Stephen G. Kochan

Updated for iOS5 and ARC

Programming in Objective-C

Fourth Edition

Developer's Library

Programming in Objective-C

Fourth Edition

says that instead of `ClassB` being a subclass of `NSObject`, `ClassB` is a subclass of `ClassA`. So although `ClassA`'s parent (or superclass) is `NSObject`, `ClassB`'s parent is `ClassA`. Figure 8.2 illustrates this.

As you can see from Figure 8.2, the root class has no superclass and `ClassB`, which is at the bottom of the hierarchy, has no subclass. Therefore, `ClassA` is a subclass of `NSObject`, and `ClassB` is a subclass of `ClassA` and also of `NSObject` (technically, it's a sub-subclass, or *grandchild*). Also, `NSObject` is a superclass of `ClassA`, which is a superclass of `ClassB`. `NSObject` is also a superclass of `ClassB` because it exists farther down its hierarchy.

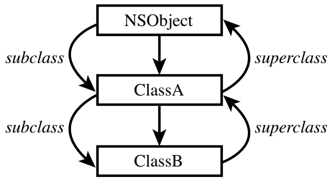

Figure 8.2 Subclasses and superclasses

Here's the full declaration for `ClassB`, which defines one method called `printVar`:

```
@interface ClassB: ClassA
-(void) printVar;
@end

@implementation ClassB
-(void) printVar
{
   NSLog (@"x = %i", x);
}
@end
```

The `printVar` method prints the value of the instance variable `x`, yet you haven't defined any instance variables in `ClassB`. That's because `ClassB` is a subclass of `ClassA`—therefore, it inherits `ClassA`'s public instance variables (in this case, there's just one). Figure 8.3 depicts this.

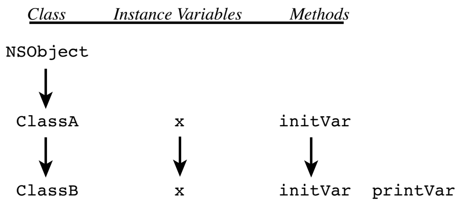

Figure 8.3 Inheriting instance variables and methods.

(Of course, Figure 8.3 doesn't show any of the methods or instance variables that are inherited from the `NSObject` class—there are several.)

Let's see how this works by putting it all together in a complete program example. For the sake of brevity, we'll put all the class declarations and definitions into a single file (see Program 8.1).

Program 8.1

```
// Simple example to illustrate inheritance

#import <Foundation/Foundation.h>

// ClassA declaration and definition

@interface ClassA: NSObject
{
   int  x;
}

-(void) initVar;
@end

@implementation ClassA
-(void) initVar
{
  x = 100;
}
@end

// Class B declaration and definition

@interface ClassB : ClassA
-(void) printVar;
@end

@implementation ClassB
-(void) printVar
{
  NSLog (@"x = %i", x);
}
@end

int main (int argc, char * argv[])
{
   @autoreleasepool {
      ClassB  *b = [[ClassB alloc] init];
      [b initVar];     // will use inherited method
      [b printVar];    // reveal value of x;
   }
   return 0;
}
```

Program 8.1 Output

```
x = 100
```

You begin by defining `b` to be a `ClassB` object. After allocating and initializing `b`, you send a message to apply the `initVar` method to it. But looking back at the definition of `ClassB`, you'll notice that you never defined such a method. `initVar` was defined in `ClassA`, and because `ClassA` is the parent of `ClassB`, `ClassB` gets to use all of `ClassA`'s methods. So with respect to `ClassB`, `initVar` is an *inherited* method.

Note

We briefly mentioned it up to this point, but `alloc` and `init` are methods you have used all along that are never defined in your classes. That's because you took advantage of the fact that they were inherited methods from the `NSObject` class.

After sending the `initVar` message to `b`, you invoke the `printVar` method to display the value of the instance variable `x`. The output of `x = 100` confirms that `printVar` was capable of accessing this instance variable. That's because, as with the `initVar` method, it was inherited.

Remember that the concept of inheritance works all the way down the chain. So if you defined a new class called `ClassC`, whose parent class was `ClassB`, like so

```
@interface ClassC: ClassB
   ...
@end
```

then `ClassC` would inherit all of `ClassB`'s methods and instance variables, which in turn inherited all of `ClassA`'s methods and instance variables, which in turn inherited all of `NSObject`'s methods and instance variables.

Be sure you understand that each instance of a class gets its own instance variables, even if they're inherited. A `ClassC` object and a `ClassB` object would therefore each have their own distinct instance variables.

Finding the Right Method

When you send a message to an object, you might wonder how the correct method is chosen to apply to that object. The rules are actually quite simple. First, the class to which the object belongs is checked to see whether a method is explicitly defined in that class with the specific name. If it is, that's the method that is used. If it's not defined there, the parent class is checked. If the method is defined there, that's what is used. If not, the search continues.

Parent classes are checked until one of two things happens: Either you find a class that contains the specified method or you don't find the method after going all the way back to the root class. If the first occurs, you're all set; if the second occurs, you have a problem, and a warning message is generated that looks like this:

```
warning: 'ClassB' may not respond to '-inity'
```

In this case, you inadvertently are trying to send a message called `inity` to a variable of type class `ClassB`. The compiler told you that objects from that class do not know how to respond to such a method. Again, this was determined after checking `ClassB`'s methods and its parents' methods back to the root class (which, in this case, is `NSObject`).

You'll learn more about how the system checks for the right method to execute in Chapter 9, "Polymorphism, Dynamic Typing, and Dynamic Binding."

Extension Through Inheritance: Adding New Methods

Inheritance often is used to extend a class. As an example, let's assume that you've just been assigned the task of developing some classes to work with 2D graphical objects such as rectangles, circles, and triangles. For now, we'll worry about just rectangles. Let's go back to exercise 7 from Chapter 4, "Data Types and Expressions," and start with the `@interface` section from that example:

```
@interface Rectangle: NSObject
@property int width, height;
-(int)  area;
-(int)  perimeter;
@end
```

You'll have synthesized methods to set the rectangle's width and height and to return those values, and your own methods to calculate its area and perimeter. Let's add a method that will allow you to set both the width and the height of the rectangle with the same message call, which is as follows:

```
-(void) setWidth: (int) w andHeight: (int) h;
```

Assume that you typed this new class declaration into a file called `Rectangle.h`. Here's what the implementation file `Rectangle.m` might look like:

```
#import "Rectangle.h"
@implementation Rectangle

@synthesize width, height;
```

```
-(void) setWidth: (int) w andHeight: (int) h
{
    width = w;
    height = h;
}

-(int) area
{
    return width * height;
}

-(int) perimeter
{
    return (width + height) * 2;
}
@end
```

Each method definition is straightforward enough. Program 8.2 shows a `main` routine to test it.

Program 8.2

```
#import "Rectangle.h"

int main (int argc, char * argv[])
{
   @autoreleasepool {
      Rectangle *myRect = [[Rectangle alloc] init];

      [myRect setWidth: 5 andHeight: 8];

      NSLog (@"Rectangle: w = %i, h = %i", myRect.width, myRect.height);
      NSLog (@"Area = %i, Perimeter = %i", [myRect area],
                 [myRect perimeter]);
    }
    return 0;
}
```

Program 8.2 Output

```
Rectangle: w = 5, h = 8
Area = 40, Perimeter = 26
```

`myRect` is allocated and initialized; then its width is set to `5` and its height to `8`. The first line of output verifies this. Next, the area and the perimeter of the rectangle are calculated with the appropriate message calls, and the returned values are handed off to `NSLog` to be displayed.

Suppose that you now need to work with squares. You could define a new class called `Square` and define similar methods in it as in your `Rectangle` class. Alternately, you could recognize the fact that a square is just a special case of a rectangle whose width and height just happen to be the same.

Thus, an easy way to handle this is to make a new class called `Square` and have it be a subclass of `Rectangle`. For now, the only methods you might want to add would be to set the side of the square to a particular value and retrieve that value. Program 8.3 shows the interface and implementation files for your new `Square` class.

Program 8.3 **`Square.h`** Interface File

```
#import "Rectangle.h"

@interface Square: Rectangle

-(void) setSide: (int) s;
-(int) side;
@end
```

Program 8.3 **`Square.m`** Implementation File

```
#import "Square.h"

@implementation Square: Rectangle

-(void) setSide: (int) s
{
    [self setWidth: s andHeight: s];
}

-(int) side

{
   return self.width;
}
@end
```

Notice what you did here. You defined your `Square` class to be a subclass of `Rectangle`, which is declared in the header file `Rectangle.h`. You didn't need to add any instance variables here, but you did add new methods called `setSide:` and `side`. Note that the `side` method does not have direct access to the `Rectangle`'s `width` instance variable; it's private and therefore not accessible by the `Square` class. However, the getter