Chapter 6
Events - Outlets for your Frustrations

It took me a year to learn to create a small program with [my previous system]. I was very frustrated that just to create a small program took me several days and thousands of lines of code. Changing one small feature in an old program was a day long task. Then I started using NeXTstep. In one week I was doing in more in a dozen lines then I could before in a thousand lines of code. Why didn't [my previous computer vendor] do this in the first place?
- Comment from NeXTstep seminar evaluation form

A lot of people are frustrated with their current software development environments. Trying to write a program that uses graphical interfaces is not a trivial task. It takes a lot of time and you sometimes need to thousands of lines of template code to just get the program started. Manageing the flow of events such as mouse clicks, typing text into a window and tracking the mouse as it drags an object accross the screen between windows are all complicated processes that must be carefully studied and then incorporated into templates that you can re-use.

This chapter deals with events. This has little to do object based computing with the exception that we need to use it to manage the way the user interacts with objects on the screen. We will introduce some terms which are used within Interface Builder and then give you a sample program that manages two events to change the state of an object.

Actions: From Events into Code

In traditional programming, programmers think of a program in which the program counter starts at the top of a pogram(in C this is call the "main") and continues in a linear manner through the end of the program. Each line of program code is executed, one after the other until the end of the program is reached. There are ways to make the program counter go in circles (called loops) and their are ways to change the order the instructions get executed, (conditional statements) but the programmer always creates a mental images of the program counter starting at the beginning and finishing at the end. Now we must unlearn much of this pre-concieved ideas of what a program is. With NeXTstep (and with most other graphics based programming environment) we have a two dimensional matrix of objects. When the program starts, the user performs actions (such as a mouse click) on any number of object they see on the screen. These actions cause the window manager to add events to a queue. The operating system then looks at the queue and the events are then dispatched to objects through action messages. In event based programming we think of a two dimensional array of "events" which send "action messages" to various sections of code which respond to these "events". The order of when each section of code is executed does not dependant on the order it appears in the file but on the order that these events occur. So thow out you old mental immages of how a program usually starts executing and start thinking of events triggering action methods.

Example 6-1: Incrementing and Decrementing an Integer

This example is very similar to the "hello, world" program we created in chapter 3.. The difference is we will create two buttons instead of just one, and that we will have each button trigger an event that will change the state of an instance variable. This example is very important. It will be the first time we use an instance variable and hook events up that change the state of an object. This is the fundamental task of most of our applications: we allow users to change the internal states of objects by causing events.

Method:

1) Create a new application with Interface Builder.

2) Drag two buttons in to the main window screen and label them "Increment" and "Decrement". The main window should look like the following:

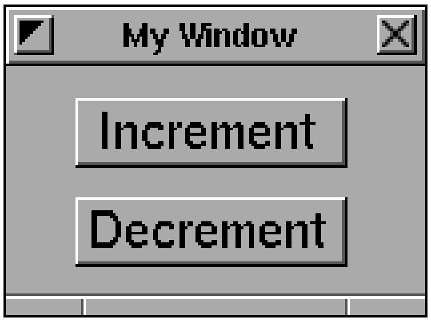

Figure 6-1: Two Buttons that trigger events

3) Create a subclass of the **`Object`** class using the class editor and the create an instance of that object using the "Instantiate" menu selection.

4) With the instance of object selected (the sphere lower left window must be selected) bring up the Inspector (select Windows/Inspector from the main IB meun) and add the following actions using the **`class`** pop-up list:

```
incrementButton
decrementButton
```

5) Connect the buttons to the instance of the object by CONTROL/CLICKing from the buttons to the generic object icon labeled **`MyObject1`** in the lower left hand corner of the screen. Connect the "increment" button to the incrementButton action and the "decrement" button to the "decrementButton" action.

6) Save the Interface builder file and create a project file like in chapter 3. With MyOject selected use the **`unparse`** command and enter Yes when it asks you if you want to add it to the project manager.

7) Add the following line (in bold) in the MyHelloClass.h file:

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

@interface MyObject:Object
{
      int myInt;
}

- incrementButton:sender;
- decrementButton:sender;

@end
```

Now add the following files in **bold** to the MyObject.m files with the following:

```
#import MyObject.h

@implementation MyObject

- incrementButton:sender
{
      myInt++;
      printf("myInt++ = %d\n", myInt);
      return self;
}

- decrementButton:sender
{
      myInt--;
      printf("myInt-- = %d\n", myInt);
      return self;
}
```

After you compile and run, you should see the numbers incrementing and decrementing. If you need to change the initial state of **`myInt`** you will need to use information in Chapter 7 - Initializing Objects.

Outlets: Giving Names to Objects

We have now learned how user actions trigger events and how these events are tied to sections of code in our programs that get executed. This is wonderful if we have a very simple interface. But what if we want to update an object on the screen from within our object? In the Increment/Decrement example above we might want to have a text field on the screen diaplay the current value our integer. How do we "talk to" the text object from within our program? We could have an attribute within the object that holds the name of the object. When we "inspect" the object we could set the name to be **`myText`** and the within our object we could add the line:

```
[myText intValue:myInt]
```

Although this is sometimes done, it also brings into play questions as - what is the scope of the variable myText? How will all the objects that use this name be informed if the name of the object changes? NeXTstep has a method that addresses some of these questions: outlets. **An outlets function is very simple: it assigns an name to a specific object in "object space" that is then used when we want to talk to that object from within our code.** For example in the above example we would use the following steps to add a text object to the screen to display the value of the integer:

Example: Using a Form

Let us take the previous example, the task of having two events change the state of an instance variable. The state of the object was printed out to the standard output, in this case the terminal. But that has the disadvantage that if you start the program from the browser you will not see any output. A better solution is to display the value to a Form object in the main window of the application. Here are the steps.

1) Use Interface Builder to add a Form object to the main window. Change the text in the form to be to be "MyInt:"

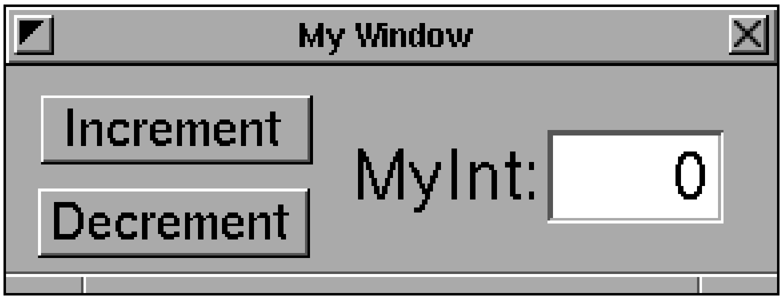

Figure 6-1: Using a Form to Diaplay MyInt

2) Add the following line to the MyObject.h file

-myText:anObject;

3) Parse in the new header file by selecting the MyObject class in the class editor and then using the "parse" selection from the pop-up list. Answer "Replace" to the alert question that informs you that you have different methods.

4) Make a connection from the object instance to the Form (use the gray not the black box)

5) Save IB, compile and run.

Example: HiLo

Suppose we wanted to write the a simple game called HiLo. The computer randomly picks a number between 1 and 100. We are to guess what the number is. The computer will tell us if our guess is too high or to low. The user interface to this program might look like the following dialog:

```
$ hilo
Guess a number from 1 to 100: 50
Guess 1 is too high
Guess a number from 1 to 100: 25
Guess 2 is too low
```

```
Guess a number from 1 to 100: 40
Guess 3 is too high
Guess a number from 1 to 100: 35
Guess 4 is too low
Guess a number from 1 to 100: 37
Guess 5 is too low
Guess a number from 1 to 100: 38
Correct in 6 guesses!
Do you want to play again (y/n)? n
$
```

If we use the traditional C approach, we would always think of starting at a main. We would have the computer pick a random number and then we would use "loops" using commands such as "for", "while" or "repeat" until the user gets the correct answer.

```
#include <stdio.h>
#include <libc.h>
extern void srandom();

main() {
int theNumber, guess, numGuesses;
char still_playing = 'y';
char still_guessing;
   while (still_playing == 'y') {
      srandom(time(0));
      theNumber = random() % ((100)+1);
      numGuesses = 1;
      still_guessing = 'y';
      while (still_guessing == 'y') {
         printf("Guess a number from 1 to 100: ");
         scanf("%d", &guess);
         if (guess == theNumber) {
            printf("Correct in %d guesses!\n", numGuesses);
            still_guessing = 'n';
            printf("Do you want to play again (y/n)? ");
            scanf("%1s", &still_playing);
         }
         else if (guess > theNumber)
            printf("Guess %d is too high\n", numGuesses);
         else if (guess < theNumber)
            printf("Guess %d is too low\n", numGuesses);
         numGuesses ++;
      }
   }
}
```

Figure 1 - Traditional C language program for the game of HiLo

WIth event based programming, we simply "draw" the user interface and then put code behind all of the events that could occur. (See figure 2). In this case, if the user entered a number we would want to check for a match, check to see if the user was too high or too low, and then update the correct areas of the screen. Notice that we would not need to check to see if the user wants to start a new game in the event handler for the input text form. All of that code is neatly seperated out and put in the event handler for the New Game button. The code to handle each event is put in a clear place that corrisponds to the event that it should handle. We don't need to be concerned about the main loop for the game and the loop that constantly checks for the correct answer. One of the added benefits the user can start a new game whenever they want. This is just one example of "user freindlyness" characteristic of event based programs. The user inteacts in two dimensions rather then one.

Figure 6-3: User Interface For Game of Hi-Lo

In this example our object contains two events. One event will be the user entering a new guess. The other will be the event caused when the New Game button is selected. These events can happen at any time. We will also have two outlets - one for sending a hint message to and one for updateing the number of trys. The header file for a HiLo object "HiLo.h" will be the following:

```
#import <objc/Object.h>
@interface HiLo:Object
{
    id hint;
```

```
    id numberGuesses;
    int number;
    int numberOfGuesses;
    int answer;
}

- setHint:anObject;
- setNumberGuesses:anObject;
- newGame:sender;
- guess:sender;

@end
```

The main program is listed below:

```
#import "HiLo.h"
#import <stdio.h>
#import <appkit/Form.h>

@implementation HiLo

// This will run when we start the program - see next chapter
+ new
{
      self = [super new];
      srandom(time(0));
      number = (random() % 100) + 1;
      return self;
}

// this will run when the "New Game" button is pressed
- newGame:sender
{
      number = (random() % 100) + 1;
      numberOfGuesses = 0;
      [numberGuesses setIntValue:0];
      return self;
}

// this will run when the a guess is entered
- guess:sender
{
      answer = [sender intValue];
      if (answer > number) {
            [hint setStringValue:"Too High"];
      }
      else if (answer < number) {
```

```
        [hint setStringValue:"Too Low"];
        }
        else if (answer == number) {
              [hint setStringValue:"That is Correct!"];
        }
        numberOfGuesses++;
        [numberGuesses setIntValue:numberOfGuesses];
        [sender selectTextAt:0];
return self;
}
@end
```

Example: Using a Text object

One of the problems with the "hello, world" program in chapter 3 is that you have to start them from the UNIX command line to see their output. If you start the program by double clicking the icon from the Window Manager's Browser, you will not see any output. Ideally we would like to double click the icon and see the state of an object reflected in an on screen object. To do that we need to use an outlet. To create the program, we do the following:

Step 1: Within Interface Builder, drag another object in the main screen. For starters I would suggest a Text object, although sliders and Form objects can also display values. Your main window should look a bit like the following:

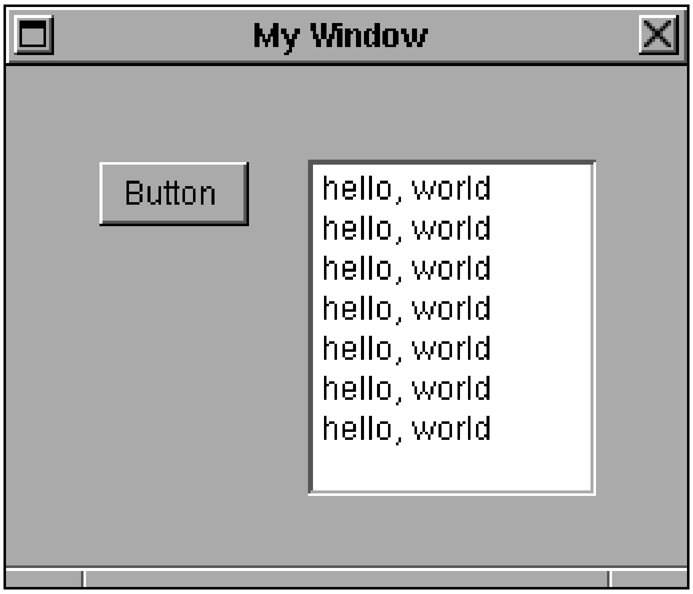

Figure 6-4: Using a Text Object

Create an outlet for MyHelloClass using the inspector panel. Call it textOutlet. This is idential to creating an action with the exception that the outlet is selected instead of an action. Make a connection from the instance of the object to the Form (note that this is the opposite direction that we used when we made a connection from the button to Instance of MyHelloClass).

Step 2: Add the following line to the MyViewClass.h file:

```
/* Generated by Interface Builder */

#import <objc/Object.h>

@interface MyObject:Object
{
    idtextOutlet;
    char myString[10000];
}

- setTextOutlet:anObject;
- buttonAction:sender;

@end
```

This will create an instance variable of type character which will be a region in memory that you can write the string to. Note that in this example I pre-allocated 10000 characters to print to. This should be replaced with a routine that will dynamically allocate memory for a real production program.

Step 3: Add the following lines to the MyViewClass.m file:

```
/* Generated by Interface Builder */

#import "MyObject.h"
#import <appkit/Control.h>
#import <strings.h>

@implementation MyObject

- buttonAction:sender
{
     strcat(myString, "hello, world\n");
     [textOutlet setStringValue:myString];
     return self;
}


@end
```

The function `strcat` will concatinate the string "hello, world" to the end of the string myString every time you press the button. Note that this assumes that myString is null or empty when the program starts up. Not always a good assumption but it seems to work in this case.

Step 3: Save all your files and run make.

Example: Using a Scrolling Text object

The program has a limitation on the number of lines that can be displayed in a text object. After the text object is full you can not see any new lines that are added at the bottom. If instead of using an ordinary text object we used a scrolling text object a scrolling view would appear once the text object was full. Instead of using a text object we will need to drag a scrolling text object form the pallete. Before we send a message to the scrollText object, we have to find out where the text object is inside the scrollText object. In this case, the object that we want is the "document view" which is the document that we are scrolling around in. We then add the following line:

```
scrollText = [textOutlet docView];
```

Instead of sending the **`setStringValue`** message we will need to send the **`setText`** message. The action method for the button then becomes:

```
- buttonAction:sender
{
     strcat(myString, "hello, world\n");
     scrollText = [textOutlet docView];
     [scrollText setText:myString];
     return self;
}
```

The after these changes are made, after the text object gets full, it automatically add a scroll bar to the left hand side. The interface will look like the following after the button has been pressed several times:

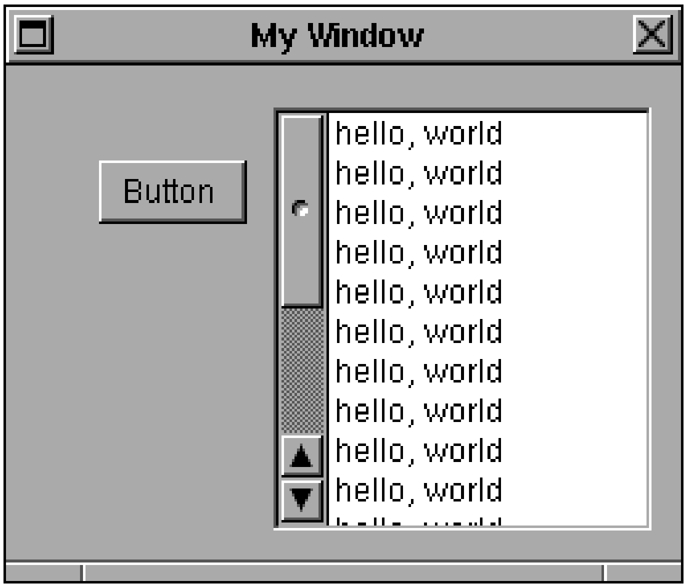

Figure 6-5: Using a Scrolling Text Object