5

Back to ObjectLand

Contents at: 'Chapter 5'

Questions of Interest

- When and what do I subclass?
- How can I benefit from inheritance?

- When do I use instance and class variables?
- How about Global and Pool variables?

Introduction

Earlier, you took a quick trip to ObjectLand and gained an understanding of the object-oriented programming paradigm. You learned a little about how to think of objects as computational entities, and a little about how to design software from objects and messages. Much later, you will discuss object-oriented thinking and design in greater detail. In this chapter, you put yourself back in ObjectLand to learn the rest of the story about the object-oriented paradigm.

You find that there are more concepts to learn:

1. encapsulation
2. polymorphism
3. inheritance

You find, too, additional information about how the system uses these capabilities.

You also begin to learn more about the organization of the class hierarchy.

Goals for This Chapter

- To learn about the object-oriented paradigm and to map some small problems into object-oriented representations.
- To learn to use encapsulation, polymorphism, and inheritance in our design and implementation.

JIM

I'm back! I love writing code in Smalltalk!

OBJECTIVE WIZARD

Good! Let us get to work! We have been making excellent progress. On this trip, I would like to introduce you to some of the moral fiber of ObjectLand. There are three underlying capabilities that are particularly interesting. We call them

encapsulation

polymorphism

inheritance

Let us look at them one at a time.

I'm game.

What is encapsulation?

Encapsulation is a big word used to describe the privacy of an object.

How can I use it?

You do not have a choice; it is enforced. Objects are encapsulated by nature, but you can design objects whose role in life is to encapsulate. These objects are called, as you might expect, "encapsulators."

There's no way I can avoid encapsulation?

No. As you probably understand in human terms, privacy is very important—but you can choose either to use encapsulation or to try to work around it. Encapsulation is not enforced to make your life difficult; instead, it is a way to keep the inside of an object from getting unexpectedly, and incorrectly, altered.

I think I see. Encapsulation keeps objects from getting trashed by some other object. They keep their original integrity because they are encapsulated.

JIM

OBJECTIVE WIZARD

Well put. And by knowing that encapsulation exists and why it is useful, you can write better code.

I don't quite see how encapsulation functions. If the object is private, how can I work with it?

An object can be accessed only by sending it a message. If an object's message interface does not include a way to access an instance variable directly, then you have no way to look at or change that variable from outside the object.

So, if I need to access some information that's kept in an instance variable, I write methods that do it.

Yes! Very good, Jim. You are learning quickly that you can work with the object's privacy, or encapsulation. Writing these kinds of messages is such a common activity that they have names. A *get* method is one written just to get and answer the value of an instance variable. A *set* method is one written just to set the value of an instance variable.

I see.

And what is polymorphism?

Polymorphism refers literally to the capability to assume different forms. In Smalltalk, polymorphism refers to the ability of a given message to assume different functions, depending on the object that receives the message.

How can a message function differently, depending on what object it is sent to?

Keep in mind that the message is a selector. The method is the actual code that results in the functionality. Polymorphism relies on this difference between message and method and the fact that Smalltalk uses dynamic, or late, binding of message to method. In Smalltalk, the binding of a message to the method it will activate occurs when the message is sent. The method to be activated is chosen by the receiving object.

JIM

OBJECTIVE WIZARD

So, it is really the receiving object that determines exactly what method is executed and, consequently, what function is performed.

How about some examples?

Sure! Look at the following code:

Expression	Answer
`3 + 4`	`7`
`3.2 + 3.14`	`6.34`
`(10@30) + (10@10)`	`20@40`
`(1/2) + (32/64)`	`1`

Here, the message `+` is sent to instances of four different classes, as follows: `Integer`, `Float`, `Point`, and `Fraction`. In each case, the method that is activated answers the sum of the receiver and the argument; each summation, however, is done in a different way.

For example, while the sum of two integers is simply a primitive operation with no further Smalltalk code, the sum of two points looks like this code:

```
+ aPoint
      "Answer the sum of myself
       and aPoint."
   ^Point new
      x: (self x + aPoint x)
      y: (self y + aPoint y).
```

So, the `+` message is polymorphic because it is part of the message interface for several classes, and each class implements it in the way that is most appropriate.

Different object-oriented languages have slightly different implementations of polymorphism, some of which are limited in some way. Smalltalk's polymorphism works all the time for every message send.

How about some more examples?

JIM

How can I use polymorphism in my code?

What is the best way to learn more about polymorphism?

OBJECTIVE WIZARD

Sure! Look at the following code:

Expression	Answer
`'ObjectLand' size.`	`10`
`#(q w e r t y ) size.`	`6`
`('one', 'two') size.`	`6`
`Set new addAll: 'one', 'two'; size.`	`5`

Here, the size message is sent to instances of classes `String`, `Array`, and `Set`. Each of these subclasses of `collection` answers the number of elements it contains, but each computes its number of elements in a different way.

Notice that, in all these cases, `size` answers the number of elements in the object to which it was sent, even though the method that computed the size was different. Two objects receiving the same message should behave in a similar way.

Well, the first step is to learn the messages that are used polymorphically in the class hierarchy. This learning must be accomplished through experience but can be augmented by selecting a message in a class hierarchy browser and using the **implementors** option in the methods menu. The implementors window will then open to a list of all classes that implement the message you have chosen. If there is only one item in the list, then the message you have chosen is not used by more than one class and, therefore, does not take advantage of polymorphism. If there are many items, then you can see what classes implement the message.

The second step is to design using polymorphism. The robot exercises in the To Do List at the end of this chapter are good examples for designing with polymorphism.

JIM

OBJECTIVE WIZARD

The best way to learn is to do. In your case, you will learn by designing your projects to use polymorphism. It is particularly effective when used with inheritance.

Inheritance?

Yes, inheritance, our next topic.

Inheritance is a powerful problem-solving tool available to Squeak programmers. It is conceptually very simple. All Smalltalk classes exist in a class-subclass hierarchy. When you view the class hierarchy using a System Browser, you will notice that some classes in the class list are indented. An indented class is a subclass of the class from which it is indented. Look at the following example:

```
Number
   Float
   Fraction
   Integer
```

In this example, the class `Number` is the superclass of `Float`, `Fraction`, and `Integer`. Conversely, `Float`, `Fraction`, and `Integer` are subclasses of `Number`.

So, how is this class-subclass relationship useful?

Think of inheritance as a problem-solving tool rather than as a concept. If you think of it as a concept, then it becomes something you must learn. If you think of it as a problem-solving tool, then it becomes something you can use. Do you understand the distinction?

Yes, but I still don't know what's useful about it.

In problem solving, it is often useful first to solve a piece of the problem in a general sense and then to solve that piece in a more specific way by filling in the details that specialize the general solution to the specific problem you are trying to solve.

With inheritance, you can solve the general characteristics of the problem in a superclass and create subclasses to implement the specific solutions. The `Number` classes are a good

JIM

OBJECTIVE WIZARD

example of this approach. (Class `Number` is a very general solution to the problem of what it means to be a number.) Much of the number behavior is implemented in the `Number` class, but the subclasses implement specialized versions of numbers (that is, `Integer`, `Float`, and `Fraction`). When inheritance is used, the subclasses inherit all the variables and methods implemented in the `Number` class and then proceed to add their own variables and methods to become specializations of `Number`.

Wow, and I am reusing much of the code I write in the `Number` class in the subclasses, right?

Right!

Just what is inherited?

Basically, all the methods and variables of a superclass are available to its subclasses.

How about an example?

Gladly. Explore the `Number` hierarchy, and find out which methods used in `Integer` and `Float` operations are actually inherited from `Number` or its superclass `Magnitude`. Also, look for the methods from `Number` that have been overridden in `Float` and `Integer`.

Overriding inheritance? What is that about?

At times, you will be working with a subclass and will realize that you are inheriting something that you do not want. If you are inheriting variables that are of no use to your subclass, you can simply ignore them. If this happens often or if there are many variables that you do not wish to inherit, then redesign the superclass-subclass relationship and rebuild the classes. If you are inheriting methods that you wish to use in a different manner in your subclass, then override them by implementing a method with the same message selector in the subclass.

JIM

OBJECTIVE WIZARD

If you wish to call an overridden method, you can use the `super` pseudo variable, and the superclass version of the method will be used. The `super` is used in your code by placing it before the message selector that you are sending.

I don't understand that part about using `super` to call overridden methods. Could you explain it to me again?

Of course. The pseudo variable `super` works just like `self`, except that it will look only for methods that you inherit. By skipping your class's so-called personal methods, it is able to call a method that has been overridden. This is the *only* reason that `super` exists.

It is very rare to use `super` other than within the method that is overriding what you are trying to call. For example, the following method uses the overridden method to do most of its work:

```
initialize
   "Private - Initialize my
    instance variables."
super initialize.
z := 0.
```

This method first calls the method it is overriding and then initializes an extra instance variable.

How about an example of inheritance?

All right, but I want you to do some of the work. Pick an object for us to work with.

How about a book?

That will be fine. We can use the point of view of a librarian trying to keep track of the books in a library.

Like the Objective Librarian?

Yes, I suppose you can think of the Objective Librarian. What things about a book would she want to track, given her somewhat idiosyncratic interests?

JIM

I wish she were here. Let's see…title and author, for sure. Maybe publication date, since she's interested in history and stuff. And probably that number used to reference books.

Dewey decimal, definitely.

Yes, I always want to know how long a book is.

Hmm…earlier you said something about creating general solutions. Is this a good time to do that?

Sure, libraries have lots of inventory besides books: magazines, reports, tapes…

OBJECTIVE WIZARD

The Dewey decimal or Library of Congress classification. That number is often crucial for librarians. Why don't you choose one type of cataloguing?

I would like to add the number of pages as well.

Here is our conceptual definition so far:

Book (Subclass of Object)
  title
  author
  deweyNumber
  pageCount

What is our next step?

A very good time. Because a library is our intended audience, I think a general class called `Document` would be in order.

Good! For this example, let us say that a `Document` has a `title` and a `deweyNumber`. Now, our design looks like this notation:

JIM

OBJECTIVE WIZARD

```
Document (Subclass of Object)
   title
   deweyNumber

Book (Subclass of Document)
   author
   pageCount
```

Note that I included the superclass in parentheses after the class name. You will find such references very important when you start designing.

I believe I understand inheritance now, but it would help if you ran through all the concepts one more time.

I can do that.

Encapsulation protects the instance variables of an object from access by the outside world. An object can be accessed only through its message interface. This saves you from worrying about some foreign code altering your object's instance variables.

Polymorphism is the ability of different objects to react differently to the same message. You need to know about this property so you can take advantage of it when you are designing or writing code.

Inheritance is a programming tool that lets you implement general solutions to a problem in a class and then specialize that solution by creating subclasses. There are many examples of this capability in the Squeak system. Inheritance applies to both variables and methods, but only methods can be overridden.

Wow, these are some really powerful concepts!

Yes, I suggest you learn them as powerful, new concepts and not simply try to fit them into your existing conceptual framework. If you want to use a truly object-oriented programming

JIM

OBJECTIVE WIZARD

language, you must work with these ideas. They are very much the essence of the paradigm. *Do not* try to learn them by thinking of polymorphism as being "like operator overloading" or methods as being "like procedures." Open up a new space in your imagination, and learn these concepts as something completely new.

Do not throw away your old paradigm…just learn to live with two paradigms. You will benefit from both.

I'm still not sure what it means to program in Smalltalk.

That is because I have not yet asked you to do any actual programming in Smalltalk. You have written Smalltalk code and you have created some classes, but you have not yet gone through the full programming process.

But you will do so soon in forthcoming discussions with me. Good Luck! Auf Wiedersehen.

Summary

New Terms

Encapsulation	Polymorphism
Inheritance	Overriding Inheritance

What Did You Learn?

- What encapsulation is and what it means to you as a programmer.
- What polymorphism is and how to find occurrences of it in the class hierarchy.
- What inheritance is and how to identify it in the class hierarchy.
- What is inherited.
- How to override methods.

Words of Wisdom

Create your subclasses so they make sense. If you can't say with confidence that your subclass is a kind of its superclass, don't use it.

Good inheritance trees are bushy. It's easy to go too far when making subclasses.

Good inheritance schemes rarely go more than four or five classes down from class `Object`.

To Do List

You are a researcher at the Captain Video Memorial Robotics Lab, and your task is to model the behavior of three robot designs. Each design employs a significantly different locomotive strategy.

Robot 1 is a human-like critter that walks on two legs. It changes direction by activating a shape memory alloy muscle. It changes velocity by changing the step rate of its gait.

Robot 2 is a monopod that hops about on one leg and changes direction by pointing its one leg to a different spot when it prepares for landing. It changes velocity by changing its landing angle.

Robot 3 is an amorphous creature that uses a pseudopodic mechanism for locomotion. It changes direction by pumping a fluid into a segment of its pseudopod. It changes velocity by changing the fluid volume in the lead portion of the pseudopod.

Implement each robot design in terms of class definitions and message interfaces. You do not have to implement the method code, but you do have to write a comment describing what each method will do after it is implemented.

When designing your class hierarchy, think in terms of how you can use inheritance to increase code reusability.

When designing your message interfaces, think in terms of behavior. Use polymorphism to facilitate extensibility.