# Automatically Discovering Common Java Code Edits in Github Repositories

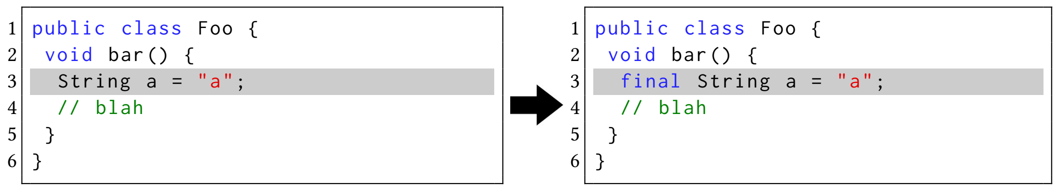
Catalog

## 1 CATALOG

In this section, we present a catalog of anomalies that developers perform in Java. For each anomaly, we include name, description, benefits, code example, and tools that include the anomaly in their catalog (if any). Each anomaly could be associated to one or more code metrics such as bug fixing, clarity, performance.

### 1.1 Field, Parameter, Local Variable Could be Final

Besides classes and methods, developers can use the `final` modifier in fields, parameters, and local variables. The semantic differs for each one of these usages. A final class cannot be extended, a final method cannot be overridden, and final fields, parameters, and local variables cannot change their value. Thus, a final modifier guarantees that fields, parameters, and local variables cannot be re-assigned. A re-assignment generates an error at compile-time. Final modifier improves clarity, helps developers to debug the code showing constructors that change state and are more likely to break the code. In addition, final modifier allows the compiler and virtual machine to optimize the code. This anomaly is included in tools such as PMD. IDEs such as Eclipse and Netbeans can be configured to add final modifiers to fields, parameters, and local variables automatically on saving. Fig. 1 shows a code example of adding the final modifier to a parameter. Variable `a` is assigned a single time. Thus, it can be declared final such as variable `b`.



```java
public class Foo {
 void bar() {
  String a = "a";
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  final String a = "a";
  // blah
 }
}
```

(b) (after)

Fig. 1. **Java Code Usage**: Field, Parameter, Local Variable Could be Final

### 1.2 Allows Type Inference for Generic Instance Creation

Since Java 7, developers can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) [6]. The empty set of type parameters, also known as diamond, allows the compiler to infer type arguments from the context. By using diamond construction, developers make clear the use of generic instead of the deprecated raw types, the version of a generic type without type arguments. Java allows raw types only to ensure compatibility with pre-generics code. The benefit of the diamond constructor, in this context, is clarity since it is more concise. Fig. 2 shows the use of the diamond operator in a variable declaration. Instead of using the type parameter `<String>`, developers can use the diamond to invoke the constructor of `List` generic class.

Author's address:

```java
public class Foo {
 void bar() {
  List<String> a = new
   ArrayList<String>();
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  List<String> a = new
   ArrayList<>();
  // blah
 }
}
```

(b) (after)

Fig. 2. **Java Code Usage**: Allow Type Inference for Generic Instance Creation

## 1.3 Remove Raw Type

Java discourages the use raw types. A raw type denotes a generic type without type arguments, which was used in the out-dated version of Java and is allowed to ensure compatibility with pre-generics code. Since type arguments of raw types are unchecked, they can cause errors at run-time. Java compiler generates warning to indicate the use of raw types into the source code. Fig. 3 shows the use of a raw type. Developers can pass any type of collection to the constructor of a raw type since it is unchecked.

```java
public class Foo {
 void bar() {
  List<String> a = new
   ArrayList();
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  List<String> a = new
   ArrayList<>();
  // blah
 }
}
```

(b) (after)

Fig. 3. **Java Code Usage**: Remove Raw Type

## 1.4 Prefer Class<?>

Java prefers `Class<?>` over plain `Class` although these constructions are equivalent [2]. The benefit of `Class<?>` is clarity since developers explicitly indicates that they are aware of not using an out-dated Java construction. The Java compiler generates warning on the use of `Class`. Fig. 4 exemplifies the use of `Class<?>`.

```java
public class Foo {
 void bar() {
  Class a = String.class;
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  Class<?> a = String.class;
  // blah
 }
}
```

(b) (after)

Fig. 4. **Java Code Usage**: Remove Raw Type

## 1.5 Use Variadic Functions

Variadic functions denote functions that use variable-length arguments (varargs). This feature was introduced in Java 5 to indicate that the method receives zero or more arguments. Prior to Java 5, if a method receives a variable number of arguments, developers have to create overload method for each number of arguments or to pass an array of arguments to the method. The benefit of using varargs is simplicity since developers do not need to create overload methods and use the same notation independently of the number of arguments. For the compiler perspective, the method receives an array as parameter. Fig. 5 shows the use of the Variadic functions.

```java
public class Foo {
 void bar(String [] s) {
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar(String ... s) {
  // blah
 }
}
```

(b) (after)

Fig. 5. **Java Code Usage**: Remove Raw Type

## 1.6 String to Character

In Java, we can represent a character both as a string or a character. For some operations such as concatenating or appending a value to a `StringBuffer`/`StringBuilder` it is better to represent the value as a character if the value itself is a character. Representing the value as a character improves performance. For instance, this edit improves from 10-25% the performance at the Guava project[1]. This transformation is included in the catalog of anomalies of tools such as PMD. Fig. 6 shows the use of the code usage **string to character**.

```java
public class Foo {
 void bar() {
  StringBuffer sb =
   new StringBuffer();
  sb.append("a");
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  StringBuffer sb =
   new StringBuffer();
  sb.append('a');
  // blah
 }
}
```

(b) (after)

Fig. 6. **Java Code Usage**: String to Character

## 1.7 `StringBuffer` to `StringBuilder`

`StringBuffer` and `StringBuilder` denote a mutable sequence of characters. These two types are compatible, but `StringBuilder` provides no guarantee of synchronizations. Since synchronization is rarely used, `StringBuilder` offers right performance over its counterpart. If developers want to synchronize a StringBuilder, they can surround the code block with a synchronized operator `synchronized(sb){}`. This class is designed to replace `StringBuffer` in places where

[1]https://github.com/google/guava/commit/8f48177132547cee2943c93837d76b898154d722

StringBuffer was being used by a single thread [5]. Java recommends the use of StringBuilder in preference to StringBuffer due to performance. Fig. ?? shows an example of the use of the StringBuilder class.

```java
public class Foo {
 void bar() {
  StringBuffer a =
   new StringBuffer();
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  StringBuilder a =
   new StringBuilder();
  // blah
 }
}
```

(b) (after)

Fig. 7. **Java Code Usage**: StringBuffer to StringBuilder

## 1.8 Use Collection isEmpty

The use of isEmpty is encouraged to verify whether the list contains no elements instead of verifying the size of a collection. Although in the majority of collections, these two constructions are equivalent, for other collections computing the size of an arbitrary list could be expensive. For instance, in the class ConcurrentSkipListSet, the size method is not a constant-time operation [4]. This transformation is included in the catalog of anomalies of tools such as PMD. Fig. 8 shows an example of use of the isEmpty method.

```java
public class Foo {
 void bar() {
  List foo = getList();
  if (foo.size() == 0) {
   // blah
  }
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  List foo = getList();
  if (foo.isEmpty()) {
   // blah
  }
 }
}
```

(b) (after)

Fig. 8. **Java Code Usage**: Use Collection isEmpty

## 1.9 Prefer String Literal equals Method

The equals method is widely used in software development. Some usages can cause NullPointerException due to the right-hand side of the equals method object reference being null. When using the equals method to compare some variable to a String Literal, developers could overcome null point errors by allowing the string literal to call the equals method because a string literal is never null. Since Java string literal equals method checks for null, we do not need to check for null explicitly when calling the equals method of a string literal. Fig. 9 shows an example of use of the code usage pattern **prefer string literal equals method**.

```java
public class Foo {
 void bar(String s) {
  if (s.equals("str")) {
   // blah
  }
 }
}
```

(a) (before)

```java
public class Foo {
 void bar(String s) {
  if ("str".equals(s)) {
   // blah
  }
 }
}
```

(b) (after)

Fig. 9. **Java Code Usage**: Prefer String Literal equals Method

### 1.10 Prefer String Constant equals Method

The equals method is widely used in software development. Some usages can cause `NullPointerException` due to the right-hand side of the equals method object reference being null. When using the `equals` method to compare some variable to a string constant, developers could overcome null point errors by allowing the string constant to call the `equals` method because a constant is rarely null. Since Java string literal equals method checks for null, we do not need to check for null explicitly when calling the equals method of a string literal. Fig. 10 shows an example of use of the code usage pattern **prefer string constant equals method**.

```java
public class Foo {
 void bar(String s) {
  if (s.equals(CONST)) {
   // blah
  }
 }
}
```

(a) (before)

```java
public class Foo {
 void bar(String s) {
  if (CONST.equals(s)) {
   // blah
  }
 }
}
```

(b) (after)

Fig. 10. **Java Code Usage**: Prefer String Literal equals Method

### 1.11 Use `valueOf` instead Wrapper Constructor

Java allows to use the `valueOf` method or the constructor to create wrapper objects of primitive types. Java recommends the use of valueOf for performance purpose since `valueOf` method cached some values. This checker is included in the catalog of anomalies of tools such as Sonar. Fig. 11 shows an example of the use of the `valueOf` method.

### 1.12 Avoid using `FileInputStream`/`FileOutputStream`

`FileInputStream` and `FileOutputStream` override `finalize()`. As a result, objects of these classes go to a category that is cleaned only when a full clearing is performed by the Garbage Collector [1]. Since Java 7, developers can use `Files.new` counterpart to improve program performance. This anomaly is described as a bug by Java JDK [3]. Fig. 12 shows an example of use of the code usage pattern **avoid using `FileInputStream`/`FileOutputStream`**.

```java
public class Foo {
 void bar() {
  Integer a =
   new Integer(1);
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  Integer a =
   Integer.valueOf(1);
  // blah
 }
}
```

(b) (after)

Fig. 11. **Java Code Usage**: Use `valueOf` instead Wrapper Constructor

```java
public class Foo {
 void bar(String p,
  byte[] content)
   throws IOException {
  FileOutputStream os =
  new FileOutputStream(p);
  os.write(content);
  os.close();
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar(String p,
  byte[] content)
   throws IOException {
  OutputStream os=Files.
   newOutputStream(Paths
    .get(p))
  os.write(content);
  os.close();
  // blah
 }
}
```

(b) (after)

Fig. 12. **Java Code Usage**: Avoid using FileInputStream/FileOutputStream

## 1.13 Do not use strings to represent paths

Although strings were not designed for this purpose, developers could use them to represent paths. In Java, some classes are specifically designed to represent paths such as `java.nio.Path`. The use of these classes could prevent two problems: first, strings can be combined in an undisciplined way, which can lead to invalid paths. The second problem comes from the use of separators. Some operational systems (OS) use the separator "\" different from other OS, which could causes bugs in the code. Fig. 13 shows an example of use of the code usage pattern **do not use strings to represent paths**.

```java
public class Foo {
 private String path;
 // blah
}
```

(a) (before)

```java
public class Foo {
 private Path path;
 // blah
}
```

(b) (after)

Fig. 13. **Java Code Usage**: Use Path to represent file path

## 1.14 Add @Nullable for parameters that accept null

In Java, developers could annotate a parameter with @Nullable to indicate that the method, as well as all its overrides, accepts null for that parameter. This annotation helps code analyzer tools such as FindBugs and Checker Framework to better analyze the code. Fig. 14 shows an example of the use of @Nullable annotation.

```java
public class Foo {
 void bar(String a) {
   // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar(@Nullable String a) {
   // blah
 }
}
```

(b) (after)

Fig. 14. **Java Code Usage**: Add @Nullable for parameters that accept null

## 1.15 Use Primitive Type Termination in literals

In Java, all non-float point literals are evaluated to `int` and all float point literals are evaluated to `float`. This behaviour could lead to numeric errors. For instance, consider this example took from a development forum (Fig 15). In this example, a receives the value `-1846840301` and b receives the value `370370362962963` although they intent to represent the same value. Fig. 13 shows an example of use of the code usage pattern **use primitive type termination in literals**.

```java
long a = 33333333 * 11111111; // overflows
long b = 33333333L * 11111111L;
System.out.println("a= "+new BigDecimal(a));
System.out.println("b= "+new BigDecimal(b));
System.out.println("a == b is " + (a == b));
```

Fig. 15. An example of a numeric problem associated with numeric literal use.

```java
public class Foo {
 void bar() {
   long a = 33333333;
   // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
   long a = 33333333L;
   // blah
 }
}
```

(b) (after)

Fig. 16. **Java Code Usage**: Use Primitive Type Termination in literals

### 1.16 Promote Inner Class

Inner classes are an important feature of Java. As a design principle classes should be declared final. If developers want to re-use an inner class, it may be the case to promote the inner class to a regular class. Fig. 17 shows an example of use of the code usage pattern **promote inner class**.

```java
//file Foo.java
public class Foo {
 public class Bar {
  // blah
 }
 // blah
}
```

(a) (before)

```java
//file Foo.java
public class Foo {
 // blah
}
//file Bar.java
public class Bar {
  // blah
 }
```

(b) (after)

Fig. 17. **Java Code Usage**: Promote Inner Class

### 1.17 Cannot Use Casts or instanceof with Parameterized Types

Java compiler erases all type parameters in generic code. Therefore, it is not possible to verify for a generic type which parameterized type is being used at run-time. Trying to do so, will generate a compiler-time error. To perform this verification, Java recommends the use of an unbounded wildcard <?> to verify the generic type. Fig. 18 shows an example of use of the code usage pattern **cannot use casts or instanceof with parameterized types**.

```java
public class Foo {
 void bar() {
  List<String> foo = getList();
  if (foo instanceof List) {
   // blah
  }
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  List<String> foo = getList();
  if (foo instanceof List<?>) {
   // blah
  }
 }
}
```

(b) (after)

Fig. 18. **Java Code Usage**: Use Collection `isEmpty`

### 1.18 Use StopWatch

Java developers often measure time using `System.nanoTime()`. An alternative approach for this task is to use a `StopWatch`. The latter has some benefits: developers can substitute the source that measures time to improve performance. Second, the result of a `StopWatch` is more interpretable than the `System.nanoTime`, which meaning can only be understood when compared to another call to `System.nanoTime`.[2] Fig. 19 shows an example of use of the code usage pattern **use StopWatch**.

[2]https://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/Stopwatch.html

```java
public class Foo {
 void bar() {
  long startTime =
   System.nanoTime();
  doSomething();
  long endTime =
   System.nanoTime();
  long totalTime =
   endTime-startTime;
  log.info("time: " + totalTime);
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  Stopwatch stopwatch =
   Stopwatch.createStarted();
  doSomething();
  long millis =
   stopwatch.elapsed(MILLISECONDS);
  log.info("time: " + stopwatch);
 }
}
```

(b) (after)

Fig. 19. **Java Code Usage**: Use StopWatch

## 1.19 Use `HH` in SimpleDateFormat

We could configure a `SimpleDataFormat` in different ways. Some configurations may lead to bugs. For instance, it causes a bug in the Ant project.[3] In some module of this project, `SimpleDataFormat` uses the 12h-format (`hh`). As a result, sometimes the chronological order of commits is broken since a commit at 14:20 (becomes 02:20), which is before a commit at 07:00. Change the SimpleDataFormat instance from `hh:mm` to 24h-format (HH:mm) fix the bug. Fig. 20 shows an example of use of the code usage pattern **use `HH` in SimpleDateFormat**.

```java
public class Foo {
 void bar() {
  SimpleDateFormat a =
   new SimpleDateFormat
    ("yyyy/MM/dd hh:mm:ss")
  // blah
 }
}
```

(a) (before)

```java
public class Foo {
 void bar() {
  SimpleDateFormat a =
   new SimpleDateFormat
    ("yyyy/MM/dd HH:mm:ss")
  // blah
 }
}
```

(b) (after)

Fig. 20. **Java Code Usage**: Use HH in SimpleDateFormat

[3]https://bz.apache.org/bugzilla/show_bug.cgi?id=11582

## REFERENCES

[1] DZone. 2017. FileInputStream/FileOutputStream Considered Harmful. (2017). At https://dzone.com/articles/fileinputstream-fileoutputstream-considered-harmful. Accessed in 2017, December 19.

[2] Bruce Eckel. 2005. *Thinking in Java (4th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[3] Java JDK. 2017. Relax FileInputStream/FileOutputStream requirement to use finalize. (2017). At https://bugs.openjdk.java.net/browse/JDK-8187325. Accessed in 2017, December 19.

[4] Oracle. 2017. Class ConcurrentSkipListSet<E>. (2017). At https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html. Accessed in 2017, December 19.

[5] Oracle. 2017. Class StringBuilder. (2017). At https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html. Accessed in 2017, December 19.

[6] Oracle. 2017. Type Inference for Generic Instance Creation. (2017). At https://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html. Accessed in 2017, December 19.

Received February 2007; revised March 2009; accepted June 2009