Chapter 7 Initializing Objects: A Tour of the Factory

fac•to•ry \'fak-t(e-)re¯\ n, pl -ries
(1582)
1: a station where resident factors trade
2a: a building or set of buildings with facilities for manufacturing
b: the seat of some kind of production (the vice factories of the slums)
— fac•to•ry•like \-,lı¯k\ adj

We have now introduced the four fundamental techniques used by object based computing systems to allow you to be productive. We will now take a bit more detailed look at how to get going creating objects. One of the first things we often need to do when creating a new object is set an initial value associated with an object. The values inside an object are stored in instance variables. They are the integers, floating point numbers, strings and other data structures inside our black boxes. In Chapter 3 we created an object that printed out the words "hello world" when a button was pressed. Exercise four suggested creating an object that would have two action methods. One would increment the internal state of an object and one would decrement the state of an object. But what if we wanted to start the initial state of the object at a value other then zero? If we put the following lines:

```
// Initialize the state of instance value
myInt = 100;
```

in either the action methods it would get re-initialized every time we press the button. We could add another button called "initialize" and have another action method run this code every time we ran the program, but that would require us to manually reset everything every time the program started up. There is a better way. We talked in earlier chapters about how all objects are "created" form in a factory. That these factories were grouped together is trees and that they defined the class of all objects. If we had a car that we wanted to set some initial conditions on, wouldn't the natural place to do that be in the factory that the car is created? This is the same way that we initialize objects. We send a message to the factory and ask it to perform some tasks for us every time it creates a new object.

Up until now, all of our methods always started with a dash ("-") character or an minus sign. This indicated that each of the messages went to instances of objects. If we start a method with a "+" sign, the message will not go to instances of the object, but will be redirected to go to the factory that created the object. Lets take a look at the code we would insert into exercise 3-4 to initialize the object:

```
+ new
{
    self = [super new];
    myInt = 100;
    return self;
}
```

The first line says "send the **`new`** message to the the superclass of this objects." In this case, we are a sub-class of the Object class so the super class is the Object class. The word **`super`** is a reserved word in Objective C. By sending a message to **`super`**, we can always talk to the superclass of any object we are working with. The result of the **`[super new]`** message is a pointer of type id. It is stored in the variable **`self`**. **`Self`** is a reserve word in Objective C that holds a pointer to the object that we are currently in. **`New`** methods are the only place in our programs that **`self`** will ever be changed. If you ever see self on the left side of the equal sign outside of a new method, waning signs should go up.

The second line initializes **`myInt`** to be 100. The last line returns a pointer to the object we just created. This line is required so other programs that ask for a new instance of our object can reference it by name. If you need a new instance of our object in another program, you should be able to add the lines:

```
id myNewHelloObject;
myNewHelloObject = [MyObject new];
```

You might ask yourself, why do I have to use bother sending the new method to the Object class? Why can't I just use "raw" C to allocate memory and initialize my object? Why can't I just leave out the [super new] line? And indeed, you theoretically can. But this defeats the whole purpose of re-using all the code already done for us. All of the C code to find the size of objects and then allocate memory for them is already done for you. It is in a centralized place and the parts that take a lot of time have been optimized for performance. If a faster or more efficient memory allocation method comes along the people at NeXT will just incorporate it into a single place an all your programs that use the Object classes new method will suddenly become faster. Things like this really start adding up if you play by the rules.

Summary

Whenever you need to initialize the state of a new object which is not a sub-class of View, add the following lines to the beginning of the class implementation file;

```
+ new
{
    self = [super new];
    // your initialization goes here
    return self;
}
```

A subclass of View requires a **`newFrame::`** instead of a new. This is covered in chapter 8: simple drawing.

Note that you don't have to declare the new method in you header file. This is because you are not creating a method from scratch, you are simply adding to the existing new method. The arguments and returned values must thus be the same as in the

Here is the complete program for creating an object that is has an integer as an instance variable, initializes that number to 100 in the factory and has two action methods that increment and decrement the state of the instance variable. The lines we add are in bold. The rest of the lines are created by the "unparse" operation form Interface Builder.

Contents of MyObject.h

```
/* Generated by Interface Builder */

#import <objc/Object.h>

@interface MyObject:Object
{
      int myInt;
}

+ new;
- event1:sender;
- event2:sender;

@end
```

Contents of MyObject.m

```
/* Generated by Interface Builder */

#import "MyObject.h"
#import <stdio.h>

@implementation MyObject

+ new {
      self = [super new];
      myInt = 100;
      return self;
}

- event1:sender
{
      myInt++;
      printf("myInt++ = %d\n", myInt);
      return self;
}

- event2:sender
{
      myInt--;
      printf("myInt-- = %d\n", myInt);
      return self;
}

@end
```

For advanced readers:

There are times when you also want to separate initialization steps from the new method. For example if you have an object that you want to reset to a set of default values you would like to initialize the state of the object when it came out of the factory but not re-allocate the memory. In this case, the Objective C convention is to use an **`-initialize`** method[1]. This allows you to over-ride other objects initialization code. Your new would look like the following:

```
+ new
{
   self = [super new];
   [self initialize];
   return self;
}

-initialize
{
   // the following used only if the super class has an initialize
   [super initialize];
   // add your own initialization code here
   return self;
}
```

Exercises

6-1. If you added the following line in the new method:

```
+ new {
      self = [super new];
      myInt = 100;
      printf("new: myInt initialized to %d\n", myInt);
      return self;
}
```

When would you expect the printf to be executed? If you said when the program starts up or "launches" you would be right.

6-2 There are several ways to initialize complex C data structures. What happens if you use a string as an instance variable? What about a matrix of integers? See a copy of A C

[1]See Journal of Object Oriented Programming, Summer 1990, John Hopkins article