Improving the Java Type System

António Menezes Leitão

February, 28, 2013

1 Introduction

Although Java is a strongly typed language, its type system is not sufficiently powerful to be usable in some interesting situations.

One such situation occurs with the initialization of fields, also known as instance variables. Although the Java language requires that local variables must be initialized before being used, it does not have a similar requirement for instance variables, instead defining its automatic initialization to default values. This behavior allows code to run with objects that were not explicitly initialized.

Another situation occurs with subtypes of the available types, for example, a field that can only contain positive integers. Since the Java language does not provide a predefined type for positive integers, the programmer must overcome this limitation by implementing a new (reference) type. Unfortunately, the elements of this new type cannot be operated with the primitive arithmetic operations, such as `+` or `*`. On the other hand, if the programmer prefers to use the primitive operations, he can declare the field as, for example, type `int` but then there is nothing that prevents that field to be assigned with a negative integer.

For a more complex example, consider a pair of fields where one of the fields must not be smaller than the other. This might be useful, for instance, to implement ranges. Again, it is not possible to declare this relation in Java in a way that the type system can enforce.

Finally, consider method declarations: it might be useful to establish constraints for the parameters or return types of methods so that it becomes possible to express some interesting properties such as the fact that, for positive arguments, the result must be positive.

In spite of Java's type system limitations, it is possible to augment a Java program with *annotations* that express additional constraints that must be checked during program execution. It might then be possible to automatically verify those constraints whenever a field is initialized, used, or updated, or whenever a method is called.

2 Goals

Implement, in Java, a set of extensions that can be applicable to the above mentioned scenarios. You must implement, at the very least:

- A mechanism for annotating a field declaration with an assertion that must be enforced each time the field is assigned.

- A mechanism that prevents an annotated field of an object to be read without having been initialized by the program (that is, without considering the automatic initialization done by the language).
- A mechanism for annotating a method declaration with an assertion that must be enforced immediately before the method returns.

We will now describe these mechanisms.

2.1 Field Assertions

It should be possible to annotate each field declaration with one assertion. As an example, consider the following class declaration:

```
public class Test {
    @Assertion("foo>0")
    int foo=1;

    @Assertion("bar%2==0")
    long bar;

    @Assertion("baz>foo")
    int baz;

    @Assertion("quux.length()>1")
    String quux;
}
```

In the above example, the field `foo` must always be a positive integer, the field `bar` must be an even number, the field `baz` must always be bigger than `foo` and, finally, `quux` must have a length greater than `1`.

To this end, you must define the annotation `@Assertion("`*expression*`")` where *expression* is any acceptable Java expression of type `boolean`, named *assertion*. Each time an annotated field is updated, the assertion is evaluated. If its value is `false`, then a `RuntimeException` is thrown, describing the assertion.

As an example, consider the following instance initializer of class `Test`:

```
{
    bar=2;
    baz=3;
    bar+=2;
    quux="foo";
    bar++;
}
```

In the above example, it should be obvious that all statements except the last verify the corresponding assertions. However, the statement `bar++` violates the assertion that `bar` must be an even number. As a result, any attempt to create an instance of `Test` will throw an exception that, if not catched, will show as:

```
The assertion bar%2==0 is false
```

In the general case, the `RuntimeException` must be initialized with the string `"The assertion` *expression* `is false"` where *expression* is the assertion expression that was violated.

2.2 Field Initialization

Any field annotated with `Assertion` will be checked for proper initialization before use.

As an example, consider:

```
public class TestInit {

    @Assertion("true")
    int foo;

    {
        foo++;
    }

    public static void main(String args[]) {
        new TestInit();
    }
}
```

As it should be obvious, the field `foo` is not explicitly initialized before the statement `foo++` attempts to update it by incrementing its *previous* value. As a result, an exception is thrown.

In the general case, when a field named *foo* was not explicitly initialized before being used, the `RuntimeException` that is thrown must be initialized with the string `"Error:` *foo* `was not initialized"`.

2.3 Method Assertions

It should be possible to annotate each method declaration with one assertion that is checked immediately before the method returns. As an example, consider the following class declaration:

```
class Base {

    @Assertion("($1>=0) && ($_>$1)")
    public int fooBar(int x) {
        return ++x;
    }
}
```

In the syntax for method assertions, the meta-variable `$_` represents the returned value of the method, the meta-variable `$0` represents the method receiver (`this`), and `$1`, `$2`, ..., `$`n, represent the corresponding method arguments.

Using this syntax, the assertion in the above example means that the method `fooBar` only accepts a non-negative argument and promises to return a number greater than its argument.

One very interesting property of method assertions is that they should be *non-overridable*. Returning to the above example, this means that a subclass of `Base` that overrides the method `fooBar` will inherit the same assertion that was established for the base method. As an example, consider:

```
public class Derived extends Base {

    @Override
    @Assertion("($1%2==0) && ($_%2==1)")
    public int fooBar(int x) {
        return x+1;
    }
}
```

Note that class `Derived` overrides method `fooBar` from class `Base`. However, the method will include not only its own assertion (namely, that the argument is a even number and the result is an odd number), but also the assertions of all overriden methods in the hierarchy chain. In this case, it means that the argument must be a non-negative even number and the result must be an odd number bigger than the argument.

2.4 Extensions

You can extend your project to further increase your grade. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Some of the potentially interesting extensions include:

- Checking array initialization
- Checking all fields involved in an assertion
- Checking method assertions on method entry and method return
- Checking assertions over constructor arguments

3 Code

Your implementation must work in Java 6 and should use the *bytecode* minipulation tool Javassist, version `3.17.1-GA`.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

You must implement a Java class named `ist.meic.pa.CheckAssertions`, containing a static method `main` that accepts, as arguments, the name of another

Java program (i.e., a Java class that also contains a static method `main`)) and the arguments that should be provided to that program. The class should (1) operate the necessary transformations to the loaded Java classes so that fields annotated with assertions will be checked during the execution of the program, and (2) should transfer the control to the `main` method of the program.

4 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 15 minutes slot (approximately, 8 slides), should be centered in the architectural decisions taken and might include all the details that you consider relevant. You should be able to "sell" your solution to your colleagues and teachers.

5 Format

Each project must be handled by electronic means using the Fénix Portal. Each group must handle a single compressed file in ZIP format, named as `checkassertions.zip`. This file must contain:

- the source code,
- the slides of the presentation,
- a `build.xml` file that compiles the Java source code and generates the `jar` file `checkassertions.jar`,
- the Javassist `jar` file for a correct building process.

The accepted format for the presentation slides is PDF. This file must be located at the root of the ZIP file and must have the name `p1.pdf`.

The file `build.xml` is an Ant file (`http://ant.apache.org`) and should build the `checkassertions.jar` file in the same location, containing all the compiled Java code necessary to run the assertion checking process, as described previously. The default target for `build.xml` should create the JAR. This means that writing in a shell

```
$ ant
```

should build the expected file `checkassertions.jar`.

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner working of the developed project, including demonstrations.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code and the slides must be handled via Fénix, no later than 20:00 of **March**, **26**.

The presentations will be done during the classes after the deadline. Only one element of the group will present the work and the presentation must not exceed 15 minutes. The element will be chosen by the teacher just before the presentation.