Chapter 1 - Object-oriented programming - essential concepts

Overview

In this chapter, we will first show what Smalltalk looks like and then introduce the essential concepts of object-oriented problem solving and illustrate them on several examples. Some of these examples are real-world situations, others are taken from Smalltalk itself.

The principle of object-oriented problem solving is the insight that many problems are best approached by constructing models of real-world situations. The basis of these models are interacting objects with well-defined properties and behaviors. Solving a problem using the object-oriented approach thus consists of identifying appropriate objects and describing the functions that they must be able to perform and the information that they must hold. A computer application can then be constructed by converting such a description into a programming language. Programming languages that provide facilities for constructing such descriptions are called object-oriented and Smalltalk is one their prime examples.

1.1 Introduction

Since you are probably eager to start writing and executing programs, we will begin with a few examples of Smalltalk code.

Example 1: Arithmetic operations

The line

```
(15 * 19) + (37 squared)
```

is a typical Smalltalk arithmetic expression and you can see that it multiplies two numbers and adds the result to the square of a third number. To test this code, start VisualWorks Smalltalk using the instructions given in your User's Guide[1]. You will get a screen containing the VisualWorks launcher window shown in Figure 1.1. Click the Workspace button and VisualWorks will open the window shown on the left in Figure 1.2.

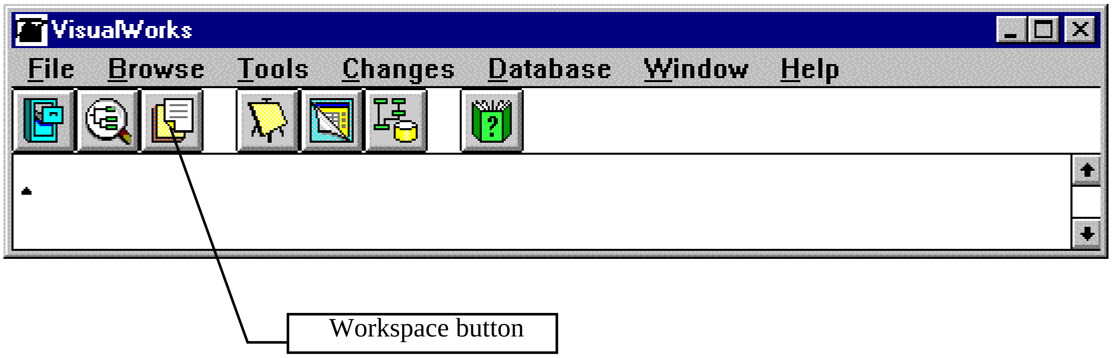
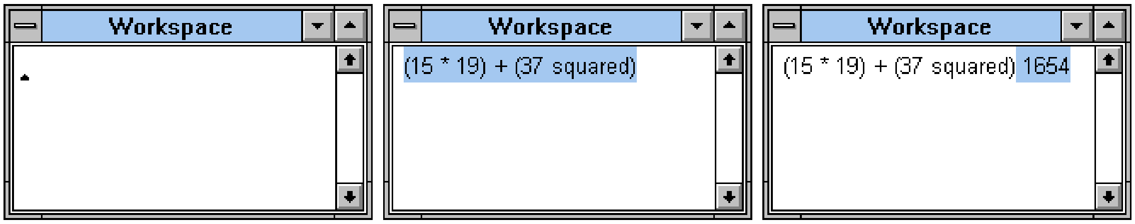


Figure 1.1. VisualWorks launcher window.

Enter the example code into the Workspace as if you were using a word processor as follows: Click the left mouse button inside the Workspace and enter the text making sure that it looks exactly like our example. Then 'select' the text as you would in a word processor Smalltalk provides several selection

[1] All our illustrations use VisualWorks Smalltalk. Other Smalltalk dialects have different user interfaces and their extended libraries are different. Most of the features covered in this book however apply.

shortcuts. For example, clicking twice at the beginning or at the end of the text ‘view’ of a Smalltalk selects all text in the view, and clicking twice at the beginning or at the end of a line selects the whole line.: Press the left button just before the start of the text and drag the cursor to the right across the text, releasing the button at the end. The text is now highlighted as in the center of Figure 1.2. Finally, press and hold down the second mouse button from the left and click the *print it* command in the displayed pop up menu[2]. Smalltalk will execute the code and print the result in the Workspace as in the right window in Figure 1.2.



```
(15 * 19) + (37 squared)
```

```
(15 * 19) + (37 squared) 1654
```

Figure 1.2. Workspace window: initial state (left), with selected text of Example 1 (middle), and displaying the result (right).

Example 2: Comparison of results of numeric expressions

The line

```
(1327 squared) < (153 * 20000)
```

is a typical test to determine whether a comparison of two expressions gives a yes or a no answer. Type it into the Workspace, select it[3], and execute it with *print it*. The answer displayed in the Workspace will be either true or false because Smalltalk treats a comparison as a question: ‘Is it true that ...?’

Example 3: String comparison

The line

```
‘abc’ < ‘xyz’
```

is a typical string comparison that determines whether the string on the left precedes the string on the right in alphabetical ordering. Guess what the result should be and test whether you guessed right.

Example 4. A string operation

As you are beginning to see, Smalltalk is designed to be easy to read. In fact, one of the original goals of Smalltalk designers was to create a programming language that even children could use. What do you think is the result of the following expression?

```
‘Smalltalk’ asUppercase
```

Test whether you guessed right.

Example 5. Simple output

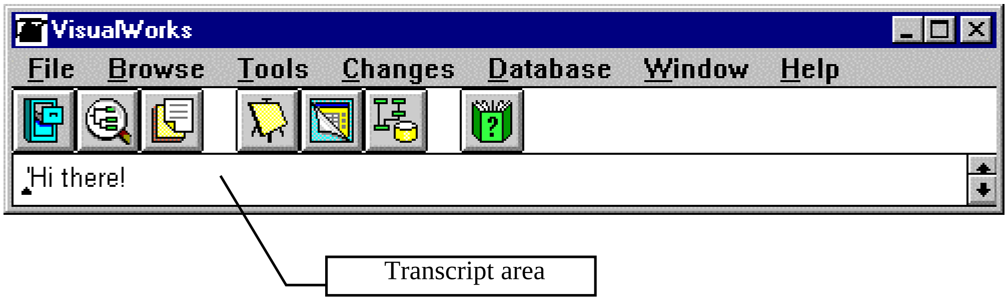
[2] Since the leftmost button of the mouse is used to make selections, Smalltalk programmers call it the <select> button. The next button displays a popup menu with operation commands such as print it and it is thus called the <operate> button.

[3] Smalltalk provides several selection shortcuts. For example, clicking twice at the beginning or at the end of the text view selects all text in the view, and clicking twice at the beginning or at the end of a line selects the whole line.

All Smalltalk code consists of ‘messages’ to ‘objects’ and this is why it is called object-oriented. Some of the messages used above include squared, factorial, and < and the objects include 27, 13, and ‘abc’. The Smalltalk library includes thousands of messages and you can easily create any number of your own, but all of them have one of three possible structures. You have seen two types of messages above (messages consisting of a single word such as squared, and messages consisting of a special symbol such as <), and this example introduces the only kind of message available in Smalltalk that you have not yet encountered. This kind of message is distinguished by the fact that its name is followed by a colon and an ‘argument’:

```
Transcript show: ‘Hi there!’
```

prints the text Hi there in the Transcript, the text area at the bottom of the launcher window (Figure 1.3).



Hi there!

Figure 1.3. Result of evaluating Transcript show: ‘Hi there!’ with *print it*.

Example 6. A more complicated comparison

Expression

```
567835 between: (27 squared) and: (13 factorial)
```

tests whether 567835 is between the values of 27 squared and 13 factorial. Guess the result and test whether the expression ‘returns’ true or false. It shows a messages of the third kind but this one has two word-colon pairs (called ‘keyords’) instead of one. There is no practical limit on the number of keywords that a message may have.

To summarize our brief experience, *Smalltalk treats everything as messages sent to objects* and its interpretation of expressions such as

```
15 squared
```

is as follows: *Take the Number object* 15 *and send it the message* squared. All number objects understand the message squared and when object 15 gets the message, it calculates its square and returns this Number object as the answer. Similarly, when Smalltalk executes

```
‘Smalltalk’ asUppercase
```

it treats the text string ‘Smalltalk’ as a string object, and asUppercase as a message to it, and returns the string object ‘SMALLTALK’.

The more complicated expression

```
(15 * 19) + (37 squared)
```

is executed in several steps. First, Smalltalk asks object 15 to execute message * with argument 19. This returns object 285, the product of 15 and 19. The original code has now, in effect, been reduced to

```
285 + (37 squared)
```

Smalltalk now asks 37 to execute message squared. This returns object 1369 and the code now effectively becomes

```
285 + 1369
```

Smalltalk now asks object 285 to execute message + with argument 1369. This returns the final result 1654. Note that this works because *all Smalltalk messages return objects*, and messages can thus be combined.

The examples that you have just seen cover all possible forms of Smalltalk messages and if they give you the impression that Smalltalk is essentially simple, you are quite right - Smalltalk is based on very few very powerful ideas. Unfortunately, the ease of reading and writing pieces of Smalltalk programs does not mean that writing significant Smalltalk programs such as a word processors or spreadsheets is very easy. Difficult task are difficult even in Smalltalk - but not as difficult as in most other languages as you will see later.

At this point, you would probably like to proceed to other Smalltalk programs, and if you really cannot resist it, you can skip to Chapter 3 which starts our discussion of Smalltalk. However, the ideas of an object and a message have some non-obvious implications and properties and we strongly suggest that you now complete this chapter to find out about some very important general concepts of object-oriented programming.

It will probably not surprise you if we also suggest that you then continue with Chapter 2 which is dedicated to the principles of finding the right kinds of objects for your application. The reason why this is important is that developing object-oriented applications requires that you first identify the right objects for your task and *then* write the code describing them in your programming language. Programming in Smalltalk thus requires two skills - familiarity with Smalltalk, and ability to find the objects that can be combined to solve your problem. Both of these skills are essential: If you know how to use Smalltalk but cannot design the proper objects for your application, you cannot even start writing programs. And if you can design but don't know Smalltalk, you obviously cannot convert your paper design into working Smalltalk code. This means that we now have a choice: Explain the Smalltalk language first and object-oriented design next, or vice versa. We decided to start with principles and design because learning about design greatly improves understanding of Smalltalk and contributes to Smalltalk skills.

Main lessons learned:

- Smalltalk is a very readable language in which all code is interpreted as messages to objects. Languages based on this principle are called object-oriented.
- Although the object-oriented principle is very simple, it has some subtle implications and consequences that must be understood before one can start writing significant programs.
- Program development requires two skills: the knowledge of a programming language and the ability to find suitable objects that can be used to solve the problem.
- The Visual Launcher is the opening window in VisualWorks Smalltalk. All other tools can be accessed from it.
- The left mouse button is called the <select> button. Use it to select text and other items. The button next to it is called the <operate> button and pressing it opens a pop up menu with commands.

Exercises

1. Execute the examples from this section in Smalltalk and explore the environment on your own.

1.2 What is object-oriented problem solving?

When you think about a problem such as planting a garden, you think of objects (a spade, a wheelbarrow, a watering can, and various kinds of plants) and the things that they can do (a spade can dig a hole, a watering can can be filled and emptied, and plants can be planted). The principle of object-oriented problem solving is the same: Think of solutions in terms of the required objects and their functionality, and the relation between all objects that participate in the solution. When put together, the program then works as follows: To perform a task, send a message to an appropriate object which directly executes those operations that it can handle, and sends messages requesting help to other objects for operations that are beyond its area of expertise. In this and the following sections, we will now illustrate this concept on several simple examples taken from programming and non-programming situations.

World 1: Airport

Consider an airport with travelers, arriving and departing planes, and the necessary ground support. One of the possible situations that we might wish to handle can be described by the following scenario:

Scenario: Getting a taxi

Mr. Brittle’s airplane is landing at Terminal 3B of the International Airport in Halifax and he wants to book a taxi to take him to the downtown. He turns on his laptop computer and negotiates the taxi with a taxi dispatcher via an e-mail conversation along the following lines:

Conversation:

1. Mr. Brittle - message to dispatcher: ‘My plane is landing and I need a taxi at Terminal 3B. My name is Brittle’.
2. Dispatcher - message to a free taxi: ‘Please go to Terminal 3B and wait for Mr. Brittle’.
3. Taxi driver - replies to dispatcher: ‘I am on my way’.
4. Dispatcher - replies to Mr. Brittle: ‘The taxi is on its way’.

The flow of the conversation is illustrated graphically in Figure 1.3. In this diagram, time is running from the top down, columns are labeled with names of participants, full lines with arrows indicate who is communicating with whom and what the communication is about, dotted lines represent confirmations of message completion.

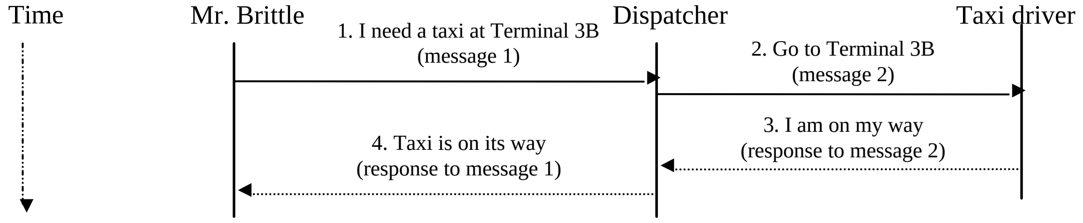


Figure 1.3. Graphical representation of Scenario 1.

The ‘objects’ participating in this scenario and their communication are shown in Figure 1.4. They include Mr. Brittle, the dispatcher, and the taxi driver, and the conversation involves two messages (one from Mr. Brittle to the dispatcher and one from the dispatcher to the driver) and confirmation of their successful execution.

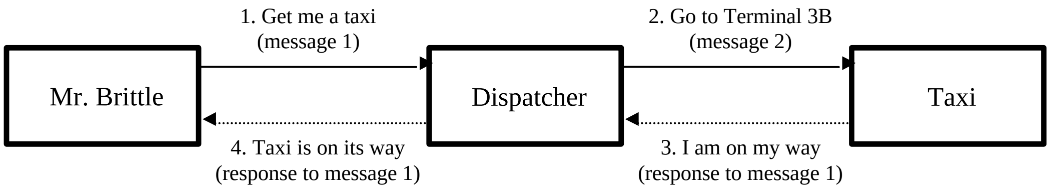


Figure 1.4. Another representation of Conversation 1.

The real airport is, of course, more complex and many other scenarios and conversations are possible. As an example, the dispatcher might inform Mr. Brittle that there is no taxi, the taxi driver might tell the dispatcher that the car is broken, access ramp of Terminal 3B might be closed, and so on. Our script is only one example presented to illustrate the nature of object-oriented thinking and the remaining examples are presented in the same spirit.

World 2: Household management

This example shows how I could deal with a little emergency in my household.

Scenario: Taking care of unexpected visitor

My sister Monica has unexpectedly arrived with her two children and I don’t have anything to offer them. Monica likes roses and chocolate cake, her children like hamburgers. Having no time to go to a store, I call a catering company. The conversation might go as follows.

Conversation:

1. I - message to caterer: ‘Please deliver a bunch of roses, a chocolate cake, and two hamburgers’.
2. Caterer to florist: ‘Please deliver a bunch of roses’.
3. Florist to caterer: Delivers roses to caterer.
4. Caterer to MacDonald’s: ‘Please deliver two hamburgers’.
5. MacDonald’s to caterer: Delivers two hamburgers to caterer.
6. Caterer to Tim Horton’s: ‘Please deliver a chocolate cake’.
7. Tim Horton’s to caterer: Delivers chocolate cake to caterer.
8. Caterer to me: Delivers rose, cake, and hamburgers to me.

The object-oriented view of this situation (Figure 1.5) has five participants, or actors: me, the caterer, the florist, MacDonald’s, and Tim Horton’s. Communications 1, 2, 4, and 6 are messages, communications 3, 5, 7, and 8 confirm their execution and deliver the requested objects. The responsibilities of participating actors - the services that they are expected to be able to perform - include the ability to satisfy requests for party items (caterer), flowers (florist), and so on.

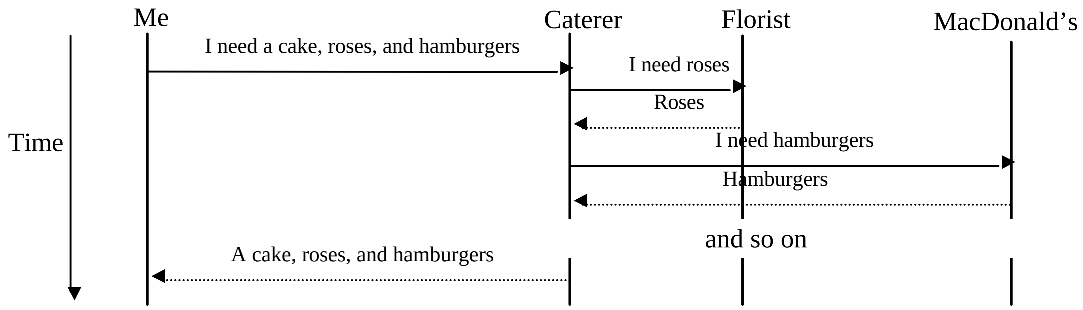


Figure 1.5. Partial script of Scenario 2.

We can now make several conclusions concerning the object-based view of problems:

- Problems can be solved by asking specialized objects to perform tasks within the scope of their expertise.
- To ask an object to perform an operation, send it a message.
- Even complex tasks can be executed by collaboration among specialized objects.
- When an object receives a message, it executes those parts of the task that it can handle and sends messages to other objects to accomplish those tasks that are outside its area of expertise.
- An object’s response to a message can be formulated as follows: Every message returns an object. The returned object may be a physical object such as a hamburger, or just a piece of information such as computer data or a confirmation that a message has been completed.

Executing a task by asking another object to execute it is called *delegation*. In our example, the caterer delegates the task of procuring roses to the florist - the florist is the caterer’s *collaborator*. The chain of messages delegating sub-tasks to collaborators may be several levels deep.

World 3: Microwave oven control

My microwave oven provides many functions performed by using the panel in Figure 1.6. I will now describe one of its uses.

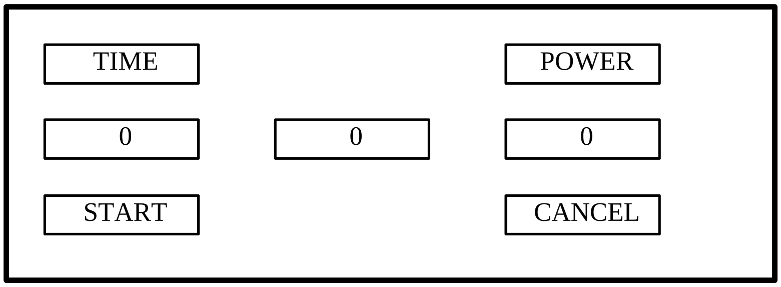


Figure 1.6. Microwave oven panel for World 3.

Scenario: Cooking porridge

Every morning, I eat a bowl of oatmeal porridge. (I don’t get any commission for mentioning it in this book.) I cook it in my microwave oven according to the following procedure that I will call a ‘conversation’ to use a term used in the previous two examples:

Conversation:

1. I press the *Power* button, which can be interpreted as sending message 'Initiate cooking time setting function' to the oven.
2. *Power* button message to oven electronics: 'Prepare to read length of cooking time'.
3. Electronics to display: 'Show message for setting time'.
4. Display performs the task and confirms: 'Done' to electronics.
5. Electronics confirms to *Power* button: 'Done'.
6. *Power* button to me: 'Message executed'. (I don't yet have a speaking oven and the button does not really tell me anything but the display indicates that my start-up 'message' has been completed.)
7. I press the *1* button, effectively sending the message 'Set cooking time to 1 minute'.
8. Button to electronics: 'Set time to 1 minute'.
9. Electronics to itself: 'Set set cooking time to 1 minute'.

and so on.

This example shows how the idea of objects and messages extends to inanimate objects. We find the idea of objects and messages is very natural and so did the designers of the first object-oriented language (Smalltalk). Any object-oriented programming language thus provides means for defining conceptualized objects, usually models of real world objects or concepts, and messages and for requesting operations and results from them. A pure object-oriented language such as Smalltalk does not provide any other means of control beyond objects and messages to create programs and it turns out that nothing else is needed.

Main lessons learned:

- Object-oriented problem solving means treating problems as mini worlds consisting of objects that understand messages and respond to them by performing tasks.
- By an object, we mean an equivalent of a black box that can respond to a predetermined set of messages. An object is often a simplified model of a real-world entity or concept.
- A message invokes an action that often generates a series of further messages to other objects.
- Every action in an object-oriented world is the result of sending a message to an object.
- A message has a sender and a receiver.
- The responsibilities of an object are the services that it is expected to be able to fulfill.
- In executing a message, an object may send a message to another object (its collaborator).
- Letting another object handle a part of a task is called delegation.
- When an object executes a message, it returns the result object to the sender of the message. This object may be just a confirmation that the message was completed.
- A scenario is a task encountered in a given problem area.
- A conversation is a sequence of message exchanges required to realize a scenario.

Exercises

1. Create an additional detailed scenario for each of the problem worlds presented in this section. Add new objects if necessary.
2. Complete the microwave conversation and draw its diagram.
3. Find objects needed to describe a basic version of the following problem worlds and list some of their responsibilities. Formulate at least one scenario for each problem world, give details of the corresponding conversation, and represent it graphically.
   a. Checkout counter in a store.
   b. Gas station.
   c. Ice cream stand.
4. Execute the conversations presented in this section and the exercises above by assigning one person to each object. Each person-object gets an index card with the name of the assigned object on it. The holder of each card lists all the responsibilities of the assigned object on the card.

1.3 Examples of objects in computer applications

The situations described in the previous section illustrate the concept of objects and messages in real-world settings. We will now give several more examples to illustrate the object-oriented paradigm in computer settings.

World 4. A computerized library catalog

In this example, we will briefly shift our attention to the problem of *finding* the objects involved in a model rather than *using* them. We will not present any usage scenarios.

Problem: Our local library asked us to write a computer program implementing a simple library catalog. The program must keep track of borrowers (their names, addresses, telephone numbers, and borrowed books), and books (authors, title, publisher, library number, location, available or on loan). The program must allow the staff to add new borrowers, sign books out, search for books, and execute other scenarios.

Solution outline: To construct this program, we must first decide which objects are needed and what functionality they have. To do this, we formulate several scenarios and determine their actors, and combine this information with our general understanding of the problem. In this problem, we will limit our investigation to a simple analysis because the situation is familiar.

One of the obvious objects that we need is a computer representation of a *borrower*. The functionality of this object is essentially equivalent to the representation of a borrower that an old-fashioned library might store on catalog cards. For the purpose of the library, the borrower's functionality must include the ability to respond to messages such as 'Set name to …', 'Set address to …', 'What is your name?' and 'What is your borrower code?', 'Change your address to ...' and 'Change your borrower code to ...' corresponding to the ability of an index card to record and display information and to allow its change. In our context, we don't care what is the borrower's weight, color of hair, and place of birth although these characteristics are important features of physical borrowers and might be essential in another problem such as a police registry.

Another object required by our program is a model of a *book* and it includes those properties of a book that are relevant for a library catalog. We would expect the library catalog model of a book to include its author, title, library code, and on-loan information, but not the table of contents of the book, the text, and the pictures. The functionality of a model book object includes the ability to initialize and modify its data, and to answer messages such as 'What is your title?' and 'Are you in the library?', as well as 'Change your status to 'on loan' and so on.

We also need a *book catalog* object. This object is a collection of book information objects, being able to add and delete books, and search for books given the name of an author or title. We also we need a *catalog of borrowers* with similar functionality.

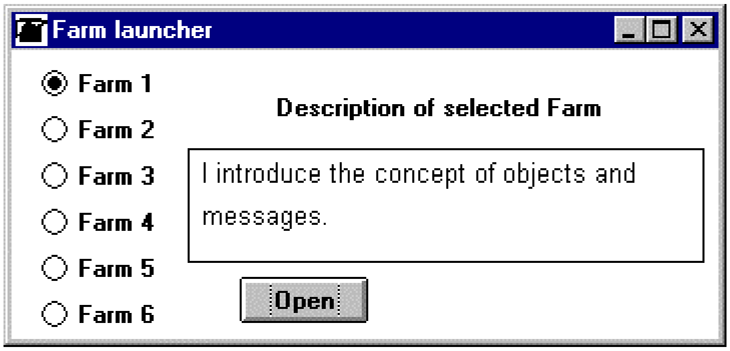
One notable feature of our solution is that although our computer models real-world objects, our model objects represent only their selected aspects. Computer model objects are not carbon copies of real-world objects.

World 5: Farm - a simulated animal world

This example revolves around several versions of our computer program called *Farm* that we designed to illustrate the main concepts of the object-oriented paradigm and to introduce Smalltalk programs. To use it, you must first open a *Farm Launcher* (Figure 1.7) by typing

```
Farm open
```

into a Workspace and executing this text with the *do it* command from the pop up menu.

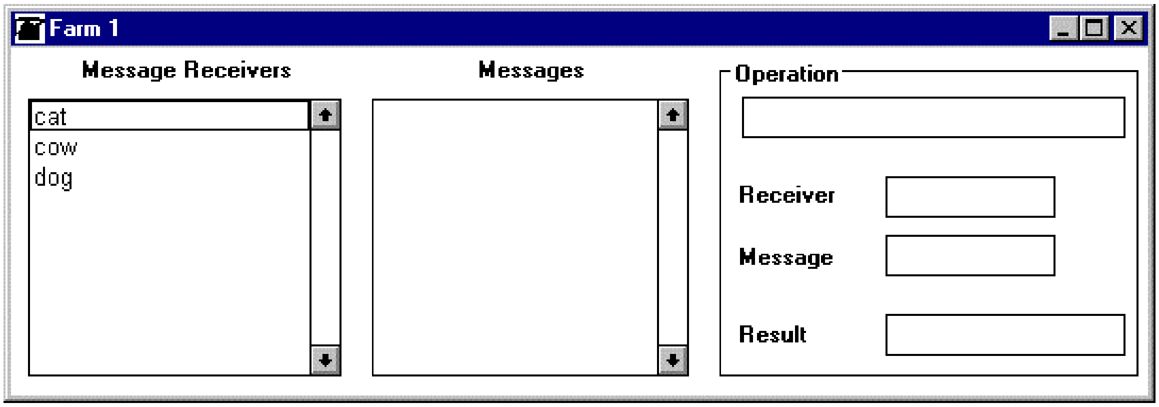


● Farm 1
○ Farm 2
○ Farm 3
○ Farm 4
○ Farm 5
○ Farm 6

I introduce the concept of objects and messages.

Figure 1.7. The opening window of *Farm Launcher*. Use it to select the first version of the *Farm* program.

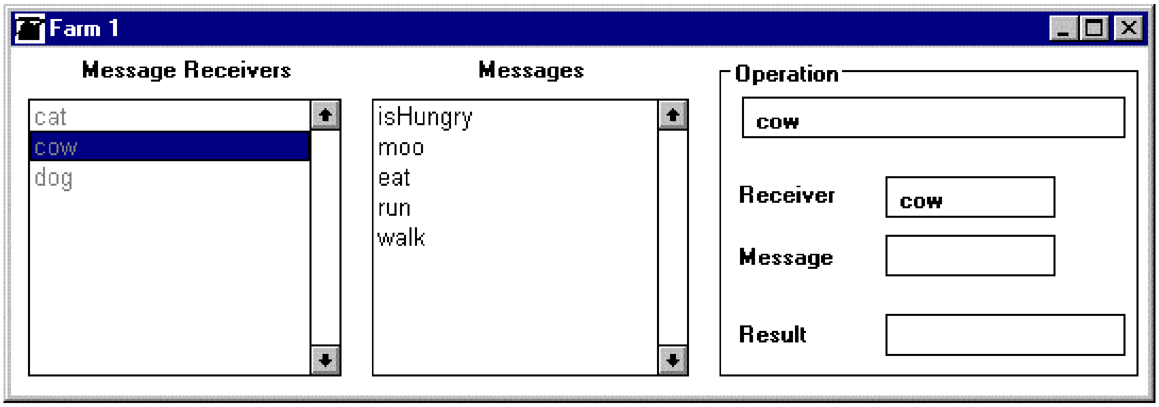
As you see, the *Farm* program has six different versions and we will naturally start with *Farm 1*. When you select *Farm 1* as in Figure 1.7 and click the *Open* button, you will get the window in Figure 1.8 showing all animals living on this farm. As you can see, the farm is rather poor but we will show you in a moment how to add more animals.



cat
cow
dog

Figure 1.8. Initial state of Farm 1.

The animals on the farm are, of course, objects that understand messages such as run or moo. To execute a task, select an animal in the list on the left. This will display a list of the messages that the animal understands in the list in the middle as in Figure 1.9.



cat
cow
dog

isHungry
moo
eat
run
walk

Operation: cow

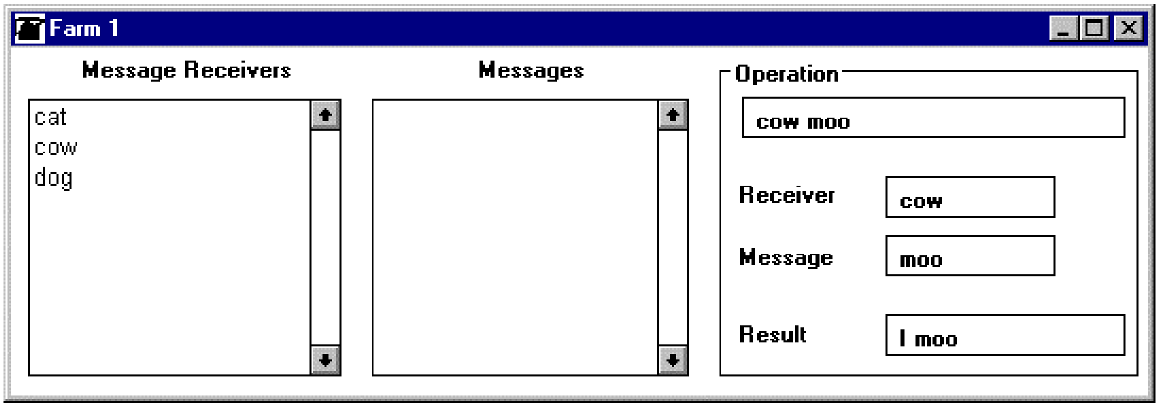
Receiver: cow

Figure 1.9. Farm 1 after selecting an animal.

When you now click a message, the program sends the message to the animal and shows a summary of the operation as in Figure 1.10 which shows that we executed expression

```
cow moo
```

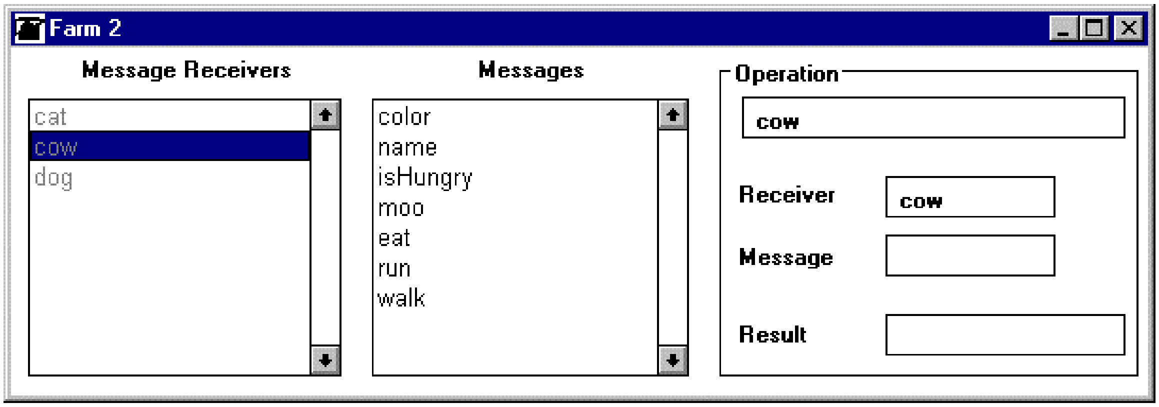
whose receiver object is cow and whose message is moo, and that the result of the execution of the message –the cow's response - is 'I moo'.



Operation: cow moo
Receiver: cow
Message: moo
Result: I moo

Figure 1.10. Result of sending message moo to object cow.

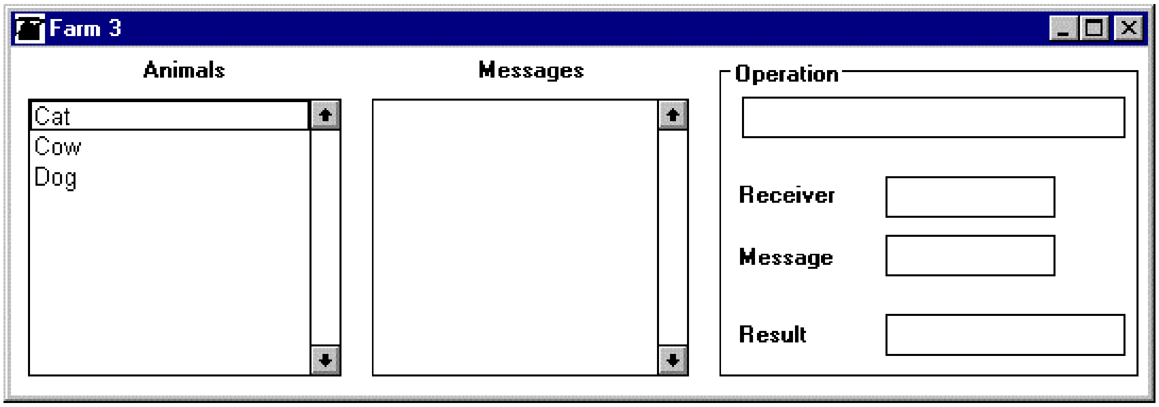
Farm 1 gave us a hands on experience with objects and showed that objects understand messages. Obviously, objects also have properties and *Farm 2* provides access to an extended set of messages that let you ask animals about their properties. Figure 1.11 shows you what these expanded message sets look like. A cow, for example, has a color and a name and it understands messages requesting these properties. We will leave it to you to try them out.



Operation: cow
Receiver: cow
Message:
Result:

Figure 1.11. *Farm 2* animals have properties including name and color.

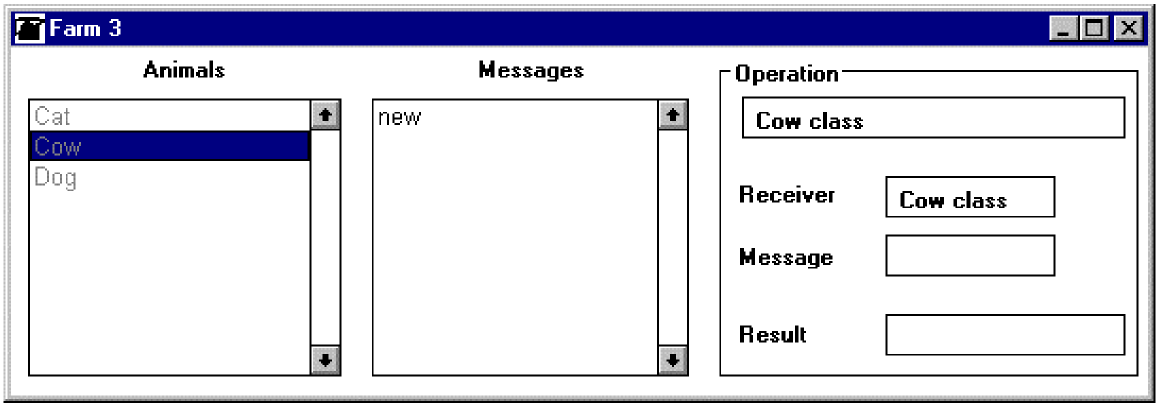
We promised that you will be to create new animals and *Farm 3* makes it possible. Its idea is somewhat different from *Farm 1* and *Farm 2* in that its window does not initially show any animals but rather 'animal factories' represented by the word Cat, Cow, and Dog (Figure 1.12).



Cat
Cow
Dog

Figure 1.12. *Farm 3* adds animal factories.

Cow factories are, of course, quite different kinds of objects than cows. Cow factories don't understand messages such as moo or eat because they are not animals. Since animal factories can only make new animals, they only understand message new as you can see when you select Cow as in Figure 1.13.



Cat
Cow
Dog

new

Operation: Cow class

Receiver: Cow class

Figure 1.13. *Farm 3* introduces the concept of classes - object producing factories.

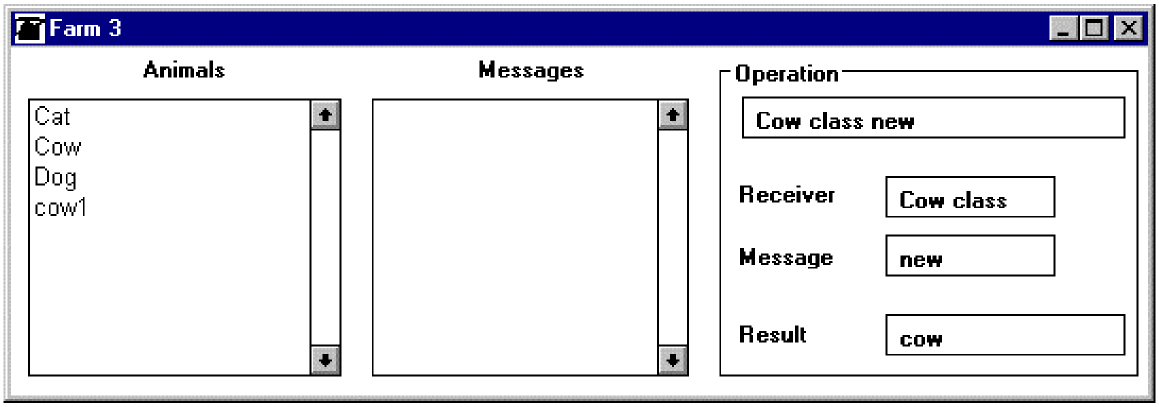
The distinction between objects and the factories that make them is natural. We are familiar with car factories (and they are a different kind of objects than cars), paint factories (and they are different from paints), and so on ,and we can easily extrapolate our experience that objects must be produced to imagine that when a program wants to create for example a new window, it needs a window factory to do that, and that when it wants a rectangle, it must use a rectangle factory to create one. Object-oriented languages thus include object factories but instead of calling them factories, they call them *classes*[4]. We have shown this in our program which displays the name of the receiver as *Cow class*, for example, instead of *Cow factory* (right-hand side of Figure 1.13).

When you send the new message to an animal factory, the program requests information about the animal (its name and color) and then adds the animal to the farm (Figure 1.14). Since we can add as many animals as we want, the programs gives each animal a number so that we can distinguish among them. In our example, we now have four objects: a Cat factory, a Cow factory, a Dog factory, and a cow displayed in the list as cow1[5]. The first three objects understand only message new (and create a corresponding animal),

[4] We will see later that classes can serve other functions beyond creating objects, but creating objects is their most common function.

[5] Following Smalltalk conventions, we distinguish instances from classes by deriving instance names from class names but starting class names such as Cow with a capital, and instance names such as cow1 with a lower case letter.

the fourth is a cow with a name and a color and understands the same cow messages as in *Farm 2*. To distinguish the products from the factories, the products are called *instances* and our current farm thus has three classes (Cat, Cow, and Dog), and one instance of class Cow shown as cow1.



Cat
Cow
Dog
cow1

Operation: Cow class new

Receiver: Cow class

Message: new

Result: cow

Figure 1.14. Farm 3 after creating a cow.

After playing with the farm and introducing the important concepts of class, instance, and object properties, let's now explore which other objects were needed to build the farm program itself.

The obvious objects required by *Farm* are classes Cat, Cow, and Dog. These objects are what is called *domain objects* and they describe the world that we are modeling – in this case a farm. But what about the instances of Cat and Cow and Dog, are they separate from the classes? The approach taken by Smalltalk is that a class is a blueprint for making instances and it thus knows how to make its instance. As a consequence, in addition to its own description a class also has a blueprint for creating its instances. As an example, class Cat has a blueprint for making cats and class Cow has a blueprint for making cows. Instance descriptions are thus always included in the definition of its class and we thus don't need separate objects for classes and their instances. We conclude that the domain objects for the Farm program are Cat, Cow, and Dog.

In addition to domain objects, we also need objects in charge of the user interface - objects that implement windows and their components such as buttons, labels, lists, and other 'widgets'. The window and the widgets must know how to display themselves on the screen and how to respond to user control such as clicking pressing a mouse button.

In addition to these rather obvious necessities, we also need a number of other objects to work in the background. As an example, we need an object "watching" the keyboard and the mouse, intercepting all keyboard and mouse events and sending them to our application. Since the screen may display any number of Smalltalk windows in addition to *Farm*, we also need an object that decides which window is currently active and should be responding to these 'input events'. And then we need mundane objects such as numbers, and strings of characters, and objects that hold collections of other objects, and so on.

The *Farm* is now beginning to look very complicated and if you looked behind the scenes and checked how many objects are active when *Farm* is running, you would find hundreds or thousands of them and probably around a hundred classes defining them! Does this mean that if we want to write a program such as *Farm*, we must invent hundreds of objects? Fortunately not. First, many of these objects are instances of the same class. As an example, the *Farm 3* window contains two list widgets (both instances of the same class), five labels (all instances of class Label), and so on. We conclude that although a running program requires many objects, the number of classes required to manufacture them is much smaller.

Moreover, we do not have to create all the classes that are involved in *Farm* because many other applications have similar needs and many generally useful objects such as buttons, windows, input sensors, and window schedulers have been created by the designers of Smalltalk and stored in its enormous *library* of prebuilt objects. To create *Farm*, we thus needed to design and implement only the specialized classes such as Cat, Cow, and Dog, and reuse those objects that are already in the library. We will, of course, store our new objects in the library too and if we come across another application that needs them (such as an animated farm), we will reuse them as well. As we and other programmers develop new applications, the library will be getting more and more complete and if we designed our objects intelligently, fewer and fewer classes will have to be created from scratch.

Reuse of previously implemented objects is one of the greatest potential benefits of object-oriented environments such as Smalltalk. Reuse of existing blueprints is not, of course, an invention of object-oriented programming but an everyday fact of life. As an example, computers use power supplies, but manufacturers of computers don't have to draw blueprints of new power supplies because power supplies are also used in many other products and can be bought from a distributor. Similarly, if you need a screw, you don't need to invent one but rather buy it in a store and use it in whatever application you please.

In addition to minimizing the need to design new objects, reuse also contributes to quality. Commercially available components have presumably been carefully tested and are known to work well. VisualWorks Smalltalk takes the concept of reuse very seriously and its library contains more than 1,000 predefined and thoroughly tested reusable classes and many more can be bought from other software manufacturers.

A word of caution is in place. Although the experienced programmer gains much productivity from object libraries, a large library initially adds to the confusion of the beginner who finds the number of prebuilt components overwhelming. This is not a new problem either: The complete collection of all the tools of a professional carpenter is also enormous and incomprehensible to a handyman - but it is a prerequisite of high productivity and quality work for the professional.

World 6: Built-in Smalltalk objects

What kinds of objects would you expect to find in a programming environment? First of all, those that are needed to build the environment. Numbers, text, windows, buttons, labels, and pictures are some of the most obvious candidates and we will now briefly look at the most basic of them - numbers. This example is not a lesson on Smalltalk but only another illustration of the object-oriented paradigm.

Since computer hardware treats different kinds of numbers differently and since different kinds of numbers have different properties, most programming languages distinguish several kinds of numbers. In Smalltalk all numbers are, of course, objects and Smalltalk recognizes many kinds of numbers including the following ones:

- Large and small *integers*. Integers are numbers without a decimal point such as 15, -562, 0, and 4590100118912313108989887676465434111101. Smalltalk distinguishes small and large integers. 'Small' integers are those integers that your computer can handle directly, 'large' integers are those that Smalltalk must handle in a special way.
- *Floating-point* numbers. These are numbers with a decimal point, such as 156.43, -467.01, and 0.000012361 and computers handle them differently from integers. Smalltalk again distinguishes two kinds of floating-point numbers according to their range and number of valid digits (precision).
- *Fractions*. These are numbers represented by an integer numerator and an integer denominator such as 13/45 or -54/13. Computers don't have any instructions to deal with fractions and Smalltalk deals with them by using integer operations.
- *Complex numbers* are numbers with a real and an imaginary part. These objects are not part of the basic Smalltalk library and are included only in extensions; they are used mainly in engineering applications and they are not directly handled by hardware.
- *Fixed-point numbers* - similar to floating-point numbers but with a fixed number of digits behind the decimal point. Handled by Smalltalk, not directly by hardware.
- *Special numbers*, such as +∞ and -∞, more accurately the results of arithmetic operations that fall outside the legal range of floating-point values. These objects are included only in the extended Smalltalk library and some of them are handled by hardware.

If we want to treat numbers as objects, their functionality must allow us to ask number objects to do all the things that we normally do with numbers including arithmetic and calculation of mathematical functions. In this perspective, a calculation such as

```
3 + 5
```

must be treated as a message + to object 3 to do addition with object 5 as in

‘Object 3, do + with object 5 and return the resulting number object’

or, to put it differently, as

‘Object 3, tell me what is the result of doing + with object 5’

The functionality of number objects in Smalltalk thus includes the ability to respond to arithmetic messages such as + and -, the ability to calculate mathematical functions such as log or cos, the ability to compare themselves to other number objects (‘are you greater than 17?’), responding to questions such as ‘Are you an integer?’, and so on. This is, admittedly, an unusual but view of numbers but it is consistent with the OO view that everything is a number or a message.

Main lessons learned:

- Objects fall into two general categories – classes and instances.
- Classes can be thought of as factories and instances as their products.
- One of the main benefits of object orientation is its potential for reuse.
- Advantages of reuse include shorter application development time and better quality.

Exercises

1. Formulate a scenario for the library catalog world and expand it into a detailed conversation between the user and the program.
2. Experiment with the *Farm* program. As an example, explore how the combination of isHungry and eat messages works.
3. Another simulated microworld included in our software is a series of *Pen Worlds*. To open the program, execute PenWorld open using *do it*. Experiment with *Pen Worlds* in parallel with *Farm* worlds, using them to draw and erase colored lines and shapes on the screen. Formulate and solve tasks such as drawing overlapping and non-overlapping rectangles and other shapes in various colors. Pay attention to the object-message interface.
4. Formulate the following messages in terms of requests for information or action:
   3 squared
   pi convertedToDegrees
   3 / 5(creates a fraction)
   pi cos (calculates the cosine of π)

1.4 How does an object-oriented application work?

Although you are now comfortable with the principle of object-oriented software, you may be wondering how object-oriented programs work. Your question might be: ‘OK, so we have a pile of objects stored in the computer, but how do they come to life and cooperate to solve a problem?’ To answer this question, the following greatly simplified script describes the operation of the *Farm* program. We present the script in an anthropomorphic form, treating the objects as if they were living organisms or robots, a perspective that many people find useful when thinking about objects.

To start the *Farm*, the user enters and highlights the expression

```
Farm open
```

and executes it with the *do it* command. This sends the open message to Farm and Smalltalk starts by looking for the definition of the open message in Farm. It finds the definition and discovers that its operation starts by a request to the Window class object to create an instance of itself from a specification provided by the Farm class, and draw it and all its components on the screen. In executing this request, the window object constructs a detailed description of the frame of the window with a label and sends a message to each of its button, label, and other 'widget' objects, asking them to draw their representation and to get ready to respond to mouse clicks. When this is done, the window asks a mouse/keyboard input sensor object to start tracking the user's actions.

The input sensor object starts monitoring keyboard and mouse buttons and when the user clicks the mouse over, for example, the *Open* button, the input sensor passes this information to the button object, which then executes the function for which it is programmed. In this case, the message is to open a window for *Farm 1*. When the message is completely executed and the *Farm 1* window opens on the screen, the input sensor starts tracking the mouse again, waiting for another mouse click, and so on, until the user closes the *Farm 1* window and the Farm launcher and terminates the application.

We conclude that a typical application operates by a combination of user input events that trigger messages to objects which then create a chain of message interchanges between objects.

Main lessons learned:

- To start the execution of an object-oriented program, send a start-up message to an object designated by the application as the start-up object. This begins a sequence of message sends, usually involving interaction with the user.

1.5 Classes and their instances

If you wondered whether the concept of a class violates our claim that the Smalltalk environment is totally populated by objects, we can reassure you. A class is simply a special kind of object whose main role usually is to manufacture instances. The following table illustrates these notions and contrasts classes and their instances on the example of animals and animal factories:

class	its instance
(producer)	(product)
Cow	cow1
(can create new cows)	(can moo, eat, etc.)
Cat	cat3
(can create new cats)	(can meow, eat, etc.)

Although classes and their instances are related, they are quite different kinds of objects. In particular, classes understand one set of messages, and their instances understand a different set of messages. As an example, *class* Cow understands the new message that creates a cow, but it does not understand and cannot execute messages such as moo, eat, or run. *Instances* of class Cow, on the other hand, understand moo, eat, and run but don't understand new.

To avoid confusion between messages understood by classes and messages understood by their instances, Smalltalk distinguishes c*lass messages* (those understood by a class), and *instance messages* (those understood by instances). For example, message new is a class message, but message moo is an instance message. The following table contains a summary of class and instance messages for class Cow:

Cow *class* messages	Cow *instance* messages

new	color name isHungry moo eat run walk

We have already noted that the class definition defines the blueprint of its instances, and both class messages and instance messages are thus a part of the definition of the class. Class message are on the class side of the definition, instance messages are on the instance side.

As we have seen, if we want to create a new instance of a class we send a class message to the class asking it to create an instance. But where do classes come from? When Smalltalk programmers need a new kind of object, they must write a definition with detailed descriptions of the class and instance messages and add it to the Smalltalk library of classes. To do this, they use a special tool called the Browser. We will see what the Browser looks like in the next section.

Since Smalltalk is an object-oriented language, Smalltalk programming consists mainly of creating new classes and reusing or expanding the ones that already exists. The advantages of Smalltalk over other object-oriented environments include that Smalltalk makes creation of new class definitions very easy, that it contains many predefined classes, and that both built-in and new classes can be reused and modified with equal ease. This greatly increases programming productivity and improves program quality because large parts of new applications consist of previously created and exhaustively tested classes.

Main lesson learned:

- Object-oriented environments distinguish two kinds of objects - classes and instances.
- Classes are used mainly to create instances, instances are the workhorses which perform most of the work.
- The messages understood by a class are distinct from the messages understood by its instances. Messages understood by a class are called class messages, messages understood by its instances are called instance messages.
- The definition of a class contains the definition of both its class messages and its instance messages.

Exercises

1. As we mentioned, class messages are used mainly to create new instances. There are, however, many class methods that perform other functions such as initialization, opening of applications (as in Farm open), provide useful information and frequently needed operations, demonstrate how to use a class, and so on. As an example, execute the following class messages in the Workspace with *print it*:
   a. Time totalSeconds "Answer the total seconds since 1901 began"
   b. Date nameOfDay: 3 “Answer third day of the week.”
   c. Float pi “Answer value of π.”
   d. ScheduledWindow openNewIn: (100@100 corner: 200@300) “Open window with upper left corner at point (100, 100) and lower right corner at point (200, 300).”
   e. Float radiansPerDegree “Answer number of radians in one degree.”
   f. Window platformName “Return the name of the operating system.”
   g. Dialog warn: ‘Click OK to continue’ “Open a notification window.”
   h. Rectangle fromUser “Get a rectangle from the user.”
   i. GraphicsContext exampleArcs “Execute example program.”

1.6. A first look at Smalltalk classes

To view classes in the Smalltalk library and to add, delete, or modify classes, Smalltalk programmers use the *System Browser*. To open it, use the *Browse* command in the *Visual Launcher* or simply click the *Browser* button in the launcher window (Figure 1.15).

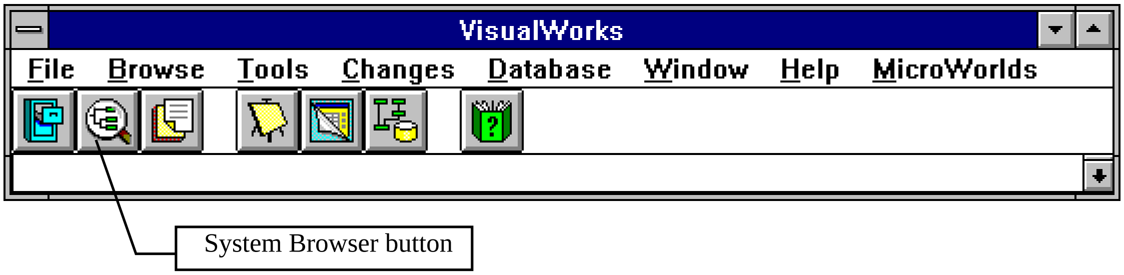


Figure 1.15. To open the System Browser, click the *Browser* button or the *Browse* command.

When the System Browser opens, it looks as in Figure 1.16. It contains four ‘views’ at the top, a text area at the bottom, and two buttons labeled *instance* and *class*; these buttons are used for selecting between class and instance methods. The leftmost top view is called the *category view* because it contains a list of *categories* of all classes in the class library Each category contains several related classes and each class is included in exactly one category. The purpose of categories is strictly organizational - they group together related classes to make it easier to find a class in the enormous library. Categories have nothing to do with class behavior.

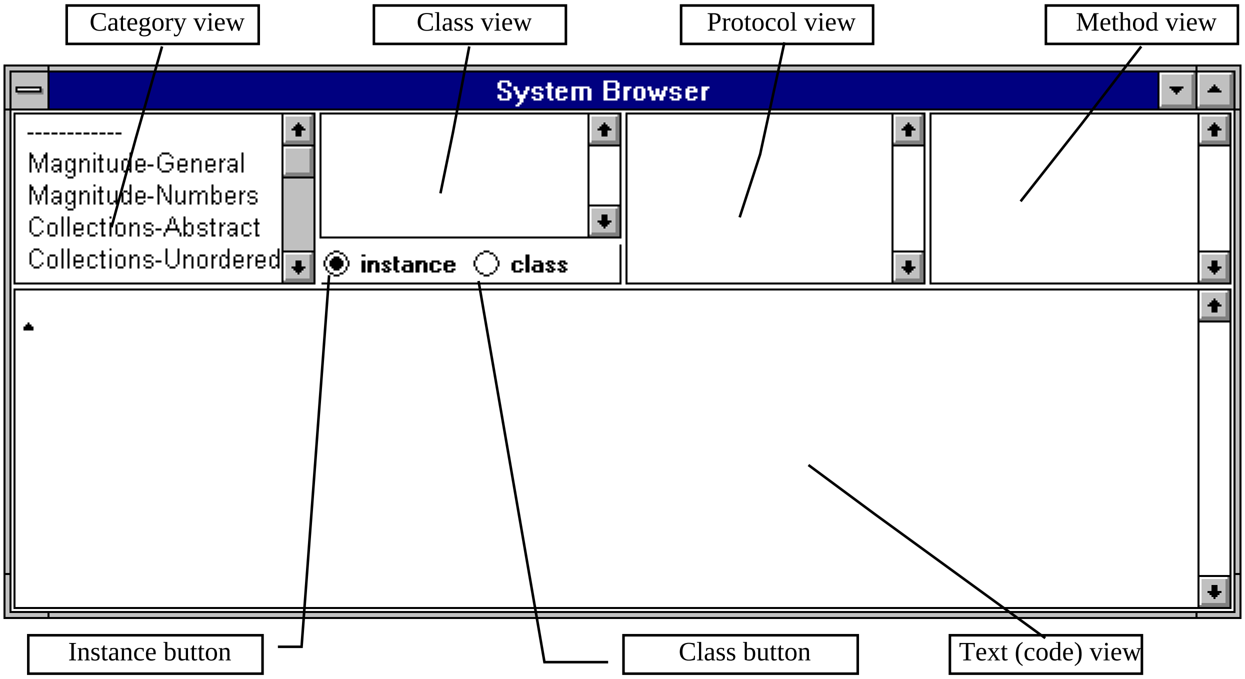
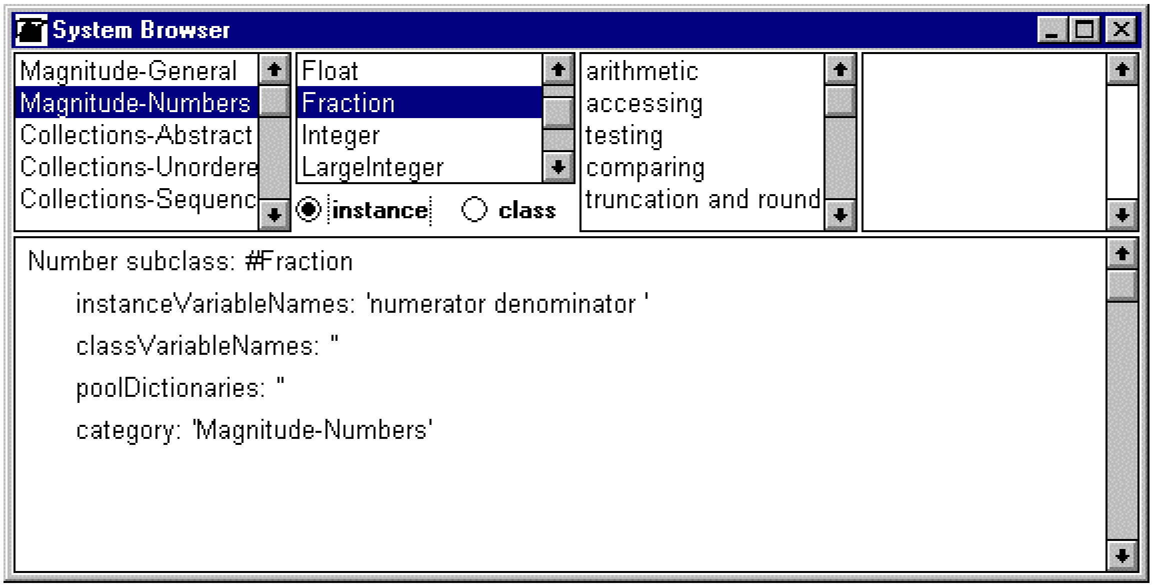


Figure 1.16. A new System Browser and its parts.

To access the definition of a class or to create a new one, click the name of its category. If you don’t know which category contains the desired class, use ‘find class’ in the <operate> menu of the category view. When you select a category, the *class view* displays all classes in this category (Figure 1.17) and the text view at the bottom of the window displays a Smalltalk expression that can be used to add new classes; this part of the Browser is irrelevant at this point. To examine information about a class, select the class in the *class view*. In our example, we selected category Magnitude-Numbers, scrolled the class view down, and selected class Fraction. The text view at the bottom of the Browser now shows the Smalltalk messages that created the class.



```
Number subclass: #Fraction
    instanceVariableNames: 'numerator denominator '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Magnitude-Numbers'
```

Figure 1.17. System Browser showing the definition of class Fraction. Details will be explained later.

The text view now displays the definition of the class and provides access to definitions of all messages defined in the class. This information can be accessed from popup menus or by making further selections in the remaining views of the browser. In this section, we are not yet interested in Smalltalk code but we might want to look at the comment that programmers write for a new class to describe its purpose and structure. To view the comment, open the <operate> menu in the class view (Figure 1.18), select the *comment* command and the comment of the currently selected class will appear in the browser.

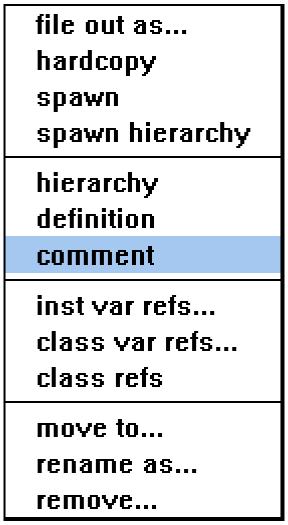


Figure 1.18. To display the comment of a selected class use the <operate> menu.

You will have to learn how to use the System Browser because it is the most important part of Smalltalk's development environment.

Main lessons learned:

- To access a class in the Smalltalk library, use the System Browser.
- The System Browser lets you view and edit definitions of all classes in the library and add new ones.
- The System Browser groups classes into categories. Each class is in one category and categories don't overlap.
- The purpose of a category is purely organizational and has no effect on the behavior of the class.
- The System Browser is the most important Smalltalk tool.

Exercises

1. Use the System Browser to display the comments of the following classes and print them out using command *hardcopy* in the text view's <operate> menu.
   a. Date (category Magnitude-General)
   b. Fraction (category Magnitude-Numbers)
2. List all classes in category Collections-Abstract.
3. Instead of opening the System Browser and viewing the whole library, you can open a specialized browser on a selected category, a selected class, or even a smaller part of the library. To do this, select the category or class in the browser and use the *spawn* command. Use this technique to open a category browser on category Magnitude-Numbers, and a class browser on class Fraction.

1.7 Object properties

We have already seen that objects understand messages and usually contain information. As an example, an instance of Cow knows its name and color and knows whether it is hungry or not. Those properties of an object that can change during its lifetime are often referred to as its *state* properties.

If we are working with several cows, each cow may be in a different state at any given moment and each must therefore carry its state information with it. References to an object's properties are called *instance variables* because the value of the state object to which they refer may change during the object's lifetime. As an example a cow that has just eaten will not be hungry for a while - but eventually will get hungry again and will need to eat. The state of the object referred to by the variable that keeps track of this aspect of the cow's condition will thus vary between true and false as the state of the cow object changes. Each instance variable of a given object has its own distinct name so that we can refer to it when we need to access or manipulate it via the object's functionality.

We can thus think of an object (Figure 1.19) as capsule consisting of a state (held in instance variables) and functionality (defined by *methods* - detailed definitions of the behavior of each individual *message*). When an object receives a message, Smalltalk looks up the corresponding method in the receiver's class and executes its definition; in this process, the method may change the object's state and thus the values of its instance variables. A list of all variables that describe an object's state are a part of its blueprint and are kept by the class of the object. You can find them in the System Browser.

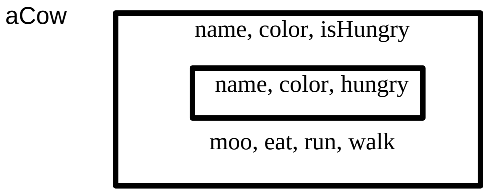


Figure 1.19. An object encapsulates state and provides functionality via messages.

As an illustration of this principle, the definition of class Cow contains

- definition of method new for creating new cow instances (*class* method of Cow),

- definitions of methods name, moo, eat, and so on (*instance* methods of Cow),
- list of names of instance variables holding a cow's name, color, and isHungry.

While the class has a list of instance variable names, each instance of Cow holds the *values* of its instance variables, and is aware of its class so that when it gets a message, it can access the corresponding method describing how to execute the message (Figure 1.20).

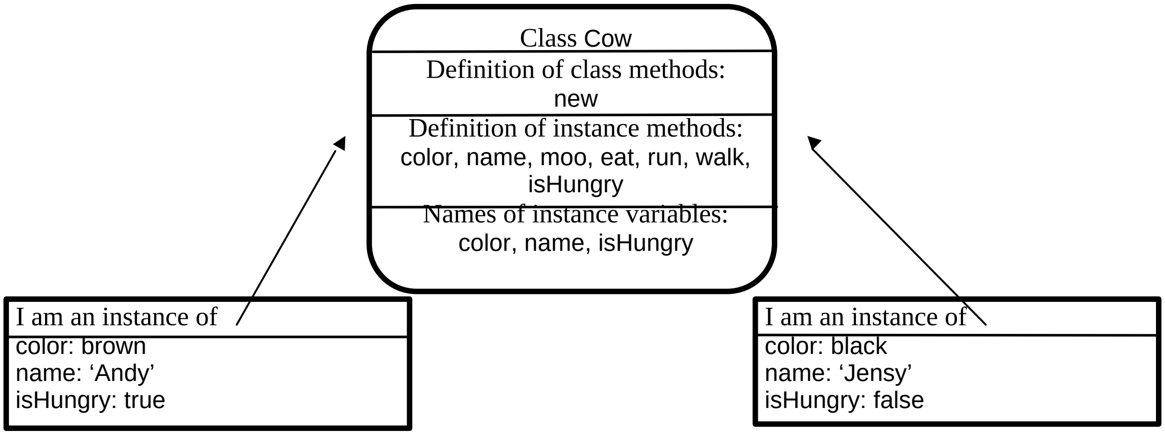


Figure 1.20. Each object knows its class and the values of its instance variables. The definition of a class contains the instance blueprint: a list of instance variable names and definitions of class and instance methods.

Our existing model of a class definition is not symmetric because it has no counterpart of instance variables. In reality, many classes also have *class variables* because not only instances but even classes themselves may have properties. Class variables are typically used for one of three reasons. One is that a class may need to keep some generally useful information, another is that a class may need to keep some information needed to create its instances, and the third is that it may need to know something common to all instances.

As an example of a class variable holding *generally useful information*, consider that many numeric applications need to know the value of π. Since π = 3.14159... - a floating-point number - one would expect that floating-point numbers should be able to supply this information. It would be wasteful for each floating-point number to carry along an identical copy of the same π object, and the value of π is thus stored only once - as the value of *class variable* Pi declared in class Float.

As an example of information that a class needs *to construct its instances*, consider text objects. All text objects are normally displayed with the same font and the class that produces text objects needs to know what this default font is. The default font could thus be stored in a class variable. (The reality is somewhat more complicated.)

An example of information that a class might need to know about its instances is found in class Window: It is often useful to know which window on the screen is currently active and class Window keeps this information in class variable CurrentWindow.

We conclude that a more realistic picture of the general structure of a class definition is as in Figure 1.21.

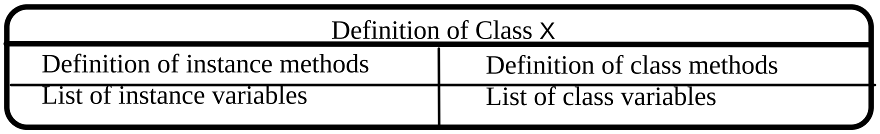

Definition of Class X
Definition of instance methods
Definition of class methods
List of instance variables
List of class variables

Figure 1.21. A class definition may include class and instance methods, and class and instance variables.

Values of instance and class variables are objects

After introducing the concept of variables as holders of object properties, it is now time to examine the nature of these properties. As one could expect, *values of instance and class variables are again objects*. This principle may extend to a great depth because these objects often have their own properties which are again objects, and the structure of an object may continue branching for a long time. Many objects are thus nested assemblies of objects and can be represented as graphs of chained references from one object to another (Figure 1.22). Eventually, of course, the chain of references must stop and some objects must contain 'real' values rather than just references to other objects. These elementary objects represent things such as basic kinds of numbers, and characters (letters, punctuation, and digits).

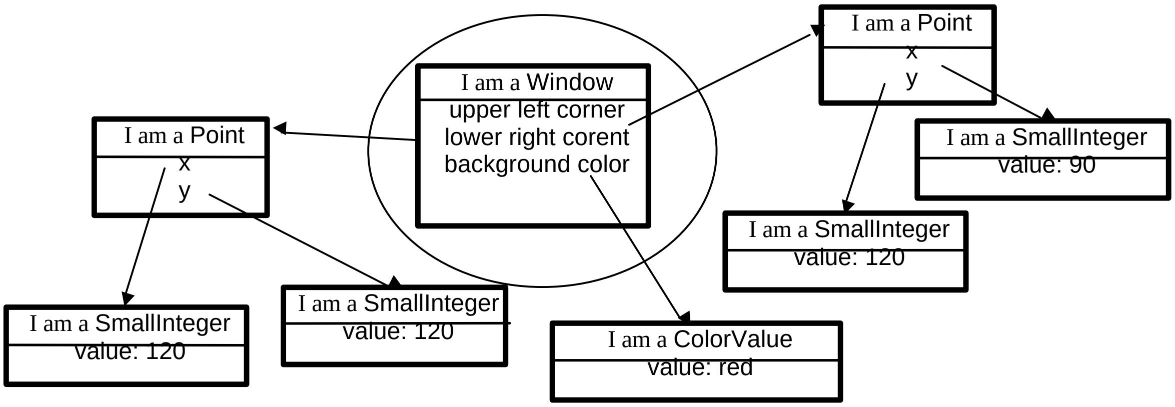


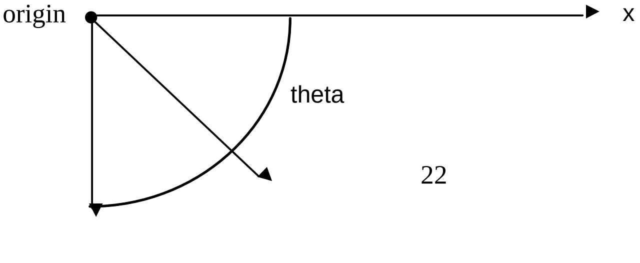
Figure 1.22. Values of instance and class variables are objects. (An idealized representation.)

The nested nature of Smalltalk objects is analogous to the nested structure of objects in the real world. As an example, a computer consists of a chassis, a motherboard, a power supply, adapter cards, connectors, and other components. A motherboard, in turn, consists of a printed circuit board, chips, connections, and connectors. A chip consists of a package, pins, an electronic circuit, and so on. Eventually, further decomposition becomes meaningless and we reach 'elementary' objects. We conclude that although the structure of objects may be very complicated, the fundamental simple principle remains unchanged - everything is an object.

Information hiding

An important property of Smalltalk objects is that their state components are hidden from other objects. An object is thus like a black box which contains some functionality and has an internal state, but the values of its components cannot be seen or changed from the outside, unless the object understands messages that access these values. In fact, other objects cannot even know what the internal representation of an object is. This property is called *information hiding* and it means that the only way to get at an object's state is via the messages that it understands.

As an example of information hiding, class Point which is a blueprint for points on the screen includes methods called x and y which return the point's x and y 'Cartesian' coordinates, and messages r and theta which return its 'polar' coordinates (Figure 1.23). Since Cartesian and polar representations can be converted into one another, we could implement Point using either instance variables x and y, or instance variables r and theta. If we used the polar implementation, methods r and theta would simply return the values of the corresponding instance variables whereas methods x and y would have to calculate x and y from the polar coordinates. In reality, Point is implemented with Cartesian representation and methods x and y return the values of instance variables x and y without calculating anything, and r and theta calculate their values from x and y. This fact is, however, hidden from other objects which can only question points using messages x, y, r, and theta.



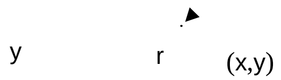


Figure 1.23. A point can be represented by Cartesian coordinates x and y or by polar coordinates r and theta.

Information hiding provides great freedom to implementers. As an example, if we decided to change Point to use polar coordinates, all programs using points would still work provided that we change the implementation of all Point methods to reflect the new implementation, because other objects can only use points through their *message interface* - by sending messages. Information hiding thus means that properly implemented changes to internal representation do not affect other objects.

To appreciate the advantage of information hiding, assume that the designer of a class such as Window could access instance variables of another class such as Point directly, without using messages, and use the values of x and y for some calculations. If the somebody changed the implementation of Point to use Cartesian coordinates x and y instead of polar coordinates r and theta, Window would no longer work because x and y would no longer exist. And if window stopped working, the browser would not work and neither would any applications using window interfaces. Because of information hiding, class Window communicates with points only through their message interface and Window will thus work even if Point changes as long as the message interface of Point is properly maintained.

Before closing this section, it is worth noting that objects don’t have to have instance or class variables. As an example, instances of class True don’t have instance variables because they don’t need any. Although the Smalltalk library includes many classes with no instance and class variables, all Smalltalk objects have functionality. There are no objects that don’t understand any messages because such objects would be totally useless.

Main lessons learned:

- Objects encapsulate functionality and state.
- Each object carries values of its state properties with it.
- References to state properties of classes are called class variables, references to state properties of instances are called instance variables.
- Values of class and instance variables are objects.
- The definition of a message is called a method.
- Method definitions and names of class and instance variables are included in the definition of the class.
- Values of state properties are hidden inside objects and other objects can access them only via accessing methods – if the class definition provides them.
- The collection of all messages understood by an object is called its message interface.
- Communication with an object is only through its message interface.

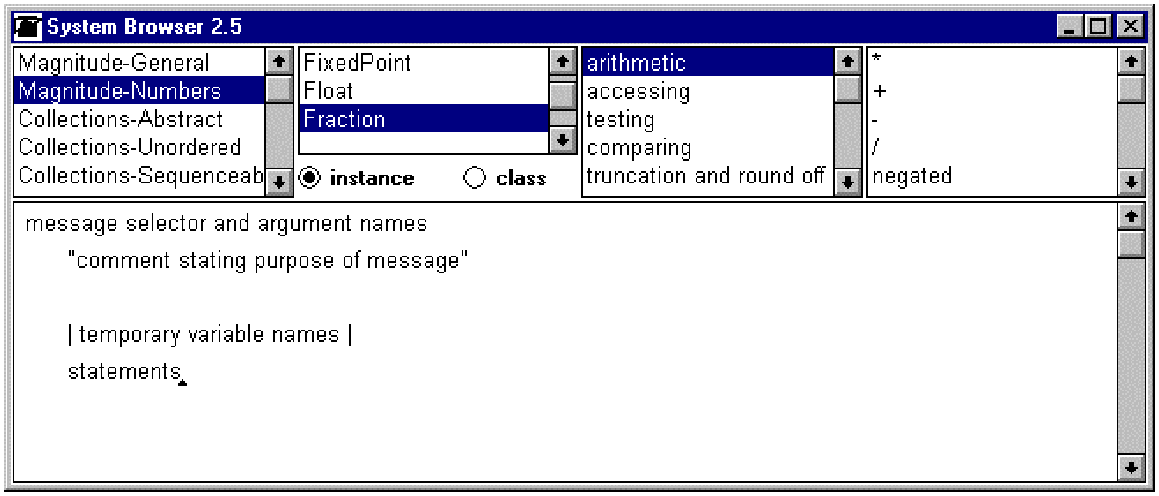
Exercises

1. Explore the properties of pens and erasers using the *Pen World*.
2. List the major components of the following objects and expand the internal structure of some of the components to a reasonable number of levels.
    a. A car from the perspective of a car mechanic.
    b. A car from the perspective of a car salesman.
    c. A bank from the perspective of a bank teller.
    d. A bank from the perspective of a bank customer.
    e. An art museum from the point of view of an art historian.
    f. An art museum from the perspective of a civil engineer.
3. An object with instance variables but no methods holds information so why would it be useless?

1.8 Using System Browser to find out about objects

To view or edit the state variables and methods of a class, open the System Browser and select the desired class. The text view will display the definition of the class, and the commands in the <operate> menu of the class view allow you to display the comment and obtain all sorts of other information about the class.

Depending on the selection of the *instance*/*class* pair of buttons, the protocol view shows all instance or class *protocols* of the class. The function of protocols is similar to that of categories - to organize methods for easier access; and just like categories, protocols don’t have any effect on object behavior. If you want to view a method, select its protocol (if you know which one it is) and click the methods name, or use the *methods* command in the protocols’ <operate> menu. In our example, we selected class Fraction and instance protocol arithmetic and obtained the display in Figure 1.24.

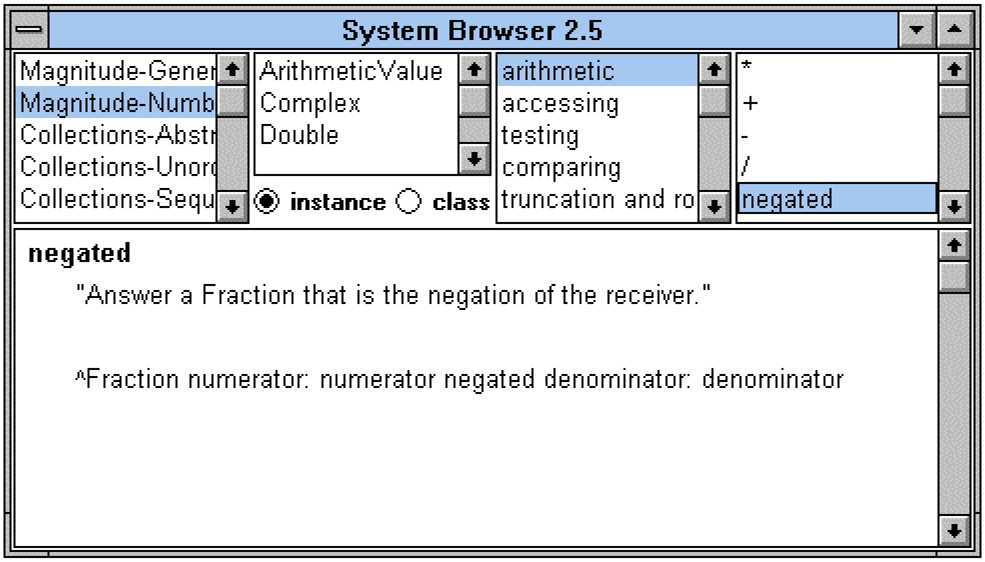


```
message selector and argument names
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

Figure 1.24. Methods in *instance* protocol arithmetic in class Fraction.

The method view on the right shows all methods in the selected protocol and the text view shows a template that can be used to create a new method. When you select a method in the method view, its Smalltalk code appears in the text view as in Figure 1.25 and you can now read it, find all references to it, modify it, delete it, and so on. We will learn later how to do this and how to understand the code.



```
negated
	"Answer a Fraction that is the negation of the receiver."

	^Fraction numerator: numerator negated denominator: denominator
```

Figure 1.25. Definition of instance method negated in the arithmetic protocol of class Fraction.

Main lessons learned:

- Class methods and instance methods are divided into protocols.
- Each method is in one protocol and protocols don't overlap.
- The concept of a protocol is purely organizational and has no impact on behavior.

Exercises

1. List all instance variables of the following classes:
   a. Circle
   b. LineSegment
2. List all instance methods of Fraction that use instance variable numerator. (Hint: Use command *inst var refs* in the <operate> menu of the class view of Fraction.)
3. List all references to class Date. (Hint: Use <operate> menu command *class refs*.)
4. List all class variables defined in the following classes:
   a. Float (category Magnitude-Numbers)
   b. Date
5. List all instance protocols defined in the following classes:
   a. Character (category Magnitude-General)
   b. Date
   c. Number (category Magnitude-Numbers)
6. Repeat the previous exercise for class protocols.
7. List all methods in the following classes and protocols:
   a. Class Fraction, instance protocol arithmetic
   b. Class Date, instance protocol inquiries
   c. Class Date, class protocol creation
8. List all methods that send message today defined in class Date. (There are two ways to do this - either by selecting the method in the browser and executing *senders* from the method view's <operate> menu, or by using command *References To...* in the Visual Launcher (Figure 1.26).
9. Find all classes that define method +. (Either open a browser on one such definition - for example in Fraction - and execute command *implementers* from the method view's <operate> menu, or use command *Implementers* in the Visual Launcher).
10. Find all messages sent by class method addTime: in class Time. (Hint: Select the method in the browser and execute command *messages* in the <operate> menu of the method view.)

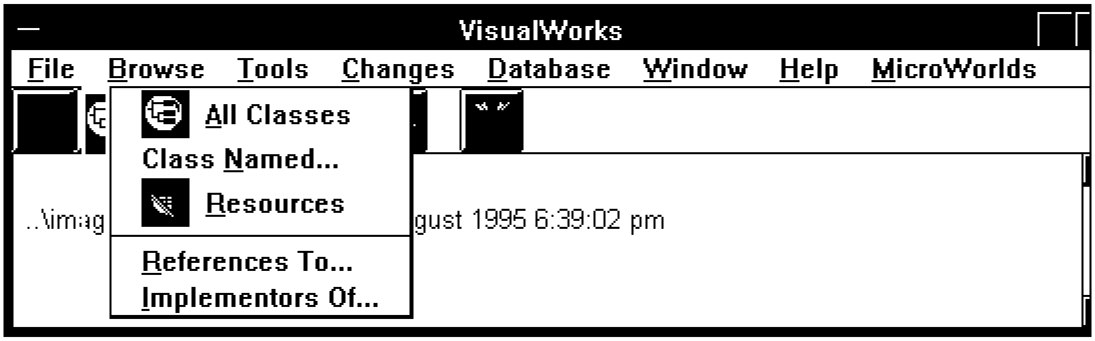


Figure 1.26. To view implementors or references to a method, use the *Browse* drop down menu.

1.9 Class, subclass, superclass, abstract class, inheritance, class hierarchy

Up to now, we dealt with classes in isolation. In this section, we will consider classes in relation to one another in the way that biologists classify plants and animals.

The system for the classification of animals looks like an upside-down tree (Figure 1.27) in which items at the bottom are animals such as lions or tigers, and higher up items are abstractions - such as mammals. Of course, even the lion at the bottom of the tree is not a real lion but a concept that defines what a lion is. In our terminology, this concept corresponds to a class. A particular lion in the wild or in a ZOO, on the other hand, is an instance of this class.

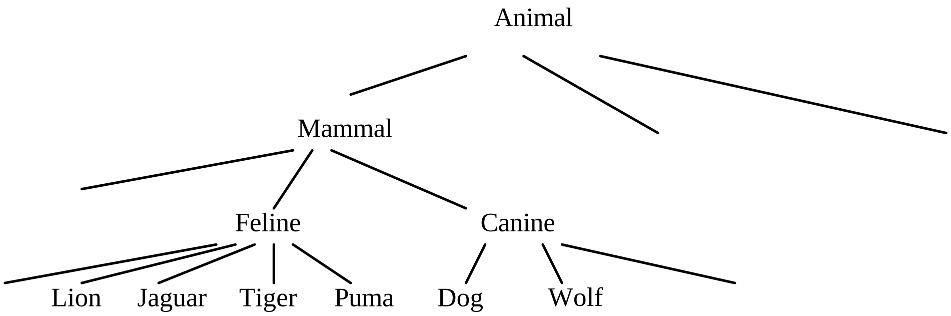


Figure 1.27. The classification tree of animals.

In our diagram, a Lion is a special kind of Feline. Similarly, a Tiger is a special kind of Feline, and so is a Puma, and a Jaguar. In class terminology, we would say that class Lion is a special kind of class Feline - a *subclass* of class Feline. Inversely, class Feline is a generalization of classes Lion, Jaguar, Tiger, and Puma - a *superclass* of Lion, Jaguar, Tiger, and Puma. The classification scheme thus defines a *hierarchy of classes, subclasses, and superclasses* and the arrangement reflects generalization (going up) and specialization (going down).

The purpose of a class hierarchy is to *factor out* shared features of related classes. As an example, since Lion is a special kind of Feline, it has all the properties and behaviors of Feline - and several additional specialized ones. Similarly a Tiger - being another subclass of Feline - has all the properties of Feline, and some additional ones. By constructing a hierarchy, we can thus gather all properties common to several related classes in one place - the superclass - and pass them down to all subclasses implicitly, without having to repeat them for each subclass. A description of a Tiger, for example, may then be quite short because we can say that a Tiger is a Feline and has certain special additional properties listed in the description. This passing of superclass properties and behaviors down to subclasses is called *inheritance*.

Besides making specifications shorter, inheritance has several other benefits. One is that by listing shared properties and behaviors in one place, we can avoid omissions and mistakes that we might commit if we kept a complete detailed description for each animal separately. Keeping shared properties in one place also means that if we decide to modify them because we have a new insight into the nature of the subclasses or because we made a mistake, we only need to modify the superclass. This again saves work and prevents omissions such as modifying individually all subclasses of Feline but forgetting to modify the Lion class.

Our example shows that class hierarchies may be any number of levels deep. Feline is a subclass of Mammal, together with Canine. Class Feline thus has all properties of Mammal and since Lion has all the properties of Feline, Lion also has all the properties of Mammal. In other words, a class inherits not only the properties of its immediate superclass, but the properties of *all* its superclasses up to the top of the hierarchy tree. This increases the savings and the security obtained by superclassing - but makes deeper hierarchies more difficult to understand.

Another look at Figure 1.27 suggests another important concept. The classes at the junctions (nodes) of the tree can be divided into two groups: those that define real animals (such as Lion, Jaguar, and Puma), and those that are only abstractions (such as Feline and Mammal). Classes that correspond to real animals have instances in the real world while those that represent abstractions don't: There is no physical animal derived directly from the abstract concept Feline, there is only a physical Lion, Puma, or Jaguar. In OO terminology, classes representing pure abstractions are called *abstract classes* whereas those that can be instantiated (represent real objects) are called *concrete classes*. The essence of abstract classes is that they are not used to create instances and their only purpose is to factor out shared behavior and state information and pass it down to the concrete subclasses.

As we have seen, the main point of subclassing is to take advantage of inheritance, and object-oriented languages distinguish two kinds of inheritance - single, and multiple. The kind of inheritance that we used so far is called *single inheritance* because it limits the number of direct superclasses that a class may have to one - each class has a single superclass (the class at the top of the tree of course does not have any superclasses). In *multiple inheritance*, a class may have any number of immediate superclasses. While the animal kingdom tree is an example of single inheritance, a real world parallel of multiple inheritance is the human family: Every human being has two biological parents and inherits some properties from one and other properties from the other. Single and multiple inheritance correspond to class hierarchy structures depicted in Figure 1.28.

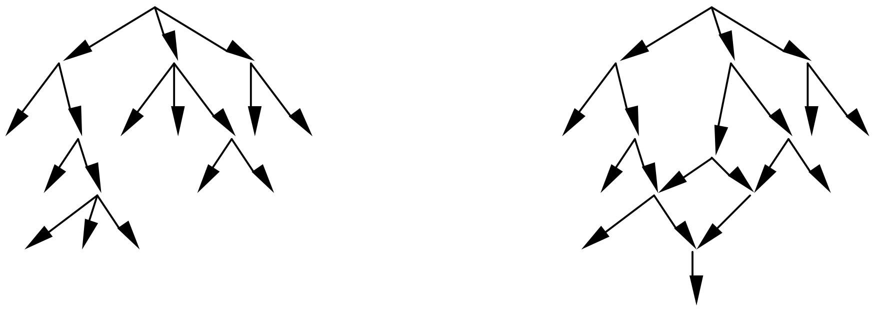


Figure 1.28. Single inheritance (left) and multiple inheritance (right).

The debate over which kind of inheritance is better is unresolved because each has its advantages and disadvantages. The advantage of multiple inheritance is that it is sometimes very natural. As an example, data files can be divided into read-only files, write-only files, and read-write files. If we were to design a class hierarchy for files, we would probably want to define one class for read-only files, one class for write-only files, and define the class for read-write files as a subclass of both (Figure 1.29). Class read-write file is thus a natural candidate for multiple inheritance. If the language does not support multiple inheritance, we must make class read-write file a subclass of read-only file, for example, and copy all the writing properties from the write-only file class. This is an unpleasant duplication.

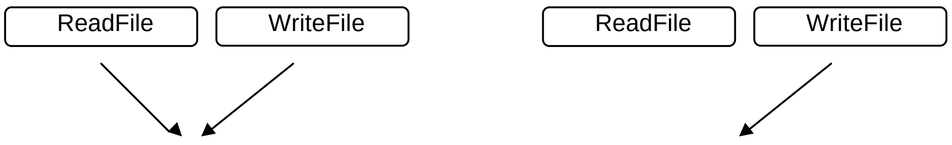


Figure 1.29. Class ReadWriteFile in a language with multiple inheritance (left) can inherit its functionality from two superclasses. In a language with single inheritance (right) some functionality must be duplicated.

The disadvantage of multiple inheritance is that the exact nature of inheritance in non-trivial hierarchies may be very difficult to understand, especially if some superclasses share identically named but different methods or variables. Moreover, control of the effect of changes in higher level classes on classes at the bottom of the hierarchy tree can be disastrous. This is, of course, a problem with single inheritance as well but it becomes more difficult with multiple inheritance.

Commercial implementations of Smalltalk take the view that ease of understanding and maintenance are more important and use single inheritance. (There is also the historic legacy that the original Smalltalk used single inheritance.) There is, however, no built-in restriction against multiple inheritance and multiple inheritance extensions of Smalltalk has also been implemented. Some other programming languages such as C++ allow multiple inheritance but Java uses single inheritance.

Main lesson learned:

- Classes are organized in a tree-like hierarchy, leading to the concepts of a superclass, subclass, and inheritance.
- Going up the hierarchy tree corresponds to generalization, going down corresponds to specialization.
- The main benefit of subclassing is inheritance: A subclass inherits all properties of its superclass.
- Subclassing should be used when the new class adds new behaviors to an existing class. In other words, inheritance should be additive.
- Since each class inherits from its superclass, each class inherits from all its superclasses. Technically speaking, inheritance is transitive.
- There are two types of inheritance - single and multiple. In single inheritance, a class may only have one immediate superclass. In multiple inheritance, a class may have any number of immediate superclasses. In both cases, a class may have any number of subclasses.
- Both single and multiple inheritance have advantages and disadvantages.
- Most Smalltalk implementations use single inheritance.
- An abstract class gathers useful shared behavior but is not used to create instances.
- A concrete class is used to create instances.

Exercises

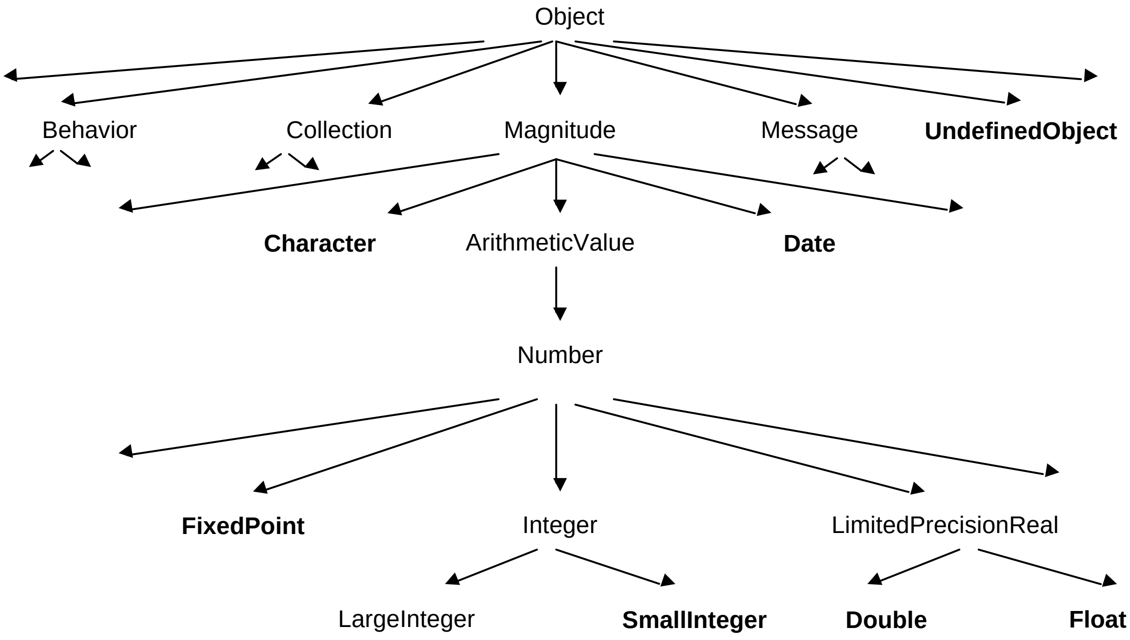
1. Consider a program for processing student information. We want to hold the following data on each student: name, address, degree studied, year of registration, courses taken, remaining required courses. Each science student has a computer account number on the ‘science computer network’, and each arts student has an account on the ‘arts network’. Each language student has a password for the language lab, each physical education student has a locker number. All computer science and computer engineering students have a quota on the maximum amount of CPU time on a super computer and their records include the amount of CPU time left. Computer engineering students have a limit on the amount of material they can get for the computer that they are building.
   Design a class hierarchy capturing these administrative requirements and list the components of all classes. Explain which classes are concrete and which are abstract.
2. The concept of classification (as in the animal kingdom) is related to the concept of subclassing but not equivalent to it. In particular, one object may be classified as substantially different from another and yet the two objects may be modeled by the same class. As an example, if the only difference between language students and computer science students is that the former use their password to enter language labs and the latter use it to enter computer labs, we may want to model the two kinds of students by the same class. Extend the previous example by adding several new categories of students and data,

construct a new class hierarchy, and analyze it in terms of the difference between subclassing and classification.

3. Multimedia computers use text, graphics, animated graphics, images, digitized movies, recorded sound, sound calculated from a musical notation, and perhaps smell, taste, and touch in the future. Propose two classifications of all the listed media, one based on single inheritance and one on multiple inheritance.
4. Given the similarity of animals in the *Farm* program, describe how their implementation could benefit from inheritance.
5. Which of the following is best characterized as 'is-a-kind-of' relationship and which is a 'has-a' (contains -) relationship? Which of them would be suitable for subclassing (define suitable abstract class if appropriate and draw the class hierarchy) and which of them would best be implemented by *aggregation* - a class with multiple components? Draw a 'containment diagram' and a class hierarchy diagram as appropriate.
   a. Arm chair, dining chair, desk chair, bed, sofa, love seat, fouton, desk, coffee table, dining table.
   b. Chair: back rest, back rest frame, back rest cushion, seat, hand rest, screws, seat frame, seat cushion.
6. Create your own example of subclassing versus aggregation similar to those in the previous exercise.
7. A subclass normally extends the functionality of its superclass but there are exceptions. As an example from biology, an ostrich is a bird that does not fly and its response to command fly must be redefined as 'sorry, I cannot'. Give two examples in which a subclass must redefine the inherited behavior.
8. An abstract class often only specifies that all its subclasses should implement a certain behavior but does not implement it itself. To do this, the corresponding method consists of the message subclassResponsibility and each subclass must define the method in its own manner. Find three methods that are 'implemented' in this way. (Hint: Use the *references to* command in the *Browse* command of the *Visual Launcher* and specify subclassResponsibility as the name of the method.)

1.10 Smalltalk's class hierarchy

We already noted that VisualWorks Smalltalk contains a very large library of built-in classes and that the user can access their definitions, modify them, delete them, and add new classes. The classes in the library use single inheritance. At the top of the hierarchy (Figure 1.30) is a single class called Object and all other classes inherit all of its properties. Our diagram shows a minute part of the hierarchy with some of the classes implementing numbers. The diagram shows that class SmallInteger, for example, is at the sixth level of depth which indicates that it inherits much functionality from its superclasses.



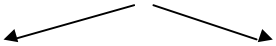
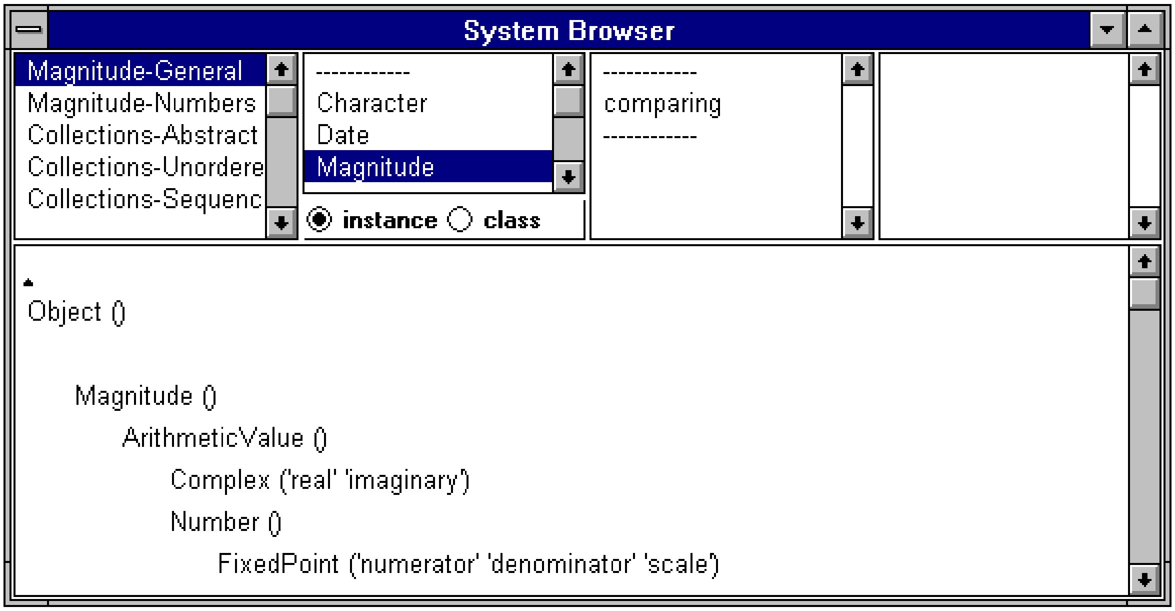


Figure 1.30. A very small part of VisualWorks class hierarchy tree. Concrete classes are shown in **boldface**, all other classes are abstract.

To find out about Smalltalk's class hierarchy, use the *System Browser*. After selecting a class such as Magnitude, select the *hierarchy* command in the class view <operate> menu and the text view will display as in Figure 1.31, showing subclasses and superclasses and all instance variables.



```
Object ()

    Magnitude ()
        ArithmeticValue ()
            Complex ('real' 'imaginary')
            Number ()
                FixedPoint ('numerator' 'denominator' 'scale')
```

Figure 1.31. System Browser showing a part of the hierarchy of class Magnitude.

The hierarchy shown in Figure 1.31 shows both abstract and concrete classes. Class Magnitude is abstract and factors out properties needed for comparison, and its subclasses include numbers, printable characters, date objects, and time objects. Magnitude has no instances because there is no need for such abstract objects. Classes ArithmeticValue, Number, and FixedPoint are also abstract but class Complex is concrete and used to create concrete numbers. Classes Character, Date, and Time further down the Magnitude hierarchy are also concrete.

In our discussion of subclassing and inheritance, we have so far talked mainly about inheritance of properties without paying attention to the effect of inheritance on the execution of messages. We will now show that inheritance affect *message resolution*, the way in which Smalltalk finds the definition of a message being executed. As we know, when a program sends a message to an object, Smalltalk examines the receiver's class and executes the method if it is defined in this class. If the method is inherited, Smalltalk does not find the definition in the receiver's class, climbs to its superclass and looks for the definition there. If it is found, it is executed, otherwise the climb continues until either a definition is found or executed or until the top of the hierarchy is reached without finding the method. In this last case, Smalltalk maintains control and creates an *exception* which allows the user to assume control and correct the error or terminate the execution of the program. We will deal with this issue in more detail later.

Main lesson learned:

- VisualWorks classes are organized into a single inheritance hierarchy.
- The top of the Smalltalk hierarchy is class Object and all Smalltalk classes inherit all its properties.
- When a message is sent to a receiver, the first definition of the message found going up from the class of the receiver is executed. If no definition of the method is found, Smalltalk creates an exception and passes control to the user.

Exercises

1. Find and print out class hierarchies of the following classes. Use the *instance* side of the browser and list all superclasses and all subclasses but nothing else.
   a. Character
   b. Date
   c. Number
   d. Text
2. How many instance variables does class Date inherit and how many new instance variables does it define?
3. How many instance and class variables do all classes inherit from Object?

1.11 Polymorphism

The word *polymorphism* means ‘occurring in many shapes or forms’. In object-oriented programming, polymorphism means that different classes may implement a method with the same name and the same basic behavior but the details of the definition are different in each class. As an example, message + is defined in all number classes and in all of them it means the same thing – addition - but its implementation is different in each of them. As another example, every Smalltalk object knows how to construct its textual description in response to message printString but different kinds of objects obviously construct their descriptions differently.

Although the term polymorphism is not used in everyday communication, the concept is not unfamiliar. As an example, when you go to a restaurant and order a soup, you don’t first ask whether the cook is a ‘microwave cook’ or a ‘stove cook’ and place a different order in each case. You simply order a soup and the cook executes the order in the way in which he or she is trained to execute it. Your message thus sounds the same and has the same effect – your soup - but its implementation by different cooks varies. As another example, if you go to a hotel and your room is cold, you change the setting of the thermostat - effectively sending the message ‘change temperature’ - without thinking about whether this activates a gas-based heater, a water-based heater, a wood furnace, or a nuclear fusion-based heater. The same ‘message’ produces ‘the same’ result but how the result is achieved depends on the object that processes the message.

Polymorphism is a very important concept and we will now illustrate it on an example from the programming world.

Use of polymorphic comparison in sorting

Sorting numbers according to magnitude and sorting text according to lexicographic order both depend on the same principle - the ability of the sorted objects to compare themselves with one another. Even a child can sort any collection using the same *algorithm* (step-by-step procedure) if it can take any two of its elements and decide which comes first and which comes second. Even when your sorting needs are more unusual, the principle of sorting remains the same. As an example, if you want to sort a collection of apples according to taste, you can use the same principle as for sorting numbers or text if you have a way to compare the taste of any two apples in the collection.

We conclude that to sort objects, we need only one sorting procedure based on the comparisons of any two objects in the collection. The method uses some form of the ‘precedes’ operation, implemented perhaps as a < message. In a programming language that allows the use of polymorphism to the extent that a sorting algorithm can work with *any* two objects that understand the < message, we thus need only one sorting method and this single method will be able to sort any collection of objects that can execute (in pairs) a < message - even if it is implemented quite differently in different classes. Smalltalk's polymorphism has this property and this gives it enormous power.

One of the beauties of polymorphism is that the programmer who wrote the sorting method did not have to anticipate all its possible uses - and, in fact, could not - and yet created a tool that solves a very common problem faced by developers of many different applications. Once the method exists, programmers will never have to think about writing a sorting method for their specialized needs - thanks to polymorphism.

Another advantage of polymorphism is that it eliminates the need to test what kind of object we are dealing with when we need to perform a polymorphic operation. As an example, if we want to add together two numbers, we don't have to test which of the many possible varieties the numbers belong to and then execute the appropriate kind of addition. Instead, we simply ask the numbers to do addition and they do it in the way appropriate to their nature. As another example, a drawing program that displays a collection of geometric objects stored in a file can use polymorphism to ask each individual object to display itself, without testing whether the object is a rectangle, a line, an arc, and so on, and then issuing the appropriate message. This simplifies programming, speeds up program execution by eliminating tests, and makes programs very flexible and easily extendible.

Polymorphism and binding

The possibility to have different implementations of the same method in different classes does not, by itself, give a programming language all the power that we described. If a language requires that we name the class of objects that can be used with a particular method, the advantage of polymorphism is severely restricted because each unrelated class then requires its own definition of sorting. Only a language that does not require specification of the class of its objects in the definition of a method can take full advantage of polymorphism. Smalltalk has this ability but some other languages don't.

The disadvantage of the flexible arrangement that we described is that if the definition of a sorting method does not specify which classes of objects it may use, the *compiler* program that converts the definition into computer code cannot decide which definition of < to bind to the code. An instance of any class that has the definition (and all its subclasses) could be the receiver and any of these methods could be executed. The compiler must thus leave the decision as to which < method to use until execution time: If the receiver of the < method is ab integer number, than the definition of < for numbers will be used. If the receiver is a string of letters, then the definition of < for strings must be used, if the collection contains dates, we must use the Date definition, and so on. The run-time look up required to make select the appropriate method takes extra time but modern implementations are so efficient that this extra time is negligible in almost all applications.

Languages that leave the decision of which kind of object will execute a message until execution time are said to use *dynamic typing* whereas languages that require that binding be specified at compile time are said to use *static typing*. Smalltalk uses dynamic typing and this is one of the major reasons for its power.

In addition to eliminating method lookup at run time, another advantage of static typing is that the compiler can catch attempts to send messages to objects that do not understand it. Proponents of dynamic bounding will counter that code that contains such basic errors has not been properly designed and inspected.

Main lesson learned:

- The term polymorphism means that a message with the same name and the same purpose is declared in several classes in a way appropriate for the nature of the class.
- Polymorphism often makes it possible to eliminate lengthy tests to decide which message in which class is suitable to perform a given task. With polymorphism, the receiver itself makes the decision by executing its own form of the message.
- When polymorphism is used to eliminate multiple decisions, it speeds up execution.
- Polymorphism makes it possible to create very general solutions that remain valid even when new classes implementing the task are added to the library. In this way, it greatly contributes to reuse.
- Dynamic typing refers to leaving the decision of which object may be the receiver or the argument of a message until run time. Dynamic typing is necessary for full fledged polymorphism.
- Languages with static typing require full specification of the kinds of receiver and arguments in the code. This restricts the use of polymorphism but provides greater security for coding and may result in slightly faster execution.
- Smalltalk uses dynamic typing.
- An algorithm is a precise description of a sequence of steps that must be executed to solve a problem.

Exercises

1. Find and count all definitions (‘implementations’) of the following methods.
   a. displayOn:
   b. squared
   c. <
   d. =
2. Give an example of polymorphism from the physical world.
3. The three animals in the *Farm* world share the following messages: color, name, eat, isHungry, run, walk. Some of them are implemented in the same way in each animal, others are not and the program executes them polymorphically. List each group on the basis of your intuition.

Conclusion

The principle of object-oriented problem solving is solving problems by building computer models of miniature worlds populated by interacting objects. Although this paradigm has been applied in a variety of settings, its major use is in computer programming with object-oriented programming languages. Object-oriented programming languages support the concept of a library of objects and provide an environment with tools for creating, deleting, and modifying them.

For the purpose of programming, an object can be thought of as a command-obeying robot specializing in a well-defined behavior and maintaining specialized knowledge. Another possible view is to think of each object as a specialized computer. An object-oriented program can then be thought of as a multitude of cooperating robots or specialized computers.

A useful introduction to the principle of object orientation is to construct scenarios and conversations - sequences of events that typically occur in the problem world at hand. Scenarios are also used by program developers to discover objects required to solve a given problem. Once these objects are found and their properties described, their representation in the selected programming language gives the computer solution of the problem.

The two basic properties of objects are that they understand messages (requests to perform a service) and that they have an internal state. In Smalltalk, all messages return an object - even if the sender of the message does not need it.

Objects can be divided into two groups. Objects that manufacture other objects are called classes, objects manufactured by classes are called their instances. A class and its instances are different objects and understand different messages. Messages understood by classes are called class messages, messages understood by instances are called instance messages. A detailed definition of the computation performed by a message is called a method.

Since all work in a pure object-oriented language is achieved by sending messages, method definitions themselves consist of message sends. Some of the messages may be directed at the object executing the method (the receiver itself) and others may be directed at other objects. Foreign objects that help in the execution of a method are called collaborators and sending messages to collaborators and taking advantage of their functionality is called delegation.

In addition to functionality, objects have a state but the state is hidden inside the object and can only be accessed by messages. An object is thus an encapsulation of state and functionality. In this, objects resemble electronic chips whose functionality and state are encapsulated inside the chip in some unknown, invisible, and ultimately irrelevant way, and whose only connection to the outside world is via signals transmitted via chip pins.

The state of an instance of a class is stored in its instance variables, the state of the class object is stored in its class variables. The complete description of the functionality of an object (its methods) and a list of its state variables are stored in its class definition. The definition may also include class variables and class methods.

In addition to instance and class variables, OO languages use other kinds of variables that will be introduced later. The common properties of all variables are that at any time, a variable refers to a single object but during the execution of a program the state of this object or even its identity may change. Note also that although one variable always refers to exactly one object, an object may be referred to by more than one variable at a time.

Classes are organized into hierarchies in which most classes have a superclass and inherit all its properties. Using inheritance, each subclass only needs to define its special features to achieve its distinct behavior. Inheritance is transitive which means that each class inherits all properties of all its superclasses. Subclassing saves work, eliminates duplication - a potential source of mistakes, and enforces shared behavior.

Object-oriented programming languages recognize two forms of inheritance - single inheritance and multiple inheritance. In single inheritance, a class may only have one direct superclass; in multiple inheritance, a class may have several superclasses. Graphically, single inheritance results in a class hierarchy that looks like an upside-down tree whereas multiple inheritance may look like a web with crossing links.

The advantage of single inheritance is its relative simplicity. Its disadvantage is that objects sometimes require properties of classes that appear on disjoint branches of the hierarchy tree and some of the properties must then be redefined and uplicated. Multiple inheritance does not suffer from this problem but its disadvantage is that if the inheritance pattern is not trivial, it may be difficult to understand its implications, especially when different superclasses contain similar properties and behaviors. Multiple inheritance also makes it more difficult to control the implications of changes in higher level classes. Most Smalltalk implementations use single inheritance, taking the view that simplicity is more important than occasional duplication. The class at the top of Smalltalk hierarchy is called Object and all Smalltalk classes inherit all its behavior.

Inheritance causes subclasses to inherit both properties and behaviors. It also affects message resolution – the process of finding the appropriate definition of a message. Message resolution consists of looking at the class of the receiver and looking for the method defining the current message, searching superclasses in upward order if the method is not found. A failure to find the method causes an exception

Some superclasses only factor out shared properties and behaviors of their subclasses and are not used as ‘object factories’. Such classes are called abstract whereas classes designed for instantiation are called concrete. Although a superclass is usually abstract, it need not be – concrete superclasses exist too. Similarly, a class at the end of a branch need not be concrete although such an arrangement would not make sense unless the class is defined as a starting point for future speciliazation.

Smalltalk classes are organized into categories - groups related by their purpose - and methods in a class are similarly grouped into protocols. The concepts of categories and protocols are purely organizational and have no effect on class behavior and no relation to class hierarchy.

One of the essential concepts of object-oriented programming is polymorphism. Polymorphism means that several different classes define a method with the same name and the same purpose, and each implements it in a way appropriate for its particular character. Polymorphism makes it possible to create methods with very general applicability, and designs that derive their simplicity from the delegation of decisions to objects performing specialized tasks. Polymorphism facilitates distribution of intelligence among many classes, a design style generally preferred to designs with fewer highly intelligent classes. It also greatly simplifies programs and increases the potential for reuse.

The power of polymorphism is related to the implementation of typing - the time at which it is decided which definition of a method will be used to execute a message. In Smalltalk, this decision is made when the message is actually sent, at run time because the kind of object sending the message is not specified in the program. This kind of typing is called dynamic typing and its advantage is that it allows programs to take full advantage of polymorphism. One disadvantage of dynamic typing is that prevents compile-time checks whether the receiver of a message and the message match, but this should not be a problem in a properly designed and tested program. Another disadvantage of dynamic typing is that the run time decision as to which method definition will be executed requires extra work on the part of the computer. Current execution techniques make this overhead negligible.

Some programming languages require or encourage their users to write programs so that the compiler can determine typing before the program is executed. This typing style, called static typing, eliminates the overhead of dynamic typing and allows prevention of errors due to mismatched receivers and messages. Its disadvantage is that it severely curtails the power of polymorphism.

Terms introduced in this chapter

abstract class - class declared for the purpose of factoring out properties and behavior shared by a group of related subclasses; not intended for instantiation (see also *concrete class*)
algorithm - orderly sequence of steps describing unambiguously how to solve a problem
argument - a value supplied with a message to make possible its execution
Browser - Smalltalk tool that allows viewing, creation, and editing of Smalltalk classes
class - an object factory, an object that manufactures instances according to a blueprint stored in its definition (*concrete class*); alternatively, a class factoring out shared properties of concrete classes (*abstract class*)
class message - message understood by a class (see also *instance message*)
class variable - holder of a state property of a class (see also *instance variable*)
compiler - a computer program that converts code written in a programming language to directly executable CPU level
conversation - a sequence of events that occur during the execution of a *scenario*
CRC card - Class-Responsibilities-Collaborator card; a card with the name of a class, its description, a list of its responsibilities, and a list of collaborators needed to fulfill them
class definition - formal statement of class properties in the selected programming language
class hierarchy - arrangement of classes in a subclass - superclass relationship
collaborator – an object whose functionality is needed to implement a class responsibility
concrete class - object factory, class designed to be instantiated (see also *abstract class*)
delegation - forwarding execution of tasks that are outside the realm of responsibilities of the receiver to objects equipped to deal with them
dynamic typing - leaving the decision as to which method definition will be used to execute a message until run time
encapsulation - gathering of state and functionality into an object
exception – an illegal event such as attempt to divide by zero or to execute a message not understood by the receiver
functionality - collection of services available from an object; the set of messages that it understands
information hiding - making state information accessible only through explicitly defined messages
inheritance - access to state variables and functionality declared in a superclass
instance - object created by sending a creation message to a class
instance message - message understood by an instance of a class (see also *class message*)
instance variable - holder of a state property of a class instance (see also *class variable*)
instantiation - the act of creating an instance
message - request for service sent by one object to another; each Smalltalk message returns an object
message interface - the collection of all messages that an object understands
message resolution – the process of finding the definition of a message prior to its execution
method - definition of a message; message is what you send, method is a formal description of how it works
multiple inheritance - inheritance where each class may have more than one direct superclass (see also *single inheritance*)
object - an entity with state and functionality
<operate> button - the middle button of a three-button mouse – displays numerous commands including text editing commands such as *copy*, and code execution commands such as *do it*
<operate> menu - pop up menu of the <operate> button
polymorphism - ability to define a method with the same name and the same purpose in more than one class
protocol - a grouping of related methods in a class definition
receiver - object to which a message is sent
responsibility - ability to perform a service
scenario - task typically executed in a given problem setting
<select> button - the leftmost button of the mouse; used to select text and other screen items
single inheritance - inheritance where each class may have only one direct superclass; the norm in Smalltalk (see also *multiple inheritance*)
static typing - deciding which method definition will be used to execute a message during compilation (see also *dynamic typing*)

subclass - class at a lower level but on the same branch of the hierarchy tree
superclass - class at a higher level but on the same branch of the hierarchy tree
variable – a named reference to an object
widget - a window component such as a button, a label, or a scrollable selection list