SWEN326: "Safety-Critical Systems"
Java Coding Standard

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

March 9, 2020

Contents

1 Introduction

This document sets out the required coding standard for SWEN326 "Safety Critical Systems". The purpose of this is to capture the spirit of several coding standards widely used in the context of safety critical systems development. However, whilst such standards are typically focused around the C programming language, this standard is focused around Java. The contents of this document are inspired by the "Power of Ten" rules [**?**] along with other standards, including MISRA-C [**?**], the JPL institutional Coding Standard [**?**] and the (draft) Safety Critical Java standard (JSR302).

An important part of this standard is the emphasis on machine-checkable requirements. Holzmann notes the following [**?**]:

> *"The most dooming aspect of many of the guidelines is that they rarely allow for comprehensive tool-based compliance checks"*

Where possible, we indicate clearly how the requirements of this standard can be automatically enforced. However, in some cases, no tool is currently available and manual enforcement (whilst undesirable) is required.

1.1 Java

This standard is focused towards the development of Java for safety critical *embedded* systems. Unlike C, Java does not have an extensive history in embedded systems development. Embedded systems are often characterised by limited resources, such memory capacity and processor power. In contrast, Java was designed to run on desktop machines which are considerably more powerful and, for example, can be considered to have unlimited memory capacity. In particular, the Java Virtual Machine (JVM) on which Java programs are executed is not at all suited to running in an embedded environment. Many aspects, such as garbage collection, reflection and the Java Native Interface are not easily adapted to an embedded environment. To that end, most efforts to run Java on embedded systems restrict Java to a very limited subset which, for example, eliminates the need for a garbage collector. This standard is not an exception here, and places some very strong restrictions on conforming Java programs to ensure they are easily compiled for an embedded system.

1.2 Tooling

This standard is particular concerned with tooling which can be used to enforce the rules contained herein. As such, it discusses various mechanisms by which specific rules can be enforced using off-the-shelf tooling. In this regard, the tools considered include *Eclipse* and *Checkstyle*.

2 The Rules

These rules are based on the well-known "Power of Ten" rules developed at NASA's Jet Propulsion Laboratory by Gerald Holzmann [**?**]. Since the original rules were designed for programs written in the C programming language, they have been adapted for Java here. In some cases, this means the rule has simply been removed as there is no sensible adaptation for Java (e.g. because no preprocessor exists in Java).

This standard assumes the reader is familiar with the original "Power of Ten" rules, and their rationale. As such, rationale for the rules themselves is not given though, in some cases, rationale for the manner in which they have been adapted is.

2.1 Rule 1: Control-Flow is Restricted

Java does not support the **`goto`**, `setjmp` and `longjmp` statements found in the C programming language. As such, they can be safely ignored. However, the following restrictions are placed on the use of control-flow in Java:

- **Recursion** is not permitted, either *directly* or *indirectly*.
- **Foreach Loop.** Java's foreach loop is not permitted as this dynamically allocates heap memory (see Rule 3 below).
- **Exceptions.** Following many safety-critical coding guidelines, exceptions (including **`try/catch`** blocks) are not permitted.

2.2 Rule 2: Loops are Bounded

This rule can be applied relatively easily to Java though, for simplicity, we refine its meaning here. All loops must be Java **`for`** loops of the form:

```
for(int i = e1;i < e2;i=i+1) {
  ...
}
```

Here, `i` is a placeholder for any valid variable name, whilst `e1` and `e2` are arbitrary integer expressions which don't involve `i`. Likewise, `i` may not be modified in the body of the loop.

2.3 Rule 3: Dynamic Memory Allocation Limited to Initialisation

This rules places some severe limitations on the style of Java code which can adhere to this standard as, in the general case, the use of **new** is prevalent in Java. However, this rule is critical as it enables conforming Java code to be *executed without the need for a garbage collector*.

To enable simple checking of this rule, we take inspiration from the Safety Critical Java specification. Specifically, all constructors are considered part of the initialisation phase and, additionally, we define the following annotations:

- `@Initialisation`. Any method annotated with this is considered part of the initialisation phase and, thus, may dynamically allocated memory. Furthermore, it can be called from a constructor as necessary. However, such a method *cannot be called from any method which is not itself annotated*.
- `@Immortal`. Any "**static final**" field annotated with this will be preallocated into immortal memory. Following SCJ, we note the cyclic dependencies between initialisers are not permitted.

The following provides a simple illustration of the `@Initialisation` annotation being used:

```
class List {
  private int[] data;

  public List(int size, int value) {
    this.data = construct(size,value);
  }

  @Initialisation
  public int[] construct(int size, int value) {
    int[] items = new int[size];
    for(int i=0;i<size;i=i+1) {
      items[i] = value;
    }
    return items;
  }
}
```

2.4 Rule 4: Functions are Restricted to 60LOC

This is a straightforward adaptation from the original "Power of Ten" rules. The intention is to ensure that the program is properly decomposed into small, well-defined functions.

2.5 Rule 5: Assertions are Used Appropriately

The intention of this rule is to promote the use of assertions within the codebase. Java provides the `assert` statement for this purpose.[1] Assertions should be used to check *pre-conditions* and other *class invariants*. The following illustrates:

```
int max(int[] items) {
  assert items != null;
  assert items.length > 0;
  //
  ...
}
```

[1] Remember that assertions must be enabled using `-ea` in Java!

Finally, as with the original "Power of Ten" rule, assertion conditions should be side-effect free.

2.6 Rule 6: Variables have Smallest Scope

This is a straightforward adaptation from the original "Power of Ten" rules. The following illustrates an example which does not adhere to this principle:

```
int find(int item, int[] items) {
  assert items != null;
  //
  int i;
  for(i = 0;i <items.length; i = i + 1) {
    if(items[i] == item) { return i; }
  }
  return -1;
}
```

This fails the smallest scope test because the variable `i` could have been declared *locally* to the **`for`** loop.

2.7 Rule 7: Return Values are Checked

This is a straightforward adaptation from the original "Power of Ten" rules. The following illustrates a program which does not adhere to this rule:

```
  ...
  int max(int x, int y) { ... }

  void update(int i, int j, int value, int[] data) {
    max(i,j);
    data[i] = value;
  }
  ...
```

This does not confirm to this rule because the return value for the invocation of `max(i,j)` is ignored, and this suggests a bug of some description.

2.8 Rule 8: References are Restricted

Whilst pointers in the C programming language are, in some sense, comparable with references in Java, they are also far more "flexible". However, the lack of real `struct` values in Java makes adapting this rule somewhat challenging. As such, this rule imposes the following restrictions:

- All instance fields must have primitive type (e.g. **`boolean`**, **`int`**, **`double`**, etc).
- The only exception is that arrays of primitive type (e.g. **`int`**`[]`, **`double`**) are permitted as instance fields provided they are **`final`**.

The latter item is really necessary to ensure the benefits of encapsulation in Java can still be exploited.

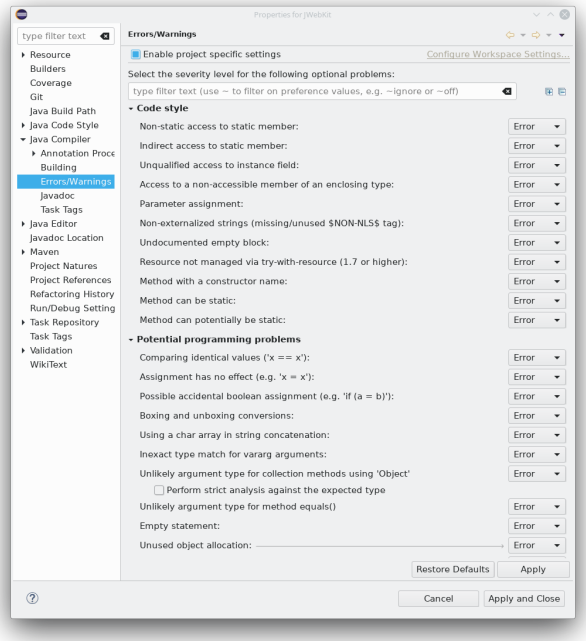
Function Pointers. The corresponding "Power of Ten" rule also prohibits the use of *function pointers*. Whilst these have no direct counterpart in Java they are comparable (in some sense) to dynamic dispatch. To that end, the following restrictions are imposed by this rule:

- **Inheritance**. Whilst class inheritance is permitted, *method overriding* is not. This places strong restrictions on control-flow and prevents, for example, "polymorphism loops" where a function indirectly calls itself.
- **Interfaces**. Classes are permitted to implement interfaces as normal.

These restrictions represent something of a compromise for Java. This is because a direct conversion of the original rule would prohibit all dynamic dispatch, *including that arising from interface methods*. However, it is felt that this would be too severe and overly restrict the benefits from encapsulation in Java.

2.9 Rule 9: Code is Free of Warnings

The intention of this rule is to ensure that code is compiled with all warnings enabled, and that no warnings are present. In otherwords, *warnings should be compiled as errors*. In Eclipse, this can be achieved in the "Java Compiler"→"Errors/Warnings" menu. Specifically, all options should be set to "error". The following illustrates how this looks:



NOTE: there are many more options which must be configured here .

JavaDoc Comments Required. In addition, JavaDoc errors must be enabled by configuring the "Java Compiler"→"JavaDoc" menu, and additionally setting all visibility options to "private" and checking "Validate tag arguments". This ensures JavaDoc comments are provided for all **`public`**, **`protected`** and **`private`** declarations, including methods and fields. JavaDoc comments may not be *malformed* (e.g. `@param` tags referring to non-existent parameters, or missing `@return` tags, etc). To illustrate, consider the following example:

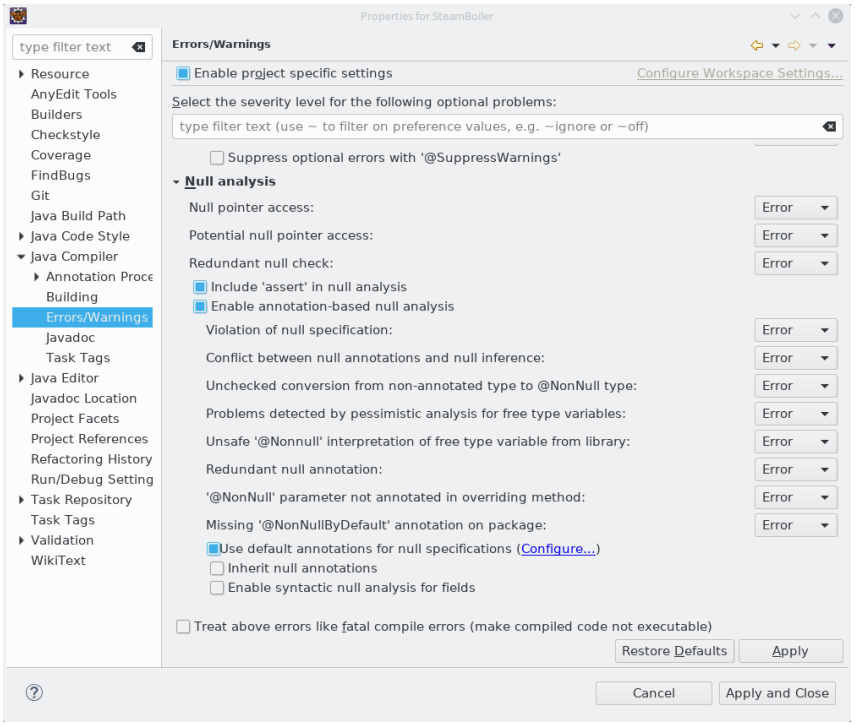
```
/**
 * Return the maximum of two integer values.
 *
 * @param x — First parameter to consider.
 * @param z — Second parameter to consider.
```

```
 */
public int max(int x, int y) {
  if (x > y) { return x; }
  else { return y; }
}
```

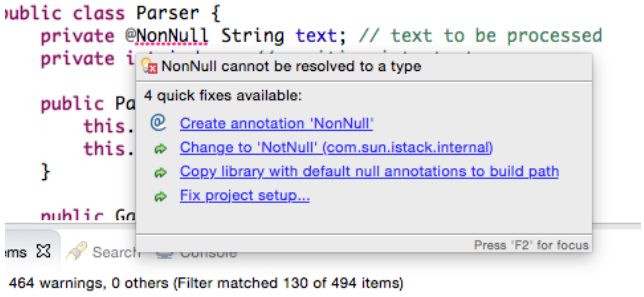
This example does not conform for two reasons. Firstly, parameter `y` does not have a corresponding `@param` tag; Secondly, the `@return` tag is missing.

NonNull Analysis Required. In addition to compiling with all warnings as errors, it's also required that Eclipse's *non-null analysis* is enabled. This is a form of static analysis which helps to ensure `NullPointerException` cannot arise. To enable the NonNull Analysis in Eclipse, proceed as follows:

1. Right-click on the project and select **"Properties→Java Compiler→Errors/Warnings"** and enable "project specific settings".
2. Check **"Include assert in null analysis"** and **"Enable annotation-based null analysis"** and select "Yes" to raising the severity of "Null Pointer Access" problems. Finally, all options should be set to "Error", as illustrated below:

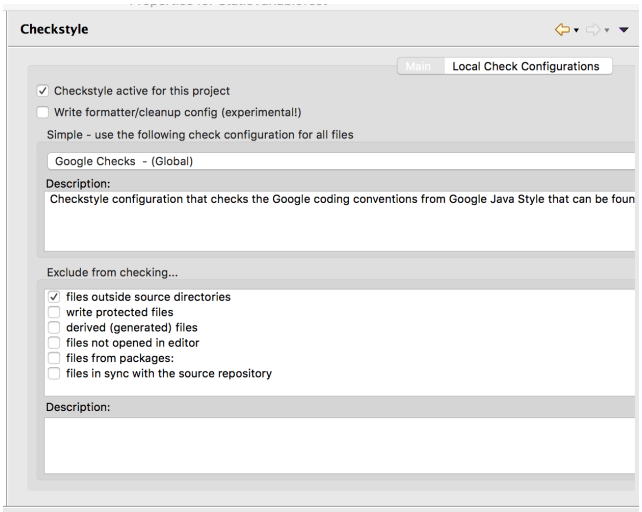


3. Copy library with default annotations. To do this, hover over a `@NonNull` annotation and select **"Copy library with default null annotations to build path"**:



2.10 Rule 10: Checkstyle is Activated

The `checkstyle` plugin must be enabled and configured to enforce the default "Google Checks". These impose a number of restrictions, such as on naming of methods, variables and fields. The following illustrates the checkstyle configuration window:



NOTE: checkstyle must be activated for the project from the *checkstyle* menu.