Comparison of two object-oriented languages: C++ and Ruby

Jeroen Heymans

December 29, 2011

1 Introduction

1.1 Foreword

This paper is written for the master course Principles of Object Oriented Languages on the VUB. It contains a comparison between two languages that contain object-oriented features.

1.2 The compared languages

The choice for this paper was C++, a well-known static typed language with good object oriented features, and Ruby, a dynamic typed language that was designed to be as object-oriented as possible.

1.2.1 C++

Originally called "C with classes", C++ is a multi-paradigm language based on the popular language C. C++ doesn't require you to write all code in an object-oriented fashion, it is also possible to write all your code in functions.

1.2.2 Ruby

The creator of Ruby, Ykihiro Matsumoto, looked at other languages to find an ideal syntax. He watned a scripting language that was more powerful than Perl and more object-oriented than Python. In Ruby, everything is an object. It was influenced by Smalltalk, every type was given methods instance variables.

1.3 What follows next

We will compare several important features that belong to object-oriented languages like inheritance, polymorphism, access control and many more.

2 Comparisons

2.1 Classes

In object-oriented programming, a class is a blueprint that is used to construct objects which we call object instances. We compare the several types of classes that can be made.

2.1.1 Abstract classes

An abstract class is a class that can not be instantiated for it is either labelled as abstract or it specifies abstract methods. The general idea of abstract classes is that the class must be extended. In C++, an abstract class is a class having at least one pure virtual function. For example, this is an abstract class in C++:

```
class A {
  public:
    virtual void f() = 0;
};
```

If we are trying to compile C++ code where we have instantiated an abstract class, for example like this:

```
int main() {
   A a = new A();
   return 0;
}
```

We will receive an output from the compiler that notifies us that this operation is not allowed:

```
g++ a.cpp -o a
a.cpp: In function 'int main()':
a.cpp:10:16: error: cannot allocate an
    object of abstract type 'A'
a.cpp:3:9: note: because the following
    virtual functions are pure within 'A':
a.cpp:5:16: note: virtual void A::f()
a.cpp:10:6: error: cannot declare variable '
    a' to be of abstract type 'A'
a.cpp:3:9: note: since type 'A' has pure
    virtual functions
make: *** [a] Error 1
```

Abstract classes in Ruby are a different story. Unlike some object-oriented languages, Ruby does not support a keyword like ”abstract” or a way like C++ to define an abstract class. Ruby does not provide the necessary tools to create abstract classes. However, the programmer himself can force a class to behave like an abstract class when he makes *:new* private. The extending class must then make *:new* public:

```
class A
  private_class_method :new
  def initialize(txt = "Hello!")
    puts txt
  end
end

class B < A
  public_class_method :new
end
```

2.1.2 Concrete classes

2.1.3 Inner classes

2.1.4 Metaclasses

In Ruby, a class is also an object. Each class is an instance of the unique metaclass which is built in the language.

2.1.5 Non-subclassable

2.1.6 Partial classes

2.1.7 Uninstantiable classes

2.2 Inheritance

In Ruby, a class can only inherit from a single other class. Some other languages like C++ support multiple inheritance, a feature that allows classes to inherit features from multiple classes.

```
class Animal
  def breath
    puts "Breathe"
  end
  def speak
    puts "Random noise"
  end
end

class Cat < Animal
  def speak
    puts "Meow"
  end
end
```

When using these two classes, one can see that methods will be overriden:

```
cat = Cat.new
cat.speak
cat.breath
```

This will output:

```
Random noise
Breathe
```

Inheritance in C++ is more advanced as it supports multiple inheritance.

Ruby mixin’s.

2.3 Access control

2.3.1 Derived classes

When declaring a derived class in C++, we can provide an access specifier: *public*, *protected* or *private*. For example if we have the base class A, we can create subclass B from A and provide one of the three access specifiers like this:

```
class A { };
class B: public A { };
```

The purpose of this access specifier is to alter the access control of the members of the base class via the derived class. The access control is not altered on the original base class. An example:

```
#include <iostream>

class A {
  public:
    int foo;
    A(): foo(0) {};
};

class B : public A { };

int main()
{
   B* b = new B();
   b->foo = 3;
   std::cout << "Value foo: " << b->foo <<
       std::endl;
   return 0;
}
```

When compiling and excecuting this code, the program will output correctly:

```
Value foo: 3
```

If we however change the line access keyword when defining the class B to for example *protected* or *private* (instead of the current *public*), we get a compilation error:

```
g++ a.cpp -o a
a.cpp: In function 'int main()':
a.cpp:5:7: error: 'int A::foo' is
     inaccessible
a.cpp:14:7: error: within this context
```

```
a.cpp:5:7: error: 'int_A::foo' is
      inaccessible
a.cpp:15:40: error: within this context
make: *** [a] Error 1
```

When deriving a class, the following situations can occur:

- Derive as *public*: public and protected members of the base class remain public and protected members of the derived class.
- Derive as *protected*: public and protected members of the base class become protected members of the derived class.
- Derive as *private*: public and protected members of the base class become private members of the derived class.

In all cases, private members of the base class remain private, e.g.: not even the derived class can access them unless there is a specific friend declaration within the base class that explicitly grants access to them.

2.4 The use of super and this/self

2.5 Polymorphism

2.6 Interfaces

2.7 Reflection

2.8 Clonable objects

2.9 Everything is an object?

3 Conclusion

4 References

http://www.ruby-lang.org/en/about/
http://www.cplusplus.com/info/history/
http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8l.doc%2Flanguage%2
http://www.klankboomklang.com/2007/10/05/the-metaclass/ http://ruby-extra.rubyforge.org/classes/Object.html
http://www.wikyblog.com/AmanKing/Metaclass$_i n_R uby http:$
$//www.klankboomklang.com/$2007/10/12/$objects-classes-and-jruby-internals/http:$
$//shiningthrough.co.uk/A-comparison-of-Ruby-pass-by-reference-and-C++-pass-by-referencehttp:$
$//rubylearning.com/satishtalim/ruby_i nheritance.htmlhttp:$
$//publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp$
%2$Fcom.ibm.xlcpp8l.doc$%2$Flanguage$%2$Fref$%2$Fcplr$130$.htm$