Coding Standards

Table 1 Document history

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 0.1 |  | ARS T&TT | Initial Draft |
|  |  |  |  |

Contents

[Contents 2](#_Toc59695031)

[1. Introduction 3](#_Toc59695032)

[2. Variable scopes 3](#_Toc59695033)

[3. Class fields and local variables 3](#_Toc59695034)

[4. Method arguments and local variables 5](#_Toc59695035)

[5. Boxing and unboxing 5](#_Toc59695036)

[6. Interfaces 6](#_Toc59695037)

[7. Strings 8](#_Toc59695038)

[8. Naming conventions 9](#_Toc59695039)

[9. Standard Libraries 9](#_Toc59695040)

[10. Immutability 10](#_Toc59695041)

[11. Testing 10](#_Toc59695042)

1. Introduction

This document describes general principles of good programming style and robust design in Java. This is aimed to improve your skills as a Java developer.

1. Variable scopes

In the Java language, every local variable, once declared, has a scope. The variable becomes visible from the place it is declared to the end of the method (or code block) it is declared in. As such, there is only one single rule to follow: declare the local variable as close to the place where it is used as possible. Let us take a look on some typical examples:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for( final Locale locale: Locale.getAvailableLocales() ) {      // Some implementation here  }    try( final InputStream in = new FileInputStream( "file.txt" ) ) {      // Some implementation here  } |

In both code snippets the scope of the local variables is limited to the execution blocks they are declared in. Once the block ends, the local variable goes out of scope and is not visible anymore. It looks clean and concise however with the release of Java 8 and introduction of lambdas, many well-known idioms of using local variables are becoming obsolete. Let us rewrite the for-each loop from the previous example to use lambdas instead:

|  |  |
| --- | --- |
| 1  2  3  4 | Arrays.stream( Locale.getAvailableLocales() ).forEach( ( locale ) -> {          // Some implementation here      }  ); |

The local variable became the argument of the function which itself is passed as the argument of the forEach method.

1. Class fields and local variables

Every method in Java belongs to some class (or some interface in case of Java 8 and the method is declared as default). As such, there is a probability of name conflicts between local variables used in method implementations and class members. The Java compiler is able to pick the right variable from the scope although it might not be the one developer intended to use. The modern Java IDEs do a tremendous work in order to hint to the developers when such conflicts are taking place (warnings, highlightings, …) but it is still better to think about that while developing. Let us take a look on this example:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | public class LocalVariableAndClassMember {       private long value;         public long calculateValue( final long initial ) {           long value = initial;             value \*= 10;           value += value;             return value;       }   } |

The example looks quite simple however there is a catch. The method calculateValue introduces a local variable with name value and by doing that hides the class member with the same name. The line 08 should have summed the class member and the local variable but instead it does something very different. The correct version may look like that (using the keyword this):

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | public class LocalVariableAndClassMember {       private long value;         public long calculateValue( final long initial ) {           long value = initial;             value \*= 10;           value += this.value;             return value;       }   } |

Somewhat naive implementation but nevertheless it highlights the important issue which in some cases could take hours to debug and troubleshoot.

1. Method arguments and local variables

Java allows to reassign non-final method arguments with a different value (however it has no effect on the original value whatsoever). For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public String sanitize( String str ) {      if( !str.isEmpty() ) {          str = str.trim();      }        str = str.toLowerCase();      return str;  } |

It is not a beautiful piece of code but it is good enough to reveal the problem: method argument str is reassigned to another value (and basically is used as a local variable). In all cases (with no exception) this pattern could and should be avoided (for example, by declaring method arguments as final). For example:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | public String sanitize( final String str ) {      String sanitized = str;        if( !str.isEmpty() ) {          sanitized = str.trim();      }        sanitized = sanitized.toLowerCase();      return sanitized;  } |

The code which does follow this simple rule is much easier to follow and reason about, even at the price of introducing local variables.

1. Boxing and unboxing

Boxing and unboxing are all the names of the same technique used in Java language to convert between primitive types (like int, long, double) to respective primitive type wrappers (like Integer, Long, Double).

Although the Java compiler tries to do its best to hide those conversions by performing autoboxing, sometimes it make things worse and leads to unexpected results. Let us take a look on this example:

|  |  |
| --- | --- |
| 1  2  3 | public static void calculate( final long value ) {      // Some implementation here  } |
| 1  2 | final Long value = null;  calculate( value ); |

The code snippet above compiles perfectly fine, however it throws NullPointerException on line 02 when the conversion between Long and long happens. The advice here would be to prefer using primitive type (however as we already know, it is not always possible).

1. Interfaces

Interfaces are not tied to any particular implementation (with default methods being an exception). They are just contracts and as such they provide a lot of freedom and flexibility in the way contracts could be fulfilled. This flexibility becomes increasingly important when the implementation involves external systems or services. Let us take a look on the following simple interface and its possible implementation:

|  |  |
| --- | --- |
| 1  2  3 | public interface TimezoneService {      TimeZone getTimeZone( final double lat, final double lon ) throws IOException;  } |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class TimezoneServiceImpl implements TimezoneService {      @Override      public TimeZone getTimeZone(final double lat, final double lon) throws IOException {          final URL url = new URL( String.format(              "[http://api.geonames.org/timezone?lat=%.2f&lng=%.2f&username=demo](http://api.geonames.org/timezone?lat=%25.2f&lng=%25.2f&username=demo)",              lat, lon          ) );          final HttpURLConnection connection = ( HttpURLConnection )url.openConnection();            connection.setRequestMethod( "GET" );          connection.setConnectTimeout( 1000 );          connection.setReadTimeout( 1000 );          connection.connect();            int status = connection.getResponseCode();          if (status == 200) {              // Do something here          }            return TimeZone.getDefault();      }  } |

The code snippet above demonstrates a typical interface / implementation pattern. The implementation uses external HTTP service (<http://api.geonames.org/>) to retrieve the time zone for a particular location. However, because the contact is driven by the interface, it is very easy to introduce yet another implementation using, for example, database or even flat file. With that, interfaces greatly help to design testable code. For example, it is not always practical to call external service on each test run so it makes sense to provide the alternative, dummy implementation (also known as stub or mock) instead:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class TimezoneServiceTestImpl implements TimezoneService {      @Override      public TimeZone getTimeZone(final double lat, final double lon) throws IOException {          return TimeZone.getDefault();      }  } |

This implementation could be used in every place where TimezoneService interface is required, isolating the test scenario from dependency on external components.

Many excellent examples of appropriate use of the interfaces are encapsulated inside the Java standard collection library. Collection, List, Set, all those interfaces are backed by several implementations which could be replaced seamlessly and interchangeably when contracts are favored, for example:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static< T > void print( final Collection< T > collection ) {      for( final T element: collection ) {          System.out.println( element );      }  } |
| 1  2  3  4 | print( new HashSet< Object >( /\* ... \*/ ) );  print( new ArrayList< Integer >( /\* ... \*/ ) );  print( new TreeSet< String >( /\* ... \*/ ) );  print( new Vector< Long >( /\* ... \*/ ) ); |

1. Strings

Strings are one of the most widely used types in Java and, arguably, in most of the programming languages. The Java language simplifies a lot the routine operations over strings by natively supporting the concatenations and comparison. Additionally, the Java standard library provides many different classes to make strings operations efficient and that is what we are going to discuss in this section.

In Java, strings are immutable objects, represented in UTF-16 format. Every time you concatenate the strings (or perform any operation which modifies the original string) the new instance of the String class is created. Because of this fact, the concatenation operations may become very ineffective, causing the creation of many intermediate string instances (generally speaking, generating garbage).

But the Java standard library provides two very helpful classes which aim to facilitate string manipulations: StringBuilder and StringBuffer (the only difference between those is that StringBuffer is thread-safe while StringBuilder is not). Let us take a look on couple of examples using one of these classes:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | final StringBuilder sb = new StringBuilder();    for( int i = 1; i <= 10; ++i ) {      sb.append( " " );      sb.append( i );  }    sb.deleteCharAt( 0 );  sb.insert( 0, "[" );  sb.replace( sb.length() - 3, sb.length(), "]" ); |

Though using the StringBuilder / StringBuffer is the recommended way to manipulate strings, it may look over-killing in simple scenario of concatenating two or three strings so the regular + operator could be used instead, for example:

|  |  |
| --- | --- |
| 1 | String userId = "user:" + new Random().nextInt( 100 ); |

The often better alternative to straightforward concatenation is to use string formatting and Java standard library is also here to help by providing the static helper method String.format. It supports a rich set of format specifiers, including numbers, characters, dates/times, etc. (for the complete reference please visit the [official documentation](https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html#syntax)). Let us explore the power of formatting by example:

|  |  |
| --- | --- |
| 1  2  3  4  5 | String.format( "%04d", 1 );                      -> 0001  String.format( "%.2f", 12.324234d );             -> 12.32  String.format( "%tR", new Date() );              -> 21:11  String.format( "%tF", new Date() );              -> 2014-11-11  String.format( "%d%%", 12 );                     -> 12% |

The String.format method provides a clean and easy approach to construct strings from different data types. It worth to mention that some modern Java IDEs are able to analyze the format specification against the arguments passed to the String.format method and warn developers in case of any mismatches detected.

1. Naming conventions

Java as a language does not force the developers to strictly follow any naming conventions, however the community has developed a set of easy to follow rules which make the Java code looking uniformly across standard library and any other Java project in the wild.

* **package names** are typed in **lower case**: org.junit, com.fasterxml.jackson, javax.json
* **class, enum, interface** or **annotation names** are typed in **capitalized case**: StringBuilder, Runnable, @Override
* **method** or **field names** (except static final) are typed in **camel case**: isEmpty, format, addAll
* **static final field** or **enumeration constant names** are typed in **upper case**, s**eparated by underscore** ‘\_’: LOG, MIN\_RADIX, INSTANCE
* **local variable** and **method arguments names** are typed in **camel case**: str, newLength, minimumCapacity
* **generic type parameter names** are usually represented as **one character in upper case**: T, U, E

By following these simple conventions the code you are writing will look concise and indistinguishable from any other library or framework, giving the impression it was authored by the same person (one of those rare cases when conventions are really working).

1. Standard Libraries

No matter what kind of Java projects you are working on, the Java standard libraries are your best friends. Yes, it is hard to disagree that they have some rough edges and strange design decisions, nevertheless in 99% it is a high quality code written by experts. It is worth learning.

Every Java release brings a lot of new features to existing libraries (with some possible deprecation of the old ones) as well as adds many new libraries. Java 5 brought new concurrency library reconciled under java.util.concurrent package. Java 6 delivered (a bit less known) scripting support (javax.script package) and the Java compiler API (under javax.tools package). Java 7 brought a lot of improvements into java.util.concurrent, introduced new I/O library under java.nio.file package and dynamic languages support with java.lang.invoke package. And finally, Java 8 delivered a long-awaited date/time API hosted under java.time package.

Java as a platform is evolving and it is extremely important to keep up with this evolution. Whenever you are considering to introduce third-party library or framework into your project, make sure the required functionality is not already present in the Java standard libraries (indeed, there are many specialized and high-performance algorithm implementations which outperforms the ones from standard libraries but in most of the cases you do not really need them).

1. Immutability

If a class you are designing or a method you are implementing could provide the immutability guarantee, it could be used mostly everywhere without the fear of concurrent modifications. It will make your life as a developer easier (and hopefully lives of your teammates as well).

1. Testing

Test-driven development (TDD) practices are extremely popular in Java community, raising the quality bar of the code being written. With all those benefits which TDD brings on the table, it is sad to observe that Java standard library does not include any test framework or scaffolding as of today.

Nevertheless, testing becomes a necessary part of the modern Java development and in this section we are going to cover some basics using excellent [JUnit](http://junit.org/) framework. Essentially, in [JUnit](http://junit.org/), each test is a set of assertions about the expect object state or behavior.

The secret of writing great tests is to keep them short and simple, testing one thing at a time. As an exercise, let us write a set of tests to verify that String.format function from the section [Strings](https://www.javacodegeeks.com/2015/09/general-programming-guidelines.html#string) returns the desired results.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20 | package com.javacodegeeks.advanced.generic;    import static org.junit.Assert.assertThat;  import static org.hamcrest.CoreMatchers.equalTo;    import org.junit.Test;    public class StringFormatTestCase {      @Test      public void testNumberFormattingWithLeadingZeros() {          final String formatted = String.format( "%04d", 1 );          assertThat( formatted, equalTo( "0001" ) );      }        @Test      public void testDoubleFormattingWithTwoDecimalPoints() {          final String formatted = String.format( "%.2f", 12.324234d );          assertThat( formatted, equalTo( "12.32" ) );      }  } |

The tests look very readable and their execution is instantiations. Nowadays, the average Java project contains hundreds of test cases, giving the developer a quick feedback on regressions or features under development.