Coding Standard and Static Analysis

We will be coding in Rust for our project. The coding standard called ‘The Power of Ten” from David J. Pearce will be used for code consistency.

The static analysis tools used will be those made for Rust. Clippy will be used. It needs to be added and run in the terminal.

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

February 25, 2022

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 [2] along with other standards, including MISRA-C [3], the JPL institutional Coding Standard [1] and the (draft) Safety Critical Java standard (JSR302).

An important part of this standard is the emphasis on machine-checkable requirements. Holz mann notes the following [2]:

“The most dooming aspect of many of the guidelines is that they rarely allow for com prehensive 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 [2]. 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 preproces sor 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 mem ory (see Rule 3 below).

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 specifi cation. 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.

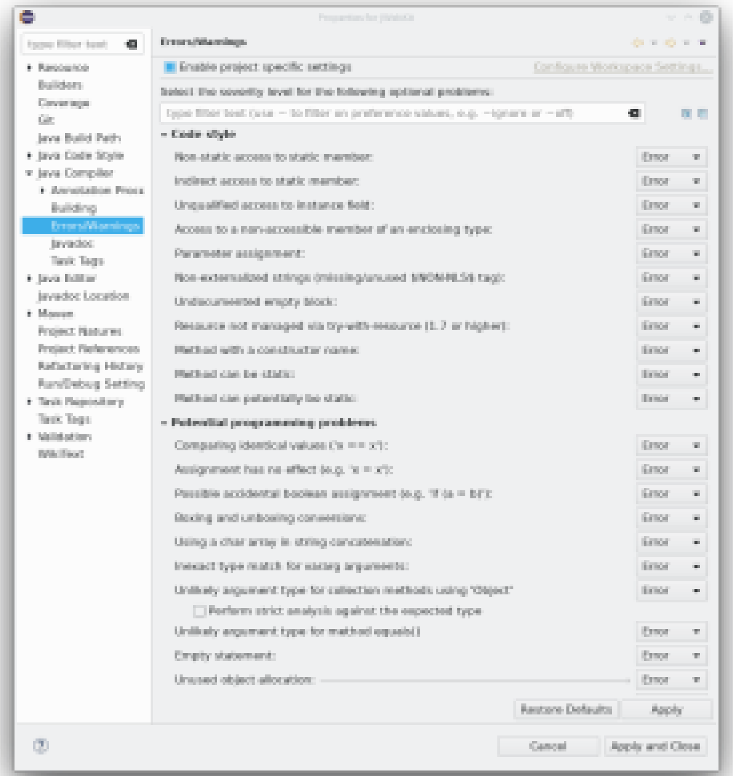
Function Pointers. The corresponding “Power of Ten” rule also prohibits the use of *function pointers*. Whilst these have no direct counterpart in Java they are comparable (in some sense) to dynamic dispatch. To that end, the following restrictions are imposed by this rule:

- 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 con version 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 encap sulation 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:



Errors/Warnings

[✓] Enable project specific settings

Select the severity level for the following optional problems:

Code style

Non-static access to static member:	Error
Indirect access to static member:	Error
Unqualified access to instance field:	Error
Access to a non-accessible member of an enclosing type:	Error
Parameter assignment:	Error
Non-externalized strings (missing/unused $NON-NLS$ tag):	Error
Undocumented empty block:	Error
Resource not managed via try-with-resource (1.7 or higher):	Error
Method with a constructor name:	Error
Method can be static:	Error
Method can potentially be static:	Error

Potential programming problems

Comparing identical values ('x == x'):	Error
Assignment has no effect (e.g. 'x = x'):	Error
Possible accidental boolean assignment (e.g. 'if (a = b)'):	Error
Boxing and unboxing conversions:	Error
Using a char array in string concatenation:	Error
Inexact type match for varargs arguments:	Error
Unlikely argument type for collection methods using 'Object'	Error
[ ] Perform strict analysis against the expected type	
Unlikely argument type for method equals()	Error
Empty statement:	Error
Unused object allocation:	Error

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:

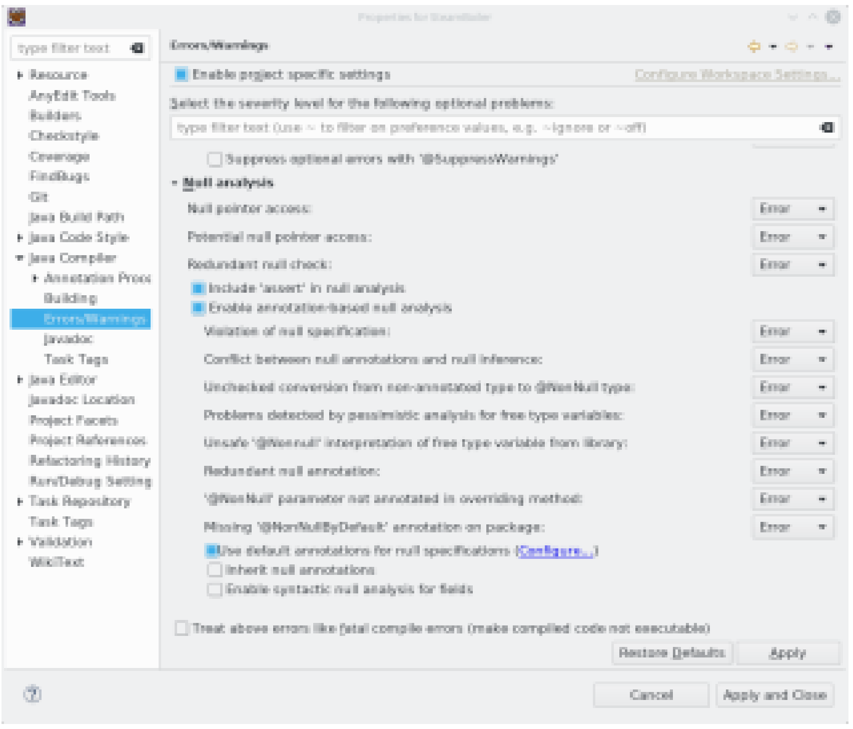
```
/**
 * Return the maximum of two integer values.
 *
 * @param x − First parameter to consider.
 * @param z − Second parameter to consider.
 */
public int max(int x, int y) {
   if (x > y) { return x; }
   else { return y; }
}
```

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

NonNull Analysis Required. In addition to compiling with all warnings as errors, it's also re quired 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:



Errors/Warnings

[✓] Enable project specific settings

Select the severity level for the following optional problems:

type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)

[ ] Suppress optional errors with '@SuppressWarnings'

Null analysis

Null pointer access:	Error
Potential null pointer access:	Error
Redundant null check:	Error
[✓] Include 'assert' in null analysis	
[✓] Enable annotation-based null analysis	
Violation of null specification:	Error
Conflict between null annotations and null inference:	Error
Unchecked conversion from non-annotated type to @NonNull type:	Error
Problems detected by pessimistic analysis for free type variables:	Error
Unsafe '@Nonnull' interpretation of free type variable from library:	Error
Redundant null annotation:	Error
'@NonNull' parameter not annotated in overriding method:	Error
Missing '@NonNullByDefault' annotation on package:	Error

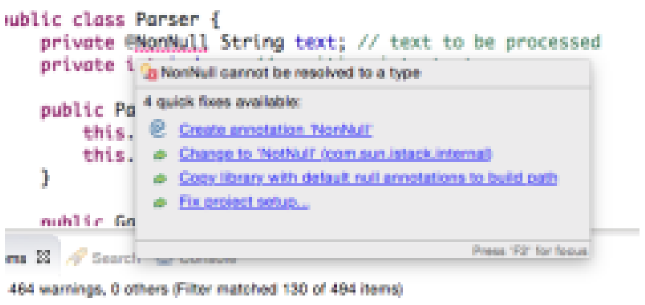
[✓] Use default annotations for null specifications (Configure...)

[ ] Inherit null annotations

[ ] Enable syntactic null analysis for fields

[ ] Treat above errors like fatal compile errors (make compiled code not executable)

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":



```
ublic class Parser {
   private @NonNull String text; // text to be processed
   private i
   public Pa
      this.
      this.
   }
   public Ge
```

NonNull cannot be resolved to a type

4 quick fixes available:

- Create annotation 'NonNull'
- Change to 'NotNull' (com.sun.istack.internal)
- Copy library with default null annotations to build path
- Fix project setup...

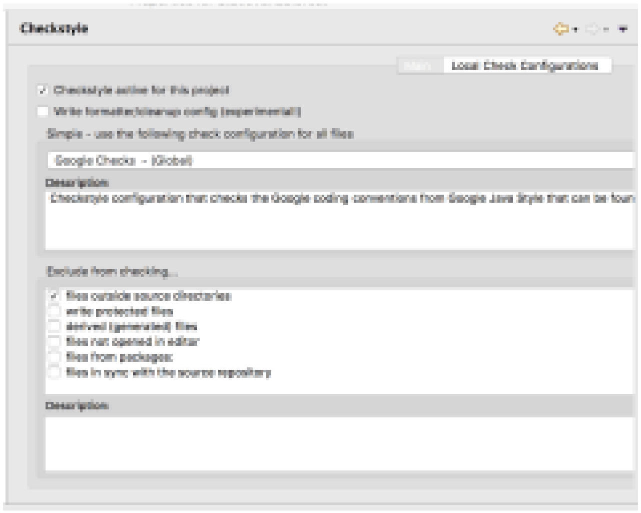
Press 'F2' for focus

464 warnings, 0 others (Filter matched 130 of 494 items)

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:


Checkstyle

Local Check Configurations

- [✓] Checkstyle active for this project
- [ ] Write formatter/cleanup config (experimental)

Simple - use the following check configuration for all files

Google Checks - (Global)

Description:
Checkstyle configuration that checks the Google coding conventions from Google Java Style that can be foun

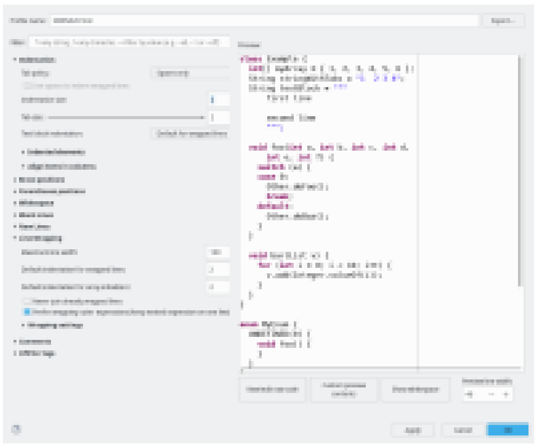
Exclude from checking...

- [✓] files outside source directories
- [ ] write protected files
- [ ] derived (generated) files
- [ ] files not opened in editor
- [ ] files from packages:
- [ ] files in sync with the source repository

Description:

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

HINT: using the Eclipse “formatter” is a very handy way of meeting some of the checkstyle rules (e.g. for indentation and line width). This can be enabled from the “Java Code Style→Formatter” section of the build path configuration:



The formatter can be set *to run automatically* when a file is saved (“Java Code Style→Clean Up”). 7

References

[1] JPL institutional coding standard for the c programming language. Technical report, Jet Propul sion Laboratory, 2009.

[2] Gerard J. Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–97, 2006.

[3] MISRA Consortium. *MISRA-C: 2004 — Guidelines for the use of the C language in critical systems*, 2004.