1

The Squeak Object Model

The object model of Smalltalk is simple and uniform, everything is an object. However, this uniformity can still be a source of problem for programmers used to other languages. In this chapter, I present the core concepts and in particular how class are handled as objects. However, I suggest interested readers to refer to books available on http://www.iam.unibe.ch/ ∼ducasse/. Again the audience for this chapter is a programmer of another object-oriented languages such as C++ or Java.

1 The Rules of the Model

The design of the Smalltalk object model is based on a set of simple rules that are applied *uniformly*. The rules are the following ones:

Rule 1. Everything is an object that has some *private* data.

Rule 2. Every object is instance of a class.

Rule 3. A class defines the behavior via *public* methods and the structure of its instances via instance variables which are *private* to the instances.

Rule 4. Each class is inheriting its behavior and structure description from a single superclass.

Rule 5. Objects *only* communicate via message passing (*i.e.*, method invocation). When an object receives a message, the corresponding method is looked up in the class of the receiver, then if not found on this class continues in the class's superclasses.

Rule 6. The class Object is the root of the inheritance tree (in Squeak this is ProtoObject the class that represents objects understanding the smallest set of messages).

Rule 7. Classes are instances too. They are instances of other classes called *metaclasses*.

1.1 Model Precisions

In Smalltalk we have only objects that are instance of classes. Class defines the structure of the instance in terms of instance variables, and methods.

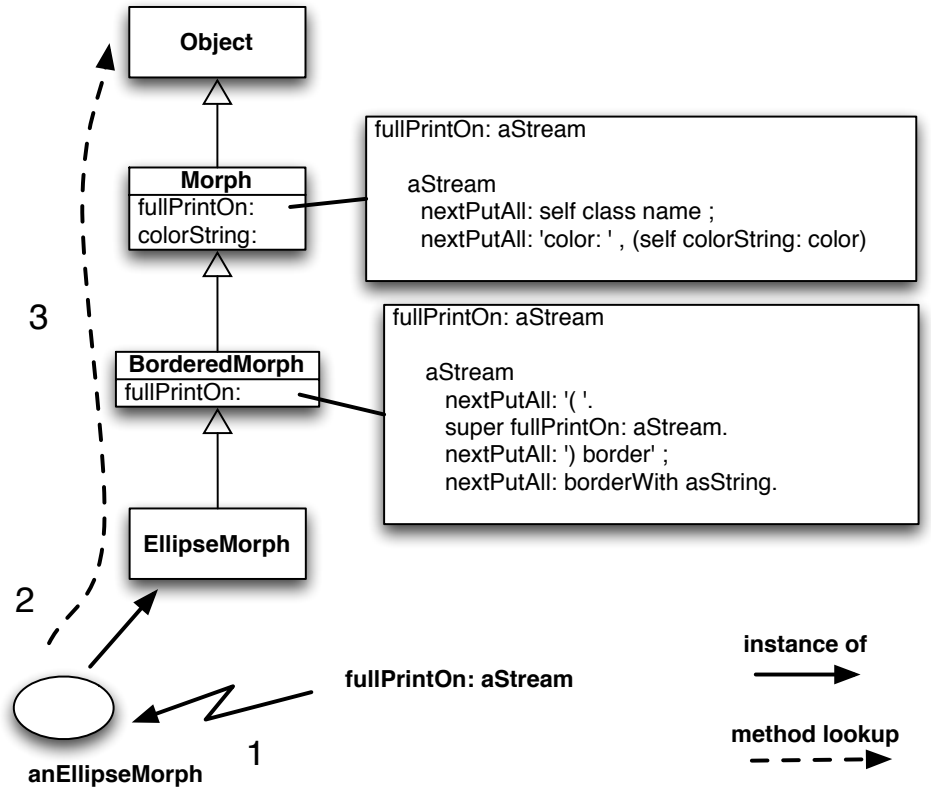

Figure 1.1: Object *only* communicate via message passing (*i.e.*, method invocation). When an object receives a message (1), the corresponding method is looked up in the class of the receiver (2), then if not found on this class continues in the class's superclasses (3). .

Instance Variables. Instance variables are *private* to the *instance* itself, contrary to Java or C++. Even instances of the same class cannot access the instance variables of an object if this one did not define accessors methods. Instance variables are accessible by all the methods of the class and subclasses. Therefore there are *protected* in the C++ jargon. However, we prefer to say that they are private because this is bad style to access directly instance variable from a subclass.

Methods. All the methods are public. A method *always* returns a value using the ^ construct. When not specified explicitly using the ^ construct, the return value of the method is the receiver of the message *i.e.*, self. self (equivalent to this in Java) represents the receiver of the message. The lookup of messages sent to self starts in the class of the receiver (as shown in the Figure 1.1).

When we define a method in a subclass it can hide method in superclasses. To access such hidden methods of a superclass messages should be sent to super and not self. super represents also the receiver of the message but the lookup of messages start in the superclass of the class of the method which issued the super invocation. In Figure 1.2, the method fullPrintOn: is looked up in the class of the receiver EllipseMorph which does not define it, therefore the lookup continues inBorderedMorph the superclass of EllipseMorph. This class defines the method which then gets executed. The expression super fullPrintOn: aStream is then executed. The lookup then starts in

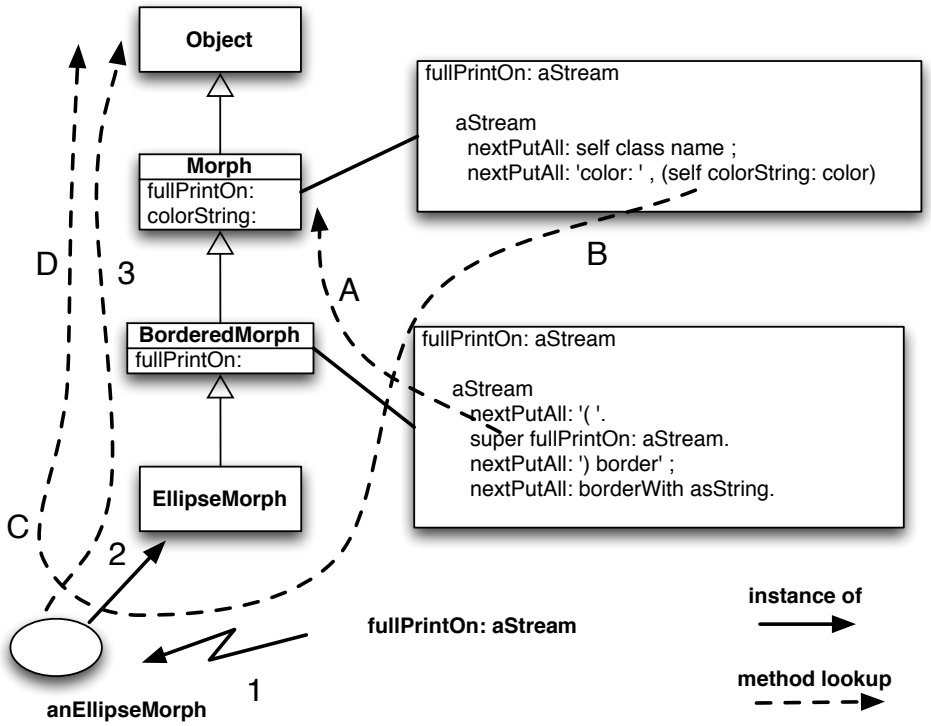

Figure 1.2: super changes the method lookup to start in the superclass of the class that issues the super call (A). self always refers to the receiver, therefore the lookup of the method colorString: invoked in the class Morph in the expression self colorString: aStream (B) starts in the class of the receiver: EllipseMorph (C).

the superclass of BorderedMorph – note that the receiver class is not involved to determine where to start the lookup. This method is defined in the superclass so it is executed.

self is said to be dynamic in the sense that it always represents the receiver of the message. This means that *all* the messages sent to self are looking up by starting in the receiver's class. For example in Figure 1.2 the message fullPrintOn: is sent to anEllipseMorph therefore the lookup of the method colorString: invoked in the class Morph in the expression self colorString: aStream (B) starts in the class of the receiver: EllipseMorph (C).

Note that the model we presented is conceptual in the sense that the virtual machine implementors use all kind of tricks and optimizations to speed up the method lookup. The main point here is to understand what is the semantics of self and super.

Abstractness. To finish with the precisions, a class can be abstract. However, there is no dedicaced construct for that. A class is considered abstract if one of its method contains the expression self subclassResponsibility stating that subclasses have the responsibility to define the method. When such a method is executed an exception is raised.

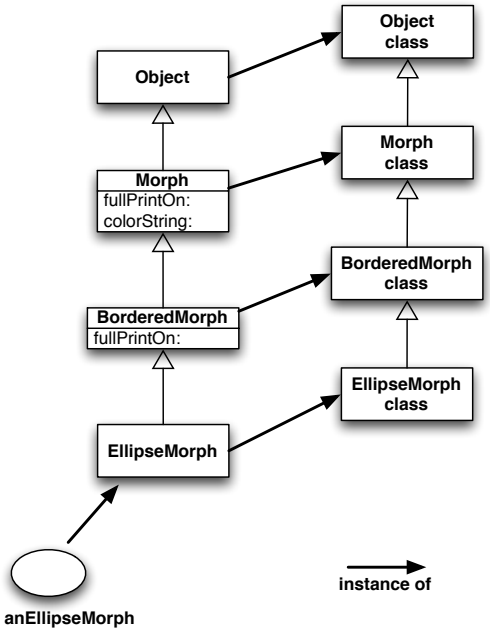

Figure 1.3: Each class is the unique instance of another anonymous class, called a metaclass,*i.e.*, a class whose instances are classes. Metaclasses follow the inheritance of their classes. EllipseMorph is the unique instance of the class EllipseMorph class.

Method 1.1

```
aMethodDeclaredAbstract
  "This is the responsibility of my subclasses to define this method"

  self subclassResponsibility
```

Note that nothing prevents you to create instance of the class having abstract methods. This will work until an abstract method will be invoked.

2 Uniformity Implications: The class side

In the first part of this chapter I stated that the model of Smalltalk is defined by a set of simple rules uniformly applied. Now I look at the implications of this uniformity and in particular how these rules apply to the class themselves. Indeed, since everything is an object instance of a class (rule one and two) then class should also be objects instances of other classes. You guess right: this is exactly the way it is!

Rule 7. In Smalltalk classes are instances of other classes, called metaclasses, *i.e.*, simply classes whose instances are other classes.

Metaclasses are plain normal classes therefore they define the structure (name, superclass, subclasses, method dictionary...) and the behavior (methods new, allSubclasses...) of *classes*. As an

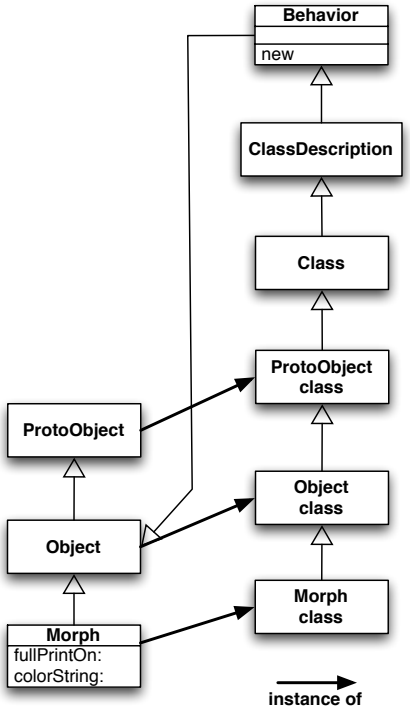

Figure 1.4: The inheritance chain of Morph and Morph class classes.

ellipseMorph is described by the class EllipseMorph defining an instance variable color and methods to draw the ellipse. The class EllipseMorph is described by a class that specifies its structure and methods.

Parallel Inheritance Tree. In fact for composition reasons, a class is the sole instance of an anonymous metaclass whose name is the X class where X is the name of the class. For example EllipseMorph is the sole instance of the class EllipseMorph class. Now inheritance of structure and behavior follow the same rules for classes than for basic objects. Therefore, metaclass instance variable description and method definition is reused via inheritance. All the class management is defined in the class, Behavior which is the essence of a class, ClassDescription which adds instance variable names, method categories, Class and Metaclass which deals with the fact that it has only one instance. The Figure 1.3 presents then the situation: each class is instance of its metaclass and metaclass inheritance follows the one of classes.

The class Behavior which inherits from Object defines the method new and new: that create instances and all the other methods crucial for instance and class management as shown in the Figure 1.4. Therefore when the message new is send to the EllipseMorph class, it is looked up in the class EllipseMorph class class and in its superclasses. Ultimately the method new defined in class Behavior will be executed to create an instance of the class EllipseMorph.

Rule 8. A class X is the sole instance of an anonymous class named X class. The metaclass X class inherits from the metaclass of the superclass of the class X.

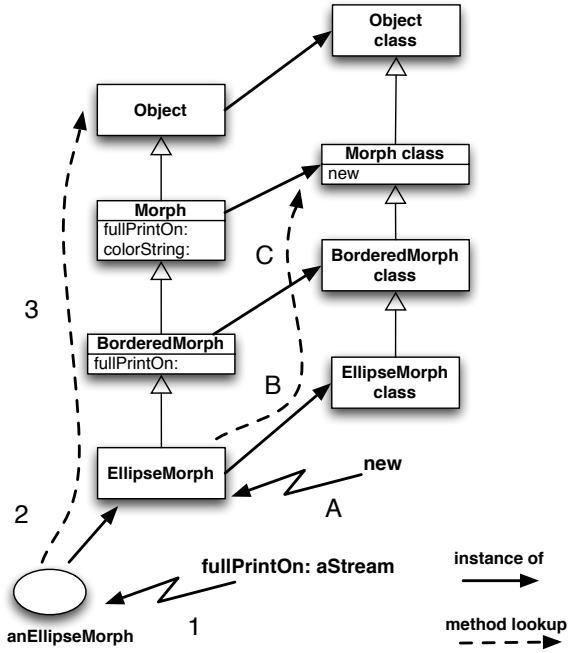

Figure 1.5: There is only one lookup mechanism to resolve all the messages sent. The fact that there is classes and basic objects does not matter. The lookup starts in the class of the receiver and follows the inheritance tree.

3 Class Instance Variables and Class Methods

In fact in Smalltalk there is only one execution model, just applied at two levels: at the instances and classes levels. Class instance variables are just instance variables of the metaclass and class methods are just methods defined on metaclasses.

Class instance variables. We used the term *class instance variables* to refer to instance variables defined by a metaclass that describe classes. For example, the instance variable superclass that describes the superclass of a class is a class instance variable (defined on the class Behavior). Note that class instance variables has exactly the same properties that instance variables: they are private to the instance.

All methods of the class Point can access the instance variable x. Similarly all methods of class can access the instance variable superclass. Instance methods cannot access class instance variable and vice-versa the class methods cannot access the instance variables.

In a first understanding, Java and C++ programmers can consider that class methods and attributes are equivalent to static members. But the uniformity of Smalltalk goes step further by having *exactly the same* semantics for instance and class in terms of methods resolution and visibility. There is then *no* constraints for class methods. They can invoke overriden methods normally as any other methods.

Class Methods. Inheritance and method lookup are exactly handled the same way at the class level that at the instance level. There is no special rules. As stated by the rule 5, when a message is sent to

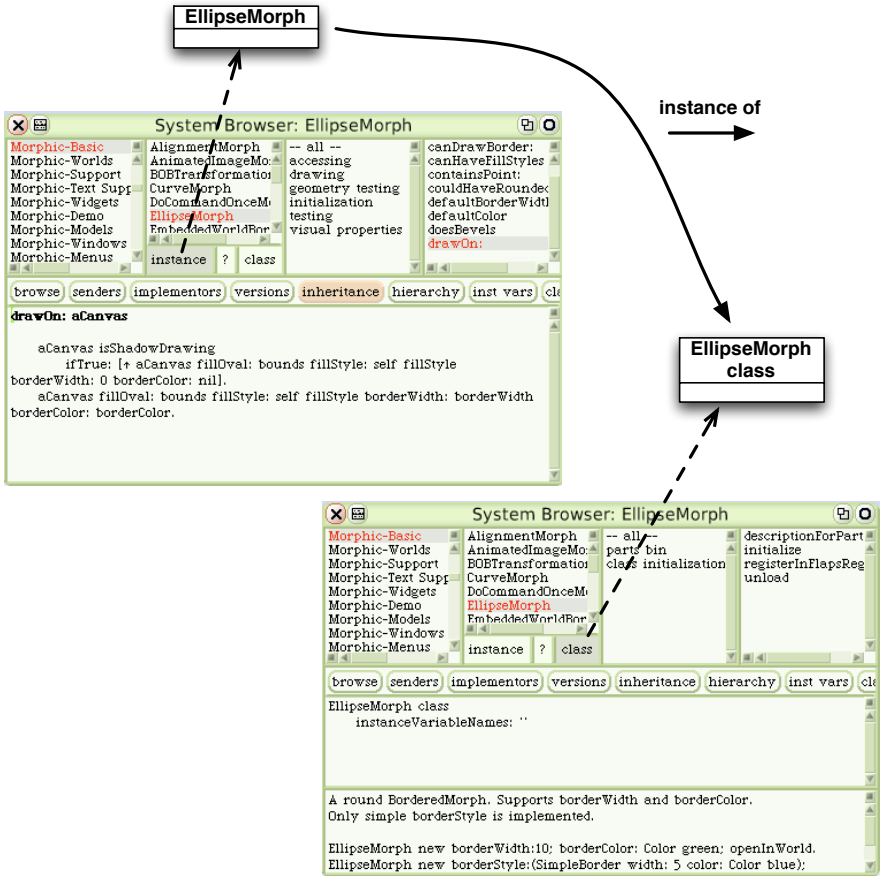

Figure 1.6: Switching between a class and its metaclass.

an object, the method is looked up in the class of the receiver. The fact that the object is a basic object or a class does not matter. The rule applies in all cases as shown in the Figure 1.5. We use the term *class methods* to talk about methods defined on the class side but there is no difference. In fact, there is only one implementation of methods and lookup.

About the Browser. A class and its metaclass are two separate classes. One is instance of the other. However the browser helps us to browse them as shown in Figure 1.6 as if they would be one single class with one static part.

Clicking on the button on the instance button browse the class EllipseMorph: the methods shown are then the ones that will be sent to instances the class EllipseMorph. Pressing the class button browses the class EllipseMorph class: the methods shown are then the ones that will be sent to the class EllipseMorph itself.

An Example: A Singleton. To give you a concrete example, imagine that we want to implement the Singleton pattern, *i.e.*, to ensure that a class only creates one and only one instance. Imagine that we want to have a class named WebServer that should only have one instance. To implement such a pattern the idea is to keep a reference to the first created instance and to give it back on demand. The

implementation is based on the definition of a class instance variable

We create the class WebServer as shown by the class definition class 1.1. Then on the class side we define the instance variable uniqueInstance. This instance variable is then private to the object that represents the class Node.

Class 1.1

```
Object subclass: #WebServer
  instanceVariableNames: 'sessions '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Web'
```

Class 1.2

```
WebServer class
  instanceVariableNames: 'uniqueInstance'
```

To forbid instance creation, we redefine the method WebServer class»new to raise an error (see method 1.2).

Method 1.2

In category instance creation
WebServer class>>new

```
  self error: 'You should use uniqueInstance to get  the unique instance'
```

Then the define the method WebServer class»uniqueInstance that only creates an instance and assign it to the variable uniqueInstance if no instance has been previously created (see method 1.3).

Method 1.3

In category singleton
WebServer class>>uniqueInstance

```
  uniqueInstance isNil
    ifTrue: [uniqueInstance := self basicNew initialize].
  ^ uniqueInstance
```

Optionally we implement the method reset to reinitialize the singleton (see method 1.4).

Method 1.4

In category singleton
WebServer class>>reset

```
  uniqueInstance := nil
```

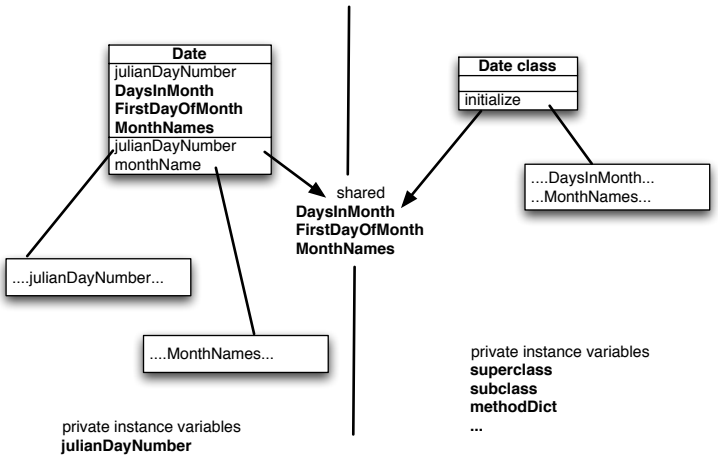

Figure 1.7: Instance and class methods accessing different variables.

4 Other Shared Variables

The Smalltalk object model also proposes some ways to share variables globally or between classes and between instances and classes. These variables are called *global variables*, *class variables*, and *pool variables*.

Global Variables. In Squeak, all the global variables are stored into the unique namespace called Smalltalk, an instance of SystemDictionary. We do not want to tell more on global variables because you should not use global variables.

Class variables: Shared Variables. Sometimes we need to have data that is shared among all the instances of a class and the class itself. This is possible by using *class variables*, shared variables in Smalltalk jargon. The term *class variables* indicates also that the lifetime of the variables is the same of the class. But what the term does not convey is that these variables are shared among all the instances of a class and the classes as shown by the Figure 1.7.

A *class variable* is a variable that has the lifetime of a class and that can be accessed by all the class (and subclasses) methods and by all the instance methods. We illustrate this kind of variable by taking a look at the class Date that represents dates. A class variable is declared using the class definition subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:

A Case Study: the class Date. A date is an object representing the date. Date today returns the object that represents the current day. If we ask this object to print itself we obtain '27 November 2002'. The Figure 1.8 shows a date object in an inspector obtained using the expression Date today inspect. What we see is that a date object only records a number of days. There is no instance variables representing the month names, the day names, the number of days per month, and so on. In fact such information is shared by all the instances of the class and is then represented by the class variables DaysInMonth FirstDayOfMonth MonthNames SecondsInDay WeekDayNames of the Date class definition as shown in the class 1.3.

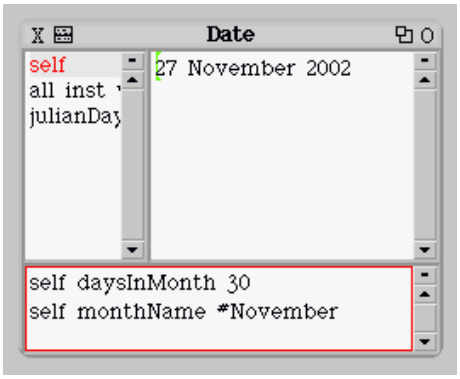

Figure 1.8: A date is an object that only represents number of days, all the information about month names, day names, ...is shared among all the instances

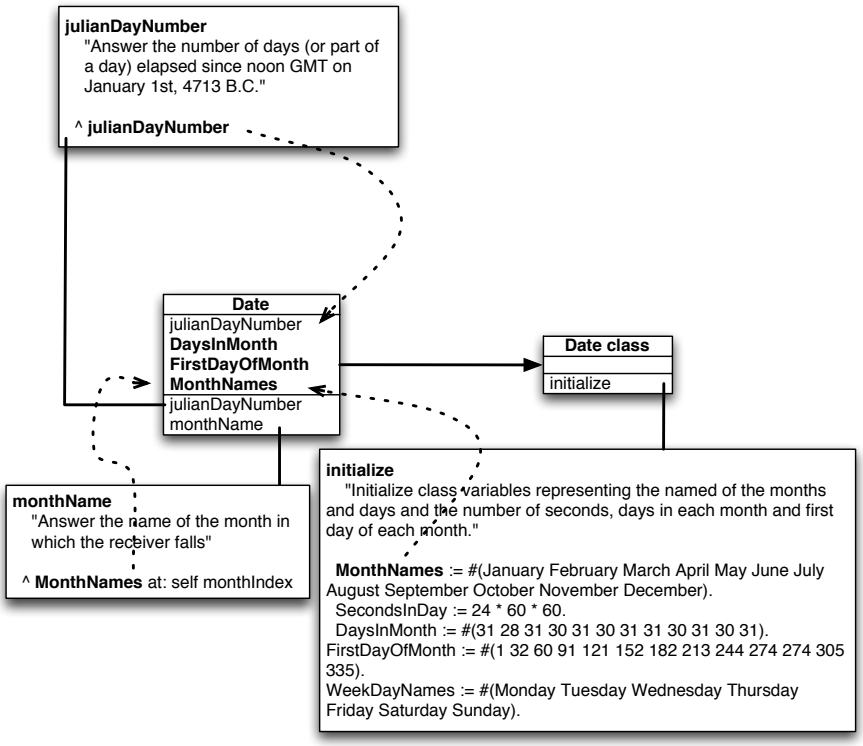

Figure 1.9: Instance and class methods accessing different variables.

Class 1.3

```
Magnitude subclass: #Date
  instanceVariableNames: 'julianDayNumber '
  classVariableNames: 'DaysInMonth FirstDayOfMonth MonthNames
    SecondsInDay WeekDayNames '
  poolDictionaries: ''
  category: 'Kernel-Magnitudes'
```

All instance methods of the class Date (and subclasses) can direclty access the class variables defined by the class Date as shown by the method mth:monthName where the method monthName accesses the class variable MonthNames.

Method 1.5

```
Date>>monthName
  "Answer the name of the month in which the receiver falls."

  ^MonthNames at: self monthIndex
```

In a similar fashion, all class methods can access class variables. The class method 1.6 nameOf-Day: accesses the class variable WeekDayNames.

Method 1.6

```
Date class>>nameOfDay: dayIndex
  "Answer a symbol representing the name of the day indexed by dayIndex, 1-7."

  ^WeekDayNames at: dayIndex
```

Pool dictionary are really static concepts and they should be defined before a method can used them. We strongly encourage you not to use them.

Class Initialization. Now the natural question that arises is how to initialize class variables. As class variables have class lifetime, they are usually initialized by metaclasses. In Smalltalk, class methods named initialize play a special role, they are used to initialize classes. When a metaclass defines a method initialize, this method is automatically called by the system when the class is loaded in memory. The method Date class»initialize method 1.7 shows how the class variables are initialized.

Method 1.7

```
Date class>>initialize
  "Initialize class variables representing the names of the months and
  days and  the number of seconds, days in each month, and first day
  of each month."

  MonthNames  :=
    #(January February March April May June July August September
     October November December ).
  SecondsInDay := 24 * 60 * 60.
  DaysInMonth := #(31 28 31 30 31 30 31 31 30 31 30 31 ).
  FirstDayOfMonth := #(1 32 60 91 121 152 182 213 244 274 305 335 ).
  WeekDayNames := #(Monday Tuesday Wednesday Thursday Friday Saturday Sunday ).
```

PoolDictionaries. The final kind of shared variables is called *pool variables*. Normally you do not need them. We present them just so that you can understand some squeak code using them. PoolVariables are grouped into PoolDictionaries. A pool dictionary defines a group of variables that are sahred among classes (but not their suclasses).

Script 1.1 (*Declaring a pool dictionary)*

```
Smalltalk at: #MyPoolDict put: (Dictionary new at: #myPoolVar put: 3 ; yourself)
```

The class using a pool declares it using the subclass:instanceVariableNames:classVariable-Names:poolDictionaries:category: method as shown in the class definition class 1.4.

Class 1.4

```
Object subclass: #MyClass
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: 'MyPoolDict'
  category: 'MyTest'
```

Then methods of the class MyClass can directly access the variables defined in the dictionary

Method 1.8

```
MyClass>>aMethod

Transcript show: myPoolVar printString ;cr.
"equivalent to"
Transcript show: (MyPoolDict at: #myPoolVar) printString; cr
```

5 What you should have learned

In Squeak everything is an object instance of a class. Classes define the structure via *private* instance variables and the behavior via *public* methods of the class instances. Each class is the unique instance of its metaclass. Class variables are private variables shared by the class and all the instances of the class. initialize class methods are called automatically when a class is loaded in memory.

Further Readings. There exist excellent free online books that go much more in details. They are all available at: http://www.iam.unibe.ch/ ∼ducasse/FreeBooks.html. I suggest you to read: Smalltalk by Example as start.