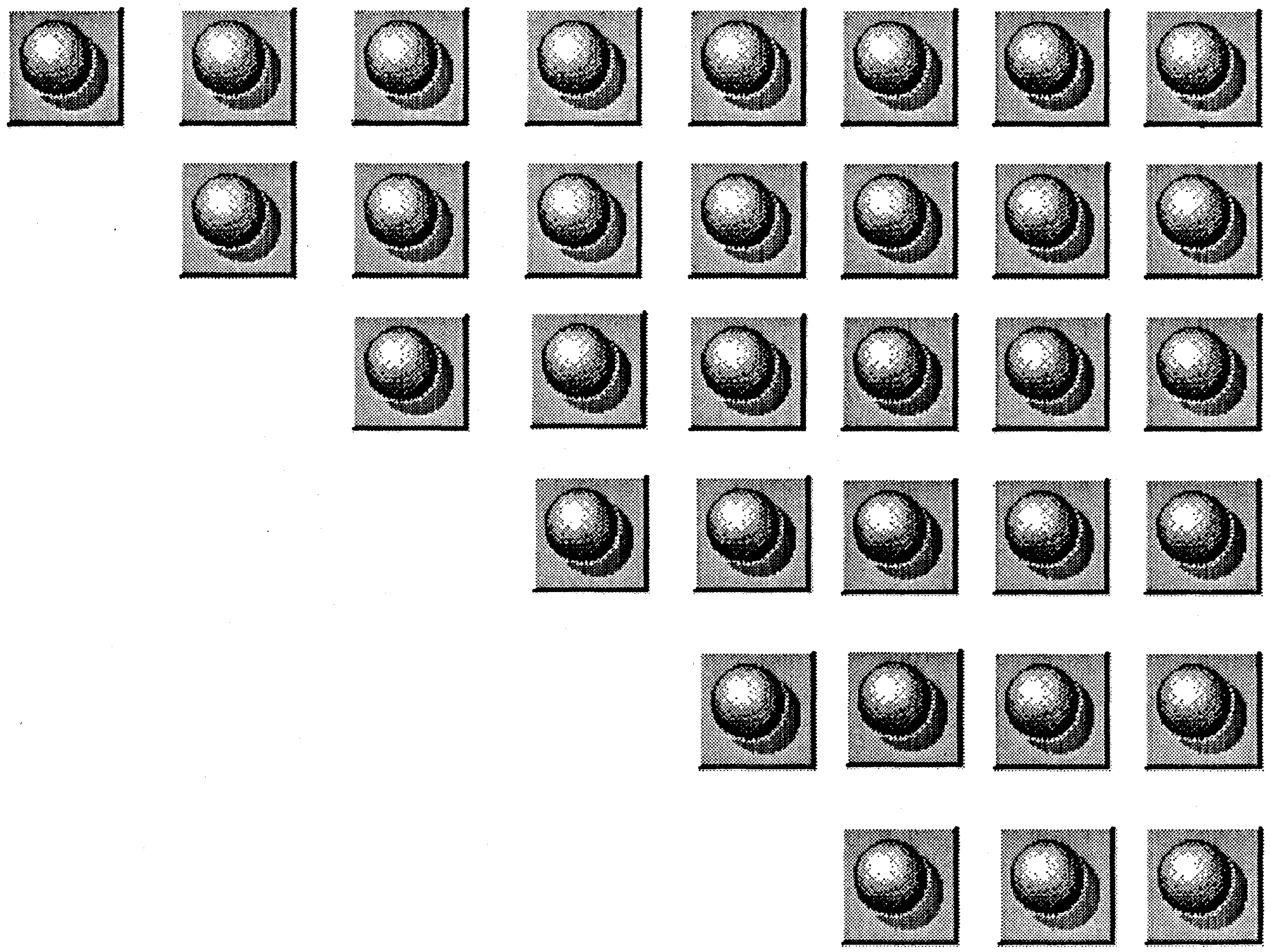
NeXT Developer's Seminar



Agenda

- Welcome
- Object-Oriented Programming
- Object-Oriented Design
- Hands-On Lab
- Lunch
- The Interface Builder
- The Application Kit
- Hands-On Lab
- Question & Answer Session

Goals

- Provide you with working knowledge and hands-on experience with the NeXT platform.
- Introduce you to the strengths of the NeXT development software environment.
- Provide an opportunity for the local NeXT developers to meet and share ideas.
- Encourage further education through Developer Training in California.

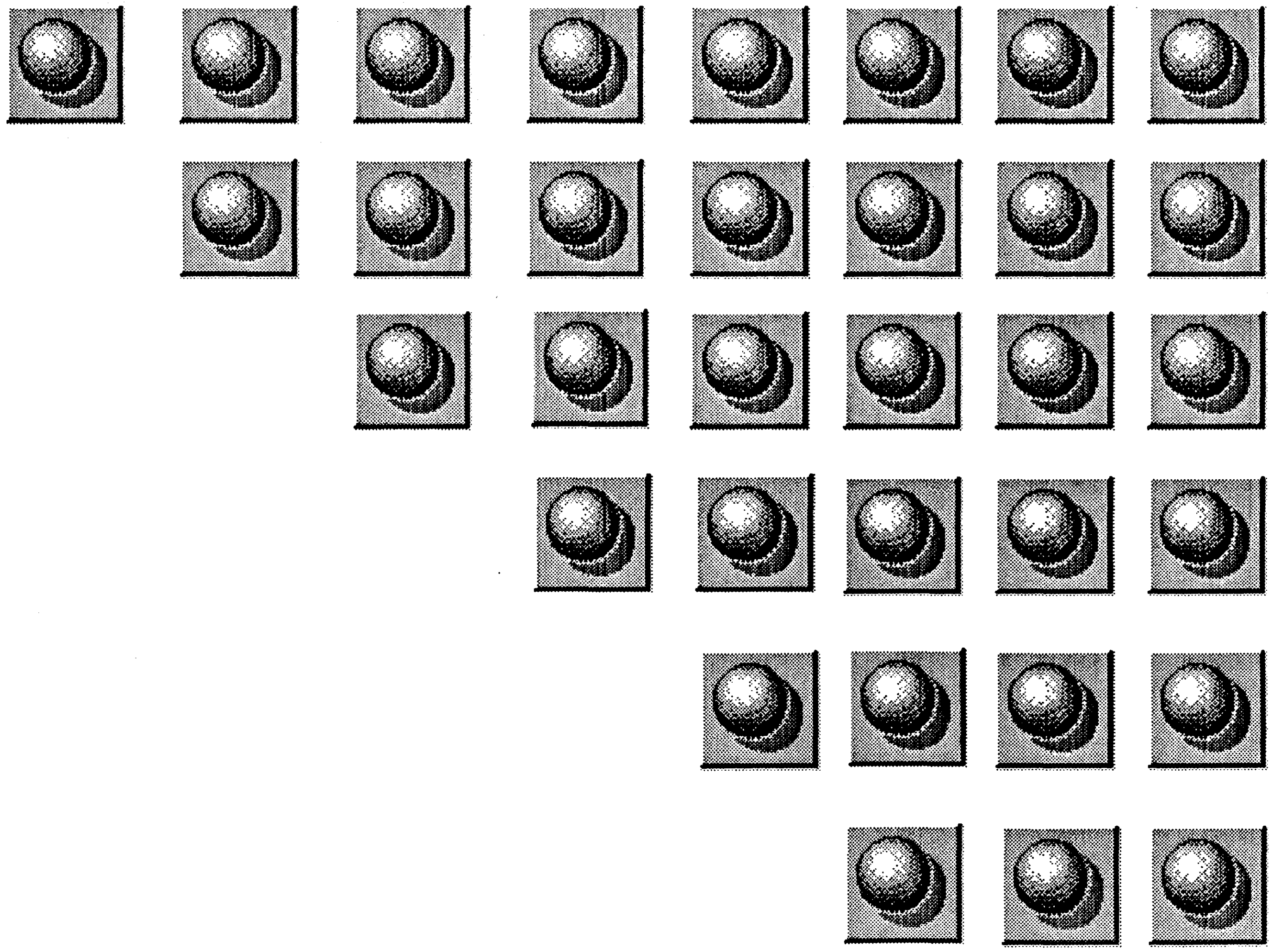
References

Object-Oriented Programming: An Evolutionary Approach, *Cox, Brad J., Addison-Wesley Publishing, 1987.*

NeXT Preliminary 1.0 System Reference Manual, *NeXT, Inc., 1990.*

NeXT Software Development Course Materials, *NeXT, Inc., 1989.*

Object-Oriented Programming



What is Object-Oriented Programming?

- **A code packaging technique**

  It packages functionality so that it can be reused.

- **Programs built around objects**

  Objects are a data structure and a group of procedures bundled together.

- **Three key features:**

  Encapsulation

  Inheritance

  Messaging

Why is it important?

- Reduced development time
- Reduced maintenance costs
- Package functionality so that it can be reused

Definitions

- **Object**

  *Private **data** and a set of **operations** that can access that data.*

- **Methods**

  *The '**procedures**' which perform the requested operations. Methods are private to an object.*

- **Instance Variables**

  *The **data** acted on by the methods. Instance variables are private to an object.*

Definitions

- **Encapsulation**

  *Encapsulation provides a shell (**object**) which contains the object's private data (**instance variables**) and a set of functions (**methods**) it can perform.*

- **Messages**

  *An object can ask another object to perform one of its methods (procedures) via a **message**.*

  Messages contain a reference to the object which is to be called, the name of the method to be executed, and any arguments, if required.

Definitions

- **Class Definition**

The prototype for a kind of object. It declares the instance variables and defines a set of methods that all objects in that class can use.

Classes are defined in two parts:

*- a file that declares the **interface** to the new class ("**.h**" extension).*

*- a file that actually defines the class and contains the code that **implements** it ("**.m**" extension)*

Class names begin with uppercase letters.

Definitions

- **Class (Factory) Object**

  Knows how to build new objects belonging to the class.

- **Instances of the class**

  *The individual occurrence of an object created by the Factory Object. These are the **objects** that do the work in your program.*

  Object names begin with lowercase letters.

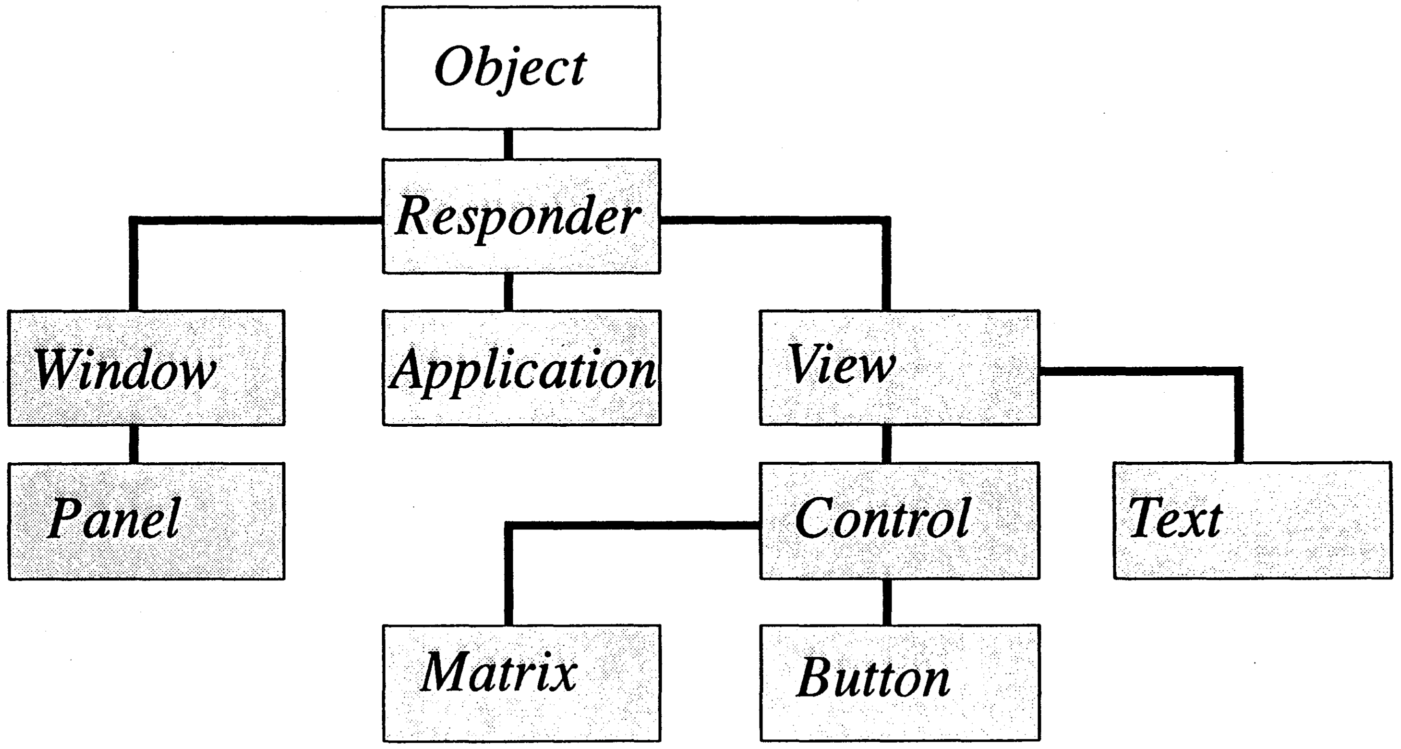
- **Selector**

  The name of the method in a message.

Definitions

- **Inheritance**

  *Allows an object to inherit features from a broader definition (**superclass**).*



Definitions

- **Inheritance (continued)**

Every class (except ***Object****) has a superclass one step nearer the root, and any class can be superclass for any number of subclasses one step farther from the root.*

Each class inherits both instance variables and methods from its superclass.

When a class object creates a new instance, the new object contains instance variables defined for its superclass, its superclass' superclass, ... to the root Object class.

Definitions

- **Inheritance (continued)**

  When a class object creates a new instance, the new object contains methods defined for its superclass, its superclass' superclass, ... to the root Object class.

  Class objects inherit only methods from their superclass because class objects do not have instance variables.

  A class may add additional instance variables and methods as well as ***over-ride*** *(re-define) methods defined in its superclass.*

Definitions

- **Dynamic Binding**

  A method and a receiving object are united when the program is running, not before. In traditional programming, the binding is done at compile time.

  Dynamic binding allows an object to send another object messages without knowing the class of the receiver until runtime.

  This also allow you to modify the user interface without recompiling the application.

Definitions

- Self & Super

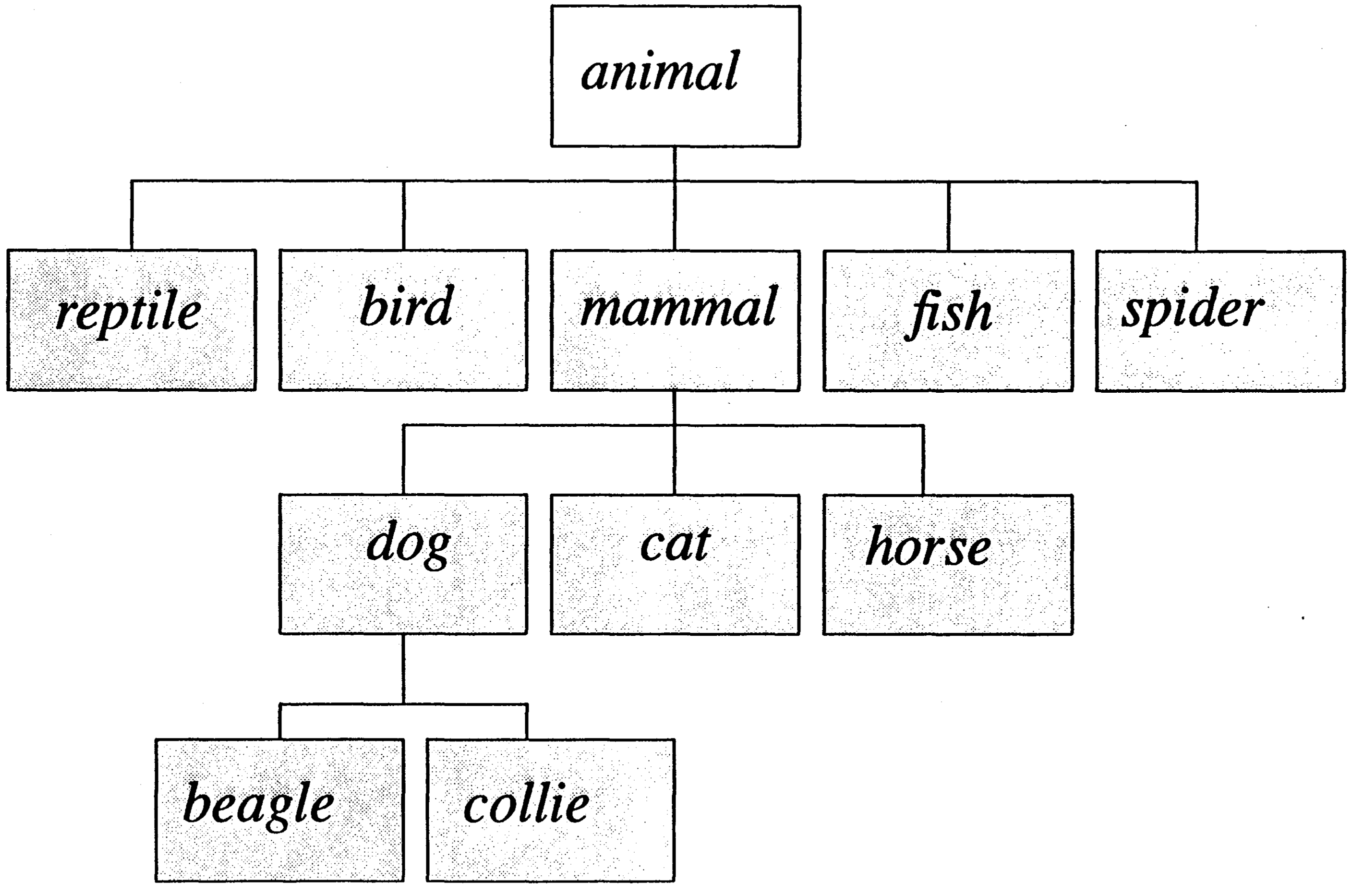
  `Self` *and* `Super` *refer to the* ***object*** *receiving a message.*

  `Self` *references the current instance.*

  `Super` *references the parent of the current instance. Used to invoke a method which has been over-ridden in a subclass.*

Example

- **A Class Hierarchy**



Example

- **Message Expression**

```
[receiver message];
```

sends **`message`** *to* **`receiver`**. *The name of the message is the* ***selector.***

```
[myMatrix display];
```

tells the **`myMatrix`** *object to perform its* **`display`** *method (draw the matrix and its cells in a window).*

```
[myMatrix moveTo:30.0 :50.0];
```

tells **`myMatrix`** *to change its location within the window to coordinates (30.0, 50.0). The* ***selector*** *is* **`MoveTo`**: *and the arguments are (30.0, 50.0).*

Example

- **Function Call vs. Message**

  An object only has access to the methods that it can perform. Each object sent a **`display`** *message could* **`display`** *itself in a unique way. This is called* ***polymorphism****.*

  A method has access to all the receiving object's instance variables; they don't need to be passed as arguments.

Example

- **Creating the instance of a class (an object)**

```
id myMatrix;
myMatrix = [Matrix new];
```

Tells the **`Matrix`** *class (**factory**) object to create a new* **`Matrix`** *instance and assign it to the* **`myMatrix`** *variable.*

Every class object has a method that allows it to produce new objects.

Objects are of type **`id`**.

Example

- **Creating the instance of a class (an object)**

```
id myClass;

myClass = [Matrix class];
```

The class name can stand for the class object ***only*** *as a message receiver. Otherwise, it must ask the class object to reveal its* **`id`** *by sending a class message).*

```
Matrix *anObject

anObject = [Matrix new];
```

The class name can also be used as a **`type`** *for instances of the class.*

Example

- **Inheritting Instance Variables**

```
View *myView;
myView = [Matrix new];
```

`myView` is statically typed to be a `View` and has been assigned a `Matrix` instance.

- **Inheriting Methods**

A new class defined in a program can use all of the methods defined for all the classes above it in the hierarchy.

This one of the major benefits of object-oriented programming.

Example

- **The Car Class**

Methods	*Values of Instance Variables*
`whatMake:`	*// Ford, Chevy, VW, Audi, Cadillac...*
`whatModel:`	*// Taurus, Jetta, Fleetwood, Sprint, ...*
`whatYear:`	*// 1900, 1901, 1902, ... , 1990*
`numberOfDoors:`	*// 2, 3, 4, 5, ...*
`sizeEngine:`	*// in cubic centimeters*

Example

- **The Car Class**

  Now let's send some messages to define **`myCar`**...

```
id myCar;
myCar = [Car new];
[myCar setWhatMake:VW];
[myCar setWhatModel:Jetta];
[myCar setNumberOfDoors:4];
[myCar setSizeEngine:1800];
[myCar setWhatYear:1985];
```

Example

- **The Car Class**

  ...and learn something about **`yourCar`**...

```
make = [yourCar whatMake];
model = [yourCar whatModel];
year = [yourCar whatYear];
doors = [yourCar numberOfDoors];
engine = [yourCar sizeEngine];
```

Summary

- **Definitions**

  encapsulation

  class

  inheritance

  message

  selector

  dynamic binding

  object

  method

  instance variables

Summary

- Syntax

  `id` -> *the* **`type`** *of an object.*

  `[receiver message]` -> *the way objects communicate with each other.*

  The receiver can be a variable or expression that evaluates to an object, a class name (indicating the class object), **`self`**, *or* **`super`** *(indicating an alternative search for the method implementation).*

Summary

- Syntax (continued)

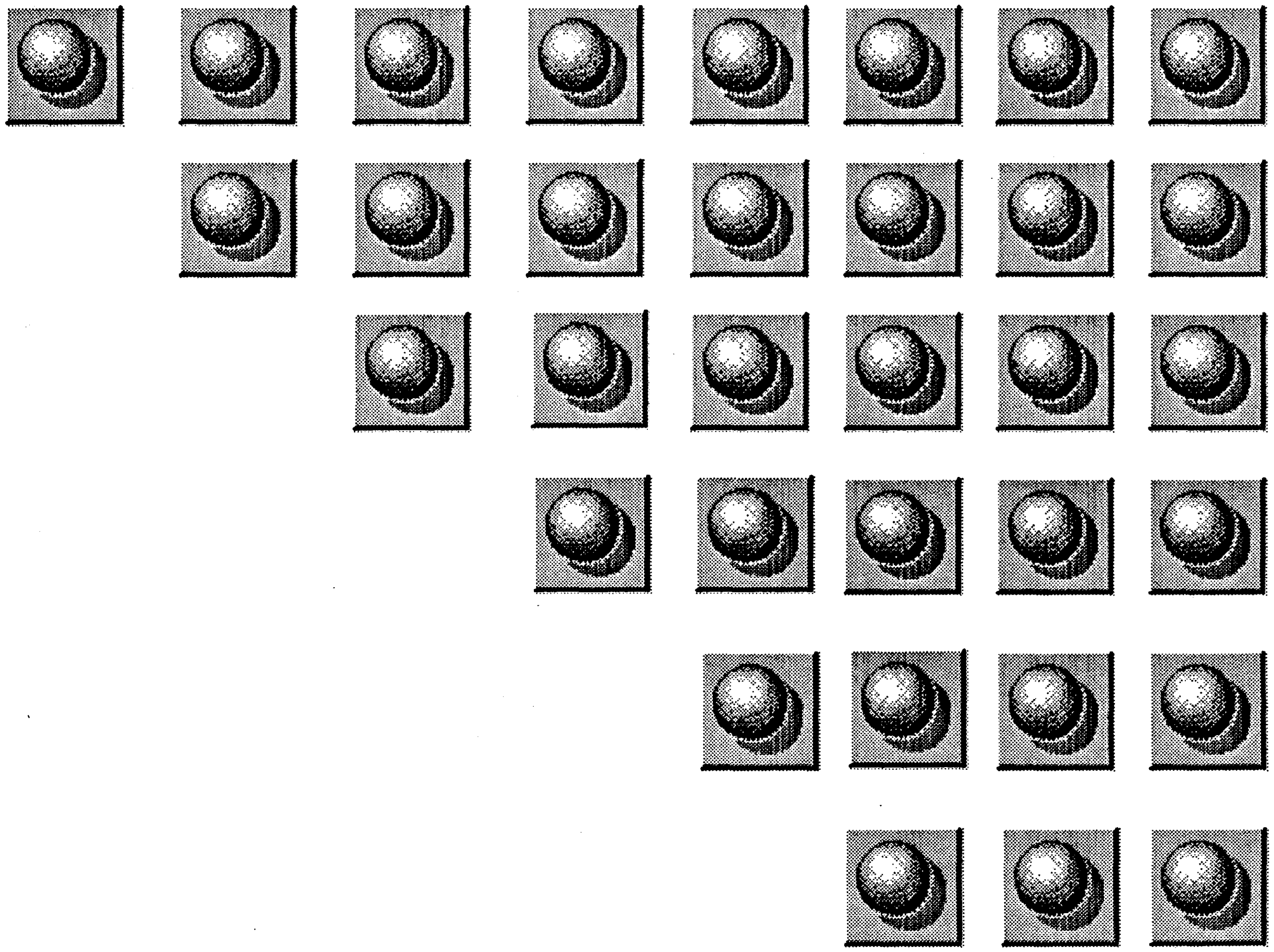
  `#import` -> *imports a header file. Used instead of* **`#include`** *because it won't include a header file more than once.*

  "**`+`**" -> *precedes declarations of class methods.*

  "**`-`**" -> *preceded declarations of instance methods.*

  "**`:`**" -> *arguments are declared after colons.*

Object-Oriented Design



Object-Oriented Design

- **An application is composed of a collection of interacting objects**
- **To build an object-oriented program, you need to answer 3 types of questions**

  What objects?

  What do the objects need to do?

  How do they interact?
- **The process**

  Structure the application as a collection of objects.

  Define each new class of object.

  Connect the objects together.

Object-Oriented Design

- **When should you define a new class?**

  When it is easier to think about the problem as a "thing".

  When there already is a class available to which you need to add instance variables or methods to meet your needs.

  Something which has useful generic behavior, but which you also are likely to want to customize for specific situations.

Object-Oriented Design

- **When shouldn't you define a new class?**

  When you have a problem which is hard to visualize or describe as a "thing".

  When you have a data structure which isn't typically modified.

  When a previously defined class will suit your needs simply by setting the values of instance variables.

Object-Oriented Design

- **Defining a new class**

  Find a class which already implements similar funtionality and subclass it by:

  adding new instance variables.

  over-riding existing methods.

  adding new methods.

  Add new instance variables when:

  an object needs additional variables in which to store its state.

  an object needs to store additional pointers to other objects so it can then send them messages.

Object-Oriented Design

- **Defining a new class (continued)**

  Over-ride an existing method when:

  you want your object to respond differently than its parent to the same message.

  Add new methods when:

  you want the object to respond to new messages.

  you want other objects to be able to access or set the new instance variables which may have been added.

Object-Oriented Design

- **Messages**

  An object sends a message when:

  it needs to access or set the instance variable of another object.

  it wants an object to perform some action.

  its state has changed and it wants to notify another object.

Object-Oriented Design

- **Messages (continued)**

  An object typically can send messages to:

  self.

  super.

  an instance variable which is also an object (otherwise know as an ***outlet****).*

  the object which sent it the message to which it is responding, assuming the `id` *of the sender was passed as part of the message.*

Object-Oriented Design

- **Messages (continued)**

  What message gets sent?

  a message is usually specified by the programmer at compile time.

  however, it can be a message specified by an instance variable which is defined to be a reference to a message.

Object-Oriented Design

- **Connections**

  Objects are typically connected via their instance variables:

  at runtime, the appropriate instance variable in one object is set to point to another object.

  these instance variables are subsequently used when the object needs to send the other a message.

  objects may be connected to any number of objects.

Object-Oriented Design

- **Connections (continued)**

  The messages sent between connected objects are:

  either predefined at compile time or,

  contained in other instance variables.

Object-Oriented Design

- **Connections (continued)**

  There are 3 types of connections:

  the target and the message are specified at compile time.

  the target is contained in an instance variable, but the messages it is sent are specified at compile time.

  both the target and the message are contained in instance variables which may be modified at any time at the request of another object.

Object-Oriented Design

- **At Runtime**

  Objects used in the program must be initialized:

  either by sending a message to the Factory object belonging to the object's class or,

  by using the "archiving" mechanism to load in a previously stored instance of the object.

- **An object's instance variables are set as needed to reflect the state of other characteristics of the objects.**

Object-Oriented Design

- **At Runtime (continued)**

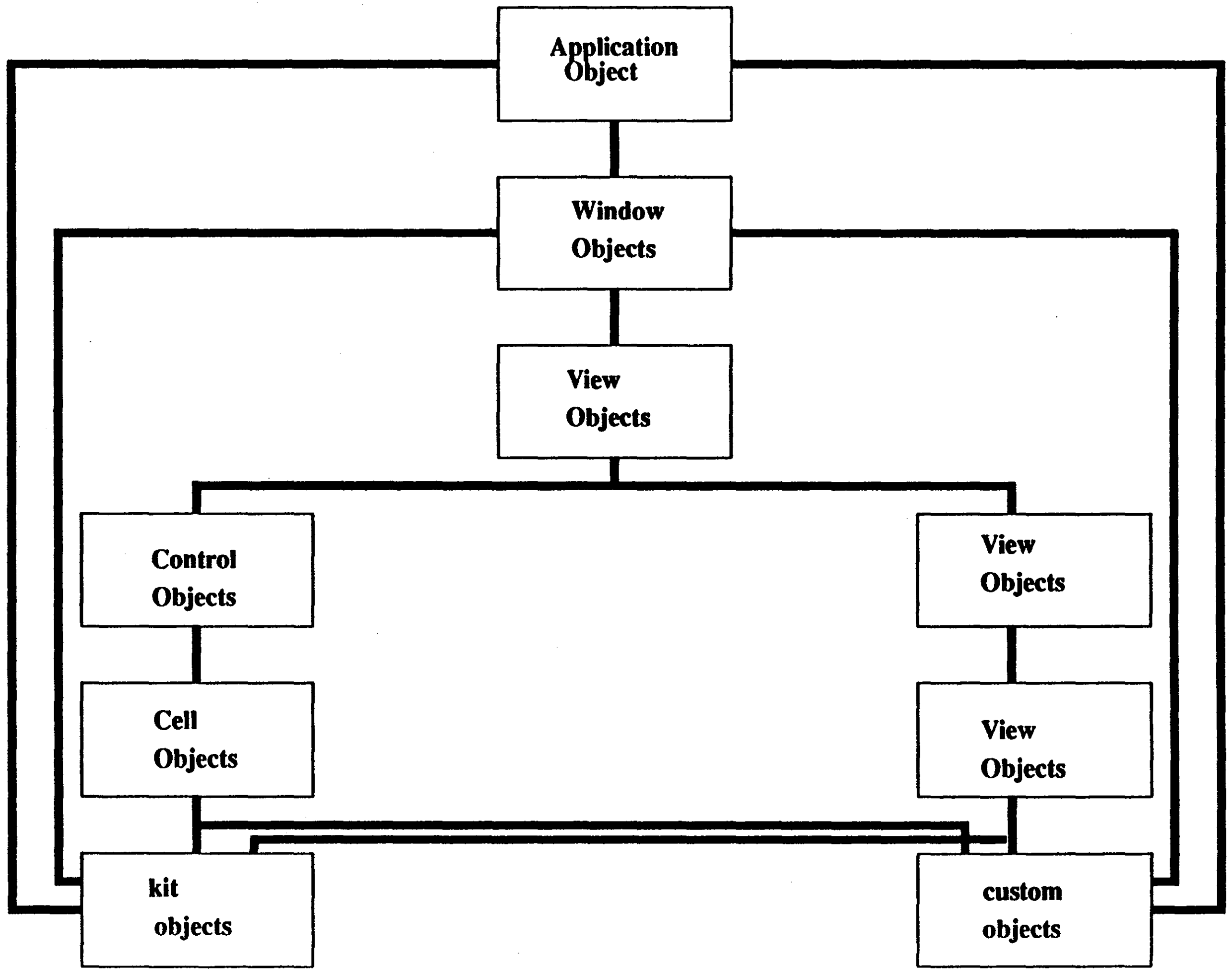
  Connections with other objects are established:

  typically by setting the appropriate instance variables.

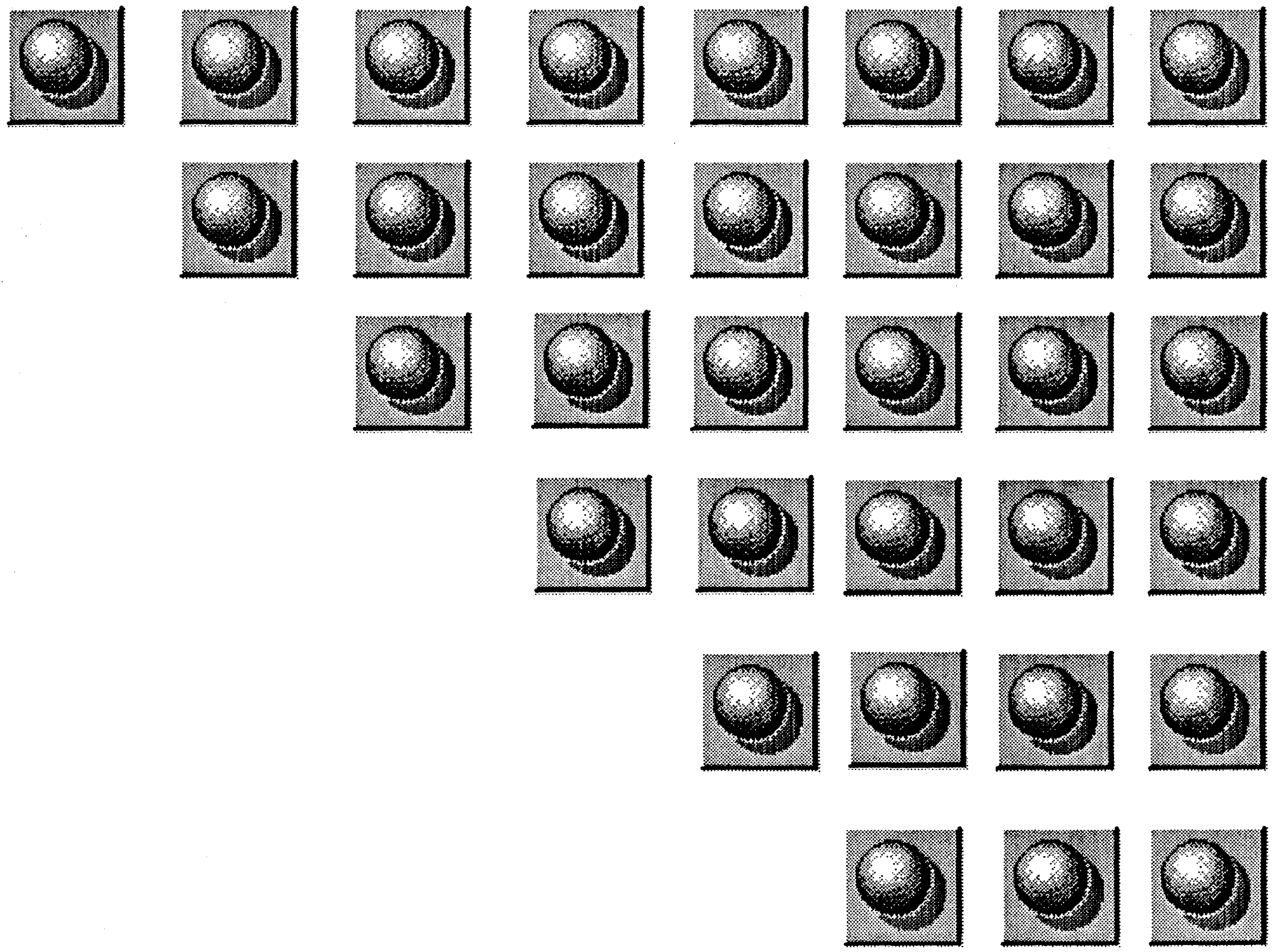
- **Additional messages are sent as the program proceeds.**

Object-Oriented Design

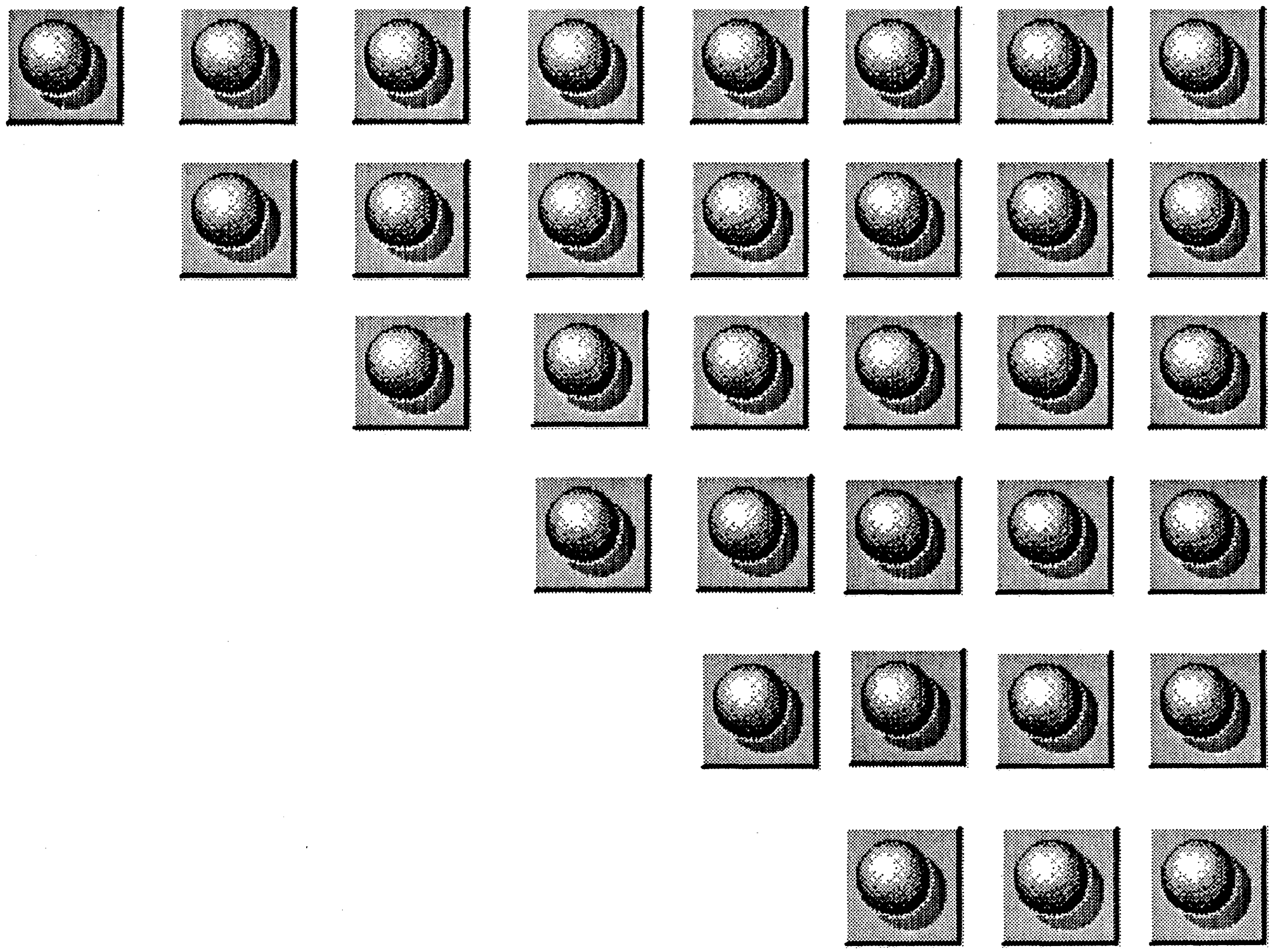
- Structure of Typical NeXTStep Application



Hands-On Lab



The Interface Builder



What is the Interface Builder?

- **A tool for creating user interfaces**

  Allows you to create an interface using on-screen graphics objects and test your interface.

- **Provides support for the code beneath those interfaces**

  The Interface Builder will actually generate stubs of code for the interface and the connections between objects.

Interface Builder

- **Application Kit**

  A library of user-interface objects that you can select from for your application.

  Allows you to graphically build the user-interface for your application.

  AppKit objects include such items as buttons, sliders, window, panels, switches, etc.

  In general, your application will include a number of AppKit objects and one or more subclasses of ***Object*** *(containing the logic of your application) and* ***View*** *(drawing code unique to your application).*

Interface Builder

- **Changing instance variable values**

  Once an object is added to your application, IB allows you to change the values of many instance variables directly.

  For example, changing the size of a button on the screen changes the values of one of the Button object's instance variables.

  Instance variables for objects not easily changed graphically can be changed using the ***Inspector Window.***

Interface Builder

- **Making Connections**

Interface Builder lets you interconnect objects so they can communicate with one another.

Connections are made through an objects ***outlets****.*

An ***outlet*** *is an instance variable of type* `id` *that allows an object to send messages to another object in the application.*

When your application begins execution, outlet variables are automatically initialized to the ***ids*** *of the objects you specified within Interface Builder.*

Interface Builder

- **The Interface File**

  *The interface you develop is saved in an interface builder file ("**.nib**" extension).*

  *The **.nib** file contains all of the class information and all specifications for the AppKit objects in your program.*

  Classes are created and objects are initialized by the AppKit using information in this file.

Interface Builder

- **The Interface File (continued)**

  The ***.nib*** *file also contains information about how outlets should be initialized, about action messages and their targets, sound and icon data, and a reference to an owner object.*

  An application can have more than one ***.nib*** *file, but only one can be the main* ***.nib*** *file.*

  When your application is compiled, information from the ***.nib*** *file is copied into the Mach-O format executable.*

Interface Builder

- **The Project**

  Each application is part of a project.

  The project directory contains all the files that are part of the application.

  *Contains a project file ("called **IB.proj**") that organizes all of the pieces of the application. (e.g., *.**h** files, *.**m** files, Makefile, ...)*

  *Contains an interface file ("**.nib**").*

  Updated every time you make a change to the application.

Interface Builder

- **The Project (continued)**

  Interface Builder uses the project file to create the files needed during compilation...

  Makefile *-> created by the Interface Builder.*

  Main file *-> contains the* **`main()`** *function.*

  *An icon header file ("**.iconheader**" extension) contains information about icons associated with the application and its files.*

Interface Builder

- **The Process**

  1. Create Interface

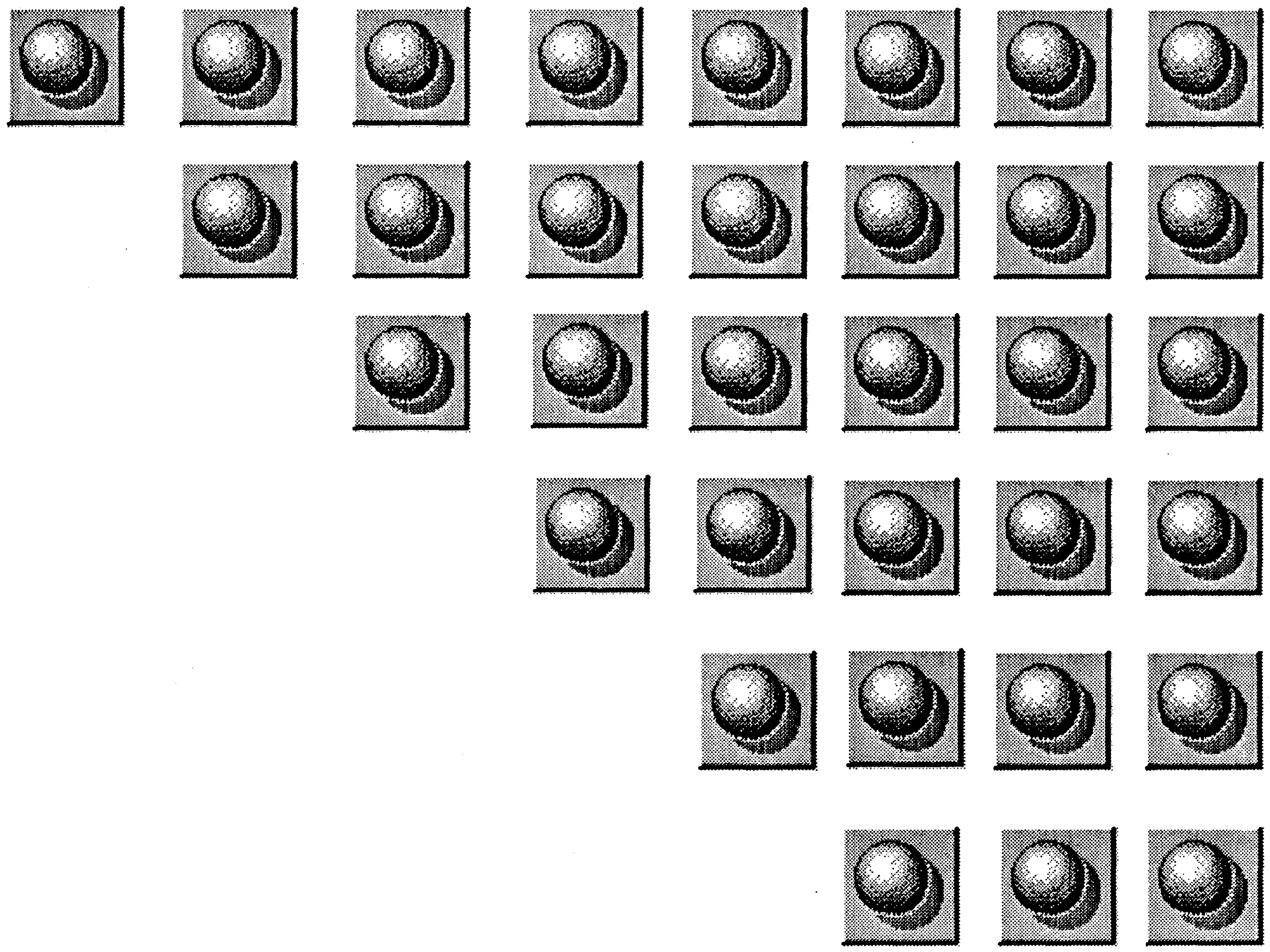
  2. Create Project

  3. Compile

  4. Run

  5. Revise if necessary

The Application Kit



Overview

- **The AppKit**

  *The NeXT interface is **event-driven**.*

  *The Application Kit provides the **main event loop** and automatically dispatches events to the appropriate object.*

  The Application Kit provides a rich set of objects for getting user input and a uniform way of interacting with those objects.

Overview

- **Events**

  Events are things like **`mouseUp`**, **`mouseDown`**, **`mouseDragged`**, *etc.*

  Events are dispatched as ***messages****.*

  Usually, an object responds to the event, then notifies another object that the event occurred.

  *Two types of messages get sent depending on the object that receives the initial message (****action*** *and* ***notification****).*

Overview

- **The Process**

  1. Window Server sends mouse, keyboard and machine events to the Application object.

  2. Within the application, the AppKit dispatches event messages to the appropriate object.

  3. The object responds.

The AppKit

- **The Process: a closer look**

  Window Server sends all events to the Application object.

  the Application object handles machine events directly. (power-off, etc.)

  if a window event, the Application object sends a window event message to the appropriate window. (close-window)

  otherwise, sends an event message to the appropriate window for dispatch.

The AppKit

- **The Process: a closer look**

The window dispatches mouse event messages to the appropriate view in the Window.

`mouseDown:` *to deepest view underneath mouse*

`mouseUp:` *or* **`mouseDragged:`** *to view which initially received* **`mouseDown:`**.

`mouseEntered:` *or* **`mouseExited:`** *to the object which "owns" the appropriate tracking rectangle.*

keyboard and **`mouseMoved:`** *event messages are dispatched to the window's "firstResponder".*

The AppKit

- **How objects respond**

  *Do nothing but pass event message on to its "**nextResponder**".*

  by default, ***nextResponder*** *is the object's superview.*

  this is the default behavior of views.

  Perform object-specific action.

  e.g., Text objects display characters corresponding to keystrokes.

The AppKit

- **How objects respond (continued)**

  Begin a modal loop

  e.g., a Button object highlights on **`mouseDown`** *and enters a modal loop waiting for* **`mouseUp`**.

  Respond to the event and send a message to another object notifying it of the event.

  two notification schemes are used here: ***Target-Action*** *and* ***Delegation-Notification***.

  1. Target-Action is used by Controls.

  2. Delegation-Notification is used by Window, Applications, and Text.

The AppKit

- **Target-Action**

  The receiving object gets a message and handles the event.

  e.g., Button Object highlights and waits for **`mouseUp`**.

  e.g., Form Object allows the user to edit a field until user hits return.

  When handling is complete, the object notifies a target object by invoking an action method owned by the target

  Action messages are always of the form:

```
[target messageName:sourceObjectId]
```

The AppKit

- **Target-Action (continued)**

  If necessary, the target interrogates source objects for additional information

  use **`state`** *to get state of button.*

  use **`stringValue`** *to get the text in last selected field of a form.*

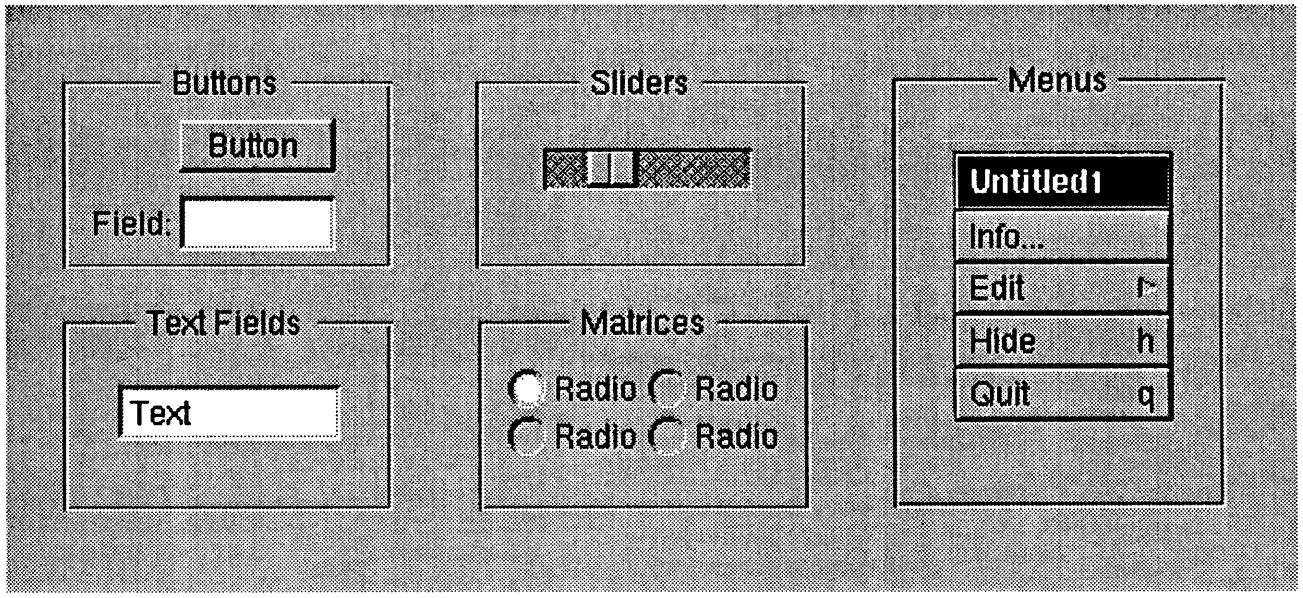
  use **`floatValue`** *to get the current value of a slider.*

  Receiving objects translate event messages into action messages.

The AppKit

- **Target-Action (continued)**

  These objects all send action messages to their specified targets...



The AppKit

- **Examples of Action Messages**

  Application

  ```
  hide:
  unhide:
  terminate:
  stop:
  ```

The AppKit

- Examples of Action Messages (continued)

  Button

  ```
  performClick:
  ```

  Control and Cell

  ```
  takeIntValueFrom:
  takeStringValueFrom:
  takeFloatValueFrom:
  takeDoubleValueFrom:
  ```

The AppKit

- **Examples of Action Messages (continued)**

  Text

  ```
  cut:
  copy:
  paste:
  clear:
  ```

  View

  ```
  printPSCode:
  ```

The AppKit

- **Examples of Action Messages (continued)**

  Window

```
miniaturize:
deminiaturize:
orderFront:
orderBack:
orderOut:
performClose:
performMiniaturize:
performResize:
printPSCode:
```

The AppKit

- **Examples of Action Messages (continued)**

  You can invoke any of these methods by sending a message to the appropriate object, passing the sender's `id` *as the single argument.*

```
[myInfoPanel orderFront:self];
```

  You will often use these messages in conjunction with Interface Builder.

The AppKit

- **Examples of Action Messages (continued)**

  When a user drags a slider and changes its value, a message **`changeSlider:`** *might be sent to the target object. The argument to the message will be the* **`id`** *of the slider. (i.e.,* **`self`***)*

  The receiving target object will understand the **`changeSlider:`** *message because it is defined in its class.*

```
-changeSlider:sender
{
 sliderValue = [sender floatValue];
 return self;
}
```

The AppKit

- **Examples of Action Messages (continued)**

    The target object uses the sender argument to get more information about the action (e.g., the value of the slider).

    Target-Action connections are set up so that when the user select the actions, the appropriate message will be sent to the target objects.

The AppKit

- **Delegation-Notification**

  An AppKit object may send messages to another object, called its ***delegate****, notifying it that the sender's state has either changed or is about to change.*

  A delegate can choose to

  ignore the notification.

  do additional processing in response to the notification.

  depending on the notification, block the change that resulted in the sender sending the notification.

The AppKit

- **Delegation-Notification (continued)**

  Delegation may be used to avoid subclassing standard kit objects such as Application, Text, and Window.

  By implementing custom behavior in an object's delegate, it is possible to utilize a standard kit object, yet provide custom behavior in response to events or changes in state.

  Application, Text & Window all send pre-defined notification messages for changes in state for which custom behavior is likely to be desired.

The AppKit

- **Delegation-Notification (continued)**

  Delegation-notification is similar to target-action...

  an object can have only 1 delegate at a time.

  a notification message is sent in response to some event or change.

  Delegation-notification is different from target-action...

  notification messages are pre-defined by the sender.

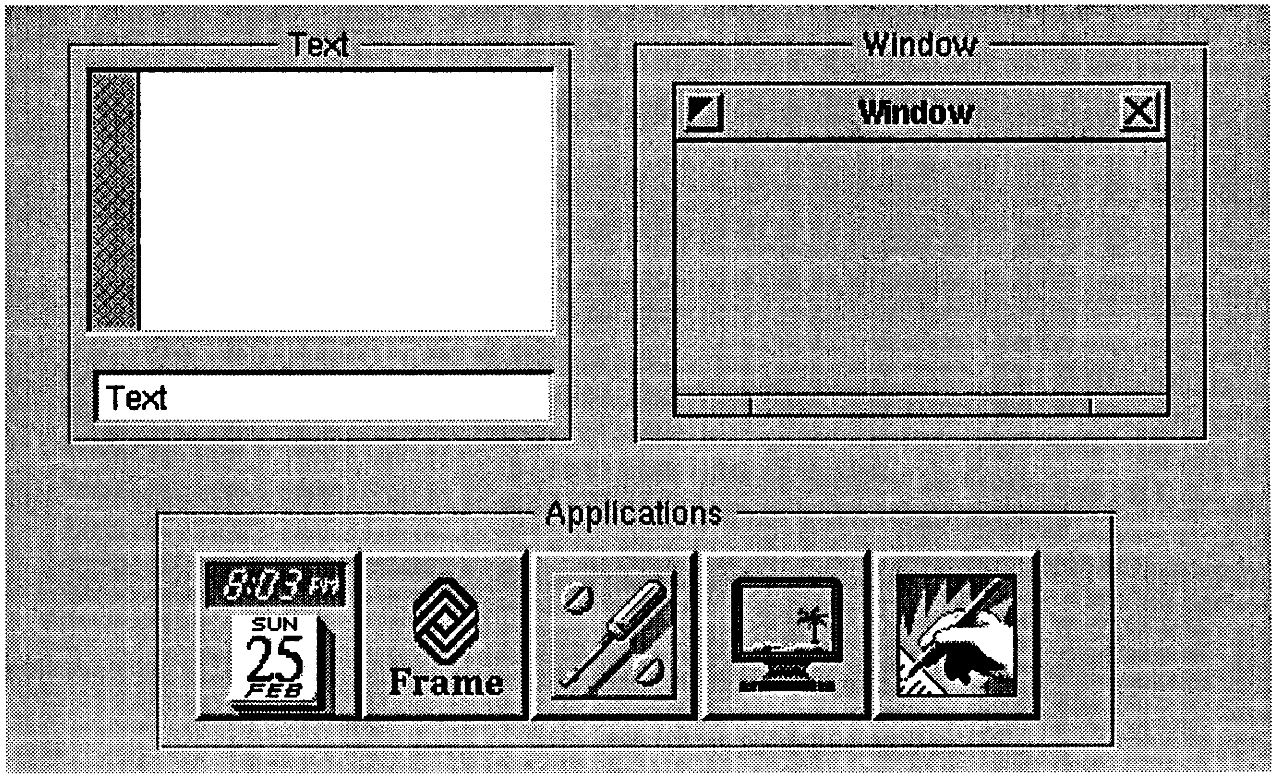
  an object may send a different notification message depending on what has occurred.

  if no delegate is set, no notification is sent.

The AppKit

- **Delegation-Notification (continued)**

  These objects have predefined notification messages for changes in state for which custom behavior may be desired...



The AppKit

- Examples of Notification Messages

  Application

  ```
  appDidInit:
  appDidAwake:
  appDidBecomeActive:
  appDidResignActive:
  appDidHide:
  appDidUnhide:
  ```

The AppKit

- **Examples of Notification Messages (continued)**

  Window

```
windowWillClose:
windowWillResize:toSize:
windowDidResize:
windowDidMove:
windowDidExpose:
windowDidBecomeKey:
windowDidResignKey:
windowDidBecomeMain:
windowDidMiniaturize:
```

The AppKit

- Examples of Notification Messages (continued)

Window (continued)

```
windowDidDeminiaturize:
```

Text

```
textWillChange:
textWillResize:
textWillEnd:
textDidResize:oldBounds:Invalid:
textDidChange:
textDidEnd:endChar:
text:isEmpty:
```

The AppKit

- **Examples of Notification Messages (continued)**

Notification messages are only sent if...

the delegate has been sent the following...

```
[myWindow setDelegate:myDelegateObject]
```

and the delegate has implemented a method of the same name.

A delegate only needs to implement methods for those notification messages to which it cares to respond.

The AppKit

- **Examples of Notification Messages (continued)**

Purpose: be notified when a user closes a window.

Redefine, or over-ride the **`windowDidClose:`** *method (without subclassing window).*

When the message is sent to the object, the delegate's method (redefinition) is invoked in place of the object's method.

Send a **`setDelegate:`** *message to the window with the delegate object's* **`id`**.

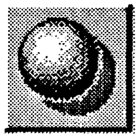
When the user closes the window, the delegate's method will be called.

The AppKit

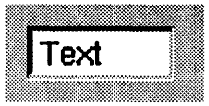
- **Outlets**

  An outlet is an interface object that the controlling object needs to access.

  A controlling object...

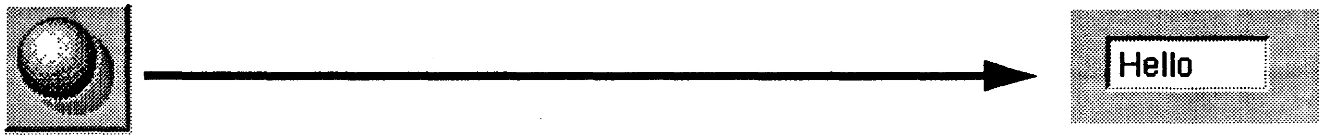


  An interface object...



  A message... `[myText setStringValue:"Hello"];`

  The result...



The AppKit

- **Outlets (continued)**

  To the controlling object, the outlet is simply an instance variable of type `id` *(object pointer).*

  Initially, all instance variables (including ones that are outlets) have invalid and uninitialized values.

  *All outlets are initialized when the "**.nib**" is "loaded", i.e., when the specification for the program's interface is loaded into memory. This is usually done in the "**_main.m**" file with a* `loadNibSection::` *message (defined in the Application class).*

The AppKit

- **Outlets (continued)**

  *All outlets are initialized when the "**.nib**" file is loaded.*

  This is done by sending initialization messages to the controlling object.

  > *if your interface has one Button and two TextFields, then one instance of the Button class and two instances of the TextField class are created (by sending* **`new`** *messages to the classes).*

The AppKit

- **Initializing Outlets**

  If outlet `myText` *is to be initialized, the Interface Builder automatically creates:*

  an instance variable `myText` *for the new class (type* `id`*).*

  an instance method called `setMyText:` *to initialize* `myText`.

```
-setMyText:anObject
{
 myText = anObject;
 return self;
}
```

The AppKit

- **Initializing Outlets (continued)**

  The first letter of the outlet name is converted to uppercase in the outlet initialization method.

  e.g., **`myText`** *becomes* **`setMyText`**

The AppKit

- **Connecting to Outlets**

  After an interface object has been created, a connection between the controlling object and the newly created objects must be established.

  *The outlet initialization method establishes this connection when the "**.nib**" is loaded.*

```
id tmp;
tmp = [TextField new];
[controlObj setMyText:tmp];
```

  The system will send the appropriate initialization messages. There is NO need to invoke these methods yourself.

The AppKit

- **Outlet Initialization (continued)**

  Initialization code specific to an instance variable that is an outlet should be placed in the outlet initialization method.

```
-setMyText:anObject{
 myText = anObject;
 /* your initialization code */
 [myText setFloatingPointFormat:No
 left:4 right:2];
 [myText setFloatValue:2.56];
 return self; }
```

The AppKit

- **Outlet Initialization (continued)**

  The instance variable **`myText`** *has no value (and should not be used) prior to the execution of the assignment statement (in the first line above).*

The AppKit

- **Using Outlets**

  After the initialization of an outlet, the controlling object can use it by referring to the instance variable.

```
[myText setFloatValue:2.45];
```

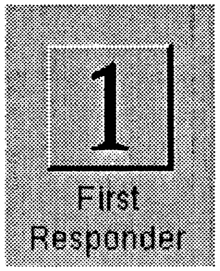
  By doing so, messages are actually being sent to the interface object placed on the application's window.

The AppKit

- **First Responder**

  An object that receives mouse and keyboard input (typically a ***text*** *object).*

  Each window has its own first responder.



  The AppKit provides methods for setting responder and managing first responder status.

  The first responder must have a Responder as ancestor class.

  *If first responder does take action, the events are passed up a "****responder chain****".*

The AppKit

- **First Responder (continued)**

  The object that is selected to be the focus of future events for a Window is the first responder. Each Window has its own first reponder, which it returns when asked:

```
id handler;

handler = [myWindow firstResponder:];
```

  The first responder is typically a View object in the Window's hierarchy, but it can be any Responder.

The AppKit

- **Speaker/Listener**

  AppKit object classes that provide a way to send messages between objects in different applications, possibly on different machines.

  Every application has default ***Listener*** *and* ***Speaker****, and their ports are automatically created for communication.*

  Speakers and Listeners agree in advance to the set of messages they can exchange.

  You specify the set of messages.

  `msgwrap` *will generate the subclasses of Listener and Speaker for you.*

The AppKit

- **Speaker/Listener: Messaging Webster**

  *Assume the **Define** button is connected to this method, and the **WordField** variable refers to a TextField.*

```
-define:sender
{
   int speakerResult, websterResult;
   port_t websterPort;
```

/ look up the public port for Webster's Listener on local host */*

```
websterPort = NXPortFromName ("Webster", NULL);
   if (websterPort == PORT_NULL) {
      fprintf(stderr, "Port was not found.\n");
      return self;
   }
```

The AppKit

- **Speaker/Listener: Messaging Webster**

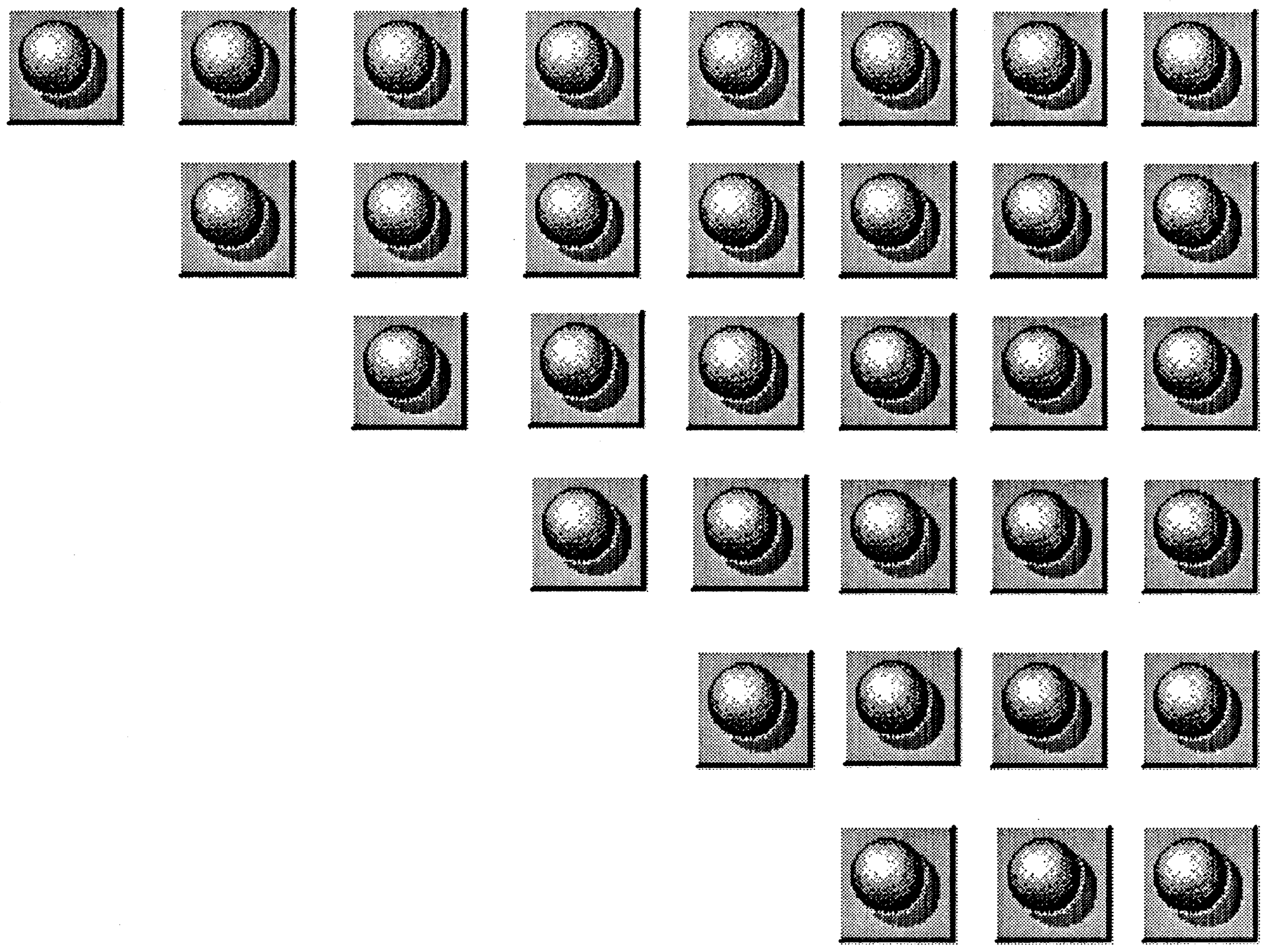
/ connect the port to the Application's Speaker */*

```
[[NXApp appSpeaker] setSendPort: websterPort];
```

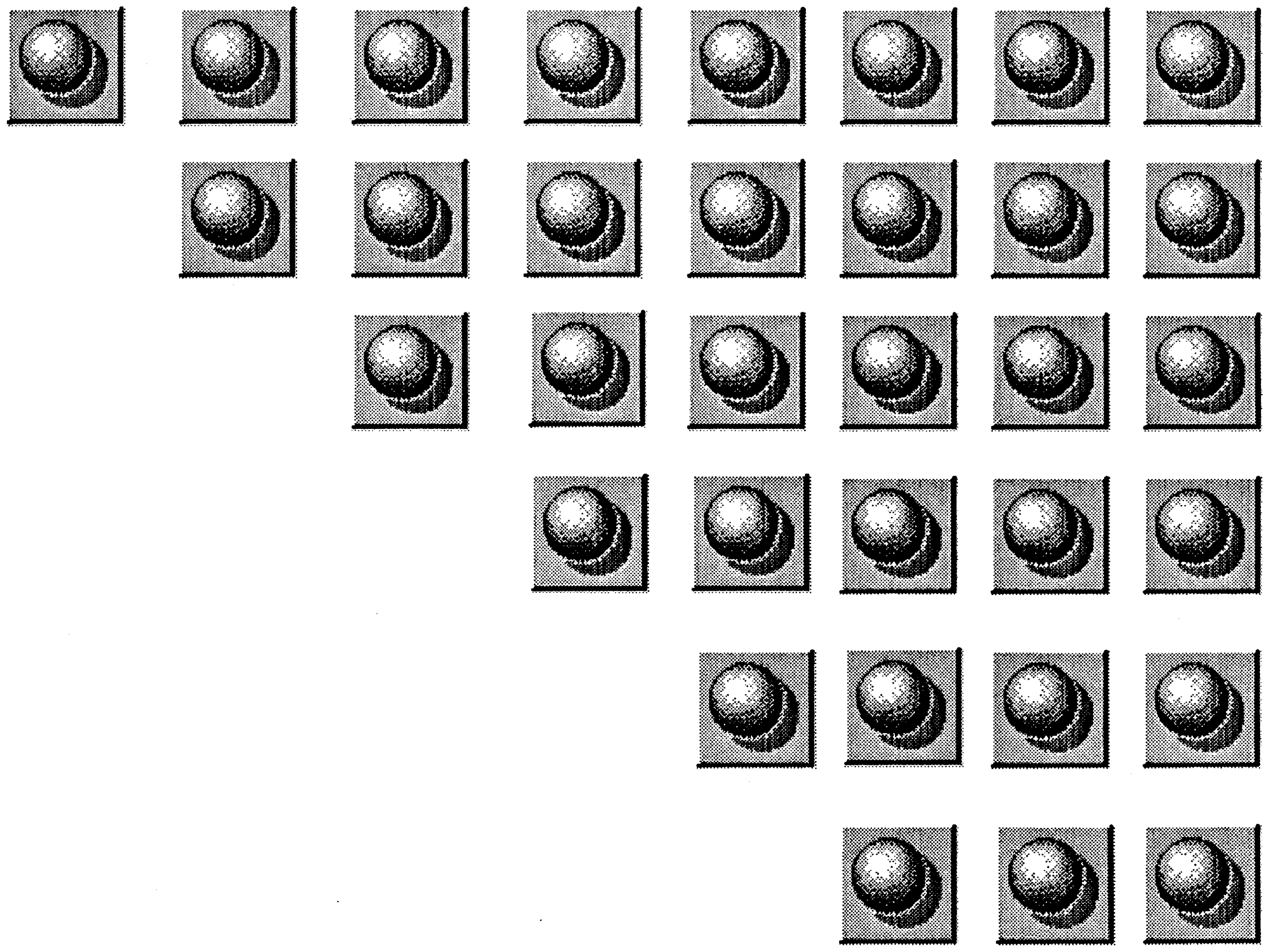
/ Webster uses the openFile:ok: method as public interface */*

```
   speakerResult = [[NXApp appSpeaker]
         openFile: [theWordField stringValue]
         ok: &websterResult];
   if (speakerResult !=0) {
      frpintf(stderr, "message failed.\n");
   }
   return self;
}
```

Hands-On Lab



Summary of Objective-C Syntax



Appendix A: Summary of Objective-C Syntax

This appendix presents a formal grammar for the Objective-C extensions to the C language. It adds to the grammar for ANSI standard C found in Appendix A of *The C Programming Language* (second edition, 1988) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall, and should be read in conjunction with that book.

This appendix follows the conventions used in *The C Programming Language*, with two exceptions:

- Literal symbols are shown in **bold** type.
- Brackets enclose optional elements and are in *italic* type. Literal brackets, like other literal symbols, are nonitalic and bold.

The Objective-C extensions introduce some new symbols (such as *class-interface*), but also make use of symbols (such as *function-definition*) that are explained in the standard C grammar. The symbols mentioned but not explained here are as follows:

compound statement	*identifier*
constant	*parameter-type-list*
declaration	*string*
declaration-list	*struct-declaration-list*
enum-specifier	*struct-or-union*
expression	*typedef-name*
function-definition	*type-name*

Of these, *identifier* and *string* are undefined terminal symbols. Objective-C adds no undefined terminal symbols of its own.

There are three entry points where Objective-C modifies the rules defined for standard C:

- External declarations
- Type specifiers
- Primary expressions

This appendix is therefore divided into these three parts.

External Declarations

external-declaration:
function-definition
declaration
class-interface
class-implementation
category-interface
category-implementation

class-interface:
@interface *class-name* *[* **:** *superclass-name]*
[instance-variables]
[interface-declaration-list]
@end

class-implementation:
@implementation *class-name* *[* **:** *superclass-name]*
[instance-variables]
[implementation-definition-list]
@end

category-interface:
@interface *class-name* **(** *category-name* **)**
[interface-declaration-list]
@end

category-implementation:
@implementation *class-name* **(** *category-name* **)**
[implementation-definition-list]
@end

class-name:
identifier

superclass-name:
identifier

category-name:
identifier

instance-variables:
{ *struct-declaration-list* **}**
{ *struct-declaration-list* **@public** *struct-declaration-list* **}**

interface-declaration-list:
  declaration
  method-declaration
  interface-declaration-list declaration
  interface-declaration-list method-declaration

method-declaration:
  class-method-declaration
  instance-method-declaration

class-method-declaration:
  + [*method-type*] *method-selector* ;

instance-method-declaration:
  – [*method-type*] *method-selector* ;

implementation-definition-list:
  function-definition
  declaration
  method-definition
  implementation-definition-list function-definition
  implementation-definition-list declaration
  implementation-definition-list method-definition

method-definition:
  class-method-definition
  instance-method-definition

class-method-definition:
  + [*method-type*] *method-selector* [*declaration-list*] *compound-statement*

instance-method-definition:
  – [*method-type*] *method-selector* [*declaration-list*] *compound-statement*

method-selector:
  unary-selector
  keyword-selector [, ...]
  keyword-selector [, *parameter-type-list*]

unary-selector:
  selector

keyword-selector:
  keyword-declarator
  keyword-selector keyword-declarator

keyword-declarator:
  : *[method-type] identifier*
  selector **:** *[method-type] identifier*

selector:
  identifier

method-type:
  (*type-name* **)**

Type Specifiers

type-specifier:
  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  struct-or-union-specifier
  enum-specifier
  typedef-name
  class-name

struct-or-union-specifier:
  struct-or-union [identifier] **{** *struct-declaration-list* **}**
  struct-or-union [identifier] **{ @defs (** *class-name* **) }**
  struct-or-union identifier

Primary Expressions

primary-expression:
  identifier
  constant
  string
  (*expression* **)**
  self
  message-expression
  selector-expression
  encode-expression

message-expression:
  [*receiver message-selector* **]**

receiver:
  expression
  class-name
  super

message-selector:
  selector
  keyword-argument-list

keyword-argument-list:
  keyword-argument
  keyword-argument-list keyword-argument

keyword-argument:
  selector **:** *expression*
  : *expression*

selector-expression:
  @selector **(** *selector-name* **)**

selector-name:
  selector
  keyword-name-list

keyword-name-list:
  keyword-name
  keyword-name-list keyword-name

keyword-name:
  selector **:**
  :

encode-expression:
  @encode **(** *type-name* **)**

Introduction to Interface Builder

Introduction

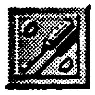
Traditionally, a programmer's first task in a new environment has been to get some input, like the words "Hello, World", to appear on an output device, like the screen or on paper. This gives one a sense of the friendliness of the environment, the type of tools used, the power of those tools and should ideally provide an enticement to continue using the environment.

In using languages like C and Pascal this exercise typically produces five or ten line programs. On the other hand, the complexity of a graphic user interface environment presents a major barrier to doing anything interesting in five or ten lines of code.

Enter Interface Builder. Interface Builder is a tool for creating user interfaces that also provides support for the programming code that underpins those interfaces. This exercise will give you some friendly exposure to this tool.

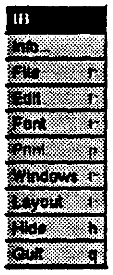
Interface Builder

Launching Interface Builder: To start the Interface Builder application, either double click its icon in the dock (it has a screwdriver above two screw heads) or select the **NextApps** directory in the Workspace Manager's Browser, select **Interface Builder** and double click on the icon in the icon well. The program's icon will highlight while it is launching.

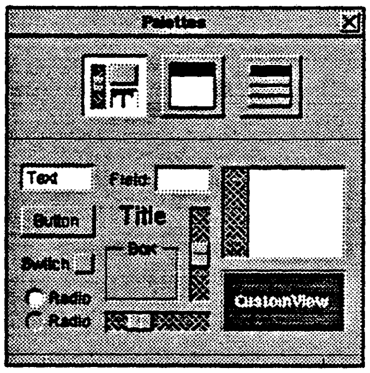


Examining the objects on the screen: Take a look at what you get. When Interface Builder opens you will see two new objects on the screen.

- Interface Builder's Main Menu (labelled **IB**) is in the upper left corner of the screen.

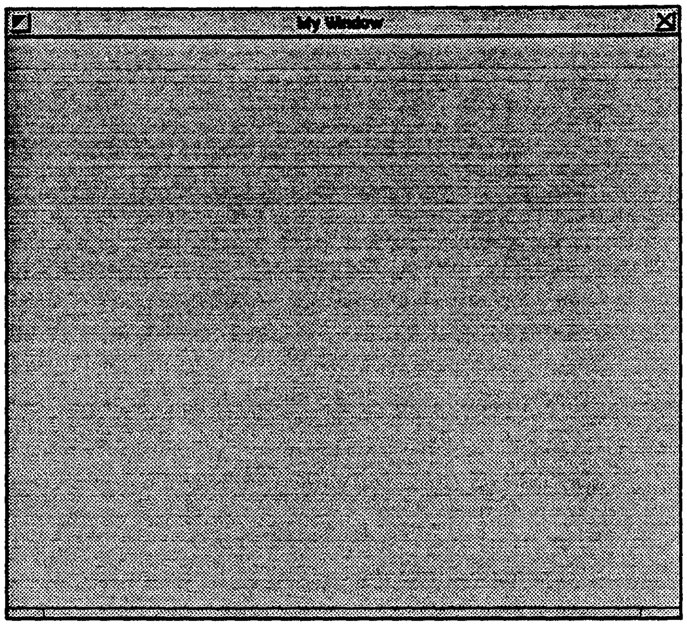


- To the right there is a window called **Palettes**.

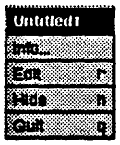


Creating a new application: Under the **File** menu choose **New Application.** Several things happen.

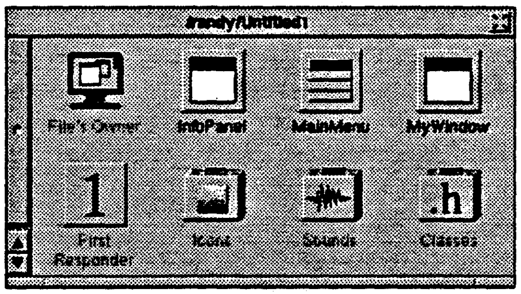
- In the center of the screen is your new window, which is empty.



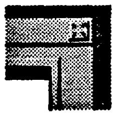
- Beside it is your menu.



There is also another window which looks like an icon view in the Workspace Manager's Browser. This is the *File window.* This contains icons that represent the files that will make up your new application.



Notice that the close box on this window indicates that the file has not been saved.

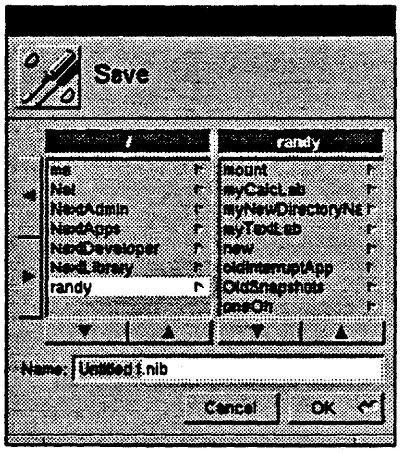


See if you can identify which files belong to the various screen objects. Are there some objects in the File window that don't have a corresponding screen object? Are there any screen objects that don't have a corresponding file icon?

What happens when you click the close box of MyWindow? Can you find a way in the File window to bring the screen object back in to view?

Creating a new directory and saving your work: You know by the partial X that your interface file is not yet saved. Click on the **File** menu and choose **Save**.

You will be greeted with a *Save panel*, a mini-browser that shows the files on the disk.



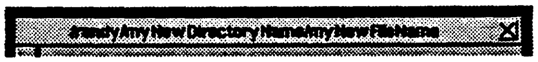
Notice the Save panel comes up open to your home directory with a default filename of **Untitled.nib.** The file extension for Interface Builder files is **.nib** (which stands for NeXT Interface Builder). This file will contain the objects that you specify for your interface (objects like windows and menus).

You need to create a new directory for your interface file. You can do this as you save. In the text field of the Save panel, type a new directory name followed by a name for your interface file:

```
Project1/MyFile
```

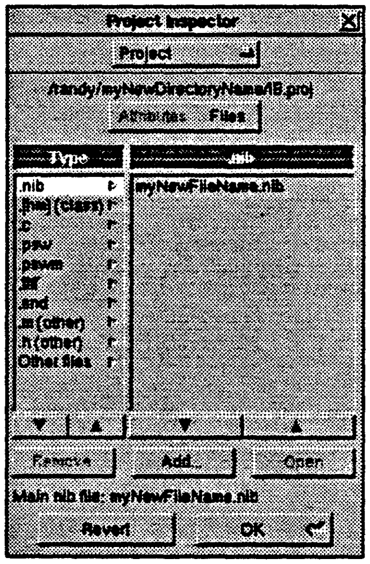
Click **OK**. You will be informed (by the **Bad Path** panel) that the path does not exist and asked if you want to create it. You do indeed. Click on **Create** to create the new directory and save your file in it.

Note that the close box on the file window changes to reflect the fact that the file has been saved and its title bar reflects the file name and path you've chosen. This indication can be useful when you have multiple interface files open in Interface Builder.



Interface Builder requires that you keep all the files for each of your projects in separate directories.

Creating a project file: Interface Builder also allows you to control the other files that you use in a programming project (in addition to your interface file). Under the **File** menu select **Project...** to create a *project file* for your application. You will first be greeted by the *Inspector panel* telling you that "There is no project file ..." and that you should "Click OK to create one". Go ahead and do as you are told. This will put the panel in the mode for inspecting project files.



Notice that the **Project Inspector** comes up indicating that the project has a file of the type **.nib**—an interface file called **MyFile.nib.**

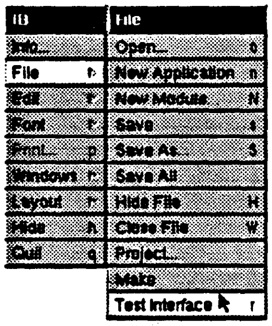
Through this project management tool, Interface Builder keeps track of the various components that you are using to create an application. Note that there is only one project per directory.

Creating a new directory, saving your interface in it and then creating a new project file are three essential steps to creating a new application in Interface Builder.

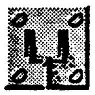
You need to do this for each application before you create any other files—so that those files can be included in your application, tracked by Interface Builder and kept updated.

To review, there is an interface file, and its included in your project. This should be no surprise—you explicitly created and saved the interface file.

Exploring the difference between Test mode and Build mode: So far you have been in Interface Builder's Build mode. On the **File** menu there is a choice called **Test Interface**. Choose that item.



A lot will happen—all of Interface Builder's windows will hide and your new application's windows will be left on the screen. Interface Builder's icon will change to look like a big switch.



- Try moving and resizing the window.

- Try the **Info...** choice from your menu and examine the Info panel.

- Note that the **Hide** command works, but to *unhide*, double click on the Interface Builder's switch icon (as your program doesn't have an icon yet).

In Test mode you get a simulation of how your application's interface will act when someone uses it.

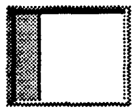
In Build mode you change the way your application's interface looks and what it contains.

To return to Interface Builder's Build mode at any time, double click the switch icon or click **Quit** on your application's menu.

Building the Interface

Locate the **Palettes** window. It's on the upper right side of the screen.

Adding a place for your text: The white field with a gray bar along its left side is an object that displays scrolling text. Drag it into your application's window. Your application will use this to hold and display text.



Now that you have an instance of a scrolling text field in your application's window, resize it by selecting it (clicking it), grabbing a *handle* (one of the little squares that appeared around the object when you selected it) and dragging it to a size you prefer. Note that it stays within the boundaries of the enclosing window.

Testing the scrolling text: Now, try the Test mode once again. Click in the first line of the text field. You should see a blinking insertion point. Try typing some text, like "Hello, World".

- You should be able to select text with the mouse or with the **Select All** option of your Edit menu.

- The selected text can be cut, copied and pasted.

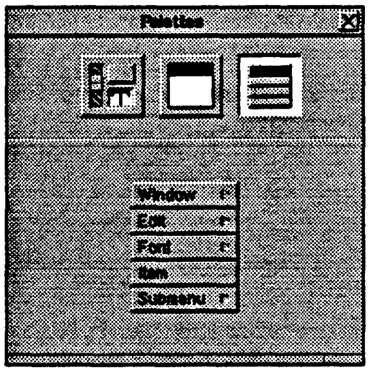
- Note that anything cut or copied from this application can be pasted into any other application that deals with text, and vice versa.

When in Test mode, this is how your application will appear to the person using it. Return to Build mode. Note that any entered text has disappeared.

Save your interface file again by choosing the **Save** command from the **File** menu.

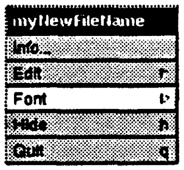
As you can tell from the Test mode, the text object already knows a lot about basic text handling. It can wrap text to a view, it can scroll and it can select text via user input. It also supports text editing via cut, copy and paste. Let's extend the functionality of this little application by adding a menu item to control font attributes.

Adding the Font Menu: Locate the **Palettes** window once again. Click on the far right selector which stands for menus.



This will reveal a menu with some default menu cells. Drag the **Font** menu cell to your application's menu.

Place it just above **Hide** and release.



Check in the File window of your application. Were any new icons added as a result of dragging the Font menu item into the interface?

Testing the Font Menu: Now go into Test mode once again and check the font commands you just added. Note that the changes you make for any character in the window are reflected throughout the window, not just in the selection, (just like the Edit applications).

Leave the Test mode. Once more save your interface file.

Modifying the Interface

Inspecting the Inspector: Let's take a look at the Inspector panel while it's out. This panel will allow you to change and initialize many attributes of many objects. Just below it's title bar is a pop-up list that will allow you to choose which mode you are in. The modes are:

- Attributes—controls colors, styles of objects; their general on screen appearance, ranges and basic state of interface objects.
- Connections—allows you to connect an object to other objects in two ways:
- Autosizing—controls the behavior of objects that are contained in other objects that resize (e.g. a text field inside a window that is being resized by the user).
- Miscellaneous—controls size and placement of certain interface objects as well as their names.
- Class—allows a programmer to describe the outlets and actions of a new class of object, one that might inherit behavior from another class (like a Window object) and add new behavior (like *closeQuick!*).
- Project—allows the management and automatic update of the various files that are the component parts of an application and of its preferred state.

Note that the Inspector's context is the currently selected object. You change that context by selecting a new object—either clicking it's icon in the Files window or clicking the object on the screen.

Personalize the Info panel: Let's touch on the Info panel. Double click its icon in the Files window to bring it on screen so you can edit it.

- By selecting the objects it contains you can modify them directly—or use the Inspector to change their values or appearance.
- To select text, double click the text you want to change.
- Use Interface Builder's **Font** menu to modify your text's attributes.

Give yourself credit as the author of this program in your Info panel by selecting an existing text field (by double clicking) and replacing the resulting selection with your name. Make any other changes you think appropriate. Experiment & be creative.

Use test mode to observe your changes: Try the test mode once again.

- See how your content and style changes are reflected in the test version of your program.
- See if the on screen look is what you want.
- Test the interface for function.
- Try resizing the window.

What happened to the field when the window was resized? If this behavior is not what you'd prefer, can you find a way (using the Inspector) to change how autosizing works?

Exercise the interface, modify it and test it again until you are satisfied.

The cycle of exercise, modify and test is one that you will often use in perfecting a user interface.

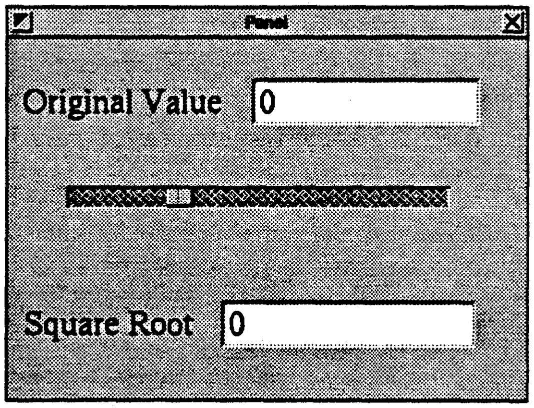
Save all your work using the **Save** command in the **Files** menu of Interface Builder.

Creating a Custom Object

What if you have additional functionality requirements for your application? Now that you have a great user interface, you need to build in your own objects that actually perform whatever task it is that you are asking of the NeXT.

Keep in mind that whenever custom objects are used the application must be compiled for testing the functionality of the custom object. User interface objects will perform their functions, but the custom object will not yet know how to respond.

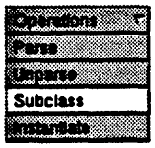
We are going to add a simple object to our application to find the square root of a number and display it in a text window on the screen. Drag a new panel from the palette of user interface objects and arrange it similar to the following image:



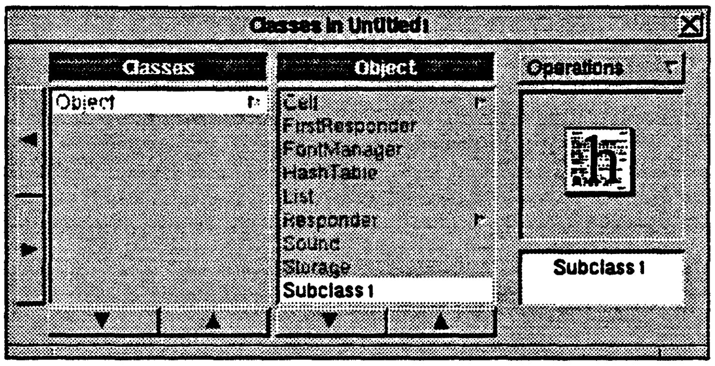
Creating a new class and object: First, we must create a new class of object. Open the Classes window by double clicking on the classes suitcase in the Interface Builder Files window.

This brings up the class browser that displays all of the default classes that are available, and it also allows us to create new classes. Scroll around through the classes. Note that whenever a triangle appears to the left of a class name that it has been **subclassed** to form new classes. What is under the Responder Class?

Now lets create our own object class. We will make it a very general class, a subclass of Object. To do this, first select the item that we wish to subclass, namely Object, by clicking on that entry in the class browser. (It is the far left entry). Next, select the Subclass item from the pull down menu on the class window.



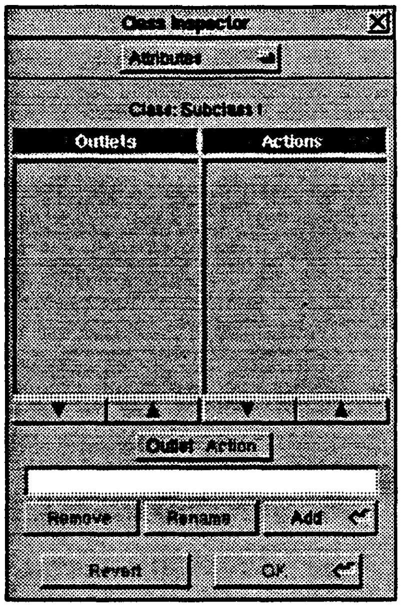
This will create a new class called Subclass1 and your class window should look similar to the following:



Now that we have a new class, we need to create an instance of this class, in other words an object that our application can use. This is referred to as **instantiating** an object. Do this by selecting the **Instantiate** item from the **Operations** pull down menu.

An icon for our new object appears in the Files window and it is named **Subclass1Instance.**

We now need to define the **instance variables and methods** for our new object; within Interface Builder these are called **outlets and actions,** respectively. These are defined by selecting the **Class** item in the **Inspector Window.**



Click the button below the scrolling windows on the **Class Inspector** to toggle whether you will be entering **outlets** or **actions.**

Select **outlets** first and enter the word **originalValue** in the text field in the **Class Inspector** then type RETURN or click the OK button. Enter another outlet named **squareRoot** in the same fashion.

Click the button to toggle to **actions**. Enter the word **calcRoot** and press RETURN. You have now defined you new Class!

Connecting your Custom Object: Connecting our custom object is very similar to connecting other user interface type objects. To connect other objects to action methods within a custom object, just draw a line ***from the object sending the action*** *to the custom objects icon* in the Files window. To connect outlets of the custom object to other objects, ***draw a line from the custom object icon*** *to the object receiving the message.*

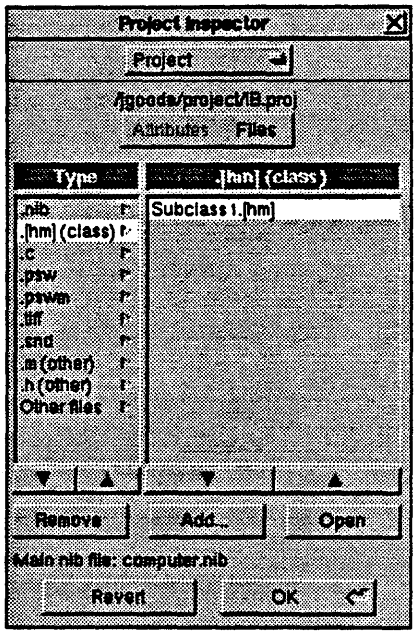
In this case, control-click on the **slider** and draw a line to the **Subclass1Instance** object icon, then select the **calcRoot:** method in the **Connections Inspector.**

Now control-click on the **Subclass1Instance** object icon and draw a line to the text window at the top of your panel and specify a connection to the **originalValue** outlet in the **Connections Inspector.** Repeat the process for connecting the **squareRoot** outlet of **Subclass1Instance** to the lower text window in the panel.

Generating code for your class: Now that the class is defined, we need to write a line or two of code to implement its functionality. Fortunately, the Interface Builder will do most of the work for us and **automatically generate** the framework of the C code needed.

To generate the code, select the **SubClass1** class in the **Class Window** and then choose the **Unparse** item from the **Class Window's** pull down menu. A panel will ask you if you want to generate **SubClass1.h** and **SubClass1.m.** Click OK and the files will be created for you. A second panel will appear asking if you want to add this **Class** to the **project file,** once again click OK.

You can now go in and add your custom code. In the **Project Inspector** window choose [**.hm**](**class**) files under the **Type** column; this will bring up and entry **Subclass1.[hm].** By double clicking on this entry, you will automatically start the **Edit** application and open the 2 files **Subclass1.h** and **Subclass1.m.**



Editing the Interface and Implementation Files: The final operation we need to perform is to actually write the code that will implement our custom object. Notice that all of the skeleton code of the object has been generated for us by Interface Builder. Now this is programming! Using Edit add the following line to the interface file Subclass1.h: (You may add the Bold text items below using cut and paste)

```
/* Generated by Interface Builder */

#import <objc/Object.h>
#import<appkit/appkit.h>
```

```
@interface Subclass1:Object
{
   id                                              originalValue;
   id                                              squareRoot;
}

- setOriginalValue:anObject;
- setSquareRoot:anObject;
- calcRoot:sender;

@end
```

And now add the following lines to Subclass1.m:

```
/* Generated by Interface Builder */

#import "Subclass1.h"
// Include the Unix math library for the square root - sqrt - function
#import <math.h>

@implementation Subclass1

- setOriginalValue:anObject
{
   originalValue = anObject;
   return self;
}

- setSquareRoot:anObject
{
   squareRoot = anObject;
   return self;
}

- calcRoot:sender
{
   // This line places a floating point number taken from the slider (sender)
   // and places it into the originalValue text window
   [originalValue setFloatValue:[sender floatValue]];

   // This line places the square root of a floating point number taken from the slider
   // (sender) and places it into the squareRoot text window
   [squareRoot setFloatValue:(sqrt([sender floatValue]))];
   return self;
}

@end
```

Save both of these files with Edit. You've just completed creating your own custom object! Now move on to the compilation step to check you work.

Compiling the Program

So, you have an interface and some underlying code. Let's compile it.

From the Files menu choose **Make.**

Using the Make utility: This command will cause a command line interface application called **Shell** to run. A command from the **make** utility (a system utility program which controls compiling programs) will be typed for you in the **Shell** window, once the current directory is changed to point to the directory that contains your new interface and project. **Make** will use the interface file you created, the *main()* routine Interface Builder wrote for you automatically and the other information you gave the **Project Inspector** to create an executable version of the program you have been working on.

Testing the compiled version: Once the make process has completed and there are no errors (there shouldn't be) test the compiled version of your application. Launch it from the Workspace Manager's Browser. The newly created application will be called **My File .debug**. It will display a generic application icon.

Once more, give the program a thorough testing.

Congratulations on creating your first program. Pat yourself on the back.

Lab 1

Goals: To develop familiarity with general use of the Interface Builder, including use of Buttons, sounds, images, Inspectors, and Connections.

Getting Started

Make sure your current directory is **/me/Lab1.**

Launch the interface builder. This can be found in **/NextApps/Interface-Builder** or in the application dock appearing as a screwdriver and plate.

Select the **File** entry in the IB menu (top left of screen).
Select **New Application.** This will create an empty window (**My Window**) and a default main menu (**Untitled**).

In the directory browser, select **/me/Images.**
Select **JFK.tiff**, and drag this icon into the **Icons** briefcase appearing in the object panel of IB (lower left of screen). This makes **JFK.tiff** available to IB for use.

In the directory browser, select **/me/Sounds.**
Select **JFK.snd** and drag this icon into the **Sounds** briefcase in the object panel of IB. This make **JFK.snd** available to IB for use.

Populate your window:

1. Drag a series of radio buttons, vertical sliders, switches, and buttons into **My Window**. Experiment with font size, alternate-drag duplication, control-drag adjustment, sizing **My Window**, and sizing buttons.

2. Drag a single button from the **Controls Palette** (top right of screen) to somewhere near the center of **My Window**. Now, open the **Icons** briefcase (by double clicking) and drag **JFK.tiff** on top of this button and release (the button will resize). Reposition the button to the center of **My Window** and close the **Icons** listing (click on the close square).

3. Open the **Sounds** briefcase. Drag **JFK.snd** on top of JFK's image (the sound will play as verification). Close the **Sounds** listing.

4. Select **Windows->Inspector** from IB main menu. Now click on **JFK** button. In the **inspector** select **attributes** from the pull down menu at top. Change the type of button to **On/Off** (be sure to click O.K.).

5. Build a labeled vertical slider on the left of JFK. Place two **titles** and change their value to 100 (at top) and 0 (at bottom).

6. Place two **text** areas (not fields), at the bottom. Enlarge the font and label them 'C' and 'F'.

7. Place a **button** between them, enlarge its font (notice how it resizes) and change its text from **'button'** to **'Convert!'**.

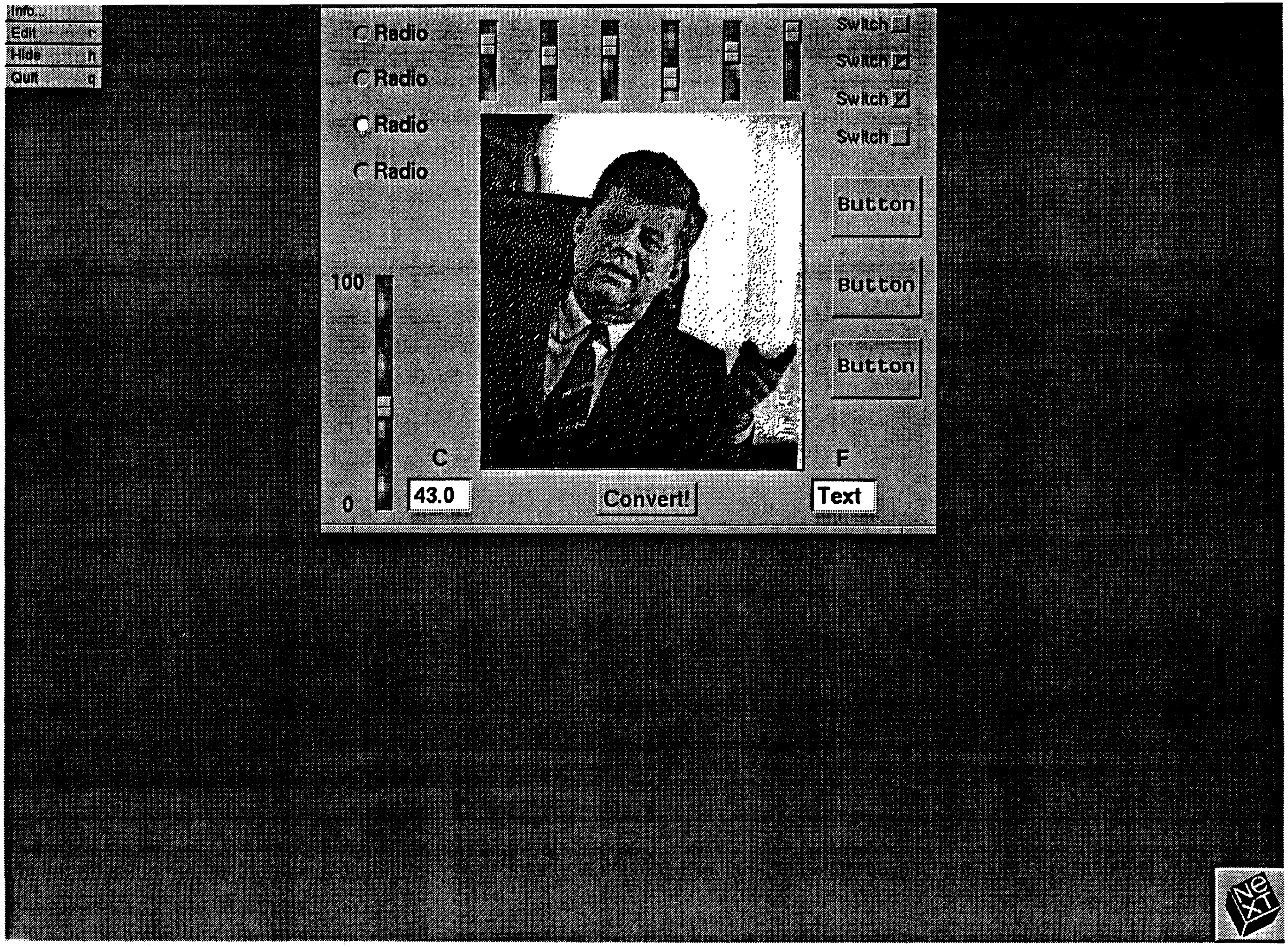
Make a connection and Test it.

1. Control-drag a line from the vertical slider on the left to the **text** item labeled 'C' and release. In the **Connection Inspector** on the right, select the action - **'takeFloatValueFrom'**. Be sure to mouse on connect. This is telling IB you want to display the current value of the slider in the **Text Area.**

2. Test your work by selecting **File ->Test Interface** or by entering the keyboard shortcut - **command-r.**

3. Quit from the test. and save your work by **command-s** or **File->Save.**

4. Quit out of Interface Builder


Info...
Edit
Hide h
Quit q
Radio
Radio
Radio
Radio
Switch
Switch
Switch
Switch
Button
Button
Button
100
0
C
F
43.0
Convert!
Text

Lab2

Goals: Use Class Browser and Project Inspector. Create custom object. Unparse the IB info to create modifiable **'Objective-C'** source files (.h and .m). Add application level 'C' code to existing interface.

Getting Started:

Make sure your current directory is **/me/Lab2.**

Begin by double clicking on **Converter.nib** to launch IB and load the interface work already prepared for you.

Now, create a **Project** folder to keep track of our work. Do this by selecting **File->Project** from the IB main menu, and respond create when prompted.

Create a Custom Object:

1. In the object directory (lower left of screen), double click on **'.classes'** briefcase, this will launch the **Class Browser.** In the **Class Browser** scroll left to find the entry **'Object'**. Click on **'Object'** (it should be the only entry highlighted).

2. Pull down **Operations** to select **'Subclass'**. This will create a new **Class Object** named **'SubClass1'**. Rename this to **'Converter'** by double clicking on the name under the icon well.

3. Now pull down **Operations** again and select **'Instantiate'**. This creates an instance of the class **Converter** (name **converterInstance**) which will be used by IB to link up **actions** and **outlets.**

4. Double click on the **'.h'** icon to bring up a **Class Inspector** (on the right). Make sure that **'Outlet'** is selected in the toggle bar. Type in two **outlet** names

'inputC' and **'outputF'.** Now toggle the bar to select **Action.**
Add in action named **'convert'** (notice IB automatically adds the ':').

Make Connections:

1. Control-drag a link from **converterInstance** (in object listing in lower left) to the two **text** areas and select the appropriate **outlet** **(inputC** for 'C' and **outputF** for 'F'). Make sure you connect them after selecting the outlet.

2. Now connect the **'Convert!'** button to the **converterInstance** and select the action **'convert'** (remember to click connect).

3. Connect the 'C' text field to the button **'Convert!'** and select the action **'performClick'.** This will allow you to enter a number directly into the 'C' field and invoke conversion with a **carriage return.**

Generate and modify source files:

1. From the **Class Browser operations** select **Unparse.** Reply yes to both queries. This will generate two files, **Converter.h** and **Converter.m** which we will modify to do the conversion.

2. In the Project inspector, select Type **'.[hm]'.** Now double click on **Converter.[hm]** to launch **Edit.**

3. In **Converter.h** we need to add **'#import <appkit/appkit.h>** just after the existing **#import** line. Do this and save the file via **command-s.**

4. In **Converter.m** we need to 'flesh out' the **convert** method as it only returns self now. The code need to do this is provided in **/me/Lab2/formula.** Simply edit this file (by double clicking) and then mark, copy, and paste the appropriate lines of code to **Converter.m** then save this file.

5. Finally hide **Edit** with **command-h** and return to IB by mousing on any of

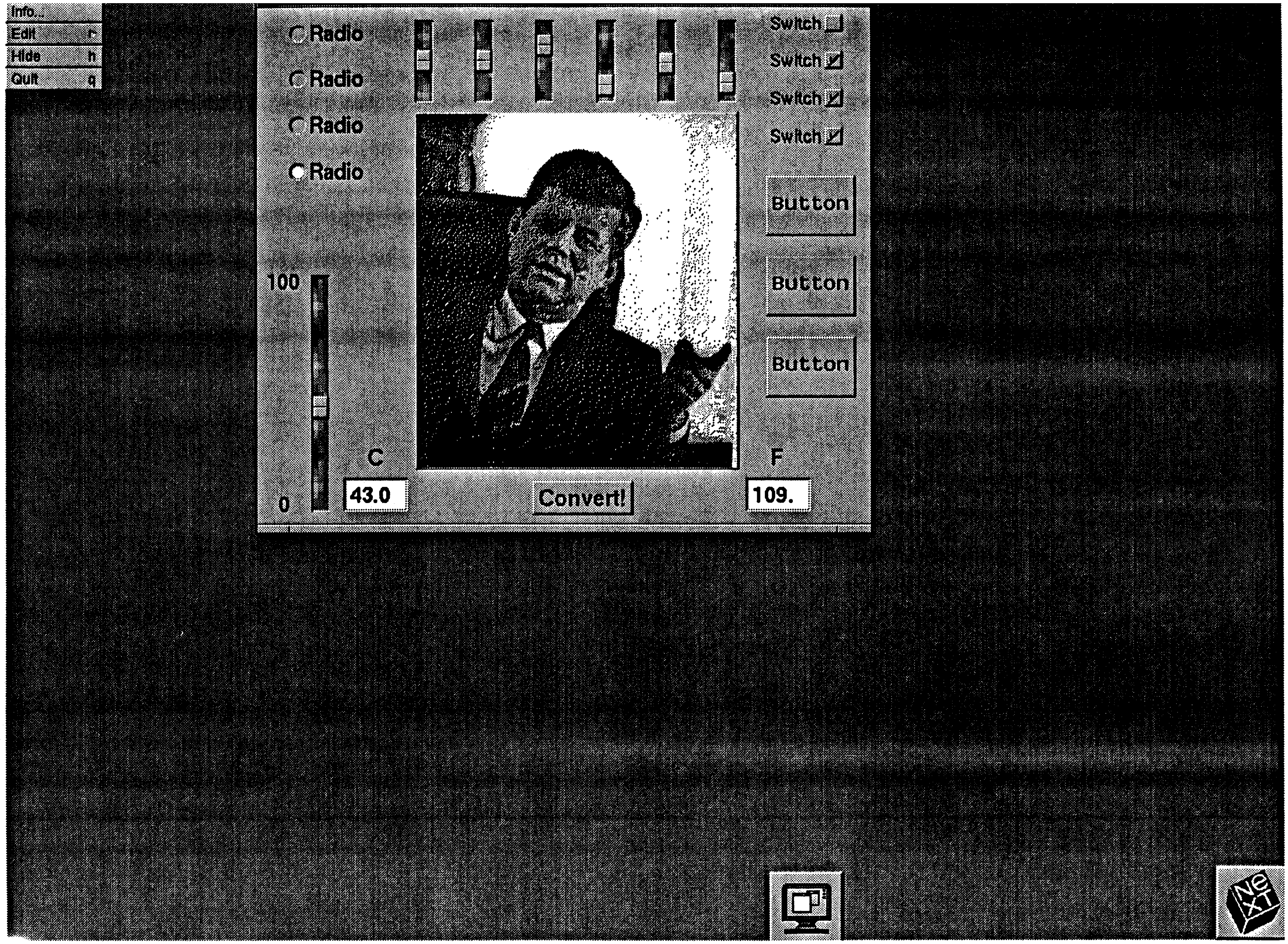
the IB windows.

Make and test the app:

1. Select **File->Make** from the IB main menu, and reply 'Yes' to save the interface. This will launch a shell and run make on the **makefile** created by IB.

2. When this is done, hide IB via **command-h** to clean up the screen, and from the directory browser double click on '**converter.debug**' to launch your application. This is done because **command-r** in IB will only test the **interface**, not any of your custom code.

3. Quit from the app, relaunch IB and quit from it.


Info...
Edit
Hide h
Quit q
Radio
Radio
Radio
Radio
Switch
Switch
Switch
Switch
Button
Button
Button
100
0
C
F
43.0
Convert!
109.

Lab3

Goals: Menu modification, Multiple windows, Parse in existing code, and Intro to PostScript drawing.

Getting Started:

Current directory must be **/me/Lab3**

Launch IB by double-clicking on **IB.proj**. This will launch using a pre-defined project file provided for you.

Create second window and modify main menu:

1. In **Palettes** listing (upper right of screen), select **palette #2** (middle one). This palette provides two objects for your use: a **window** and a **panel.** Drag the window out anywhere on the screen and release. This will create a second window for your app.

2. Now select **palette #3** (far right). This palette provides **cells** to be used to customize your application's menus. Drag the **window cell** to your converter menu. Drop it anywhere you like in the menu (you can move it later if you wish). Change the title to "**Window Two**". Delete New, Save, Save As..., Save To..., and Revert to Saved from the submenu (delete by selecting a cell or a group of cells and pressing the **delete** key).

3. Control drag from the **Open** cell to the **title bar** of your second window and select action '**makeKeyAndOrderFront**'. Connect it. This will allow you to hide the second window and reopen it.

4. While the second window is selected (mouse anywhere in it), use the **Inspector** to modify its **attributes.** Change it to not be **Visible at Launch Time** and change its title to **Window Two.**

5. Select **palette #1** (controls) and drag in a **CustomView** to the second window. Size this to fill half the window and leave room at the bottom for a slider

6. Drag in a horizontal **slider** and size to fit.

7. Drag in a **scrollable text area**, position to the right of your custom view and size to fill that right half.

8. Change the maximum value of the **slider** to 360. Do this by selecting the slider, then using the attributes inspector to set the maximum value.

Create a Custom View - PieView:

1. Launch the **Class Browser**.
Find the **View** class (it is a subclass of **Responder**).
Subclass the **View** class and name it **'PieView'**.

2. **Parse** in the **PieView** files (provided for you) by selecting from the class browser, **Operations->Parse.**

3. Select the **CustomView** in your second window. Make it a **PieView** class by using the **CustomView** Inspector (on right).

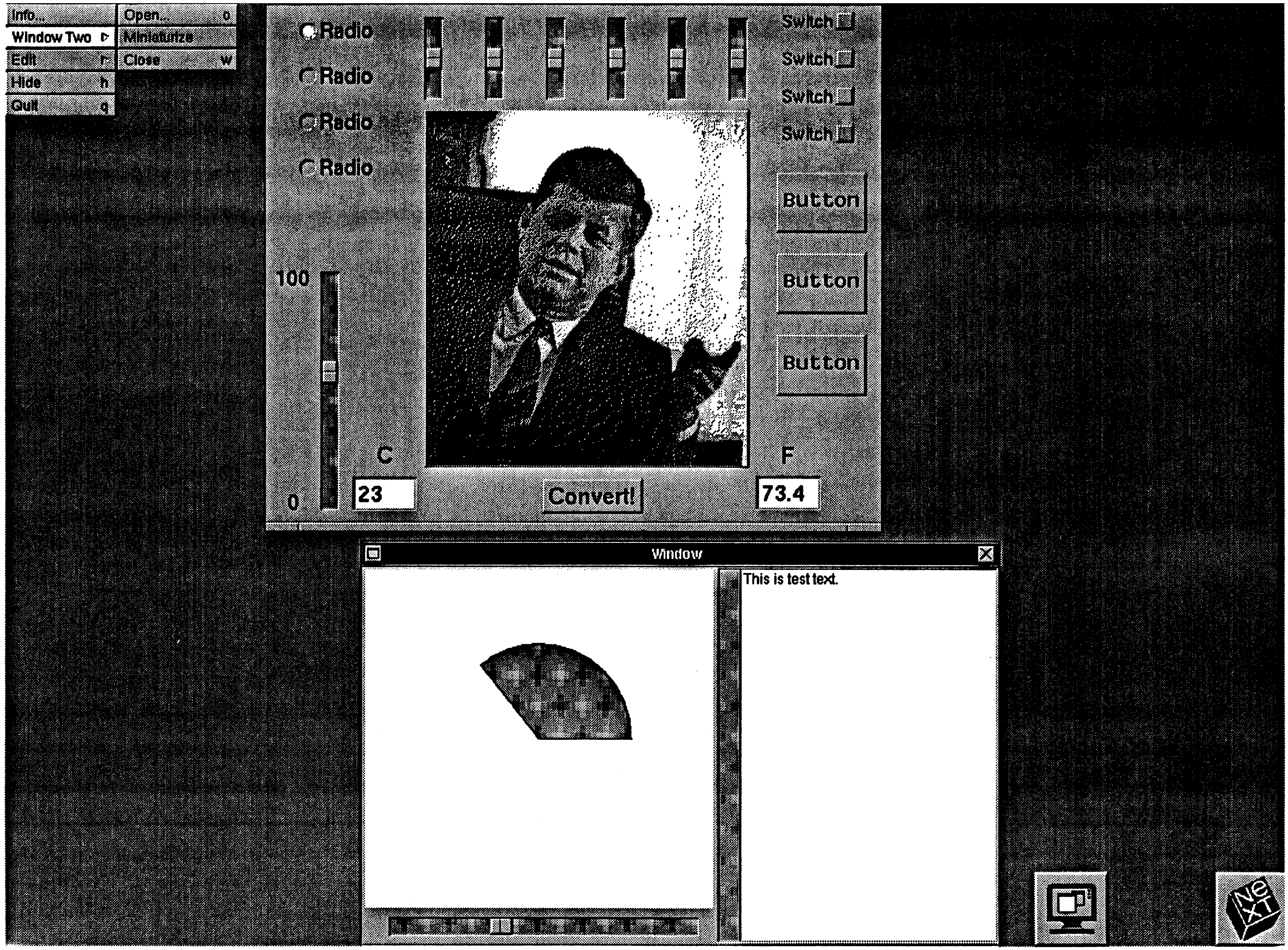
4. Connect the **slider** to the **PieView** (control-drag) and select the action named **'getSlider'**.

5. Add **PieView.m** to the **Project** inspector under type **'.[hm]'**. This tells IB that you need to include these files somewhere in the **makefile.**

Make your app and Test:

Use the same procedure as in **Lab2.** (File->make, reply yes to save, hide IB, wait for shell to finish, and test via **converter.debug**).

Quit your app, quit IB, and hide the shell.


Info...
Window Two
Edit
Hide
Quit
Open...
Miniaturize
Close
Radio
Radio
Radio
Radio
Switch
Switch
Switch
Switch
Button
Button
Button
100
0
C
F
23
Convert!
73.4
Window
This is test text.

Lab4

Goals: Add font control, Add command key, gain familiarity with IPC and Speaker/ Listener model, and Message Webster to define words.

Getting Started:

In **/me/Lab4**, launch IB via **IB.proj**.

More Menu work:

1. Select menu palette.
Drag **Font** cell to converter main menu and position where you like.

2. Drag 'SubMenu' to converter main menu, rename it to "**Request**".
Rename '**Item**' to '**Define in Webster**'.

3. Double click just to the right of '**Webster**' in the menu cell. This will open a small box, type in the character '='. This has just defined a keyboard command equivalent.

Message Webster:

1. Parse in **Converter.m** via the **Class Browser**. Reply yes to replace query. Double click on the '.h' icon to see the new action.

2.Connect the menu sub-item '**Define in Webster**' to the **converterInstance.** This will invoke the messaging method for Webster.

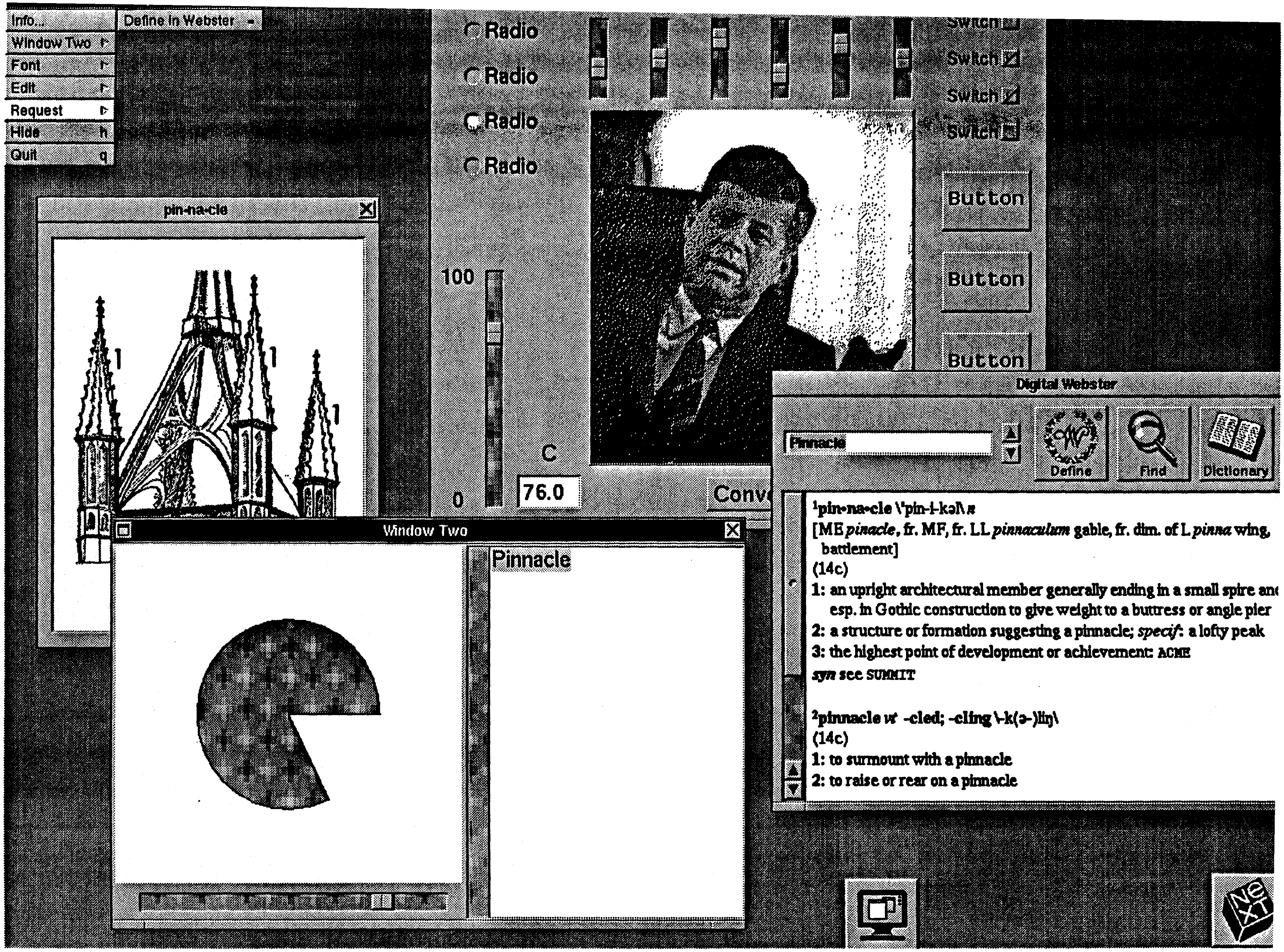
Misc. Clean up (optional):

1. Select **My Window** and select **Attributes** from the **Inspector**. Change the name to **Converter.**

2. Add a **Quit** panel.
Select **palette #2** and drag a **panel** to your application area.
Drop two **Buttons** in this panel.
Rename one Button to **yes,** the other to **no.** Add a **Title** which would say something like **Really Quit?** Select the **quit** cell in **converter** main menu.
Disconnect this item from **File's Owner** action **terminate,** and connect it to

the **Panel** you just added with the action **makeKeyAndOrderFront.**
Now connect the **Yes** button to **File's Owner** (in the object listing at lower left)
and connect the **No** button to
the **Panel** object (in the object listing) with the action **orderOut**.

3. Select the **InfoPanel** object from the Object listing. Change whatever you like.



¹pin•na•cle \ˈpin-i-kəl\ *n*
[ME *pinacle*, fr. MF, fr. LL *pinnaculum* gable, fr. dim. of L *pinna* wing, battlement]
(14c)
1: an upright architectural member generally ending in a small spire an
esp. in Gothic construction to give weight to a buttress or angle pier
2: a structure or formation suggesting a pinnacle; *specif*: a lofty peak
3: the highest point of development or achievement: ACME
syn see SUMMIT

²pinnacle *vt* -cled; -cling \-k(ə-)liŋ\
(14c)
1: to surmount with a pinnacle
2: to raise or rear on a pinnacle