Chapter 5 - Numbers

Overview

VisualWorks library contains many classes representing numbers of various kinds, mainly because computers internally represent and process different kinds of numbers differently. In addition, computer programs often require facilities that go beyond basic hardware capabilities and Smalltalk number classes satisfy these additional needs.

Although the main use of numbers is for calculations, numbers are also used to implement iteration. This includes execution of a block of statements a fixed number of times or over a range of numbers with given start and end values. Several methods implementing these very important functions are included in the library.

5.1. Numbers

All Smalltalk number classes are defined in the Magnitude - ArithmeticValue part of the class hierarchy. Its structure is as follows:

- Object ()
  - Magnitude ()
    - ArithmeticValue ()
      - Complex ('real' 'imaginary')
      - Number ()
        - *FixedPoint* ('numerator' 'denominator' 'scale')
        - **Fraction** ('numerator' 'denominator')
        - Integer ()
          - LargeInteger ()
            - **LargeNegativeInteger** ()
            - **LargePositiveInteger** ()
          - **SmallInteger** ()
        - LimitedPrecisionReal ()
          - *Double* ()
          - **Float** ()
        - MetaNumeric ()
          - Infinitesimal ('negative')
          - Infinity ('negative')
          - NotANumber ()
          - SomeNumber ('value')
      - **Point** ('x' 'y')

In this diagram, the most important classes are **boldfaced**, less important classes are *italicized*, and classes rarely encountered are in regular font.

The hierarchy starts with the abstract class Magnitude which gathers the general properties based on comparison[1]. Subclasses of Magnitude include mainly number classes but also other classes whose instances can be compared such as Date, Time, and Character. Numeric subclasses of Magnitude are collected under ArithmeticValue, another abstract class, which gathers methods shared by all numbers such as addition, subtraction, and absolute value. Concrete definitions of most of these methods are left to concrete subclasses but some are defined here, for example changing the sign of an instance by methods abs and negated.

Some of the subclasses of ArithmeticValue, such as Complex, are little used and Complex and MetaNumeric are not even included in the standard library and must be filed in. All the frequently used kinds of numbers are subclasses of the abstract class Number which is responsible for scalar entities -

[1] Note that although most classes that understand comparison are gathered under Magnitude, some important classes such as strings are located under a different class.

numbers that have magnitude but cannot be used as vectors - indicators of direction or coordinates of points in space. (Instances of Complex and Point are related to vectors.) Number defines methods such as division and mathematical functions such as sin and cos, mostly on the basis of conversion to floating-point and delegation to floating-point number classes. A large conversion protocol is included which allows conversion of almost any kind of number to almost any other kind.

The concrete subclasses of Number are LargeNegativeInteger, LargePositiveInteger, and SmallInteger (all of them numbers without a decimal point), Fraction (based on integer instance variables numerator and denominator), FixedPoint (with instance variables numerator, denominator, and scale), Double and Float (numbers with decimal point and fractional part), and special classes Infinitesimal, Infinity, NotANumber, and SomeNumber - all subclasses of MetaNumeric. If you are interested in numbers, the implementation of some of the methods provides rich material for the study of very interesting algorithms.

Before introducing the most important properties of number classes in detail, we will now briefly outline their purpose and functionality.

Integers

Integers are numbers without a decimal point such as 13, -45700089, and 0. Computers implement integers and integer arithmetic differently from other kinds of numbers, and the number of bits that they use to store integers determines the maximum size that they can handle in hardware. In VisualWorks Smalltalk, integers directly processed by computer hardware are implemented by class SmallInteger and their maximum and minimum values can be obtained as follows:

```
SmallInteger minVal                "Returns, for example, -536870912"
SmallInteger maxVal                "Returns, for example,  536870911"
```

Although the exact value of the limit depends on your computer's CPU, it is obvious that the range of SmallInteger is not so small after all. In fact, the name SmallInteger is due to the fact that Smalltalk also implements essentially unlimited integers using two 'large integer' classes. One special feature of SmallInteger objects is that they are among the few objects that are not accessed by pointers: Whereas the most other objects are represented by the memory address of their detailed description (a pointer to the internal representation), SmallInteger objects are directly represented by the binary code of the integer value itself; this makes their access very efficient.

The idea behind introducing special classes for large integers is that if your computer has a lot of memory, there is no reason why the available bytes could not be used to store a representation of very large magnitudes, in principle an unlimited sequence of digits. Unlike SmallInteger objects, such numbers cannot be processed by a single CPU instruction and require special treatment which is implemented by classes LargeNegativeInteger and LargePositiveInteger, both subclasses of LargeInteger. These 'large integers' allow arithmetic without any concern about magnitudes. As an example, LargePositiveInteger allows you to calculate the factorial of 200, in other words, the number obtained by multiplying 200*199*198*197*...*3*2*1. The expression

```
200 factorial
```

returns

```
788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113462857270763317237396988943922445621451664240254033291864131227428294853277524242407573903240321257405579568660226031904170324062351700858796178922222789623703897374720000000000000000000000000000000000000000000000000
```

This is a result that few other languages would let you obtain as easily. Although there are not many applications that need to calculate the factorial of 200, unlimited numbers are very useful in business and other applications. (Unfortunately, national debts easily go beyond the limits of SmallInteger.) There is, of course, a price for this computational power and this price is that calculations with LargeNumber take considerably longer than calculations with SmallInteger. We will see how much longer later, when we show how to measure the time that it takes to execute a statement.

The unlimited range of Smalltalk numbers is a valuable feature but if you had to treat 'small' and 'large' integers differently and keep track of the boundaries between them during arithmetic, the advantage of having unlimited integers would be small. This is why Smalltalk performs conversion between the three classes automatically and you don't even have to know that the three classes exist; all you need to understand is the concept of an integer. Technically speaking, implementation of integers is *transparent* to the programmer.

Floating-point numbers

Floating-point numbers are numbers with a decimal point such as 3.14, -789000000.123, 0.000000341, 0.0 or 1.33185e17 (1.33185 * 10^{17}). Like most other languages, Smalltalk has two floating-point representations implemented by two floating-point classes - Float ('single precision') and Double ('double precision'). The difference between them is how many bytes they use and how much precision and what range they provide. Unlike the automatic conversion between SmallInteger and LargeInteger, conversion between Float and Double is not automatic and requires conversion messages asFloat and asDouble. The comment of Double explains the principle of the class as follows:

Instances of class Double represent floating-point numbers in IEEE 64-bit format.
These floating-point numbers are good for *about 14 or 15 digits of accuracy*, and the range is between plus and minus 10^307.
Here are some valid floating-point examples:
    8.0   13.3   0.3   2.5d6   1.27d-30 1.27d-31 -12.987654d12
The format consists mainly of no imbedded blanks, little d for tens power, and a *digit on both sides of the decimal point*.

The bizarre phrase '*about 14 or 15 digits of accuracy*' reflects the fact that internal representation of floating-point numbers is binary and conversion of decimal values to binary usually results in loss of accuracy.

Representation using letter d is mainly for very large and very small magnitudes and the letter separates the magnitude part from the exponent part. As an example

1.27d-30 = 1.27 * 10^{-30} = 0.00000000000000000000000000000127
-12.987654d12 = -12.987654 * 10^{12} = 12987654000000.0
127d3 = 1.27 * 10^{5} = 127000.0

Note that .123d is illegal because the decimal point must be surrounded by digits.

When you *inspect* a Double number, such as the Double class constant Pi (returned by class message Double pi) the inspector shows that Double numbers use eight bytes which agrees with the comment which states that 64 bits are used. The actual bytes are normally of no interest to the programmer.

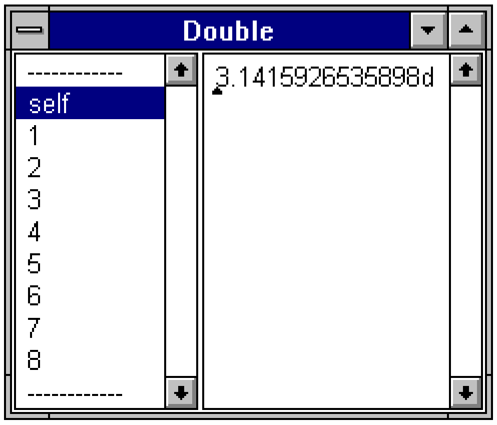


Figure 5.1. Inspector on Double class constant Pi obtained by message Double pi.

The comment for class Float is similar to the comment for Double, the main difference being the range and accuracy. Note also that Float representation uses letter e instead of letter d forndicate the exponent part of the code:

Instances of the class Float represent floating-point numbers in platform-dependent short float format.
These floating-point numbers are good for *about 8 or 9 digits of accuracy*, and the *range is between plus and minus 10^38*.
Here are some valid floating-point examples:

```
        8.0   13.3   0.3   2.5e6   1.27e-30  1.27e-31 -12.987654e12
```

The format consists mainly of no imbedded blanks, little e for tens power, and a *digit on both sides of the decimal point*.

An important point about floating-point numbers is that the displayed value is not exactly what is stored in memory because of the incompatibility of the internal binary representation and the decimal representation on the screen. The number of digits displayed on the screen is controlled by the value returned by the message defaultNumberOfDigits:

```
Float defaultNumberOfDigits
```

returns 6, and

```
Double defaultNumberOfDigits
```

returns 14 which explains the Inspector in Figure 5.1. Both Float and Double numbers are thus displayed with less than their full accuracy. You can change this constant if you wish by changing the definition of defaultNumberOfDigits but no matter how many digits you print, the internal representation is fixed and changing defaultNumberOfDigits will only change the display but not the precision of arithmetic.

It is useful to note that the library contains a class that converts numbers to formats other than the default. This class is called PrintConverter and the display method is print: number formattedBy: string. It is used as follows:

```
PrintConverter  print: 1000 * Float pi formattedBy: '####,###.##'            "Returns   3,141.59."
```

but other formats are also available.

Fractions

Instances of class Fraction represent values such as -3/5 or 14/27. Each fraction is internally represented by two integer objects - an integer numerator and an integer denominator - so that there is no conversion and no loss of accuracy in arithmetic. Arithmetic on Fractions uses the well-known principles of combining numerators and denominators. As an example, the Fraction method for addition is defined essentially as follows[2]:

```
+ aFraction
        ^Fraction numerator:   numerator * aFraction denominator + (aFraction numerator * denominator)
                denominator: denominator * aFraction denominator
```

The body first calculates the value of the numerator and denominator, and then creates a new fraction with these numerator and denominator values. The process requires the creation of a new instance, three multiplications (notoriously time-consuming), and one addition; it is thus clear that addition of Fraction objects will take much longer that addition of Integer objects. The redeeming advantage of

[2] In reality, the definition uses double dispatching which is explained later in this section.

fractional arithmetic is that it fully preserves accuracy, which is not true for floating-point representation which is inherently inaccurate.

Fixed point

According to the comment of class FixedPoint

FixedPoint numbers represent 'business' numbers - numbers with arbitrary precision before the decimal point, but limited precision after the decimal point. A prime example of their use is in expressing currency values, which are always rounded off to the nearest hundredth, but which could easily have more digits than a Float (or possibly even a Double) could accurately express.

Main lessons learned:

- Smalltalk's rich hierarchy of number classes is rooted in the abstract classes Magnitude, ArithmeticValue, and Number.
- The most important types of numbers are integers, floating-point numbers, and fractions. Each uses a different internal representation and performs arithmetic differently. Some types of numbers are directly implemented in hardware, others are not, and this results in different ranges, accuracy, speed of arithmetic, and memory requirements.
- Smalltalk integers consist of 'small' integers (directly implemented in hardware) and 'large' integers for essentially unlimited magnitudes. Smalltalk automatically converts between the different forms when this is required by the calculation.
- Floating-point numbers represent values with a decimal point. They are implemented by classes Float and Double and the difference between them is in accuracy and range. Like small integers, floating-point arithmetic is directly implemented by hardware.
- Floating-point numbers are inaccurate binary representations of their decimal counterparts.
- Class Fraction provides uncompromised accuracy based on integers at the cost of very slow execution.
- Class FixedPoint is an alternative of Fraction intended for decimal data.

Exercises

1. List all concrete method definitions in ArithmeticValue and Number.
2. Methods fully defined in abstract classes usually depend on methods defined in concrete subclasses. These methods, left as *subclass responsibility*, should be listed in the class comment. Read the comment of Magnitude, list methods that must be defined in subclasses, and methods that this "buys you", in other words, methods that need only be defined in Magnitude and will work for any or almost any subclass.
3. Write a half page summary of class MetaNumeric.
4. Write a half page summary of class Complex.
5. If you are familiar with the principles of floating-point representation, examine how class Float represents floating-point numbers.
6. How many digits does 2000 factorial have? (Hint: Message size returns the number of characters in a String.)
7. To gain insight into floating-point numbers, step through the execution of the following code fragment:
```
self halt.
Transcript show: 3.1 printString
```
8. Write a short description of the most useful features of class PrintConverter.
9. Class Magnitude gathers shared behavior of all objects that can be compared. Most of its subclasses are numbers but another very important subclass is Character which is mainly used for the representation, conversion, and testing of characters including letters, digits, and punctuation symbols. Read the comment of Character and list the most important methods in the *converting*, *testing*, and *accessing untypeable characters* protocols.
10. Characters are a subclass of Magnitude because they can be compared. List all useful methods that they inherit.

11. Class ArithmeticValue disables the new message. This means that numbers cannot be created with new - except for class Complex. Examine and explain how class Complex gets around this limitation.

5.2 Operations on numbers

Most number operations can be performed on all types of numbers with the usual limitations such as no division by 0, logarithms and square roots of positive numbers only, and so on. We will not list all the available messages because few applications require more than basic arithmetic. Besides, number messages are quite self-explanatory and don't require any comments; as an example, if you know what arcsin is, you will not have any problem finding it in the browser and using it.

We must regretfully admit that Smalltalk is not ideal for large amounts of mathematical calculations (especially on floating-point numbers) because its very principle – treating everything as objects - requires overhead that slows arithmetic down. If you want to write a program to calculate the state of the Universe in the first 30 nanoseconds after the Big Bang, don't use Smalltalk beyond the first prototype or implement the extensive calculations in another language. You may be better of using C or, better still, assembly language. For extensive floating-point calculations, benchmark measurements give a wide range of comparisons but C is probably about five times faster on the average than unaided Smalltalk.

After this introduction we will now examine the most common arithmetic, mathematical, and conversion messages.

Arithmetic messages

All numbers support all arithmetic operations including addition, subtraction, multiplication, and division.

+ - *	work in the usual way on all combinations of numbers
/	produces a *Fraction* when both the receiver and the argument are integers, Float if floating-point numbers are involved
quo:	calculates the integer quotient of division with truncation towards zero
rem:	calculates the remainder of division in terms of quo:

Examples with integers:

3 / 5	"Returns fraction 3 / 5."
17 quo: 3	"Returns 5."
17 rem: 3	"Returns 2 because 5*3 + 2 = 17."
-17 quo: 3	"Returns -5."
-17 rem: 3	"Returns -2 because -5*3 + (-2) = -17."

Examples with floating point numbers:

3 / 5	"Returns 0.6."
1.7 quo: 0.3	"Returns 5."
1.7 rem: 0.3	"Returns 0.2 because 5*0.3 + 0.2 = 1.7."
-1.7 quo: 0.3	"Returns -5."
-1.7 rem: 0.3	"Returns -0.2 because -5*0.3 + (-0.2) = -1.7."
rounded	"Returns the integer nearest the receiver."
truncated	"Returns the integer nearest the receiver toward zero."
1.7 truncated	"Returns 1 - receiver stripped of its decimal part."
1.4 truncated	"Returns 1."
1.7 rounded	"Returns 2 - the nearest integer - compare with truncation."
1.4 rounded	"Returns 1."
1.5 rounded	"Returns 2."
-1.7 truncated	"Returns -1."
-1.4 truncated	"Returns -1."
-1.5 rounded	"Returns -2."
13 even	"Returns false - number 13 is not even."

Mathematical messages

```
Float pi/2 sin              "Returns 3.45497 - not sin(π/2) = 1 as we expected. Why?"
(Float pi/2) sin            "Returns 1.0 - this is what we had in mind."
(Float pi/2) cos            "Returns -4.37114e-8 or -0.000000043711. Close to 0 but not exact."
(Float pi/4) tan            "Returns 1.0"
10 log                      "Returns 1.0 because 10^1 = 10"
10 raisedTo: 0.5            "Returns 3.16228, the square root of 10"
10 sqrt                     "Returns 3.16228 - same as the previous message, as expected."
```

Conversion messages

```
3.14 asInteger              "Returns 3, same as message truncated."
17 asFloat                  "Returns 17.0"
Float pi asRational         "Returns  (918253/292289) - approximates π as a fraction."
Double pi asRational        "Returns (130161051938194/41431549628009)"
180 degreesToRadians        "Returns 3.14159 because 180 degrees is the same as π."
```

Other

```
2 max: 7                    "Returns 7, the larger of the two numbers"
2 min: 7                    "Returns 2, the smaller of the two numbers"
```

Example 1: Accuracy of conversion between floating-point numbers and fractions

Method asRational converts numbers to instances of Fraction. The message is understood by almost all numbers and defined in several classes but the only interesting definition is in class LimitedPrecisionReal; this definition is inherited by Float and Double. Its principle is conversion to continued fractions, numerical objects of the following kind:

$$\cfrac{375}{1 + \cfrac{43}{1 + \cfrac{13}{1 + \dots}}}$$

To check how accurate the conversion is, we converted a floating-point number to a Fraction and then back to a floating-point number, and compared this with the original number as follows:

```
12345678.1476 asRational asFloat - 12345678.1476          "Returns 0.0"
12345678.76 asRational asFloat = 12345678.76              "Returns true."
31.76 asRational asFloat - 31.76                          "Returns 1.90735e-6"
31.76 asRational asFloat = 31.76                          "Returns false."
```

The calculation appears to be very accurate but we will make a more thorough test later.

Example 2: Defining a new arithmetic method
Problem: Incrementing a number by 1 is so common that it might be useful to have a special method to implement it as a unary message. The method could be used as in

```
| n |
n := 15.
some calculations here
n increment
```

Solution: The problem is simple and the solution is to take the receiver, add 1 to it, and return the result as in

```
increment
"Increment receiver by 1."
	^self + 1
```

The method should be useful for any number and we will thus define it in class Number. Executing

```
15 increment
```

returns 16 and the method thus appears to work. However, executing

```
| n |
n := 15.
n increment.
n
```

with *print it* returns 15 as the value of n and this does not appear to be correct. What is happening? The receiver of

```
n increment
```

is the object bound to n, in other words 15. The method takes the receiver, adds 1 to it, and returns 16. However, the binding of n to 15 is not affected and the value of n thus remains 15. Our fundamental goal thus has not been achieved but we learned an important lesson.

Main lessons learned:

- VisualWorks library includes many number classes, mainly because different kinds of numbers require different internal representation and different hardware implementation.
- Smalltalk is not ideal for problems involving large amounts of arithmetic, particularly when it involves floating-point numbers.
- Classes such as LargeInteger and Fraction provide extra range and accuracy found in few other languages. The cost is long processing time.

Exercises

1. Test the following messages in short examples:
   a. asFloat (in particular in class Fraction)
   b. asRational
   c. fractionPart
   d. rounded
   e. truncated
   f. degreesToRadians
   g. radiansToDegrees
   h. squared
2. Compare the results of Float pi asRational and Double pi asRational.
3. The result of division of integers, is automatically reduced. Test this by executing 4/6 with *inspect*.

5.3 Implementation of binary arithmetic - double dispatching and primitives

Smalltalk arithmetic is based on double dispatching. We will now explain this important principle and refer you to an alternative description in method aboutDoubleDispatching in the class protocol documentation in class ArithmeticValue.

Assume that you want to execute the message 3.14 + (5/8). The two operands are different kinds of numbers must first be converted to the same class. Since Fraction arithmetic is based on integers and since 3.14 cannot be converted to an integer without loss of accuracy, the Fraction is converted to Float and the two numbers can then be added using floating-point arithmetic.
In general, 3.14 + x with an arbitrary argument could be implemented as follows:

1. Check if the argument is a Float. If it is, do Float addition and exit.
2. Check if the argument is a Double. If it is, convert the receiver to Double, do Double arithmetic and exit.
3. Check if the argument is a SmallInteger. If it is, convert it to Float, do Float arithmetic and exit.
4. Check if the argument is a Fraction. If it is, convert it to Float, do Float arithmetic and exit.
5. Check if the number is a LargePositiveInteger. If it is, try convert it to Float, do Float arithmetic, and exit.
6. And so on and on, a long chain of tests, conversions, and operations.

If the class of the argument is somewhere at the end of this chain, it will take a long time before we even know what to do. For larger amount of arithmetic, this would be a serious drawback because each test adds overhead. The code is also too complicated. And if we add a new number class, we will have to change the definition of all arithmetic messages in all existing number classes to deal with this new type of argument. All in all, this is an unattractive approach.

We had a similar problem before when we considered drawing different kinds of geometric objects and we solved it using polymorphism. To eliminate tests, we relied on the receiver to do ‘the right thing’. Unfortunately, our problem is different - the reason for testing is the argument, not the receiver. The problem is thus ‘turned around the operation’ and we cannot use polymorphism. But maybe the solution can be based on turning the problem around – exchanging the receiver and the argument - and this is exactly what double dispatching does.

Smalltalk implementation of Float addition method first checks whether it can do addition using CPU code. If it can (which happens if the argument belongs to the same or a closely related class), it does, and execution of the method is finished. If it cannot, the code *exchanges* the receiver and the argument (in our example3.14 + (5/8), the receiver is now Fraction 5/8) and sends a message to it (a Fraction object), indicating that the argument is a Float. The definition of the method in Fraction takes this hint, converts the receiver 5/8 to a Float (without any testing) and sends the + message (with the fraction converted to float) to the Float receiver. Float now executes + directly because both numbers are now Float objects (Figure 5.2). Message + is thus dispatched once to the original receiver and then, if necessary, again - to the argument. The result is that the arithmetic operation requires at most two extra message sends but no explicit checks of the class of the argument. Since message sends require only a minimal number of CPU cycles, this approach is more efficient - and the code much neater. And if we add a new class, we only have to add new double dispatching messages and none of the existing messages is affected.

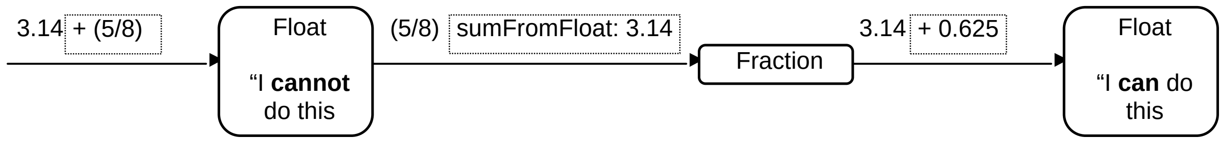


Figure 5.2. Execution of 3.14 + (5/8) uses double dispatching.

We can now examine the definition of + in class Float which is as follows:

```
+ aNumber
        "Answer a Float that is the result of adding the receiver to the argument.
        The primitive fails if it cannot coerce the argument to a Float."

        <primitive: 41>
        ^aNumber sumFromFloat: self
```

The code means that if <primitive: 41> fails to convert the two numbers to a common form and add them (ignore the concept of a primitive for a moment) the *argument* aNumber becomes the receiver of the message sumFromFloat:. In our example, aNumber is a Fraction, the message thus goes to class Fraction, and the definition of sumFromFloat: in class Fraction is

```
sumFromFloat: aFloat

        ^aFloat + self asFloat
```

In our case, self is 5/8, self asFloat is 0. 625, and the method thus sends + 0.625 to Float. This is the same + method as above but this time, the argument is a Float and the primitive executes and returns the sum. The operation is finally completed. We suggest that you test how this works using the Debugger on

```
self halt.
3.14 + (5/8)
```

The deinition above introduced the concept of a *primitive*, an expression written in a special syntax that does not indicate any message sends. A primitive is not implemented in Smalltalk but in a lower level language such as C or assembly language. In our example, primitive 41 executes a machine language program that adds two floating-point numbers if both are floating-point numbers or if the program knows how to convert the second number to float. There are many other primitives, such as primitive 1 which adds two small integers.

When a method contains a primitive, the primitive is always the first statement. If it fails to provide the desired result, execution continues to the next statement in the definition. The best way to think of a primitive is as a machine level instruction of some fictitious CPU. As an example, think of primitive 1 used in the definition of + in SmallInteger as 'machine instruction number 1' which checks whether the argument is a SmallInteger, performs addition if it is or if it can be easily converted, and exits with some special result if it is not.

You will now probably say: 'Aha, after all, Smalltalk is not all that pure. In the end, it must descend to the level of the CPU and work just like any other language'. And you are quite right! Every programming language must, in the end, be translated to machine level and the binary codes must be executed by the CPU. But this is true for every language.

One last question concerning primitives remains: Where is the code that executes primitives? The answer is that the implementation of primitives is included in the Smalltalk *Virtual Machine* (VM), a program that provides an interface between Smalltalk and the CPU. The VM is thus essential for Smalltalk execution and an application written in Smalltalk requires two parts to run: The Virtual Machine program (called, for example, oe.exe - derived from Object Engine, another name for VM) and the image file containing data related mainly to the class library and stored in a .im file. Creating a stand-alone Smalltalk application thus requires creating an image file containing information about the application's classes, and a copy of the VM file. If you are developing the library, you also need a changes file .ch which contains all the changes that you made to the library, and the .sou file which contains the source code in textual form. For proper operation, the .im, .ch, and .sou files must be synchronized which automatically happens if you use the versions saved by the *save* or *save then exit* commands.

The concept of a Virtual Machine has recently attracted general attention with the emergence of the Java language which is also based on a virtual machine. To make Java portable (executable on different CPUs), Java programs are translated into *bytecodes* which are identical across all CPUs, and executed by virtual machines which are tailored to specific CPUs. Smalltalk uses the same principle.

Main lessons learned:

- Smalltalk arithmetic uses the concepts of double dispatching and primitives.
- Double dispatching means re-sending (re-dispatching) a one-argument message to the argument, using the original receiver as the argument.
- Double dispatching eliminates tests because the dispatched message implies the type of the argument.
- The main use of double dispatching is in arithmetic but the principle has general validity.
- A primitive is an operation implemented at 'low level', possibly directly in machine code. Each primitive has a number and can be thought of as a machine code of a fictitious CPU.
- Primitive functions use special syntax of the form <primitive: 1>. If a method contains a primitive, the primitive is always the first statement; the remaining code is executed only if the primitive fails.
- Primitives are implemented via a virtual machine.
- A virtual machine (also called object engine) is a program that interfaces between Smalltalk and the CPU.
- The concept of a virtual machine makes it possible to compile code into one form shared across all platforms. The elements of this representation are called bytecodes. They can be executed on any machine that has a virtual machine for processing them.
- A completed standalone Smalltalk application requires a virtual machine and an image file.

Exercises

1. Find and explain the + and / methods in the following classes:
   a. SmallInteger
   b. Double
   c. Fraction
2. Extend the principle of double dispatching from one argument to two arguments.

5.4 Using numbers for iteration - 'repeat n times'

In some problems, we need to evaluate a block of statements a known number of times. For example, we might want to flash a message on the screen three times, or execute a block 1,000 times to get an accurate measure of its execution time. Although all forms of repetition can be implemented with whileTrue:, iteration based on counting is so common that Smalltalk provides specialized messages which are more convenient to use. We will present one of them in this section and the remaining ones later.

Repeating a block a fixed number of times – timesRepeat: aBlock

If you want to flash a label in a window three times, you can use the following construct:

```
3 timesRepeat:	[“statements to display the label”
		“statements to create a short delay to keep the label displayed”
		“statements to hide the label”
		“statements to create a short delay to keep the label hidden”]
```

Method timesRepeat: is defined in the abstract class Integer because its receiver may be any kind of integer number, and its operation is described by the flowchart in Figure 5.3.

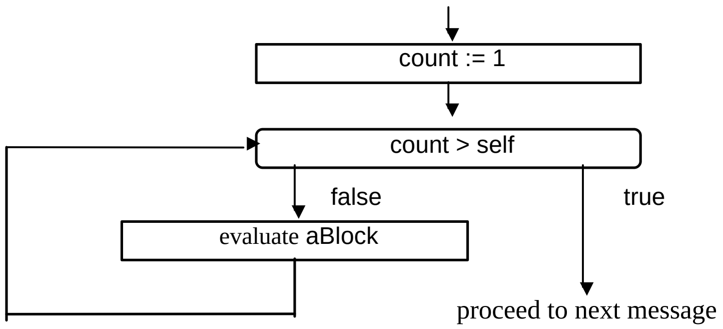


Figure 5.3. Execution of n timesRepeat: aBlock.

Example 1. Ringing the bell

All computers have a built-in beeper, conventionally referred to as the ‘bell’. To beep the bell five times, execute

```
5 timesRepeat: [Screen default ringBell]
```

Class Screen is an interesting part of the library which knows various things about your computer, such as how many pixels its screen has, how to ‘ring the bell’, and how to perform interesting graphics operations. To access its single instance, send it the default message as above.

Example 2. Testing the quality of Smalltalk’s random number generator

We have already mentioned that one of the motivations for object-oriented programming was simulation of problems such as queues in a bank. Problems of this kind are not deterministic because customers arrive at unpredictable times, and simulation thus require tools for generation of random numbers. To satisfy this need, Smalltalk contains a class called Random.

Sending message new to class Random returns a new random number generator, a manufacturer of random numbers. Every time you send message next to the random number generator (Figure 5.4), it returns a floating-point number between 0 and 1. When you examine a sufficiently large collection of these random numbers, they should be distributed uniformly. In other words, if I1 and I2 are two arbitrary but equally long intervals between 0 and 1 and generate many random numbers, the number of samples falling into I1 should be about the same as the number of numbers in I2. Although most applications require random numbers distributed in some non-uniform way, any distribution can be generated from uniform distribution and a uniform random number generator is thus a sufficient basis for all probabilistic simulations.

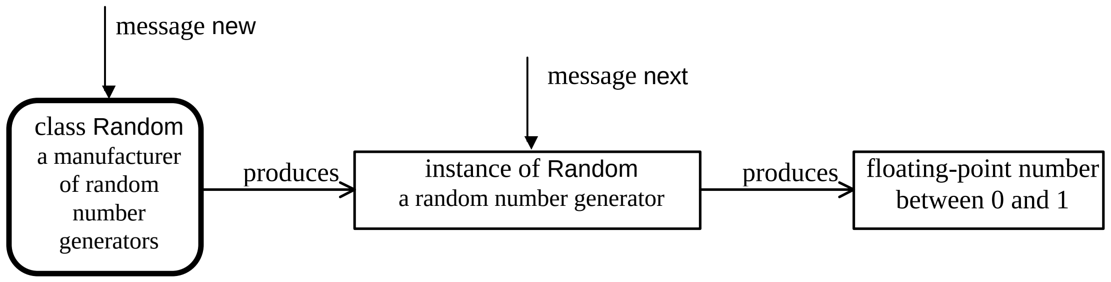

message new
message next
class Random
a manufacturer of random number generators
produces
instance of Random
a random number generator
produces
floating-point number between 0 and 1

Figure 5.4. The use of class Random.

In this example, we will test how well Random accomplishes its purpose. We will select an interval I between 0 and 1, generate 10,000 random numbers, and count how many of them fall into I. We will then evaluate whether this number is appropriate for the width of the interval. As an example, if I is the interval from 0 to 0.1, its width is 1/10 of the 0 to 1 interval and the number of random numbers falling into I should be about 1/10 of the total number of generated samples. The test code is quite simple:

```
| count generator |
“Initialize count and create a random number generator.”
count := 0.
generator := Random new.
“Run the test.”
10000 timesRepeat: [ “Generate a random number and test whether it falls into our interval. If so,
                        increment the count.”
                        ((generator next) between: 0 and 0.1)
                                ifTrue: [count := count + 1].
Transcript clear; show: ‘The number of samples falling between 0 and 0.1 is ‘, count printString
```

When we executed this code, we got

```
The number of samples falling between 0 and 0.1 is 987
```

This is quite nice - the perfect answer would be 1,000 but we cannot expect that it will be achieved because the numbers are random. However, what if we were just lucky? Let’s modify the program to repeat the whole test 10 times as follows:

```
| count generator |
“Initialize count and create a random number generator”
count := 0.
generator := Random new.
Transcript clear.
10 timesRepeat: “Do the whole test 10 times.”
        [count := 0.
        10000 timesRepeat: ‘How many of the 10,000 numbers fall into our interval?.”
                [((generator next) between: 0 and: 0.1) ifTrue: [count := count + 1]].
        Transcript show: ‘The number of samples falling between 0 and 0.1 is ', count printString; cr]
```

The structure of this code fragment is a *nested loop* - one loop (10000 timesRepat:) inside another (10 timesRepeat:). When Smalltalk executes the inner loop 1000 times, it checks whether the outer loop has been executed enough times. If not, it repeats the inner loop 1000 times again, checks again, and so on until the outer loop is executed 10 times (Figure 5.5).

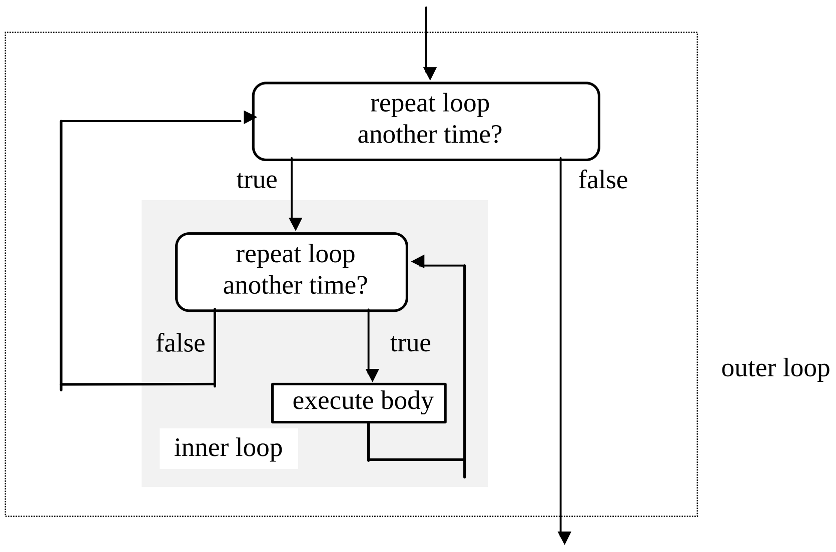


Figure 5.5. Nested loops.

When we executed this program, we got the following output:

```
The number of samples falling between 0 and 0.1 is 1001
The number of samples falling between 0 and 0.1 is 1006
The number of samples falling between 0 and 0.1 is 993
The number of samples falling between 0 and 0.1 is 1013
The number of samples falling between 0 and 0.1 is 976
The number of samples falling between 0 and 0.1 is 993
The number of samples falling between 0 and 0.1 is 1004
The number of samples falling between 0 and 0.1 is 1023
The number of samples falling between 0 and 0.1 is 1000
The number of samples falling between 0 and 0.1 is 1006
```

which is quite satisfactory. By the way, note that the numbers are different in each run. This is to be expected because the numbers are random.

Example 3. Definition of timesRepeat:
To conclude this section, let us examine the definition of timesRepeat:.

```
timesRepeat: aBlock
"Evaluate the argument, aBlock, the number of times represented by the receiver."
        | count |
        count := 1.
        [count <= self]
                whileTrue: [aBlock value.
                          [count := count + 1]
```

The implementation is based on whileTrue: and so timesRepeat: is not strictly necessary because we can get the same effect by using whileTrue: directly. However, the solution with repeatTimes: is much neater. When you compare

```
5 timesRepeat: [Transcript show: ‘Time to go home!’; cr]
```

with

```
count := 1.
[count > 5] whileTrue: [Transcript show: ‘Time to go home!’; cr. count := count + 1]
```

you will agree that the second form is longer (requiring more typing), more difficult to understand, and more error prone because we must remember to keep track of the count. (If we forget, we will get an infinite loop). Message timesRepeat: thus makes programming easier and a sufficient justification why this and many other apparently redundant messages are in the Smalltalk library.

Main lessons learned:

- To repeat a block of statements a known number of times, use message timesRepeat:.
- An iteration enclosed inside another iteration is called a nested iteration.
- If the Smalltalk library contains a message that does exactly what you need, use it. This will save you time re-inventing the wheel and making mistakes.

Exercises

1. Truly random numbers should never repeat the previously generated sequence but random number generators are based on arithmetic and produce repetitive sequences (they are thus properly called pseudo-random). The length of the sequence is called its period and a good pseudo-random number generator should thus have a large period. Is the period of Random large enough?
2. Class Screen defines various interesting methods. Explore and execute the following ones: resolution, contentsFromUser, bounds, dragShape:..., and zoom:to:duration:. Some of them contain example code.
3. Write a code fragment to print
   a. one line containing ten * symbols (simple loop)
   b. five lines, each containing ten * symbols (nested loops)
4. Modify the test of Random as follows:
   a. Allow the user to specify how many random numbers should be generated.
   b. Allow the user to specify the start and end points of the interval.
   c. Allow the user to specify how many times to repeat the test.
   d. Calculate the average number of samples falling into the interval over all tests.
5. Convert the random generator test into a test method in class protocol testing in class Random.
6. Some people think that the timesRepeat: aBlock message is not natural and that aBlock repeatTimes: anInteger as in aBlock repeatTimes: 17 would be better. Define such a method and test it.
7. Write a summary of the main features of class Random and find all references to it in the library.

5.5 Repeating a block for all numbers between a start and a stop value

Method repeatTimes: is rarely used because it does not happen very often that you want to do exactly the same thing several times. Much more often, you need to do something with a value moving over a range - such as when you want to print a table of factorials from 1 to 100. The message that performs this task is to:do: and its basic use is as follows:

```
startNumber to: endNumber: do: oneArgumentBlock
```

where oneArgumentBlock is a block with one *internal argument*. The block is repeatedly executed with all successive values of the internal argument from startNumber to endNumber in increments of 1 (Figure 5.6).

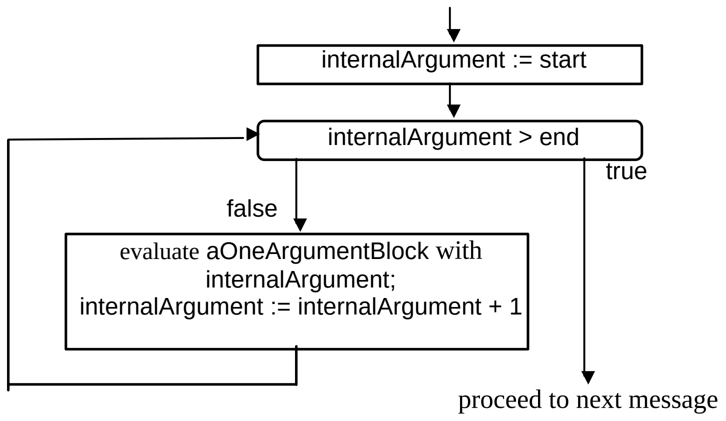

internalArgument := start
internalArgument > end
true
false
evaluate aOneArgumentBlock with internalArgument;
internalArgument := internalArgument + 1
proceed to next message

Figure 5.6. Execution of start to: end: do: aOneArgumentBlock.

Example 1: Print all numbers from 1 to 5 with their squares and cubes in the Transcript
Solution: Repeat the calculation and output for each value of the argument as it ranges through the interval:

```
Transcript clear.
1 to: 5 do:
        [:arg | "arg is internal block argument. During the execution, it will assume values 1, 2, ..., 5."
                Transcript show: arg printString; tab;
                        show: arg squared printString; tab;
                        show: (arg squared * arg) printString; cr]
```

The output is

```
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
```

Blocks and arguments obey the following rules:

- A block may have any number of arguments. We have seen blocks without arguments such as the block used in ifTrue:, a method with one argument (to:do:) and we will see methods with more arguments later.
- All arguments defined for a block are listed immediately behind the opening bracket, each preceded by a colon. The whole list is followed by a vertical bar. A block with three arguments, for example, would be used as follows:

  ```
  [:arg1 :arg2 :arg3 | "Body of block."]
  ```
- The name of block arguments must not clash with identifiers already declared in a surrounding scope. As an example, if a one-argument block is used in a method with a local variable or keyword argument called number, the block argument must not be called number. Similarly, if the block is nested inside another block, their argument names must not clash. The same applies to a conflict with the names of instance variables of the class. If any of these scoping rules are violated, Smalltalk displays a warning message as in Figure 5.7.

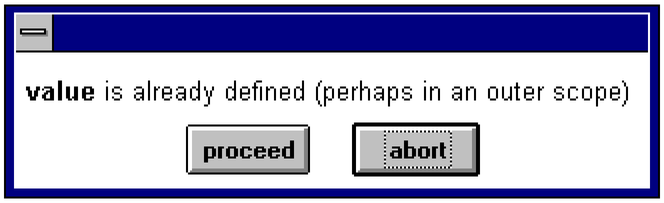

value is already defined (perhaps in an outer scope)

Figure 5.7. Block argument value is already defined in an outer scope.

- A block must not assign a new value to any of its arguments and any attempt to do so is caught by the compiler. (The same is true for method arguments. In fact, the concepts of a method and a block are very similar.)
- The role and nature of individual block arguments depends on the definition of the message that uses the block. In the case of to:do:, the block argument must be a number and its role is a counter or an index.

Before we close this section , we will now show how to:do: might be defined to illustrate how a method acquires a block argument. Our definition is different from the definition in the library because the library definition uses several concepts that will be covered later.

```
to: end do: aOneArgumentBlock
"For each number in the interval from the receiver to the argument, incrementing by 1, evaluate the block."
| index |
	index := self.
	[index <= end] whileTrue: [aOneArgumentBlock value: index. index := index + 1]
```

The critical part of the definition is message value: index which assigns index as the value of the block argument and causes evaluation of the block. In other words, the value: message is the one-argument-block equivalent of the zero-argument-block value message

The fact that the definition of to:do: evaluates aOneArgumentBlock with the value: message (rather than the value message) is the reason why you must use a one-argument block when you send to:do:. The number of arguments in a block is thus dictated by the block's use in the definition. The role of the argument, of course, depends on how the argument is used. In this case, the argument is evaluated with index and index is used to do arithmetic, so the argument must be a number.

As a more direct illustration of the operation of value messages, evaluating the zero-argument block in

```
[Transcript show: ‘A string’] value
```

has the same effect as

```
Transcript show: ‘A string’
```

whereas evaluation of the one-argument block in

```
[:aString |Transcript show: aString] value: ‘Block argument’
```

has the same effect as

```
Transcript show: ‘Block argument’
```

and

```
[:string1 : string2 |Transcript show: string1; show: string2] value: ‘argument 1 ‘ value: ‘argument2’
```

has the same effect as

```
Transcript show: 'argument 1 '; show: 'argument2'
```

Finally, a word of caution. If you search for all references to value: you will get a very long list but most of them refer to the value: method used with 'value holders' and have completely different meaning.

Main lessons learned:

- To execute a block for a range of values in increments of 1, use to: anInteger do: aOneArgumentBlock.
- The block in message to:do: has one internal argument.
- The receiver of to:do: may be any number.
- The number of arguments required in a block used as an argument of a message depends on the definition of the method in which the block argument is used.
- Use value to evaluate a zero-argument block, value: to evaluate a one-argument block, and value:value: to evaluate a two-argument block. Class BlockClosure defines additional value messages.

Exercises

1. List all numbers between 1 and 1,000 that are divisible by 11 and 13 in the Transcript. Use to:do:, timesRepeat:, and whileTrue and compare the complexity and readability of the code.

5.6 Repeating a block with a specified increment

Message to:do: lets you specify the start and end values but not the step (increment) between consecutive values of the block argument. Message to:by:do: allows you to specify the increment too (Figure 5.8). As an example,

```
1 to: 100 by: 2 do: [:number | Transcript show: number printString; cr]
```

prints all odd numbers from 1 and 99.

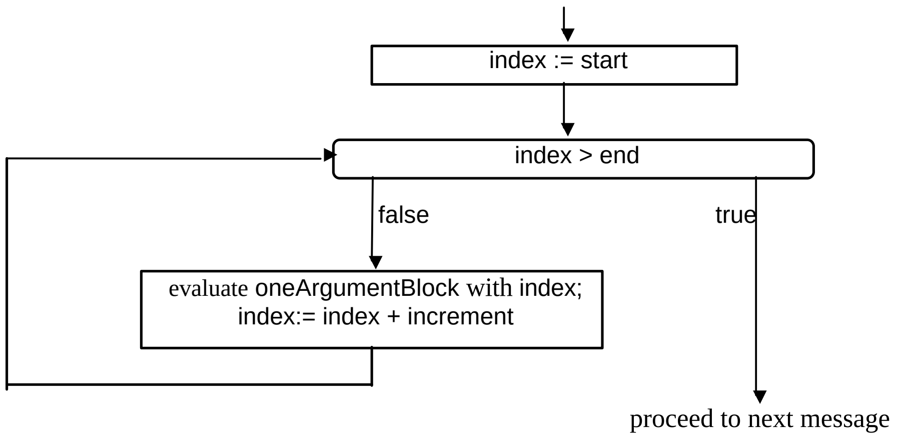

index := start
index > end
false
true
evaluate oneArgumentBlock with index;
index:= index + increment
proceed to next message

Figure 5.8. Execution of start to: end by: increment do: oneArgumentBlock.

All the numeric arguments of the message (start, end, and increment) may be any numbers - positive, negative, integer, floating-point, fractions, and so on. Don't forget, however, that floating-point arithmetic is inaccurate[3] and using floating-point numbers with to:by:do: may produce unexpected results. The following example illustrates this point.

[3] Smalltalk comments use the word of accuracy where others use the term precision.

Example 1: Print a table of base-10 logarithms for arguments from 1 to 10 in increments of 0.1
The problem is easily solve with the following expression:

```
Transcript clear.
1 to: 10 by: 0.1 do: [:number | Transcript show: number printString;
                                            tab;
                                            show: number log printString;
                                            cr]
```

Unfortunately, the code stops short of outputting the logarithm of 10.0 because the accumulated inaccuracy of adding 0.1 causes the program to miss 10.0. To avoid this problem, we can use a Fraction increment as follows:

```
Transcript clear.
1 to: 10 by: 1/10 do: [:number | Transcript show: number asFloat printString;
                                            tab;
                                            show: number log printString;
```

This code produces the desired result but seems very slow - and so is, in fact, the original floating-point solution. This could have one of two reasons: either the calculation of logarithms is so slow, or the display is slow. In this case, the cause of the problem is the output and the solution is to delay output until the string is complete and then print it out all at once. To do this, use nextPutAll: aString to accumulate the individual strings in the Transcript object, and then flush to output:

```
Transcript clear.
“Calculate and gather the results.”
1 to: 10 by: 1/10 do:[:number | Transcript nextPutAll: number asFloat printString;
                                    tab;
                                    nextPutAll: number log printString;
                                    cr].
“Output the accumulated results.”
Transcript flush
```

This version is much faster and the trick is worth remembering.

Example 2: How accurate is asRational?
In Section 1, we showed that asRational performs very accurate conversion from a floating-point number to a Fraction. We would now like to find how often the original number and the fraction converted back to a floating-point number are equal, how often they are different, and what is the maximum error when they are not different. As for the error, we are interested in the relative accuracy, the value of

```
(aNumber asRational asFloat) - aNumber) / aNumber
```

To find the answer, we will repeat the conversion for all numbers between 1 and 10,000 in steps of 0.1, a total of 100,000 argument values.
Solution: With the stepwise iteration message that we just introduced, the solution is as follows:

```
| count maxError |
“Initialize.”
count := 0. maxError := 0.
“Perform calculation.”
1 to: 10000 by: 0.1 do:
        [:number| |error| (error := number asRational asFloat - number) isZero not
                ifTrue: [count := count + 1. maxError := (maxError max: error / number)]].
“Output the results.”
Transcript show: ‘Number of inaccurate conversions: ‘, count printString; cr;
        show: ’Max error: ' maxError printString
```

When we executed this code, we got (after worrying for a little while whether we did not have an infinite loop)

```
Number of inaccurate conversions: 9357
Max error: 2.38419e-7
```

We conclude that asRational is very accurate because only 9,356 out of 100,000 calculations are inexact and this corresponds to about 0.9%. In other words, the conversion produces the same result as the original in 99% cases - at least in the chosen range. Moreover, when the results are not equal, the worst relative inaccuracy is only 0.00002%.
A few points are worth mentioning about the code fragment:

- We used a *temporary variable* error inside the block. You can declare any number of temporary variables inside a block whereas the number of block arguments is *given* by the value message used to evaluate the block. The temporary variable's scope is limited to the block.
- Blocks whose body does not depend on any external context are called *clean*, other blocks are either *copying* blocks or *full* blocks. As an example, a block using only internal temporary variables is clean whereas one that depends on externally declared local variables or arguments is not. Clean blocks are much more efficient, and temporary variables should thus always be declared inside blocks if possible. See the comment in BlockClosure for more details.

Main lessons learned:

- To iterate over a range of numbers, use to:do: if the step is 1, and to:by:do: if the step is not 1.
- For multiple output to Transcript, replace multiple show: messages with nextPutAll: and flush.
- Blocks may include any number of internal temporary variables.
- Blocks that don't depend on external context are called clean. They are more efficient than blocks that depend on external context, for example external temporary variables or method arguments.

Exercises

1. Print sin and cos values of arguments from 0 to π/2 in increments of 0.05 in the Transcript.
2. Repeat Exercise 1 but print results in reverse order of arguments.
3. Print a table of all multiples of 13 between 1 and 1,000 using to:by:do:
4. Run the test of asRational for a different range of values to check whether our result is representative. Repeat for Double numbers.
5. Which of the following combinations is correct?
   a. method1: [:arg1 arg2 | ....]		"Definition of method1 uses value:value:"
   b. method1: [:arg1 :arg2 | ....]		"Definition of method1 uses value:"
6. Method to:do: is a specialization of to:by:do: that takes 1 as increment. This kind of specialization is very common and class Dialog contains several other examples. List how many such specializations occur in its file name dialogs protocol and comment on their implementation.

5.7 Measuring the speed of arithmetic and other operations

After implementing the code of an application, we may want to optimize its critical parts to achieve satisfactory response time. To do this, we need some means of measuring how much time a piece of code requires to run, and where it spends most time. This activity is called *profiling* and the Advanced Tools extension of VisualWorks provides tools to perform such analysis. For simple measurements, a method in class Time gives useful answers too and this is the approach that we will use in this section.

Example 1. Comparing addition of different kinds of numbers

When we talked about different kinds of numbers, we said that some of them perform arithmetic faster than others. We will now do a little experiment to compare the speed of addition of SmallInteger, LargePositiveInteger, Float, Double, and Fraction. We will do this by packing addition into a block and measuring how long the block takes to execute. This is easy because class Time knows what time it is and to measure how long a block of code requires to execute, we just need to record the time before we start (time t1), the time just when the block is finished (time t2), and then subtract t2-t1 to get the result. Although this idea is correct, it would not work because measuring time in seconds is too long for calculations. For this purpose, we will thus replace Time now with Time millisecondClockValue which returns a millisecond count of the computer (not derived from Time now). The solution is thus

```
| startTime endTime runTime |
startTime := Time millisecondClockValue.
“Execute the code here.’
endTime := Time millisecondClockValue.
runTime := startTime - endTime
```

In fact, measuring the length of execution is so useful that class Time includes a method called millisecondsToRun: that performs this task. It takes a block as its argument and returns the number of milliseconds that it takes to execute. As an example,

```
Time millisecondsToRun: [x := 2 + 3]
```

returns the number of milliseconds required to add 2 and 3 and assign the result to variable x. The definition of millisecondsToRun is

```
millisecondsToRun: timedBlock
"Answer the number of milliseconds timedBlock takes to return its value."
	| initialMilliseconds |
	initialMilliseconds := self millisecondClockValue.
	timedBlock value.“Evaluate the block argument here.”
	^self millisecondClockValue - initialMilliseconds
```

Now that we have the tool for measuring execution time, we must devise a strategy for getting a good answer to our question. The obvious approach is to execute addition for a pair of small integers and measure how long it takes, do this again for large integers, and so on, something like

```
| t |
Transcript clear.
“Do SmallInteger first”
t := Time millisecondsToRun: [2+3].
Transcript show: 'The time needed to add two small integers is ', t printString, ' milliseconds'; cr.
“Do LargePositiveInteger next.”
t := Time millisecondsToRun: [20000000000+30000000000].
Transcript show: 'The time needed to add two large integers is ', t printString, ' milliseconds'; cr.
“And so on for other kinds of numbers.”
```

Unfortunately, this will not work because addition of two integers takes only a few CPU cycles and modern CPUs running at 100 million cycles per second or more take only microseconds rather than milliseconds to execute them. To get a realistic value in milliseconds, we must do the addition many times. We should thus do something like

```
Time millisecondsToRun: [ 10000 timesRepeat: [2+3]]
and so on
```

This is not perfect either because it measures the time to execute 2+3 plus the time needed to execute the repeatTimes: message. To eliminate this effect, we will correct the obtained time by subtracting the time needed to run the repeatTimes message by itself as in

```
t1 := Time millisecondsToRun: [ 10000 timesRepeat: [2+3]].
t2 := Time millisecondsToRun: [ 10000 timesRepeat: [“nothing”]].
timeToAddIntegers := t1 - t2
```

Our complete solution is as follows:

```
| t1 t2 |
Transcript clear.
“SmallInteger: Two small integers.”
t1 := Time millisecondsToRun: [ 10000 timesRepeat: [2+3]].
t2 := Time millisecondsToRun: [ 10000 timesRepeat: []].
Transcript show: 'The time needed to add two small integers is ',
            ((t1-t2) / 10000) asFloat printString, ' milliseconds'; cr.
“LargePositiveInteger: Two large integers.”
t1 := Time millisecondsToRun: [ 10000 timesRepeat:
            [20000000000+30000000000]].
Transcript show: 'The time needed to add two large integers is ',
            ((t1-t2) / 10000) asFloat printString, ' milliseconds'; cr.
“and so on.”
```

When I executed this program on my laptop computer, I got the following result:

```
The time needed to add two small integers is 1.0e-4 milliseconds
The time needed to add two large integers is 0.0041 milliseconds
The time needed to add two very large integers is 0.0075 milliseconds
The time needed to add floating-point numbers is 9.0e-4 milliseconds
The time needed to add two fractions is 0.0325 milliseconds
```

which confirms our expectations: Floats take longer than integers, large integers take longer than floats and the time depends on the size of the number because it requires repeated addition over all consecutive parts. Fractional arithmetic takes very long, even longer than arithmetic on very large integers.

If you repeat this experiment, you should consider that I have a standalone computer that I don’t share with anybody. If you are using a CPU shared over the network, the result may be completely misleading because your CPU may be switching your work and somebody else’s work during the execution of your program and the test will include the total elapsed time. But even if you have a standalone computer, the result may not be correct because the computer may be doing some other work during the calculation such as collecting unused objects (garbage collection).

More sophisticated profiling tools do not measure the time but instead sample program execution, taking a peek once every little while and keeping count of how many times each message is encountered. In the end, they produce the percentage of time spent in individual messages. You can then focus on the messages that are sent most often. Profiling tools can also tell you about the size of your objects.

<u>Example 2 - Evaluating the speed of recursion</u>

Recursion sometimes very naturally solves a problem but it is generally considered inefficient and undesirable. Besides, it tends to use more memory space than other solutions. A classic example of a problem that lends itself very well to recursive solution is the calculation of the *factorial* whose recursive definition is as follows:

The factorial n! of an integer n is
- *undefined for n < 0*
- *1 if n = 0*
- *n * (n-1)! if n >0*

According to this definition, the factorial of 3 is 3 times the factorial of 2, which is 2 times the factorial of 1, which is 1. Altogether, the factorial of 3 is thus 3*2*1.

VisualWorks library already contains a definition of factorial and this definition is based on iteration since n! = n*(n-1)*(n-2)* ... * 2 * 1. In essence, the library definition is as follows:

```
factorial
        | tmp |
        self < 0 ifTrue: [^self warn: 'Factorials are defined only for non-negative integers'].
        tmp := 1.
        2 to: self do: [:i | tmp := tmp * i].
        ^tmp
```

However, the recursive definition can be more easily implemented as follows:

```
factorialRecursive
        self < 0 ifTrue: [^self error: ‘Negative argument for factorial’].     “Open exeception window.”
        self = 0 ifTrue: [^1].
        ^self * (self - 1) factorialRecursive
```

We will now examine whether recursion is slower and if so, how much slower. We will use the technique from Example 1 as in the following code fragment:

```
| t |
Transcript clear.
t := Time millisecondsToRun: [1000 factorial].
Transcript show: 'Nonrecursive factorial. Time: ', t printString; cr.
t := Time millisecondsToRun: [1000 factorialRecursive].
Transcript show: 'Recursive factorial. Time: ', t printString; cr
```

When I executed this code, I obtained the following results

```
Nonrecursive factorial. Time: 131
Recursive factorial. Time: 190
```

and concluded that although there is a difference, it is smaller than I expected. We will thus analyze the code and try an find the explanation.

Both methods begin with the same test and the essence of the remaining calculation is multiplication of intermediate results. In the case of factorial, this is followed by incrementing the number which should be rather negligible with respect to multiplication, and another iteration. Could it be that multiplication is the culprit? Could it be that multiplication takes so long that it masks everything else and that our tests actually mostly measure the duration of multiplication rather than the effect of recursion? This is quite possible, especially when the numbers become LargePositiveInteger (which they do very quickly) - we have already seen how slow their arithmetic is.

To test this hypothesis, we will do recursion with a block that takes much less time - calculation of the sum of numbers from 1 to n using a recursive approach based on

```
sum(n) = sum(n-1) + n
```

and an iterative approach based on

```
1 to: n: do: [:number | sum := sum + number]
```

We will not use the obvious solution

```
sum(n) = n*(n+1)/2
```

because our goal is to compare recursion with iteration doing over the same operation.

When we defined a sum method corresponding to these two styles(left as an exercise) and ran the following test

```
| t |
Transcript clear.
t := Time millisecondsToRun: [10000 sum].
Transcript show: 'Nonrecursive sum. Time: ', t printString; cr.
t := Time millisecondsToRun: [10000 sumRecursive].
Transcript show: 'Recursive sum. Time: ', t printString; cr
```

the result was much more like what we expected and confirmed our hypothesis:

```
Nonrecursive sum. Time: 2
Recursive sum. Time: 22678
```

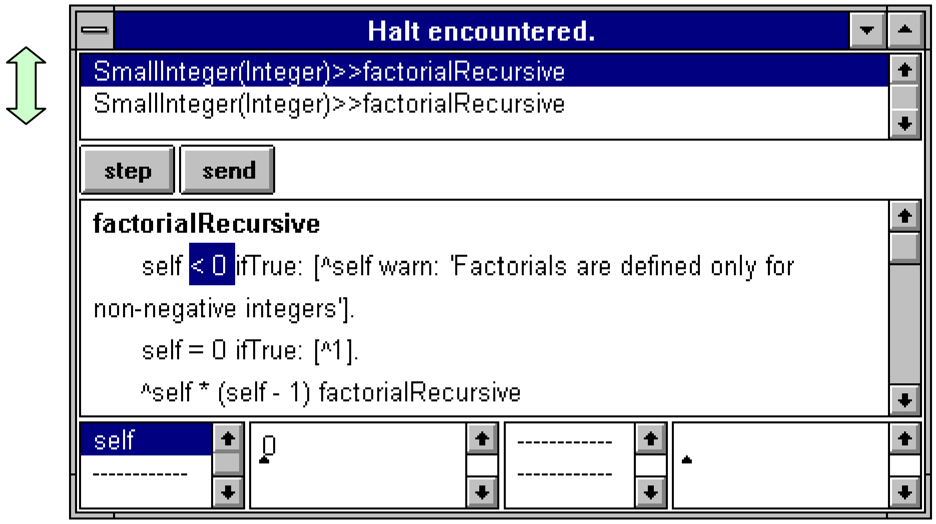
This leads to the following conclusion: Recursive programs generally execute slower than their iterative counterparts. However, when the calculation performed during each recursive step is time-consuming, the difference between recursion and iteration may be negligible. A corollary is that if the calculation is substantial, converting a natural recursive solution to a less obvious iterative solution may not be worth the trouble. The elegance of recursive solutions makes them easier to understand which contributes to the maintainability of code. This sometimes makes recursive solutions preferable in complex problems, particularly in earlier stages of development before the code is optimized for speed.

Example 3: Observing recursion in action

To complete this section, execute the following code fragment

```
self halt.
5 factorialRecursive
```

and follow the execution of the recursive definition of the factorial all the way from 5 to 0 by using *send* in the Debugger. When self decrements from the initial value of 5 to the end value of 0, the debugger window is as in Figure 5.9, showing all the stacked recursive calls. Recursion then begins to unwind, completing the messages piled up on the stack, and eventually returning to the original program fragment with the final result. This little experiment shows very nicely the nature of recursion.



```
SmallInteger(Integer)>>factorialRecursive
SmallInteger(Integer)>>factorialRecursive
```

```
factorialRecursive
    self < 0 ifTrue: [^self warn: 'Factorials are defined only for
non-negative integers'].
    self = 0 ifTrue: [^1].
    ^self * (self - 1) factorialRecursive
```

Figure 5.9. The top two levels of the message stack during the execution of 5 factorialRecursive.

Main lessons learned:

- The first implementation of an application should not focus on efficiency. Speed should only become a concern when the running application is too slow.
- In trying to improve speed, focus on those parts of the code where the application spends most of its time.
- Determining where the application spends most of its time is called profiling.
- Profiling can also be used to check how much memory an application uses and where.
- Analyzing code for efficiency requires careful analysis because the most natural conclusions may be incorrect.
- Recursion requires overhead but this overhead may be outweighed by calculation required in each recursion step.
- Solving problems with recursion is sometimes much easier than using other approaches and recursive solutions are often preferred until final speed optimization.

Exercises

1. Method timesRepeat: is implemented by the whileTrue: message and so timesRepeat: must be slower. How much slower is it?
2. Test whether our precautions in Example 1 (eliminating the effect of repetition) were justified.
3. Add Double arithmetic to Example 1 and compare with Float.
4. We noted that the duration of addition of large integers depends on their size. Obtain an experimental dependency between the duration of addition and the length of the operands, and explain it qualitatively. To do this, create some large integers such as 100 factorial, 500 factorial, and 1000 factorial, store them in variables x, y, and z, and perform x+1 and y + 1 and z + 1 (without assignment) a sufficient number of times, measuring the speed. Print the results along with some indication of the size of the numbers, draw a graph of the dependency, and explain it.
5. Compare the speed of + - * and / in the most important number classes.
6. Write recursive and non-recursive definitions of the methods used in Example 3 and test their relative speed.
7. About 2,000 years ago, the Greek mathematician Euclid invented the following algorithm for calculating the greatest common divisor gcd(m,n) of two positive integers m and n:
   If m = n, gcd = m
   If m < n, gcd(m,n) = gcd(n,m)
   If m > n, gcd(m,n) = gcd(m-n,n)
   Use this recursive definition to define method gcdRecursive: to calculate gcd recursively.
8. VisualWorks library contains a built-in gcd: method. Compare its speed with the speed of the recursive implementation. Analyze the result.
9. The Fibonacci series is defined as follows: Fib(n) = 1 for n = 1, 2, Fib(n-1) + Fib(n-2) for n > 2, and Fundefined for all other values of n. Write and test method fibonacci to calculate Fibonacci numbers.

5.8 A new class: Currency

In this section, we will define a new class as a simple application of numbers. In doing this, we will introduce new techniques useful for testing.

When we discussed number classes, we noted that FixedPoint was designed as a possible solution of the problem of representing currency. The problem is not as simple as it might seem, for example because operations such as conversion from one currency to another are very sensitive to tiny inaccuracies whose effect becomes significant when we deal with billions of dollars or convert back and forth between currencies.

We will now define an alternative simple Currency class for currencies whose units consist of dollars and cents. Our solution is far from perfect and we leave the fine points of currencies to experts.

Design

The behaviors that we expect of Currency include creation (create a new Currency object), arithmetic (add or subtract two Currency objects), comparison (equal, not equal, less than, and so on), and printing (for testing and inspecting). Each of these behaviors will be implemented as a protocol, and the notion of dollars and cents suggests the use of two instance variables called dollars and cents.

We have now decided all the major features of Currency except where to put it in the class hierarchy. Currency objects are somewhat like numbers, so they will be somewhere on the Magnitude branch of the class tree. Since currencies can do simple arithmetic, we must consider whether the superclass should be ArithmeticValue or even Number. However, although we want to be able to add and subtract Currency objects, we don't need any other arithmetic and in this sense, Currency objects are thus distinct from numbers. Also, the comment of ArithmeticValue says that its subclasses must implement certain arithmetic messages needed by some other messages, and since we don't really need most of them, this reinforces our opinion that Currency should not be a subclass of ArithmeticValue. Most importantly, however, currency objects are conceptually different from numbers and it does not make sense to think of them as specialized numbers.

We conclude that making Currency a subclass of ArithmeticValue or Number has too many disadvantages and few advantages and we will thus define it as a subclass of Magnitude, inheriting certain comparison messages. In this, we will follow in the footsteps of those who defined classes Time and Date. We will put Currency into the existing category Tests as we did with class Name.

We are now ready to define the new class. Start by opening the browser on category 'Tests', edit the class template as in

```
Magnitude subclass: #Currency
        instanceVariableNames: 'dollars cents '
        classVariableNames: ''
        poolDictionaries: ''
        category: Tests'
```

and execute *accept* from the <operate> menu of the text view.

As the next step, add a comment explaining the purpose of the class, its instance variables, and any special notable features. Our comment for Currency will be as follows:

I implement a decimal currency such as dollars and cents.

Instance variables:

dollars	<Integer> - the dollar part of the currency
cents	<Integer> - the cent part of the currency

We are now ready to implement the behaviors. Since we cannot do anything unless we can create an instance, we will start with the creation message.

Creation

The style that we want to use for the creation message is to specify initial values for dollars and cents as in

```
Currency dollars: 100 cents: 45
```

Message dollars:cents: is obviously a class message because its receiver is Currency, and its implementation will follow Smalltalk's established pattern for creation messages:

1. Create a new Currency object by sending new to the Currency class. This inherited message creates an uninitialized instance of Currency with its dollars and cents instance variables set to nil.
2. Send dollars: anInteger to this instance, setting its instance variable dollars to the value of the argument. (Accessing method dollars: does not yet exist and we will have to define it.)
3. Send cents: anInteger to the Currency object, setting its instance variable cents to the value of the argument. (We will again have to define method cents: first.)
4. Return the new Currency object.

The definition written along these lines is as follows:

```
dollars: dollarsInteger cents: centsInteger
"Creates a new initialized Currency object."
^ self new
        dollars: dollarsInteger;
        cents: centsInteger
```

Note the following points:

- The effect of self new is equivalent to Currency new but the self new version is preferable: If we created a subclass of Currency such as NewCurrency, the version using self new would create an instance of NewCurrency (which is what we would probably expect) whereas the version using Currency new would create an instance of Currency (probably not what we want). Conclusion: Don't refer to a class explicitly unless you must.
- Although the inherited version of new is defined in class Behavior, it creates an instance of Currency or whatever class sends the message.
- It is a very common mistake to forget the return operator ^ in a creation method. The method then returns the class rather than the new instance.
- Use cascading to send initialization messages to the new instance of Currency.

Although we could now enter the definition of dollars:cents: into the browser and compile it before creating dollars: and cents:, it will be better to define accessing methods dollars: and cents: first. Both are instance messages because their receiver is self new - an instance of Currency - and they simply assign new values to instance variables:

```
dollars: anInteger
        dollars := anInteger
```

and similarly for cents:.

The methods are so trivial that they don't require a comment. Create these instance methods and the dollars:cents: creation method, and test whether everything works by executing

```
Currency dollars: 100 cents: 47
```

with *inspect*. When you examine the values of instance variables, you will find that everything is OK.

<u>Printing</u>

At present, the only way to see the values of instance variables of Currency objects is to open an inspector or to access them. Our life would be much easier if we could display the components of Currency by printString but this would, of course, only produce the string 'a Currency'. If we want printString to produce useful information about instances of a new class, we must change the printString mechanism and to do this, we must understand how it works.

When we browse the implementers of printString we find that there is essentially only one - class Object - and that the definition reads as follows:

```
printString
	"Answer a String whose characters are a description of the receiver."
	| aStream |
	aStream := WriteStream on: (String new: 16).
	self printOn: aStream.
	^aStream contents
```

The critical message is obviously self printOn: aStream and to change the behavior of printString, we must therefore redefine printOn:. There are many implementors of printOn: because this method customizes printString to produce useful information. As an example, the definition in class Contract is as follows:

```
printOn: aStream
	aStream nextPutAll: self class name, ' as ', self server name,' :: ', name printString; cr
```

and we will use it as a template even though we don't understand very well yet what it does. First, however, we must decide how we want a Currency to display itself. We selected the following style:

```
Currency dollars: 100 cents: 47
```

executed with print it (thus using printString) should produce

```
a Currency 100 dollars, 47 cents
```

To produce this format, our printOn: message must execute the following steps:

1. Output the name of the class into aStream (we don't care what a Stream is, at this point).
2. Output the dollars part followed by ' dollars '.
3. Output the cents followed by ' cents'.

According to this algorithm and the printOn: template, we define printOn: in class Currency as follows:

```
printOn: aStream
"Append to the argument, aStream, the description of a Currency."
aStream nextPutAll: self class name,
	self dollars printString,
	 ' dollars, ',
	self cents printString,
	' cents, '
```

To test how this method work, we execute

```
Transcript show: (Currency dollars: 13 cents: 27) printString
```

and we indeed obtain the desired output.

Note again that we did not need to understand the details, such as what a Stream is, in order to write a working definition. This is quite all right - don't try to understand every object that you must use; all you need is to know how to use it. A carpenter also does not need to know what is inside a drill if he wants to use one, and you don't care about the internal operation of the mail system if you want to send a letter. (On the other hand, it never hurts to understand your tools and if you don't have any pressing matters on your agenda, go ahead and learn about a new class.)

It is interesting to note that printString is also used by the inspector to display self and when you execute

```
Currency dollars: 13 cents: 27
```

with *inspect*, you will again obtain the desired information under the self entry in the inspector window.

Arithmetic and comparison

We will restrict ourselves to addition and leave subtraction to you. To add two currencies, we will create a new Currency object and set its dollars part to the sum of the dollars parts of the receiver and the argument, and the cents part to the sum of the cents parts as follows:

```
+ aCurrency
	^Currency dollars: (self dollars + aCurrency dollars) cents: (self cents + aCurrency cents)
```

We assume that you defined the accessing messages dollars and cents. Add the + method to the arithmetic protocol of Currency and test it by executing

```
(Currency dollars: 100 cents: 47) + (Currency dollars: 25 cents: 32)
```

with *inspect* or *print it*. The result shows that everything is OK[4].

Finally the *comparison* protocol. We already know that some comparison methods are defined in class Magnitude and we will now check its comment to find which comparison methods must be defined so that we can inherit the rest. The relevant part of the comment of Magnitude is

```
Subclasses must implement the following messages:
		<
		=
		hash
```

We will start with equality. Two Currency objects are equal if they have the same number of dollars and the same number of cents, hence

```
= aCurrency
	^(self dollars = aCurrency dollars) and: [self cents = aCurrency cents]
```

Our next target is the hash method. Its purpose is to assign an integer value to an object, such that can be used as its shorthand representation, something like a student number. Just like a student number or any other id, each object should have its unique hash value to distinguish it from other similar objects. This may be difficult and so hashing follows a weaker rule, namely that equal objects should have the same hash value. Because of this, and because some very important operations in Smalltalk use hashing instead of equality, redefining equality should always be accompanied by redefining hash.

[4] If you have reservations about our approach, wait for the improved implementation later.

A hash method is already defined in class Object but since we redefined equality, we will also redefine hash. Our class has two internal variables and we will thus borrow a hash definition from another class with two variables - class Fraction - and convert it for our needs. Our definition will be

```
hash
        ^dollars hash bitXor: cents hash
```

We leave the definition of < to you as an exercise.

Main lessons learned:

- When constructing new classes and methods, start with creation, accessing, and printing protocols and test your definitions immediately.
- The recommended style for creating a new instance is to initialize its instance variables.
- Don't refer to a class explicitly unless you must.
- A very frequent mistake in writing a creation method is to forget the return operator. The method then returns the receiver class rather than the new object.
- When you need to define a specialized version of an existing method, check how other classes do it and follow the template. You don't have to understand the details.

Exercises

1. Add class protocol testing and write methods to perform complete testing of Currency.
2. As an alternative to the previous exercise, remove testing from Currency and create a new class called CurrencyTesting containing test methods in a class protocol. The Currency objects used in the tests can be stored in class variables and the class variables initialized in the *class* initialization method initialize.
3. Write and test the - method for subtraction of currencies.
4. Write and test the implementation of <. What about <=, >, and >=?
5. In the definition of + aCurrency, we had to access the components of aCurrency with accessing methods dollars and cents. The dollars and cents instance variables of the *receiver*, however, could have been accessed directly. Many Smalltalk experts argue that instance variables should always be accessed by accessing methods even though it is not strictly necessary. Give one argument supporting this position and one argument against it.
6. Define a new class called UniformRandom that allows the user to create a random number generator for numbers uniformly distributed in any interval, not just between 0 and 1 as Random does. The creation message should be UniformRandom from: startNumber to: endNumber and the instance message for creating a new random number should be next. (Hint: It is tempting to make UniformRandom a subclass of Random because they are so closely related. However, UniformRandom would not inherit any useful functionality or components from Random, it would have to redefine next, and its creation message is different - so there is no point in subclassing. Since there is no related class in the Smalltalk hierarchy, make UniformRandom a subclass of Object and define it with instance variables generator - an instance of Random to be used to calculate random numbers in <0,1> - and start and end to hold the endpoints of the interval. Calculations of new random numbers will depend on getting a new random number from generator, and scaling it using the start and end values of the interval. As we mentioned before, this approach is called *delegation* because UniformRandom passes responsibility for a part of its functionality to one of its components. *Delegation is generally preferred to inheritance which should be used only in cases of pure specialization.*)

5.9 Another implementation of Currency

Our existing implementation of Currency is fine in terms of its protocols but unsatisfactory in implementation. As an example, expression

```
Currency dollars: 100 cents: 3412
```

creates a Currency object with 100 dollars and 3412 cents but we would prefer a Currency object with 134 dollars and 12 cents. Similarly,

```
Currency dollars: 100 cents: -34
```

creates a strange object that does not make sense, and our comparison message = returns false for

```
(Currency dollars: 100 cents: 3412) = (Currency dollars: 134 cents: 12)
```

whereas we would probably expected true.

If we had thought about the implications of our design more carefully (in other words, if we did the design properly), we would have discovered the problem and changed the design before implementing it. At this point, we will have to make some changes. One possible approach is to leave the protocols, and redefine Currency in terms of cents only and eliminate dollars. Let’s do this and see what implications it has for the methods that we had defined.

Creation

We still want to be able to create Currency objects from dollars and cents and so our creation message must convert dollars to cents. The new version of the creation method is simply

```
dollars: dollarsInteger cents: centsInteger
 “Creates a new initialized Currency object.”
	^ self new cents: centsInteger + integer1 * dollarsInteger
```

Accessing

We don’t really need method dollars: (it was only needed for creation) but we will leave it in because we don’t want to change the message interface, in case somebody wrote another class that depends on it. We will leave it to you to redefine it to change dollars to cents. Methods cents and dollars must also be changed, as in

```
dollars
“Return dollar part of amount.”
	^ cents quo: 100
```

and

```
cents
“Return cents part of amount.”
	^ cents \\ 100
```

Arithmetic

The old implementation is

```
+ aCurrency
	^Currency dollars: self dollars + aCurrency dollars cents: self cents + aCurrency cents
```

but with the new creation method cents:, we can define addition much more easily as follows:

```
+ aCurrency
	^Currency cents: self cents + aCurrency cents
```

The only problem is that we don’t have a creation message called cents: but this is easy to fix:

```
cents: centsInteger
 "Creates a new initialized Currency object."
	^ self new cents: centsInteger
```

Note that we now have two different cents: methods but that's OK because one is a class method and the other is an instance method. This means that Currency cannot be confused because only one cents: method exists for it, and instances cannot be confused either.

Comparison

Comparison also becomes much simpler - and we don't have to worry about cents above 100 or below -100

```
= aCurrency
	^self cents = aCurrency cents
```

The hash method can use the hash value of integer cents as follows:

```
hash
	^cents hash
```

Printing

We will leave this protocol to you as an exercise.

This completes our redesign. The changes that we made are very small and simplify the code substantially. Note that they don't affect any other code that may use Currency because the old message interface remains, both in form and in effect. In particular, the test methods suggested in exercises at the end of the previous section will also work and can be used to test the new implementation. The principle of information hiding thus allowed us to reimplement Currency without any effect on any applications that may already be using it. (Note, however, that if we created some Currency objects using the first implementation and stored them in a file, we could not reuse them with the new implementation without some additional conversion.)

Main lessons learned:

- If we modify a class without changing its message interface, objects already using the class will not be affected. This is a consequence of information hiding.

Exercises

1. Complete the new implementation of Currency. Don't forget to change the comment and to retest the new implementation.
2. Compare our implementation of Currency with FixedPoint with scaling factor 2.
3. In an article in Smalltalk Report, Kent Beck described an implementation of currency that goes beyond dollars and cents and avoids conversion inaccuracy as much as possible. His approach is based on the fact that a wallet might contain any combination of currencies such as US $3.50, Canadian $6.75, and Japanese ϒ300. This combination does not automatically convert to one currency unless explicitly required. Implement a Currency class based on this idea but restricted to four currencies: Canadian dollars, US dollars, British pounds, and French francs. When an amount is added or subtracted, arithmetic is done only on equal denominations and conversion is performed only when explicitly required.

4. Currency is a class that operates on objects measured in units. Develop a class called Length to handle the metric system with millimeteres, centimeters, meters and kilometers, and the British system with inches, feet, yards, and miles. Use the same approach as in the previous exercise.
5. Repeat the previous exercise for weight objects in units of grams, dekagrams, and kilograms on one side, and ounces and pounds on the other.

5.10 Generalized rectangles

As another example of a new class, we will now implement a new class for rectangles. One rectangle class, called Rectangle, is already in the library but it is restricted to rectangles with horizontal and vertical sides because its intended use is for windows on the screen. Our class will handle any rectangles and we will thus call it AnyRectangle. Before we start developing it, let’s decide what we want to do with it and what functionality it should have.

Our reason for adding AnyRectangle is that rectangles are fairly common geometric objects and that we might want to use them in a drawing program and similar applications. The main purpose of the new class is thus to support display and graphics operations such as *move* and *rotate*. Purpose determines functionality and we thus decide on the following protocols:

- creation
- accessing - get and set selected rectangle parameters
- transformation - move and rotate
- display - drawing on the screen
- printing - for testing purposes

The next question is how to represent a rectangle. The horizontal/vertical Rectangle class uses the coordinates of the upper left and lower right corner because the orientation of the sides is known. In our case, two corners are not enough because there is an infinite number of rectangles that have the same two corners. As an example, if the two given corners are C1 and C2 in Figure 5.10, any rectangle whose third corner lies on a circle with diameter C1 and C2 is acceptable.

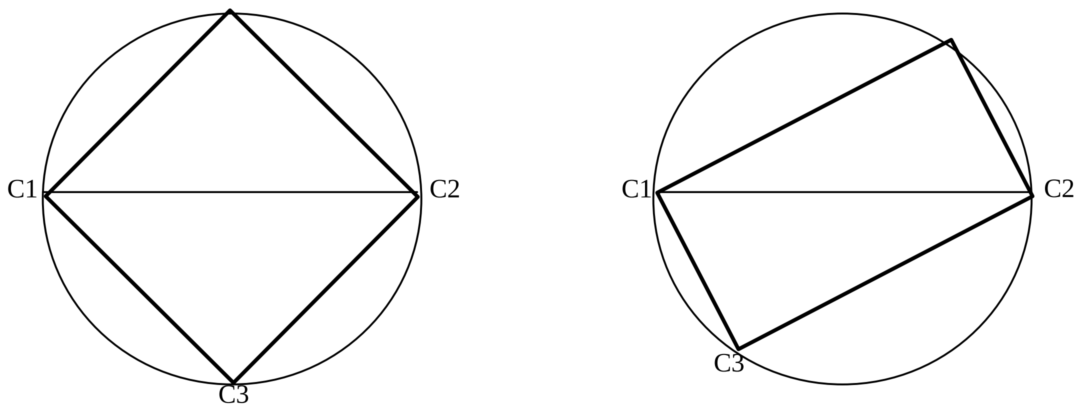
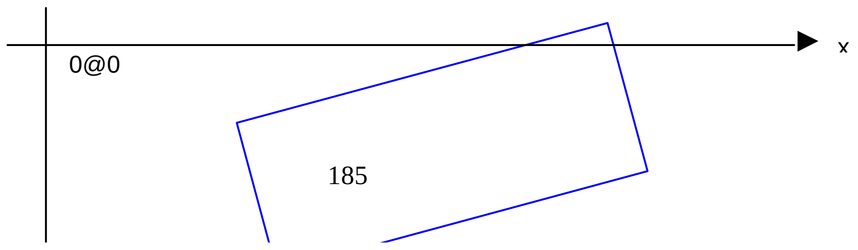


Figure 5.10. Two corners, such as C1 and C2, don’t define a unique rectangle.

Although two corners don’t identify a rectangle, three corners do. However, three points don’t necessarily define a right angle and so three points are not automatically corners of a rectangle. We will thus represent a rectangle by its two opposite corners and an angle, thinking of the rectangle as a rotated horizontal-vertical rectangle (Figure 5.11).



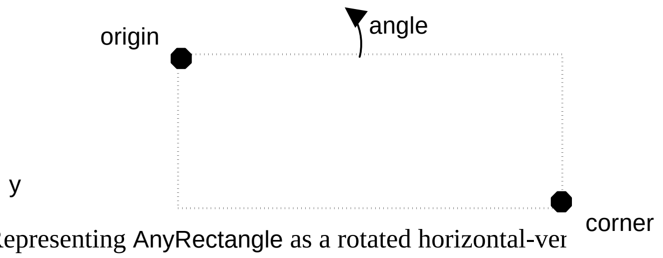


Figure 5.11. Representing AnyRectangle as a rotated horizontal-ver ıgle.

Having made this decision, we can now start implementing the methods. We will use Rectangle as the superclass of AnyRectangle because it already defines all of the protocols we need, but we will leave it to you to check whether all inherited protocols remain valid as an exercise.

Creation. Since a rectangle is defined by three instance variables, the creation method will have three keywords. It will create a new instance and then send instance messages to assign instance variable values:

```
origin: point1 corner: point2 angle: radians
"Create a rectangle with three corners."
	^self new origin: point1; corner: point2; angle: radians
```

where the instance accessing methods are standard setter methods. A typical creation message would look like

```
AnyRectangle origin: 1@100 corner: 11@90 angle: 3.14/4
```

Transformation.
Moving a rectangle means shifting the x coordinate and the y coordinate of each corner without changing the angle. This means that AnyRectangle is moved the same way as an ordinary Rectangle and the inherited methods moveTo: and moveBy: should thus work without change.
Rotation. Before we start thinking about implementation, we must decide what kind of rotation we want. First, we will assume that we perform rotation by keeping the upper left corner (inherited instance variable origin) fixed and rotating the rectangle around it. In principle, we can see two ways of doing this: by specifying the angle of rotation, and by dragging the opposite corner interactively.
The first kind is easy – we just increment the angle as follows:

```
rotateByAngle: aNumber
	angle := angle + aNumber
```

If we want to lay a foundation for letting the user rotate the rectangle interactively, the task gets a bit harder and we must do some trigonometry. The logic of the operation is as follows:

1. Calculate the angle of rotation from the new position of the dragged corner.
2. Change the angle instance variable by the calculated rotation angle.

We leave it to you to figure out the details but remember that the screen coordinate system measures x from left to right and y *from top to bottom*. Thanks to predefined methods for calculating distance and similar parameters, this problem is easier than you might think.

Printing. From the section on Currency, we know that if we want a new printString behavior, we must redefine the printOn: method. There is no reason to make the string for AnyRectangle much different from the string for Rectangle (the only difference is that we need add the angle variable) and so we will reuse printOn: from superclass Rectangle

```
printOn: aStream
"Append to the argument aStream a sequence of characters that identifies the receiver. The general format is originPoint corner: cornerPoint."
```

```
	origin printOn: aStream.
	aStream nextPutAll: ' corner: '.
	corner printOn: aStream
```

and extend it as follows:

```
printOn: aStream
"Append to the argument aStream a sequence of characters that identifies the receiver. The general format is originPoint corner: cornerPoint angle: angle."
	super printOn: aStream.
	aStream nextPutAll: ' angle: '.
	angle printOn: aStream
```

In this definition, super is a special identifier that allows us to access an identically named method defined higher up in the class hierarchy. We will have more to say about it in Chapter 6.

Displaying. We will draw the rectangle by drawing the individual sides one after another. The subject of drawing will not be covered until Chapter 12 and we will thus implement this method by imitating the display method in class LineSegment which is defined as follows:

```
displayStrokedOn: aGraphicsContext
"Stroke the receiver on the supplied GraphicsContext."
	aGraphicsContext displayLineFrom: start to: end
```

In this context, ‘stroking’ means simply drawing, and we don’t need to know what a GraphicsContext is at this point. The method for displaying AnyRectangle must calculate the corners and draw the four lines. The whole definition is

```
displayStrokedOn: aGraphicsContext
"Stroke the receiver on the supplied GraphicsContext."
	| topRightCorner bottomLeftCorner bottomRightCorner |
	“Calculate corners.”
	topRightCorner := self topRightCorner.
	bottomLeftCorner := self bottomLeftCorner.
	bottomRightCorner := topRightCorner + bottomLeftCorner - origin.
	“Draw straight lines connecting corners.”
	aGraphicsContext displayLineFrom: origin to: topRightCorner.
	aGraphicsContext displayLineFrom: topRightCorner to: bottomRightCorner.
	aGraphicsContext displayLineFrom: bottomRightCorner to: bottomLeftCorner.
	aGraphicsContext displayLineFrom: bottomLeftCorner to: origin
```

where method topRightCorner calculates the top right corner on the basis of simple trigonometry

```
topRightCorner
	^origin + ((self width * angle cos) @ (self width * angle sin) negated)
```

and the remaining corner finding methods are left as an exercise. To test the new class, execute the following program to display several rectangles in the active window

```
| anyRectangle |
anyRectangle:= AnyRectangle origin: 10@80 corner: 50 @ 130 angle: 0.
anyRectangle displayStrokedOn: Window currentWindow graphicsContext.
anyRectangle moveBy: (100@50).
anyRectangle displayStrokedOn: Window currentWindow graphicsContext.
anyRectangle moveBy: (100@-50); rotateBy: (Float pi / 2).
anyRectangle displayStrokedOn: Window currentWindow graphicsContext.
anyRectangle moveBy: (100@50); rotateBy: (Float pi / 4).
anyRectangle displayStrokedOn: Window currentWindow graphicsContext.
```

Exercises

1. Explore class Rectangle and write its short description, listing its major behaviors. Comment of the number of useful behaviors that AnyRectangle inherits from Rectangle.
2. Complete AnyRectangle and test it. Check whether defining it as a subclass of Rectangle is appropriate and which of its behaviors might have to be redefined.
3. Add a method for drawing a filled rectangle. (Hint: Examine Rectangle.)
4. Drawing editors usually allow shrinking and stretching a geometric object. Decide how this could be done and implement the appropriate methods.

Conclusion

VisualWorks library includes a large hierarchy of number classes, mainly because hardware represents and processes different numbers differently. Another reason why there are so many number classes is that some computer applications require numbers that cannot be directly processed by hardware but are very easy to implement and provide extra programming convenience.

The most important number classes are integers and floating-point numbers. Smalltalk integers are implemented by three classes that separate the class directly implementing arithmetic in hardware from those that are added to provide unlimited integer range. Conversions from one class to another are transparent - performed automatically as needed.

Floating-point numbers are available in two implementations that differ in the number of bytes internally used to represent them and provide different accuracy and range. Conversion between high precision and ordinary floats is *not* automatic. In spite of their high precision, floating-point numbers are inherently inaccurate for decimal calculation because decimal numbers are incompatible with computer binary codes. Ignoring this principle may have serious consequences for comparison and iteration with floating-point ranges and increments. Floating-point arithmetic is also much slower than SmallInteger arithmetic and floating-point numbers require more memory. Floating-point numbers should thus be used only when necessary.

In addition to integer and floating-point numbers, Smalltalk also implements fractions with an integer numerator and denominator, fixed point numbers with accurate internal representation and a fixed number of decimal digits, complex numbers, and various kinds of special numbers representing values such as infinity. Of these, fractions and fixed-point numbers are the most important. They are inherently accurate but their arithmetic is very slow.

All Smalltalk number objects understand many messages implementing various kinds of arithmetic, mathematical, and conversion operations. Arithmetic is implemented on the basis of primitives and double dispatching. Primitives are messages translated into CPU code and implemented without further message sends. Double dispatching means re-sending (re-dispatching) a one-argument message to the argument, using the original receiver as the argument. Double dispatching eliminates tests because the dispatched message implies the type of the argument. Although the main use of double dispatching is in arithmetic, the principle has general validity.

In addition to calculations, numbers are also used for iteration - repetition of a block of statements a fixed number of times or over a range from a given starting value to a given end value, in specified or default fixed increments. The latter messages use blocks with internal arguments.

Blocks may have zero, one, or more internal arguments and the number of arguments is given by the type of value message used to evaluate the block. All value messages for block evaluation are defined in class BlockClosure. Blocks may also have internal temporary variables and their use is preferable to temporary variables declared in the surrounding context because closed blocks independent of the surrounding context are more efficient.

Profiling and timing refers to measurement of the time, message send profile, and memory required to execute a block of code. We have shown that millisecondsToRun: defined in class Time provides a simple but primitive way of timing. The Smalltalk profiler tool produces much more accurate and detailed answers.

Information hiding makes it safe to change internal implementation of an existing class. As long as the message interface does not change, changes of internal implementation don’t affect existing applications that depend on the class.

When implementing a method that is an extension of an inherited behavior, we usually execute the inherited behavior and then additional code. Reference to inherited behavior (methods defined higher up in the class hierarchy) can be obtained by using identifier super.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

ArithmeticValue, *Date*, *Double*, *FixedPoint*, **Float**, *Fraction*, **Integer**, **Number**, LargeInteger, LargeNegativeInteger, LargePositiveInteger, MetaNumeric, Random, *Rectangle*, SmallInteger, *Time*.

Terms introduced in this chapter

block argument - argument defined and used inside a block; specified using syntax [:x :y| ...]
closed block - one whose body does not depend on the external context
double dispatching - re-sending a one-argument message to the argument, with the original receiver as argument
fixed-point number – number with a fixed number of decimal digits
floating-point number - number with decimal point
integer number - number without decimal point
nested block - block within another block
message expression implemented directly by machine code and used instead of a message send
profiling - experimental analysis of program behavior with regards to its execution speed, relative frequency of message sends, and memory requirements
random number generator - an object that can produce an unlimited stream of numbers randomly distributed according to some probabilistic formula
temporary block variable - variable declared and valid inside a block; distinct from block argument