

Translation Guidelines

From JQuantLib

Understand C++ code sometimes is very hard. The difficulty comes to the fact that a C++ programmer is free to do almost whatever they wish regarding how to organise source code, assign synonyms to anything via macros, templates and typedefs... oh, well the list of tricks may be exhaustive and varies a lot but mainly due to the intrinsic complexity of the original object model.

Contents

- 1 Divide to Conquer
 - 1.1 A class or an interface?
 - 1.2 Copy the header file
 - 1.3 Access modifiers for fields and methods
 - 1.4 Translate constructors first
 - 1.5 Use TokenType
- 2 To be organized

Divide to Conquer

What do you do when you face a class with dozens of methods belonging to a tangled object model?

Divide the problem into smaller pieces. This is frequently the only feasible approach.

Start from the beginning: create a very simple declaration of a Java class relative to that C++ class and put our copyright notice on top of it. Copy the original C++ copyright notice after our copyright notice.

If you think that you need to translate two or more classes in parallel due to their similarity, hierarchy, coupling or any other reason, the first step is really identify what their names must be and put the copyright notices on the top.

This is not very much but it is at least the first step done, done once and done properly and you will never have to return to the copyright notice again.

Do it right at your first attempt and you will never have to do it again.

A class or an interface?

The next step is amend the declaration of your Java class in order to extend the correct classes and implement the correct interfaces. Another important point to consider is how the current class you are working on participate in the object model. You need to consider that Java does not provide multiple inheritance. Depending on the way *this* class participate in the object model, you'd better create an interface and change *this* class in order to implement that interface. There are scenarios where it's clear we need an interface even because the original C++ code looks pretty much like an interface or even because it's explicitly stated in the original comments. When we start to understand well what is happening with the original C++ code, it starts to become clear whether an interface should be used or not. The rule is more or less like this:

Do your best efforts to determine if you need a class or an interface. If after enough analysis you still in doubt, start using a class and continue translating the extended classes, etc. By the time it will become clear in your mind whether you need to refactor your code or not.

Copy the header file

Copy the C++ header *.hpp* file definitions inside your Java class. References (the **&** symbol) and *typedefs* in the C++ sources, which will give you a lot of compiler errors in Java. You may become confused with so many compiler errors.

*You can do some basic housekeeping simply replacing **&** by "" (nothing). You can also substitute **->** by "." (period). You can also substitute keywords like **virtual** and **const**, **mutable** by their "equivalents" in Java.*

C++	Java	Comments
const	final	On field, method and parameter modifiers
const	@ReadOnly	See this article
virtual	abstract	The keyword <i>virtual</i> means " <i>not-final</i> ", meaning that the method can be overridden. It cannot be translated directly to <i>abstract</i> in Java: you have to judge yourself whether <i>abstract</i> should be applied or not.
mutable		This is another trick of C++ language. This excellent article (http://www.highprogrammer.com/alan/rants/mutable.html) explains in detail and gives several examples

Care must be taken with the original C++ code. Remember that QuantLib is very well designed and implemented. Be sure that there are always good reasons for the tricks they use in their code. If you are not sure of what is happening, simply

*Put **//TODO: code review :: blah, blah blah** in your Java code so that you can return later and fix possible issues when you have better understanding of the entire scenario.*

Access modifiers for fields and methods

You can have a bad time trying to identify if fields and methods should be private, public, protected, final, abstract or virtual. It happens usually because the C++ code is disjoint, I mean: definitions in one file and implementation in another file, usually... As I said, the C++ programmer is free to do whatever mess they wish to do.

Understand properly the visibility and protection of fields and methods is much more important than it seems at first glance. This understanding is critical for your translation as it will help you visualise the object model and interdependencies between classes. Understanding properly, you will be able to eventually propose a slight different object model that fits better in the way Java applications are built. The general rule is:

Copy the header file into your Java file and start translating from there. The header file will give you precious information about visibility of fields and methods.

After you copied the header file, translate the code and reorganise it in order to explicitly expose the

visibility and protection of fields and methods. Organise methods and fields accordingly, keeping all *public final* methods together, for instance. The general rule here is:

Eradicate the mess from the beginning. The earliest you organise your code, the easiest it will be to understand the original C++ code and produce high quality Java code.

Below you can see an example how a C++ header file was translated and reorganised.

```
//  
// private static final fields  
//  
private static final String NON_NEGATIVE_RATE = "rate cannot be negative";  
  
//  
// private final fields  
//  
private final double rate;  
private final double time;  
  
//  
// private fields  
//  
private double max;  
private int counter;  
  
//  
// public constructors  
//  
MyExample(double rate, double time) {  
}  
  
//  
// public final methods  
//  
double final getRate() {  
}  
  
double final getTime() {  
}  
  
//  
// TODO: code review  
// It's not clear what these methods do and how they are used  
//  
private final getSuspiciousThing() {  
}
```

Sometimes is not that easy to do this initial step because a C++ header file may contain implementation code as well, not only declarations.

Again, focus your efforts on organising the code properly. It will help you create small clusters of code without compiler errors in the beginning of your translation. As your translation goes further, your well organised code will help you isolate small clusters of code which do not compile yet.

Sometimes, due to the use of *typedefs* or templates probably, you are really confused about a fields or method. You may even be in doubt if you have a field or a method in your hands and it is not rare to be in doubt if it is a local element or something shared from some base class, or something that will be defined in an extended class.

In case of doubt, assign "private final" to a field or method. Don't worry, the compiler is your friend, in the future it will tell you where you were wrong or not.

Translate constructors first

Starting translating first the constructors is more or less natural because they appear on the top of the source code in general. Not only for this reason, but...

Because we need to understand what the original C++ code does, it's critical to understand what constructors do and how they are called by extended classes and other classes.

It's a good opportunity to reinforce your understanding about the object model. For instance, if you believed that *this* class is extended from a certain base class, but then you realize that there's a pointer around and fields are being accessed via this pointed. Imagine that, after some analysis, you discover that *this* class not only extends a certain base class, but also has an association or composition (<http://ootips.org/uml-hasa.html>) to objects of that same base class. obviously, this understanding is critical for your translation.

To translate a constructor, follow the suggested sequence:

1. Copy its code from the C++ implementation *.cpp* file into the previous skeleton constructor you created when you first organised your code.
2. Substitute *typedefs* by JSR-308 compliant counterparts.

Use TypeToken

Consider the following C++ template:

```
template <class T>
class Klass : public T {
    ...
}
```

There is no equivalent for this in Java. The following code will NOT compile.

```
public class Klass<T> extends T {
    ...
}
```

The "conventional way" would be (a) allocate the delegate instance and (b) pass the allocated instance to the class constructor

```
abstract class SomeAbstractClass {
}

class OneConcreteClass extends SomeAbstractClass {
}

class AnotherConcreteClass extends SomeAbstractClass {
}

class Klass<T> extends SomeAbstractClass {
    private T delegate;

    public Klass(T t) {
        this.delegate = t;
    }
}
```

This certainly compiles and comes close to the C++ template mechanism but is not wholly correct because the original C++ sources sometimes is interested on extending `OneConcreteClass` or `AnotherConcreteClass` and so on. Also this implementation is open to abuse as seen in the following example:

```
OneConcreteClass delegate = new OneConcreteClass();  
  
Klass<OneConcreteClass> klass = new Klass(delegate); // This is correct  
  
Klass<AnotherConcreteClass> klass1 = new Klass(delegate); // This also compiles but is not desirable
```

The solution is allow `Klass` to allocate the class it needs (a class derived from `SomeAbstractClass`) depending on the generic parameter passed as [generic] parameter. Enter `TypeToken`. Using `TypeToken`, you keep the responsibility of allocating the delegate inside the constructor, like this:

```
class Klass<T extends SomeAbstractClass> {  
  
    private T delegate;  
  
    public Klass() {  
        this.delegate = null;  
        try {  
            delegate = (T) TypeToken.getClazz(this.getClass()).newInstance();  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

And now all you have to do is this:

```
Klass<OneConcreteClass> klass = new Klass();
```

What is the downside to doing this? Well one obvious one is this style is not particularly conducive to injection. Also, if you use some type of factory pattern to create delegates, then that might also need to be reworked.

But all in all this is a much cleaner and correct way of doing things and in addition, using `TypeToken` keeps JQuantLib API very close to original C++ QuantLib API.

To be organized

Why you put *final* everywhere?... even primitive type parameters?

The keyword *final* means that a certain variable cannot be changed. It's a conceptual thing. It does not matter whether your variable is primitive type, array or Object. We are simply willing to keep its value constant during its lifetime. Period.

In particular, when *final* appears in parameters, some people erroneously think that we are trying to guarantee unmutability of the variable which was originally passed as reference. This is not completely correct:

- This is correct to think we are really willing to guarantee unmutability of the original object passed as argument. We are really willing to keep its original value for the lifetime of the method call. When Objects are passed, the called method is able to change its parameters contents by directly changing them or calling methods which are not *final* in them. This is a good programming practice to avoid this possibility.
- But it is wrong to think there's such thing as *pass by reference* in Java. It's very important to understand that Java **never** (I repeat: **never!**) passes arguments by reference: it does not matter if you passing a primitive type, an array or an Object. This concept is very important to be understood specially by people migrating from C/C++. This concept is very well explained at <http://javadude.com/articles/passbyvalue.htm> and will not be repeated here. Please have a look. In addition, you may want to consider the following example:

```
public class QuickTest {

    public static void main(String[] args) {

        Double d1 = 0.0; // Doubles are immutable

        f(d1);

        System.out.println(d1);

        double d2 = 0.0; // doubles are immutable

        f(d2); // autoboxing! :(

        System.out.println(d2);

        Structure s1 = new Structure(0.0); // classes are mutable

        g(s1);

        System.out.println(s1.value); // ooops!

    }

    private static void f(Double d) {

        d = 1.0;

    }

    private static void g(Structure s) {

        s.value = 1.0;

    }

    private static class Structure {

        public double value;

        public Structure(double value) {

            this.value = value;

        }

    }

}
```

But ... *final* even for primitive types? Why?

Simply because the value is conceptually immutable. In particular it may be a primitive type but it does not change the concept, anyway. Consider a theoretical situation where you suddenly decide to search/replace some occurrences of *double* by *MutableDouble* in your application: your methods will

continue to enforce immutability of parameters. You will not be allowed to change the calling variable by accident.

What the hell is those annotations @Time, @ReadOnly, etc all around?

Conceptually 5 pears are different from 5 apples and you should not assign the quantity of apples to a variable intended to store the quantity of pears. The upcoming JDK7 will provide means of detecting such conceptual errors. For more information see Strong Type Checking and Providing unmutability to receivers

Use primitives

Prefer double instead of Double, long instead of Long and so on, unless you really need to test nullity for some reason. Even in this case, its preferable to use Double.NaN instead of null to mark a special condition where a double variable cannot be considered.

Retrieved from "http://www.jquantlib.org/index.php/Translation_Guidelines"

-
- This page was last modified 22:31, 16 October 2008.
 - Content is available under GNU Free Documentation License 1.2.