Module 5: Use of Inheritance, Class Variables and Protocols

This module reviews message sending and the method search mechanism, and considers the effect of messages to self and super. This very important aspect of using Smalltalk is illustrated with several small examples.

This module also explores three types of variable not yet considered — the *class* variable, the *class instance* variable and the *pool* variable. The module goes on to explore protocol typically implemented in classes, including 'instance creation', 'class initialization' and 'examples'. These aspects are explored using class Date and class Time as examples. Other examples are taken from classes ScheduledWindow and FileBrowser.

5.1. Methods revisited

A *method* describes how an object will respond to a *message*. It is made up of a *message pattern* and a *sequence of expressions* (separated by full stops). The names of arguments in the message pattern are accessible within the method. Temporary variables may also be used — they must be declared at the beginning of the method (just after the message pattern). Temporaries are initially nil, and are forgotten after the end of the method evaluation. The pseudo–variables self and super can also be used with methods to refer to the receiver itself. Once the method has finished evaluating, an answer is returned to the sender of the message. By default, the object returned is the receiver itself (i.e. self).

However, other objects can be returned by use of a return expression — an expression preceded by an "up–arrow" ('^'). The return expression must be the last expression evaluated in a method.

5.2. Review of Message Sending

When a message is sent, the methods in the receiver's class are searched for one with a matching selector. If none is found, the methods in that class's superclass are searched. The search continues upwards until a match is found, or the top of the hierarchy is reached. (The superclass of Object is nil.) If a match is made, the corresponding method is evaluated. This will probably cause further messages to be sent.

If no match is found during the search, the message doesNotUnderstand: is sent to the receiver. The argument is the offending message. A method with selector doesNotUnderstand: is implemented in the instance protocol of Object, which causes an error message to be displayed in a Notifier.

5.3. Messages to self and super

When a method contains an expression that causes a message to be sent to self, the search for the corresponding method starts in the class of the instance *regardless of which class contained the method containing self*. It is just as if some other object had sent the message — the search starts exactly as before.

The pseudo–variable super is also available for use within a method. It also refers to the receiver of the message (just like self). However, the search for the method does *not* start in the receiver's class. Instead, it starts in the *superclass of the class in which the method using super is located*. This allows methods to

access other methods defined in classes further up the hierarchy, even if they have been *overridden* in subclasses.

Ex 5.1: Type in the class descriptions and methods given below.

class	One
superclass	Object
instance variables	*none*
class variables	*none*
pool dictionaries	*none*
class category	Messages-example
message protocol	tests

```
result1
    ^self test

test
    ^1
```

class	Three
superclass	Two
instance variables	*none*
class variables	*none*
pool dictionaries	*none*
class category	Messages-example
message protocol	tests

```
result2
    ^self result1

result3
    ^super test
```

class	Two
superclass	One
instance variables	*none*
class variables	*none*
pool dictionaries	*none*
class category	Messages-example
message protocol	tests

```
test
    ^2
```

class	Four
superclass	Three
instance variables	*none*
class variables	*none*
pool dictionaries	*none*
class category	Messages-example
message protocol	tests

```
test
    ^4
```

Evaluate the following expressions, one at a time:

```
| ex1 |
ex1 := One new.
ex1 test.

| ex1 |
ex1 := One new.
ex1 result1.

| ex2 |
ex2 := Two new.
ex2 test.

| ex2 |
ex2 := Two new.
ex2 result1
```

Explain what happens.

Ex 5.2: Also, explain what happens with each of the following expressions (create the appropriate temporary variables, and evaluate each test and result expression in isolation):

```
ex3 := Three new.
ex4 := Four new.
ex3 test.
ex4 result1.
ex3 result2.
ex4 result2.
ex3 result3.
ex4 result3
```

These examples are from the “Blue Book”, pages 62–66.

Ex 5.3: Augment the methods with extra expressions that write to the System Transcript.

5.4. An Example of Inheritance

In module 3 we introduced the class SpendingHistory as a simplistic model of a person’s spending habits. Suppose we also want a class that is a more complete model of a person’s overall finances — one that includes *income*. Rather than develop a class from scratch we can subclass the existing SpendingHistory class, and consequently *inherit* both its behaviour and data structure. The class description of the new subclass, called FinancialHistory, is specified below:

```
SpendingHistory subclass: #FinancialHistory
    instanceVariableNames: 'incomes'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Financial Tools'
```

Add a new class category called ‘Financial Tools’ and edit the class template so that it appears as above. The class description introduces one extra instance variable incomes. Select **accept** from the <operate> menu.

The class FinancialHistory inherits the instance variables of its superclass SpendingHistory (i.e. cashOnHand and expenditures) and also its instance and class methods. Hence, there is no need to create an additional instance creation method, it’s inherited. However, we do have to create a method setInitialBalance:, which corresponds to the message sent in the initialBalance: instance creation method; this appears below:

```
setInitialBalance: anAmount
    super setInitialBalance: anAmount.
    incomes := Dictionary new.
```

This method uses the pseudo–variable super as the receiver of the message setInitialBalance:. The consequence of sending this message is that the method setInitialBalance: in class SpendingHistory is evaluated, i.e. cashOnHand refers to amount, and expenditure refers to a new empty Dictionary. The only additional operation provided by the method above is that incomes refers to another new empty Dictionary. You should add this method to instance protocol private of class FinancialHistory.

Repeat the above for method totalReceivedFor: in protocol inquiries, for method receive:for: in protocol transactions and for method printOn: in protocol printing. The code is below:

```
totalReceivedFor: reason
    "return the amount received from reason. Answer
    0 if reason is not used for incomes"

    ^(incomes includesKey: reason)
        ifTrue: [incomes at: reason]
        ifFalse: [0]

receive: anAmount for: reason
    "Receive anAmount for a reason and increase the cashOnHand"

    incomes
        at: reason
        put: (self totalReceivedFor: reason) + anAmount.
    cashOnHand := cashOnHand + anAmount.

printOn: aStream
    "print a suitable representation of myself on aStream"

    super printOn: aStream.
    aStream space.
    incomes keysAndValuesDo: [:reason :amount |
        aStream cr.
        reason printOn: aStream.
        aStream tab.
        amount printOn: aStream]
```

Note how the last method printOn: inherits from the superclass, so that all the programmer has to add are the message sends necessary to print out details of income.

To test the new class, type the expressions shown below in a Workspace. Select and evaluate the code using **print it**[1].

[1] Note that this example uses cascaded expressions.

```
| spendingHistory |
spendingHistory := FinancialHistory initialBalance: 800.
spendingHistory spend: 220 on: 'rent';
                    spend: 30 on: 'food';
                    spend: 45 on: 'drink';
                    spend: 20 on: 'petrol';
                    receive: 300 for: 'salary';
                    receive: 50 for: 'expenses';
                    receive: 50 for: 'overtime'.
spendingHistory
```

Ex 5.4: Experiment with the class FinancialHistory, perhaps extending it with extra expressions that write to the System Transcript.

5.5. Summary

To summarise, classes can inherit properties from other classes; this process is repeated to form a "tree" of classes, rooted at Object.

- As stated above, a subclass inherits properties from its superclass.
- Subclasses are used to *refine* the functionality of the superclass, for either the external protocol or the internal implementation.
- Superclasses which should *not* themselves have direct instances are known as abstract superclasses. They are intended to support a partial implementation of features which are completed (differently) in subclasses.
- Each class is a (possibly indirect) subclass of class Object.

5.6. Browsing a Class Hierarchy

We saw in module 3 that the <operate> menu available in the Class Names pane also has a **spawn hierarchy** option. This option creates a *Hierarchy Browser* on all classes in the hierarchy of the selected class (see figure 5.1). A Hierarchy Browser provides the same information access, viewing and editing capabilities as a System Browser, except the information available is for a specified class and its superclasses and subclasses. This is a convenient way of browsing a "vertical slice" of the hierarchy. (Note how the structure of the class hierarchy is reflected in the format of the list of class names.)

A Hierarchy Browser can also be created by evaluating an expression. For example, a Hierarchy Browser that includes class Point can be opened using:

```
HierarchyBrowser openHierarchyBrowserFrom:
                                (Browser new onClass: Point)
```

Ex 5.5: Open a Hierarchy Browser on class FinancialHistory.

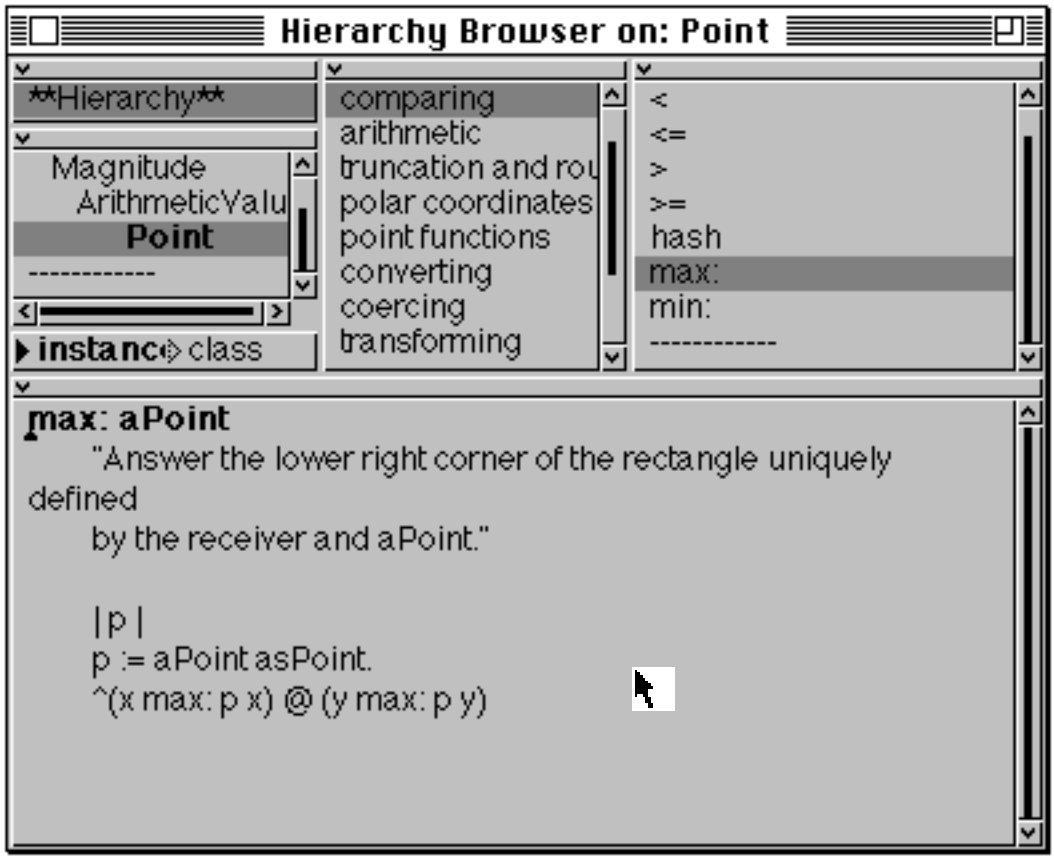

Figure 5.1: A Hierarchy Browser

5.7. Class Variables

We have already seen one type of shared variable in Smalltalk — the global variable. Another type of shared variable is the *class variable*. This is accessible by all instances of the class in which the variable is defined or any of its subclasses. Typical uses of class variables are:

- "Constant" values used by all instances of the class and its subclasses, but which might need to be changed occasionally.
- Private communication between instances.

In general, shared variables provide another mechanism for communication (as well as message passing). The over–use of shared variables is frequently an indication that a solution has not been well designed.

For example, class Date has five class variables, which are declared in the class definition as follows:

```
Magnitude subclass: #Date
    instanceVariableNames: 'day year'
    classVariableNames: 'DaysInMonth FirstDayOfMonth MonthNames
SecondsInDay WeekDayNames'
    poolDictionaries: ''
    category: 'Magnitude-General'
```

The instance variables (day, year) change with each instance, so they can only be accessed directly by the same instance. The class variables, however, keep the same values across instances. So when an instance of Date wants to access the Array of Integers referenced by the DaysInMonth variable, for example, it does not have to send a message to itself. It can use the variable in its methods just as naturally as it would use an instance variable. Objects that are not in the inheritance chain would have to query Date for the information, for example sending it the message nameOfMonth: (see figure 5.2).

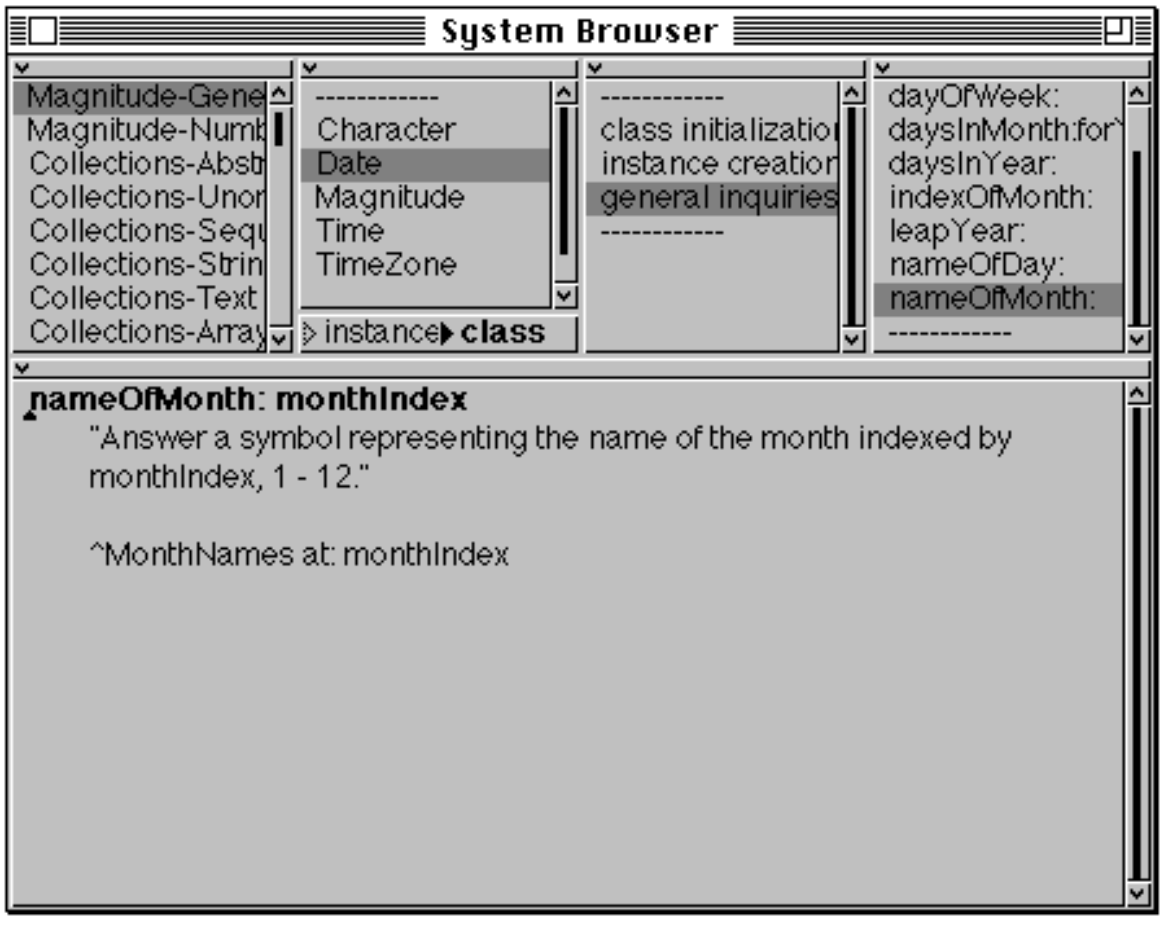

Figure 5.2: The class variable DaysInMonth holds on to a constant

The initial value of a class variable is usually assigned in a class method (normally named initialize) and the corresponding message is typically sent as the final act of creating a class.

Ex 5.6: Browse the class protocol of Date. Try some examples (e.g. Date today).

Ex 5.7: Also explore the instance protocol of Date. Try some examples, including comparing and arithmetic.

Ex 5.8: Create a new class protocol called 'examples'. Install some of your examples there. Remember to document your examples in method comments!

There are a number of examples of the use of class variables to control the appearance and behaviour of the VisualWorks user interface. Here we describe three of them.

5.7.1. ScheduledWindow

By default, VisualWorks will allow you to choose where a new window is positioned. If you always want the window manager to choose where it goes, evaluate[1]

```
ScheduledWindow promptForOpen: false.
```

Ex 5.9: What is the name of the class variable that controls where a new window is positioned?

5.7.2. FileBrowser

The FileBrowser was described in module 3. It's possible to specify an initial "file pattern" (held by the class variable DefaultPattern) for retrieving the file name list by sending the message defaultPattern: to the class FileBrowser, supplying the (argument) pattern in the form of a string.

Ex 5.10: Set the default file pattern of the FileBrowser so that, when opened, the file list contains the contents of the current directory. **Hint**: remember the use of "wildcards".

5.8. Class Instance Variables

The use of class instance variables is little understood, mainly because there are few examples of their use in VisualWorks. As its name suggests, a class instance variable is an instance variable for a class. At first this may seem a little confusing, but you should remember that every object is an instance of some class, thus every class is also an instance of some class (called its *metaclass*). We have seen earlier that instance variables are inherited by classes, similarly, class instance variables are inherited by metaclasses.

It's important to distinguish between the use of a class variable and a class instance variable. For example, suppose class Persian inherits from class Cat. If Cat has a class variable, then Persian has the **same** class variable, i.e. if an

[1] Note that this user preference may be controlled via the Settings Tool, described in module 2.

instance of Persian modifies it, then instances of all subclasses of Cat will refer to that new value.

On the other hand, if Cat has a class instance variable, then all subclasses of Cat (including Persian) have their own copy of that variable.

Although there are not many examples of the use of class instance variables in VisualWorks, there is one which is a good example: class UILookPolicy. This class is an abstract superclass for classes that emulate the "look–and–feel" of various window managers; its subclasses provide specific emulation for Macintosh, Windows, Motif, etc. It introduces three class instance variables: systemAttributes, systemWidgetAttributes and useSystemFontOnWidgets. Each of its subclasses initialize these variables in their respective class initialize methods[1].

Ex 5.11: Open a Hierarchy Browser on class UILookPolicy. Browse references to the class instance variables mentioned above. Where are they initialised?

Ex 5.12: Open a Browser on all those classes that contain class instance variables. Hint: The following code returns true if the receiver has a class instance variable.

```
aClass class instVarNames isEmpty not
```

5.9. Pool Variables

A *pool variable* is a variable whose scope is a defined subset of classes. Pool variables are stored in *pool dictionaries*, instances of PoolDictionary — collections of name/value associations. The variables in a pool dictionary can be made accessible to any class by declaring the pool dictionary in the pool dictionary list of the class definition. Smalltalk, the dictionary of global variables, is itself a pool dictionary that is globally accessible. The class variables of a class are also stored in a pool dictionary that is accessible to the class, its subclass, and instances of the same. However, unlike class variables, pool variables can be referenced by other unrelated classes.

The purpose of a pool dictionary is to provide quick access to the contents of the dictionary, bypassing the usual dictionary look–up mechanism. One common use of pool variables is as a means of providing application–specific constants.

For example, the pool dictionary TextConstants is shared by all classes that deal with the display of text (including classes Text, ParagraphEditor and ComposedText). It includes variables referring to basic character constants, providing the mapping from characters such as <tab>, <CR>, <space>, etc. to

[1] Note that the variables can only be accessed by class methods.

their ASCII equivalents, allowing unprintable ASCII characters to be referenced by name.

For example, class Text declares the pool dictionary TextConstants in its class definition:

```
CharacterArray subclass: #Text
    instanceVariableNames: 'string runs'
    classVariableNames: ''
    poolDictionaries: 'TextConstants'
    category: 'Collections-Text'
```

The keys of the pool dictionary are referenced within Text methods as if they were global or class variables (figure 5.3).

Ex 5.13: Inspect the pool dictionary TextConstants.

A new pool dictionary may be created by declaring the dictionary as a global variable. Pool variables may then be added to the dictionary using the message at: put:. For example:

```
Smalltalk at: #ExamplePoolDictionary put: PoolDictionary new.
```

(This creates a new global variable named ExamplePoolDictionary that refers to a new instance of PoolDictionary.)

```
ExamplePoolDictionary at: #ExamplePoolVariable put: someObject
```

(This creates and adds a new pool variable named ExamplePoolVariable to the dictionary ExamplePoolDictionary.)

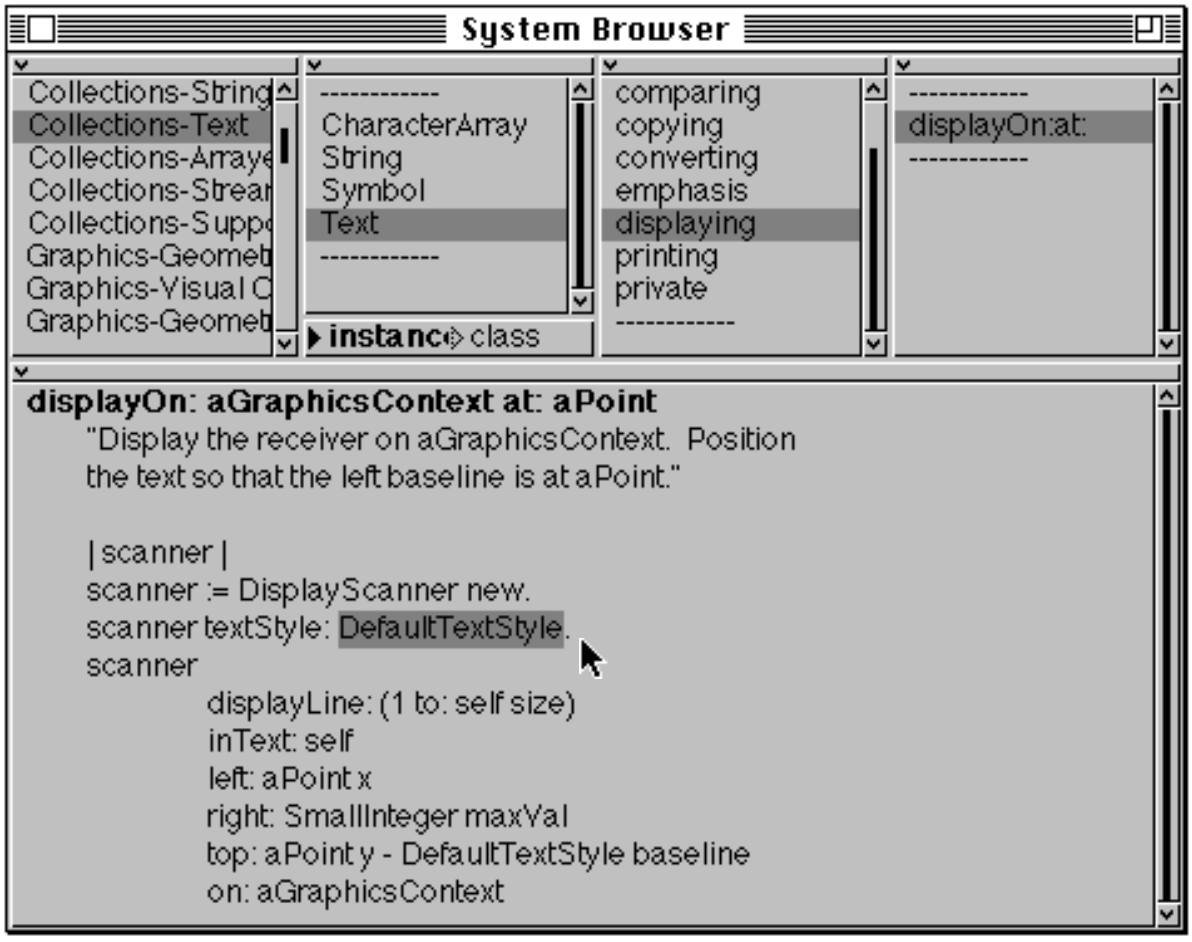

Figure 5.3: The pool variable DefaultTextStyle referenced in the method displayOn:at: in class Text

5.10. Class Protocol

When constructing a class, it's often difficult to arrive at a consistent naming convention for class protocols. If you browse the VisualWorks image you will notice that some protocols become familiar. They are described below:

instance creation	This very important protocol is very common. New instance creation methods to suit subclasses are frequently defined, in order to (for example) initialize the new instance appropriately. Examples include those to create an instance from data in a stream (e.g. readFrom:).
class initialization	When a class has class variables, a class method (by convention, initialize) is required to set up the initial (or default) values of the variables.

examples	Helpful examples to explain the operation of a class are often provided. This is good practice.
documentation	Class methods solely for documentation are sometimes used. Generally, the class comment is a better place for this information. The method guideToDivision in class SmallInteger is an example of a class method provided solely as documentation (the documentation is in double–quotes and typically the last line of the method contains the expression ^self error: 'comment only')
accessing	Class methods to access class variables.
inquiries	General inquiries about information the class encapsulates are provided by many class methods. The method nameOfMonth: in class Date is an example (figure 5.2).
instance management	A class may wish to provide some control over its instances, such as restricting the number or enforcing an ordering. The method currentWindow in class ScheduledWindow is an example.
private	Methods not for general use, provided to support the above protocols.

Ex 5.14: Explore class Time. This is a subclass of Magnitude. Note the class initialization and general enquires methods, and how they use the class variables. Try Time now.