Chapter 3
Hello World:
Creating a New Class of Objects

Some day, far in the future, we will create the first being that can speak intellegently. Even odds the first words we hear will be: "hello, world".

- Anon

Now that you are familiar with using tool-kits to create user interfaces, let's take a look at what we must do to create one of our own objects. Our first program will be very simple and yet very important. It will have one button that when pressed will print out the words "hello, world" in a "UNIX shell" window. It will demonstrate the foundations of the actions required to build our own objects. It will be different from the simple "C" program that is created to do similar things in several ways. The progra will contain an "Object" that prints "hello, world" whenever a button is pressed. After this object is created, we will be able to re-use this object over and over again without recompiling the object. This object will respond to mouse clicks and will aid our understanding of event based programming in a later chapter. This object can also be part of an application object that can later be used to respond to messages from other programs.

Our first task is to start a new project within Interface Builder. If you are already running Interface Builder with another project you should go to the main menu and chose File.. and then Close File. If you are just starting, select File and New Application as in chapter 2. Our interface will be very simple. It will only have one button. This will be connected to one "custom object" we will call "MyObject". When this button is pressed it will run the line of program that will print a message to the shell.

We will create the object using the following steps. You are not expected to understand what is going on in each of these steps. We will take a detailed look at each of them in the next chapter.

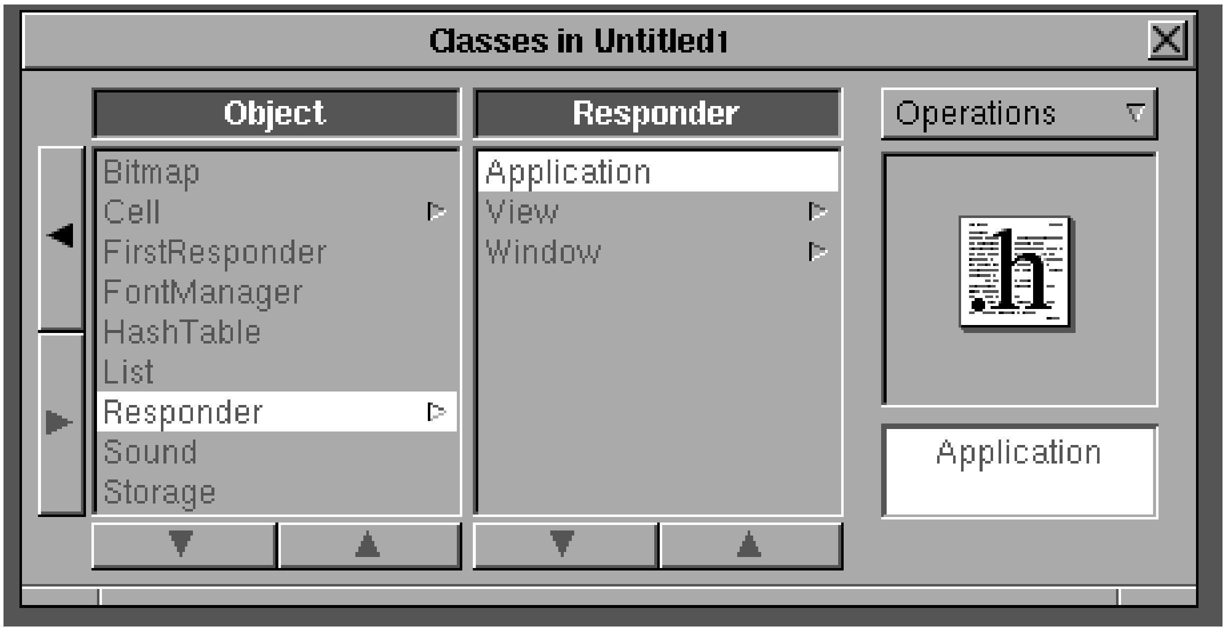
1) Add a button to the main menu.

To do this, drag the object marked "button" from the palette menu and drag it into the main window. You can then double-click on the text of the button and type in the word "hello<CR>". You can then resize the main window to fit around the button so your man window will look like the following:

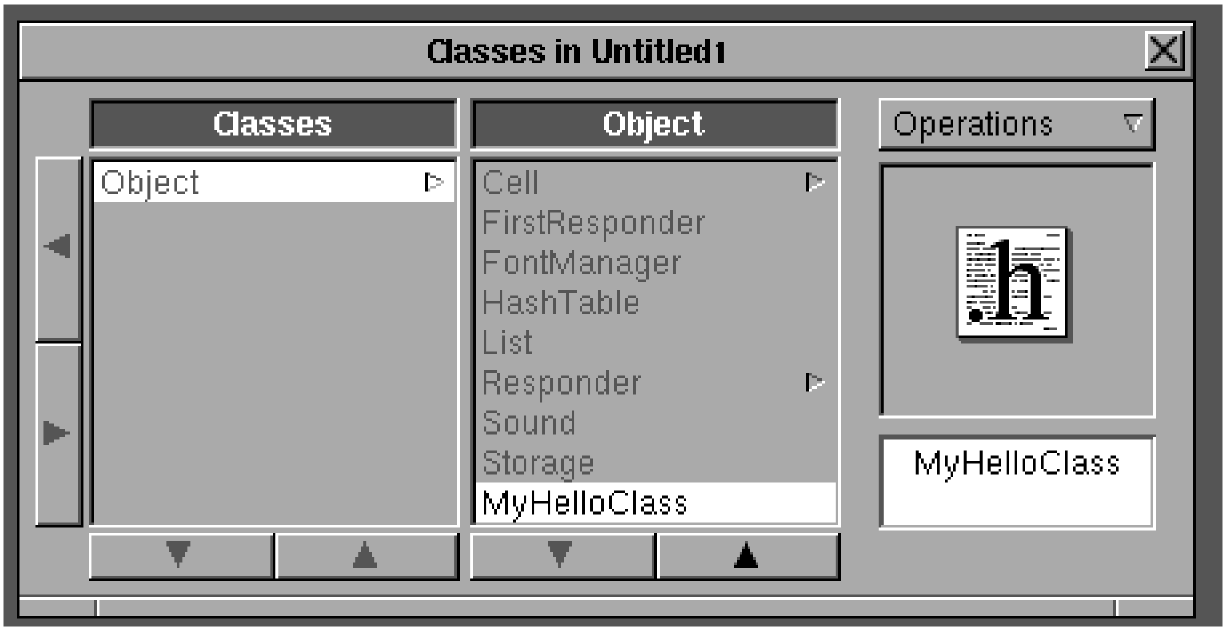
2) Create a sub-class of the "Object" class called MyHelloClass.

This can be done by double clicking on the brief-case icon in the lower left window labeled "Classes". This looks like the icon below:

After you double click this icon a "Class Browser" will appear that looks like the following:



We will first click the black arrow on the left edge of this browser to go to the left-most column so that the word "Object" is the only word showing in the left column. We then want to select that column and verify that the object was selected by noting that the word "object" also appears below the ".h" icon in the right section of the window. We want to use the pop-up list above the ".h" icon labeled "Operations" and select the "Subclass" button. The text below the icon should now be "Subclass1". You should double-click over that text and change it to be "MyHelloClass". The class editor should now look like the following:



Note that we used an uppercase letter to begin the class name. This is an important convention which we will expain in the next chapter.

3) Create an instance of the MyHelloClass

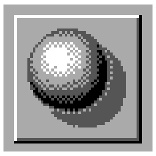
This is done by clicking on the Operations pop-up menu of the class browser and clicking the "Instantiate" selection. You will now notice that the window in the lower left corner of the srceen has an object which has a sphere as its icon (the generic object icon) and below is the text MyHelloClassInstance.

4) Create an action message for MyHelloClass

We will not create a link between our button and our object. This link is a "message" that we will add to our class. Before we do this we need the Inspector window. To get this to appear on the screen we go to the main menu and select Windows and Inspector. Select the MyHelloClassInstance object by clicking it. The inspector should have a pop-up menu at the top of it that is, by default,e set to Attributes. Select this pop-up menu and make the Class slection. In that window you will see two columns: one for outlets, and one for actions. We want the actions selection since pressing our hello button is technically an "action" caused by the user clicking the mouse over the button. To do this click on the button below the two columns so that the word "actions" is in bold. Now type in the word "helloAction" followed by a <Return>. When you are done the Inspector panel should have a single entry in the action column and the text "helloAction:" (Note: the colon is automatically added for you at the end).

5) Make the connection from the button to the instance of the object

This is done by pointing to the button and Control-Dragging a line from the button to the icon in the lower left corner of the screen labeled MyHelloClass Instance.



Icon used for the a generic object

As you release the mouse over the object the inspector will again appear. But this time the pop-up menu will be changed to be the connection panel. After the inspector panel comes up you must click on the word "helloAction" in the column on the right labeled "Actions of the Destination". After you have selected it you must then click on the "Connect" button or enter a carriage return. A small knob will appear next to the message name indicating that it has been connected.

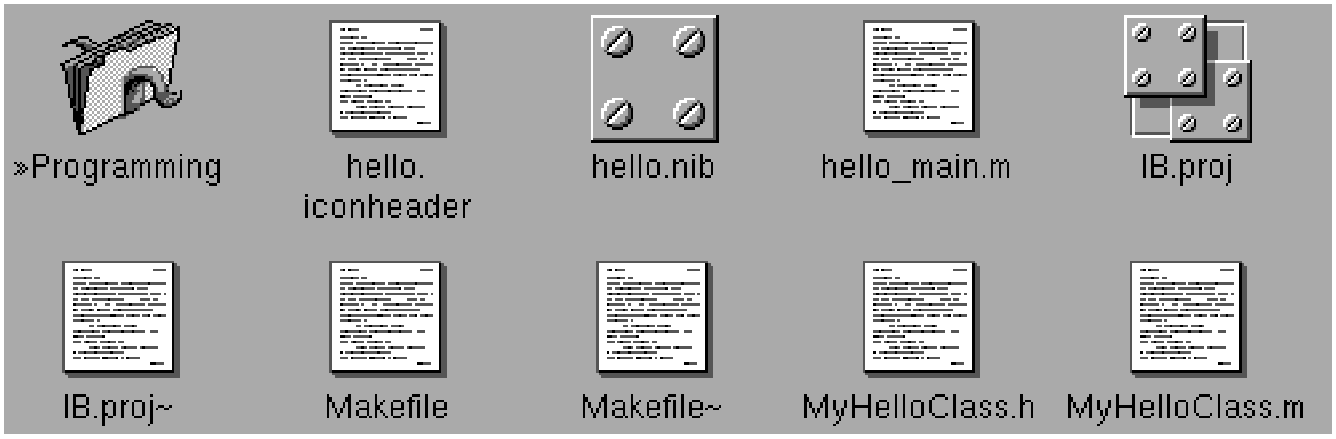
6) Create Objective C files

This actually requires two smaller steps. One involves "Saving" your project in a new directory and the other involves "unparsing" the files related to our new custom object. The first step can be done by going the main menu and selecting the "Files" followed by the "Save". This will bring up a save panel from which you can create the directory and the name of the file you want to save your work in. One suggestion would be to enter the path:

Programming/Hello/hello.nib

This will create the directories (folders) for Programming and Hello if they do not already exist and then save your work in the file "hello.nib". After you have created a directory folder for your work you will also need a file to keep track of all your objects and the steps necessary to compile and link them. This is called a "project" file. It contains information similar to a UNIX Makefile. The user does not need to know what is in the files other that it contains the "recipe" for building the program. This includes things like compiling, linking and installing of the program. To create the project file from the main menu select File and then "Project...". It will bring up an inspector for the project file and tell you there is currently no project file. To create one just enter Return or select the OK button.

Our last step is to create the template file into which we will enter our line of program source code. To do this we need to return to our class editor. Make sure that "MyHelloClass" is the chosen class (MyHelloClass must be below the ".h" icon) and select the Operations pop-up list and then use the "Unparse" selection. It will then ask you if you want to add these files to the project manager. You should select the default "Yes". You have now created all the files you need for the last step. You should be able to view all of the files you created from the browser by selecting the icon view after you are in the Programming/Hello folder. From the workspace Browser, the icon view of these files should be similar to the following:



Note that the files that have the "~" (tilde) characters after them are the backup files created by Interface Builder. You can revert back to these if you make a mistake on the current version.

7) Add print command statement and compile

Our last step will be to add a two lines of code to the file MyHelloClass.m and compile it. The line that will do the work is the following:

```
printf("hello, world\n");
```

To edit the file you can double click on either the line in the project inspector or the icon in the Browser.

The entire file will look like this after you have added this line. The two lines you add are in bold. Comments (which you don't have to add) are all the text after the double slashes (//).

```
/* Generated by Interface Builder */

#import "MyHelloClass.h"
#import <stdio.h>             // add this for type checking

@implementation MyHelloClass

- helloAction:sender
{
     printf("hello, world\n");    // add this line
    return self;
}

@end
```

Note that the one additional line which includes the file <stdio.h> should also be included to get rid of compiler warnings but is not necessary for the program to work. To compile we go to the main Interface Builder menu, select "File" and "Make" and we will see the following messages being sent to a UNIX shell:

```
pushd /dan/Programming/Hello; make debug; popd
mspdemo> pushd /dan/Programming/Hello; make debug; popd
~/Programming/Hello ~
make hello.debug "OFILE_DIR = debug_obj"  "CFLAGS =  -g -DDEBUG -Wimplicit"
mkdirs debug_obj
cc -g -DDEBUG -Wimplicit  -c MyHelloClass.m -o debug_obj/MyHelloClass.o
MyHelloClass.m: In method 'helloAction:'
MyHelloClass.m:10: warning: implicit declaration of function 'printf'
cc -g -DDEBUG -Wimplicit  -c hello_main.m -o debug_obj/hello_main.o
cc -g -DDEBUG -Wimplicit -segcreate __ICON __header hello.iconheader -segcreate __ICON
app /usr/lib/nib/default_app_icon.tiff -segcreate __NIB hello.nib hello.nib -o
hello.debug debug_obj/MyHelloClass.o debug_obj/hello_main.o -lNeXT_s -lsys_s
```

We can then enter the following in the shell:

```
cd Programming/Hello
hello.debug
```

Each time you press the hello button you should see the "hello, world" message printed on your shell output. When you are done select the "quit" from the hello main menu.

Congratulations! If you made it this far you have completed the most difficult section of this entire book: creating your first object. You have created a structure which is encapsulated, you have used Inheritence, your have sent a message from a user interface object (in this case a button object), and you have, perhaps for the first time, used event based programming and connection based programming tools. It is time to pat yourself on the back and tell everyone around you that you have entered the world of object based computing!

But what was it that we really did? What did that class browser have to do with re-useing objects? How do I create new objects that are sub-classes of existing objects and connect them all together? How do I get this new object to send messages to other objects? What if you wanted to have the output of **`printf`** go to a on screen text object rather then the standard output of the shell? These questions and many more will be explained in the next chapter. Before we answer them we need to know a bit more about events and the way they are handeled in the Application Kit.

Exercises

3-1. Create another button and another instance of the MyHelloClass. Connect them together and then save. After you do another make and rerun the program what happens to the output?

3-2. Change the printf statement to the line

```
system("date")
```

You will need to another import file called **`<stdlib.h>`** instead of **`<stdio.h>`**. This will cause the UNIX date command to be run whenever the button is pressed. Can you think of any useful UNIX commands that could use a better user interface?

3-3*. Now that you have finished the program, try it again. This time time yourself. How long do you think an experience object based programmer would need to create a new object? How many lines of code did you have to enter to make this program work? How many lines of code would you need to do this same program in other graphics or iconic programming systems?

3-5 Instead of just the same action taking place, what if you wanted to have two buttons connected up to the same object. How would you change the above procedure?