# The Green Language

José de Oliveira Guimarães

*Departamento de Computação, UFSCar, São Carlos - SP 13565-905 Brazil*

**Abstract**

Green is a statically-typed object-oriented language which supports parameterized classes, metaobjects, introspective reflection, and classes as first-class objects. Its exception system is completely object-oriented for it encapsulates in classes not only exceptions but also exception handling. The language definition of subtyping is more encompassing than subclassing, thus improving polymorphism. Classes are classless objects which have themselves types. This makes classes first-class objects without the problems associated to languages in which every class is an object of another class, its metaclass. Every basic value such as `7` or `'A'` is considered as an object whenever necessary which makes programming easy and increases polymorphism.

*Key words:* object-oriented languages, Green, type systems, polymorphism, exception handling system

## 1 Introduction

Green is a statically-typed object-oriented language which supports parameterized classes, classes as first-class objects, runtime metaobjects, introspective reflection, and garbage collection. This language offers a completely object-oriented exception system, assertions, and methods with variable number of parameters. Green considers a class S subtype of class T if S defines at least all methods of T — it does not matter whether S inherits from T or not. Multiple inheritance is not supported.

Green uses a Pascal-like syntax with some C elements, as can be seen in Figure 1. This Figure declares a class `Date` which keeps the day, month, and year. The constructor of the class is always called `init` and appears before the `public` section. There may be several of them, with variable number/types

*Email addresses:* jose@dc.ufscar.br (José de Oliveira Guimarães), Tel.: 55 16 260-8607 Fax: 55 16 260-8233 (José de Oliveira Guimarães).

```
class Date
     proc init(month, day, year : integer)
        begin
        self.month = month;
        self.day = day;
        self.year = year;
        end
  public:
     proc getDay() : integer
        begin
        return day;
        end
     proc getMonth() : integer ...
     proc getYear() : integer ...
     proc set(month, day, year : integer) ...
  private:
     var day, month, year : integer;
end
```

Fig. 1. Class `Date`

of parameters. Methods are declared with keyword `proc` and variable declarations are preceded by `var`. An object is created by sending message `new` to the class:

```
     var xmas : Date;
     xmas = Date.new(12, 25, 2003);
       // print the year  -- this is a comment
     Out.writeln( xmas.getYear() );
```

As will be seen later, class `Date` is an object and therefore it can receive messages as `new`.

All variables whose types are classes, like `xmas`, are in fact pointers to objects. They obey reference semantics. If the type of a variable is a basic type (`char`, `integer`, `byte`, `real`, ...), the variable obeys value semantics — it holds a value, it does not point to a value (object). A variable may be declared with type `@Date` to mean it obeys value semantics — it would be an *expanded* variable. This concept has several peculiarities which will not be discussed in this paper. It was based on a similar concept of Eiffel [15].

The rest of the paper discusses the features of Green. Its type system is presented in Section 2. The exception system is described in Section 3. Section 4 introduces the reflective features of Green. The constructs that do not fit in any of the previous sections are presented in Section 5. Section 6 concludes.

## 2 The Type System

In Green, the type of a class is the set of signatures of its public methods. The signature of a method is composed by its name, parameter types, and return value type (if any exists). For example, the type of class `Date` of Figure 1 is

```
{ getDay() : integer, getMonth() : integer,
  getYear() : integer, set(integer, integer, integer), ... }
```

in which `{` and `}` delimit the set as usual. In this set the last item, "`...`", represents some methods the compiler adds to every class — that will be discussed later.

The `init` method, the constructor, is not in the public part and it does not belong to the type of the class. The public section of a class cannot declare instance variables.

The type of a class is usually taken by the class itself. So, we say "type `Date`" meaning "the type of class `Date`".

A type `S` is subtype of type `T` if $\texttt{T} \subset \texttt{S}$. That is, a subtype has at least all the method signatures of the supertype and usually more. A method of a supertype should be defined at the subtype with exactly the same signature — contra-variant and covariant rules [13] are not used.

The language supports abstract classes which may declare abstract methods (bodiless methods). Objects of abstract classes cannot be created.

A subclass inherits all methods of the superclass. Then the subclass has the methods it defines plus the superclass methods. This means the subclass is subtype of the type of the superclass. Subclassing means subtyping — all subclasses are subtypes. And there may be subtypes that are not subclasses — the definition of subtype does not use the word "inheritance". Therefore subtyping is a much more embracing concept than subclassing. There are more subtypes than subclasses, which increases polymorphism in the language.

The Green definition of subtyping has been used by POOL-I [1], School [18], and Emerald [17]. This last language does not support inheritance.

A class may be subtype of several classes although it may inherit from just one class. Multiple subtyping may act as a substitute for multiple inheritance.

Note that is not necessary to declare "class `S` is subtype of class `T`". A class `S` will be considered subtype of `T` whenever `S` contains at least all `T` method

```
object Date
   public:
        // return the last day of the month
      proc getMaxDay(month, year : integer) : integer ...
        // get today
      proc getCurrentDate() : Date ...
      const NumMonths : integer = 12;
end
```

Fig. 2. Class object `Date`

signatures.

### *2.1 Class Objects*

Figure 2 declares an object that is *the* class `Date`. Classes are objects in Green, although objects that do not have themselves classes. The declaration of object `Date` starts with keyword `object` and may contain methods and variables as the declaration of a class. In addition to that, constants may be declared.

The code of Figure 2 should appear before that of Figure 1 in a file called "`Date.g`".

An object like `Date` of Figure 2 is called *class object*, an object that represents a class. If the programmer did not supply class object `Date`, the compiler would create one automatically. If only the class object is supplied, the compiler creates a class `Date` without any methods except the ones added to every class by the compiler.

A class object is much like an object of a prototype or delegation-based language — it exists from the start to the end of the program execution. And a class object `B` is completely unrelated to class object `A` even if class `B` inherits from class `A`.

As objects, class objects can receive messages:

```
today = Date.getCurrentDate();
```

Constant `NumMonths` may be accessed similarly:

```
n = Date.NumMonths;
```

Class object methods like `getCurrentDate` are similar to static methods of C++ and Java. These methods are the equivalent of class methods of Smalltalk [6]. Class methods are the methods of the class of the class (considered as an object). The class of the class is called a metaclass. Then `getCurrentDate` works as a method of the metaclass of class `Date`. The class object `Date` plays the rôle of metaclass of class `Date`. Class objects have some of the responsibilities of metaclasses:

- they may provide variables and methods shared by all instances of the class;
- they supply `new` methods to create class instances.

Since *class* `Date` (Figure 1) defines a constructor `init(integer, integer, integer)`, the compiler adds to *class object* `Date` (Figure 2) a method `new(integer, integer, integer)`:

```
var xmas : Date = Date.new(12, 25, 2003);
```

This is very important: the compiler adds a `new` method to the *class object* `Date` so that an object of *class* `Date` is created by a method call. Then object creation in Green is made through message send and not through a special operator as in Java/C++. Eiffel [15] uses an operator with an optional message send.

Class objects have a few of metaclass responsibilities. However, metaclasses [2] [5] [6] [14] have much more duties and are more complex than class objects. The complexities of metaclasses are discussed by Guimarães [12]. One of the difficulties with metaclasses is that if each class is an object of its metaclass, then there is an infinite chain of metaclasses. Class `A` is an object of `Metaclass-A` which is an object of `Meta-Metaclass-A`, and so on.

The class object concept eliminates this infinite regression by stating that class objects do not have classes. However, class objects do have types, which can be got by the compile-time function `type`:

```
var d : type(Date);
d = Date;
   // print the day of today
Out.writeln( d.getCurrentDate().getDay() );
```

By allowing to type classless class objects, the Green type system integrates classes, considered as objects, into the type system. We are not aware of any other statically-typed language with similar or equivalent feature.

Arrays are classes in Green. Since each class has an associated class object, each array class has a class object. The class name of a one-dimensional integer array is "`array(integer)[]`". The class object has this same name and can receive messages such as `new` for object creation. See the example below.

```
var ia : array(integer)[];
  // allocates an array with 100 positions
ia = array(integer)[].new(100);
  // all elements are set to 0
ia.fill(0);
```

```
Any
   AnyClass
      Date
      AnyArray
         array(char)[]
         AnyClassArray
            array(Date)[]
   AnyClassObject
      * class object Date
```

Fig. 3. Class and type hierarchies of Green

Since arrays are classes, array objects are dynamically allocated. The number of elements of each array dimension is specified at runtime in the parameter to `new`.

### *2.2 The Subtype/Subclass Hierarchies and Polymorphism in Green*

Any non-basic class that does not explicitly inherits from another class is made a subclass of `AnyClass`. This class inherits from class `Any`, the top-level class.

All arrays of basic types (`char`, `integer`, ...) are classes and inherit from class `AnyArray`, which inherits from `AnyClass`. The Green class and type hierarchies are shown in Figure 3. In this Figure, if class `B` inherits from class `A`, `B` is put in the line below `A` and to the right of it. We used classes `char` and `Date` as representatives of basic classes and regular classes.

Class "`array(char)[]`" is an array of `char` and inherits from `AnyArray`. The `*` in the last line means "subtype". So class object `Date` is subtype of `AnyClassObject` – this will soon be explained.

Class `Any` defines some generic methods such as

```
shallowClone() : Any
equals(other : Any) : boolean
```

Class `AnyClass` defines some methods, among them,

```
getClassObject() : AnyClassObject
```

which returns the class object of the object. Then, if `d` is set as

```
var d : Date = Date.new(12, 25, 2003);
```

`d.getClassObject()` returns class object `Date`.

Objects in Green have types. The type of an object is the set of its public method signatures. Therefore a class object has a type and may be subtype of a class. In fact, every class object is a *subtype* of class `AnyClassObject`, even though class objects are *classless* objects. `AnyClassObject` inherits from `Any` and defines some methods specific to class objects. To illustrate the concept

of "objects with types", let us show an example:

```
var co  : AnyClassObject = Date;
var any : Any = Date;
```

The assignments above are legal. They obey the rule that says that assignments of the kind "type = subtype" are legal. Note that the "`Date`" that appears in the code above is the *class object* `Date`. In

```
var d : Date;
```

the "`Date`" is the *class* `Date`.

Since every class object is subtype of `AnyClassObject`, which inherits from `Any`, every class object defines all methods declared in these two classes. These methods should be declared and not inherited, for class objects are classless objects.

The programmer need not to declare all `Any-AnyClassObject` methods in every class object. The compiler does that automatically. This mechanism integrates classless class objects into the type system.

Basic values such as `'A'`, `5`, and `3.14` are *special* objects for variables of basic types obey value semantics — the variables contain the values, they do not point to them. However, they may be packed into objects of wrapper classes in order to become normal objects:

```
var I : Integer;     // wrapper class
I = Integer.new(0);  // packs 0
```

Such mechanism can be provided by any language. The novelty in Green is that there is automatic conversion between wrapper objects and basic values:

```
I = 1;   // cast 1 to wrapper object
var i : integer;
i = 5*I + i/I;
```

Then every basic value may be considered as an object if the programmer wishes so. In particular, a variable of type `Any` may receive anything in an assignment or parameter passing: an object, a class object, or a basic value of any type (`char`, `integer`, `real`, etc). For short, we can program in Green as if everything were an object.

Basic values are special objects but are objects anyway. The number `5` for example is an object of class `integer` which defines methods `toString() : String` and `get(i : byte) : byte` (returns the $i^{th}$ byte of the integer). Then the following code is legal.

```
var s : String;
   // s = "5"
s = 5.toString();
   // prints "integer"
Out.writeln( 5.getClassInfo().getName() );
```

```
class CatchDateException
  public:
     proc throw( exc : ZeroYearException )
        begin
        Out.writeln("Year 0 does not exist !");
        end
     proc throw( exc : IllegalMonthException )
        begin
        Out.writeln("Month should be between 1 and 12. Found " +
                        exc.getMonth() );
        end
     proc throw( exc : IllegalDayException )
        begin
        ...
        end
end
```

Fig. 4. Catch class `CatchDateException`

Every array inherits directly or indirectly from `AnyArray` which defines some general methods such as "`getSize() : integer`", "`set(value : Any; index : integer)`", and "`get(index : integer) : Any`". Arrays of basic classes define some interesting methods as "`forEach( f : Function(T) )`" in which `T` is the basic class. `Function` is an abstract parameterized class which takes a parameter `T` — parameterized classes are explained in Section 5. Class `array(integer)[]` then defines "`forEach( f : Function(integer) )`". This method scans the array calling method `exec` (defined in class `Function`) on each array element. Of course, the programmer should subclass the abstract class `Function` and define `exec`. There are other methods in the arrays of basic classes such as "`collect(f : Filter(T)) : array(T)[]`" and "`remove(f : Filter(T))`". Method `collect` returns another array with the elements `x` of the array such that "`f.test(x)`" is `true`. Method `remove` removes from the array all elements `x` that make `test` evaluate to `true`.

Arrays of non-basic classes such as `array(Date)[]` are subclasses of `AnyClassArray` — see Figure 3. `AnyClassArray` defines methods similar to the basic-class arrays but with class `Any` replacing `T`. For example, it defines "`forEach(f : Function(Any))`".

For sake of efficiency, array classes cannot be subclassed.

## 3 The Exception System

The exception system of Green is completely object-oriented. It encapsulates exceptions in objects as usual in object-oriented languages. But it also encapsulates exception handling in objects. To understand that, imagine a try command of C++/Java. It is followed by several `catch` clauses. In Green, these `catch` clauses are transformed into methods of a *catch class*. An object of this class is passed to the `try` statement as parameter and it becomes responsible for exception handling. Let us see an example:

```
catchDate = CatchDateException.new();
try(catchDate)
   // start of the try block
   ...
   if year == 0
   then
        // throw or signal an exception
      exception.throw( ZeroYearException.new() );
   endif
   if month < 1 or month > Date.NumMonths
   then
      exception.throw( IllegalMonthException.new(month) );
   endif
   ...
end // end of the try block
```

`CatchDateException`, shown in Figure 4, is a catch class. It has a `throw` method for each exception the programmer wants to catch in the try block — the block between `try` and `end` (see example above). The parameter of the `throw` method must be an exception class, which should inherit from class `Exception`. An object of `CatchDateException` is passed to `try` in the line "`try(catchDate)`".

An exception is thrown (or signalled) by calling method `throw` of pseudo-object `exception`. Then the corresponding method `throw` of the catch object `catchDate` is called. After that, the control is transferred to the first statement following the try block. Everything happens as if `exception` referred to the same object as `catchDate`.

Everything becomes clear when we study the exception signaling

```
exception.throw( IllegalMonthException.new(month) );
```

When this statement is executed, a search for a method `throw` that may accept a `IllegalMonthException` is made at the class of the `catchDate` object, `CatchDateException` (Figure 4). This search is made from the first to the

last declared method. The method found is executed. If none were found, the exception would be propagated — the same action is taken in C++/Java when no *catch clause* can accept the exception thrown in the attached `try` block.

`try` statements may be nested:

```
try(catchTwo)
   try(catchOne)
      ...
      exception.throw(anException);
      ...
   end
end
```

When the exception `anException` is thrown, a search in the class of object `catchOne` is made for a `throw` method that can accept `anException` as parameter. If this search fails, a new search is made in the class of `catchTwo`.

The parameters to `try` statements are pushed into a stack called "*stack of catch objects*". The search for a `throw` method is made from top to bottom in this stack. The program can inspect this stack at runtime — see Section 5.

Methods may declare the exceptions they may throw using the following syntax:

```
proc setBirthday(day, month, year : integer)
                 (exception : CatchDateException)
   begin ... end
```

This means `setBirthday` may throw `ZeroYearException`, `IllegalMonthException`, and `IllegalDayException`, the parameter types of the `throw` methods of `CatchDateException`.

A method should either catch the exceptions it may throw or declare them in its interface. In this way, no user exception may be thrown and not caught by the program.

An unchecked exception is a special exception that need not to be caught by the user code. Unchecked exceptions may only be thrown by the runtime system, never by the user code. `OutOfMemoryException` is an example of unchecked exception.

Green has default handlers for all unchecked exceptions. These handlers are `throw` methods of class object `HCatchUncheckedException`. The default handlers can be changed at runtime by supplying a new catch object to the runtime system:

```
Runtime.setCatchUnchecked( MyDefaultHandlers.new() );
```

Now handlers of `MyDefaultHandlers` will have precedence over the handlers

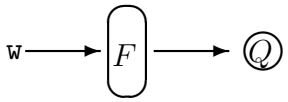


Fig. 5. A shell $F$ attached to an object $Q$

of class object `HCatchUncheckedException`.

Class objects `CatchAll` and `HCatchAll` of the Green Standard Library catch every exception. Besides that, `HCatchAll` also terminates the program. Then we can use "`try(CatchAll) ... end`" where we would use "`try { ... } catch(Exception e) {}`" in C++/Java.

## 4 Reflection

A program written in a reflective language may change and/or inspect itself at runtime. A language that supports *behavioral* reflection allows the program to *change* some aspects of itself at runtime. The program may load new classes, intercept message sends (and redirect them to other objects), replace methods of a class, and so on.

A language that supports *introspective* reflection allows a program to get information (inspect) on itself at runtime. The program knows the names of all of its classes, the methods of each class, the name and types of each method parameter, the methods of the stack of called methods, and so on.

### *4.1 Behavioral Reflection*

Green supports both behavioral and introspective reflection. Behavioral reflection in Green only allows one to intercept messages sent to an object. It is implemented by metaobjects, which are called *shells* in Green.

A *shell* is a pseudo-object with methods and instance variables. It may be attached to an object as graphically represented in Figure 5. After that, any message sent to object $Q$ through any variable will be first searched in shell $F$ and then in the object.

A shell class `Border` is defined in Figure 6. A shell `Border` is an object of shell class `Border`. It may be attached to objects of `Window` and its subtypes. Assume class `Window` has a parameterless method `draw`. Shell class `Border` can only define methods already defined in `Window` — `Border` should be supertype

```
shell class Border(Window)
  proc init()
    begin
    end
  public:
    proc draw()
      begin
      self.drawBorder();
        // call object method
      super.draw();
      end
    private:
      proc drawBorder()
        ...
end
```

Fig. 6. A shell class `Border` to subtype-`Window` objects

of `Window`.

Suppose `w` is a `Window` object. A `Border` shell is attached to this object by the statement

```
Meta.attachShell(w, Border.new());
```

This results in the configuration shown in Figure 5 in which $F$ is the shell of shell class `Border` and $Q$ is the `Window` object. Now, a call "`w.draw()`" will invoke method `draw` of `Border` — the method is first searched for in the shell and then in the object (see Figure 5). In the shell, method `draw` of shell class `Border` is found and called — see Figure 6. In this method, "`super.draw()`" will invoke method `draw` of `Window` — shell $F$ of Figure 5 calls `draw` of object $Q$.

A method

```
proc interceptAll( mi : ObjectMethodInfo;
                   vetArg : array(Any)[] )
```

may be declared in the shell class. This method will then intercept all messages sent to the object the shell is attached to. The message will be packed into parameters `mi` and `vetArg`. The first one describes the method of the object that would be called if there were no shell. Parameter `vetArg` contains the parameters of the call. As an example, suppose shell class `Border` defines an `interceptAll` method (and only it) and variable `w` refers to a `Window` object with an attached `Border` shell (as shown in Figure 5). In a message send "`w.draw()`", method `interceptAll` of shell $F$ would be called. Parameter `mi` would describe method `draw` of `Window`. Parameter `vetArg` would contain zero elements since there is no argument to `draw`. Using `mi`, the $Q$ method `draw` can be called by "`mi.invoke(vetArg)`" inside the method `interceptAll` of object $F$ of shell class `Border`.

Shells may be attached to class objects, which are also objects. A shell may control object creation. For example, suppose a shell class `WindowControl` defines a method `new()` and class `Window` defines a constructor `init()`. Then class object `Window` will have a `new()` method that creates a new `Window` object. After a `WindowControl` shell is attached to class object `Window`, the call "`Window.new()`" will invoke method `new` of the shell.

### *4.2 Introspective Reflection*

Green offers an almost complete support for introspective reflection. There are 31 classes in an Introspective Reflection Library (IRL) used to describe all the aspects of a Green program. At runtime a program knows all its classes, the instance variables and methods of each class, the parameters of each method, the inheritance relationships, and so on.

As an example, the code below prints the names of the methods of class `Student`.

```
// assume Student is subclass of Person
var s : Person;
s = Student.new("Mary", 1980);
var v : array(ClassMethodInfo)[];
v = s.getClassInfo().getMethods();
   // v now has a description of all Student methods
for i = 0 to v.getSize() - 1 do
   Out.writeln( v[i].getName() );
```

Green gives the same treatment to regular objects and class objects. Although class objects are classless, we can get information on their methods through the IRL. But in this case we should use methods not shown in the previous example.

## 5 Other Language Features

Green supports assertions, which are expressions that should be true before and/or after a method is called at runtime. For example, the method

```
proc push( x : integer )
   assert
      before not full();
      var oldSize : integer = getSize();
      after not empty() and oldSize == getSize() - 1;
```

```
   end
   begin // method body
   ...
   end // push
```

of a class `Stack` defines a pre-condition "`not full()`" which should be true when the method is called and before its body is executed. If this condition is false, an unchecked exception is thrown. "`full`", "`empty`", and "`getSize`" are methods of `Stack`.

The expression following "`after`" is the post-condition. It should be true after the method ends its execution. If it is not, an unchecked exception is thrown. Between the `before` and `after` clauses there may appear zero or more variable declarations. These variables are set after the pre-condition is tested and they should be used only in the `after` clause.

A method may have a variable number of parameters. An example is the method

```
   // the ... is part of the Green syntax
proc print( numSpaces : integer;
            v : ... array(Person)[] )
     var i : integer;
   begin
   for i = 0 to v.getSize() - 1 do
      begin
      printSpaces(numSpaces);
      Out.writeln( v[i].getName() );
      end
   end
```

The real arguments passed to `print` should be an integer (for `numSpaces`) and any number of objects of class `Person` or its subtypes.

Green has some standard class objects used for input/output/memory management/runtime support. Class objects `In` and `Out` are used for standard input and output. Class object `Memory` supplies methods for memory management such as `doGarbageCollection`, `collectionOn`, and `collectionOff`.

Class object `Runtime` defines methods `exit` (ends a program), `getClasses` (returns a vector with information on all program classes), `getCatchObjectStack` (returns a stack with the catch objects — see Section 3), etc.

Green supports parameterized classes. An example is class Stack:

```
     class Stack(T)
```

```
    ...
end
```

A stack is declared as in "`var s : Stack(char);`". Parameter `T` is replaced by `char` in the *text* of class `Stack` and a new class is created and compiled. Each new parameter means a new class. `T` must be used as a type inside class `Stack`.

Class `Stack` may have been declared as "`class Stack(T : Window)`". In this case, the real parameters to class `Stack`, besides being classes, should be subtypes of `Window`. And they should define all `init` methods `Window` defines. That is, if class `Window` defines "`init()`" and "`init(integer)`", so should the classes that are real arguments to `Stack`.

## 6 Conclusion

All of the features of the Green language were designed simultaneously. That prevented incompatibilities between newly added and old features. During the design of the language, the introduction of a new construct/feature triggered an examination of the whole language for incompatibilities between features or implementation problems. Frequently a new construct caused modification of old ones, keeping the parts compatible with each other.

This interactive language design is also responsible for the *relative* simplicity of the language.[1] Sometimes a construct was modified to make other simpler. For example, metaobjects of other languages [3] [4] [7] [8] use special protocols to intercept object creation.[2] We could have supplied a protocol for object creation in Green too. But we chose to add `new` methods to class objects to create new objects. Then to intercept object creation in Green is to intercept a `new` method of a class object — this is a regular operation of shells (metaobjects) that needs no special protocol. Then the `new` methods of class objects simplified the design of metaobjects.

In other statically-typed languages [16] [19], the set of static variables and methods of a class do not compose an object. If they did, it would be difficult to assign a type to this object, since it does not have a class. Green solves this problem by allowing *objects* to have types. The static variables and methods of a class are grouped into a classless *class object* that has a type. Then a

[1] We consider it simple by the power it offers.

[2] These protocols are some special commands that have the only duty of intercepting object creation.

class object can be assigned to a variable, passed as a parameter, etc. The integration of class objects into the type system is only possible because:

- objects have types;
- the compiler adds methods to each class object to make it a subtype of `AnyClassObject` (and therefore a subtype of `Any`).

The use of classless class objects brings several benefits to Green: there is no infinite regression of metaclasses, types are applied to both classes and objects, and classes become first-class objects. Besides that, class objects are easy to implement: they are just the single object of a hidden class. Note that a class object can be used as a *single* object — an object of a class that has a single instance (like `Earth`, for example).

In Green, a method `show(any : Any)` can accept anything as parameter: an object of any class, any class object, and any basic value (`2.71`, `3`, `'A'`). The basic value is first cast automatically to an object of its wrapper class. Then, `show('A')` is transformed into `show(Char.new('A'))`. The automatic casting of basic values and the integration of class objects into the type system endows Green with Smalltalk-like polymorphic power.

One of the most important innovations of Green is its Exception Handling System (EHS). The EHS is original because it is object-oriented: exception handling is encapsulated into `throw` methods of catch clauses. Therefore, exception handling can be reused, which is not so easy in conventional EHS. The maintenance of exception handling in Green is not difficult because in general the handling of a specific exception will be put in a single `throw` method (or in a few methods of different catch classes). Changes in the handling can be made by changing this method (or these few methods). Compare with conventional EHS in which the handling of an exception is in catch clauses spread throughout the code. It is easy in Green to handle an exception in exactly the same way whenever it is thrown in any place of the program.

Green uncouples error detection, made by `try` blocks, from exception handling, made by catch objects. We can change the catch objects at runtime, changing dynamically error handling. This makes it possible to start with a simple error handling which may easily and smoothly evolve to a more sophisticated treatment.

Green uses object-oriented rules to check the correctness of exception signaling. A statement "`exception.throw(anException)`" is legal only if `exception` has a `throw` method that may accept `anException` as parameter. The type of exception is a union of types of catch objects (parameters to try blocks) and the type of the `exception` parameter of the method, if one exists. To explain completely this subject is out of the scope of this paper. It is discussed in length by Guimarães [11]. The important point to note is that

Green uses object-oriented rules in its exception system.

Exception classes may be subclassed as in other languages. The novelty in Green is that catch classes may also be inherited. For example, a class could inherit from `CatchDateException` redefining just one `throw` method. The programmer can build hierarchies of catch classes tailoring exception handling.

At runtime, the program may inspect the stack of catch objects (parameters to try blocks). It may also attach a shell (metaobject) to one of the catch objects thus changing exception handling.

Green brings the object-oriented advantages to its exception handling system: there may exist catch and exception hierarchies, polymorphism in exception handling (by changing the catch objects), polymorphism with exception objects, and reuse of code for error handling. Besides that, the type system is used for checking exception handling. The EHS is integrated in the language, interplaying with other language features.

The Green metaobjects are called shells and have two main characteristics: they are simple and efficient. Shells use a simple sintax and have a simple semantics. They can be learnt in a few minutes, a contrast with more complex metaobject protocols. Shells are also very efficient — a comparison of the performance of shells with metaobjects of other languages is made by Guimarães [9].

Green supports some features that are not fundamental to object-oriented programming but that certainly makes programming easy and/or safe. These features are assertions, standard class objects for I/O/memory/runtime support, and parameterized classes. Other small features of the language were not discussed in this paper as casting made by methods (for example, "`letter_A = char.cast(65);`" ), expanded variables (variables that are not pointers — they obey value semantics), constructors of class objects, enumerated constants, `subclass` section of a class, methods of the array classes, methods of basic types, the standard exception and catch hierarchies, dynamic extensions (to replace methods of all objects of a class), and the runtime model. All of this can be found in the Green manual [10].

A Green compiler is available in the Internet [10]. It implements everything discussed in this paper but shells, expanded variables, and methods with variable number of parameters. However, shells have been implemented by a previous compiler.

**Acknowledgments.**

This work was partially financed by FAPESP under process number 99/13006-8.

## References


[1] America P, Linden F V D. A parallel object-oriented language with inheritance and subtyping. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). New York: ACM, 1990. p. 161-168.

[2] Bouraqadi-Saâdani N, Ledoux T, Rivard F. Safe metaclass programming. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). New York: ACM, 1998. p. 84-96.

[3] Brandt S, Schmidt R. Reflection in a statically typed and object-oriented language — a meta-level interface for BETA. `http://www.daimi.au.dk/~beta`, 2003.

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

[5] Cointe P. Metaclasses are first class: the ObjVlisp model. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). New York: ACM, 1987. p. 156-162

[6] Goldberg A, Robson D. Smalltalk-80: the language and its implementation. Reading, MA: Addison-Wesley, 1983.

[7] Golm M. Design and implementation of a meta architecture for Java. Master's thesis. University of Erlangen-Nrnberg, 1997.

[8] Golm M, Kleinöder J. MetaXa and the future of reflection. UTCCP Report, Center for Computational Physics, University of Tsukuba, Tsukuba, Japan, 1998.

[9] Guimarães J. Reflection for statically typed languages. In: European Conference on Object-Oriented Programming (ECOOP). Berlin: Springer, 1998. p. 440-461.

[10] Guimarães J. The Green language. http://www.dc.ufscar.br/~jose/green/green.htm, 2003.

[11] Guimarães J. The Green language exception system. http://www.dc.ufscar.br/~jose/green/green.htm, 2003.

[12] Guimarães J. The Green language type system. http://www.dc.ufscar.br/~jose/green/green.htm, 2003.

[13] Harris W. Contravariance for the rest of us. Journal of Object-Oriented Programming 1991; 4(7):10-18.

[14] Ledoux T, Cointe P. Explicit metaclasses as a tool for improving the design of class libraries. In: International Symposium on Object Technologies for Advanced Software (ISOTAS). Berlin: Springer, 1996. p. 38-55.

[15] Meyer B. Eiffel: the language. New York: Prentice Hall, 1992.

[16] Niemeyer P, Peck J. Exploring Java. Second Edition. Sebastopol, CA: O'Reilly & Associates, 1997.

[17] Raj R, Tempero E, Levy H, Black A, Hutchinson N, Jul E. Emerald: a general-purpose programming language. Software: Practice and Experience 1991; 21(1):91-118.

[18] Rodriguez N, Ierusalimschy R, Rangel J. Types in School. SIGPLAN Notices 1993; 28(8):81-89.

[19] Stroustrup B. The C++ programming language. Second Edition. Reading, MA: Addison Wesley, 1991.

## Vitae

Guimarães, José de Oliveira: is an Associate Professor of Computer Science at the Federal University of São Carlos (UFSCar), São Carlos-SP, Brazil. He received a BSc, a MSc, and a PhD in Computer Science in 1989, 1992, and 1996, respectively. His research interests include object-oriented programming, computational reflection, compiler optimizations, and programming languages.