Object-Oriented Programming

José de Oliveira Guimarães
Departamento de Computação
UFSCar, São Carlos - SP
13565-905
Brazil
jose@dc.ufscar.br

September 17, 2003

Contents

List of Figures

Chapter 1

An Introduction to Object-Oriented Programming

This chapter presents the basic concepts of object-oriented programming[1] using the Green Language [15]. This language was chosen because it is easy to understand and supports almost all the concepts that will be studied in this book. Occasionally we will cite languages C/C++, Pascal, Smalltalk, and Java. This report assumes the reader knows C or Pascal, although someone that does not can understand it.

1.1 Classes, Messages, and Objects

A C/C++ *struct* or a Pascal *record* is used to group related data as the characteristics of a person or geometric figure. For example, a circle would be represented as

```
  /* C/C++ */
struct Circle {
  double x, y, radius;
  };

  { Pascal }
type
  Circle = record
    x, y, radius : real
    end
```

One can declare a Circle variable and handle its fields:[2]

```
  /* C/C++ */
Circle c;
c.x = 0;
c.radius = 10;

  { Pascal }
var c : Circle;
...
c.x:= 0;
c.radius:= 10;
```

[1]The term "object-oriented programming" is historically used not only for programming but also for everything related to object orientedness.

[2]A field is a variable declared inside the *struct* or *record*.

Of course, it has not been shown the context in which the code above is used. Variable `c` can be passed as a parameter to a procedure,[3] returned by a procedure, assigned to another variable, and so on. A real world entity, a circle, is represented through a single abstraction: a *struct* or a *record*. The program considers `c` as a single entity, not three.

The *struct* and *record* declarations of `Circle` are *type declarations*. They create a new type, `Circle`, which can be the type of variables, parameters, etc. Variable `c` has all the fields declared in its type. Then, variable `c` has field `x`, accessed through the syntax "`c.x`".

Note the type does not exist at runtime. It exists only at compile time. What exists at runtime is the piece of memory with the same structure as the type. This piece of memory is handled through a variable. We will soon discover that this piece of memory is called "object".

Structures and records have fields that are variables. A natural extension of them would be structures and records with fields that are procedures:

```
  // C++
struct Circle {
  double x, y, radius;
  double area() { return pi*radius*radius; }
  void   draw() { /* draw the Circle */ }
  };

  { Some object-oriented Pascal }
type
  Circle = record
    x, y, radius : real;
    function area : real;
      begin
      area:= pi*radius*radius
      end

    procedure draw;
      begin
        { draw the circle }
      end
    end
```

In fact, the structure above is legal in C++ and the record could be legal in an object-oriented extension of Pascal. The procedures are used through "`.`" as the fields that are variables:

```
  // C++
Circle c;
c.radius = 10;
a = c.area();   // calls area of structure Circle
c.draw();
```

1.2 Classes in Green

From this point onwards, we will employ language Green [15]. A complete program in this language is shown in Figure 1.1. This program has a class `Store` and a class object `Main`. Class objects will be seen later on. A class is like a C++ structure with procedures. A class in Green starts with "`class classname`" and ends with keyword "`end`". A class object is a classless object that represents a class — this will be explained later. The program execution starts at method `run` of a class object specified

[3]Procedure, function, subroutine, and routine will be considered synonymous.

```
class Store
    proc init()
      begin
      end
  public:
    proc set(pn : integer)
      begin
      n = pn;
      end
    proc get() : integer
      begin
      return n;
      end
  private:
    var n : integer;
end

object Main
  public:
    proc run()
        var s, p : Store
      begin
      s = Store.new();
      p = Store.new();
      s.set(5);
      p.set(7);
      s.n = 5;
      Out.writeln( s.get(), "  ", p.get() );
      end
end
```

Figure 1.1: A Green program

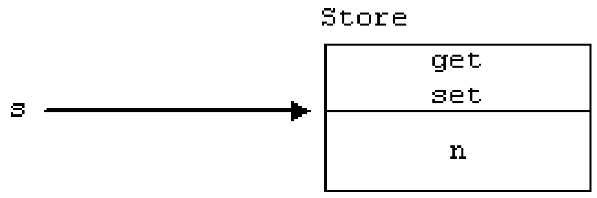

Figure 1.2: A `Store` object referenced by variable `s`

by the programmer at link time. For this example, assume the execution starts at method `run` of class object `Main`.

The procedures of a class, like `set` and `get` of `Store`, will be called *methods*. Variable `n`, declared inside `Store`, will be called *instance variable*.

Class `Store` has a public and a private section. Each one begins with "`public:`" or "`private:`". The meaning of these sections will soon be explained. Before the public section there is a constructor named `init`. All constructors have this name and are called to initiate an object just after its creation. In class `Store`, the constructor do nothing and could be removed. Constructors will be fully described in Section 1.8.

Note methods `set` and `get` use variable `n` declared after them. This is legal and works as expected.

1.3 Objects and Variables

We will study some statements of method `run` in order to introduce some basic object-oriented concepts. The declaration

```
    var s, p : Store;
```

creates two variables, `s` and `p`, of type `Store`. These variables work like pointers, which means memory for them need to be dynamically allocated.

In Green, a variable will behave like a pointer, as `s` and `p`, whenever its type is a class. If its type is a basic type as `integer`, `char`, `boolean`, `real`, etc, then the variable *is not* a pointer.

Since `s` is a pointer, it should point to a dynamically allocated memory, which is made in the first statement of method `run`,

```
    s = Store.new();
```

The call "`Store.new()`" creates an *object* of class `Store`. This object occupies some bytes of memory, exists only at runtime, and has everything that is declared in its class. Then, `s` points to an object that has methods `get` and `set` and instance variable `n` — see Figure 1.2. An object is an instance of a class. That is why variable `n` is called *instance variable* — it is a variable of an instance (object) of class `Store`. That means an instance has a field `n`. This same reasoning applies to the methods. One could say "instance method" although we usually say just "method".

Class `Store` is a template, a model for its objects. An object of `Store` has everything class `Store` has. The class exists only at compile time and the object only at runtime. There is no program memory associated to a class. But there is always a piece of memory associated to an object.

The second statement of method `run`,

```
    p = Store.new();
```

makes `p` point to *another* object created at runtime. This new object has the same class as the previous one but it occupies another memory.

A class corresponds to an abstraction or model of concrete things, the objects of the class. The

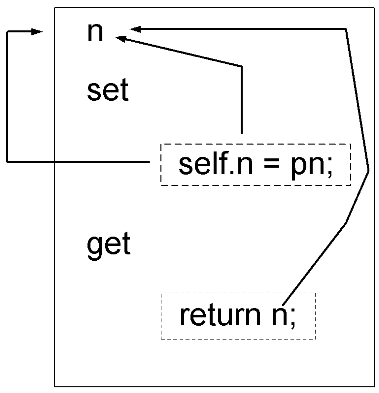

Figure 1.3: References to `self` in object `s`

concept "Car" would be a class. The car that I drive would be an object. The concept is a model. My car is concrete, it really exists. There is a characteristic of cars called color. My car has a value associated to this characteristic — it is blue.

Message Sends

The third statement of method `run`,

```
    s.set(5);
```

is a *message send*. We say message "`set(5)`" is sent to object pointed to by `s`. In general we say "object `s`" instead of "object pointed to by `s`" or "object referred by `s`". Variable `s` is only the name used to access the object. Therefore, it is better to say "object referred by `s`" than "object `s`", although we will usually use the last form. Note we use the verbs point and refer as synonymous.

Sometimes we say that "`s.set(5)`" is a *method call*. This form is worse than to say "message send" because the object-oriented flavor is lost — after all, the word "call" is also used in the jargon of procedural languages.

Each class defines its own scope for methods. Two classes may have two methods with the same name and one will never be taken as if it were the other.

At runtime, method "`set`" of `Store` is called in statement "`s.set(5)`". Formal parameter `pn` receives `5` as usual. Then the method body,

```
    n = pn;
```

is executed.

But what exactly does this means ? Variable `n` is a variable declared in class `Store`, which exists only at *compile time*. Parameter `pn` is created at runtime as expected. Then "`n = pn`" uses a runtime and a compile time variable, which is nonsense.

In fact, the `n` of this statement is not the `n` declared in class `Store`, but the `n` of object `s`. Since `set` was called through `s` in "`s.set(5)`", `set` will use the instance variables of `s`. Any reference to variable `n` inside `set` means "`s.n`".

In the same reasoning line, "`p.set(7)`" will call method `set` of `Store` which will use "`p.n`". A method always uses the variables of the *message receiver*, which is `p` in "`p.set(7)`".

Self

Method `set` may have been written as

```
proc set(pn : integer)
  begin
  self.n = pn;
  end
```

This method is completely equivalent to the one of Figure 1.1. `self` is a special variable that always point to the message receiver. Then, in "`s.set(5)`", `self` refers to the same object as `s`. Method `set` becomes equivalent to

```
    s.n = pn;
```

See Figure 1.3. In the same way, the message send "`p.set(7)`" will call method `set`. Inside `set`, `self` and `p` will refer to the same object. Then "`self.n = pn`" will be equivalent to "`p.n = pn`".

Although the `self` concept is simple, the reader is invited to think of it deeply. This is extremely important.

1.4 Information Hiding

The fifth statement of method `run` of Figure 1.1 is

```
    s.n = 5;
```

This causes a compile-time error. The compiler will say `n` is a private instance variable and cannot be used outside `Store`. Anything declared inside the private section can only be used in the class body. The public methods can be used anywhere in the program, but always with an object. One cannot call a method without using an object. Methods do not exist freely as procedures in procedural languages. They are always attached to objects at runtime and can only be called through message sends.

In Green, the public section cannot declare instance variables. The private section can declare methods and instance variables.

To use data or information hiding (IH) is to prevent the direct access to instance variables of objects. If IH is used, instance variables can only be changed by the methods of the class. Green enforces IH and then prohibits the code

```
    s.n = 5;
```

One should use

```
    s.set(5);
```

instead. Instance variables should not be accessed outside their classes. To understand the reason for this restriction, let us study class `Stack` of Figure 1.4. In Green, we can only use objects of this class by calling its methods:

```
...
var s : Stack = Stack.new();
s.create();
s.push(1);
...
```

A variable can be declared, in Green, anywhere a statement may appear. This feature was used above to declare `s` among regular statements. The variable should receive a value in the declaration.

If information hiding were not enforced, we could have a code like this:

```
...
var s : Stack;
s = Stack.new();
s.top = -1;
vet = array(integer)[].new(Max);
++s.top;
s.vet[s.top] = 1;
```

The last two statements are equivalent to "`s.push(1)`".

A comparison of these two equivalent codes reveals the reasons for using information hiding:

- it makes it easy to change the class instance variables. A pretty common maintenance task is to change the class instance variables while keeping the class interface. That is, the instance variables

```
class Stack
  public:
    proc create()
      begin
      top = -1;
        // allocate an array
      vet = array(integer)[].new(Max);
      end
    proc push(x : integer)
      begin
      if top + 1 < Max
      then
        ++top;
        vet[top] = x;
      endif
      end
    ...
    // other methods
  private:
    var top : integer;
        vet : array(integer)[];
end
```

Figure 1.4: Class `Stack` implemented using an array

are replaced or modified, the method bodies are re-codified but the method interfaces[4] are left untouched.

As an example of this transformation, class `Stack` may be re-implemented using a linked list. The bodies of all methods need to be rewritten. But the interface of method `push` will continue to be

```
push(integer)
```

A code that uses objects of `Stack` need not to be modified since it knows only the `Stack` interface, which remains the same.

That would be different if Green allowed direct access to instance variables, breaking the information hiding. If the `Stack` instance variables were changed (from an array representation to a linked list), all code that access them directly should be changed. That could mean a lot of work if `Stack` variables were accessed all throughout the program;

- it increases the abstraction level of programming. To abstract is to discard all irrelevant details to our objective. So, let us compare

  ```
  s.push(1);  // code 1
  ```

  to

  ```
  ++s.top;    // code 2
  s.vet[s.top] = 1;
  ```

  Code 1 uses information hiding, code 2 does not.

  Code 1 is more abstract because it discards irrelevant details to someone who just wants to push `1` into stack `s`. These irrelevant details are clearly shown in code 2: we need to know that there are variables `top` and `vet`, that `vet` is an array, and that `top` is initiated with `-1` (since it is incremented before used);

[4]Also called method signatures or headers. It consists of the method name, its parameter types and return value type, if one exists. So, the interface of method `push` is "`push(integer)`".

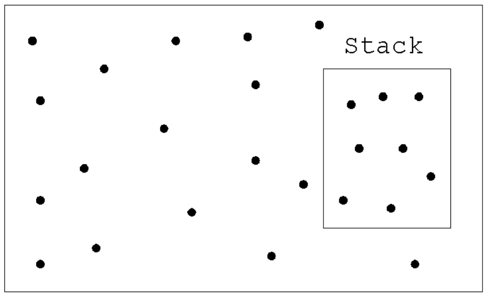

Figure 1.5: `self`

- it is easy to protect the data when it is accessed in just one place, the class. In method `push`, a test is made to check if there is space for a new stack element. If a call to push is replaced by the code
  ```
      ++s.top;
      s.vet[s.top] = x;
  ```
  in dozens of places spread throughout the program, the probabilities are that the programmer will forget to check the correct use of the stack. In the above code, no test is made to check if there is space in the stack.

The rectangle of Figure 1.5 represents a program that uses class `Stack`. The block dots show the places where instance variables of `Stack` are accessed. Of course, this program does not use information hiding since there are dots outside class `Stack`, represented as a small rectangle. If one changes the `Stack` instance variables, all places represented by the black dots should also be changed. In all these places the data should also be protected so it does not receive illegal values. That reduces the area of use of `Stack` instance variables to a single and well-known file, that containing class `Stack`. Only that file should be checked after a change in the instance variables.

The rectangle of Figure 1.5 represents.

Breaking Information Hiding

Even though some languages such as Green enforces information hiding by prohibiting the declaration of public instance variables, the programmer can still code classes that violate it. There are a lot of ways to do that:

- a class may have a public method that reveals its private part. As an example, class `Stack` could have a method

  ```
  proc getArray() : array(integer)[]
    begin
    return vet;
    end
  ```

  It is then possible to access the stack implementation directly, possibly damaging it:
  ```
      s.getArray()[100000] = 0;
  ```

- a class may define a public method that accesses its private part without any control. For example, class `Stack` could have methods

```
class StorePrint subclassOf Store
  public:
    proc print()
      begin
      Out.writeln( get() );
      end
end
```

Figure 1.6: An example of inheritance

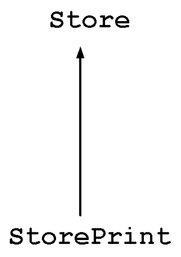

Figure 1.7: Representation of inheritance of `Store` by `StorePrint`

```
proc get(i : integer) : integer
proc set(i, x : integer)
```

to `get` and `set` stack elements. Then someone could change the first stacked element without popping off all other elements, a operation not supported by the abstract data type stack.

These two items show us another problem: some public methods may be targeted for specific data structures. Methods `get` and `set` are easily and efficiently implemented if class `Stack` uses an array to store the stack elements. But then it becomes difficult to change the array by a linked list, since `get` and `set` are inefficient when used with such a data structure. The same problem occurs with method `getArray` of the first item.

Languages C++ and Java do not demand the use of information hiding — one can declare public instance variables. However, the access to private instance variables outside the class body is prohibited. These languages do not enforce IH for efficiency reasons — they exchange easy maintenance for runtime speed. Of course, it is fast to access a variable directly as in "`s.n = 5`" than to call a method "`s.set(5)`".

1.5 Inheritance

Inheritance is a key element in object-oriented programming. It allows a class to inherit methods and instance variables from another class. Inheritance is indicated in Green by keyword `subclassOf` as shown in Figure 1.6. Class `StorePrint` inherits from class `Store`.

Class `StorePrint` has everything `Store` has plus the methods and instance variables it declares by itself. Then class `StorePrint` has instance variable `n` and methods `get`, `set` (inherited), and `print`. Inheritance is graphically represented as shown in Figure 1.7. The inherited class, `Store`, is called the *superclass*. The inheriting class, `StorePrint`, is called the subclass[5] — see Figure 1.8. To make reasoning easy, always remember the superclasses are above. That is, the classes that donate features to subclasses are always at the top. Superclasses are also called *ancestors* and subclasses are called *descendents*.

In this book we will also use the UML [] representation of classes, which is shown in Figure 1.10. UML is further explained elsewhere. A textual inheritance representation is made by indenting subclasses as shown in Figure 1.11. The subclass appears two columns to the right and below a superclass. Then `Polygon` and `Circle` are subclasses of `Figure` and `Rectangle` and `Triangle` are subclasses of `Polygon`.

[5]The class that has more things is sub and the class with less features is super.

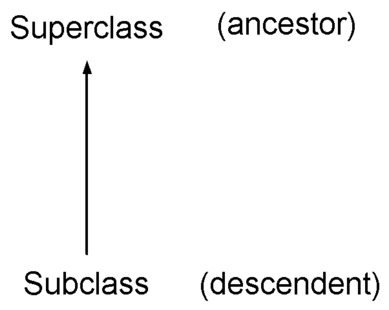

Figure 1.8: Jargon of inheritance

```
object Main
  public:
    proc run()
        var sp : StorePrint;
      begin
      sp = StorePrint.new();
      sp.set(3);
      Out.writeln(sp.get());
      sp.print();
      end
end
```

Figure 1.9: Using `StorePrint`

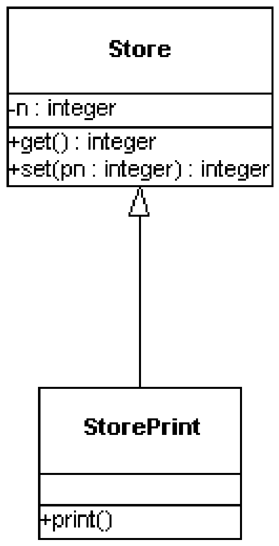

Figure 1.10: UML representation of `Store` and `StorePrint`

```
Figure
  Polygon
    Rectangle
    Triangle
  Circle
  Elipse
```

Figure 1.11: A hierarchy of classes

A class object `Main` that uses `StorePrint` is shown in Figure 1.9. In method `run`, variable `sp` receives messages "`set(3)`" and "`get()`". The corresponding methods are inherited by the `sp` class and therefore the code is correct. It could be different. If `run` had a statement

```
sp.show();
```

then there would be a compile-time error. To explain that, note *the declared class* of `sp` is `StorePrint` because of the declaration

```
var sp : StorePrint;
```

At every message send, the compiler searches for the correct method in the declared class of the variable. If it is not found there, the search continues in the superclass, superclass of the superclass, and so on. If the method is definitely not found, a *compile error* is signaled.

In "`sp.show()`", the *compiler* searches for a method `show` taking no parameters in the *declared* class of `sp`, `StorePrint`. Since no method `show` is found, the search continues at `Store`, the superclass. Again there is no `show` method and no superclass. At this point, the compiler issues an error message in the screen or standard output.

Method `print` of `StorePrint` has a call to method `get` — see Figure 1.6. This call *must* use a message receiver, as there is never a call to a method without a receiver. Since there is no receiver in this case, the default one is used, `self`. Then `print` is equivalent to

```
proc print()
  begin
  Out.writeln( self.get() );
  end
```

At runtime, the call

```
sp.print();
```

of method `run` (Figure 1.9) will call method `print` of `StorePrint`. The message send "`self.get()`" will cause a *runtime* search for a method `get()` beginning in the class of `self`. The class of `self` is the class of `sp`, since `self == sp`. The class of `sp` is `StorePrint` since this variable was initiated with "`StorePrint.new()`".

Then the search for `get()` begins at class `StorePrint`, fails, and continues at class `Store` where the method is found and called. This example shows that a message send to `self` in a subclass may call a superclass method. Note we used a *runtime* search to find which method would be called. This topic will be further explained in Section 1.6.4 and 1.6.5.

The Meaning of Inheritance

Inheritance is used to express relationships of the kind "is-a". That is, "an object of a subclass is-an object of the superclass". For example, it is correct, in general, to make a class `Student` inherit from class `Person` because "a student is a person". Class `Worker` may also inherit from `Person` because "a worker is a person". Other examples are given below.

- A public server is a worker.

- A public university employee is a public server.
- A Boing 747 is a plane.
- A plane is a vehicle.
- A digital watch is a watch.
- A graphical text panel is a panel.
- A thesis is a document.
- A PhD thesis is a thesis.
- A rectangle is a polygon.
- A company owner is a worker.

But ... is really a worker a company owner ? They have special rules for salary increases, vacations, and so on because they make the rules themselves. And other operations cannot be applied to them. For example, they cannot be dismissed. Then a company owner has a lot in common with workers but there should be no inheritance relationship among these two concepts. Problems like this are discussed in subsection 1.7.6.

1.6 Polymorphism

Polymorphism means multiple faces. Something polymorphic can be used in more situations it was originally intended to be. Or it has more than one meaning, which varies according to the context. Polymorphism in object-oriented programming is always related to types and code reuse. It applies to values, variables, parameters, and methods.

1.6.1 Polymorphic Values

A polymorphic value has more than one type, as object `nil` in Green. This object belongs to class `Nil` and represents the null object, being equivalent to `NULL` in C++ or `null` in Java. Object `nil` can be assigned to variables of any of the non-basic classes. For example, the code

```
var p : Person;
var f : Figure;
...
p = nil;
f = nil;
```

is correct. But

```
    var i : integer;
    ...
    i = nil;
```

is illegal.

`nil` has infinite types because in an assignment “`x = y`” variables `x` and `y` should have the same declared type. Then value `nil` has types `Person`, `Figure`, and so on. Potentially, `nil` has really infinite types.

1.6.2 Polymorphic Variables and Parameters

The assignment of an object of a subclass to a superclass variable is legal. Assuming `Circle` is subclass of `Figure`, the code

```
    var f : Figure;
    var c : Circle = Circle.new(3, 2, 7);
    f = c;
```

is legal. The last assignment is of the kind

Class = subclass

since the declared types[6] of `f` and `c` are `Figure` and `Circle`, a subclass of `Figure`. The correctness of the assignment "`f = c`" comes from the fact that every `Circle` is a `Figure`. That is, every `Circle` object can be considered a `Figure` object. A subclass should be related to its superclass by the relation "is-a" — see Section 1.5.

Variable `f` is polymorphic because it can refer to objects of potentially infinite classes — `Figure` and all its subclasses.

The relation "is-a" between a subclass and its superclass is only demanded because assignments of the kind

Class = subclass

are legal. In an assignment "`f = ...`", it is required that the "`...`" be a `Figure`, the declared type of variable `f`. But a circle is also a figure as guaranteed by the inheritance relationship between `Circle` and `Figure`. Then a `Circle` object can be assigned to a `Figure` variable. An alternative view of this assignment is "if someone asks me a `Figure`, I can give her a `Circle`, since a `Circle` is a `Figure`". If someone asks me a `Fruit`, I can give her an `Orange` or an `Apple`, making the code below legal. Assume `Orange` and `Apple` inherit from `Fruit`.

```
var aFruit : Fruit;
...
var anApple : Apple = Apple.new();
...
aFruit = anApple;       // ok
AFruit = Orange.new();  // ok
...
```

1.6.3 Polymorphic Methods

Parameter passing follows the same rules as assignments. A method

```
    proc insert( f : Figure )
```

can accept as real parameter any object of `Figure` or its subclasses. Method `insert` is polymorphic because one of its parameters (in fact, its sole parameter) is polymorphic.

1.6.4 Search for Methods in Message Sends

Some points on the behavior of polymorphic variables at runtime need to be explained, which will be made by studying the following example.

```
var f : Figure = Circle.new(3, 2, 7);
f.draw();
```

The declared class of `f` is `Figure` but it refers to a `Circle` object at runtime. When message `draw` is sent to `f` at runtime, a search for `draw` is made in class `Circle`, where it is found and called. If it were not found, the search would continue in the superclass, the superclass of the superclass, and so on. Note each class in Green has at most one superclass and therefore there is no choice in which superclass to continue the search.

[6]The declared type of a variable is the type used in its declaration.

compile time	run time
declared class of the variable	class of the object the variable refer to

Figure 1.12: Where does the search for a method begin ?

```
var f : Figure;
...
if rounded
then
  f = Circle.new(0, 0, 7);
else
  f = Rectangle.new(0, 0, 5, 10);
endif
f.setRadius(13);
...
```

Figure 1.13: A compile-time type error

The search for a method is always made at runtime beginning in the class of the object (`Circle`) and not in the declared class of the variable (`Figure`). The object class is an information that in general cannot be known at compile time. That is, the compiler (and us) cannot always know the class of the object a variable refer to. The variable may refer to an object of a subclass of its declared class.

Since the object class is not known at compile time, a search for a method at runtime in the object class could fail. But that will never occur. The compile time rules plus the runtime search algorithm guarantee a method will always be found at runtime. The idea of that is simple: in "`f.draw()`" the compiler checks if the declared class of `f` or its *superclasses* have a `draw` method. If they do not, there is a *compile-time* error. Then at runtime the declared type of `f`, `Figure` or its superclasses will have a `draw` method. Variable `f` may point to an object of a subclass of `Figure` at runtime. The runtime search for `draw` will begin at this subclass and proceed towards `Figure` and its superclasses. Then `draw` will be found at least in class `Figure` or its superclasses. For sure there is a `draw` method in one of these classes because the compiler guaranteed that. If there was not one, there would be a compile error and the program would not be running. Table 1.12 shows the classes in which the searches at compile and runtime begin. The understanding of these searches is important when programming. We can know when a message send is legal at compile time and what occur at runtime when the message is really sent to the object.

The compiler may reject code that would be correct at runtime but looks incorrect at compile time. For example, consider this:

```
var f : Figure = Circle.new(0, 0, 7); // radius is 7
f.setRadius(13);
```

An unlucky radius change. Although `f` will refer to an object `Circle` that will have a `setRadius` method at runtime, the compiler will point an error at "`f.setRadius(13)`" because the declared class of `f`, `Figure`, does not declare a `setRadius` method. A language could consider legal the code above. But then it should have tricky type rules and it would certainly be difficult to understand.

The compiler cannot be sure of the runtime class of an object because it cannot foresee the control flow. As an example, in the code of Figure 1.13 the compiler cannot tell which `if` branch will be executed at runtime. Therefore it does not know the class of object `f` in the message send "`f.setRadius(13)`".

1.6.5 Polymorphism in Messages to `self`

There is another subtle kind of polymorphism that occurs when messages are sent to `self`. In Figure 1.14, method `getTime` sends two messages to `self` to call `getHour` and `getMin`. Method `getHour` (and `getMin`)

```
class Clock
    proc init(hour, min : integer)
      begin
      self.hour = hour;
      self.min = min;
      end
  public:
    proc getTime() : String
      begin
      return self.getHour().toString() + ":" +
             self.getMin().toString();
      end
    proc getHour() : integer
      begin
      return hour;
      end
    proc getMin() : integer
      begin
      return min;
      end
    ...
    // other methods
  private:
    var hour, min : integer;
end
```

Figure 1.14: Class `Clock`

```
class Clock12 subclassOf Clock
  public:
    proc getHour() : integer
      begin
        //  % is the module operator
      return super.getHour()%12;
      end
end
```

Figure 1.15: Class `Clock12` overriding `getHour`

returns an integer converted to string by `toString`. A basic value, as `13`, can be converted to a string, `"13"`, by sending to it message `toString`:

```
var s : String = 13.toString();
```

Then basic values are objects[7] in Green.

The "+" in `getTime` is the string concatenator. A typical return value of this method would be "`13:52`". The important aspect of `getTime` is that it calls other class methods through `self`. By overriding these other methods in subclasses, we can indirectly change `getTime` in class `Clock`. To see that, we will use class `Clock12` of Figure 1.15 which considers that hours cannot be greater that `11`. In `Clock`, `0 <= hour <= 24`.

In the code

```
var c12 : Clock12 = Clock12.new(13, 52);
Out.writeln( c12.getTime() );
```

method `Clock::getTime` will be called. Variable `self` will be bound to `c12`, a `Clock12` object. When message "`getHour()`" is sent to `self`, the search for a `getHour` method will start at class `Clock12`, in which a method `getHour` is found and called. Method `getTime` will return

```
"1:52"
```

since `13` is converted to `1`.[8]

1.7 More on Inheritance

1.7.1 Inheritance and Information Hiding

A subclass knows its superclass but a superclass does not know who are its subclasses. That is, a superclass has no information on its subclasses, which can be created regardless of its permission and knowledge.

Then, a subclass should never access the private part of its superclass. Suppose this occurs and the superclass instance variables, which are in the private part, are renamed, deleted, or changed. The subclasses become invalid. The programmer that changed the superclass has no control over the subclasses. Maybe they were made by other programmers or people outside her group or company. All these people should modify their classes.

In fact, the subclasses have no permission to access the superclass instance variables in Green, C++, or Java. But they do in Smalltalk.

1.7.2 Method Overriding and Calls to `super`

A subclass may redefine an inherited method as shown in Figure 1.16. We say method `get`, defined in `Store`, was *overridden* or *redefined* in `StorePrint`. Method `get` of `StorePrint` has a call

```
super.get();
```

[7] In fact, a restricted kind of objects.

[8] $13\%12 == 1$

```
class StorePrint subclassOf Store
  public:
    proc get() : integer
      begin
      return super.get() + 1;
      end
    proc print()
       begin
       Out.writeln( get() );
       end
end
```

Figure 1.16: Overridden of method `get`

```
class StoreLess subclassOf StorePrint
  public:
    proc get() : integer
      begin
      return super.get() - 1;
      end
end
```

Figure 1.17: A subclass overridding `get`

This is a tricky statement. It means "send message `get()` to `self` but do the search for the method at compile time and start the search at the superclass".

Calls to `super` are tricky because they are message sends to `self`, although the syntax[9] does not make that clear. And the search is made at compile time, differently from every other message send. The call "`super.get()`" will then call method `get` of `Store` on `self`.

The code

```
var sp : StorePrint = StorePrint.new();
sp.set(1);
sp.print();
```

will be used to study the overridden of `get`. In the last message send, the following sequence of calls happens:

- `StorePrint::print` is called with `self == sp`. A method `m` of class `A` is represented as "`A::m`";
- in the message send to `self`
  `get()`
  inside `print`, the search for `get` begins at the *runtime* class of `self`, `StorePrint`. Remember `sp` was initiated with "`StorePrint.new()`". Method `StorePrint::get` is called;
- in "`super.get()`" inside `StorePrint::get`, method `Store::get` is called with `self == sp`. This method returns value of `n` of `sp`, `1`, set in "`sp.set(1)`".

Method `get` may be further overridden in subclass `StoreLess` of `StorePrint`, as shown in Figure 1.17. The code

[9]We refer to the syntax of Green and any major object-oriented language.

```
class Car
  public:
    proc turnOn()
      begin
      engine.turnOn();
      end
    // other methods
    ...
  private:
    var engine : Engine;
    ...
    // other instance variables
end
```

Figure 1.18: `Car` has an engine

```
var sl : StoreLess = StoreLess.new();
sl.set(1);
sl.print();
```

calls method `print` of `StorePrint` in its last statement. The search for `print` at *runtime* starts at `StoreLess` but the method is only found in class `StorePrint`. Method `print` sends a message `get` to `self` in

```
    Out.writeln( get() );
```

`self` is `sl`, the original message receiver. Then the search for a `get` method at runtime starts at the class of the object referred by `self`, `StoreLess`. Variable `sl` was initiated with an object of this class. Method `StoreLess::get` is found in the search and called. This example shows that a method of a superclass, `print`, may call a method of a subclass, `StoreLess::get`. Then the behavior of the superclass method is somehow modified by the overridden subclass method. But this only occurs in subclass objects.

Note till now the object referred by a variable has a class equal to the variable class, as can be clearly seen in

```
    var sl : StoreLess = StoreLess.new();
```

That may be different, as will be seen soon.

1.7.3 The Relation Part-Of

Inheritance is very often misused. A common mistake is to take the relation "part of" as "superclass of". For example, a car can be turned on, accelerated, and turned off. The car engine also supports these operations. Then one can make class `Car` inherit from class `Engine`. It may be the case that all operations on an engine are necessary in class `Car`, making the inheritance perfect.

But a car is not an engine — the inheritance is wrong. A car has an engine, an engine is part of a car. Class `Car` should forward the messages "turn on", "accelerate", and "turn off" to its part engine. See Figure 1.18.

1.7.4 Multiple Inheritance

Inheritance expresses the relationship "is-a". A student is a person, a circle is a figure. Sometimes a concept represents more than one entity and more than one "is-a" relationship is needed. For example, a hydroplane is a plane and is a boot. A computer science professor may be both a professor and a programmer. A text window is a text and a window. To express these multiple "is-a" relationships, some languages support *multiple inheritance*: a class can inherit from more than one superclass. Green and Java do not support this mechanism but C++ does. Figure 1.19 shows a fake example in Green of

```
class ClockCalculator subclassOf Clock, Calculator
  public:
    proc reset()
      begin
      super(Clock).reset();
      super(Calculator).reset();
      end
   // other methods
end
```

Figure 1.19: An example of multiple inheritance

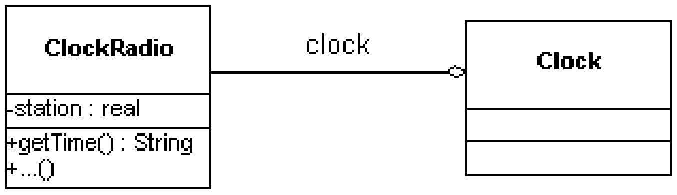

Figure 1.20: Simulation of inheritance of `Clock` by `ClockRadio`

multiple inheritance — the syntax is hypothetical since the language does not support this concept. Class `ClockCalculator` inherits from `Clock` and `Calculator`. Method `reset` calls method `reset` of `Clock` through the syntax "`super(Clock)`" which means "the superclass `Clock`". Of course, `super` alone would be ambiguous since there are two superclasses.

Multiple "is-a" relationships such as the examples described above abound in the real world. But surprisingly they are not that common in object-oriented programming. And when they are necessary, they can be simulated pretty well — see section 1.7.5.

There are several problems with multiple inheritance. One of them is name collision — what to do when a method `set` is inherited from both superclasses ? Which one is inherited ? This problem and others are discussed in Section 1.7.6.

1.7.5 Simulation of Inheritance

Inheritance can be easily *simulated* in any object-oriented language. As an example, we will simulate the inheritance of `Clock` by `ClockRadio`. class `ClockRadio` declares a variable `clock` of class `Clock`. This

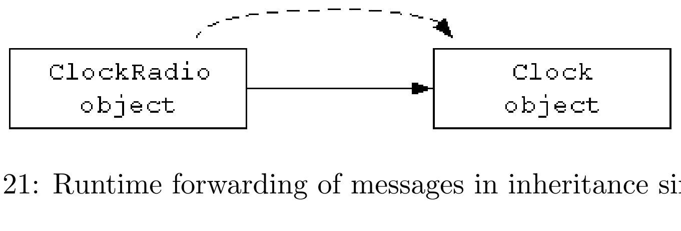

Figure 1.21: Runtime forwarding of messages in inheritance simulation

variable points to a Clock object — see Figure 1.20. Whenever a ClockRadio object receives a message related to Clock, it forwards this message to the object clock. A method of the Clock object is then called. This is shown by Figure 1.21 in which message sends are represented as dashed lines.

See Figure 1.20 for a UML diagram of the simulation of inheritance of Clock by ClockRadio. Figure 1.22 presents the source code of both classes. The constructor of ClockRadio creates a Clock object that is assigned to instance variable clock. Methods that should be inherited from Clock (getTime, getHour, getMin, etc) are defined in ClockRadio and just call the methods of the same name of object clock.

This simulation of inheritance is not perfect. To explain the differences between the inheritance and the simulation of it, we will use the following code:

```
var c : Clock;
var cr : ClockRadio;
cr = ClockRadio.new(12, 20);
  // print the hour
Out.writeln( cr.getHour() );
  // print the time
Out.writeln( cr.getTime() );
c = cr;
...
```

This code is correct if ClockRadio inherits from Clock. But the last assignment is incorrect if we use the classes of Figure 1.22. The classes of cr and c, ClockRadio and Clock, are not related through inheritance and therefore the assignment is not of the kind “Class = subclass”. To make possible this kind of statement, class ClockRadio defines a method getClock. This method returns the object representing the clock in a clock-radio. Then, “c = cr” should be replaced by “c = cr.getClock()”.

Class ClockRadio does not override any method “inherited” from Clock. In this situation, the simulation of inheritance is perfect — the ClockRadio methods behave as if this class inherited from Clock. But is that true when ClockRadio redefines some methods ? Let us study two cases:

(a) ClockRadio overrides method getTime which is not called in Clock in message sends to self.

(b) ClockRadio overrides method getHour which is called in Clock::getTime in message sends to self.

With ClockRadio inheriting from Clock, case **(a)** is shown in Figure 1.23. When simulating inheritance, method getTime of ClockRadio is the same as the method shown in this example. In the code

```
var cr : ClockRadio;
var s  : String;
cr = ClockRadio.new(1, 39);
s = cr.getTime();
```

getTime has the same behavior when using inheritance and when simulating it. When using inheritance, getTime will call getHour and getMin of the superclass Clock. When simulating inheritance, getTime will call getHour and getMin of ClockRadio itself (see Figure 1.22). These methods will call methods of class Clock through variable clock.

With ClockRadio inheriting from Clock, case **(b)** is shown in Figure 1.24.

When simulating inheritance, methods getHour and getTime should be defined as shown in Figure 1.25. In the code

```
var  cr : ClockRadio;
var  s  : String;
var  h  : integer;
cr = ClockRadio.new(13, 43);
h = cr.getHour();
```

```
class Clock
    proc init( hour, min : integer )
      begin
      self.hour = hour;
      self.min = min;
      end
  public:
    proc getTime() : String
      begin
      return self.getHour() + "h " +
             self.getMin()  + " min ";
      end
    proc getHour() : integer
      begin
      return hour;
      end
    ...
end // Clock


class ClockRadio
    proc init( hour, min : integer )
      begin
      clock = Clock.new(hour, min);
      end
  public:
    proc getClock() : Clock
      begin
      return clock;
      end
    proc getTime() : String
      begin
      return clock.getTime();
      end
    proc getHour() : integer
      begin
      return clock.getHour();
      end
    ...
  private:
    var clock : Clock;
        station : real;
end // ClockRadio
```

Figure 1.22: Code that simulates the inheritance of `Clock` by `ClockRadio`

```
class Clock
    proc init( hour, min : integer )
      begin
      self.hour = hour;
      self.min = min;
      end
  public:
    proc getTime() : String
      begin
      return self.getHour() + "h " +
             self.getMin()  + " min ";
      end
    proc getHour() : integer
      begin
      return hour;
      end
    ...
end // Clock


class ClockRadio subclassOf Clock
    proc init(...) ...
  public:
    proc getTime() : String
      begin
      return getHour() + ":" + getMin() + " ";
      end
    // below only radio methods are declared
    ...
end // ClockRadio
```

Figure 1.23: `ClockRadio` overrides `getTime`

```
class ClockRadio subclassOf Clock
    proc init(...) ...
  public:
    proc getHour() : integer
      begin
      return super.getHour()%12;
      end
    // below only radio methods are declared
    ...
end // ClockRadio
```

Figure 1.24: `ClockRadio` overrides `getHour`

```
class ClockRadio
    proc init(...) ...
  public:
      // method getTime not overridden. Just forwards the call
    proc getTime() : String
      begin
      return clock.getTime();
      end
        // method getHour overridden
    proc getHour() : integer
      begin
      return clock.getHour()%12;
      end
    ...
  private:
    var clock : Clock;
        station : real;
end
```

Figure 1.25: Overridden of `getHour` when simulating inheritance

The last statement does not depend on the implementation of `ClockRadio` we use. It will produce the same result whether we use class `ClockRadio` of Figure 1.24 or Figure 1.25. But when using the class of Figure 1.24, in

```
s = cr.getTime();
```

method `Clock::getTime` is called since `cr` points to a `ClockRadio` object and `ClockRadio` inherits from `Clock` which has a `getTime` method. Method `Clock::getTime` sends a message `getHour` to `self` — see Figure 1.23. This calls method `getHour` of the subclass `ClockRadio`, which returns `1`, since `13%12 = 1`.

When simulating inheritance, "`cr.getTime()`" will call method `getTime` of `ClockRadio` — see Figure 1.25. This method calls `getTime` of object `clock`, which is `Clock::getTime`. This method sends message "`getHour`" to `self`, which is the object `clock`. Method `Clock::getHour` is called, since `Clock` is the class of `clock`. This method returns `13`. Method `ClockRadio::getHour` is not called in this case because the `self` reference is not `cr` after "`clock.getTime()`". It is `clock` instead. This loss of the `self` reference makes the simulation wrong or incomplete — it depends on your point of view. A method, `getHour`, was overridden in the pretended subclass, `ClockRadio`, because the inherited method was inadequate. Then, `getTime` of `ClockRadio` should use this overridden method `getHour` instead of `Clock`'s method. But it does not. Then our simulation is not equivalent to language-supported inheritance. Note this problem can be corrected by defining method `getTime` in `ClockRadio` — a less than perfect but useful solution.

Nonetheless, the simulation just presented works in most real cases. It is used to simulate multiple inheritance: if a class `ClockRadioAlarm` should inherit from `Radio`, `Clock`, and `Alarm`, we can make it inherit from `Radio` and simulate the inheritance from `Clock` and `Alarm`. But what if we need to change method `getHour` like in the previous example ? Simply create a class `MyClock` that inherits from `Clock` and overrides `getHour`. In this class, `getTime` will change its behavior because of the overridden of `getHour` — just like in Figure 1.24. Now make `ClockRadioAlarm` inherit from `Radio` and simulate the inheritance from `MyClock` and `Alarm` — see Figure 1.26.

1.7.6 Problems with Inheritance

The interrelationships among super and subclass methods are essential to object-oriented programming. These interrelationships occur because of message sends to `self` and `super`, polymorphism, and overriding of methods in subclasses. A message sent to `self` in a superclass may call a subclass method. Then the

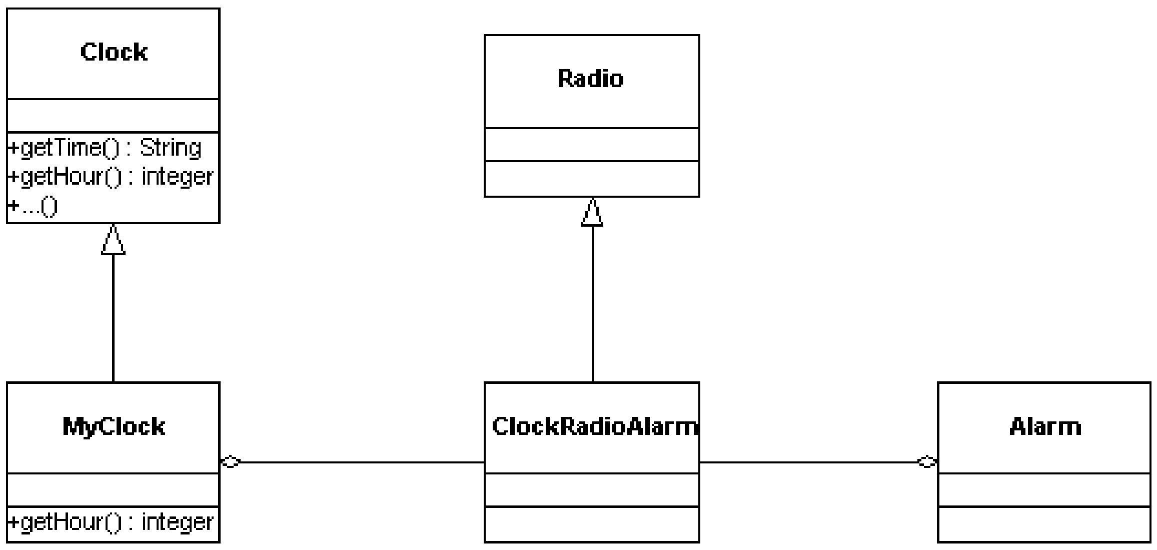

Figure 1.26: References to `self` in object `s`

superclass is correct only if the subclass does what the superclass expects it to do. Polymorphism allows one to supply a subclass object when a superclass object is expected. Then the subclass should be tuned with the superclass. A method overridden in a subclass should obey the semantics of the superclass method. Even though we override a method in a subclass to give it more or different responsibilities than it has in the superclass.

Mismatch between Super and Subclass Methods

A method overridden in a subclass should obey the semantics of the superclass method. The reason is that whenever a superclass object is expected, one can use a subclass object. Then the subclass should have a behavior compatible with that of a superclass. As an example, suppose a method

```
proc weightedAverage( v : RealArray )
     var i : integer;
         s : real;
  begin
  for i = 0 to v.getSize() - 1 do
    s = s + v.get(i)*weight(i);
  return s/v.getSize();
  end
```

calculates the weighted overage of the numbers of an object of class `RealArray`. Method `get` of this class, used in "`v.get(i)`", expects an index `i` between `0` and the number of array elements, "`v.getSize()`", minus `1`. A subclass `FastRealArray` of `RealArray` overrides method `get` to accept a parameter between `1` and the number of array elements. Since a `FastRealArray` is also a `RealArray`, the code below is type correct.

```
var fra : FastRealArray;
```

```
fra = FastRealArray.new(10);
...
wa = stat.weightedAverage(fra);
...
```

But now `weightedAverage` calls "`v.get(0)`" because it follows the semantics of `RealArray`, the declared class of its parameter `v`. A runtime error occurs — index `0` is illegal for fast arrays. Note the comments of method `get` of `RealArray` should specify the valid index range. And the subclasses should follow this range.

This kind of mistake can be more subtle. Suppose method `getTime` of class `Clock` of Figure 1.14 has a comment "returns a String in the format hh:mm". This implies the returned string has at most five characters. We define a class `OfficialClock`, subclass of `Clock`, and redefine method `getTime` to

```
proc getTime() : String
  begin
  return getHour().toString() + "h " + getMin().toString() + "min ";
  end
```

One can show the time got by a `Clock` object in a window, leaving space sufficient for five characters. When an `OfficialClock` object is used, an error occurs.

Other subtle errors may occur with this same example. One can use class `Clock` to create file names of back-up files. The names would use the format

```
     "backup" + clock.getTime()
```

such as "`backup3:45`" or "`backup13:58`". If variable `clock`, maybe a method parameter, points to an `OfficialClock` object, the file names would have spaces such as "`backup3h 45min`". The operating system used may not accept spaces in file names, causing an error when opening the files. Here the error may be in class `OfficialClock` or in the code that creates the backup file names. Comments in `Clock::getTime` may not put any restrictions on the characters of returned strings by the methods, thus allowing spaces. Then the code that creates backup file names is wrong in assuming the strings returned by `getTime` do not contain any spaces.

Class `OfficialClock` is incorrect if `Clock::getTime` explicitly disallows spaces in the returned string. However, in general programmers are not very precise in specifying a method semantics. If there were, programming productivity would drop drastically — the effort in commenting a method would far exceed the effort of codifying it.

The error may be yet more subtle. Suppose a subclass `Clock12` of `Clock` redefines `getHour` to return an hour between `0` and `11` instead of between `0` and `23`. The backup files could have names made using a `Clock12` object. Then the times `2:00` and `14:00` would be the same. If the back-ups follow a 24 hour period, a file created at 14h would override a file created at 2h. That would be an error if the programmer expects to keep every back-up file for at least 24h, as would be the case if she used a `Clock` object.

How Inheritance Breaks Method Abstraction

When a subclass overrides an inherited superclass method, as `getHour`, it should know the circumstances this method is called by other methods of the class, as `getTime`. By knowing that, the subclass programmer can override method `getHour` in order to achieve the desired side effect in `getTime`. But the circumstances `Clock::getTime` send messages `getHour()` to `self` is a private information to `Clock` which is hardwired in the source code of `getTime`. We can only conclude the subclass programmers need to know private superclass information in order to reuse the superclass methods. Then inheritance breaks method abstraction.

Message sends to `self` can cause unexpected and hard-to-discover errors. An example is given by Norvig [25] which uses the `Hashtable` class of JDK. This class has a `put` method to insert pairs (key, value) in a hash table. Objects of any class may be inserted, although in this example the keys will be strings with English words.

Subclass `HashtableWithPlurals` of `Hashtable` overrides method `put` to insert not only a word but also its plural in the table:

```
public Object put( Object key, Object value ) {
  super.put( key + "s", value );
  return super.put( key, value );
}
```

So simple a code, even without an `if` or `while`, should never cause an error. But it does. Suppose an object of `HashtableWithPlurals` receives a `put` message. Method `put` of this class is called which calls the superclass `put` method. This by its turn discovers the table is full. Method `put` of `Hashtable` will allocate a new larger array for the table. Then it will copy the elements from the old to the new array by sending messages `put` to `this` (the `self` of Java). This will call method `put` of `HashtableWithPlurals` which will insert in the table plurals of every word that was in the table, even the 50% of them that were plurals themselves. If `"dog"` and `"dogs"` were in the table, `"dog"`, `"dogs"`, `"dogs"`, and `"dogss"` will be inserted in the table.

1.8 Constructors

In the statement

```
    s = Store.new();
```

an object is created by method `new`. This method belongs to the class object `Store`, which is an object representing the class. This concept will be studied elsewhere. For the moment, just assumes `new` is a method that exists somewhere. Method `new` allocates memory for the object and sends to it a message `init`. Then method `init` of `Store` is called. This method is called the constructor of the class and should be always declared before the public section.

In this example, `init` does nothing. But usually it is used to initialize the object instance variables. It was necessary to declare an empty `init` method because the language Green requires that a class has at least one method `init`. It may have more than one because Green supports method overloading - two methods with the some name are considered different if they have different number/types of parameters.

Then `Store` could have methods

```
    init()
    init( pn : integer)
```

The last constructor would be called when a parameter is passed to `new`:

```
    s = Store.new(0);
```

1.9 Class Objects

In Green, classes are also objects. They can receive messages, as in “`Store.new()`”.Some methods are added to an object that represents a class. In the case of `Store`, method

```
    proc new() : Store
```

was added. Note “`Store`” in the line above means a type: method `new` returns an object of class `Store`. In

```
    s = Store.new();
```

`Store` represents an object. Then outside declarations of methods, such as `new`, or variables, such as `s` and `p` in run of class `Main`, the name `Store` represents an object.

The object `Store` representing the class `Store` is called a class object. It does not have a class. It exists from the beginning to the end of the run time. Although class object `Store` is classless, one may add methods to it. That is easy: just declare it explicitly as in the following example.

```
object Store
  public:
```

```
object Time
   public:
     proc  getCurrentTime() : Time
       begin
       end

class Time
    proc init( phour, pmin, pseg : integer)
      begin
      hour = phour;
      min  = pmin;
      seg  = pseg;
      end
  public:
    proc getHour() : integer
      begin
      return hour;
      end
    ...
end
```

Figure 1.27: Class and class object `Time`

```
    proc getMax ( ): integer
           begin
           return 31;
           end
end
```

Class object `Store` now has methods

```
     proc  new() : Store
     proc getMax() : integer
```

Note that in

```
     s = Store.new();
```

`Store` is the object declared above and that "`Store.new()`" returns an object of class `Store` declared in Figure 1.1

Class objects are used to declare public constants as in

```
object Math
  public:
    const  pi = 3.1415956535;
    proc sqrt( x : real ) : real
    proc sin( x : real ) : real
    ...
end
```

They are used as in

```
     C = 2*Math.pi*radius;
```

Methods `sqrt` and `sin` could be attached to classes `real` and `double`. But that would be a bad practice since a math library would be incorporated into the language.

Figure 1.27 shows another use of class objects: to define methods that are not specific to a single object. Method `getCurrentTime` returns the current time and should not be defined in class `Time`. Class `Time` declares methods an instance of `Time` should have: return the hour, minute, and second, print the time, return the difference between this and another time, etc. An instance of `Time` should be used to keep the time of a doctor's appointment, a movie in the theater, or a TV news. Method `getCurrentTime` has nothing to do with instances of `Time`. Therefore, it was declared in the class object. Note in the declaration of this method,

```
    proc getCurrentTime() : Time
```

the word "`Time`" means *class* `Time`. In

```
    var t: Time;
    t = Time.getCurrentTime();
```

the first "`Time`" is class `Time` and the second "`Time`" is class object `Time`.

1.10 More on Polymorphism

1.10.1 Polymorphism and Software Reuse

Polymorphism is important in software reuse. We do not need to code methods

```
    proc insertCircle( c : Circle )
    proc insertRectangle( r : Rectangle )
    proc insertTriangle( t : Triangle )
```

We can just code method `insert(Figure)` instead. Then the code of this method is reused for each of the subclasses of `Figure`. And more: suppose method `insert(Figure)` belongs to class `FigureList`. This class abstracts a list of figures. Then this method can receive as parameter objects of `Figure` and any of its subclasses that exist now *or will be created later*. This is very important because it means the addition of a new functionality to the program, a subclass of `Figure`, will work with code like `insert(Figure)` that was not intended to work with this new subclass. Method `insert` need not to be changed to accept objects of this new subclass of `Figure`. This works because `insert` demands its parameter be subclass of `Figure` and nothing more.

If polymorphism did not exist, method `insertCircle`, `insertRectangle`, ... should belong to `FigureList`. The addition of a new subclass of `Figure` would demand the creation of a new `insert` method in `FigureList`.

Polymorphism is increased with multiple inheritance. A variable of any the superclasses may point to an object of the subclass. The more superclasses inherited, the more polymorphism.

1.10.2 Method Parametrization by Messages to `self`

This example shows that a inherited, non-overridden superclass method may have a different behavior in a subclass object. Method `getTime` of `Clock` is parameterized by `getHour` and `getMin`.

A method is parameterized by its calls to `self`. By overridden the called methods in subclasses, we change the behavior of the superclass method, tailoring it to the subclass necessities. As a consequence, the superclass method, as `getTime`, is reused in more situations it was originally intended to. Without this parameterization, method `getTime` should also be overridden in `Clock12`, wasting the code of `Clock::getTime`.

Through message sends to `self`, a superclass method may call a subclass method. In this case, the father behavior depends on the child's wishes. This fact is fundamental to object-oriented programming because the superclass method is reused in more situations the programmer foresaw. For example, `getTime` of `Clock` was not coded to work with hours between `0` and `11`. But by redefining `getHour` we adapted it to the subclass semantics. The child vision of the world changed the parent's one.

This strong interaction between the methods of a class hierarchy is fundamental to object-oriented programming because it promotes code reuse. Without it, a language cannot be claimed to be object-oriented.

One could think that to use an object of a class `A` we should know the relationship among the `A` methods. That is, we should know which methods of `A` call other `A` methods through message sends to self. That is not true. To use a class `A` we need only to read the class documentation, which describes what each method does regardless on how it was implemented.

1.10.3 More Examples of Polymorphism

Class `FigureList` declared below represents a list of figures.

```
class FigureList
  public:
    proc insert( f : Figure )
      begin
      if last >= max - 1
      then
        self.error();
      else
        ++last;
        v[last] = f;
      endif
      end
    proc draw()
        var i : integer;
      begin
      for i = 0 to last do
        v[i].draw();
      end
    ...
    // other methods
  private:
    var last : integer;
           v : array(Figure)[];
end
```

An object of `FigureList` can store objects of `Figure` or any of its subclasses. Method `draw` scans the array `v` with the stored objects and sends message "`draw()`" to each of them. Because of the runtime search for a method, each object will call the appropriate `draw` method. That is, a `Circle` object will call `Circle::draw`, a `Triangle` object will call `Triangle::draw`, and so forth.

Class `MenuItem` represents a choice or option of a menu in a graphical user interface. A menu item is associated to an action to be performed when that item is chosen with the mouse or keyboard. Item "Open" of menu "File" of a text editor, for example, would be associated to the action of asking the user for a new file name. Or maybe with the action of presenting her a list of files of the current directory and letting her choose.

Class `MenuItem` is shown in Figure 1.28. When the user chooses a menu item, message `clicked` is sent to an object of `MenuItem`. It then asks object `command` to execute the action. By varying the class of the object `command`, we can vary the action taken.

Every item of a menu, as `New`, `Open`, `Save As`, `Print`, etc, is represented by a `MenuItem` object. The associated actions may all be different because each action depends on which object instance variable `command` refer to.

Polymorphism is used in this example when we allow variable `command` to refer to objects of different classes, resulting then in different actions for each menu item. Variable `command` may refer to objects of class `Command` and its subclasses.

One may change the action associated a menu item at runtime. She just needs to call `setAction` with a new object of `Command` or its subclasses.

```
class MenuItem
  public:
    proc setAction( c : Command )
      begin
      command = c;
      end
    proc clicked()
      begin
      command.execute();
      end
  private:
    var command : Command;
end
```

Figure 1.28: Class representing a menu item

We will now show an example that uses class `MenuItem`. `OpenCommand`, `SaveCommand`, and `SaveAsCommand` are subclasses of `Command`. Therefore, objects of these classes can be passed as parameters to `setAction`, which expects a `Command` object. This use of polymorphism makes it possible to associate different actions to different `MenuItem` objects. And two actions are associated to `mi_save` at different times.

```
var mi_open, mi_save : MenuItem;

mi_open = MenuItem.new( );
mi_open.setAction( OpenCommand.new() );
...
mi_open.clicked();  // call execute of OpenCommand

mi_save = MenuItem.new();
mi_save.setAction( SaveCommand.new() );
mi_save.clicked();  // call execute of SaveCommand

mi_save.setAction( SaveAsCommand.new() );
mi_save.clicked();  // call execute of SaveAsCommand
```

The above example was taken from the essential book "Design Patterns" [10]. It is an instance of the pattern Command.

1.11 Abstract Classes

The common characteristics of two or more classes may be gathered and used to create a superclass which is then inherited by the classes. The superclass created by this mechanism may not represent a real-world entity. Frequently, some of its methods have empty bodies. The methods of the subclasses may be so different that no code is generic enough to be put in the superclass. But we know all subclasses should have such and such methods, and that is why the methods are declared in the superclass.

This kind of superclass is called abstract and uses a special syntax in most languages. In Green, an abstract class is declared as shown in Figure 1.29. Keyword `abstract` is used before the declaration of the class and before each abstract method. An abstract method may only be declared in an abstract class and does not have a body. Regular methods may be defined in an abstract class as shown in the example.

```
abstract class Container
  public:
    abstract proc add( x : integer )

    abstract proc get() : integer

    proc empty() : boolean
      begin
      return getSize() == 0;
      end

    proc full() : boolean
      begin
      return getSize() == getMaxSize();
      end

    abstract proc getSize() : integer

    abstract proc getMaxSize() : integer;
end
```

Figure 1.29: Class `Container` for storing objects

Class `Container` of Figure 1.29 was based on the class with the same name of the Standard Green Library. Class `Container`[10] is the superclass of several data structure classes such as `List`, `Stack`, `Queue`, etc. These subclasses of `Container` should all have a method to add an element to the structure. That is why `Container` defines a method `add`. These subclasses use very different implementations for `add` which have no common code. That is why `Container add` method is abstract.

Since an abstract class is incomplete, no object of it can be created. But constructors (`init` methods) may be declared for they can be called in subclasses. Instance variables are also allowed.

An abstract class can be the type of a variable or parameter. This variable/parameter will refer at runtime to objects of subclasses of the abstract class. Abstract classes are designed to be inherited. Whenever possible, a class should inherit from an abstract class instead of a regular class. But ... why ? Does not a regular class offer more opportunities for code reuse since it supplies the code of all of its methods ? Why should a class inherit from an incomplete class ?

An abstract class is *abstract* which means:

- objects of it should not be created and it does not represent any real world entity. That means it will change far less than a regular class over time. By changing much less, maintenance of subclasses become much easier;

- some of its methods may not have bodies. Usually a lot of its methods do not have bodies. That means the relationships among the class methods brought by message sends to `self` (see Section 1.6, page 27) do not exist. Method overridden in subclasses will cause much less errors than in regular classes.

Some people believe inheritance should only be made from abstract classes, a rule enforced by the first object-oriented language, Simula-67.

[10]In fact, class `Container` of the Standard Green Library is different from the class presented here — it is an abstract parameterized class and has much more methods. See section 1.12 for the definition of parameterized classes.

```
class Stack(T)
  public:
    proc push( x : T )
      ...
    proc pop() : T
      ...
    ...  // other methods
  private:
    var v : array(T)[];
        top : integer;
end
```

Figure 1.30: A parameterized class `Stack`

1.12 Parameterized Classes

Suppose you need to code a stack class for integers. This class would look like this:

```
class Stack
  // no error handling is attempted in this class. If the stack is
  // full, the element is just not returned
  public:
    proc push( x : integer )
      begin
      if top < max - 1
      then
        ++top;
        v[top] = x;
      endif
      end
    proc pop() : integer
      begin
      var x : integer;
      x = v[top];
      --top;
      return x;
      end
    ...
  private:
    var v : array(integer)[];
        top : integer;
end
```

It may be necessary to create a stack for real numbers which differs only slightly from the stack above. In some places, the word "`integer`" should be replaced by "`real`", the type for real numbers. This is usually achieved by copying and pasting the stack class followed by a search and replace operation in a text editor. This solution duplicates the source code of stack making the maintenance harder: if the stack for integers should be changed, so should the stack for reals, thus duplicating the maintenance effort.

Some object-oriented languages offer a construct called parameterized or generic class that allows one to define a generic stack class. A parameterized class has one or more *type* parameters which may be used in the class code as any other type. A parameterized stack class is shown in Figure 1.30. Parameter `T` should be supplied when `Stack` is used:

```
var s : Stack(char) = Stack(char).new();
s.push('O');
s.push('K');
```

Note `Stack` is not really a class — it is only a skeleton for building classes. `Stack(char)`, `Stack(String)`, and `Stack(Circle)` are real classes created at compile time. The compiler duplicates the source code of `Stack(T)` and replaces `T` by the real parameter. Then the new stack class is compiled and syntactic and semantic analyses are made by the compiler. The replacement of the formal by the real parameters of a parameterized class is called *instantiation*.

Suppose class `Stack(T)` defines a method

```
proc print()
     var i : integer;
  begin
  for i = 0 to top - 1 do
    v[i].print();
  end
```

The type of "`v[i]`" is `T`, since the type of `v` is "`array(T)[]`". The call "`v[i].print()`" then implies type `T` has a `print` method. Now a declaration

```
     var s : Stack(char);
```

will cause a compile-time error since `char` has no `print` method. The type checking for the `Stack` class is made at each instantiation, not just once.

Some compilers of some languages do not duplicate the code of the parameterized class at each instantiation. They try to share code among all instantiations of a parameterized class whenever possible. Sometimes a method code should be duplicated because no compiled code is general enough to be used by all instantiations of the class.

The Green language allows one to specify a class for a type of a parameterized class:

```
class Stack( T : Printable )
  public:
    ...
end
```

Assume `Printable` is a class with a single method, "`proc print()`". Class `Stack` can only be instantiated with classes that define a `print` method. The real parameter should be subtype[11] of the formal parameter. Using the information "`T` is a subtype of `Printable`", the compiler can check if variables of `T` are being used correctly inside `Stack`. The declaration `Stack(char)` would be considered incorrect by the compiler because `char` is not `subtype` of `Printable`.

1.13 Subclass and Protected Sections

We have seen a class has `public` and `private` sections. There is a third section, `subclass`, whose methods and variables are visible only inside the class and its subclasses. An example is shown in Figure 1.31 and used in class `StackPrint` of Figure 1.32. Methods `get` and `set` of `Stack` are visible in `StackPrint` only in message sends to `self`. That is, message sends

```
     self.set(i, Person.new("Churchil"));
     set(i, Person.new("Garrincha"));
```

are legal inside `StackPrint` but

```
     var s : Stack = Stack.new();
     s.set(0, Person.new("Zuze"));
```

are not. Variable `s` could refer to an object of a subtype of `Stack` which does not define a method `set`

[11]Assume for a while that subtype is subclass.

```
class Stack
  public:
    proc push( x : Person )
      ...
    proc pop() : Person
      ...
  subclass:
    proc get(i : integer) : Person
      begin
      return v[i];
      end
    proc set(i : integer;
             value : Person)
      begin
      v[i] = value;
      end
    proc getSize() : integer
      begin
      return top + 1;
      end
  private:
    var v : array(Person)[];
        top : integer;
end
```

Figure 1.31: Class with `subclass` section

```
class StackPrint subclassOf Stack
  public:
    proc print()
        var i : integer;
      begin
      for i = 0 to getSize() - 1 do
        get(i).print();
      end
end
```

Figure 1.32: Use of the subclass section for efficiency

in its `subclass` section. Of course, this restriction also applies to parameters, making the method below illegal, be it added to `Stack` or `StackPrint`.

```
proc concat( other : Stack )
      // concat stack other to the top of stack self
    var i : integer;
  begin
  for i = 0 to other.getSize() - 1 do
    push(other.get(i));
  end
```

The compiler would point errors at "`other.getSize()`" and "`other.get(i)`".

The `subclass` section is used for

- putting low level methods which are methods that access the instance variables directly. When used carelessly, these methods will damage the object data, since they do not check the legality of its use. As an example, `get` and `set` are low level methods of `Stack`. Neither checks if parameter `i` is in a valid range. And method `set` allows a code to change any value in the stack, not only the top. This breaks the semantics of a stack. If not used with care, `set` will produce unexpected results in stack clients;
- putting methods which are not operations of the class. In class `Stack`, clearly `push` and `pop` are valid methods. But `get` and `set` are not operations on stacks and therefore should not be in the public section;
- increasing the performance of the program. This can be clearly seen in method `print` of `StackPrint`. Without method `get`, the stack should be emptied, stacked into another stack, and then printed:

  ```
  proc print()
      var s : Stack;
          p : Person;
    begin
    s = Store.new();
    p = pop();
    while p <> nil do
      begin
      s.push(p);
      p = pop();
      end
    p = s.pop();
    while p <> nil do
      begin
      p.print();
      p = s.pop();
      end
    end
  ```

1.14 Casting Subclass/Class

A variable declared as having type `Person` may refer to an object of a subclass `Employee` of `Person`. Through this variable, one cannot use the methods declared in class `Employee` of the object:

```
var person : Person;
var salary : real;
```

```
...
person = Employee.new("Peter", "Manager", 60000,00);
salary = person.getSalary();  // compile-time error !!!
```

An `Employee` variable should point to the object and method `getSalary` should be called through this variable:

```
var person : Person;
var salary : real;
var employee : Employee;
...
person = Employee.new("Peter", "Manager", 60000,00);
employee = person;
salary = employee.getSalary(); // ok
```

However, the compiler will not accept "`employee = person`" because in general it cannot be sure `person` will refer, at runtime, to an object of class `Employee`, the declared class of variable `employee`. Variable `person` may refer to objects of `Person` and its subclasses. In particular, `person` may refer to an object of class `Student`, subclass of `Person`, which is not related to `Employee`. In this case the assignment "`employee = person`" would make a variable of `Employee` point to a `Student` object. If this were allowed, the message send "`employee.getSalary()`" should cause a runtime type error, since `Student` objects do not have a `getSalary` method. But Green is a statically-typed language, which means all type errors are discovered at compile time. Therefore the assignment "`employee = person`" is illegal.

The assignment "`employee = person`" should be replaced by

```
    employee = Employee.cast(person);
```

`cast` is a method of the class object `Employee`. It casts its parameter to an `Employee` object. This method is declared as

```
    proc cast( x : Any ) : Employee
```

`Any` is the superclass of every other class. If the object cannot be cast to `Employee`, exception `TypeErrorException` is thrown.

This method `cast` is automatically added by the compiler to class object `Employee`. In fact, to every class object the compiler adds a similar method.

Note method `cast` just checks if the class of object `person` at runtime is `Employee` or its subclasses. Method `cast` does not change the pointer `person` before assigning it to `employee`. Both variables will refer to the same object after `cast` is called, although they have different types.

Casting objects from subclass to superclass should be made with care. In general, it means your code is badly designed, although sometimes the casting is inevitable.

Bibliography

[1] Agha, Gul. An Overview of Actor Languages. *SIGPLAN Notices*, v. 21, n. 10, p. 58–67, October 1986.

[2] America, Pierre and Linden, Frank van der. A Parallel Object-Oriented Language with Inheritance and Subtyping. *SIGPLAN Notices*, Vol. 25, No. 10, October 1990, OOPSLA 1990.

[3] Blake, Edwin and Cook, Steve. On Including Part Hierarchies in Object-0riented Languages, with an Implementation in Smalltalk. Proceedings of ECOOP 87. Lecture Notes in Computer Science No. 276.

[4] Borning A. and O'Shea, T. DeltaTalk: An Empirically and Aesthetical Motivated Simplification of the Smalltalk-80 Language. Proceedings of ECOOP 88. Lecture Notes in Computer Science No. 322.

[5] Chiba, Shigeru and Masuda, Takashi. Designing an Extensible Distributed Language with a Meta-Level Architecture. Proceeding of ECOOP'93. Lecture Notes in Computer Science No. 707, 1993.

[6] Chiba, S. Open C++ Programmer's Guide. Technical Report 93-3, Department of Information Science, University of Tokyo, Tokyo, Japan, 1993.

[7] Chiba, S. A Metaobject Protocol for C++. *SIGPLAN Notices*, Vol. 30, No. 10, October 1995, pg. 285-299, OOPSLA'95.

[8] GC FAQ — draft. Available at http://www.centerline.com/people/chase/GC/GC-faq.html

[9] Coplien, James O. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[10] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns.* s.l., Addison-Wesley, 1995.

[11] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[12] Goldberg, Adele and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[13] Guimar aes, Jos'e de Oliveira. Filtros para Objetos. Tese de Doutorado, 1996.

[14] Guimar aes, Jos'e de Oliveira. Reflection for Statically Typed Languages. *Lecture Notes in Computer Science* Vol. 1445. *ECOOP 98.*

[15] Guimar aes, Jos'e de Oliveira. The Green Language. http://www.dc.ufscar.br/ ~jose/green/green.htm.

[16] Guimar aes, Jos'e de Oliveira. The Green Language Type System. http://www.dc.ufscar.br/ ~jose/green/green.htm.

[17] Foote, Brian and Johnson, Ralph. Reflective Facilities in Smalltalk-80. *SIGPLAN Notice*, Vol. 24, No. 10, October 1989. OOPSLA 89.

[18] Ibrahim, Mamdouh and Bejcek, W. and Cummins, F. Instance Specialization without Delegation. *Journal of Object-Oriented Programming*, June 1991.

[19] Ibrahim, Mamdouh. Reflection in Object-Oriented Programming. *International Journal on Artificial Intelligence Tools*, Vol. 1, No. 1, pg. 117-136, (1992).

[20] Johnson, Ralph and Foote, Brian. Designing Reusable Classes. *Journal of Object-Oriented Programming*, v. 1, n. 2, p. 22–35, 1988.

[21] Lippman, Stanley B. *C++ Primer*. Addison-Wesley, 1991.

[22] Maes, Pattie. Concepts and Experiments in Computational Reflection. *SIGPLAN Notice*, Vol. 22, No. 12, December 1987. OOPSLA 87.

[23] Meyer, Bertrand. Tools for the New Culture: Lessons from the Design of the Eiffel Libraries. *Communications of the ACM*, v. 33, n. 9, September 1990.

[24] Murata, Makoto and Kusumoto, Koji. Daemon: Another Way of Invoking Methods. *Journal of Object-Oriented Programming*, July/August 1989.

[25] Norvig. Java Infrequently Asked Questions. Available at `http://www.norvig.com`.

[26] O'Shea, Tim. Panel: The Learnability of Object-Oriented Programming Systems. *SIGPLAN Notices*, Vol. 21, No. 11, November 1986, OOPSLA 86.

[27] Perry, Dewayne and Kaiser, Gail. Adequate Testing and Object-Oriented Programming. *Journal of Object-Oriented Programming*. January/February 1990.

[28] Stroustrup, Bjarne. *The C++ Programming Language*, Second Edition, Addison-Wesley, 1991.

[29] Taenzer, David; Ganti, Murthy; Podar, Sunil. Object-Oriented Software Reuse: The Yoyo Problem. *Journal of Object-Oriented Programming*, p. 30–35, September/October 1989.

[30] A Taligent White Paper. Leveraging Object-Oriented Frameworks. 1996.

[31] Wegner, Peter. The Object-Oriented Classification Paradigm. In: Shriver, Bruce; Wegner, Peter, eds. *Research Directions in Object-Oriented Programming*. s.l., MIT Press, 1987. p. 479–559.