100 Java Mistakes and How to Avoid Them

Tagir Valeev

MANNING

MEAP Edition
Manning Early Access Program

100 Java Mistakes and How to Avoid Them

Version 3

welcome

Thank you for purchasing the MEAP of *100 Java Mistakes and How to Avoid Them*. You probably already have some experience with Java and want to enhance your knowledge.

I have been programming for 30 years, including 13 years of commercial programming in Java. One thing that I constantly encounter in my daily job is bugs. I introduce bugs myself or discover bugs introduced by other programmers. When I see and fix one I always think: is it possible to avoid similar bugs in the future? How can I help myself and other programmers on my team to not run into this particular problem again?

Nine years ago, I encountered the Java static analyzer FindBugs. This was really an astonishing discovery for me. I launched it against my project, and it found many problems, both very stupid and very tricky ones. I learned a lot about Java simply by reading FindBugs reports.

Soon I started to notice the limitations of FindBugs and I felt that I could improve it. So I started to contribute there, on small things at first, and then on whole new detectors to find new types of bugs. Unfortunately, the project was not actively maintained and had architectural problems, so it was hard to implement all the ideas I had. For some time, I tried to write my own static analyzer, but later I took the opportunity to join the IntelliJ IDEA team.

Now, hunting for bugs and teaching the static analyzer to recognize them is my daily job, and I'm very happy with it. Still, I learned that even the best static analyzers have limitations and cannot replace a programmer's mind. I've seen many mistakes both in my projects and in other projects, many of which we will analyze in the book, and I noticed that the same problems repeat again and again and could be avoided if programmers knew about them in advance. So I decided to summarize my experience and write this book.

The book reveals the pitfalls that you may encounter in the Java language itself and in the most used library methods. For each mistake, I try to formulate ways to avoid or quickly detect it. You'll be able to train your eye and not only avoid mistakes by yourself but easily spot suspicious patterns in your colleagues' code. I hope you'll enjoy reading, learn many interesting things about Java and discover how to make static analyzers your friend.

Please let me know your thoughts in the liveBook Discussion forum.

—Tagir Valeev

brief contents

1 Managing code quality

This chapter covers

- **The purpose and the structure of this book**
- **Various techniques to improve the code quality**
- **Advantages and disadvantages of static analysis**
- **Approaches to make static analysis more useful**
- **Testing and assertions**

Every software developer introduces bugs to the code. There’s no way to avoid them completely. Some of the bugs cause very subtle change in program behavior and nobody cares about it. Other bugs may cost millions of dollars, lost spacecraft, or even human lives. Many of them are in the middle: they don’t have disastrous effects but they annoy users and require hours of debugging and fixing.

Some bugs are caused by misunderstanding of the specification requirements, miscommunication between the software development team and the customer, and so on. Another category of bugs comes from the miscommunication between the developer and the machine. That is: the developer correctly understands the problem but writes the wrong code, so the machine solves the problem incorrectly. This book is devoted to the latter category.

Some bugs are complex and unique. For example, one module may fail to handle a rare corner case, and another one may rely on that corner case. Everything was fine until these modules started talking to each other. In rare cases, three or more independent components are involved, and the problem appears only when all of them are connected to each other. Investigation of such problems can be as exciting as a good detective story.

However, there are also repetitive bugs that are produced by many programmers in many programs. Experienced developers have seen a number of such bugs in their practice, so they know in advance what kind of code is dangerous and requires special care. They can spot the erroneous code patterns just because they already saw something similar in the past. Less experienced developers overlook repetitive bugs more often, so the bug has more chance to slip into production and cause serious consequences.

The purpose of this book is to summarize this experience. I list the common mistakes that appear in Java programs. For every mistake discussed in the book, I show a code example, explain why this problem usually happens and advise on how to avoid the problem.

This book doesn't speak about mistakes that cause a compilation error. Such mistakes also often occur, of course, but there's not much to say about them other than to read the Java compiler output and correct them. Luckily, the Java compiler is quite strict and many things that are acceptable in other programming languages cause compilation errors in Java. This includes many instances of unreachable code, use of an uninitialized local variable, impossible type cases, and so on. Thanks to the compiler, I can omit these cases from the book and concentrate on the problems that will actually survive compilation and slip into your production code.

I have seen most of the mistakes listed in this book in real production code bases. More often though I provide synthetic code samples based on real code, as this allows us to concentrate on the described bug without going deep into the details of a particular project. I try to keep code samples as short as possible. In most of the cases I omit the class declaration or even the method declaration leaving only the method body. I also tend to use more compact code formatting than it's usually used in real world Java programming. For example, we may omit the `@Override` annotation on the overridden method, despite the fact that it's recommended. Don't consider our code samples as code style recommendations, unless it's explicitly stated.

1.1 The structure of this book

The book is organized into chapters. This chapter explains common approaches used in software engineering to manage code quality and avoid bugs. The subsequent chapters cover various individual mistakes, which are grouped by category. The mistake sections are mostly independent, so feel free to skip something if you already know the given pattern or feel that it's not applicable to your daily job.

Chapter 2 shows the mistakes inside individual expressions, such as problems with precedence, mixing one operator with another, or pitfalls with variable-arity method calls. Chapter 3 concentrates on mistakes related to the structural elements of a Java program. This includes problems with statements like for-loops, as well as higher-level structure issues like circular initialization of superclass and subclass.

Chapter 4 covers issues that happen when you work with numbers in Java, including the infamous problem with numeric overflow. Chapter 5 concentrates on several of the most common exceptions that happen in Java like `NullPointerException` or `ClassCastException`. Chapter 6 tells about the caveats of string processing.

Chapter 7 is devoted to comparing objects and speaks mostly about using and implementing methods like `equals()`, `hashCode()` and `compareTo()`. Chapter 8 concentrates on mistakes that happen when you use collections and maps. Chapter 9 covers some easy-to-misuse library methods not covered in previous chapters. Finally, Chapter 10 describes mistakes that may happen when writing the unit tests.

The appendix briefly describes how to enhance some static analysis tools to catch the problems specific to your project. It’s optional to read, but if you are ready to introduce custom static analysis rules to your project, it may serve as a starter guide.

I believe that this book is most useful for middle-level software developers who already know the Java language but may have not enough practical programming experience. Probably some bug patterns described in the book are not known to senior software developers as well. Less experienced developers or even advanced students might also find this book interesting.

Before we start speaking about individual mistakes, it’s reasonable to discuss common approaches to avoiding bugs. These approaches include code review, pair programming, static analysis, and various kinds of testing including unit-testing, property testing, integration testing, smoke testing, and so on.

1.2 Code review and pair programming

The code review technique assumes that the changes committed by developers are reviewed by their peers. It’s quite possible that a second look by another person with different experience and background will allow a bug that was overlooked by an original author to be caught. Code reviews are useful for more than simply catching bugs. They may also help to improve the code architecture, readability, performance, and so on. Also, code review improves knowledge sharing and mentoring inside the organization. There are books, conference talks and blog posts on how to make code reviews efficient. For example, you may read [Reference] “What to Look for in a Code Review” by Trisha Gee.

Pair programming is a technique that emerged from the extreme programming methodology. It’s an extreme version of code review in which two programmers work together, either at one workstation or using a real-time collaborative code editing tool (remote pair programming). In this case, the code is reviewed at the same time as it’s written. Compared to code review, pair programming might be more efficient but it requires more discipline. Again, there are many resources on how to use pair programming in practice. E. g., [Reference] ”Practical Remote Pair Programming: Best practices, tips, and techniques for collaborating productively with distributed development teams” by Adrian Bolboacă.

1.3 Code style

There are many ways to write a program in Java to do the same thing. You can use conditional expressions instead of `if`-statements, replace `while` loops with `for` loops, wrap statement bodies into braces or omit them, consistently mark local variables as `final`, use different indentation styles, and so on. While many such decisions are a matter of taste, some of them will definitely reduce the number of bugs in the code.

For example, to specify that a number has the `long` type, it’s possible to use either lowercase ‘l’ or uppercase ‘L’ suffix, like `10l` or `10L`. However, it’s generally agreed that uppercase ‘L’ is better, because it’s possible to accidentally mix lowercase ‘l’ with digit ‘1’.

Another example is braces around the `if`-statement body that contains only one statement. The Java language allows to omit them:

```
if (a < b)
  System.out.println("a is smaller!");
```

However, this may cause unpleasant mistake when a developer wants to add another line of code under the condition:

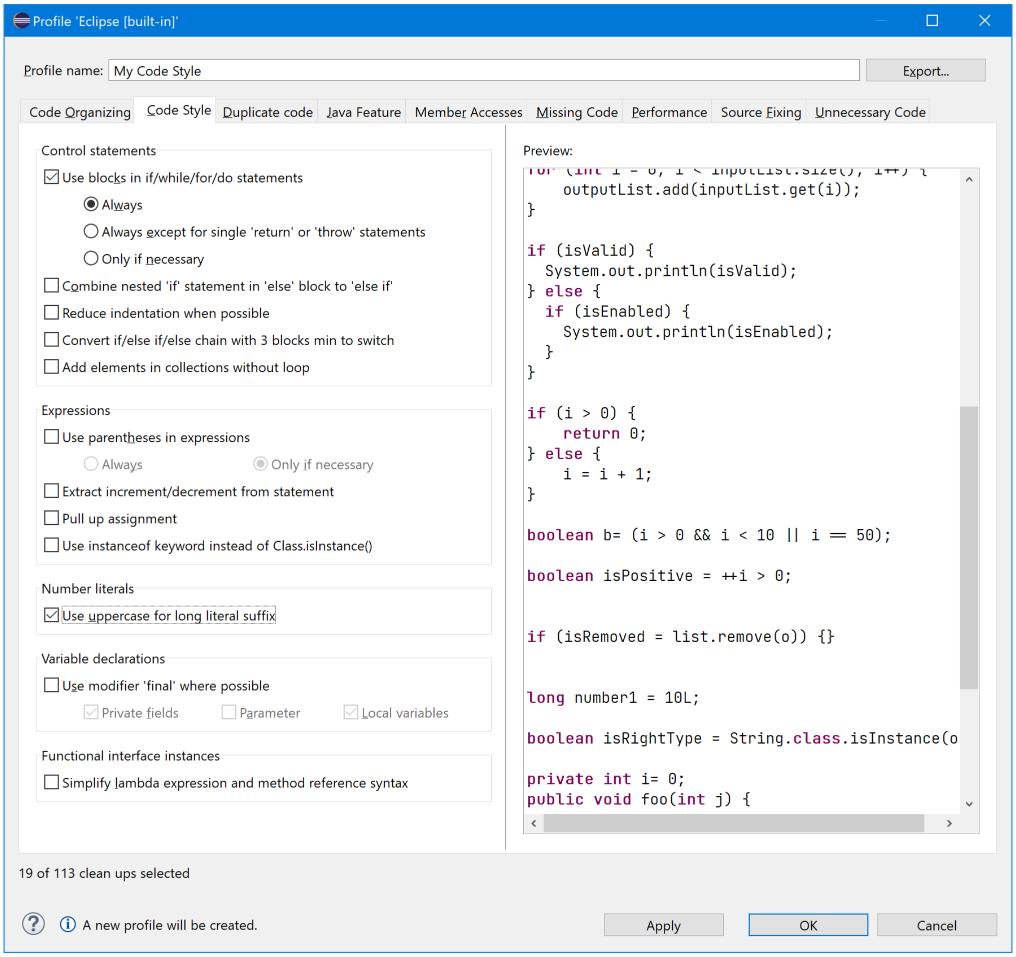
```
if (a < b)
  System.out.println("a is smaller!");
  System.out.println("and b is bigger");
```

In this case, the last line will be erroneously executed unconditionally. To avoid such a problem, many projects and development teams require to always put braces around the `if` body:

```
if (a < b) {
  System.out.println("a is smaller!");
}
```

In your projects you can create your own code style or follow an existing one. I can recommend taking a look at the Google Java Style Guide (https://google.github.io/styleguide/javaguide.html). It does not cover every aspect of Java programming and usually lags in describing new features, but it's a good start to derive your own style from. As an alternative, you can base your style on Java code conventions from Oracle (https://www.oracle.com/java/technologies/javase/codeconventions-contents.html), though it's even older and was updated even before generics were introduced in Java. Many code style aspects can be configured in an IDE so that it will format code automatically for you. For example, in Eclipse IDE, you can find code style settings in the Preferences window, under Java → Code Style. You will find most of the interesting options under 'Clean Up'. Press the 'Edit' button to see the detailed configuration window (Figure 1.1). There you will find options for both examples listed above ("Use uppercase for long literal suffix" and "Use blocks in if/while/for/do statements"), as well as many other interesting options. Try to change every option and see what changes in the Preview pane. Think which settings may help you to avoid bugs.

While a consistent code style is helpful to make code more readable and less error prone, it protects you only against a small set of errors. Most of the mistakes covered in this book cannot be solved via code style alone, so you have to use other approaches as well to make your code robust.


Profile 'Eclipse [built-in]'

Profile name: My Code Style

Export...

Code Organizing | Code Style | Duplicate code | Java Feature | Member Accesses | Missing Code | Performance | Source Fixing | Unnecessary Code

Control statements

- [✓] Use blocks in if/while/for/do statements
  - (●) Always
  - ( ) Always except for single 'return' or 'throw' statements
  - ( ) Only if necessary
- [ ] Combine nested 'if' statement in 'else' block to 'else if'
- [ ] Reduce indentation when possible
- [ ] Convert if/else if/else chain with 3 blocks min to switch
- [ ] Add elements in collections without loop

Expressions

- [ ] Use parentheses in expressions
  - ( ) Always (●) Only if necessary
- [ ] Extract increment/decrement from statement
- [ ] Pull up assignment
- [ ] Use instanceof keyword instead of Class.isInstance()

Number literals

- [✓] Use uppercase for long literal suffix

Variable declarations

- [ ] Use modifier 'final' where possible
  - [✓] Private fields [ ] Parameter [✓] Local variables

Functional interface instances

- [ ] Simplify lambda expression and method reference syntax

Preview:

```
for (int i = 0; i < inputList.size(); i++) {
    outputList.add(inputList.get(i));
}

if (isValid) {
  System.out.println(isValid);
} else {
  if (isEnabled) {
    System.out.println(isEnabled);
  }
}

if (i > 0) {
    return 0;
} else {
    i = i + 1;
}

boolean b= (i > 0 && i < 10 || i == 50);

boolean isPositive = ++i > 0;


if (isRemoved = list.remove(o)) {}


long number1 = 10L;

boolean isRightType = String.class.isInstance(o

private int i= 0;
public void foo(int j) {
```

19 of 113 clean ups selected

A new profile will be created.

Apply | OK | Cancel

Figure 1.1 Eclipse Clean Up configuration window.

1.4 Static analysis

Static analysis is another very helpful technique that may help to detect bugs. To some extent, it could be called an automated code review: it points to suspicious, confusing, or likely erroneous code just like a human reviewer could do. Still, it cannot replace code review. While static analysis is very good at detecting specific kinds of bugs, there are also many bugs that could be easily spotted by human but will never be detected by a static analyzer. Often, static analyzers look at specific code patterns and any deviation from these patterns may confuse the analyzer.

You may also encounter a term 'lint' or 'linter'. Lint was the first well-known static analysis tool for C language which was initially developed in 1978. Since then, this word is commonly used to refer to static analyzers, and some of them even use 'lint' in their name, like JSLint which is a static analyzer for JavaScript.

1.4.1 Static analysis tools for Java

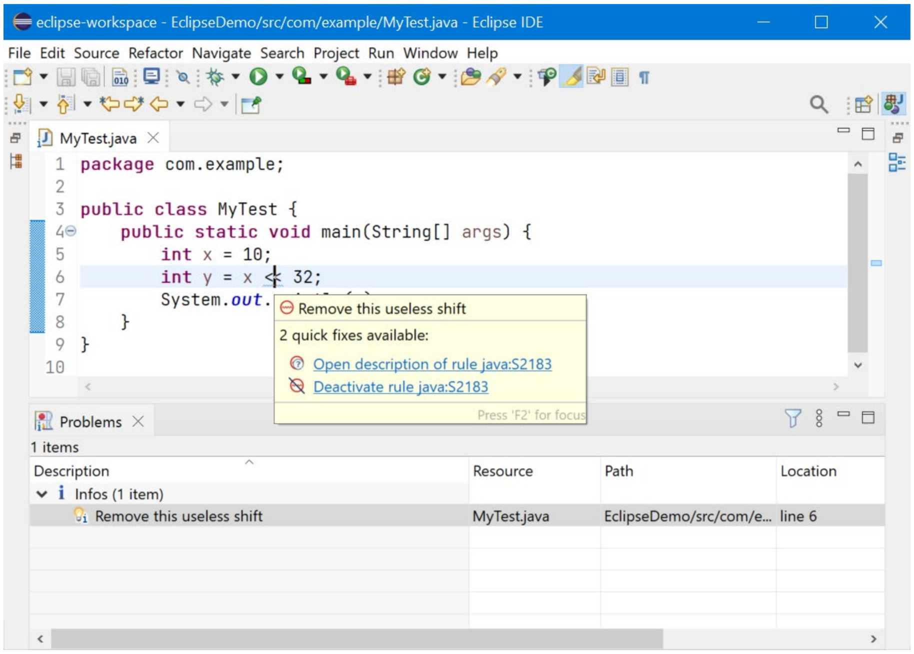
There are many static analyzers suitable for checking Java programs. Here's an incomplete list of them:

- IntelliJ IDEA: a Java IDE which has a powerful built-in static analyzer. Supports many languages including Java. While it's a commercial IDE, a free and open-source Community Edition is available, which includes most of the static analysis capabilities. In this book, we refer only to the functionality available in the free version of IntelliJ IDEA. https://www.jetbrains.com/idea/
- SonarLint: a free static analyzer by Sonar, which is available as an IDE plugin. Supports many languages including Java. This static analyzer is also integrated into Sonar continuous code quality inspection platform SonarQube. https://www.sonarsource.com/products/sonarlint/
- Error Prone: an open-source static analyzer developed by Google that works as a Java compiler plugin. https://errorprone.info/
- PVS-Studio Analyzer: a proprietary static analyzer for C, C++, C# and Java. https://www.viva64.com/en/pvs-studio/
- PMD: An open-source rule-based extensible cross-language static code analyzer. Aside from Java, it detects various problems in other languages such as XML or EcmaScript. https://pmd.github.io/
- SpotBugs: static analyzer that checks Java bytecode rather than source files. https://spotbugs.github.io/
- Coverity: a proprietary static code analysis tool from Synopsis. It covers many languages and supports more than 70 frameworks for Java, JavaScript, C#, and so on. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html
- Klocwork: a static code analysis tool owned by Perforce. It's focused on security and safety and covers a range of programming languages, including C, C++, C#, Java, JavaScript, and Python. https://www.perforce.com/products/klocwork
- CodeQL: a static analyzer integrated with GitHub. If your project is hosted on GitHub, you can set up a workflow to scan the project automatically. It supports many languages and is capable of providing deep interprocedural analysis. It also includes a query language which allows you to extend the analyzer. [Reference] https://codeql.github.com/

Other analyzers exist as well. Each of them has their own strong and weak sides. We are not going to compare them here, it's up to you which one to use. Just note that using any static analyzer is much better than using none. Some projects use several analyzers at once to be able to catch more problems.

1.4.2 Using static analyzers

Some static analysis tools can be installed as IDE plugins or extensions. For example, you can easily install SonarLint as a plugin for popular Java IDEs. In Eclipse IDE, open Help → Eclipse Marketplace, then search for “SonarLint”. After installation, it will report problems automatically in the editor. For example, let’s consider a simple mistake when you shift an int value by 32 bits. This operation has no effect in Java, and such code is likely a mistake, so it’s reported by virtually any Java static analyzer. Let’s see how it’s reported by static analyzers. Figure 1.2 shows how it may look in Eclipse IDE with SonarLint plugin installed:


eclipse-workspace - EclipseDemo/src/com/example/MyTest.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
MyTest.java
Problems

```
package com.example;

public class MyTest {
    public static void main(String[] args) {
        int x = 10;
        int y = x << 32;
        System.out.
    }
}
```

Remove this useless shift
2 quick fixes available:
Open description of rule java:S2183
Deactivate rule java:S2183
Press 'F2' for focus

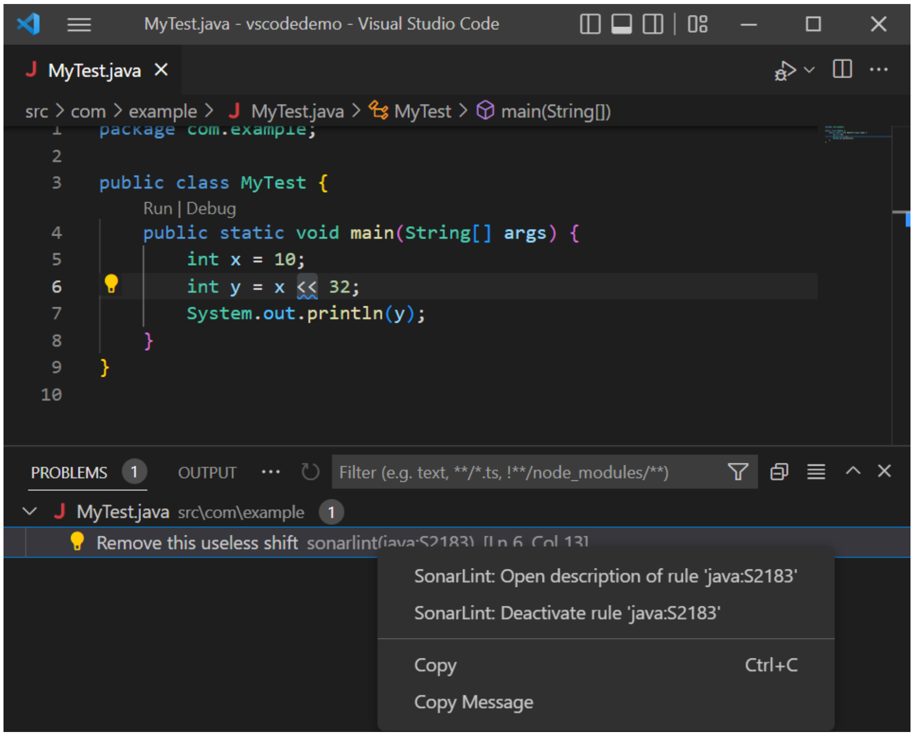
1 items

Description	Resource	Path	Location
Infos (1 item)			
Remove this useless shift	MyTest.java	EclipseDemo/src/com/e...	line 6

Figure 1.2 Eclipse IDE with SonarLint plugin installed.

The mistake is underlined in the editor and listed in the “Problems” view. If you hover over the mistake, you have options to read the description or deactivate the rule. SonarLint also allows you to run the analysis on a whole project or some part of it (e. g., a particular package).

SonarLint is also available as an extension for Visual Studio Code. Select the action “Install Extensions”, then search for “SonarLint” and install. It will immediately start analyzing your source code and looks very similar to Eclipse, as shown in figure 1.3.


MyTest.java - vscodedemo - Visual Studio Code
MyTest.java
src > com > example > MyTest.java > MyTest > main(String[])
Run | Debug
PROBLEMS 1
OUTPUT
Filter (e.g. text, **/*.ts, !**/node_modules/**)
MyTest.java src\com\example 1

```
package com.example;

public class MyTest {
    public static void main(String[] args) {
        int x = 10;
        int y = x << 32;
        System.out.println(y);
    }
}
```

Remove this useless shift sonarlint(java:S2183) [Ln 6, Col 13]

SonarLint: Open description of rule 'java:S2183'
SonarLint: Deactivate rule 'java:S2183'
Copy Ctrl+C
Copy Message

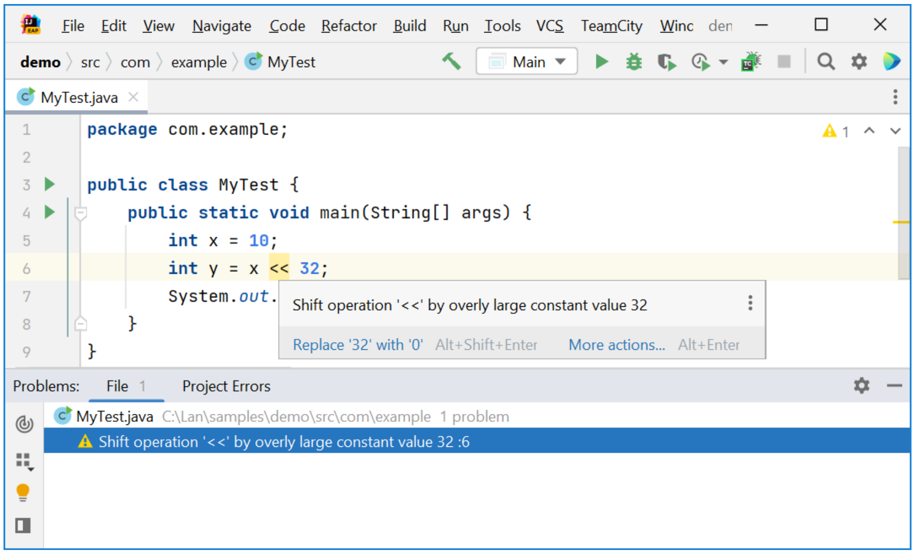
Figure 1.3 Visual Studio Code with SonarLint plugin installed.

Again, you have a problem listed in Problems view and the ability to deactivate the rule or read its description.

Plugins and extensions for other static analysis tools like SpotBugs or PVS-Studio can be installed in a similar way. Note that PVS-Studio generally requires a paid license.

Static analyzers like Error Prone are better integrated into build process. You can add the corresponding plugin to your Maven or Gradle build script, depending on the build system used. See the installation documentation at https://errorprone.info/docs/installation for further information.

If you use an IDE like IntelliJ IDEA, you get a powerful static analyzer automatically. Android Studio, an IDE from Google for Android developers, is based on IntelliJ IDEA and contains its static analyzer as well. In this case, you don't need to additionally install or configure anything to start using static analysis. Simply make a mistake, and it will be highlighted right in the editor as an inspection message, as shown on figure 1.4. Still, if you want to use SonarLint in IntelliJ IDEA, you can install it as a plugin from the plugin marketplace.


File Edit View Navigate Code Refactor Build Run Tools VCS TeamCity Winc den
demo 〉 src 〉 com 〉 example 〉 MyTest
Main
MyTest.java
Problems: File 1 Project Errors

```java
package com.example;

public class MyTest {
    public static void main(String[] args) {
        int x = 10;
        int y = x << 32;
        System.out.
    }
}
```

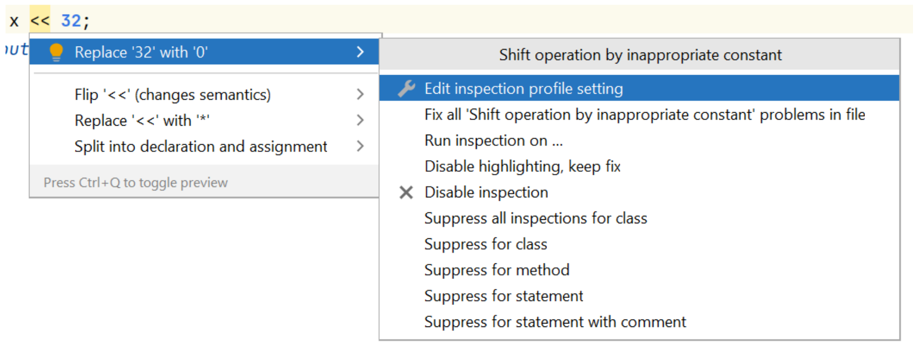
```
Shift operation '<<' by overly large constant value 32
Replace '32' with '0'  Alt+Shift+Enter    More actions...  Alt+Enter
```

```
MyTest.java  C:\Lan\samples\demo\src\com\example  1 problem
  Shift operation '<<' by overly large constant value 32 :6
```

Figure 1.4 IntelliJ IDEA built-in static analyzer warning

An inspection may suggest a quick-fix action to eliminate the mistake. Don't be very eager to apply it though. Sometimes quick-fix actions just simplify the code, preserving current semantics. But if it's really a bug, then the semantics should be changed. Here, for example, the quick fix is suggested to change the operation to `x << 0`. This preserves the code semantics but it's unlikely to fix the problem. Probably the author wanted to specify a different constant or use the `long` type instead of `int`, where shifting by 32 bits makes sense. We don't know for sure, but the suggested quick fix won't help us. Always investigate the root cause of the warning instead of applying a quick fix blindly.

If you press Alt+Enter, you may see more fixes and context actions. Pressing the right arrow, you'll find options to edit the inspection settings, suppressing the warning or disabling the inspection completely (Figure 1.5).



```
x << 32;
```

Replace '32' with '0'
Flip '<<' (changes semantics)
Replace '<<' with '*'
Split into declaration and assignment
Press Ctrl+Q to toggle preview
Shift operation by inappropriate constant
Edit inspection profile setting
Fix all 'Shift operation by inappropriate constant' problems in file
Run inspection on ...
Disable highlighting, keep fix
Disable inspection
Suppress all inspections for class
Suppress for class
Suppress for method
Suppress for statement
Suppress for statement with comment

Figure 1.5 Static analysis warning additional menu that allows you to suppress the warning, edit inspection options, or disable the inspection.

Many static analyzers are extendable. Usually, it’s a kind of plugin API which allows users to write a custom detector for new problems. It also could be a kind of flexible configuration or scripting, like the Structural Search & Replace feature in IntelliJ IDEA. We will discuss extending static analyzers in more detail in the appendix of this book.

1.4.3 Limitations of static analysis

While in-editor or online static analysis is powerful, it has limitations due to limited CPU resources and necessity to process the constantly changing code. Offline static analysis can afford spending more CPU time and perform deeper analysis. In particular, the online analyzer usually does very limited inter-procedural analysis. For example, assume that you are passing null value to method A, which passes it to method B, which passes it to method C where it’s unconditionally dereferenced yielding `NullPointerException` at runtime. This might be too difficult for online static analyzer to discover this. However, some offline analyzers can process the call graph and discover this problem.

Another problem with online analysis is that changes in one class may introduce problem in a completely different class which uses the current one. In this case, even if online analysis can detect this error, you may not notice it, unless you explicitly check all the call sites of all the methods you’ve changed. In this case, offline analysis that checks the whole project will be more helpful.

Offline static analysis is usually performed on a continuous integration (CI) server. Each offline analyzer usually provides a manual on how to integrate it with popular continuous integration tools. It’s recommended to configure static analysis on CI, monitor its results and investigate new problems. Sometimes, it’s even possible to integrate static analysis with code review tools, so newly introduced static analysis warnings will be shown as code review comments.

As I said above, in modern IDEs, online code analysis does not require any setup. However, sooner or later you will notice that it can be annoying. Some of reported problems could be irrelevant to your project. For example, static analysis might suggest you avoid using `System.out.println()` and prefer logging frameworks. This is a good advice for enterprise software but if you have a small research project, it's probably ok to rely on simple console output rather than deal with logging configuration.

Other warnings could be technically correct, but the reported code does not cause real problems in your project, and it would take too much effort to fix the warnings resulting in less readable code. For example, calculating average value as `(a + b) / 2` has an overflow risk if `a` and `b` are big enough (we will discuss this in more detail in Mistake #26, "Numeric overflow" in Chapter 4). However, it's possible that in your program `a` and `b` are always small and the problem never happens, so dealing with this warning would be just a waste of time.

Finally, sometimes static analysis warnings are plain wrong, as even the most advanced static analyzer is not perfect and cannot analyze a program completely. Also, don't forget that static analyzers are also programs, and they also contain bugs. It's possible that the static analyzer authors never considered a code pattern used in your project, thus wrongly report it as suspicious.

1.4.4 Suppressing unwanted warnings

In any case, you'll see a significant number of unwanted warnings. Analyzers provide various ways to remove them. Most of static analyzers consist of individual tools called 'diagnostics', 'rules', 'detectors', or 'inspections'. It's possible to turn them off selectively, either for the whole project, or for the specific part of it. For example, you may keep an inspection on in production code while disabling it in the unit tests. Additionally, analyzers may provide configuration options for inspections. This allows turning off the warning in specific situations or help the analyzer recognizing your own classes, like custom logging framework. Static analysis tools usually store the configuration in separate files, and you can commit them to version control and share it within your team or reuse the same configuration on a CI server.

Often, it's desired to suppress a particular warning, rather than disabling a whole inspection. Usually, this is done via code comments or annotations. IntelliJ IDEA supports both these ways. Suppression comments start with the "noinspection" prefix. It's also a good idea to explain every suppression. For example, assume that we create a collection and fill it but never read its content. IntelliJ IDEA does not like it and reports that the collection is written but never read. However, we are using it to be able to get useful information from memory dumps when something goes wrong. We can suppress the warning like this:

```
//Collection is never read: intended (used for debug)
//noinspection MismatchedQueryAndUpdateOfCollection
List<String> debugData = new ArrayList<>();
```

Alternatively, it's possible to use a `@SuppressWarnings` annotation:

```
//Collection is never read: intended (used for debug)
@SuppressWarnings("MismatchedQueryAndUpdateOfCollection")
List<String> debugData = new ArrayList<>();
```

Unfortunately, suppression annotations can be applied to variable declarations, methods, or classes only. If you put a suppression annotation on the method, no warning of a specified kind will be displayed inside that method. If you do this there's a risk that later another more useful warning of the same kind will appear in the same method, but it will be suppressed as well, and you will miss it. I recommend applying suppressions to the narrowest scope possible.

Even if different analyzers report the same problem, suppression ID and even the suppression annotation could be different. For example, table 1.1 shows how to suppress a warning on `x << 32` expression in various analyzers.

Table 1.1 Suppression of bad shift amount in different static analyzers

Static analyzer	Suppression annotation
IntelliJ IDEA	`@SuppressWarnings("ShiftOutOfRange")`
Error Prone	`@SuppressWarnings("BadShiftAmount")`
SonarLint	`@SuppressWarnings("squid:S2183")`
SpotBugs	`@SuppressFBWarnings("ICAST_BAD_SHIFT_AMOUNT")`

This complicates things if you use several different static analysis tools (e.g., one tool in the IDE and another on the CI server). If your code triggers a warning by several tools, you will need to suppress it for each tool separately.

Sometimes if you see an unwanted static analysis warning, it's better to modify the source code in a way that makes the analyzer happy, rather than adding a suppression. One may argue that we should aim to produce correct, performant, readable, and maintainable code but not the code that pleases the analyzer. However, often, fixing the code for analyzer also makes it better for human readers. It's not always easy to understand how to avoid a warning, but there are some common techniques that might be helpful:

- If you are using an unusual code pattern in many parts of your program and your static analyzer does not like it, consider extracting the pattern to a utility method and call that method everywhere instead. Now, you only need to suppress one warning inside that method. You can also write a documentation for the method and explain the rationale behind the unusual code pattern. This might be helpful for future code readers as well.

- If you calculate the same non-trivial value twice, consider extracting it to a local variable. Aside from removing the duplication, this will make analyzer to know that recalculation yields the same value. For example:

```
// May return null for some input values
@Nullable String calculateData(int value) {...}
void printData() {
  if (calculateData(1000) != null) {
    // Method invocation 'trim()'
    // may produce NullPointerException
    System.out.println(calculateData(1000).trim());
  }
}
```

Here, we are using the method `calculateData()` twice with the same parameter value. We assume that the method result depends only on the input parameter and does not change between invocations. However, the static analyzer may not know this. As the method is annotated as `@Nullable` (we will cover this annotation in more detail in Mistake #41 NullPointerException in Chapter 5), the analyzer assumes that its result value should always be checked against null, hence it produces an unwanted warning when `trim()` is called on the result of `calculateData()`. Instead of silencing the analyzer with a suppression, we can help it and extract the repeating computation into a new local variable:

```
String data = calculateData(1000);
if (data != null) {
  // no warning anymore
  System.out.println(data.trim());
}
```

Now, the analyzer knows that the value stored in the `data` was checked against null, thus no warning is issued anymore.

- Check if there's a better way to achieve your goal. For example, we observed a project where the effective assertion status was checked in many places, like this:

```
boolean assertsAreEnabled = false;
assert assertsAreEnabled = true;
if (assertsAreEnabled) {
  ...
}
```

When assertions are enabled the code authors decided to perform additional invariant checks and more logging to simplify testing. In production, assertions are usually disabled, and these additional branches are not executed.

Static analyzers don't like this code and produce several warnings like "Result of assignment is used" or "Side-effect in assertion statement". Should we suppress them?

Well, in general, the approach used here is questionable. Usually, a similar effect is achieved via logging libraries where you may have a more flexible configuration and more debug levels. But what is even more questionable is the way it's achieved. It would be more canonical to extract advanced checks to separate methods returning `boolean` and use them:

```
assert additionalInvariantsHold();
```

As an assert expression is not executed at all when assertions are disabled, you will effectively have the same behavior. It will require a bit of refactoring, but the code will be clearer.

Even if you actually want to write an `if` statement checking the assertion status, there's an API method for this:

```
if (getClass().desiredAssertionStatus()) {
  ...
}
```

Static analysis

Throughout the book, you'll see sidebars like this one, with the specifics of static analysis results for a given class of mistakes. There I mention how the corresponding inspection or rule is named in popular tools like IntelliJ IDEA or SonarLint, speak about the limitations of static analysis or additional configuration, which is necessary to report a particular problem.

1.5 Annotation packages

To make static analysis more efficient, it's useful to augment your Java program with additional hints. This is usually done via a special set of annotations distributed as a separate package. We saw an example in the previous section where a `@Nullable` annotation was used to indicate that method may return null and it's not safe to unconditionally dereference its result.

Most static analyzers either provide their own packages or understand annotations declared in other packages.

- Error Prone annotations. This package is provided together with the Error Prone static analyzer developed by Google. It declares a number of annotations recognized by Error Prone analyzer. Some of these annotations might be recognized by other static analysis tools as well. For example, `@CheckReturnValue` is recognized by IntelliJ IDEA. This package does not contain nullity annotations.

  Project page: https://errorprone.info/

  Package name: `com.google.errorprone.annotations`

  Maven coordinates: `com.google.errorprone:error_prone_annotations`

- The Checker Framework annotations. This package contains hundreds of annotations to augment the Java type system. The Checker Framework can use these annotations and report errors when augmented types are incompatible.

  Project page: https://checkerframework.org/

  Package name: `org.checkerframework`

  Maven coordinates: `org.checkerframework:checker-qual`

- JetBrains annotations. Annotation package supported by JetBrains and recognized by IntelliJ IDEA static analyzer. Aside from static analysis annotations, this package contains annotations that can aid debugging (e. g., `@Debug.Renderer`).

  Project page: https://github.com/JetBrains/java-annotations

  Package names: `org.jetbrains.annotations` and `org.intellij.lang.annotations`.

  Maven coordinates: `org.jetbrains:annotations`

- Android annotations. Annotation package created by Google to aid Android application developers. Aside from usual things like `@Nullable`, it contains Android-specific annotations like `@RequiresPermission`. These annotations are recognized by Android Studio IDE.

  Project page: https://developer.android.com/jetpack/androidx/releases/annotation

  Package name: `androidx.annotation`

  Maven coordinates: `androidx.annotation:annotation` (in Google repository)

- JDT Annotations for Enhanced Null Analysis. This package contains nullity annotations recognized by Eclipse compiler for Java and Eclipse IDE by default.

  Project page: http://www.eclipse.org/jdt/

  Package name: `org.eclipse.jdt.annotation`

  Maven coordinates: `org.eclipse.jdt:org.eclipse.jdt.annotation`

- “Java Concurrency in Practice” Book Annotations. This package contains annotations mentioned in the book “Java Concurrency in Practice” [Reference] to aid static analysis for concurrent programs. It includes four annotations: `@GuardedBy`, `@Immutable`, `@ThreadSafe`, and `@NotThreadSafe`. Some static analyzers like SpotBugs recognize them and may produce specific warnings. This package is not updated since the book’s publication.

  Project page: https://jcip.net/

  Package name: `net.jcip`

  Maven coordinates: `net.jcip:jcip-annotations`
- JSR 305[1]: Annotations for Software Defect Detection. It’s an abandoned initiative to standardize static analysis annotations. The package was published to Maven Central by FindBugs static analyzer developers. It never had an official status, and nobody supports it for a long time already. Nevertheless, you may still see these annotations in old projects and libraries. Even new projects occasionally depend on these annotations, likely because the package name (`javax.annotation`) looks somewhat official. The same package name is used in another artifact, namely JSR 250 Common Annotations, which serves another purpose, not related to static analysis. If you use both JSR 250 and JSR 305 packages in the same project you won’t be able to use Java Platform Module System, as it forbids using the same package name in several modules (so called ‘split-packages’). I advise to avoid using JSR 305.
- FindBugs/SpotBugs annotations. These annotations are recognized by FindBugs static analyzer and its successor, SpotBugs. This package mainly repeats JSR 305, only with different package name to solve the problem mentioned above. However, it still depends on meta-annotations declared in the JSR 305 package, so you’ll still have JSR 305 as a transitive dependency. This is probably not a big issue, but you should be careful to avoid accidental import of wrong annotation. For example, the annotation `javax.annotation.Nonnull` will be available in the project, along with `edu.umd.cs.findbugs.annotations.NonNull`, and it’s possible to mistakenly use the wrong one via code completion.

  Project page: https://spotbugs.github.io/

  Package name: `edu.umd.cs.findbugs.annotations`

  Maven coordinates: `com.github.spotbugs:spotbugs-annotations`

Here are some examples of what semantics static analysis annotations may convey:

- Annotation to indicate that return value of the method should not be ignored (e. g., `@CheckReturnValue` in Error Prone package);
- Annotation to depict that a method does not produce any side effects (e. g., `@SideEffectFree` in Checker Framework);

[1] JSR stands for Java Specification Request. JSRs are formal documents that describe proposed specifications and technologies for adding to the Java platform within the Java Community Process (JCP)

- Annotation to indicate that a method result is always a non-negative number (e. g., `@Nonnegative` in JSR 305 annotation package);
- Annotation to indicate that a method returns an immutable collection (like `@Unmodifiable` in JetBrains annotations);
- Annotation that requires overriding methods in subclasses to call this method (e. g., `@OverridingMethodsMustInvokeSuper` in Error Prone package).

And so on. Aside from hinting the static analyzers, these annotations serve the purpose of additional code documentation. E.g., seeing that a method is annotated with `@Nonnegative`, you immediately know that negative values are never returned by this method.

Unfortunately, it looks like there's no standard package recognized by a majority of static analysis tools, and similar annotations from different tools might have slightly different semantics. As a result, it might be hard to migrate from one static analysis tool to another if you already extensively use annotations recognized by the original tool. IntelliJ IDEA tries to understand annotations from different packages but it's not implemented in a consistent way, so some annotations are recognized while others are not.

1.6 Dynamic analysis

Aside from static analysis, there are also tools that perform dynamic code analysis to find bugs when executing your code. These tools are tailored to detect very specific bugs, mostly related to concurrency. One example of such a tool is Java Pathfinder[2]. It can detect data races, unhandled exceptions, potentially failed assertions and so on. It requires quite a complex setup and a lot of system resources to analyze even a small application.

Another tool that worth mentioning is Dl-Check[3]. It's a Java agent that aims to find potential deadlocks in multi-thread programs. Dl-Check is quite simple to run and it's really useful.

These tools don't replace static analysis but can augment it. Usually, dynamic analysis finds problems that are very hard to find via static analysis and vice versa. Static analysis tools usually detect many more bug patterns compared to dynamic analysis. Also, dynamic analysis detects problems in code that is actually executed, so if you use dynamic analysis with a test suite then the problems could be found only in code covered by tests. In contrast, static analysis can detect issues in any part of code, even if it's not covered by tests.

1.7 Automated testing

Automated testing is another technique to discover bugs. There are many kinds of automated tests ranging from unit testing to property-based testing, functional testing, integration testing and so on. There are whole methodologies around testing, like TDD (Test-driven development) in which you write tests before the actual code. You can learn more about TDD from many books, including "Test Driven: TDD and Acceptance TDD for Java Developers" [Reference]. To learn more about JUnit test framework, you can read "JUnit in Action" [Reference: https://www.manning.com/books/junit-in-action-third-edition].

[2] https://github.com/javapathfinder/
[3] https://github.com/devexperts/dlcheck

I especially like property-based testing, as this approach allows finding bugs in new code even without writing any new unit tests. The idea is to perform random operations with your program and check that the program has some expected properties, or its behavior is within the expected bounds. For example, no unexpected exceptions happen, read-only operations don't actually modify anything, and so on. If the property is violated, the testing framework provides you with a way to replay the sequence of operations and reproduce the problem. It also tries to remove some operations from the replay sequence if the problem is still reproducible without them. If you want to create property tests for Java software, you can try the jqwik framework[4].

Unfortunately, it's rarely possible to discover all the bugs with automated testing alone. Usually, the quality of tests is measured by a code coverage tool, like JaCoCo[5]. It instruments the Java bytecode and reports which fraction of the existing code is actually executed during automated tests run. It can report covered percentage in terms of lines of code, bytecode instructions, or branches.

My experience shows that when coverage exceeds 80%, increasing it further requires much more effort, which grows exponentially as coverage approaches 100%. So, if you set very high bar for test coverage in your project, you may end up spending all your time writing tests instead of actual functionality. Some lines of code are simply impossible to cover. For example, assume that you implement the `Cloneable` interface to be able to clone objects. As `Cloneable` is implemented, you don't want clients to deal with checked `CloneNotSupportedException` which would never happen, so you handle it by yourself:

```
class MyObject implements Cloneable {
  public MyObject clone() {
    try {
      return (MyObject) super.clone();
    } catch (CloneNotSupportedException e) {
      // never happens as we implement Cloneable
      throw new AssertionError(e);
    }
  }
}
```

It's impossible to write a unit-test that covers the catch section, and it's likely that your code coverage tool will always report the corresponding line as not covered.

On the other hand, even 100% test coverage doesn't mean that your code has no bugs. Consider a simple Java method that averages two `int` values:

```
static double average(int x, int y) {
  return (x + y) / 2;
}
```

Assume that we call it once in unit-tests with a single pair of values, like:

```
assertEquals(5, average(0, 10));
```

[4] https://jqwik.net/
[5] https://www.jacoco.org/jacoco/

This test passes, and the coverage tool will show 100% coverage. Does it mean that the method is free of bugs? In fact, it contains two problems:

- It uses integer division in a floating-point context. As a result, `average(0, 1)` returns 0.0 instead of 0.5.
- If an input number is too big, the result may overflow. For example, `average(2000000000, 2000000000)` returns -1.47483648E8 instead of 2.0E9.

It appears that both problems can be reported by static analysis tools. We will discuss them in more detail in Chapter 4.

Another problem with coverage analysis is that it cannot check whether your tests actually assert the expected behavior. There are many ways to write a test incorrectly, so that it executes your code but doesn't test its correctness. The most trivial mistake is to forget the assertion call. If your test simply calls `average(0, 10)` you'll still have the method covered, even if you don't check whether the result is correct.

Do not forget that tests are also code, so they may contain bugs which may prevent them from functioning properly. For example, an incorrectly written test may not fail if the code which is being tested functions improperly. We will discuss some bug patterns related to testing in Chapter 10.

1.8 Mutation coverage

There's a better approach to evaluate the quality of automatic tests called mutation coverage. When the tests are launched under mutation coverage engine, it instruments the code adding trivial edits. For example, changing `>=` comparison to `>` or `<=`, replacing a numeric variable `x` with `x + 1`, negating `boolean` variables, removing some calls, etc. Then for each edit (usually called 'mutations') it checks whether any of the tests fails after that. If yes, then the mutation is considered to be killed, otherwise it survives. The final report contains the percentage of killed mutations, and higher numbers mean that your test suite quality is better. The report also contains detailed information about the mutations that survived, so you can quickly understand which code requires more tests. The most popular mutation testing system for Java is Pitest[6].

Mutation coverage analysis helps to detect some problems that are not detected by normal coverage. For example, if you ignore the result of a method in an automated test, most of mutations introduced in that method will survive (some of the mutations may lead to exceptions or infinite loops, thus might be killed even if the method result is ignored). However, there are also downsides of this approach. First, complete mutation analysis is many times slower than normal tests run. It's acceptable if you have a small project and running all the tests normally takes several minutes. However, for bigger projects, it might be too slow. There are ways to perform mutation testing incrementally but it complicates the continuous integration process.

Another problem is that there are mutations that cannot be killed at all. For example, consider the following method:

[6] https://pitest.org/

```
public static int length(int from, int to) {
  if (from > to) return 0;
  return to - from;
}
```

It returns the length of the interval assuming that the length is zero if `from` is greater than `to`. The method is very simple, and it should be enough to have these tests to cover it (let's ignore the possible overflow problem):

```
assertEquals(0, length(1, 1));
assertEquals(100, length(0, 100));
assertEquals(0, length(100, 0));
```

However, if we run Pitest 1.7.6 with default options for this code it says that only 75% of mutations (3 of 4) are killed. That's because one of mutations (created by 'Conditional Boundary Mutator') replaces '`from > to`' condition with '`from >= to`'. As you can see from the code, this change doesn't affect the method result for any input, so it cannot be caught by any test. While sometimes changing the comparison operator might break the code significantly, here it's just a matter of preference whether to use '>' or '>=' operator. Of course, Pitest cannot know whether a particular mutation actually changes the code behavior. So even if you write every possible test, you cannot kill all the mutations.

Finally, you should really check all the survived mutations to find possible omissions in your automated test suite. If your coverage is not close to 100% too many mutations may have survived. So, it's a good idea to increase normal coverage at least to 90% before using mutation coverage.

1.9 Code assertions

A technique which is close to unit testing is adding assertions right into the production code. This way you can assert the expected state of your program and if it falls into unexpected one, it will immediately fail. This doesn't allow you to avoid bugs but helps to find them earlier and increases the chances that the bug will be found during the testing before the release. Also, if the code is well covered with assertions, the assertion failure happens close to the actual bug, so it becomes much easier to debug the problem. Without assertions, the system might exist in the incorrect state for much longer time until it's noticed.

Java provides a standard mechanism for assertions: the `assert` statement. It contains a `boolean` expression which must be true in a correctly written program and optionally, an additional expression that may add some explanation or context when the assertion is violated:

```
assert condition;
assert condition : explanation;
```

When Java programs are executed, assertions can be enabled via the JVM command line option `-ea`. They are disabled by default, and it's expected that they are disabled in production. If you use assertions, be sure that they are enabled during testing. It's also possible to enable or disable assertions for particular classes or packages, using additional parameters for the `-ea` and `-da` command line options, though this functionality is rarely used.

In some projects, it's desired to keep assertions enabled even in production. In this case, you should always think how to deliver the failed assertions back to developers and how to recover the application from failed assertion. Essentially, this requires catching all the assertions at some points in your program and either logging them (more suitable for server-side applications) or providing an error reporting mechanism for external customers. In this case, assertions don't differ much from other kinds of runtime exceptions like `IllegalArgumentException` or `IllegalStateException`. Sometimes, developers avoid using the standard Java assertion mechanism and replace it with some home-grown solution. For example, the IntelliJ platform contains `Logger.assertTrue()` method, which is wired to an error reporting tool, so if an assertion is violated, users can report it to developers.

Starting with Chapter 2, we will describe individual bug patterns, so you can recognize them early and see how to avoid them.

1.10 Summary

- There are various techniques that help you to reduce the number of bugs and find them quicker but there's no silver bullet.
- Different approaches are effective against different types of bugs, so it's reasonable to use several of them (e.g., static analysis, code reviews and automated testing) to some extent instead of concentrating all the efforts on one approach.
- Static analysis may help you to avoid many common mistakes. However, it may create unwanted warnings. You can reduce their amount configuring static analyzer, using suppression annotations, or rearranging your code to make it easier to understand for the analyzer.
- When actively using a static analyzer, it's advised to use an annotation package to give additional hints to the analyzer. Annotations like `@NotNull` or `@CheckReturnValue` not only make static analysis more efficient but also document code for future maintainers.
- Dynamic analysis tools may discover specific problems, notably concurrency-related. Such problems are unlikely to be detected by other tools.
- Unit tests are a very important quality technique, as they allow you to ensure that the code works as expected and get notified in case of regressions. Test coverage and mutation coverage metrics can be used to control how much of your code is actually covered. However, these metrics are not ultimate. 100% test coverage is usually impossible to achieve and even if achieved, it doesn't mean that the program is bug-free.

2 Expressions

This chapter covers

- **Precedence-related errors**
- **Common mistakes when one operator is used instead of another**
- **Pitfalls when using a conditional expression**
- **Mistakes with method calls and method references**

This chapter discusses the common bugs that are localized inside the single Java expression, such as using a wrong operator or assuming wrong operator precedence. Such bugs may result in unexpected resulting value of the expression. I will also briefly discuss the bugs when expression value is calculated correctly but ignored.

I don't discuss here bugs related to specific data types like numbers, strings or collections. These are covered in subsequent chapters.

2.1 Mistake #1. Wrong precedence in arithmetic expressions

Many programming languages, including Java, provide a number of operators that have different priorities, which affect the order of expression evaluation. The precedence can be changed using parentheses. Table 2.1 shows the operator precedence in Java.

Table 2.1 Operator precedence in Java

Precedence	Operators
1	Postfix operators: ++, –
2	Prefix operators: ++, –, +, -, ~, !
3	Multiplicative: *, /, %
4	Additive: +, -
5	Shift: <<, >>, >>>
6	Relational: <, >, <=, >=, instanceof
7	Equality: ==, !=
8	Bitwise &
9	Bitwise ^
10	Bitwise |
11	Logical &&
12	Logical ||
13	Conditional ?:
14	Assignment: =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

Some priorities are pretty natural, and rarely cause mistakes. For example, we all know from school that the multiplication and division operations have higher precedence than addition and subtraction. Java follows the same convention, so no surprises there.

2.1.1 Bitwise operators

Problems start to appear if you use more exotic operators, notably ones that manipulate with bits. For example, consider the following code sample:

```
int updateBits(int bits) {
  return bits & 0xFF00 + 1;
}
```

Try to guess how this expression is evaluated. What comes first, addition or bitwise 'and'?

Many developers mentally associate addition and subtraction with low-priority operators. However, it appears that they have higher precedence than bitwise operators. So, in this case, contrary to the authors intention, the expression is computed as `bits & 0xFF01`, and the least significant bit will be copied from `bits`, rather than set unconditionally.

The simplest advice for avoiding this kind of mistake is to not mix bitwise and arithmetic operators in single expression. It is rarely necessary and may only add confusion. In this sample, it's better to use the bitwise `or` operator:

```
return bits & 0xFF00 | 1;
```

Note that now, while the code is correct, it may still be confusing to readers. Bitwise `&` has higher precedence than bitwise `|`, but not every developer remembers this. It would be clearer to add parentheses:

```
return (bits & 0xFF00) | 1;
```

Static analysis

Some static analyzers may report expressions where it's desired to use parentheses to explicitly specify the precedence. For example, SonarLint has rule "S864: Limited dependence should be placed on operator precedence" and IntelliJ IDEA has the inspection "Multiple operators with different precedence". Both are turned off by default, as they may produce too many warnings and some people may find it annoying. However, if you consistently use this style from the very beginning of your project and fix warnings as they appear, you may avoid such unpleasant bugs.

Ways to avoid this mistake:

- Avoid mixing bitwise and arithmetic operations within a single expression.
- Add parentheses when using uncommon operations. Even if you know Java operator precedence perfectly, another developer on your team may not know it and may interpret your code incorrectly.

2.1.2 Binary shift

Another group of operators whose precedence is often confusing is binary shift operators. They are semantically close to multiplication and division, so it's natural to expect that their precedence is higher than addition. However, this is not the case:

```
int pack (short lo, short hi) {
  return lo << 16 + hi;
}
```

Here, the code author expected that the binary shift will be evaluated before the addition, like in `(lo << 16) + hi`. However, as addition's precedence is higher, this will be evaluated as `lo << (16 + hi)`, resulting in incorrect value. This kind of mistake cannot be caught by the compiler because there's no type mismatch. Normally this could be caught by unit-tests, but sometimes such bugs appear in production code. I saw such mistakes in manual `hashCode()` method implementations:

```
public int hashCode() {
  return xmin + ymin << 8 + xmax << 16 + ymax << 24;
}
```

Here, shift operators are executed after summation, so the result will be completely unexpected (figure 2.1). While this doesn't result in a behavioral error, the hash code distribution becomes extremely poor. In particular, as the last operation is shifting left by 24 bits, the least significant 24 bits of a hash code are always zero. This may produce unexpected performance problems if the object is used as a HashMap key.

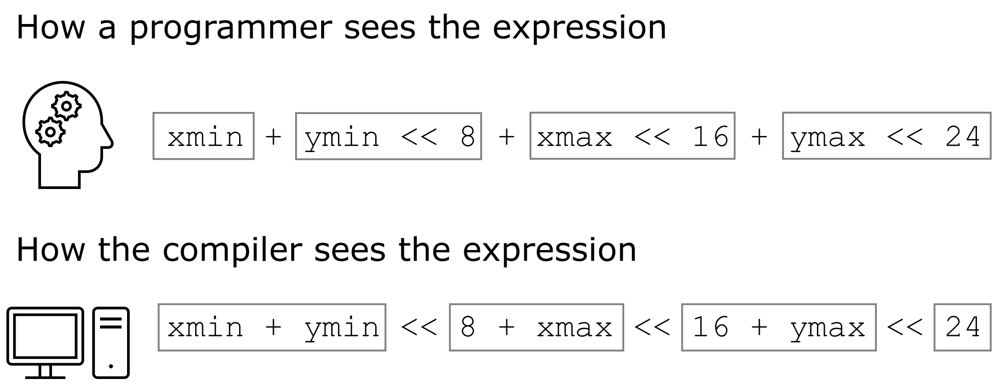

How a programmer sees the expression
xmin + ymin << 8 + xmax << 16 + ymax << 24
How the compiler sees the expression
xmin + ymin << 8 + xmax << 16 + ymax << 24

Figure 2.1 Numeric addition has higher precedence than bitwise shift.

A similar problem may occur if you want to add 25% or 50% to some amount (usually, a size of some buffer). I saw the following code in Apache Avro project:

```
class OutputBuffer extends ByteArrayOutputStream {
    static final int BLOCK_SIZE = 64 * 1024;
    ...
    public OutputBuffer() {
        super(BLOCK_SIZE + BLOCK_SIZE >> 2);
    }
}
```

Here, the author wanted to specify the initial capacity of `ByteArrayOutputStream` to be 25% bigger than the `BLOCK_SIZE` constant. Unfortunately, it was overlooked that bitwise shift has a lower precedence than addition. As a result, the size is computed as `(BLOCK_SIZE + BLOCK_SIZE) >> 2`, which is equal to `BLOCK_SIZE / 2`.

Static analysis

It's interesting that the authors of this project actually used a static analyzer, and it reported this problem, saying that it's better to add parentheses to the expression that contains both addition and bitwise shift. The authors followed the recommendation blindly and added the parentheses to preserve the current semantics as a part of bigger clean up. After that, the code looked like this:

```
public OutputBuffer() {
  super((BLOCK_SIZE + BLOCK_SIZE) >> 2);
}
```

The static analyzer was happy and it didn't report anything else. Yet the problem is still there. This is quite a didactic case. If static analyzer highlights the suspicious code, don't blindly accept the suggested quick fixes. They may simply hide the problem instead of solving it.

Ways to avoid this mistake:

- Remember that all bitwise shift operators have lower precedence, compared to addition and subtraction. Always add explicit parentheses around them, like `(lo << 16) + hi`, or `BLOCK_SIZE + (BLOCK_SIZE >> 1)`.
- Avoid using bitwise shift operators in favor of multiplication or division. E. g. the expressions above could be rewritten as `lo * 0x10000 + hi` and `BLOCK_SIZE * 5 / 4` (or `BLOCK_SIZE + BLOCK_SIZE / 4` if overflow is possible) respectively. Now, parentheses are unnecessary, and the code is probably even more readable. Please note that JIT and AOT compilers used in modern JVMs are capable to rewrite the multiplication and division with the shift operation when emitting the machine code, so the performance will be the same.
- Avoid writing `hashCode` method manually. Instead, rely on code generators that are available in most of Java IDEs. For example, in Eclipse, you can use "Source → Generate hashCode() and equals()" menu action. Alternatively, you may use a library method `Objects.hash()`.

2.2 Mistake #2. Lack of parentheses in conditions

In Java there's no implicit conversion between the Boolean type (either primitive `boolean` or boxed `Boolean`) and any other type. Thanks to this, it's quite a rare case that unexpected operator precedence in condition causes an error that cannot be detected during the compilation. For example, a common mistake is the lack of parentheses around a negated `instanceof` operator; this will be immediately detected by the compiler:

```
// '!' operator not applicable to Object
if (!obj instanceof String) {}
// Correct
if (!(obj instanceof String)) {}
```

Another common problem is bitwise 'and' (`&`), 'or' (`|`), and 'xor' (`^`) operations. They have pretty low precedence. In particular, their precedence is lower than the comparison operation, which could be unexpected:

```
// '|' operator cannot be applied to boolean and int
if (flags == Flags.ACC_PUBLIC | Flags.ACC_FINAL) {}
// Correct
if (flags == (Flags.ACC_PUBLIC | Flags.ACC_FINAL)) {}
```

Luckily, the compiler also helps us here.

2.2.1 && and || precedence

However, this will not always be the case. One common source of confusion is the precedence of the `&&` and `||` logical operators. Both operators work with `boolean` operands and produce a `boolean` result, so compiler type checking does not help us here. Similar to bitwise `&` and `|`, logical `&&` has a higher precedence than `||`. However, it's easy to forget that and write a condition like this:

```
if (index >= 0 && str.charAt(index) == ' ' ||
                  str.charAt(index) == '\t') { ... }
```

Here it was intended to protect against negative index value before using the `charAt` method. However, the `&&` operator has a higher precedence, the last `charAt` call is not protected (Figure 2.2). So if a negative index is possible at this point, an exception is inevitable.

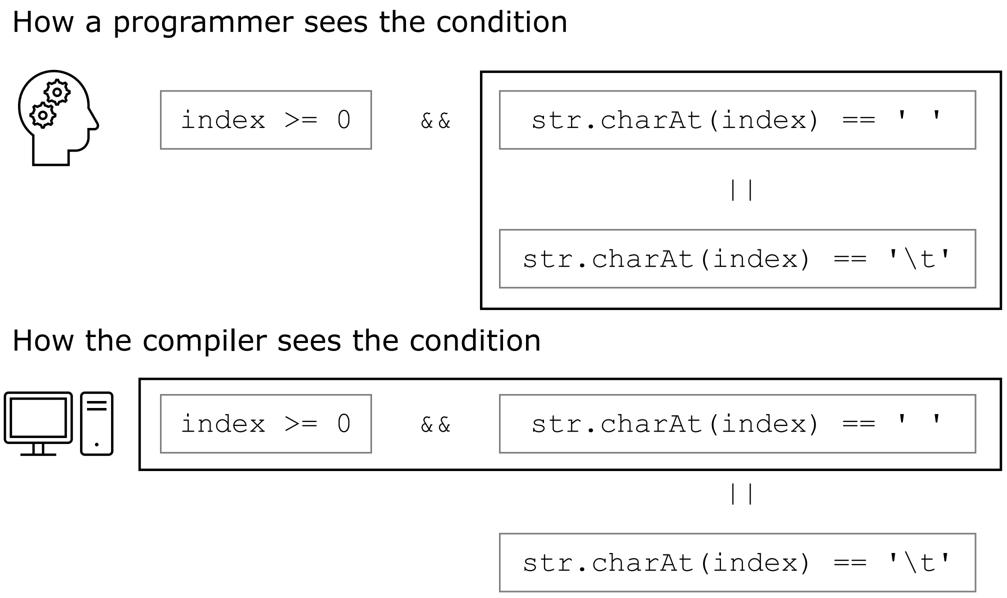

How a programmer sees the condition
index >= 0
&&
str.charAt(index) == ' '
||
str.charAt(index) == '\t'
How the compiler sees the condition
index >= 0
&&
str.charAt(index) == ' '
||
str.charAt(index) == '\t'

Figure 2.2 && has higher priority than ||

The correct code would be:

```
if (index >= 0 && (str.charAt(index) == ' ' ||
                   str.charAt(index) == '\t')) { ... }
```

Ways to avoid this mistake

- Pay special care when the condition contains both the `&&` and `||` operators. Reread it, think about all the possible inputs, and make sure that the output will be what you expect. Consult the Java operator precedence table if you have any doubts about precedence.
- Consider using parentheses every time the `&&` and `||` operators appear in single expression.
- Split complex conditions into simpler ones in separate `if`-statements. This not only helps to avoid mistakes but it also makes the code more readable. For example:

```
if (index >= 0) {
  if (str.charAt(index) == ' ' ||
      str.charAt(index) == '\t') { ... }
}
```

2.2.2 Conditional operator and addition

The conditional operator (`?:`, also known as the ternary operator) has one of the lowest priorities and this is sometimes irritating. Luckily, most of the mistakes involving conditional operator precedence will be easily caught by the compiler:

```
// Type mismatch: expected boolean, found String
return "Found " + multiple ? "multiple problems"
                           : "a problem";
// Correct
return "Found " + (multiple ? "multiple problems"
                            : "a problem");
```

However, the following is valid Java code:

```
static String indentString(String str, int indent) {
  int capacity = str.length() + indent < 0 ? 0 : indent;
  StringBuilder sb = new StringBuilder(capacity);
  for (int i = 0; i < indent; i++) {
    sb.append(' ');
  }
  sb.append(str);
  return sb.toString();
}
```

The idea is to add the requested amount of whitespace before the passed string, ignoring the negative amount. For the sake of allocation optimization, the code author tried to pre-calculate the capacity of the `StringBuilder`. However, due to missing parentheses, the calculation is wrong (Figure 2.3).

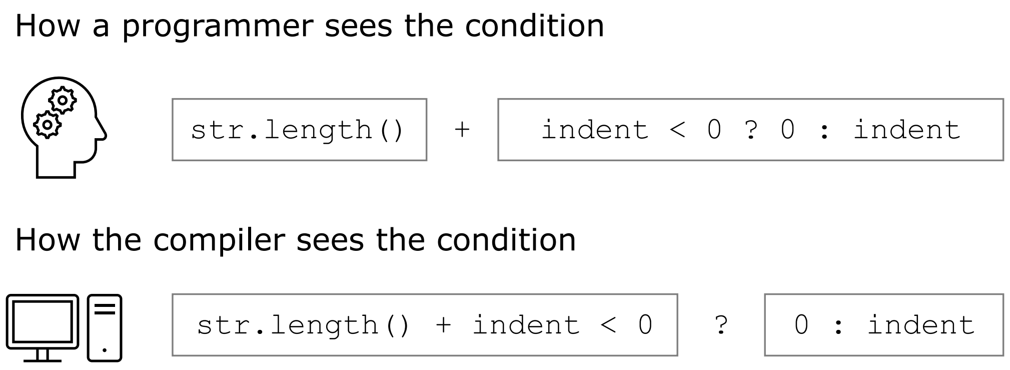

How a programmer sees the condition
str.length()
+
indent < 0 ? 0 : indent
How the compiler sees the condition
str.length() + indent < 0
?
0 : indent

Figure 2.3 Conditional operator has lower priority than plus.

At first, `str.length() + indent` is calculated and compared to zero. Depending on the result of comparison, the initial capacity is either set to `0` or to `indent`. For positive indents and for negative indents smaller than `-str.length()`, the method works correctly, even though the initial capacity is completely wrong. However, if the `indent` is between `-str.length()` and `-1`, the method fails with `NegativeArraySizeException`, as `StringBuilder` tries to allocate the array having `indent` elements.

WARNING This problem hardly could be detected by static analyzer, as there's nothing particularly suspicious in this calculation. In another context, it could be correct.

Ways to avoid this mistake:

- Always put the parentheses around a conditional expression when it's used as a part of more complex expression, as it likely has the lowest precedence.
- In this particular case, the problem could have been avoided by using the library method to calculate the max value:

```
int capacity = str.length() + Math.max(indent, 0);
```

- In general, prefer using library methods over manual computation, even if the computation looks very simple.

2.2.3 Conditional operator and null check

Another flavor of the same problem occurs when you compare a reference to null after a string concatenation and produce string values in conditional expression branches. For example, consider the following method:

```
String format(String value) {
  return "Value: " + value != null ? value : "(unknown)";
}
```

Here, the intent was to return `Value: (unknown)` if the value is null, and concatenate with an actual value if it's not null. However, the return expression is parsed as follows (figure 2.4):

```
(("Value: " + value) != null) ? value : "(unknown)";
```

As a result, the concatenation is executed first, then the null check is performed, which is always successful, as concatenation can never produce null. So, we will just return value always without prefix, and null will be returned as is. This could be unnoticed during manual testing, as the tester may assume that this behavior is intended. I discovered a bug like this in the procedure that generated some internal diagnostics information.

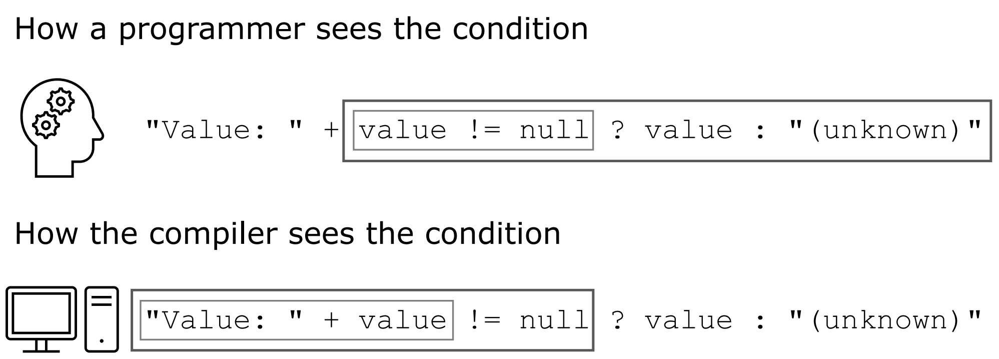

How a programmer sees the condition
"Value: " + value != null ? value : "(unknown)"
How the compiler sees the condition
"Value: " + value != null ? value : "(unknown)"

Figure 2.4 String concatenation has higher precedence than null-check or conditional operator.

Static analysis

This scenario can be easily caught by static analyzers, as it's very suspicious to compare the result of concatenation to null. IntelliJ IDEA reports the "Condition is always true" warning here and SonarLint reports "S2583: Conditionally executed code should be reachable".

Ways to avoid this mistake:

- Split complex expressions involving different operators using the intermediate variables:

```
String displayedValue = value != null ? value : "(unknown)";
return "Value: " + displayedValue;
```

Again, using the formatting method will make the code more robust (at the cost of some performance degradation):

```
return String.format("Value: %s",
                     value != null ? value : "(unknown)");
```

- Since Java 9, there's the `Objects.requireNonNullElse` API method that can replace such conditional expressions. It may look quite verbose but sometimes it adds clarity and solves the problem with operator precedence (use static import for this method):

```
return "Value: " + requireNonNullElse(value, "(unknown)");
```

I don't recommend using it always, it's more a matter of taste.

2.2.4 Initial capacity

I discovered a combination of the two problems discussed above in the Elasticsearch project, in code that initializes an `ArrayList` object with initial capacity. The original code is roughly equivalent to this method:

```
List<String> trimAndAdd(List<String> input, String newItem) {
  List<String> result =
    new ArrayList<>(input.size() + newItem == null ? 0 : 1);
  for (String s : input) {
    result.add(s.trim());
  }
  if (newItem != null) {
    result.add(newItem.trim());
  }
  return result;
}
```

Here, the number of elements in the resulting List is known in advance, and it looks like a good idea to pre-allocate it. However, due to the lack of parentheses, the plus operator is interpreted as string concatenation rather than numeric addition. As the result of concatenation is never null, the `ArrayList` always has the initial capacity of one element now, which is likely worse than the default value (Figure 2.5). Luckily, this bug can be easily caught by a static analyzer.

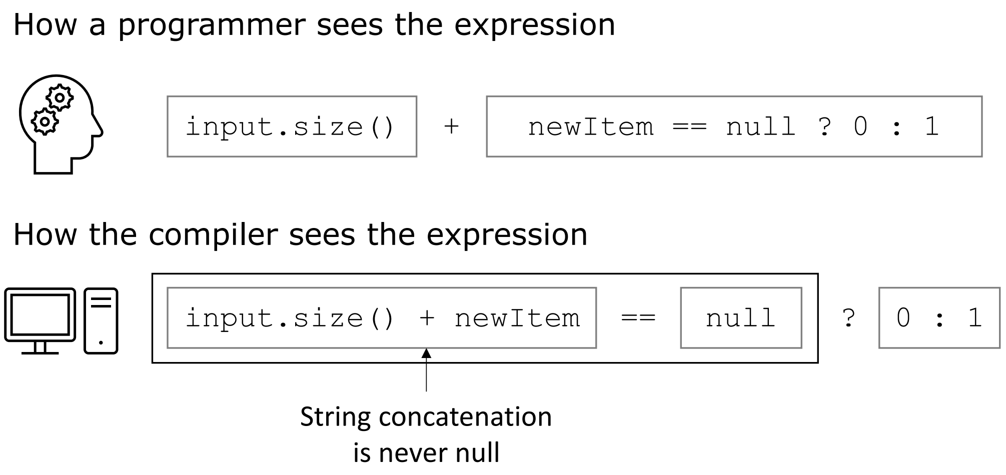

How a programmer sees the expression
input.size()
+
newItem == null ? 0 : 1
How the compiler sees the expression
input.size() + newItem
==
null
?
0 : 1
String concatenation is never null

Figure 2.5 Addition is interpreted as concatenation.

WARNING In general, setting the initial capacity of `ArrayList` and `StringBuilder` is a fragile optimization. Even if you never forget the parentheses, changes in the subsequent logic (e. g., adding one more element to the list) may require changes in initial capacity as well, which is often forgotten, and it's unlikely that the mismatched capacity will be caught by any test.

2.2.5 Conditional operator returning a boolean value

Sometimes precedence problems might arise if the conditional operator returns the `boolean` type. For example:

```
public boolean canHandle(boolean valid) {
  return isInitialized() &&
         valid ? hasValidHandlers() : hasHandlers();
}
private boolean hasHandlers() {...}
private boolean hasValidHandlers() {...}
private boolean isInitialized() {...}
```

The `canHandle()` method was intended to return true if the current object is initialized and has handlers. In the case that the supplied argument is true, we should have not any handler but a valid handler. However, as the logical precedence of `and` is higher than a conditional operator's precedence, the result is completely different. In particular, the result of `hasHandlers()` will be returned if this object is not initialized at all. To fix this, parentheses must be added:

```
return isInitialized() &&
       (valid ? hasValidHandlers() : hasHandlers());
```

Another flavor of this mistake happens when the additional condition is added after the conditional expression:

```
return valid ? hasValidHandlers() : hasHandlers()
       && isInitialized();
```

In this case, `isInitialized()` won't be called at all if the value of `valid` is true.

Ways to avoid this mistake:

- Always put a conditional operator in parentheses.
- Avoid using a conditional operator that returns a `boolean` type as a part of more complex expression. Use intermediate variables to make the code more readable and less error-prone:

```
boolean hasHandlers =
  valid ? hasValidHandlers() : hasHandlers();
return isInitialized() && hasHandlers;
```

- Prefer a chain of `if`-statements instead of complex conditional expressions involving different Boolean operators. This not only helps you to avoid precedence mistakes but also reduces cognitive load for readers of the code and simplifies debugging. For example, the method above could be rewritten as:

```
if (!isInitialized()) return false;
if (valid) {
  return hasValidHandlers();
}
return hasHandlers();
```

2.3 Mistake #3. Accidental concatenation instead of addition

One of the problematic things in Java syntax is a binary `+` operator that can be used both for string concatenation and for numeric addition. Along with implicit conversion of numbers to strings inside the concatenations, this may cause unexpected bugs.

```
String entryName = "Entry#" + index + 1;
```

Here the code author wanted to adjust the zero-based index, so, e. g., for `index = 4`, it was desired to produce the string `Entry#5`. However, Java executes all the `+` operations left-to-right and interprets this line as a sequence of two string concatenations, resulting in the string `Entry#41`. Depending on the context, this may result in a runtime error, querying irrelevant data, or displaying an incorrect message in the UI.

Ways to avoid this mistake:

- Always put arithmetic operations inside string concatenations in parentheses:

```
String entryName = "Entry#" + (index + 1);
```

Add them, even if parentheses are redundant, like here: `(price + tax) + "$"`. This will make the code more readable.

- Do not mix arithmetic operations and string concatenation in a single expression. Extract all the arithmetic operations to the intermediate variables:

```
int adjustedIndex = index + 1;
String entryName = "Entry#" + adjustedIndex;
```

- Avoid string concatenation in favor of formatting calls:

```
String entryName = String.format("Entry#%d", index + 1);
```

Or

```
String entryName = MessageFormat.format("Entry#{0}",
                                        index + 1);
```

Unfortunately, in modern JVMs, formatting calls are significantly slower than string concatenation, so this might be undesirable on hot code paths. However, if the result of concatenation is about to be displayed in the UI, then UI-rendering code performance will likely dominate the formatting performance.

2.4 Mistake #4. Multiline string literals

Before Java 15 introduced text blocks, it was common to represent long multiline string literals as a concatenation, like this:

```
String template =
  "<html>\n" +
  "  <head><title>Welcome</title></head>\n" +
  "  <body>\n" +
  "    <h1>Hello, $user$!</h1>\n" +
  "    <hr>\n" +
  "    <p>Welcome to our web-site</p>\n" +
  "  </body>\n" +
  "</html>\n";
```

There's nothing wrong with this approach if you cannot upgrade to Java 15 or newer. The care should be taken if you want to process the whole concatenation somehow. For example, imagine that you want to perform replacement of `$user$` placeholder right here. It's quite easy to make the following mistake:

```
String greetingPage =
  "<html>\n" +
  "  <head><title>Welcome</title></head>\n" +
  "  <body>\n" +
  "    <h1>Hello, $user$!</h1>\n" +
  "    <hr>\n" +
  "    <p>Welcome to our web-site</p>\n" +
  "  </body>\n" +
  "</html>\n".replace("$user$", userName);
```

It might not be evident at the first glance, but the replacement is applied to the last string segment `</html>\n` only. As a result, the `replace()` call does nothing and the placeholder is not substituted. This will not result in a compilation error nor a runtime exception but a user will see the `$user$` placeholder instead of the correct user name.

Static analysis

Pay attention to static analysis warnings. IntelliJ IDEA has "Replacement operation has no effect" inspection, which can sometimes detect if string replace operation does nothing. This can help to detect some of bugs like this.

Ways to avoid this mistake:

- Don't forget that string concatenation has lower precedence than a method call. Always wrap method call qualifier into parentheses, unless it's array element access, another method call, or a new expression.
- Avoid calls on complex qualifiers. Extract a complex qualifier to a separate variable whenever possible.
- Update to Java 15 or newer and use text blocks instead of string concatenation when possible.

2.5 Mistake #5. Unary plus

Java has a unary `+` operator. In most cases it’s completely harmless as it does nothing. Note however that it performs a widening conversion from the `byte`, `char` and `short` types to the `int` type. This may cause an unpleasant bug when it's accidentally used inside a string concatenation before a character literal:

```
String error(String userName) {
  return "User not found: " +
        + '"' + userName + '"';
}
```

Can you notice the problem? Here, duplicate plus was accidentally written around the line-break. In this case, the second plus is interpreted by the compiler as a unary plus applied to the double-quote character. This results in a widening conversion, changing the double-quote character to its code (34). So the method will return `User not found: 34User"` instead of the expected `User not found: "User"`. Exact UI messages are not always covered by unit tests, especially if they are displayed in error handling code. As a result, such mistakes may slip into production.

Another typo is possible due to existence of unary plus:

```
x =+ y; // x += y was intended
```

Here, ‘`=+`’ was mistakenly used for compound assignment instead of ‘`+=`’. This code successfully compiles and can be executed but the result will be different, as old variable value will be ignored.

Static analysis

Some static analyzers may recognize adjacent ‘=+’ and issue a warning. SonarLint has a rule named ‘S2757: "=+" should not be used instead of "+="’ to report this. Unfortunately, it’s possible that automatic formatting is applied later to the code, and the code will look like this:

```
x = +y; // x += y was intended
```

Now, static analyzer sees that there’s a space between ‘=’ and ‘+’ and may assume that unary plus was intended, so the warning is not issued anymore. We found a similar issue in an old codebase when we looked for all unary plus instances.

Ways to avoid this mistake:

- Avoid using character literals in favor of string literals where possible. E. g., in this case it would be better to use a one-character string `"\""` instead of the character literal `'"'`. Any arithmetic operation including unary plus applied to a string literal will result in a compilation error. The performance penalty is marginal in most cases.
- Avoid string concatenation in favor of formatting calls like `String.format`.
- Avoid blindly reformatting the code if it contains static analysis warnings. Fix the warnings first. It's quite possible that automatic reformatting will only hide the problem.
- Avoid unary plus completely and configure your static analyzer to report all instances of unary plus operator. In this case, automatic formatting will not suppress the static analysis warning.

Unfortunately, a similar problem may appear with unary minus as well:

```
x =- y; // x -= y was intended.
```

It’s not practical to ban unary minus from the code base. Again, static analyzer may help you, unless automatic formatting is applied to the code. Still, you can rely on careful code review and unit tests. Luckily, compound subtraction is used much less, compared to compound addition. We checked a big project and found that `‘+=’` occurs seven times more, than `‘-=’`.

2.6 Mistake #6. Implicit type conversion in conditional expression

Conditional expression, or ternary expression is quite similar to the if-statement. Often developers assume that the conditional expression like this

```
return condition ? thenExpression : elseExpression;
```

is completely equivalent to the following if-statement

```
if (condition) {
  return thenExpression;
} else {
  return elseExpression;
}
```

Often, it’s so but when you deal with numbers, the semantics could be different.

2.6.1 Boxed numbers in conditional expressions

The problem is that conditional expression performs non-trivial and sometimes counter-intuitive type conversions on then-expression and else-expression. This is especially important when you have boxed numbers. For example, consider the following method:

```
Double valueOrZero(boolean condition, Double value) {
  return condition ? value : 0.0;
}
```

It looks like when the condition is true we just return the argument unchanged. However, this is not what the Java Language Specification says (see JLS §15.25). The procedure to determine the final type of conditional expression is rather complex, and it's not practical to learn them by heart, but it's useful to remember a few things:

- If the conditional expression deals with primitive types or primitive wrappers, its type is always determined by the expression itself, not by the surrounding context. Speaking in Java Language Specification term, it's a 'standalone expression', rather than a 'poly expression'. In our example, the fact that the method return is `Double` is completely ignored when determining the expression type.
- When then-expression is a boxed number or `Boolean` and else-expression is a primitive number or `boolean` (or vice-versa), the primitive type wins, so the result of the conditional expression here is the primitive `double`.

Only after the conditional expression is executed, its result is boxed again, as the method return type is boxed `Double`. Java implicitly calls static methods like `valueOf()` for boxing conversion (to get a boxed type from the primitive one), and instance methods like `doubleValue()` for the opposite unboxing conversion. So, if we make all the implicit operations explicit, this code will be equivalent to:

```
Double valueOrZero(boolean condition, Double value) {
  return Double.valueOf(
          condition ? value.doubleValue() : 0.0);
}
```

Now the problem is pretty clear. First, the `value` is always reboxed, so you may unexpectedly waste more memory having two separate boxed values. Second, and more important: due to implicit dereference calling `valueOrZero(true, null)` results in `NullPointerException`.

Ways to avoid this mistake:

- Avoid using boxed primitives. Sometimes they are necessary if you work with generic types. However, it's better to convert them to primitive values as early as possible. This will also show clearly that the variable cannot be null.
- If you still need a boxed primitive, try not to use it with a conditional expression. Use an `if`-statement. It's more verbose but will make the code much less confusing. Even if you totally understand all the subtleties of the conditional expression, the code could be fragile, and a slight change in types may cause an unexpected behavior change.
- Remember that numeric conditional expression is a standalone expression, which means that the surrounding context does not affect its type.

2.6.2 Nested conditional expressions

Unexpected behavior may also be observed if you use nested conditional expressions with boxed primitive type:

```
static Integer mapValue(int input) {
  return input > 20 ? 2 :
         input > 10 ? 1 :
         null;
}
```

This may look okay at first glance: we return a boxed `Integer` value, so 2, 1, or `null` look like valid return results. However, this method throws `NullPointerException` if the `input` happens to be 10 or less. Even more amusing is that if you change the order of conditions, it will work as expected:

```
static Integer mapValue(int input) {
  return input <= 10 ? null :
         input <= 20 ? 1 :
         2;
}
```

Here the rules we saw previously are also helpful. But we need one more rule: when `null` meets primitive (like `int`), the resulting type is a boxed primitive (like `Integer`). So, in the first sample, the conditional expression type is determined via the pattern shown in figure 2.6.

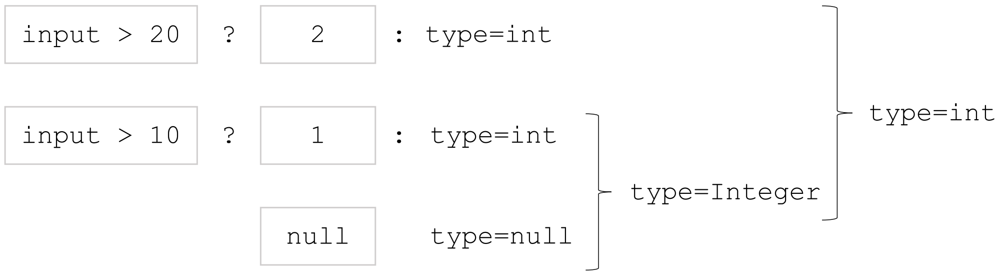


Figure 2.6 Type calculation: resulting type is int

On the other hand, the second method has a different pattern, thus a different result type, as in figure 2.7.

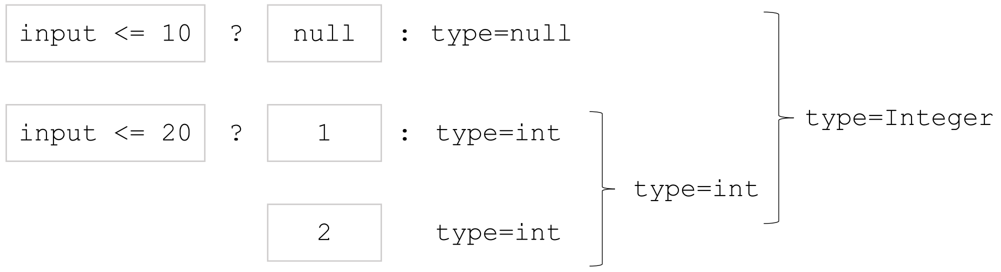


Figure 2.7 Type calculation: resulting type is Integer

As a result, the second code sample does not involve unboxing of `null`, thus it’s safe. Probably it will be easier to understand if we make all the boxing conversions explicit. The first sample has two implicit auto-boxing conversions and one auto-unboxing conversion:

```
return Integer.valueOf(input > 20 ? 2 :
    (input > 10 ? Integer.valueOf(1) : null).intValue());
```

The second sample is simpler, it has only one implicit auto-boxing conversion:

```
return input <= 10 ? null :
       Integer.valueOf(input <= 20 ? 1 : 2);
```

It’s ok if you don’t fully understand all the type conversions here. The thing you should remember is that conditional expressions have very non-trivial and counter-intuitive rules when it comes to boxing.

Static analysis

IntelliJ IDEA reports possible that the `NullPointerException` is possible as a result of unboxing (inspection name: 'Nullability and data flow problems'). PVS-Studio also reports this problem with the "V6093. Automatic unboxing of a variable may cause NullPointerException" warning. Unfortunately, SonarLint doesn't report possible exception here. It complains that the precedence of the operations could be confusing and suggests adding parentheses. After doing this, it becomes happy and doesn't highlight this code anymore, while the problem is still there.

Ways to avoid this mistake:

- Avoid nested conditional expressions. Use if statements instead.
- Avoid conditional expressions whose branches return different type. It's especially dangerous when at least one of the branches has a primitive type. The automatic type conversions are sometimes counterintuitive. It's much safer to use if-statement.

2.7 Mistake #7. Using non-short-circuit logic operators

Java provides two kinds of logical ‘and’ and ‘or’ operators. Short-circuit operators `&&` and `||` do not evaluate the next operand if the resulting value does not depend on it. Non-short-circuit operators `&` and `|` evaluate all the operands, regardless of the intermediate result. It appears that, in most cases, short-circuiting logic is much more natural, so it’s usually recommended to use them. However, accidental use of non-short-circuit operators is possible and will not be reported as compilation error. Sometimes such a typo is harmless (e. g., if the condition is like `a==null & b==null`). Sometimes you may have unnecessary performance degradation, if the second condition takes a considerable amount of time to execute. However, unpleasant consequences like runtime exceptions may also happen. For example:

```
boolean isPositive(int[] data, int index) {
  return index >= 0 && index < data.length & data[index] > 0;
}
```

Due to the typo, the second part of the bounds-check doesn’t work as expected. If the `index` exceeds the array length, array will still be queried, resulting in `ArrayIndexOutOfBoundsException`.

Note that compound assignment operators in Java never short-circuit. For example, sometimes programmers write a series of checks in the following way:

```
boolean result = true;
result &= check1();
result &= check2();
result &= check3();
```

This may look pretty and uniform but you should note that the methods `check1()`, `check2()`, and `check3()` will be always called, even if the result is already false. This does not always make your program incorrect, as the checks might be independent. However, this approach may require redundant computation. Unfortunately, there’s no short-circuiting compound assignment operator in Java, like `&&=`, so to avoid unwanted computations, you need to write this in its full form:

```
boolean result = true;
result = result && check1();
result = result && check2();
result = result && check3();
```

Another possible problem with accidental use of non-short-circuit operators is different precedence. We have seen a bug in a complex condition like this:

```
interface First {}
interface Second {}
interface Third {}
interface Exclude extends Second {}

boolean checkObject(Object obj) {
  return obj instanceof First && checkFirst((First)obj) |
         obj instanceof Second && !(obj instanceof Exclude) ||
         obj instanceof Third && checkThird((Third)obj);
}
```

Here, we have three acceptable interfaces, and we want to perform some additional checks for them. As we saw above, logical 'or' has lower precedence than logical 'and', so it looks safe to use a mix of `&&` and `||` without additional parentheses. However, due to an accidental edit, the first `||` was replaced with `|` (according to the version control history; initially the condition was correct). Non-short-circuiting logic operators have higher precedence than short-circuiting ones, so now the condition is evaluated like this (Figure 2.8):

```
return (obj instanceof First &&
        (checkFirst((First) obj) | obj instanceof Second) &&
        !(obj instanceof Exclude))
       ||
       (obj instanceof Third && checkThird((Third) obj));
```

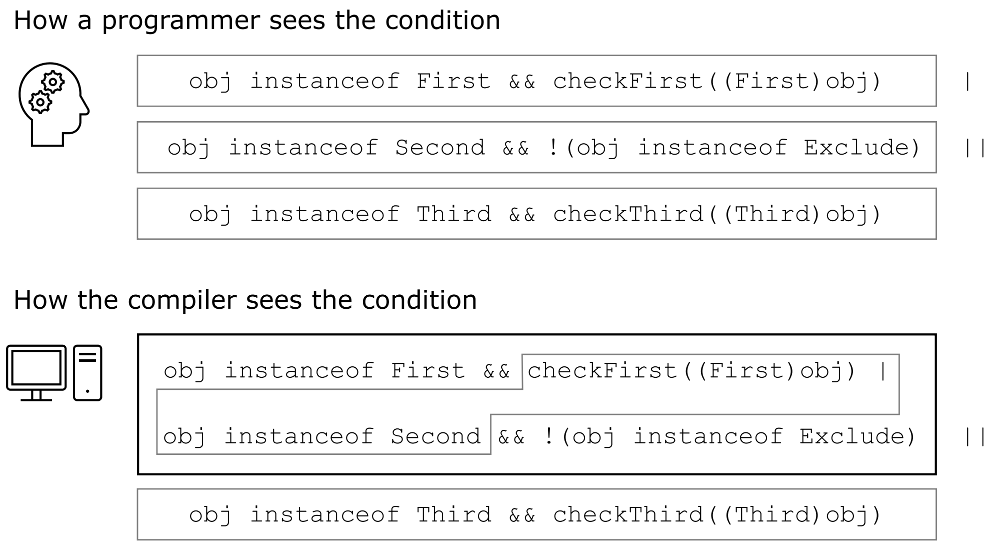

How a programmer sees the condition
obj instanceof First && checkFirst((First)obj) |
obj instanceof Second && !(obj instanceof Exclude) ||
obj instanceof Third && checkThird((Third)obj)
How the compiler sees the condition
obj instanceof First && checkFirst((First)obj) |
obj instanceof Second && !(obj instanceof Exclude) ||
obj instanceof Third && checkThird((Third)obj)

Figure 2.8 Non-short-circuiting logic operators have higher precedence.

In particular, if the input happens to implement the `Second` interface, it cannot satisfy the condition unless it implements the `First` interface at the same time. This cannot be encoded in the Java type system, but according to the program logic, `First`, `Second`, and `Third` interfaces are mutually exclusive, so now, objects of type `Second` stopped passing this condition.

Static analysis

Dataflow-based static analyzers can detect some dangerous uses of non-short-circuit logic. For example, a classic null-check before dereference is reported in IntelliJ IDEA by “Nullability and data flow problems” inspection if `&` is used instead of `&&`:

```
// Method invocation 'isEmpty' may produce
// 'NullPointerException'
if (str != null & str.isEmpty()) {}
```

However, this inspection may not report more complex cases, like the one shown in figure 2.8. A simple solution would be to ban non-short-circuit logic operators in the project completely. SonarLint reports every use of `&` or `|` in conditions with the rule “S2178: Short-circuit logic should be used in boolean contexts”. IntelliJ IDEA has a similar inspection called “Non-short-circuit boolean expression” but you need to switch it on in the inspection profile, as it's turned off by default.

Ways to avoid this mistake:

- Do not use non-short-circuiting logical operators, as they may lead to unwanted computations and even produce mistakes if the next condition depends on the previous one.
- Note that the compound assignment operators `&=` and `|=` are not short-circuiting.
- In a complex conditions that involves mix of `&&` and `||` operators, use clarifying parentheses, even if you know the precedence of Java operators by heart. The accidental edit in the `checkObject` example above would be harmless if `&&`-expressions were initially parenthesized.
- Extract parts of complex conditions involving different operators. You can use intermediate variables, extract short methods, or split the condition into several statements. This way, the code will be more readable and less error prone. For example, you can extract `instanceof` checks to separate if-statements:

```
if (obj instanceof First) {
  return checkFirst((First)obj);
}
if (obj instanceof Second) {
  return !(obj instanceof Exclude);
}
if (obj instanceof Third) {
  return checkThird((Third)obj);
}
return false;
```

This way, you don't use `&&` and `||` operators at all. If you are using Java 16 or higher, you can use pattern matching in `instanceof` to get rid of cast expressions:

```
if (obj instanceof First first) {
  return checkFirst(first);
}
if (obj instanceof Second) {
  return !(obj instanceof Exclude);
}
if (obj instanceof Third third) {
  return checkThird(third);
}
return false;
```

Finally, starting with Java 21, it will be possible to use pattern matching in switch, making code more uniform and more readable:

```
return switch (obj) {
    case First first   -> checkFirst(first);
    case Second second -> !(second instanceof Exclude);
    case Third third   -> checkThird(third);
    case null, default -> false;
};
```

Evaluating both operands

Sometimes, it's still desired to have non-short-circuit logic, if both operands produce intentional side-effect. It may look like this:

```
if (updateA() & updateB()) {
  ...
}
```

Such code could be confusing to readers, as they might not be sure whether `&` instead of `&&` was intended or it's a typo. If you configured your static analyzer to report all non-short-circuiting logic operators, an explicit suppression comment will be necessary here. While some developers don't like suppressions and prefer to disable static analysis checks altogether, in such a case, the comment may add clarity not only for static analyzer but also for readers. For example:

```
//Evaluating both operands is intended!
//noinspection NonShortCircuitBooleanExpression
if (updateA() & updateB()) {
  ...
}
```

A better alternative though is to refactor the code to avoid non-short-circuit logic completely:

```
boolean isAUpdated = updateA();
boolean isBUpdated = updateB();
if (isAUpdated && isBUpdated) {
  ...
}
```

This way, it's evident that evaluating both `updateA()` and `updateB()` is intended. In general, avoiding side effects in conditions leads to clearer and less error-prone code.

2.8 Mistake #8. Mixing && and ||

Human developers have trouble using logical "and" and logical "or" correctly. I have seen many bugs where one of them was used instead of another. One of the error patterns that frequently occurs looks like this:

```
void validate(int byteValue) {
  if (byteValue < 0 && byteValue > 255) {
    throw new IllegalArgumentException(
              "byteValue is out of range");
  }
}
```

Here, the author of the code wants to reject the parameter value that is outside of the range 0..255. However, due to an unfortunate mistake, the `&&` operator was used instead of `||`

operator. As you can see, no value can be less than zero and greater than 255 at the same time. As a result, this validation does nothing, and any `byteValue` is actually accepted.

We fixed more than ten of bugs like this. Here's an example of such an erroneous method:

```
private static boolean isPrintableAsciiString(byte[] array) {
  for (byte b : array) {
    char c = (char) b;
    if (c != 0 && c < 0x20 && c > 0x7F) {
      return false;
    }
  }
  return true;
}
```

The method should tell whether a given byte array contains only printable symbols. However, due to the incorrect check, it always returns true, so the `byte[]` array was always displayed as a string.

Some developers mix `||` and `&&` when trying to negate the condition manually by incorrectly applying De Morgan’s law. For example, assume that you are processing the text input and need to treat the lines starting with “#” or “//” as comments. You may have the following method:

```
void processLine(String line) {
  if (line.startsWith("#") || line.startsWith("//")) {
    processCommentLine(line);
  } else {
    processContentLine(line);
  }
}
```

Later, you decide that no special processing of comment lines is necessary, and you just need to skip such lines, so you need to keep only the `else`-branch:

```
void processLine(String line) {
  // Ignore comments
  if (!(line.startsWith("#") || line.startsWith("//"))) {
    processContentLine(line);
  }
}
```

Now, you may think that eliminating the parentheses would be a good idea. According to De Morgan’s law, `!(a || b)` is equivalent to `!a && !b`, so if you eliminate the parentheses, you should also change `||` to `&&`. But it’s quite possible to forget this and make a mistake instead:

```
void processLine(String line) {
  // Ignore comments
  if (!line.startsWith("#") || !line.startsWith("//")) {
    processContentLine(line);
  }
}
```

Such code is treacherous, as even for a seasoned developer; it will take a while to understand that no comment lines will be ignored. If a line starts with "#" it cannot start with "//" at the same time, so all the lines will be passed to the `processNormalLine()` method, and one can only guess what will happen next.

Static Analysis

This kind of mistake can be reported by static analyzers that are capable to perform range analysis of integral values. For example, IntelliJ IDEA has a "Constant values" inspection that will report "Condition is always false" warning on the `validate()` sample above. However, static analyzers may fail to detect the mistake if the code is more complicated:

```
void validate(int value) {
   if (Math.abs(value) < 10 && Math.abs(value) > 20) {
      throw new IllegalArgumentException();
   }
}
```

Not every static analyzer is aware that two calls of the `Math.abs()` method with the same argument will always yield the same value, so the problem is basically the same. Mutually exclusive conditions could be more complex. For example, the `processLine()` sample shown above is not reported by most static analyzers, despite the fact that the conditions `line.startsWith("#")` and `line.startsWith("//")` are mutually exclusive. That's because many static analyzers don't know the semantics of the `startsWith()` method and don't track the already filtered prefixes.

Ways to avoid this mistake:

- Pay attention to static analyzer warnings. Use advanced static analyzers that are capable to track value ranges and understand the semantics of various library methods. They may help you to identify always-true and always-false conditions.
- Extract the repetitive parts of condition to temporary variables. This may help static analyzers to track the values. For example, IntelliJ IDEA doesn't warn you in `Math.abs()` sample above. However, it will warn if you extract the `Math.abs()` call to the intermediate variable:

```
void validate(int value) {
    int abs = Math.abs(value);
    if (abs < 10 && abs > 20) {
        throw new IllegalArgumentException();
    }
}
```

This may also improve the performance and the code readability.

- Pay attention to the conditional logic when writing the unit tests. Test uncommon code paths like validation logic. When testing a method returning `boolean` type, write at least one test each for the case when the method returns true and for the case when the method returns false. This way you can easily detect if the method always accidentally returns the same value.
- To negate complex conditions, use IDE functions instead of doing it manually. If you want to keep only the `else`-branch of an `if`-statement, you can use the "Invert 'if' condition" action in IntelliJ IDEA or Eclipse to swap branches and then delete the `else`-branch. If you already have a condition in the form of `!(a || b)`, there's an action "Replace '||' with '&&'" in IntelliJ IDEA and an equivalent "Push negation down" action in Eclipse. These actions will correctly apply De Morgan's law to the condition, preserving the code semantics.

2.9 Mistake #9. Incorrect use of variable arity calls

Since Java 1.5, it's possible to declare the last method parameter as a variable arity parameter. This allows passing an arbitrary number of arguments when calling this method. Upon the method call, all the passed arguments are automatically wrapped into an array.

2.9.1 Ambiguous variable arity calls

It was decided to allow also to pass an array explicitly. For example, assume that you have the following variable arity method:

```
static void printAll(Object... data) {
    for (Object d : data) {
        System.out.println(d);
    }
}
```

Now, these two calls have the same result:

```
printAll("Hello", "World");
printAll(new Object[]{"Hello", "World"});
```

The ability to pass either an explicit array or a flat list of arguments is useful for code evolution: you can upgrade the fixed arity method having an array parameter to a variable arity method, and this change will not break the existing call sites. However, this convenience comes at a cost. When you have exactly one argument passed, it's not immediately clear whether it will be wrapped to array or not. For example, consider the following call:

```
printAll(obj);
```

Here, it's unclear whether `obj` will be wrapped into array. It depends on its type: if it's a non-primitive array, it will not be wrapped. It's very important to understand that the decision is taken based on the declared type, not on the runtime type. Consider the following code:

```
Object obj = new Object[]{"Hello", "World"};
printAll(obj);
```

The variable obj contains an object array, so one may expect that no wrapping is necessary. However, the declared type of the variable is simply Object, not array. The decision on whether to wrap is performed during the compilation, and the compiler does not know the real type of the object stored inside obj. So it decides to wrap an array into another array. As a result, this code prints something like [Ljava.lang.Object;@7106e68e, a default string representation of an array.

The situation becomes more confusing if you pass null to variable arity method:

```
printAll(null);
```

Some developers might expect that single null will be wrapped to array, and as a result, null string will be printed. However, according to the Java Language Specification (see JLS §15.12.4.2), the argument is not wrapped to array if it can be assigned to a variable of the corresponding array type. As null can be assigned to any array variable, it's not wrapped into array, so the data parameter becomes null inside the printAll method, and this call results in NullPointerException.

Static analysis

Static analyzers can warn you about ambiguous calls like here. IntelliJ IDEA has 'Confusing argument to varargs method' inspection for this purpose. SonarLint reports this problem as “S5669: Vararg method arguments should not be confusing”.

Ways to avoid this mistake:

- Be careful when passing only one argument as a variable arity parameter. When an ambiguity is possible, add an explicit type cast:

```
// I want an array containing null
printAll((Object)null);
// I want just null array reference
printAll((Object[])null);
```

- Avoid using nulls where possible, especially when you are dealing with arrays and variable arity methods.

2.9.2 Mixing array and collection

One more problem we encountered with variable arity method is using collection instead of array. For example, assume that you have the following utility method to check whether array contains a given element:

```
static <T> boolean contains(T needle, T... haystack) {
    for (T t : haystack) {
        if (Objects.equals(t, needle)) {
            return true;
        }
    }
    return false;
}
```

For convenience, the method was declared as variable arity method, so it could be used to compare a value against the set of predefined values:

```
boolean isDayOfWeek = contains(value,
    "SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT");
```

Again, this convenience comes at a cost. Now it’s possible to call this method with a collection argument:

```
static final List<String> allLanguages =
    Arrays.asList("Java", "Groovy", "Scala", "Kotlin");

static boolean isLanguage(String language) {
    return contains(language, allLanguages);
}
```

As arrays and collections are semantically similar, a developer may forget that `allLanguages` is actually not an array. Another potential way for this mistake to happen is that `allLanguages` was an array before but the developer decided to change it to the `List` and fix all compilation errors after that. However, the `contains(language, allLanguages)` call can still be compiled without errors. Now, the compiler infers type parameter `T` as `Object` and wraps the collection into single element array. So again, we have a `contains` method that compiles successfully but always returns false.

Ways to avoid this mistake:

- Be careful when changing the variable type from array to collection. Check every occurrence of the variable, not only compilation errors, as it’s possible that at some places no compilation error appears but the code semantics will change.
- Pay attention to 'Iterable is used as vararg' inspection warning in IntelliJ IDEA. This inspection tries to detect problems like this.

2.10 Mistake #10. Conditional operators and variable arity calls

Another source of confusion is when a conditional operator is used inside the variable arity method call. Assume that we want to print the formatted string via `System.out.printf()` but use the default substitution parameter if it wasn't supplied:

```
static void printFormatted(String formatString,
                           Object... params) {
  if (params.length == 0) {
    System.out.printf(formatString, "user");
  } else {
    System.out.printf(formatString, params);
  }
}
```

Let's test this method:

```
printFormatted("Hello, %s%n");
printFormatted("Hello, %s%n", "administrator");
```

These two calls print the following text, as expected:

```
Hello, user
Hello, administrator
```

Now, the developer notices that the branches of the `if`-statement inside the `printFormatted` method look very similar and wants to collapse it using the conditional expression:

```
static void printFormatted(String formatString,
                           Object... params) {
  System.out.printf(formatString,
          params.length == 0 ? "user" : params);
}
```

After this refactoring, the code looks more compact, and some people may like it more. The problem is that the code semantics changed. Now, our test prints something like:

```
Hello, user
Hello, [Ljava.lang.Object;@7291c18f
```

If we look at the original method, we'll see that the branches of the `if`-statement have an important difference: the then-branch uses a variable arity call style, and the compiler automatically wraps it into an array. On the other hand, the else-branch passes an array explicitly, so wrapping doesn't occur.

However, when we are using the conditional expression, the compiler cannot wrap only one branch of it. It perceives the conditional expression as a whole. As the type of the then-branch is `String`, and the type of the else-branch is `Object[]`, the type of the whole conditional expression is a common supertype of `String` and `Object[]`, which is simply `Object`. This is not an array, so the compiler adds automatic wrapping, which means that the `params` array is wrapped one more time. You can fix this if you avoid relying on the compiler and wrap the `user` string manually:

```
params.length == 0 ? new Object[]{"user"} : params
```

However, it’s probably less confusing to avoid conditional expression inside variable arity method call at all.

Static analysis

Pay attention to static analyzer warnings. IntelliJ IDEA has “Suspicious ternary operator in varargs method call” inspection that reports if one of conditional operator branches has an array while other is not.

Ways to avoid this mistake:

- Be careful when using conditional expression as a variable arity argument. Prefer using an if statement instead.
- If you still want to keep a conditional expression, extract it to a variable using "Introduce variable" or "Extract variable" refactoring in your IDE:

```
Object adjustedParams = params.length == 0 ? "user" : adjustedParams;
System.out.printf(formatString, adjustedParams);
```

Now, it's visible in the code that the expression type is `Object`, and the compiler will wrap it one more time.

2.11 Mistake #11. Ignoring the method result value

Like many other programming languages, Java allows you to ignore the resulting value of a method call. Sometimes this is convenient, such as when the resulting value is of no interest. For example, a `map.remove(key)` call returns the value that was associated with the removed key. Often, we already know it or it’s not necessary for the subsequent code.

A more extreme example is `list.add(element)`. The `List.add` method is an override of `Collection.add`, which returns false if the collection does not support duplicates and the element was already there. This is useful for `Set.add`, as you cannot add a repeating element to the set, so you can use the return value to check whether the element was already in the set before the call. However, the `List` interface supports duplicates, which implies that the `List.add` method always returns `true`. It cannot be changed to `void`, because the return value is required by a parent interface, but there’s nothing useful we can do with this return value.

Unfortunately, this ability causes many mistakes: people often ignore the return value when it should not be ignored. Probably the most common kind of mistakes is assuming that you are dealing with a mutable object that updates itself, while in fact it’s immutable and returns the updated value. For example:

```
String s = "Hello World!";
// Attempt to remove "Hello " from the string
s.substring(6);
BigInteger value = BigInteger.TWO;
// Attempt to increment the value
value.add(BigInteger.ONE);
```

In these cases, it would be more correct to reassign the variable or create another one:

```
String s = "Hello World!";
s = s.substring(6);
BigInteger value = BigInteger.TWO;
BigInteger result = value.add(BigInteger.ONE);
```

Note that you can ignore the return value in lambdas and method references as well. For example, here the code author mistakenly tried to trim all the strings inside the list:

```
void trimAll(List<String> list) {
  list.forEach(String::trim);
}
```

The `forEach` method allows you to use all the collection elements but it does not allow you to update them. It accepts the `Consumer` functional interface, whose abstract method `accept` returns `void`, so the result of the `trim` call is silently ignored. It would be correct to use the `replaceAll` method here:

```
void trimAll(List<String> list) {
  list.replaceAll(String::trim);
}
```

Another class of mistakes arises from using the wrong method. Well, it’s possible to use the wrong method even if you use a return value, but the ability to ignore it makes this problem happen more often. For example, we have seen a mistake like this:

```
void processBitSets(BitSet s1, BitSet s2) {
  s1.intersects(s2);
  processIntersection(s1);
}
```

Here, the author wanted to create an intersection of two `BitSet` objects, retaining only those bits that are set in both `s1` and `s2`. The `BitSet` object is mutable, and there is indeed a method that performs the desired operation: `s1.and(s2)`. However, the author mistakenly used the `intersects()` method, which just tests whether the intersection is non-empty without updating the sets. Unlike `and()`, this method returns a `boolean` value, which was ignored. A similar mistake is possible if you want to call the `interrupt()` method of the `Thread` class but you mistakenly call `interrupted()` instead, probably due to selecting the wrong item in the code-completion popup.

In modern Java, it’s popular to design APIs in a fluent style where you can chain several calls together. It looks nice but sometimes people forget to make the final call that should do the actual thing. For example, consider the following unit-test that uses the AssertJ [Reference] library:

```
@Test
public void testCreateList() {
  List<String> list = createList();
  assertThat(!list.isEmpty());
}
```

The last line may look perfectly okay to developers who are not very experienced in using AssertJ. In fact, `assertThat` by itself asserts nothing. Instead, it returns an `Assert` object that should be checked via a successive call:

```
assertThat(!list.isEmpty()).isTrue();
```

A similar problem happens with Stream API. For example, here the developer wants to filter the list of strings, removing empty strings:

```
stringList.stream().filter(str -> !str.isEmpty());
```

We also saw attempts to sort the list like this:

```
stringList.stream().sorted();
```

Here, filtering and sorting is not actually performed, as the terminal operation is not called, and the result is not assigned. The correct code would look like this:

```
stringList = stringList.stream().filter(str -> !str.isEmpty())
        .collect(Collectors.toList());
stringList = stringList.stream().sorted()
        .collect(Collectors.toList());
```

Alternatively, if the original list is known to be mutable, one can use the `removeIf()` and `sort()` methods to modify the list in-place:

```
stringList.removeIf(String::isEmpty);
stringList.sort(null);
```

There are also methods that actually produce some side effect, and it looks like the return value is not important. Some developers may even not know that the method has non-void return value. Yet, in fact it can be important. For example, some old file handling APIs in Java return false to indicate failure instead of throwing an `IOException`. This includes some methods from the `java.io.File` class: `createNewFile`, `delete`, `mkdir`, `mkdirs`, `rename`, etc. Filesystem operations may fail for a variety of reasons, including lack of drive space, insufficient access rights, inability to reach the remote filesystem due to network failure and so on. Ignoring these errors may cause more problems later, and it will be hard to find the original cause.

The methods `InputStream.read` and `Reader.read`, which accept arrays, are also deceptive. For example, one might want to read the next 100 bytes of the stream and process them somehow:

```
void readPrefix(InputStream is) throws IOException {
  byte[] arr = new byte[100];
  is.read(arr);
  process(arr);
}
```

However, the `read` method is not obliged to fill the whole array. Depending on the stream source, it may fill only part of the array, even if the stream contains more. That's because, by specification, this method blocks at most once. Also, if the end of the stream is reached, the method will return `-1` and not write to an array at all. Without checking the return value of the `read` method, it's impossible to say how many bytes were actually read.

Static analysis

Static analyzers can report if the result of method is ignored while it should not be. IntelliJ IDEA has the inspection 'Result of method call ignored' that reports all the methods listed above. SonarLint has separate rules to check return value of methods without side effects ("S2201: Return values from functions without side effects should not be ignored") and to check the result of methods like `InputStream.read` ("S2674: The value returned from a stream read should be checked"). Error Prone analyzer has "CheckReturnValue" bug pattern that also reports some of such problems.

Unfortunately, it's hard to say in general whether the result of a specific method should be used or not. The IntelliJ inspection knows about standard library methods, as well as some other popular APIs like AssertJ. However, there are thousands of Java libraries, as well as your own project code, and inspection does not know in advance where it's safe to ignore the return value. You can set the inspection to report all non-library calls where the return value is ignored but be prepared to tons of false-positive reports.

You can help the static analyzer with annotations. There are annotations named `@CheckReturnValue` in different annotation packages to signal that the return value must be checked. Such annotations exist in javax.annotation (JSR-305 annotation package), and in com.google.errorprone.annotations (Error Prone static analyzer annotation package). They could be applied not only to individual methods, but also to classes, interfaces, and whole packages to signal that the return value must be checked for every method declared inside. There's also `@CanIgnoreReturnValue` annotation in Error Prone package that allows you to override the `@CheckReturnValue` effect. If you don't want to depend on these packages, IntelliJ IDEA allows you declaring an annotation named `@CheckReturnValue` in any package you want. It will also be recognized, though in this case only if applied to the methods directly.

There's another annotation in JetBrains annotations package, namely `@Contract`. If you mark your method with `@Contract(pure = true)`, IntelliJ IDEA will know that the method produces no side effect and will warn if the resulting value is not used. Sometimes, IntelliJ IDEA can infer this annotation automatically. You can help the inference procedure if you explicitly declare classes as final when it's not intended to extend them.

Ways to avoid this mistake:

- Avoid using old APIs that report errors via `boolean` return type. For file operations, use newer `java.nio.file.Files` API. E. g., the old way to create a directory `path`, along with parent directories is:

```
new File(path).mkdirs();
```

Since Java 7, it's preferred to use

```
Files.createDirectories(Paths.get(path));
```

This way, you'll get proper exceptions if something went wrong.

- As for `InputStream.read` method, since Java 11 there's an alternative `readNBytes`, which has better interface and may block several times until the requested number of bytes is read.
- When learning a new API, pay attention to the mutability of the objects. Both in JDK and in third-party libraries, modern APIs prefer immutable objects, which no method can modify, so the return result must always be used. In particular, this is a common practice in APIs that follow the functional programming style.
- Old-style Java APIs often use methods prefixed with `set-` to update the object state. Modern APIs prefer the `with-` prefix, which says that a new object is returned every time this method is called. If you call a method with the `with-` prefix, it's likely that you should use the result. For example, a HexFormat API, which appeared in Java 17 and allows you to present binary data in hexadecimal format, follows this convention. It's also an example of an immutable object:

```
static void printInHex(byte[] data, boolean uppercase) {
  HexFormat format = HexFormat.ofDelimiter(":");
  if (uppercase) {
    // HexFormat object is immutable,
    // don't forget to reassign 'format' when calling with- method
    format = format.withUpperCase();
  }
  System.out.println(format.formatHex(data));
}
```

- When designing a new API, avoid returning results in both the argument and return values. This may encourage users to ignore the part of the result returned in the return value. Instead, create a new object that contains all the pieces of the result and return it.

Immutability and performance

Sometimes, writing to a mutable object passed as method parameter is justified from a performance point of view. However, this approach makes the API harder to use. In modern Java, object allocation and garbage collection is significantly improved compared to the older versions, and now it's preferred to create a new object instead of modifying an existing one, unless you have a good reason. This is reflected in the new API method `readNBytes`, which appeared in Java 11. Aside from the `readNBytes(byte[] array, int offset, int length)`, which repeats the signature of the older `read` method, there's `readNBytes(int length)`, which just returns a new array. It's very convenient and it's quite hard to misuse this method, as if you ignore the return result, you'll simply get nothing. Of course, depending on the usage scenario, this method may require additional data copying which could be unacceptable in hot code paths. On the other hand, in many places additional overhead is negligible and by using this method you can simplify your code and avoid possible errors. Don't forget about this when designing new API methods.

2.12 Mistake #12. Not using newly created object

Java allows you to write an object creation expression (or `new` expression) without using its return value, like this:

```
new Object();
```

This could be useful under rare circumstances when the object constructor produces some side-effect like registering a newly created object somewhere. Sometimes, this also used to check whether the constructor throws an exception. For example, we observed that people use `new java.net.URI(uriString)` to check whether `uriString` is a syntactically valid URI. The constructor throws `URISyntaxException` when the supplied string violates the URI format, as specified in RFC 2396, so by catching this exception you may validate URIs supplied by a user.

Unfortunately, this ability also causes mistakes in the code when a programmer creates an object but forgets to assign it to the variable or return from the method. This does not cause compilation errors, and static analyzers won't always help you, as they might not be sure that not using the newly created object is not intended.

One issue that happens particularly often is creating an exception without throwing it:

```
void process(int value) {
  if (value < 0) {
    new IllegalArgumentException("Value is negative");
  }
  // use value
}
```

Here, an exception object is created and not thrown, so it will be just silently discarded, and the method will proceed with the negative `value` without validation.

Static analysis

Static analyzers usually report if an exception object was created and discarded. IntelliJ IDEA has “Throwable not thrown” inspection that reports this case. SonarLint reports “S3984: Exceptions should not be created without being thrown”.

Ways to avoid this mistake:

- Avoid any side-effects and complex validation in constructors. This way, dangling `new` expressions will be meaningless, and it will be easier to avoid them if you never need to write them on purpose.
- It’s a reasonable idea to prohibit not using the newly created object completely. Configure your linter or static analyzer to warn about any unused newly created object. This way, you’ll need to add an explicit suppression comment to every such an object creation expression, which will signal that this code is intended.

2.13 Mistake #13. Binding a method reference to the wrong method

Method references, introduced in Java 8, provide a compact way to specify functions in Java. Unlike lambdas, when you use method references, you don’t need to think up variable names to pass them from functional interface method to a target method. Compare, for example:

```
IntBinaryOperator op = Integer::sum;
IntBinaryOperator op = (a, b) -> Integer.sum(a, b);
```

When you are using lambda, you need to specify a parameter list for the called method. As a result, the call often becomes longer and more verbose. No wonder many developers like method references for their brevity.

However, this brevity may also add confusion. When you have several overloads of the method or the constructor, looking at the method reference you cannot easily say which one of them is called, even if all the overloads have a different number of parameters. If you don’t remember the exact declaration of the functional interface, you may end up binding to a completely different method or constructor.

Consider the following class, where you store textual content associated with some filenames. The clients may create new files or append more lines to existing files. So, you may decide to maintain a map that associates filenames with `StringBuilder` objects. The `add()` method allows adding a new line and the `dump()` method writes the current content to the standard output.

```
public class Storage {
  private final Map<String, StringBuilder> contents =
                  new TreeMap<>();

  void add(String fileName, String line) {
    contents.computeIfAbsent(fileName, StringBuilder::new)
            .append(line).append("\n");
  }

  void dump() {
    contents.forEach((fileName, content) -> {
      System.out.println("File name: " + fileName);
      System.out.println("Content:\n" + content);
    });
  }
}
```

Now, let’s try to use this class:

```
Storage storage = new Storage();
storage.add("users.txt", "admin");
storage.add("users.txt", "guest");
storage.add("numbers.txt", "3.1415");
storage.dump();
```

It's expected to get the following output:

```
File name: numbers.txt
Content:
3.1415

File name: users.txt
Content:
admin
guest
```

In fact, we get something different:

```
File name: numbers.txt
Content:
numbers.txt3.1415

File name: users.txt
Content:
users.txtadmin
guest
```

As you can see, the content starts with the key which is not what we wanted. Let’s take a closer look at the `add()` method:

```
contents.computeIfAbsent(fileName, StringBuilder::new)
```

Here, we create a new entry in the map if it wasn’t created before. We’ve used a method reference as a factory for new entries. However, which `StringBuilder` constructor are we calling here? If we convert the method reference to lambda (you can use an automated quick fix provided by your IDE), you’ll get the following:

```
contents.computeIfAbsent(fileName, s -> new StringBuilder(s))
```

Now, it's clearer. The function passed to `computeIfAbsent()` has an argument which is the key passed as the first parameter. It's easy to forget about this, as usually you don't need this argument and, if you do need it, you can capture the original `fileName` variable instead. In lambda, you can just ignore this argument. However, when a method reference is used, the compiler looks for a constructor that accepts a string. As there's actually such a constructor in `StringBuilder`, the compilation finishes successfully but the newly created `StringBuilder` will not be empty.

It would be correct to use a lambda and the default constructor here. A method reference is not suitable in this place at all:

```
contents.computeIfAbsent(fileName, s -> new StringBuilder())
```

Aside from overloads, variable arity methods may also cause unexpected behavior. In one project, we have a utility method named `newHashSet` that creates a `HashSet` and prepopulates it from the supplied arguments. The method is declared in the following way:

```
class Utils {
  @SafeVarargs
  public static <T> HashSet<T> newHashSet(T... elements) {
    return new HashSet<>(Arrays.asList(elements));
  }
}
```

Note that this method can be used without arguments at all. In this case, an empty `HashSet` will be created.

In another part of the code, we populated the following multi-map that contains objects of some `Item` type:

```
Map<Item, Set<Item>> map = new HashMap<>();
```

The addition of a new element looks pretty straightforward:

```
map.computeIfAbsent(key, Utils::newHashSet).add(value);
```

Here, we have the same problem. We want to add a new value to the existing set or create an empty set if it doesn't exist for a given key. However, an argument is mistakenly passed to `newHashSet`, and as a result, the key `Item` was also always added to the set.

A similar surprise is possible with the `Arrays.setAll()` method. For example, suppose you want to create an array containing `StringBuilder` objects for subsequent filling. It may seem like a good idea to use the `setAll()` method and a method reference:

```
StringBuilder[] array = new StringBuilder[size];
Arrays.setAll(array, StringBuilder::new);
```

For small values of the size variable, this may even work as expected. But as the size grows bigger, it may become unbearably slow and it could even end up with `OutOfMemoryError`. If you replace the method reference with a lambda, you'll see the following:

```
StringBuilder[] array = new StringBuilder[size];
Arrays.setAll(array, capacity -> new StringBuilder(capacity));
```

A function passed to `setAll()` accepts an index of the current element. It's unnecessary for our purposes, but the compiler selects a constructor that accepts an initial capacity of `StringBuilder`. For small indexes this may go unnoticed, but for bigger ones you will pre-allocate much more memory than necessary. Figure 2.9 is an illustration of how the memory layout will look like after the `setAll()` call.

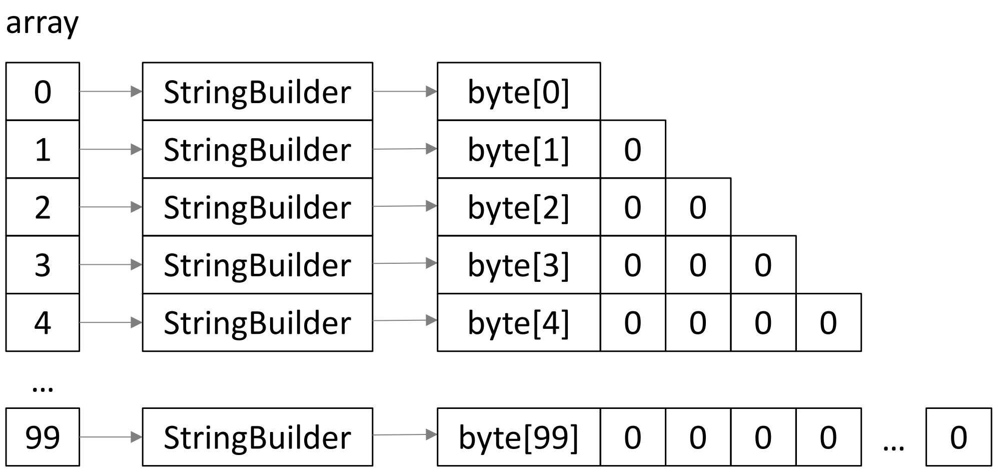


Figure 2.9 Final memory layout after invoking Arrays.setAll for size = 100. The internal byte[] buffer held by StringBuilder becomes bigger and bigger towards the end of the array.

Ways to avoid this mistake:

- Use method references carefully when pointing to a method or constructor that has overloads or variable arity parameters. You may accidentally link to a wrong method or pass more arguments than intended. When in doubt, use lambda. Some developers avoid method references completely and use lambdas always. You can configure your linter or static analyzer to highlight all the method references. Unfortunately, I doubt that any static analyzer can highlight only incorrectly bound method references as it cannot know whether this behavior is intended.
- In some cases, it's easy to forget that a functional argument accepts a parameter. The `Map.computeIfAbsent()` and `Arrays.setAll()` are such unfortunate methods, so you should be especially careful when using method references with them.

2.14 Mistake #14. Using the wrong method in a method reference

It’s easy to write a method reference to map to a particular functional interface. This ease may play a bad joke on you, as you may mistakenly use a completely wrong method. Probably the most amusing example we’ve seen involves mapping the `Integer.max()` or `Integer.min()` static methods to a `Comparator` interface. Consider the following code:

```
List<Integer> list = Arrays.asList(0, -3, -2, 3, -1, 1, 2);
list.sort(Integer::max);
System.out.println(list);
```

Here, the author wanted simply to sort the list of integers. As the `sort()` method requires a comparator, the author thought that `Integer::max` would work as one, as the method name seems to suggest that it will compare two numbers and select the bigger of them. This code compiles but produces a totally incorrect result:

```
[0, -1, -2, -3, 3, 1, 2]
```

The result might be different if the sorting algorithm in the standard Java library changes in the future. But the thing is that it doesn’t sort anything. For `Comparator`, you need to implement a single `compare()` method that accepts two `Integer` values and returns an `int` value whose sign shows which of the input values is larger. The method `Integer.max()` accepts two `int` values (Java performs the automatic unboxing conversion for you) and returns `int`, so from the Java point of view, it fits the `Comparator` interface. However, the sign of the returned value doesn’t show which of the inputs is bigger. It’s just the sign of the bigger number itself. So the sorting works incorrectly.

More pitfalls connected to the comparators you may find in Chapter 7.

Ways to avoid this mistake:

- Pay attention to static analysis messages. IntelliJ IDEA reports “Invalid method reference used for 'Comparator'” warning when `Integer::min` or `Integer::max` is used for comparator.
- Write unit tests, even if the code looks evident. Here, one may think that list sorting is a straightforward operation that cannot be broken. However, as we see, this is not always the case.

2.15 Summary

- Priority among mathematical operators is not always intuitive. In particular, bitwise and logical operators can be tricky. Use parentheses every time you are not sure about precedence.
- The plus sign has different meanings when applied to numbers and when applied to strings. In some cases, it's possible to mix them. Try to avoid mixing addition and concatenation in single expressions and ban unary plus in your projects.
- When creating a long string using a series of concatenations, it's possible to erroneously apply a subsequent method call to the last fragment only instead of the whole string. Prefer text block syntax, available in newer Java versions.
- Conditional expressions have non-trivial and counter-intuitive type conversion rules when their branches have expressions of different types. Prefer an `if`-statement when in doubt.
- Avoid boxed types like `Integer`, `Double`, or `Boolean` as much as possible. They may cause implicit conversions, which will not only cause performance overhead but also may produce a `NullPointerException` if null is accidentally stored there.
- While variable arity methods are convenient to use, they add ambiguity when exactly one argument is passed at a variable arity position. Sometimes, you may have an array or collection accidentally wrapped into another array.
- Java allows you to ignore the result of a method call or `new` expression. While sometimes this is intended, often it causes unpleasant mistakes. Remember that many classes are immutable and produce a new object instead of modifying themselves. Also, be sure to add a `throw` word before a newly created exception.
- Method references provide neat and concise syntax to create a functional expression. This conciseness may play a bad joke when the method or constructor is overloaded, as the compiler may bind to the wrong method. When in doubt, prefer lambda syntax.

3 Program structure

This chapter covers

- **Mistakes that often happen when using if-else chains**
- **Problems using `while` and `for` loops**
- **How to avoid the pitfalls around the initialization of fields and classes**
- **Missing call to a superclass method**
- **What happens if you accidentally declare a static field instead of instance one**

Java provides a number of constructions to alter the program control flow. This includes branching statements (switch, if-else), loop statements (for, while) and so on. Programmers tend to make many mistakes in control flow statements that may result in visiting the wrong branch, not visiting the desired branch, having more loop iterations than expected, and so on.

Another class of mistakes discussed in this chapter comes from the errors in program structure outside of the method body, like incorrect initialization order or accidental use of the `static` modifier where it should not be used.

3.1 Mistake #15. Malformed if-else chain

One of the common patterns in Java programming is an if-else chain that allows you to perform different actions depending on multiple conditions, like this:

```
if (condition1) {
  …
}
else if (condition2) {
  …
}
else if (condition3) {
  …
}
else if (condition4) {
  …
}
else {
  …
}
```

When this chain is quite long, it sometimes happens that one of the intermediate `else` keywords are missing:

```
if (condition1) {
  …
}
else if (condition2) {
  …
}
if (condition3) {
  …
}
else if (condition4) {
  …
}
else {
  …
}
```

This would be no problem if there were no final else block, and all the conditions were mutually exclusive:

```
if (input == 1) {…}
else if (input == 2) {…}
else if (input == 3) {…}
else if (input == 4) {…}
```

In this case, two conditions cannot be equal at the same time, so the behavior of the program is not affected if we remove some or all of the `else` keywords. However, this is not always the case. Sometimes, conditions are ordered from more specific to more general, but the program logic assumes that if more specific condition is satisfied then more general one should be ignored. An unconditional else branch at the end is the partial case of this scenario: it's the most general behavior that should be executed if no specific conditions were satisfied.

When this kind of mistake is present, two branches could be executed. In the example above, if `condition1` or `condition2` is true, then the final, unconditional `else` will be executed as well. This may be unnoticed during testing if only `condition3` and `condition4` were covered. Unfortunately, static analyzers cannot always report this problem, because quite often two independent if chains are actually intended.

It’s possible to make code less error prone if you can shape your if-chain so that every branch does nothing except assign the same local variable (or several variables). Let’s see how it can be done. Consider the following `if`-chain:

```
if (condition1) {
  process(getData1());
} else if (condition2) {
  process(getData2());
} else if (condition3) {
  process(getData3());
} else if (condition4) {
  process(getData4());
} else {
  process(getDefaultData());
}
```

Here, we call the same `process` method with different argument. It would be better to introduce the intermediate variable for the argument and move the `process()` call out of the branches:

```
Data data;
if (condition1) {
  data = getData1();
} else if (condition2) {
  data = getData2();
} else if (condition3) {
  data = getData3();
} else if (condition4) {
  data = getData4();
} else {
  data = getDefaultData();
}
process(data);
```

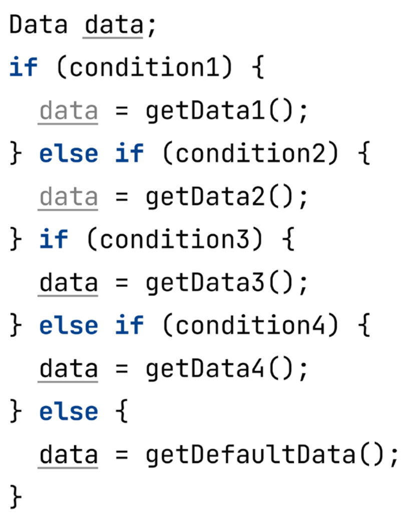
Now, the code has more lines, but it’s more error-proof. For example, let’s skip one of `else` keywords:

```
Data data;
if (condition1) {
  // static analysis warning: result of assignment is unused
  data = getData1();
} else if (condition2) {
  // static analysis warning: result of assignment is unused
  data = getData2();
} if (condition3) {
  data = getData3();
} else if (condition4) {
  data = getData4();
} else {
  data = getDefaultData();
}
process(data);
```

You can go further and declare the local variable as final to indicate that you intend to assign it only once. Now, the absence of `else` will be immediately noticed by the compiler:

```
final Data data;
if (condition1) {
  data = getData1();
} else if (condition2) {
  data = getData2();
} if (condition3) {
  // compilation error: data is reassigned
  data = getData3();
} else if (condition4) {
  data = getData4();
} else {
  data = getDefaultData();
}
process(data);
```

I personally don’t like declaring local variables as `final` because this adds too much of visual noise to the code. However, I must admit that it helps with catching errors like this, and I know developers who consistently use `final` on locals for extra safety. Alternatively, you can rely on an IDE. For example, IntelliJ IDEA underlines reassigned variables by default (Figure 3.1). So, you may pay attention to this: if the variable is underlined but you didn’t want to reassign it, then something is probably wrong.



```
Data data;
if (condition1) {
  data = getData1();
} else if (condition2) {
  data = getData2();
} if (condition3) {
  data = getData3();
} else if (condition4) {
  data = getData4();
} else {
  data = getDefaultData();
}
```

Figure 3.1 A chain of conditions in IntelliJ IDEA editor with missing else. Note that the data variable is underlined, which means that it’s assigned more than once, so it cannot be declared final. Additionally, IntelliJ IDEA static analyzer highlights data in gray color in the first two branches to indicate that these assignments are always overwritten.

As an alternative, you can extract the if-chain to a separate method, assuming that it has enough information to evaluate the conditions:

```
Data getData() {
  if (condition1) {
    return getData1();
  } else if (condition2) {
    return getData2();
  } else if (condition3) {
    return getData3();
  } else if (condition4) {
    return getData4();
  } else {
    return getDefaultData();
  }
}
```

Now the original method is simplified to:

```
process(getData());
```

When using `return` in branches, you are free to keep or remove `else` keywords. They don't change the program behavior:

```
Data getData() {
  if (condition1) {
    return getData1();
  }
  if (condition2) {
    return getData2();
  }
  if (condition3) {
    return getData3();
  }
  if (condition4) {
    return getData4();
  }
  return getDefaultData();
}
```

Compared to the original code, this shape is much better. In particular, now you have separated the side-effect code (`process()` call) and side-effect free code (computing the data). It's much easier to test the side-effect-free method `getData()`, so it could be tested separately. Also, it could be reused.

Ways to avoid this mistake:

- Format the code so that `else if` is placed on the same line as the closing brace of the previous branch. In this case, the problem will be more eye-catching after reformatting:

```
if (condition1) {
  …
} else if (condition2) {
  …
}
if (condition3) {
  …
} else if (condition4) {
  …
} else {
  …
}
```

- Cover all the conditional branches with unit tests.
- Try to keep if-branches in one of two shapes: either return something or assign the same variable. In the first case, `else` is not required at all. In the second case, if you miss the `else` keyword, you will get a static analyzer warning that the result of the assignment is unused, or get a compilation error if the variable is declared as final.

3.2 Mistake #16. Condition dominated by a previous condition

Another problem that can arise from a series of `if-else` conditions occurs when the conditions are not mutually exclusive. In this case, the order of conditions affects the program behavior. If two or more conditions can be true at the same time, only the first branch will be executed, so it might be important which branch is placed the first.

In a more specific scenario, one condition dominates another one. We say that condition X dominates the condition Y if Y can be true only when X is also true. In this case, `if (Y) {…} else if (X) {…}` is a reasonable code pattern while `if (X) {…} else if (Y) {…}` is clearly a mistake: the last branch is never taken.

Simple cases of condition dominance include numeric ranges and `instanceof` checks. For example, consider the following code that determines the tariff plan based on the customer age:

```
enum Plan {CHILD, FULL, FREE}
Plan getPlan(int age) {
  if (age >= 6) return Plan.CHILD;
  if (age >= 18) return Plan.FULL;
  return Plan.FREE;
}
```

The idea was to assign the `CHILD` plan for customers aged between 6 and 17 (inclusive). However, the order of conditions is wrong: `age >= 6` dominates over `age >= 18`. As a result, `age >= 18` is never true and the `FULL` plan is never returned.

The following method (adapted from a real production code base) to convert an object to a `BigInteger` is an example of condition dominance involving `instanceof` checks:

```
BigInteger toBigInteger(Object obj) {
  if (obj instanceof Number)
    return BigInteger.valueOf(((Number) obj).longValue());
  if (obj instanceof BigInteger)
    return (BigInteger) obj;
  if (obj instanceof BigDecimal)
    return ((BigDecimal) obj).toBigInteger();
  return BigInteger.ZERO;
}
```

The idea was to convert any number (`Long`, `Double`, `Integer`, etc.) via the intermediate `long` value but apply a better algorithm for `BigInteger` and `BigDecimal` inputs. However, as `BigInteger` and `BigDecimal` classes also extend the `Number` abstract class, such inputs also fall into the first condition making the following ones useless.

Mistakenly ordered `instanceof` checks are harder to spot, as the type hierarchy could be complex and not completely known by a code writer or code reader.

Static analysis

Data flow based static analysis like “Constant values” inspection in IntelliJ IDEA is capable of detecting many dominating conditions. In both samples above, it reports that the subsequent conditions are always false. PVS-Studio has a similar warning “V6007. Expression is always true/false”.

Some static analyzers can detect even more complex dominance conditions. For example, consider the following erroneous implementation of a classic FizzBuzz problem:

```
for (int i = 1; i < 100; i++) {
  if (i % 3 == 0) System.out.println("Fizz");
  else if (i % 5 == 0) System.out.println("Buzz");
  else if (i % 15 == 0) System.out.println("FizzBuzz");
  else System.out.println(i);
}
```

The idea is to print “FizzBuzz” when `i` is divisible by 15. However, `i % 15 == 0` condition is dominated by previous ones, so it’s unreachable. IntelliJ IDEA analyzer is capable to detect this case.

Still, even an advanced analyzer cannot detect all the cases. For example, most common Java static analyzers don’t warn in the following code that parses command line arguments:

```
for (String arg : args) {
  if (arg.startsWith("--data")) {
    setData(arg.substring("--data".length()));
  } else if (arg.startsWith("--database")) {
    setDataBase(arg.substring("--database".length()));
  } else { … }
}
```

As any string that starts with “–database” also starts with “–data”, the second condition is unreachable. In theory, it’s possible to detect this statically but in IntelliJ IDEA it’s not implemented as of this writing, so you cannot rely on static analysis completely. In any case, it’s always possible to have more complex conditions (e. g., involving custom method calls from your project) where static analyzer cannot do anything.

It should also be noted that even incorrect `instanceof` chains may not be reported by static analyzers, as type hierarchies do not always provide all the necessary information. For example, consider the following method that processes collections and has optimized implementations for random access lists and other lists:

```
void process(Collection<?> data) {
  if (data instanceof List) {
    processList((List<?>) data);
  } else if (data instanceof RandomAccess) {
    processRandomAccessList((RandomAccess) data);
  } else {
    processAnyCollection(data);
  }
}
```

As the `RandomAccess` interface doesn't extend the `List` interface, you may implement it without implementing the `List`. Thus, it's theoretically possible that the second condition will be true and static analyzers in general will be silent here. However, as stated in the documentation, `RandomAccess` is a marker interface for `List` implementations, so correct implementations of `RandomAccess` also implement `List`, and they will never reach the `processRandomAccessList` call. Arguably, this was a mistake in JDK implementation: `RandomAccess` should be declared as implementing `List`. Do not make the same mistake in your code: always declare intended superinterfaces.

Ways to avoid this mistake:

- When you have a series of if-else branches, check the test coverage report and make sure that all the branches are covered with tests. If you see that a branch is not covered, write a test to cover it. If you have a condition dominance problem, you will quickly notice it.
- Try to keep conditions mutually exclusive. Particularly for numeric ranges, specify both range bounds. While this makes the code more verbose, it's also more explicit and more flexible, as reordering the conditions does not affect the behavior. For example, the `getPlan()` method could be rewritten in the following way:

```
Plan getPlan(int age) {
  if (age >= 0 && age <= 5) return Plan.FREE;
  if (age >= 6 && age <= 17) return Plan.CHILD;
  if (age >= 18) return Plan.FULL;
  throw new IllegalArgumentException("Wrong age: " + age);
}
```

We also specified the complete range for the `FREE` plan which makes the code even more robust. Now, it throws an exception if a negative age is passed to the method. Also, an exception will be thrown if we mistakenly specified the wrong bounds, skipping some ages. Suppose that we introduce a new tariff plan for young people aged between 18 and 24. Now, the `FULL` plan is applicable for ages 25+. Now, imagine that we fixed the condition for the `FULL` plan but forgot to add a condition for a new plan:

```
Plan getPlan(int age) {
  if (age >= 0 && age <= 5) return Plan.FREE;
  if (age >= 6 && age <= 17) return Plan.CHILD;
  if (age >= 25) return Plan.FULL;
  throw new IllegalArgumentException("Wrong age: " + age);
}
```

Thanks to our `throw` statement, the problem will immediately show itself when the system is tested with a person aged between 18 and 24. The exception is much easier to spot than the wrong plan silently applied, so it's much more likely that the problem will be discovered during early testing.

3.3 Mistake #17. Accidental pass-through in switch statement

This problem is tightly connected with the previous one. Due to unfortunate historical reasons, the switch operator in Java falls through into the next branch by default, so you have to explicitly use the `break` statement to exit the `switch`:

```
void performAction(Button button) {
  switch (button) {
    case YES:
      actOnYes();
      break;
    case NO:
      actOnNo();
      break;
    case CANCEL:
      actOnCancel();
      break;
  }
}
```

If you accidentally miss one of the `break` statements, two actions will be performed.

Fortunately, this problem was addressed in Java 14: now there's an alternative arrow syntax for switch branches, which doesn't allow pass-through behavior at all:

```
void performAction(Button button) {
  switch (button) {
    case YES -> actOnYes();
    case NO -> actOnNo();
    case CANCEL -> actOnCancel();
  }
}
```

Static analysis

Static analyzer may warn about pass-through switch branches. For example, SonarLint has a rule “S128: Switch cases should end with an unconditional ‘break’ statement”, which warns by default. If you use IntelliJ IDEA, check the inspection “Fallthrough in 'switch' statement” which might be turned off. There’s a convention between static analyzers to suppress the warning if pass-through is marked with a comment like `// fallthrough`:

```
switch (button) {
 case YES:
  actOnYes();
  // fallthrough
 case NO:
  actOnYesAndNo();
  break;
 case CANCEL:
  actOnCancel();
  break;
}
```

Here, the `// fallthrough` comment explicitly indicates the intent to fall into the next case. Static analyzers recognize this comment and do not warn anymore. The approach to explicitly mark all the intended fall through switch branches with a comment is supported by Google Java Style Guide.

Ways to avoid this mistake:

- Always prefer the new arrow syntax if you can use Java 14 or newer.
- Try to keep switch branches in one of two shapes: either return something or assign the same variable. In case of return, you don’t need to add a break statement at all. In case of assignment, you can declare the variable final and accidental fall through will cause a compilation error as the variable will be assigned twice:

```
final Boolean answer;
switch (ch) {
    case 'T':
    case 't':
        answer = true;
        // 'break' accidentally omitted
    case 'F':
    case 'f':
        // compilation error: answer is reassigned
        answer = false;
        break;
    default:
        answer = null;
}
```

When migrating to Java 14 or newer, such switch statements could be easily transformed to switch expressions which will make the code even more clear and robust.

3.4 Mistake #18. Malformed classic for loop

The classic `for` loop in Java was inherited from C. Although it's flexible, this construct is pretty dangerous and becomes a source of many subtle bugs. Usually it appears in the idiomatic way, when it is necessary to iterate over the range of numbers:

```
for (int i = from; i < to; i++) {
  // body
}
```

One of the problems is that the same variable must be mentioned three times there. So, it's quite easy to mix the variables, especially if you have several nested loops. Erroneous code may look like this:

```
for (int i = 0; i < 10; i++) {
  for (int j = 0; j < 10; i++) {
    // use i and j
  }
}
```

The inner loop increments the variable of the outer loop. As a result, the inner loop never finishes. Another example of an infinite loop is like this:

```
for (int i = 0; i < 10; i++) {
  for (int j = 0; i < 10; j++) {
    // use i and j
  }
}
```

Here, we increment the correct variable but we have the wrong condition. While infinite loops are usually easy to detect during testing, things could be more complicated if the inner loop has the break statement on some condition:

```
for (int i = 0; i < 10; i++) {
  for (int j = 0; j < 10; i++) {
    // use i and j
    if (condition()) {
      break;
    }
  }
}
```

In this case, the loop can finish via the condition and sometimes the program may look like it's working correctly but it's still wrong. It's also possible to have a typo in both the condition and the update clauses:

```
for (int i = 0; i < 10; i++) {
  for (int j = 0; i < 10; i++) {
    // use i and j
  }
}
```

Now the loop finishes but of course not all possible combinations of `i` and `j` are processed. Such bugs could lurk in code bases for years, e. g. in validation or normalization loops. For instance, imagine that we want to replace all negative values in a square array with zero. The code may look like this:

```
void normalizeData(int[][] data) {
  int size = data.length;
  for (int i = 0; i < size; i++) {
    for (int j = 0; i < size; i++) {
      if (data[i][j] < 0) {
        data[i][j] = 0;
      }
    }
  }
}
```

It will require very careful review to notice that the inner loop has the wrong condition and update expressions. Also, this works for some inputs. If negative numbers rarely appear, this bug could slip through quality assurance unnoticed and go into the production code.

Static analysis

Dataflow-based static analysis tools can easily detect infinite loops like the ones listed above. For example, IntelliJ IDEA highlights loop conditions and issues warnings like ‘Condition is always true’ if only the condition or the update expression are wrong. Unfortunately, this will not help if both of them are mistaken.

Some static analysis tools provide inspections to report non-idiomatic loops in various ways. One possibility is to report when the loop variable is modified inside the loop body. If you use this kind of inspection the `normalizeData` example will be reported, as the inner loop modifies the loop variable of the outer loop. Unfortunately, such inspections produce many undesired warnings in legitimate cases, so to use them efficiently you either need a very strict code style or you’ll have to suppress many warnings. In IntelliJ IDEA, such an inspection is called “Assignment to 'for' loop parameter” and it’s turned off by default. SonarLint provides the rule “S127: ’for’ loop stop conditions should be invariant”. Some malformed for loops can be caught by the PVS-Studio warning “V6031. The variable 'X' is being used for this loop and for the outer loop”.

Ways to avoid this mistake:

- Use code templates provided by your IDE to type idiomatic range iteration loops. For example, IntelliJ IDEA has the 'fori' live template. When you type 'fori' and press Enter the whole loop is generated. After that, you can change the variable name and it will be properly updated in all the places. Eclipse has a similar template to iterate over an array. Instead of typing the `for` loop manually, type 'for' and press Ctrl+Space to invoke the code completion popup. Select "for – use index on array" option to generate the for loop automatically.
- When testing the looping code, don't forget to cover the cases in which both inner and outer loops have at least two iterations. In `normalizeData` sample a unit-test will catch a mistake if it passes an array like `{{-1, -1}, {-1, -1}}` and ensures that it's normalized.

3.5 Mistake #19. Not using the loop variable

Sometimes, developers write the loop correctly but forget to use the loop variable in the body, using a constant value instead. An example of such use was discovered by the PVS-Studio team in the Apache Dubbo project. Here, a classic code pattern was written which maps the content of one array to another:

```
String[] types = …
Class<?>[] parameterTypes = new Class<?>[types.length];
for (int i = 0; i < types.length; i++) {
  parameterTypes[i] = ReflectUtils.forName(types[0]);
}
```

Unfortunately, `types[0]` was mistakenly written instead of `types[i]`. This code might work correctly if there's only one element in the `types` array, or if all the elements are the same, so it may bypass simple tests. Also, simple static analysis diagnostics like "variable is not used" do not help here, as the 'i' variable is in fact used in the loop body.

Static analysis

PVS-Studio analyzer has a heuristic diagnostic "V6016. Suspicious access to element by a constant index inside a loop". It may produce false-positive reports, as, in rare cases, using a constant index might be intended. Nevertheless, it's a good idea to check these reports.

Ways to avoid this mistake:

- Consider using `Arrays.setAll()` to fill an array:

```
Arrays.setAll(parameterTypes, i -> ReflectUtils.forName(i));
```

In this case, the array index is available as a lambda parameter, so if you don't use it, it will be much more visible.

- Try to avoid using indices at all. In many cases it's possible to rewrite code using higher-level constructs, like stream API. For example, here one may write:

```
Class<?>[] parameterTypes = Arrays.stream(types)
     .map(ReflectUtils::forName)
     .toArray(Class<?>[]::new);
```

  Now all the indices are encapsulated inside the stream API implementations and don't appear in the client code. Using stream API might have additional costs but in most cases it's insignificant.

3.6 Mistake #20. Wrong loop direction

Sometimes, it's necessary to modify the classic `for (int i = lo; i < hi; i++)` pattern. We saw an example of such a modification in the previous section when we iterated the collection in backwards direction. Care should be taken to avoid an off-by-one error. The forward loop visits all the numbers starting from `lo` (including) and finishing just below `hi` (excluding `hi` itself). If you want to iterate the same numbers in backwards direction, you should write

```
for (int i = hi - 1; i >= lo; i--)
```

One of the mistakes that may occur when you write the backwards direction loop, or changing the loop direction manually is to forget changing `i++` to `i--`:

```
for (int i = hi - 1; i >= lo; i++)
```

Luckily, this problem could be detected statically. Also, off-by-one errors are possible in the initial value (using `hi` instead of `hi - 1`) or bound (using `i > lo` instead of `i >= lo`). These problems may not be detected by static analyzers, as such initial values and bounds could be intended.

Static analysis

IntelliJ IDEA has an inspection "Loop executes zero or billions of times" that reports a warning if there's a mismatch between variable update direction and bound condition, like `i >= lo; i++`. Similarly, SonarLint reports "S2251: A 'for' loop update clause should move the counter in the right direction" warning. It's very unlikely if such a combination is intended, as it will be executed either zero times (if the bound condition is false since the beginning), or billions of times, until loop variable overflow occurs.

Ways to avoid this mistake:

- IDEs may provide an action to reverse the direction of an existing loop. For example, IntelliJ IDEA has an intention action "Reverse direction of for loop" available on the loop declaration. It could be more robust to write an ascending iteration loop and then use an IDE action to convert it to the descending one.

- If you want to iterate over an array or List in reverse direction, IDEs may provide code templates to do this. For example, IntelliJ IDEA has a postfix template called 'forr' to create a reverse loop over an array or List. For example, given an array named `arr`, one may type `arr.forr` to generate a loop declaration:

```
for (int i = arr.length - 1; i >= 0; i--) {

}
```

3.7 Mistake #20. Loop overflow

Sometimes it's desired to include the upper bound of a loop, and the commonly used pattern is `for (int i = lo; i <= hi; i++)`. Note the less-than-or-equal operator instead of a simple less-than in the loop condition. In most cases, this kind of loop works fine, but you should ask yourself whether `hi` could be equal to `Integer.MAX_VALUE` (or `Long.MAX_VALUE` if the loop variable has a long type). In this rare case, the loop condition will become always true, and the loop will never terminate, as the loop counter will silently overflow and continue with negative numbers. This case might sound unrealistic, but I actually had this bug in production code that resulted in an infinite loop. The problem could be fixed in several ways, e. g. by adding an additional condition:

```
for (int i = lo; i >= lo && i <= hi; i++)
```

Ways to avoid this mistake:

- While the problem is extremely rare, pay attention to less-than-or-equal iteration loop conditions and ask yourself whether the bound could be `MAX_VALUE`. Cover this corner case with a test if possible.
- Consider using Stream API instead of loops, like `IntStream.rangeClosed(lo, hi)`. This is much more concise and the `MAX_VALUE` corner case is properly handled inside the `rangeClosed` implementation, so you don't need to care about it. Unfortunately, using Stream API may add performance overhead, so use it with care in performance-critical code.

We will cover more overflow-related problems in Chapter 4.

3.8 Mistake #21. Idempotent loop body

An idempotent operation is an operation that produces a side-effect, but it doesn't change anything if performed several times subsequently. For example, a simple assignment is idempotent:

```
x = 0;
```

This statement has a side effect: it changes the value of the `x` variable. However, it doesn't matter if you perform this operation once, twice, or a hundred of times: the result will be the same. On the contrary, the following operation is not idempotent:

```
x += 10;
```

It updates the value of x, using its previous value, so the result will be different depending on how many times this operation is performed.

Here's a simple rule: all the code executed during a loop iteration must not be idempotent. By 'all the code' we mean the loop body, the loop condition, and the update statement, if we are speaking about the for-loop. When this rule doesn't hold, the loop likely has a bug. To understand this, let's consider a while loop:

```
while (condition) {
    body;
}
```

Let's assume that the condition has no side-effect, and the body is idempotent. In this case, we have three options:

1. The condition is false initially: the loop is not executed at all.
2. The condition is true initially, but the body makes it false: the loop is executed only once.
3. The condition is true initially and the body doesn't change it: the loop is infinite and the program hangs.

If 3 never occurs in practice, then the loop is completely redundant and could be rewritten using an if statement. Here's a simple example:

```
while (x < 0) {
    x = 0;
}
```

Here, if x was a negative number initially, it will be changed to 0 at the first iteration. Otherwise, x will be unchanged. In no case, will the loop continue to the second iteration, so technically this is not a loop at all and can be replaced with a conditional statement:

```
if (x < 0) {
    x = 0;
}
```

Of course, you should be careful when fixing such an error. It's also possible that the loop was intended but something should be changed in its body.

It's more interesting when case 3 actually occurs. Let's consider the following code snippet, which is very similar to one I have actually seen in production code:

```
static String makeUniqueId(String id, Set<String> usedIds) {
    int i = 1;
    String uniqueId = id;
    while (usedIds.contains(uniqueId)) {
        uniqueId = id + "_" + i;
    }
    return uniqueId;
}
```

This method is intended to generate a unique identifier based on the proposed one (passed as id parameter). If the proposed one is not used yet (is not contained in usedIds set) then we use it. Otherwise, we try to add a suffix like _1 to the identifier until we find an unused one.

If you look more closely at this code, you will realize that the loop body is idempotent and the condition has no side effect, so we have a problem. The loop body assigns a variable `uniqueId`, based on the values of `id` and `i`, but `id` and `i` are never changed inside the loop. So we never try suffixes like _2 or _3. Instead, we end up in an infinite loop trying the same suffix again and again, as shown in figure 3.2.

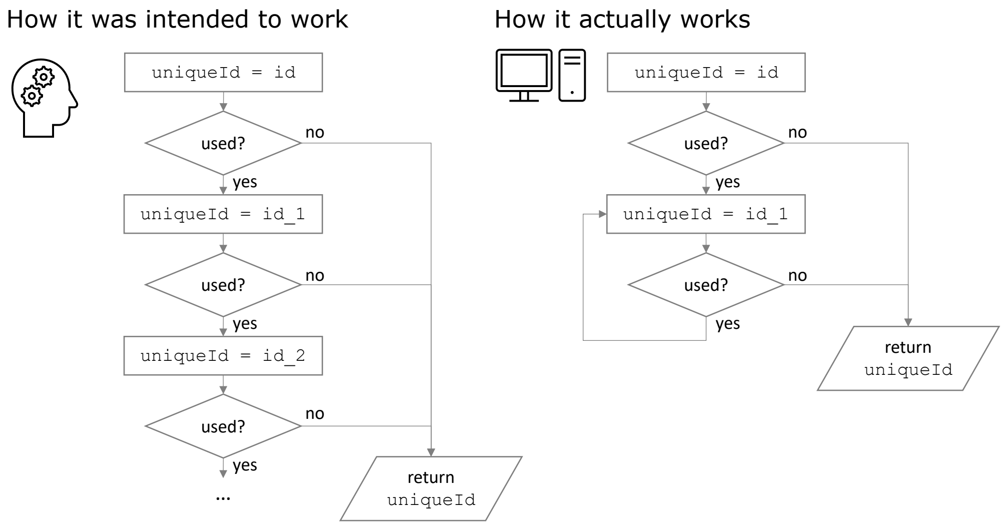


Figure 3.2 It was intended to find the unused string for id. In fact, after first attempt, it falls into the infinite loop.

This mistake is dangerous because it may easily slip through unit testing. The developer could test this method using the following assertions:

```
assertEquals("id", makeUniqueId("id", Set.of()));
assertEquals("id_1", makeUniqueId("id", Set.of("id")));
```

This works fine, so the developer might think that the method is implemented properly. A code coverage tool will also show that these assertions cover all the instructions and branches of the method. Yet, the method implementation is wrong, and these tests are not enough. The following call ends up in an infinite loop:

```
makeUniqueId("id", Set.of("id", "id_1"));
```

Once the problem is identified, the fix is simple: just add `i++` to the loop body.

Static analysis

IntelliJ IDEA has "Idempotent loop body" inspection that can detect simple cases like the ones shown above. Unfortunately, it may not report if interprocedural analysis is necessary to prove the loop body idempotence.

Sometimes a loop body becomes idempotent only under some circumstances, so more "normal" loop iterations may happen when the program state changes.

Suppose that you need a method to analyze a format string similar to one passed to `printf()`. The method must return true if the format string contains the `"%d"` format specifier. However, you cannot simply use `str.contains("%d")`, as it's also possible to escape a `"%"` symbol itself, duplicating it, so, for example, for the input `"%%d"` we should return false, but for `"%%%d"` input we should return true. One may come up with the following implementation:

```
static boolean hasPercentD(String str) {
  int pos = 0;
  while (true) {
    pos = str.indexOf('%', pos);
    if (pos == -1 || pos == str.length() - 1) return false;
    char c = str.charAt(pos + 1);
    if (c == 'd') return true;
    if (c == '%') {
      pos += 2;
    }
  }
}
```

Here, two argument versions of `indexOf()` method are used, so we can continue to search from the previous place. We look for a percent sign. If it's followed by 'd', we immediately return true and if it's followed by another percent sign, we skip it and continue searching. This implementation correctly handles any combination of `"%%"` and `"%d"` in the single string, so one may write quite a few successful unit tests. For example, the following tests pass:

```
assertTrue(hasPercentD("Hello %d!"));
assertFalse(hasPercentD("%%d!"));
assertFalse(hasPercentD("%%%%"));
assertFalse(hasPercentD("%%%%%"));
assertFalse(hasPercentD("%%d%%d"));
assertTrue(hasPercentD("%%d%%%d"));
```

Moreover, if you count code coverage with JaCoCo, you may happily note that these tests cover 100% of the `hasPercentD()` method lines or instructions. Even mutation coverage with Pitest reports that 100% of the mutations are killed. So, looking at metrics, the developer might be confident that the method works correctly and it's well-tested.

Still, the code has a very unpleasant bug. If a percent character is followed by something else, not 'd' or '%', then the loop falls into an idempotent state. The `pos` variable is not updated in this case, so the `indexOf()` call will find the same percent character again and again, and the method never finishes. You may try calling it with, e. g., `"%s"` input.

Unfortunately, static analyzers are silent here. While it's theoretically possible to detect such an issue statically, I am not aware of static analyzers that are capable of reporting this case. The only hint you may get from testing is that test coverage reports a missing branch at the `c == '%'` condition: for the test suite above, this condition is never false when reached. So it's a good idea to pay attention to branch coverage, even if line coverage or instruction coverage is 100%.

Ways to avoid this mistake:

- When writing unit-tests for loops, make sure that not only 0 and 1 iterations are covered by tests. The loops are intended to iterate more than once (otherwise it's just a conditional statement), so it's a good idea to test the more-than-once scenario.
- Be careful with methods like `indexOf()` when you process the string. The pattern `pos = str.indexOf(ch, pos)` is quite common to move to the next occurrence of `ch`. However, if you don't update `pos` after that in some branch of your code, it's possible that you'll end up in an endless loop.

3.9 Mistake #22. Incorrect initialization order

Initialization is a complex problem in programming. When you initialize a whole program, a subsystem, or an individual object, you have to deal with the fact that some part of it is not fully initialized yet, so you cannot freely use any functions. You should carefully order initialization steps, otherwise you may end up accessing a non-initialized field.

3.9.1 Static fields

Here's a simple example, which is based on real bug fix in Kotlin IntelliJ Plugin, though the original problem appeared in the Kotlin language rather than Java. For brevity, let's use Java 16 records:

```
record Version(int major, int minor) {
  public static final Version CURRENT =
              parse(Application.getCurrentVersion());
  private static final Pattern REGEX =
              Pattern.compile("(\\d+)\\.(\\d+)");

  static Version parse(String version) {
    Matcher matcher = REGEX.matcher(version);
    if (!matcher.matches()) {
      throw new IllegalArgumentException();
    }
    int major = Integer.parseInt(matcher.group(1));
    int minor = Integer.parseInt(matcher.group(2));
    return new Version(major, minor);
  }
}
```

Here, we have a simple parser of application version strings. For convenience, a public field `CURRENT` is provided, which stores a version of the currently running application. We assume that there's `Application.getCurrentVersion()` static method that returns the current version as a string.

As the parser uses a regular expression, it's a common practice to extract it as a constant. This approach improves the performance, because you need to compile the regular expression only once.

Unfortunately, this class cannot be initialized at all. During the initialization of the `CURRENT` field, we need the `parse` method to function properly which requires the `REGEX` field to be initialized. However, as the `REGEX` field is declared after the `CURRENT` field, it's not initialized yet, so when `parse()` is called, it's still equal to `null`. As a result, class initialization fails with `NullPointerException` inside the `parse()` method.

This particular problem can be easily fixed by reordering the `REGEX` and `CURRENT` field declarations. Also, it was discovered quickly because the problem always manifests itself via exception when anybody tries to use this class. Sometimes, however, the initialization order affects only one code path. I've seen a code pattern like this:

```
class MyHandler {
  public static final MyHandler INSTANCE = new MyHandler();
  private static final Logger LOG =
          Logger.getLogger(MyHandler.class.getName());

  private MyHandler() {
    try {
      …
    }
    catch (Exception ex) {
      LOG.log(Level.SEVERE, "Initialization error", ex);
    }
  }
}
```

Here, the singleton instance of `MyHandler` is created during the class initialization. Usually, everything will work fine. However, if an exception happens during the `MyHandler` construction, it won't be logged as intended. Instead, the uninitialized `LOG` field will be dereferenced and class initialization will fail with `NullPointerException`, masking the original problem.

Static analysis

IntelliJ IDEA has only simple analysis for this purpose. Yet it can catch the first problem: "Nullability and data flow problems" inspection reports possible `NullPointerException` at `REGEX.matcher` call. This warning may look confusing, and some developers might even think that the inspection is wrong here, because the `REGEX` field seems to be properly initialized. Check the initialization order if you see a warning like this.

Standalone static initializers like PVS-Studio have more advanced initialization cycle analysis. PVS-Studio reports both code snippets listed above, and points to the exact code locations where the fields are declared and used.

Ways to avoid this mistake:

- Be careful when initializing classes or objects. Try to avoid complex calls during the initialization or move them to the initializer end. If it’s not possible, make sure that called methods don’t expect your class or object to be fully initialized.
- Don’t forget to test rarely visited code paths. Error reporting code executes rarely but it’s crucial for further investigation of the problem. If it doesn’t function correctly, the error report you get from the user or from server logs might be completely meaningless.

3.9.2 Subclass fields

Another initialization problem which is common in Java occurs when an overridden method is called from a superclass constructor. For example, consider a class like this that represents an entity having a numeric identifier:

```
class Entity {
  final int id;

  Entity() {
    id = generateId();
  }

  protected int generateId() {
    return 0;
  }
}
```

The constructor calls `generateId()` method here, so subclasses can override it and provide their own strategy to generate entity identifiers. Now, suppose that we have a subclass that generates identifiers using random numbers. It could be implemented like this:

```
class RandomIdEntity extends Entity {
  final Random random = new Random();

  @Override
  protected int generateId() {
    return random.nextInt();
  }
}
```

At first glance, it may look ok. However, this doesn’t work: any attempt to construct a `RandomIdEntity` instance yields to `NullPointerException`. The initialization of the `random` field is effectively a part of the `RandomIdEntity` class constructor which is executed after a superclass constructor. Essentially, this class is equivalent to the following:

```
class RandomIdEntity extends Entity {
  final Random random;

  RandomIdEntity() {
    super();
    random = new Random();
  }

  @Override
  protected int generateId() {
    return random.nextInt();
  }
}
```

When the superclass constructor is executed, subclass fields are not initialized yet (except in rare cases when the final fields are initialized with compile-time constants). However, we are calling an overridden method `generateId()`, which assumes that the `random` field is already initialized. The initialization flow is illustrated in figure 3.3.

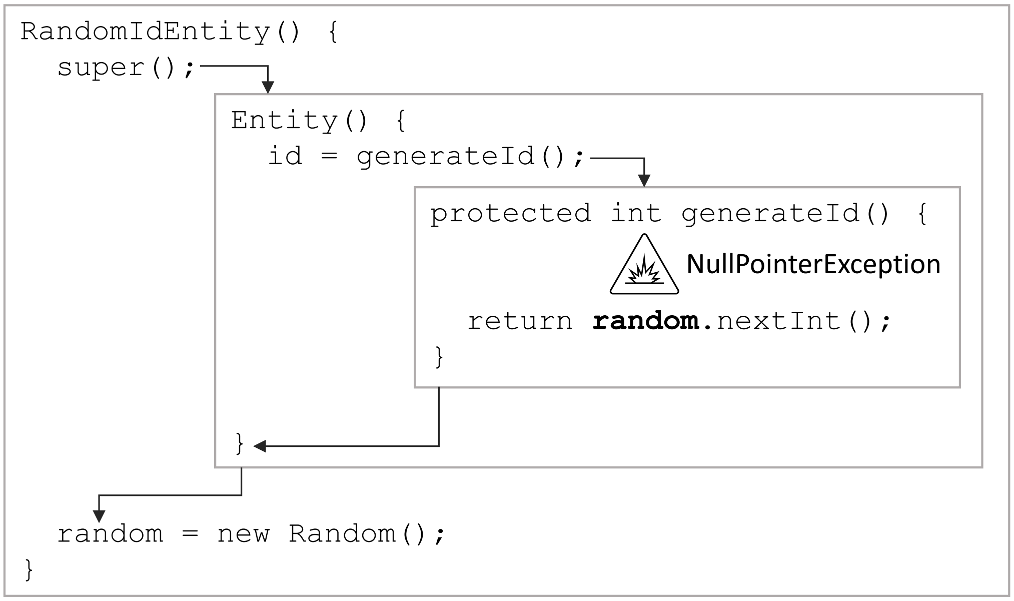


Figure 3.3 Initialization flow of RandomIdEntity() class. The random field is used via superclass constructor before it was initialized.

In this sample, the problem will become clear as soon you try to instantiate the subclass. However, this is not always the case. Sometimes, accidental use of the uninitialized fields may lead to very subtle and hard-to-diagnose bugs.

Static analysis

Some static analyzers may detect possible problems like this. For example, SonarLint has the rule "S1699: Constructors should only call non-overridable methods" (off by default) that reports any calls to overridable methods from constructors. Similarly, IntelliJ IDEA has "Overridable method called during object construction" inspection to report this problem.

Ways to avoid this mistake:

- Avoid calling non-final instance methods from constructors of non-final classes. In general, keep your constructors as simple as possible. If you need to use any helper methods, prefer declaring them as static, private, or final, so subclasses cannot override them.
- Try to move non-trivial initialization logic out of the constructor. For example, you may initialize the `id` field lazily when it is first accessed (though lazy initialization has its own caveats when it comes to thread safety):

```
class Entity {
  Integer id;

  int getId() {
    if (id == null) {
      // not thread safe
      id = generateId();
    }
    return id;
  }

  protected int generateId() {…}
}
```

- If you cannot avoid calling an overridable method in constructor, clearly state in the method's Javadoc specification that this method might be called when the object is not fully initialized.

3.9.3 Class initialization order

Another kind of problem occurs when the initializers of two classes depend on each other. The class initializer includes `static {}` sections declared in the class body and also initializers of static fields. The JVM ensures execution of a class initializer only once and its execution should be finished before you use the class, so that all the static fields are completely initialized when you access them.

The problems start to appear when the initializer of class `A` depends on class `B`, whose initializer depends on class `A`. To understand better how class initializers work, let's consider the following synthetic sample:

```
class A {
  private static final B unused = new B();
  static final String WELCOME = "Hello".toUpperCase();

  public static void main(String[] args) {
    System.out.println(B.MSG);
  }
}
class B {
  static final String MSG = A.WELCOME;

  public static void main(String[] args) {
    System.out.println(B.MSG);
  }
}
```

As you can see, we have two classes `A` and `B` and the `main()` method of each has an identical body. Does this mean that the result of the execution of both classes will be the same? Also, how will the `unused` field affect the program behavior? Think about it.

Let's assume that we launched the `A` class. You may notice that initializer of class `A` uses class `B`. Namely, it creates a new instance of type `B`. This will trigger the initialization of class `B` even if class `A` is not completely initialized yet. The order of initialization is the following:

```
// Class A initializer starts
// Class B initializer starts
B.MSG = A.WELCOME; // which is still null!
// Class B initializer ends
A.unused = new B();
A.WELCOME = "Hello".toUpperCase();
// Class A initializer ends
```

So when `MSG` is assigned, `WELCOME` is still not initialized, and the program will output `null`, which might be unexpected. However, if we launch the `B` class, the initialization order will be different, as the class `B` initializer will start first:

```
// Class B initializer starts
// Class A initializer starts
A.unused = new B(); // class B is not fully initialized but this works
A.WELCOME = "Hello".toUpperCase();
// Class A initializer ends
B.MSG = A.WELCOME; // now its value is "HELLO"
```

This time, we will see `HELLO` in the output, which is more expected.

Field initializers are normally executed in the same order in which fields are declared in the class, so if we swap the `unused` and `WELCOME` fields, then the class launching `A` will output `HELLO`. There's one exception, however. If the field initializer is a compile-time constant expression of a primitive or the `String` type, then the field will be initialized during class loading, before initialization starts. If you remove the `toUpperCase()` call from our sample, you'll also see that null is not printed anymore, as the `WELCOME` field will be initialized during class loading.

While using a class that is not completely initialized may have unpleasant consequences, this is not the biggest problem with circular dependency in class initializers. The biggest problem occurs when you start using both of these classes from different threads. As the virtual machine guarantees that a class initializer is never executed twice, it acquires a lock during class initialization. When one class is initialized during another class initialization, two locks will be acquired. If another thread tries to acquire the same two locks in reverse order, a deadlock will happen. As a result, both threads will be completely stuck until the program is terminated. Any other thread that tries to access any of these classes later will be stuck as well. This may render the program completely unusable.

We can demonstrate this problem creating a separate class that initializes `A` and `B` in different threads:

```
public class Test {
  public static void main(String[] args) {
    Runnable[] actions = {() -> new A(), () -> new B()};
    Stream.of(actions).parallel().forEach(Runnable::run);
    System.out.println("ok");
  }
}
```

This program when launched may be deadlocked without printing “ok”. On my machine, it gets stuck more than 50% of the time, though it may depend on the hardware, OS, and the version of the JVM used.

To diagnose this problem, you can run the jstack tool, specifying the PID of the java process that got stuck. Here’s an excerpt from the output using Java 11:

```
"main" #1 prio=5 os_prio=0 …
   java.lang.Thread.State: RUNNABLE
    at com.example.B.<clinit>(Test.java:22)
    at com.example.Test.lambda$main$1(Test.java:8)
    at com.example.Test$$Lambda$2/0x0000000800061040.run(Unknown Source)
…
"ForkJoinPool.commonPool-worker-19" #21 daemon prio=5 os_prio=0 …
   java.lang.Thread.State: RUNNABLE
    at com.example.A.<clinit>(Test.java:14)
    at com.example.Test.lambda$main$0(Test.java:8)
    at com.example.Test$$Lambda$1/0x0000000800060840.run(Unknown Source)
```

Here you can see that two threads spending their time inside the `<clinit>` method, which is a special method name to denote a class initializer. So you may guess that something is wrong with initializers of these classes. If you use Java 17 or newer, you will see an additional hint:

```
"main" #1 prio=5 os_prio=0 …
   java.lang.Thread.State: RUNNABLE
    at com.example.B.<clinit>(Test.java:22)
    - waiting on the Class initialization monitor for com.example.A
    at com.example.Test.lambda$main$1(Test.java:8)
    at com.example.Test$$Lambda$2/0x0000000800c00c18.run(Unknown Source)
…
"ForkJoinPool.commonPool-worker-1" #23 daemon prio=5 os_prio=0 …
   java.lang.Thread.State: RUNNABLE
    at com.example.A.<clinit>(Test.java:14)
    - waiting on the Class initialization monitor for com.example.B
    at com.example.Test.lambda$main$0(Test.java:8)
    at com.example.Test$$Lambda$1/0x0000000800c00a00.run(Unknown Source)
```

Now the JVM reports "waiting on the Class initialization monitor", so you can easily see that initializer of class `B` waits for the initializer of class `A` and vice versa.

While our sample is synthetic, such problems really happen in practice. One common case of an initialization loop happens when a superclass declares a field of a subclass type:

```
class A {
  public static final A INSTANCE = new B();
}
class B extends A {
}
```

This code may look completely harmless but here deadlock is also possible if one thread triggers the initialization of class `A` while another thread triggers the initialization of class `B`. You can observe the deadlock using the same `Test` class above. The most unpleasant problem here is that initially, when your application has a small number of users, you may rarely have tasks executed in parallel and deadlock may not occur. However, as your application becomes more popular and more actively used, deadlocks may appear, freezing the application completely. At this point, it might be hard to refactor your application to avoid cyclic dependency. For example, the `A.INSTANCE` public field may already be used by third-party plugins, so you cannot simply move it to another class without breaking other code.

Static analysis

The latest sample with subclass is easy to detect statically, so various static analyzers report it. SonarLint has "S2390: Classes should not access their own subclasses during initialization" rule, and IntelliJ IDEA has "Static initializer references subclass" inspection that reports here. Don't ignore these reports, they indicate a very serious problem.

Ways to avoid this problem

- Keep class initializers as simple as possible. If you need something complex, consider lazy initialization. Instead of exposing a public static field, create a method that will perform initialization on demand.
- If you know that your program has an initialization loop, do not postpone fixing it until you actually hit a deadlock. Fixing the problem at an earlier stage may be much easier.
- If your program is expected to serve many clients in parallel, be sure that your automatic tests cover this case. It may happen that your program works fine sequentially but has serious problems when different requests are handled in parallel.
- Upgrade to newer Java versions. As you can see, monitoring tools like jstack are improved in newer versions, which helps with investigating these problems.

3.10 Mistake #23. Missing super method call

When inheriting a class, it's desired sometimes to augment superclass method behavior with additional logic. Often this happens in UI programming when a UI event is handled in both the subclass and the superclass. For example, suppose that you are developing a user interface control using some framework and want to handle specifically when the user presses the 'x' button on the keyboard. In some frameworks, you should override a method like `onKeyDown` and process the corresponding event.

```
public void onKeyDown(KeyEvent event) {
  super.onKeyDown(event);
  if (event.getKeyChar() == 'x') {
    // handle key 'x'
  }
}
```

In such scenarios, it's essential to explicitly call a superclass method. Otherwise, other standard hotkeys will not be handled. However, it's easy to miss the super call or accidentally remove it during refactoring. The compiler won't warn you and probably you won't have runtime exceptions either. However, the program behavior will be incorrect.

Sometimes, the method of a superclass is not called intentionally, even though it has non-trivial behavior. In this case one may say that there's no mistake, but it's still a code smell, which is called "refused bequest". In general, refused bequest means that the subclass intentionally removes some functionality of its superclass. In this case, it's desired to rethink your class hierarchy. Probably it's better to avoid inheritance at all, replacing it with delegation.

Ways to avoid this mistake:

- Annotation packages for static analysis may contain an annotation like `@MustBeInvokedByOverriders` (in JetBrains annotations) or `@OverridingMethodsMustInvokeSuper` (in Error Prone annotations). These annotations could be used to mark super class methods that must be invoked by the corresponding subclass method. Static analysis tools that support these annotations will check and report methods in subclasses that fail to invoke the superclass method. The corresponding IntelliJ IDEA inspection is called 'Method does not call super method', and Error Prone bug pattern is called "MissingSuperCall".
- Avoid designing an API that requires implementation inheritance and overriding non-abstract methods in particular. Instead, prefer interfaces.

For example, to handle UI events, it would be more robust to declare an interface like `KeyListener` and ability to add new listeners to the UI component. In this case, you don't need to do extra steps to invoke previously registered listeners, they will be invoked by a UI component automatically. In fact, the Java Swing toolkit is designed this way, so it's rarely required to override a non-abstract method there, as you can customize many things implementing the interfaces.

3.11 Mistake #24. Accidental static field declaration

Sometimes, developers accidentally declare a static field when an instance field is intended. This may happen due to refactoring or a copy-paste error. As a result, a single field instance is shared between all the objects of a given class. This kind of bug could be very annoying if it was not spotted during code review or via static analysis. That's because, depending on the class usage pattern, such code may pass simple tests and even sometimes produce correct behavior in production. For example, object instances might be usually used one after another. For example, imagine the following simple class:

```
class Person {
  private static String name;

  public void setName(String newName) {
    name = newName;
  }

  public String getName() {
    return name;
  }
}
```

It's possible that the sole purpose of this object is to be deserialized from a database, used, and thrown away. As long as two objects don't exist at the same time, the program might function properly. However, once the number of users increases, the chance for two users to have simultaneous sessions also increases.

This bug is very dangerous, as when it happens you are unlikely to have an exception. Instead, the wrong data might leak to another place, potentially violating privacy or security.

Static analysis

IDEs might address this problem by having different highlighting for static non-final fields. Developers can train themselves to pay special attention to such highlighting because static non-final fields have limited application and should not be generally used. Also, there are static analysis inspections that report when a static field is modified from a non-static method. For example, IntelliJ IDEA has "Assignment to static field from instance context" inspection, though it might be turned off by default. SonarLint reports "S2696: Instance methods should not write to "static" fields" by default.

Unfortunately, marking the field as final doesn't solve the problem completely, as you might have a mutable object stored in static final field. For example, it might be a mutable collection. In this case, you don't assign the field directly but data is still shared between objects. Imagine that you have a tree-like structure where every node can return a list of its child nodes:

```
interface TreeNode {
  List<TreeNode> getChildren();
}
```

Now, we have a `TreeWalker` class that simplifies the breadth-first traversal of this tree:

```
class TreeWalker {
  private static final Queue<TreeNode> toProcess =
    new ArrayDeque<>();

  TreeWalker(TreeNode root) {
    toProcess.add(root);
  }

  TreeNode nextNode() {
    TreeNode next = toProcess.poll();
    if (next != null) {
      toProcess.addAll(next.getChildren());
    }
    return next;
  }
}
```

It's assumed that clients create the `TreeWalker` object passing a tree root as a constructor argument. Then clients repeatedly invoke `nextNode()` until it returns null. Figure 3.4 illustrates traversing a simple tree.

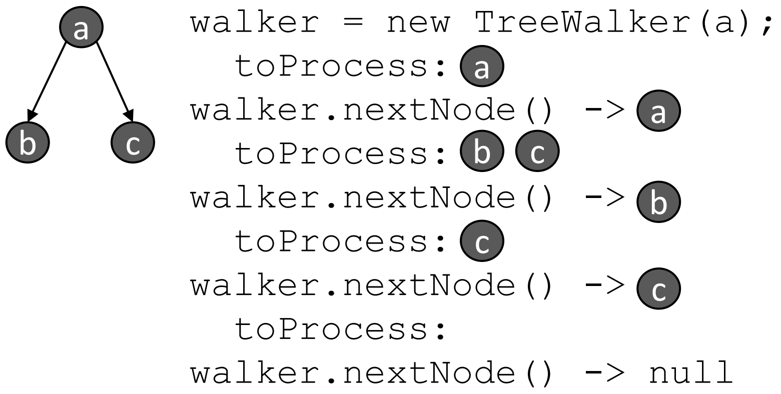


Figure 3.4 TreeWalker processes a simple tree containing three nodes a, b, and c. Everything works correctly as long as only one instance of TreeWalker is used at a time.

As you can see, the `toProcess` field was accidentally declared as static. As long as we never have two `TreeWalker` objects used simultaneously and always process the whole tree till the end, we won't notice a problem. However, if two `TreeWalker` objects are created in separate threads, the nodes from different trees may go to the wrong place. Figure 3.5 shows a possible execution when two threads start processing two independent trees. In this case, nodes from thread 2 tree like 'y' may be retrieved in thread 1, and vice versa. The situation can be even more complicated, as `ArrayDeque` methods like `poll()` and `addAll()` called inside nextNode() are not thread safe, and you may completely break the internal structure of `ArrayDeque`. In this case, you may observe exceptions like

`ArrayIndexOutOfBoundsException` popping from `ArrayDeque` internals. Having an exception is actually a happy case, as you immediately see that your program is wrong and needs to be fixed. However, the exception is not guaranteed.

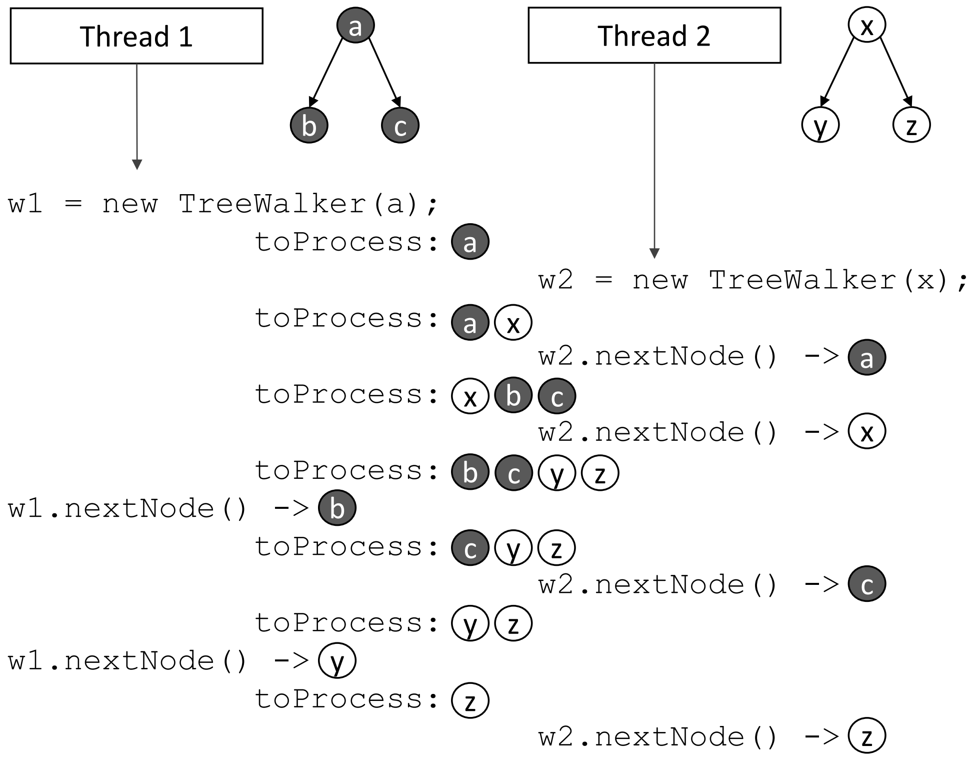


Figure 3.5 A possible execution when using TreeWalker from two threads concurrently. As the toProcess variable is accidentally shared, nodes may leak from one thread to another.

Once I processed the codebase of RubyMine IDE to remove usages of an obsolete `java.util.Stack` collection. This collection is a part of the old Java 1.0 collection framework where all the modification operations are synchronized. This is rarely useful. If you actually need to access a collection from several threads, modern alternatives like `ConcurrentLinkedDeque` would be preferred. And if the collection is never accessed concurrently, this synchronization just adds unnecessary overhead.

I have seen a class (say, `Xyz`) where an instance field of type `Stack` was declared and used in clearly non-thread-safe way. I concluded that as the code is not thread-safe already, it's not a problem to make it even less thread-safe, so I replaced the `Stack` with `ArrayList`-based implementation. Unexpectedly, some tests started to fail sporadically with `ArrayIndexOutOfBoundsException`. Further investigation revealed that `Xyz` instance was stored in a static field of another class which was clearly a mistake: it was not intended to share a single `Xyz` instance between all the users. This produced very subtle errors in highlighting of Ruby source files in the IDE when two or more editors were opened at once. As synchronized collection was used before, we had no exceptions, just slightly incorrect behavior. Thanks to replacing the collection with an unsynchronized one, we revealed the problem.

Mutable objects

Sometimes, the object might look immutable while in fact it's not. There are several infamous examples in Java standard library, notably `java.text.DateFormat` and its inheritors like `SimpleDateFormat`. It looks like a good idea to cache a `DateFormat` object in a static field and reuse it to format many dates in the same way. However, when you call the `format()` method, it mutates the private fields in the object, so calling `format()` concurrently may produce unexpected results. Static analyzers may know about this and warn if you try to store such an object in a static field. For example, SonarLint has a rule named "S2885: Non-thread-safe fields should not be static" that reports such problems. In modern Java, it's preferred to use a modern class to format dates named `DateTimeFormatter`, which is immutable and can be shared between threads.

Ways to avoid this mistake:

- Avoid declaring static non-final fields. They are useful in very limited contexts. Set up your IDE to have a distinctive highlighting for such fields. The highlighting should alert you that something unusual is going on in this code.
- Set up your static analysis tool to report warnings when static fields are updated from non-static contexts. While this is sometimes correct, such updates are very suspicious and could be incorrect in multi-thread programs.
- Avoid storing mutable objects in static fields. If you are unsure whether a given object is mutable or not, check the class documentation. When creating mutable classes, explicitly state this in the class Javadoc.
- Avoid using thread-safe mutable objects when thread-safety is unnecessary and you have a non-thread-safe alternative. Thread-safe objects may introduce additional performance overhead and may hide problems when the object is accidentally used from several threads.
- If the object is expected to be used from several threads, consider writing integration tests that emulate this situation.

3.12 Summary

- While the `if`-statement is a very simple concept, it has its pitfalls as well. One common problem is forgetting `else` in the middle of if-else chain. Another problem is incorrect order of conditions, which may cause some branches to be never visited. Ensure that every `if` branch is covered by your tests.
- Due to legacy reasons, `switch` statements in Java require an explicit `break` to avoid visiting the subsequent branch. If you accidentally forget it, the program behavior may become subtly different. Prefer using `switch` with the arrow syntax available in newer Java versions, as this form of `switch` is not affected.
- Classic `for` loops that iterate over a range of numbers are surprisingly tricky and error prone. When possible, prefer generating them automatically using your IDE.
- Mistakenly written `while` and `for` loops may result in the program state not changing after the first loop iteration. This might be unnoticed if the loop is usually iterated at most once. Test your loops to ensure that they work correctly when two or more iterations are necessary.
- A lot of mistakes happen in programming in general during initialization, as the initialization code must deal with the program which is not fully initialized yet. In Java, these problems may appear if field initializers depend on each other, or if a superclass constructor calls a subclass method when the subclass is not initialized yet.
- Special care should be taken to avoid class initialization loops, as this may cause a complete program freeze due to deadlock when different classes are initialized from different threads.
- Pay attention to the difference between static fields that have one instance per program and regular fields that have one instance per object of a given class. Sometimes a field is mistakenly declared as static, and this may result in bugs that are hard to reproduce.

4 Numbers

This chapter covers

- **Problems due to numeric overflow or rounding**
- **Mistakes that can occur when handling special numbers like Integer.MIN_VALUE, -0.0, or Double.NaN**
- **How to avoid mistakes when doing bit manipulation and using hexadecimal numbers**
- **Caveats in seemingly simple numerical algorithms like oddness check and clamping**
- **Why it's better to avoid certain numeric data types**

Modern programs usually operate with very high-level abstractions. Still, numbers and primitive math is used by programmers quite often. Unfortunately, computer math differs in many ways from real math, and these differences often produce unpleasant bugs. An experienced Java programmer should be aware about many peculiarities of number processing in this language, including numeric overflow and implicit conversions.

4.1 Mistake #25. Accidental use of octal literal

Due to historical reasons, Java supports octal literals. An octal literal looks like a normal number, except that it starts with `0`. For example, `037` is an octal literal that represents a decimal number 31 (as 3*8+7 = 31). It's extremely rare situation when octal literals are actually necessary. Probably the only reasonable use of octal literals that comes to mind is representing Unix file permissions, as there are three bits of permissions (read/write/execute) for different classes of users (user, group and others). So sometimes it's convenient to represent them as a constant like 0644, which means that the user can read and write, while group and others can only read. There is no other widely known context where octal numbers could be useful.

However, occasionally people put an extra leading zero before the decimal literal by mistake. Some developers are simply unaware of octal numerals and may use a leading zero to align a series of numbers:

```
int n1 = 9876;
int n2 = 5432;
int n3 = 0123;
int n4 = 4567;
```

Here, a leading zero was added to the series of constants to make it look more uniform. As the literal doesn't contain the digits 8 or 9, it's parsed as a valid octal literal, but its numeric value is completely different (83 instead of 123). Depending on further use of the constant, such a mistake might go unnoticed for a long time.

Ways to avoid this mistake:

- The best solution is to ban octal literals from your project completely. Configure your style checker or static analyzer to report octal literals as errors. IntelliJ IDEA has the "Octal integer" inspection that reports every octal literal. The corresponding SonarLint rule is called "S1314: Octal values should not be used" (turned off by default). If you like to align numbers in adjacent lines, use spaces instead of leading zeroes.

4.2 Mistake #26. Numeric overflow

To make computations fast, computers must allocate a fixed number of bits for every number by default. This means that you have to deal with numeric overflow when the result of the computation does not fit the memory chunk allocated to store the value. What happens exactly in this case depends on the programming language and the data type used.

4.2.1 Overflow in Java

By default, numeric operations in Java overflow silently, just yielding the incorrect result. Java performs math computations on four data types: `int`, `long`, `float`, and `double`. When other numeric primitive types like `byte`, `short`, and `char` participate in computation, they are widened to `int` before the actual operation happens. The overflow behavior for integral types differs significantly from the floating-point types.

For `int` and `long`, if the result of computation exceeds the type, the higher-order bits are simply discarded. We don't want to dig into details of two's-complement number format representation. Instead, let's formulate the practical rules for the `int` type (for `long`, they are similar):

- If the result of addition or subtraction of two `int` values is bigger than `Integer.MAX_VALUE` (which equals 2'147'483'647) then 2^{32} is subtracted from the correct result, so you'll always get the negative number instead of positive one.
- If the result of addition or subtraction of two `int` values is less than `Integer.MIN_VALUE` (which equals -2'147'483'648), then 2^{32} is added to the correct result, so you'll always get the positive number instead of negative one.

- If the result of multiplication of two `int` values overflows, you may get completely random number that doesn't resemble the original one. Yes, strictly speaking, it's not random, but it's rarely practical to try to get anything useful from it.
- The result of division of two `int` values never overflows, except in only one case: `Integer.MIN_VALUE/-1`. In this case, you'll get `Integer.MIN_VALUE`, as the `int` type does not include the the corresponding positive number. We'll discuss this number in mistake #28 below.

The problem with the overflow is that it happens silently. If you don't detect it immediately, you may not notice that the result is incorrect until much later, after many additional computations, and it could be really hard to find the cause of the mistake. For example, let's calculate the expression `(x * x + 1_000_000) / x` for different values of `x`.

Table 4.1 Step-by-step computation of (x * x + 1_000_000) / x expression in Java for different values of x. Incorrect results that happen due to numerical overflow are shown in *italics*.

x	x * x	x * x + 1000000	(x * x + 1000000)/x	correct result
10000	100000000	101000000	10100	10100
46300	2143690000	2144690000	46321	46321
46340	2147395600	*-2146571696*	*-46322*	46361
46350	*-2146644796*	*-2145644796*	*-46292*	46371
100000	*1410065408*	*1411065408*	*14110*	100010

Table 4.1 shows how this formula is calculated for different values of `x`. As you can see, we can get correct results for x values up to 46'300. However, as the square of `x` approaches to `Integer.MAX_VALUE`, we start having trouble. First, the result of addition overflows, and we suddenly get the negative result. Very soon after that the multiplication overflows as well. This is especially sad because the correct final result after division would fit the `int` type perfectly but we cannot get it due to intermediate overflow.

When we are just beyond the overflow, we can detect the problem, as the number becomes negative. However, when we are going further, it may become positive again. For example, for x = 100'000, the result will be 14'110 which is, although positive, not even close to the correct result 100'010. Unfortunately, Java gives us no clue whether overflow happened or not.

The behavior is somewhat better for floating-point numbers, as if you have intermediate overflow, you will likely end up with special numbers like Infinity or NaN, so you can just check the result of the whole operation. Let's calculate the same expression but using the `double` type and assuming that x = 10^{200}:

```
double x = 1e200;
double res = (x * x + 1_000_000) / x;
// res is now Infinity due to intermediate overflow
```

So, you can check the floating-point number against Infinity or NaN after the series of calculation and report the error.

One famous overflow problem is average calculation. The math formula to calculate the average of two numbers is as simple as `(a + b) / 2`. However, it doesn't work if you use the Java `int` or `long` types and the sum exceeds the type domain. The expected average always fits the type domain, as it cannot exceed both input numbers, but you may get the incorrect result:

```
int a = 1_000_000_000;
int b = 2_000_000_000;
int average = (a + b) / 2;
// average is -647_483_648, which is incorrect
```

This bug was discovered in 2006. It appeared in the binary search procedure in standard libraries for a number of programming languages, including Java. The binary search may work incorrectly if you have an array containing more than 2^{30} elements. This bug can be fixed using an unsigned bitwise shift, which cancels out the overflow effect:

```
int a = 1_000_000_000;
int b = 2_000_000_000;
int average = (a + b) >>> 1;
// Now average is 1_500_000_000 as expected
```

Alternatively, you may switch to the `long` type when computing the intermediate sum. While it's fixed in the standard library, the bug may still reside in other places, and the problem is that it appears only with very large inputs.

It's interesting that a similar bug still exists in the Java standard library in another place. Stream API allows you to calculate the average of long numbers, but internally it calculates the sum in the `long` variable as well, so the result could be surprising when the input numbers are big:

```
LongStream.of(Long.MAX_VALUE, Long.MAX_VALUE)
          .average().getAsDouble()
```

This expression counter-intuitively evaluates to -1.

Static analysis

While static analyzers try to help here, they cannot cover most of possible overflow cases, as almost any computation can potentially overflow. The simple thing analyzers detect is overflow in constant expressions. For example, try declaring a variable like this:

```
int microseconds = 24 * 60 * 60 * 1000 * 1000;
```

The result of this computation doesn't fit the `int` type, so it will silently overflow. IntelliJ IDEA reports this problem with "Numeric overflow" inspection. Note however that bytecode-based analyzers like SpotBugs cannot report such problems at all, as constants are computed during compilation, so there's no computation inside the bytecode and analyzer cannot know whether it overflows or not. In any case, if any variables are used in the computation, analyzer will unlikely help.

Some static analyzers may report the problem with average value computation described above. For example, SpotBugs has "Computation of average could overflow" warning. However, it quickly becomes annoying, as in many cases overflow never happens. E. g. it's often necessary to calculate average for the screen coordinates to place some UI elements. The assumption that screen coordinates will never exceed one billion looks quite safe, so using normal division is ok. Yet, static analyzer may report every instance of such division, which will result in tons of noise warnings. So, I would not recommend static analyzers here. Instead use a general advice: be careful when working with too big numbers.

Ways to avoid this mistake:

- Always think about possible overflow when doing integer math. Multiplication is the most dangerous operation, as it may produce a result that is significantly larger than the operands. Use the following rule of thumb: if at least one of your `int` operands exceeds 50'000, it's better to take a closer look and check for possible multiplication overflow.
- Try to rewrite your formula using algebraic transformations or bitwise operations to avoid intermediate overflow if the final result usually fits your data type. For example, for the `int` type, the expression `(x * x + 1_000_000) / x` overflows when `x` exceeds 46'330. However, if you simply change it to `x + 1_000_000 / x`, it won't overflow anymore.
- Use library methods for complex computations where possible, as they might be carefully written to avoid intermediate overflow. For example, `Math.hypot(x, y)` is essentially the same as `Math.sqrt(x * x + y * y)`, but `Math.hypot` won't overflow even if the intermediate expression `x * x` overflows.

- When you need to perform integer computations without possible overflow, you may use the `BigInteger` class. This could be much slower than simple operations, but it is guaranteed not to create overflow. Using the `long` type instead of `int` may also help in many practical cases without significant performance degradation. Note that at early stages of some project development, numbers above 2 billion might look unrealistic, but sometimes the amount of data grows quickly. For example, the NuGet service used the `int` type to track the downloads count (it was C#, not Java, but the idea is the same). As the service got more popular, the downloads count of one package exceeded one billion with 10 million new downloads weekly, so overflow would be reached in quite a realistic timeframe. Luckily, the problem was noticed in advance, and there was enough time to fix it.
- It's possible to catch the numeric overflow if you use `Math.*exact` methods instead of plain operators. E. g. `Math.addExact(x, y)` instead of `x + y`. These methods throw `ArithmeticException` in case of overflow, so you can catch it and react. These methods are usually intrinsified by VM, so when there's no overflow the performance is expected to be comparable to simple operations.

Unfortunately, using methods like `addExact` may make your code hard to read. Java doesn't provide any kind of operator overloading, so you cannot tell the compiler to treat a simple '+' operator as `addExact`. However, sometimes these methods are exactly what you need. For example, they are used in the implementation of the JDK method `str.replace(target, replacement)` that allows you to replace all the occurrences of the string `target` within the `str` with the `replacement`. Naturally, the resulting string could become longer if the replacement is longer than the target. Before the resulting string is created, it's necessary to calculate its length. At this point it's already known how many replacements should be made, so the length of the resulting string should be

```
bufferSize = str.length() +
  (replacement.length() - target.length()) * replacementCount;
```

As you can see, this calculation may overflow in multiplication and addition, and you may get a completely incorrect buffer size and start filling it. The desired behavior would be to throw `OutOfMemoryError`. This can be easily done with exact operations, like this:

```
int bufferSize;
try {
  bufferSize = Math.addExact(str.length(),
      Math.multiplyExact(
          replacement.length() - target.length(),
          replacementCount));
} catch (ArithmeticException e) {
  throw new OutOfMemoryError();
}
```

Looks quite verbose but it's ok for widely used library code.

4.2.2 Assigning the result of int multiplication to a long variable

One common overflow case is when the result of integer multiplication is assigned to a long variable or passed as a long parameter. This happens when you need to perform unit conversion. For example, if you have the time in seconds, and want to pass it to the method accepting microseconds, you may write a code like this:

```
void process(int seconds) {
  long microseconds = seconds * 1_000_000;
  useMicroseconds(microseconds);
}
```

This will work correctly if the number of seconds doesn't exceed 2147 (about 35 minutes). If it's bigger, then the multiplication will silently overflow, and you may get incorrect behavior. Please note that in Java, the target variable type is not used to determine the type of mathematical expression, so even if we assign the result to the `long` variable, the multiplication is performed using the `int` type. The code can be easily fixed using the `L` suffix:

```
long microseconds = seconds * 1_000_000L;
```

Static analysis

IntelliJ IDEA analyzer reports this error with "Integer multiplication or shift implicitly cast to 'long'" inspection. SpotBugs has the bug pattern named "Result of integer multiplication cast to long".

Ways to avoid this mistake:

- If you are about to multiply two numbers and assign the result to `long`, check that at least one of the operands is also `long`, to avoid possible overflow. You may add an explicit `(long)` cast or use the `L` suffix for literal.
- Use API methods instead of performing manual computations. In this case, you can use `TimeUnit` enum:

```
long microseconds = TimeUnit.SECONDS.toMicros(seconds);
```

- When possible, test your code with large numbers.

4.2.3 File size and financial computations

Aside from time computations, pay attention to the file offset computations. For example, sometimes binary files are organized in fixed-size blocks, and every block is numbered, usually starting with 0. When it's desired to read or write the x'th block, it's necessary to seek within the file using the `RandomAccessFile.seek()` method or similar. In this case, one should multiply the `x` by the block size to get the offset. The problem may appear if the multiplication is performed using the `int` type. People rarely work with files of sizes exceeding 2 Gb, so the size in bytes usually fits the `int` type, and overflow doesn't happen.

However, when the file size exceeds 2 Gb, calculations using the `int` type may cause unpleasant bugs like corrupted files and data loss.

Another dangerous area is financial computations. It's usually recommended to store an amount of money like an item price or an account balance using the `BigDecimal` type. However, this adds a significant performance overhead which could be unacceptable. Using `float` or `double` is a bad idea as rounding errors may badly affect the value. Sometimes, a good alternative is to use an `int` or `long` type scaled to the minimal precision you need. For example, if you round all the values to 1¢, it might seem like a good idea to store amounts of money in cents and perform all the computations in cents. However, there's also a big risk of overflow. It's hard to predict how the quantities may scale in the future. Amounts of money to process may unexpectedly go up due to inflation or business growth.

One story happened in 2021 at Nasdaq. It stored share prices in 32-bit unsigned integer values using a 0.01¢ scale, and the maximum share price that could be represented with this data type is $(2^{32}-1)/10,000$ = \$429496.7295. Probably at the time of development it looked like a practical limit, as no share price was close to this number. However, Berkshire Hathaway's share price exceeded this number on May 4th, 2021, causing a numerical overflow in the stock exchange's backend. Using 64-bit data type would be much safer here.

Another overflow bug happened in January, 2022 with Microsoft Exchange Server. It has a versioning scheme for Microsoft Filtering Management Service updates that contains 10 digits like YYMMDDNNNN. Where YY is the last two digits of the year; MM is the month; DD is the day, and NNNN is update number within that day. It appeared that these version strings were parsed to 32-bit signed integer values. It worked perfectly till the end of 2021, as 2112310001 is less than 2^{31}. However, the version string for the first update in year 2022 was 2201010001, which is more than 2^{31}. This caused a severe malfunction of Microsoft Exchange Server and required manual actions for every customer to fix it.

Ways to avoid this mistake:

- Unless you have very strong reason, always use the `long` type to store file size and offset within the file. Convert operands to the `long` type before any offset computation.
- Test your code with files that are larger than 2 Gb.
- Do not underestimate the quantities your software will have to deal with in the future. Numbers that look ridiculously big today might be completely reasonable tomorrow.

4.3 Mistake #27. Rounding during integer division

Sometimes, integer division is used where floating-point division is expected. The common pattern looks like this:

```
void process(int value) {
  double half = value / 2;
  // use half
}
```

Here, before the assignment, an integer division will be performed truncating the fractional part. As a result, the `half` variable will never contain the fractional part and, for example, `process(2)` and `process(3)` will return the same result.

Static analysis

Such a mistake can be easily detected by many static analyzers. The corresponding IntelliJ IDEA inspection name is "Integer division in floating point context" and SonarLint rule is called "S2184: Math operands should be cast before assignment". However, this may become more dangerous if developers tend to over-use implicit type for local variables:

```
void process(int value) {
  var half = value / 2;
  // use half
}
```

Now, the type of the `half` variable will be inferred to `int`. It becomes unclear whether it was intended to truncate the result of division or it's a mistake and the floating-point type was intended instead. Static analyzers are usually silent here, even if the `half` variable is used in floating-point context later (with automatic widening).

Ways to avoid this mistake:

- Be careful with division. Always ask yourself whether you want the truncated result or you need the fractional part. Use an explicit floating-point constant or a cast operator in the latter case:

```
double half = value / 2.0;          // explicit double constant 2.0
double half = (double) value / 2; // explicit cast operator
```

- Never use the implicit variable type for primitive variables. You won't save many characters but the inferred type might differ from what you are expecting. This also confuses static analyzers, and they cannot help you anymore because they don't know your intent.

4.4 Mistake #28. Absolute value of Integer.MIN_VALUE

The `Integer.MIN_VALUE` (-2147483648) number is special: there's no positive counterpart. As integer numbers take 32 bits in the computer memory, we have 2^{32} possible numbers, which is an even number. However, as we need to represent zero, we have uneven count of positive and negative numbers. This leads to an equation that completely violates the laws of math:

```
// prints "true"
System.out.println(Integer.MIN_VALUE == -Integer.MIN_VALUE);
```

To make things worse, this also spoils the behavior of the `Math.abs()` method:

```
// prints "-2147483648"
System.out.println(Math.abs(Integer.MIN_VALUE));
```

That is, the math says that the absolute value of the number is never negative. Java doesn't agree with this: the result of `Math.abs(Integer.MIN_VALUE)` is negative.

When does this matter? Some people use `Random.nextInt()` method when they actually need a random number within some bounds. E. g. to get a random number between 0 and 9 one may use:

```
Random random = new Random();
// Random number between 0 and 9 (almost always)
int randomNumber = Math.abs(random.nextInt()) % 10;
```

This works in 4,294,967,295 cases out of 4,294,967,296. However, once in 2^{32} runs, `nextInt()` returns `Integer.MIN_VALUE`, and the random number will become -8, which may result in extremely rare and hard to reproduce bug in production. Table 4.2 shows how `randomNumber` is calculated for different values returned by `random.nextInt()`.

Table 4.2 Resulting randomNumber for different values returned by nextInt().

random.nextInt()	Math.abs(random.nextInt())	Math.abs(random.nextInt()) % 10
0	0	0
123	123	3
-1024	1024	4
-2147483647	2147483647	7
-2147483648	-2147483648	-8

Another scenario appears when it's necessary to distribute some objects evenly (e. g. to process them in a thread pool, put them into hash-table buckets, distribute the computation to a cluster, etc.). In this case, an object hash code is often used to assign the target thread, bucket, or cluster node. The code may look similar to this snippet:

```
static final int NUMBER_OF_NODES = 10;

void processObject(Object obj) {
  int node = Math.abs(obj.hashCode()) % NUMBER_OF_NODES;
  process(node, obj);
}
```

This case is similar, though probably it's easier to reproduce. If the object hash code is `Integer.MIN_VALUE`, then resulting node will be -8 and it's unclear what will happen next.

Note that you won't have this problem if your divisor (`NUMBER_OF_NODES` in the sample above) is a power of 2, because in this case, the remainder will be zero.

Static analysis

SonarLint reports these problems via rule "S2676: Neither "Math.abs" nor negation should be used on numbers that could be "MIN_VALUE"". Some cases like the one mentioned above can be detected by SpotBugs via bug patterns "Bad attempt to compute absolute value of signed 32-bit hashcode" and "Bad attempt to compute absolute value of signed random integer".

Ways to avoid this mistake:

- Use `Math.abs` with care; remember that the result could be negative.
- If you use Java 15 or later, consider using `Math.absExact` instead. Instead of returning a negative number, it will throw an `ArithmeticError` exception in case the input is `Integer.MIN_VALUE`.
- Avoid using `Math.abs` with a subsequent `%` operation. Use `Math.floorMod` instead. It's a variation of the remainder operation but the result sign is the same as that of the divisor rather than the dividend. If the divisor is a positive number, you will always get a positive result:

```
void processObject(Object obj) {
  int node = Math.floorMod(obj.hashCode(), NUMBER_OF_NODES);
  process(node, obj);
}
```

- If you need a random number within given range, do not use the remainder calculation at all. Instead, use a library method, `nextInt()`, that accepts a parameter. It works correctly and also produces a uniform distribution. A remainder technique may produce slightly skewed results: lower numbers may appear more often.

```
Random random = new Random();
// Random number between 0 and 9 (always)
int randomNumber = random.nextInt(10);
```

- Create objects with an `Integer.MIN_VALUE` hash code in the tests and pass them to your code. This would allow catching such mistakes before they slip into production.

4.5 Mistake #29. Oddness check and negative numbers

Sometimes it's necessary to check whether a given number is odd or not. There's no ready method to do this in the Java standard library but it looks like it's a simple algorithm, so programmers usually do this check manually. How would you check whether a given integral number is odd? Some of you may come up with the following algorithm:

```
public static boolean isOdd(int value) {
  return value % 2 == 1;
}
```

This approach works correctly for positive numbers. However, as we saw in the previous section, when a negative value is used with the `%` operator, the result cannot be positive. So, this method never returns true for a negative value. Table 4.3 shows what happens inside the `isOdd()` method depending on the input value.

Table 4.3 Computation inside isOdd() method.

value	value % 2	value % 2 == 1
-3	-1	false
-2	0	false
-1	-1	false
0	0	false
1	1	true
2	0	false
3	1	true

In many cases, people don't work with negative values, so the formula `value % 2 == 1` will work correctly. This may add a false sense of security that the formula is correct, so developers will use it automatically without thinking that anything may go wrong. And one day negative numbers appear on the horizon, making the program behave incorrectly.

Static analysis

The `value % 2 == 1` pattern can be easily detected by static analyzers. IntelliJ IDEA has the inspection "Suspicious oddness check" to report such checks. Unfortunately, it's turned off by default. SonarLint has the rule "S2197: Modulus results should not be checked for direct equality". SpotBugs reports this problem using the "Check for oddness that won't work for negative numbers" warning.

Ways to avoid this mistake

- It's safer to compare the remainder to zero instead of a positive or negative number. So you may use `value % 2 != 0` formula instead.
- An alternative way is to rely on bitwise arithmetic extracting the lowest bit from the number: `value & 1 == 1`.
- If you ever need an oddity check in your program, instead of writing a formula inline, create a utility method like `isOdd`, test it thoroughly, and use it everywhere.

4.6 Mistake #30. Widening with precision loss

Primitive widening is a type conversion that changes the value of one primitive type to another, usually a wider one, preserving its value in most cases. The problem is that 'most cases'. As widening occurs implicitly in arithmetic operations, assignments, method returns, or method arguments passing, the problem could go unnoticed.

Three widening conversions may result in precision loss. Namely: `int` → `float`, `long` → `float` and `long` → `double`. The problems occur only when numbers become big enough. For `long` → `double` conversion the number should be at least 2^{53} (~9×10^{15}). For `int` → `float` and `long` → `float` the problems appear with much smaller numbers, starting from 2^{24} (close to 17 million). After that, the corresponding floating-point number cannot hold all the digits of the original int or long number, so the rounding occurs. E. g. consider the following sample:

```
long longVal = 1L << 53;
double doubleVal = longVal;
double doubleValPlusOne = longVal + 1;
System.out.println(doubleVal == doubleValPlusOne);
// "true" is printed
```

It's obvious that longVal and longVal + 1 are different numbers. However, when converted to double, they become the same due to rounding.

Ways to avoid this mistake:

- Avoid using the float type, unless you are absolutely sure that it's justified due to the performance reasons. This will rule out the int → float and long → float conversion problems and you will be safe until you reach 10^{15} which is quite enough for many domains.
- Avoid implicit widening to floating-point numbers. Add explicit type casts to show the readers that the rounding is intended:

```
double doubleVal = (double) longVal;
```

- Always ask yourself whether the rounding is harmful in your case.
- If you care about precision more than about performance, use the BigDecimal type. It allows you to control the precision completely.

4.7 Mistake #31. Unconditional narrowing

Sometimes, you have mismatched numeric types and have to apply an explicit primitive conversion. Probably the most dangerous one is conversion from long to int type. For example, some standard API methods, like Stream.count(), or File.length() return a value of the long type, but you have to pass it to a method that accepts the int type. The simplest and shortest way to do this is using the type-cast operator:

```
processFileLength((int) file.length());
```

Here, however, we have the same concerns that were listed in the "Numeric overflow" mistake considered previously. If the file length happens to exceed 2 Gb, the result of type cast will overflow to a negative number. If the file is even bigger and exceeds 4 Gb, the resulting number will be positive again but much smaller than the actual file length. This may cause an exception much later, incorrect behavior, or even silent data corruption.

Static analysis

Static analysis might help to identify narrowing casts. IntelliJ IDEA has the "Numeric cast that loses precision" inspection (off by default). Unfortunately, such an inspection may report many harmless casts which undermines its usefulness.

Ways to avoid this mistake:

- When adding an explicit narrowing cast, for example, to convert `long` value to `int`, always ask yourself whether the value never exceeds the maximum possible value of the target type.
- If you are unsure, consider using the `Math.toIntExact()` method instead of `(int)` cast. This method throws an `ArithmeticException` if the input `long` value cannot be exactly represented via `int` type. In this case, the exception report will point exactly to the problematic place, and you can avoid incorrect program behavior or data corruption.
- When possible, use the `long` type for quantities like duration in milliseconds, file size, downloads count, etc. In the modern programming world, it's not uncommon for these numbers to exceed two billion, even if it looks like an unreachable limit when you start your program.

4.8 Mistake #32. Negative hexadecimal values

Numeric types in Java are signed, so for example, the `int` type range is −2'147'483'648..2'147'483'647 and you cannot have an `int` literal outside of this range. If we write this in hexadecimal notation, we will have `-0x8000_0000..0x7FFF_FFFF`. However, when you are working with hexadecimal numbers, it's often more convenient to treat them as unsigned. Java allows you to do so, and the positive literals from `0x8000_0000` to `0xFFFF_FFFF` are also legal, but they map to the corresponding two's complement negative numbers. For example, `0xFFFF_FFFF` becomes −1, `0xFFFF_FFFE` becomes −2, and so on. This is often convenient but may cause mistakes when the widening conversion takes place.

For example, suppose that you want to apply a mask to a long number, clearing 32 higher bits. It looks like the code like this does the thing:

```
long mask = 0x0000_0000_FFFF_FFFF;
long result = value & mask;
```

However, this doesn't work. The literal `0x0000_0000_FFFF_FFFF` is the `int` literal, not the `long` one (the 'L' suffix is missing). Normally, you can use an `int` literal just as fine, as it will be widened to the equivalent `long` one. Here, however, the `int` literal represents −1. When it's widened to `long`, it's still −1, or `0xFFFF_FFFF_FFFF_FFFFL` (all bits set). So, applying this mask does not change the value at all.

Static analysis

In some cases, static analyzers might be helpful. E. g., IntelliJ IDEA warns you when the int hexadecimal constant is used where long type is expected (the inspection name is "Negative int hexadecimal constant in long context").

Ways to avoid this mistake:

- Be careful when dealing with bitwise arithmetic. Remember that numbers are signed, and this affects automatic widening.
- Always use the 'L' suffix on literals when working with long values, as this will prevent an unexpected implicit type conversion. Note that it's not recommended to use lowercase 'l', as it can be mixed with the digit '1' depending on the font used.

4.9 Mistake #33. Implicit type conversion in compound assignments

A compound assignment is an assignment like `x *= 2` when the previous value is modified using a binary operation. Some developers assume that `x *= y` is equivalent to `x = x * y` but in fact it's equivalent to `x = (type_of_x)(x * y)`. So while it's not evident from the code, implicit narrowing conversion may be performed during compound assignment. This allows amusing assignments like this:

```
char c = 'a';
c *= 1.2; // c is now 't'
```

Here, the 'a' character is implicitly widened to `double` number 97.0, which is multiplied by 1.2, yielding 116.4. This result is implicitly narrowed to the `char` type, yielding 't'. Luckily, this scenario occurs mostly in puzzlers, rather than in real code. However, the following pattern was actually discovered in a production code base (in the Spring framework project, to be more specific):

```
byte b = getByte();
if (b < 0) {
  b += 256; // value of b is unchanged!
}
```

The author of this code wanted to convert negative byte values to positive ones (like changing −1 to 255). However, this doesn't work. In fact, the assignment inside the condition is equivalent to

```
b = (byte)((int) b + 256);
```

So, we widen `b` to `int`, add 256 (producing the correct result), then narrow it back to `byte`, which simply restores the original value of `b` in order to fit the `byte` type range `-128..127`. It would work correctly if the variable `b` were declared as `int`.

Another compound assignment example that may do nothing to bytes is unsigned right shift:

```
byte b = -1;
b >>>= 1;
```

Here, the byte value is first widened to `int` and becomes `0xFFFFFFFF`. Then after shift, it becomes `0x7FFFFFFF`. Finally, it's implicitly narrowed back to `byte`. At this point, higher bits are dropped, and it becomes −1 again.

Static analysis

Both cases above are reported by “Nullability and data flow problems” IntelliJ inspection. The warning sounds like “Variable update does nothing”. However, this inspection may not report more complex cases and will be silent if the result is not exactly the same. So, you cannot rely too much on static analyzers here.

Ways to avoid this mistake:

- Pay attention to the target variable type of a compound assignment. Ask yourself whether narrowing conversion is possible in your case.
- Avoid using the `byte` and `short` types. Prefer using `int` or `long` where possible. In many cases this will save you from undesired overflow problems. Using `byte` or `short` fields or array elements could be justified if you need to save memory. However, you won’t have any performance degradation if you replace `byte` or `short` type of local variables, method parameters or return values with `int`.
- If you really need to use the `byte` or `short` types, test your code on negative inputs or add assertions to ensure that negative inputs are not possible.
- Use library methods where possible, even for simple math operations. For example, converting negative byte values to positive ones could be done via the `Byte.toUnsignedInt` method:

```
int b = Byte.toUnsignedInt(getByte());
```

In this case, implicit narrowing to `byte` is impossible, and declaring the variable type as `byte` will result in a compilation error.

4.10 Mistake #34. Division and compound assignment

Another kind of mistake is possible if the right-hand operand of a compound multiplication-assignment is a division operation. Assuming that the type of `a`, `b`, and `c` is `int`, one may naively think that `a *= b / c` is equivalent to `a = a * b / c`. In fact, it’s `a = a * (b / c)`, which may produce completely different result taking into account integral division semantics. We encountered such a problem in the Sipdroid project. Originally, the buffer size for audio recording was calculated using the following formula:

```
int min = AudioRecord.getMinBufferSize(...) * 3 / 2;
```

This code worked correctly. However, later it was decided that increasing it by 50% is necessary only if the buffer is shorter than 4 KB. So, the code was changed in the following way:

```
int min = AudioRecord.getMinBufferSize(...);
if (min <= 4096) min *= 3 / 2;
```

Now, the division 3 / 2 evaluates before the multiplication, and it always results in 1. As a result, we always multiply the `min` by one, not changing its value at all.

Static analysis

In code patterns like this, IntelliJ IDEA reports "Suspicious integer division assignment". Additionally, another warning, "Variable update does nothing," is reported here, as we effectively multiply the variable by 1.

Ways to avoid this mistake:

- Be careful when you have another math operation on the right side of a compound assignment. Better to avoid a compound assignment at all if you are not sure.
- Check your math operations with simple code snippets (e. g. using JShell). Just set the input variables to some reasonable values, then copy the math operations from your program and check whether the result is expected.

A somewhat similar problem might occur if you use `a -= b - c`: it's equivalent to `a = a - b + c`, not `a = a - b - c`. Again, if a compound assignment looks confusing, it's better to avoid it to make the code clear.

4.11 Mistake #35. Using the short type

In the mistake #33 we suggested to avoid using the `byte` and `short` types. We would like to additionally stress this for the latter. While the `byte` type is occasionally useful, especially when you are actually working with bytes (e.g., reading and decoding binary data), the `short` type is probably the least useful primitive type in Java. It may become especially dangerous when it leaks into an API. If you have methods that accept `short` parameters or return `short` values, sooner or later you may produce an overflow problem when the type simply cannot fit all the values you have. In this case, refactoring the code might be hard, especially if your code is in a library with external clients.

A problem like this happened in the software package BioJava. It contains a method to align protein structure and used the `short` type to calculate alignment penalty which is derived from the number of amino acids in the protein sequence. While most proteins are rather short (around 300-500 amino acids), there are also a few huge proteins. The human protein called 'titin' may contain more than 34000 amino acids, and this causes overflow of the `short` type which can hold values up to 32,767 only. As a result, the alignment procedure worked incorrectly for such a huge protein. To fix this, it was necessary to change the signatures of public methods to migrate from `short` to `int` type.

Ways to avoid this mistake:

- Do not use the `short` type, even if you are completely sure that your number will never exceed 32,767. Your estimation could be wrong, or new data may appear later. Even if it's correct, it will be hard to use your API. Because Java automatically widens the `short` type to `int` in math operations, users of your API will often need to explicitly downcast their values to `short` which could be very annoying. It's unlikely that you will have any practical benefit from using the `short` type. It can be useful in extremely rare cases when you want to have a compact footprint of a long-living object to save some memory.

- If you carefully measured the memory footprint and still believe that using the `short` type provides a significant benefit in terms of memory utilization, consider not exposing it to clients. Use the `short` type only for private fields and private methods.

4.12 Mistake #36. Manually writing bit-manipulating algorithms

Sometimes, programs consider numbers not as quantities but as bit vectors and manipulate individual bits using operations like bitwise-and or shift. Such manipulations are also called "bit twiddling". One possible application is to pack several values into a single number. For example, it's quite common to have a variable where different bits represent different Boolean values. This allows saving computer memory and making the operations blazingly fast, as processors naturally work with bits and bitwise arithmetic operators in Java usually map directly to assembly instructions.

Bitwise operations may look simple, but this simplicity can be deceiving. Many simple algorithms have corner cases. For example, it looks like an easy task to keep the desired number of least-significant bits in a number:

```
int value;

void applyMask(int keepBits) {
  value &= (1 << keepBits) - 1;
}
```

For example, if `keepBits` is 8, `1 << keepBits` evaluates to `0x100`, and after subtracting one, we get `0xFF`. As a result, 8 least significant bits of value are preserved, while the others are zeroed.

This works fine for `keepBits` between 0 and 31. However, one may expect that when `keepBits` is 32, the method will preserve all 32 bits of value, or simply do nothing. In fact, `1 << 32` is the same as `1 << 0`, which is simply 1. After subtracting 1, we get 0, so the method will reset the `value` to 0 instead of doing nothing. This could be unexpected in some applications.

Another example of a corner case is bitwise shift operation. Its result can be either `int` or `long`, depending on the left operand type (unlike other arithmetic operations where the resulting type is defined by both operands). This becomes problematic for left shift operations if the result is expected to be long. In this case, the left operand is often 1 and it's easy to forget to add the L suffix. Let's consider a simple method that sets a bit inside a field of type `long`:

```
long mask;

void setBit(int k) {
  mask |= 1 << k;
}
```

Here, `1` was written erroneously instead of `1L`, so the shift operation has the `int` type. According to the Java language specification, when you shift an `int` value by `k` bits, the actual amount of shift is determined by the least five bits of `k`, so the shift amount is always between 0 and 31. On the other hand, as our `mask` is long, users would normally expect that the `setBit()` method should work for `k` between 32 and 63 as well. But this is not the case. E. g., calling `setBit(40)` will actually set the 8th bit of the mask rather than the 40th. Depending on context, it's possible that unit tests cover only small values of `k`, so such kind of error could go into production unnoticed.

Static analysis

Static analyzers may catch the latter mistake, as it's suspicious when an `int` result of a left shift is implicitly cast to `long`. IntelliJ IDEA inspection "Integer multiplication or shift implicitly cast to 'long'" reports this problem.

Ways to avoid this mistake:

- Avoid littering your code with many bitwise operations. Instead, extract them to isolated, well-tested utility methods and reuse these methods.
- If you use bitwise operations on long values, write unit tests where higher bits are used.
- Always think about corner cases. Write unit tests for boundary inputs.
- Do not invent bitwise algorithms by yourself. Consult authoritative sources like Hacker's Delight, by Henry S. Warren, Jr. (O'Reilly Media).
- Think twice whether you actually need to pack several values into a single number manually. Consider using library classes. The `BitSet` class is a more convenient representation of bit string. It provides many methods with clear semantics that work for any inputs. Also, the number of bits is not limited to 32 or 64, so if you need more bits in future, you won't need to change your code much. The performance overhead is acceptable in many cases.
- If you want to represent a number of flags, consider declaring an enum that enumerates these flags using `EnumSet`. This will make your code much more clear and robust with very little additional cost.

4.13 Mistake #37. Forgetting about negative byte values

In Java, the `byte` primitive type allows storing a number between -128 and 127. This could be counter-intuitive, as usually people think of bytes as of values between 0 and 255. We already saw in mistake #33 how it may cause unexpected behavior when a `byte` value is updated using a compound assignment. However, problems are possible in other code patterns as well. For example, when it's necessary to compose a number from two individual bytes, the following code may look okay at first glance:

```
int readTwoBytes() {
  byte lo = readLowByte();
  byte hi = readHighByte();
  return (hi << 8) | lo;
}
```

Unfortunately, it works as expected only if bytes are within the range 0 and 127. Suppose that the call to `readLowByte()`returned -1. In this case, it's widened to the `int` type, and its hexadecimal representation will become `0xFFFFFFFF`. So, the whole `readTwoBytes` method will return -1, regardless of the `hi` value.

Static analysis

Static analyzers may report this problem sometimes. SonarLint has "S3034: Raw byte values should not be used in bitwise operations in combination with shifts" rule that reports it. However, it covers only specific code patterns, so even having the static analyzer, it's still possible to make such a mistake.

Ways to avoid this mistake:

- When performing bitwise operations in Java, remember that negative numbers are possible. If widening occurs, higher bits will become ones instead of zeroes. To cancel this effect, use a mask, like this:

```
return ((hi & 0xFF) << 8) | (lo & 0xFF);
```

- Encapsulate low-level bit manipulation operations into utility methods and test them really well. After that, they can be reused throughout a project without introducing additional problems.

4.14 Mistake #38. Incorrect clamping order

The following problem appears quite often in practice: having to adjust an input number so it fits the given interval. This could be written using an if-chain:

```
if (value < 0) value = 0;
if (value > 100) value = 100;
// value now in 0..100 interval
```

Or using a conditional expression:

```
value = value < 0 ? 0 : value > 100 ? 100 : value;
```

However, this may look too verbose. Also, this requires mentioning the lower and upper bounds twice and sometimes these are not simple numbers but something more complex.

One may avoid the repetition using a combination of `Math.min()` and `Math.max()` calls:

```
value = Math.max(0, Math.min(value, 100));
```

This looks smart and short. However, this construct is error prone. The whole expression is not very intuitive, so it's quite easy to accidentally swap the max and min calls:

```
value = Math.min(0, Math.max(value, 100));
```

Now the result is always 0, which is not evident from how this expression looks. We saw a bug like this in a procedure to render a progress bar: it got stuck at 0%. Also, a similar bug was discovered in a Jenkins CI.

Static analysis

Some static analysis tools detect such errors when bounds are constant values. IntelliJ IDEA reports the warning “Result of 'min' is the same as the first argument making the call meaningless”. SonarLint reports this problem via rule S3065: “Min and max used in combination should not always return the same value”. SpotBugs reports an “Incorrect combination of Math.max and Math.min” bug pattern. Unfortunately, static analyzers might not help if the bounds are the result of non-trivial computation, as they need to be able to compare them without executing your program.

Ways to avoid this mistake:

- Pay attention when using the `Math.min()` and `Math.max()` methods. Ask yourself which result you will get when the first operand is bigger and when it’s smaller. Occasionally, developers use `min` instead of `max` or vice versa in other contexts as well.
- As with every confusing code pattern, it’s best to extract it to a utility method and use everywhere. A method like this would work:

```
public static int clamp(int value, int min, int max) {
  if (min > max) {
    throw new IllegalArgumentException(min + ">" + max);
  }
  return Math.max(min, Math.min(max, value));
}
```

It also checks that the supplied min value doesn’t actually exceed the max value, so you can detect cases when this invariant is violated.

4.15 Mistake #39. Ignoring the difference between +0.0 and −0.0

There are two floating-point zeros: +0.0 and −0.0. They have distinct representation in computer memory but they appear to be equal if you compare them via “==”. You can get −0.0 as a result of some computations. For example, the following code prints −0.0:

```
System.out.println(-1 * 0.0);
```

This behavior is explicitly stated in the Java Language Specification §15.17.1. It has the following clause: “the sign of the result is positive if both operands have the same sign, and negative if the operands have different signs”. This behavior is not unique to Java. Any programming language that follows the IEEE 754 standard for floating point numbers behaves this way.

It should be noted that when a `double` value gets boxed into `Double`, +0.0 and −0.0 become different. You can see the difference in the following code:

```
Double d = 0.0;
System.out.println(d.equals(-0.0)); // false
System.out.println(d == -0.0); // true
```

`Double.equals()` considers these different values, but when comparing to a primitive −0.0, the `Double` value is unboxed and the primitive comparison returns true.

This difference may show up unexpectedly. For example, assume that you have values from the following set -1.0, -0.5, 0.0, 0.5, 1.0. Now, you want to get all the possible results of multiplication of two such numbers. You may want to store them into the set like this:

```
Set<Double> multiplications = new HashSet<>();
for (double i = -1; i <= 1; i += 0.5) {
  for (double j = -1; j <= 1; j += 0.5) {
    multiplications.add(i * j);
  }
}
System.out.println(multiplications);
```

Now you'll see that the resulting set contains 8 values instead of 7, as it includes both 0.0 and -0.0:

```
[1.0, 0.5, -0.0, -0.5, -1.0, 0.25, -0.25, 0.0]
```

You may also try `LinkedHashSet` and `TreeSet`. The resulting set will still contain 8 values, even though the order might be different.

Even if you never use boxed types like `Double`, you may run into this difference. For example, if you declare Java `record` (available since Java 16), you'll get methods like `equals()` and `hashCode()` for free. The question is how they are implemented. If you have a `double` component, the comparison rules will be the same, as for boxed `Double`. For example:

```
record Length(double d) {}
Length l1 = new Length(0.0);
Length l2 = new Length(-0.0);
System.out.println(l1.equals(l2)); // prints false
```

Ways to avoid this mistake:

- Do not forget that Java distinguishes +0.0 and −0.0 internally, and sometimes this difference shows up in your code as well. If necessary, normalize them before storing into fields or collections, like:

```
if (d == -0.0) d = 0.0;
```

  This line of code is somewhat strange, as +0.0 satisfies the condition as well, but ultimately it does the intended thing: replaces −0.0 with +0.0 without changing any other values.

- Avoid comparing floating-point numbers for exact value in general. It's probably not a good idea to store them in the set as well. Even if you never have negative numbers, you may not get exact comparison due to the limited machine precision. Here's a classical example:

```
Set<Double> set = new HashSet<>();
set.add(0.3);
set.add(0.1 + 0.2);
System.out.println(set);
```

One may expect that the set will contain only one number after these operations, but in fact it will contain two: [0.3, 0.30000000000000004]. As floating-point numbers are converted to binary, most finite decimal fractions like 0.1, 0.2, and 0.3 become infinite, so it's necessary to cut their binary value. As a result, there actually are no floating-point numbers that correspond to decimal 0.1, 0.2, and 0.3 exactly. When you are summing these approximate values, errors add up as well, and you may get a different decimal value when going back from binary.

4.16 Mistake #40. Forgetting to handle the NaN value

NaN stands for "not a number". `Double.NaN` and `Float.NaN` are very special values. These are the only values in Java that aren't equal to themselves. E. g. the following code prints `false` twice:

```
System.out.println(Double.NaN == Double.NaN);
System.out.println(Float.NaN == Float.NaN);
```

It's recommended to use the `Double.isNaN` or `Float.isNaN` library methods to check if a number is `NaN`. Explicit comparison to `Float.NaN` or `Double.NaN` is detected by most static analyzers (IntelliJ IDEA inspection name is "Comparison to Double.NaN or Float.NaN", and the Error Prone bug pattern name is "EqualsNaN") and nowadays such mistakes rarely appear in Java code. However, `NaN` could be implicit as well. For example, consider the following method:

```
void process(double a, double b) {
    if (a < b) {
        // process the a < b case
    } else if (a > b) {
        // process the a > b case
    } else {
        // process the a == b case
    }
}
```

The author of the code assumes that if `a` is not bigger than `b` and is not smaller than `b` then `a` is equal to `b`. However, it's also possible that either `a` or `b` is `NaN`. In this case, the last branch will be taken, and the results could be unexpected.

Ways to avoid this mistake:

- When comparing floating-point numbers always ask yourself if it's possible to have `NaN` at this point.
- If `NaN` values are not expected, consider adding asserts:

```
assert !Double.isNaN(a);
assert !Double.isNaN(b);
```

For public APIs, use preconditions:

```
if (Double.isNaN(a) || Double.isNaN(b)) {
    throw new IllegalArgumentException();
}
```

This will allow catching the bug earlier if `NaN` values unexpectedly appear in this method.

4.17 Summary

- Octal numerals may be used accidentally resulting in a different numeric value. Prohibiting them completely in your codebase is a good idea.
- Numeric overflow is one of the most annoying bugs in programming. When you have numerical calculations in your program, ask yourself whether overflow is possible and how the program should behave in this case.
- There are some numbers that behave unusually. This includes `Integer.MIN_VALUE` (equals to itself when negated), -0.0 (equals to +0.0) and `Double.NaN` (not equal to itself). Check if your code handles such inputs correctly.
- Compound assignment operators are tricky, as they may perform implicit narrowing conversion and change the evaluation order, producing a different result than expected.
- When speaking of bytes, developers usually imagine a number between 0 and 255, but the `byte` type in Java ranges from -128 to 127.
- Avoid the `short` type in favor of `int`, and the `float` type in favor of `double`. You'll rarely get the performance benefit or reduced memory usage of `short` and `float`, but they may introduce problems due to reduced precision or overflow.
- Many numeric algorithms like oddness check, clamping values, and setting a bit in the number are deceivingly simple, so developers prefer writing them in-line. Yet it's quite possible to write them incorrectly. Isolate every algorithm in a separate, well-tested method.

5 Common exceptions

This chapter covers

- **Ways to avoid NullPointerException in Java programs**
- **Which mistakes cause IndexOutOfBoundsExceptions**
- **Different reasons for ClassCastException**
- **Finite and infinite recursion in Java**

In this chapter we discuss the most common virtual machine exceptions that happen in Java programs due to bugs. This chapter differs somewhat from the others, as we will concentrate on the effect of the bugs, rather than their causes.

All the exception classes in Java inherit a single `Throwable` class. Any exception can be thrown explicitly via a `throw` statement. Some exceptions can be thrown by the virtual machine itself. Here, we will mainly speak about such exceptions. Figure 5.1 shows the hierarchy of exceptions covered in this chapter.

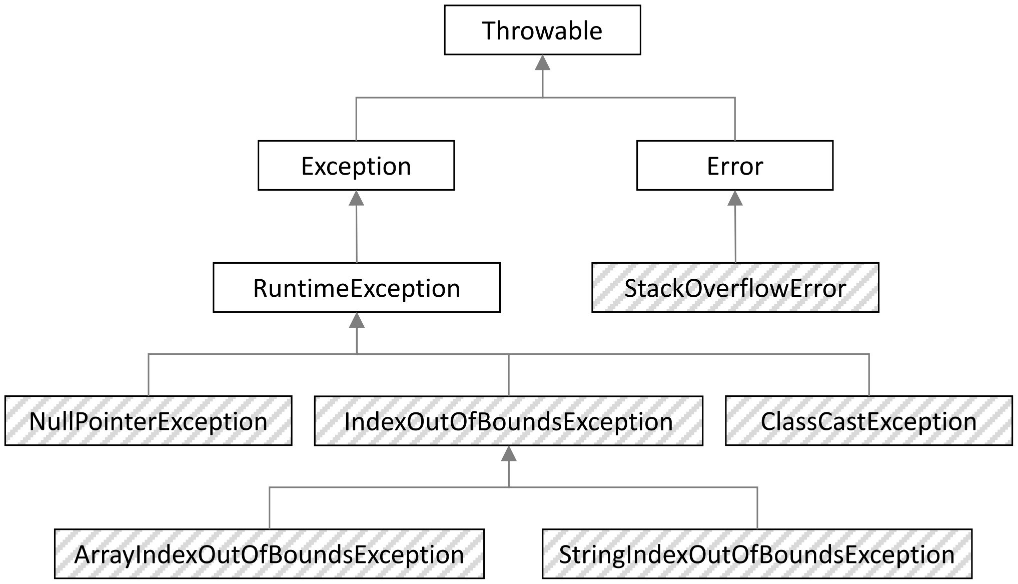


Figure 5.1 Hierarchy of exception classes. All Java exceptions inherit `Throwable`, and many of them have `Exception`, `Error` or `RuntimeException` as superclass. Exceptions covered in this chapter are shaded.

The common thing to understand about exceptions is that exceptions are your friends, not enemies. Many programmers dislike exceptions, but they are there to help you to write programs without bugs. It's much better to have an exception as early as possible instead of continuing the execution of your program in an incorrect state, which may cause much more severe consequences like data loss or a security breach.

5.1 Mistake #41. NullPointerException

`NullPointerException` constantly chases Java developers. Java language design cannot prohibit storing null into non-primitive variables or returning null from methods whose return type is non-primitive. Developers often use null as a sentinel value for many purposes:

- An uninitialized value during object construction or just after creation of an array
- A lazily initialized value for which a non-null value will be written the first time anybody needs this value
- An unspecified value, for example a `middleName` field in a `User` class for users who have no middle name or prefer not to specify it
- A not found value, for example, a library method `Map.get` that returns null if the corresponding mapping is not found in the map
- An empty collection sentinel, for example methods of a `Queue` interface like `poll()`, or `peek()` that return null if the queue is empty

And so on. The problem is that the null value itself does not bear any semantics. So, when a Java developer encounters a reference value, the following questions arise: is it possible that null is stored into this value, and if yes, what exactly does it mean?

From the developer's point of view, many variables never contain null, and many methods never return null. We can call them as non-nullable variables and non-nullable methods. Other variables and methods may contain or return null under some circumstances. Let's call them nullable. Some programming languages, like Kotlin, encode value nullability directly in the type system. There, you can declare a variable of non-nullable type, and the compiler will prohibit storing a potentially nullable value. Also, if the value type is nullable, you cannot dereference it directly without a null-check or a special safe-call operator. Thanks to this, an unexpected `NullPointerException` occurs in Kotlin programs much less often than in Java. Unfortunately, Java has nothing like this, and it's unlikely that things will change in the foreseeable future.

To reduce the risk of `NullPointerException`, there are several main approaches:

- Avoid null values as much as possible in your code base.
- Use annotations to augment the Java type system with nullability information.
- Use advanced interprocedural static analysis tools that can detect many potential `NullPointerException` problems.

Let's cover these in more detail.

5.1.1 Avoiding nulls and defensive checks

To avoid null values as much as possible, you should not return null from your methods and you should never accept null as a parameter. Well, sometimes it's hard to do. For example, if you implement an interface or extend a class that is declared in a standard library and by contract requires returning null under some circumstances. Think of implementing your own `Map`. In this case, you are required to return null from the `Map.get` method if the supplied key is absent in your map. Some people go even further and avoid using APIs that require returning or accepting nulls. Unfortunately, this approach is not always practical.

Another important thing is sanitizing the input values, at least in public methods and constructors. Just add `Objects.requireNonNull()` call for every reference parameter you receive, right at the method entry. If a non-null parameter is supplied, this method just returns the parameter, so by using this return value one can conveniently initialize fields:

```
public class Person {
  private final String firstName, lastName;

  public Person(String firstName, String lastName) {
    this.firstName = Objects.requireNonNull(firstName);
    this.lastName = Objects.requireNonNull(lastName);
  }
}
```

Doing this helps you to catch a `NullPointerException` as early as possible. If you don't do this, you may accidentally store a null value into a field, effectively putting the program into an invalid state. As a result, the problem will occur much later when you dereference the field, and you'll have no clue how the null value appeared in the variable.

What if your method accepts a collection as the input? You can also traverse it at the method entry to ensure that it doesn't contain nulls:

```
public void process(Collection<String> data) {
  data.forEach(Objects::requireNonNull);
  ...
}
```

Unfortunately, collection traversal is not free. It costs CPU time that depends on the collection size, and this could be unacceptable, especially if you pass the same collection through many methods and every method checks all the collection elements. One possibility is using the `List.copyOf`, `Set.copyOf` and `Map.copyOf` collection factory methods included since Java 10:

```
public void process(Collection<String> data) {
  data = List.copyOf(data);
  ...
}
```

These methods produce unmodifiable copy and disallow nulls. But what's more important, if the input collection is already a result of `copyOf` method, it doesn't make an extra copy and simply returns the argument. So, if you want to pass the same collection through many methods, adding `copyOf` to every method will result in at most one copying and null check.

Even if you think that avoiding nulls everywhere is too restrictive, and you consider null as a valid input or output sometimes, there are certain data types where using the null value is a bad practice:

- Collections, iterables, arrays, and maps. In these cases, the absence of data is usually designated by an empty collection, an empty array or an empty map. It's a bad practice to assign different semantics to null and empty collections. Many clients won't expect null instead of an empty container and, as a result, a `NullPointerException` may happen at runtime.
- Unfortunately, this problem appears in the Java standard library itself. The `java.io.File` class contains methods like `list()` and `listFiles()`, which return `null` in case of any I/O error. It's quite convenient to iterate through the directory files using a for-each loop:

```
for (String fileName : new File("/etc").list()) {
    System.out.println(fileName);
}
```

However, if there's no '/etc' directory on the file system, instead of graceful error handling you will get a `NullPointerException`. Luckily, the newer NIO file API (e. g., the `Files.list` method) addresses this problem. Here, a proper `IOException` is thrown in case of error, so you will have detailed information what happened and may handle this case:

```
try(Stream<Path> stream = Files.list(Path.of("/etc"))) {
    stream.forEach(System.out::println);
}
```

- `Stream` or `Optional` type. Again, use `Stream.empty()` or `Optional.empty()` to denote an empty result. Returning null is especially bad here, as `Stream` and `Optional` classes are designed for a fluent call chains style, so it's quite expected that users will immediately call something like `map()` on the result of the method that returns `Stream` or `Optional`.
- Enum type, especially if we are speaking about enum declared in the project, not in the standard library. In this case, the possibility to return null is just another option, so it's much better to encode this option directly as an enum constant. Doing this, you can choose a good name, `NOT_FOUND`, `UNSPECIFIED`, `UNKNOWN`, `UNINITIALIZED`, or whatever else, and provide clear documentation. For example, the `java.math.RoundingMode` enum from Java standard library specifies how to round the result of calculation. Sometimes, it's desired not to round at all. In this case, null could be used to specify the absence of rounding, but the JDK authors wisely decided to create a separate constant `RoundingMode.UNNECESSARY`. This constant declaration contains an explanation what will happen if the computation cannot be performed without rounding (an `ArithmeticExpression` is thrown), so it's unnecessary to explain this at every method that uses `RoundingMode`.

  Sometimes, you may find out that different methods returning the same enum value assign a different meaning to the null result, so there's a chance that it will be incorrectly interpreted. You can add several distinct enum constants and replace a null result with the appropriate one, resolving the ambiguity.
- `Boolean` type. In general, using primitive wrapper types like `Boolean` or `Integer` is suspicious, unless you are instantiating the type parameter, like `List<Integer>`. In essence, the `Boolean` type is just an enum containing two values, `Boolean.TRUE` and `Boolean.FALSE`. If you assign a special semantics to a null value, you are turning it into a three-value enum. In this case, it would be much less confusing to declare an actual three-value enum.

Using enum instead of `Boolean` will also allow giving better names for `TRUE` and `FALSE` values. For example, assume that you are working on a web-shop application, and you want to handle the case when a user removes an item from the basket. You want to ask the user whether they want to move the item to the wish list instead. So, you are going to display a confirmation window, like the one shown in figure 5.2.

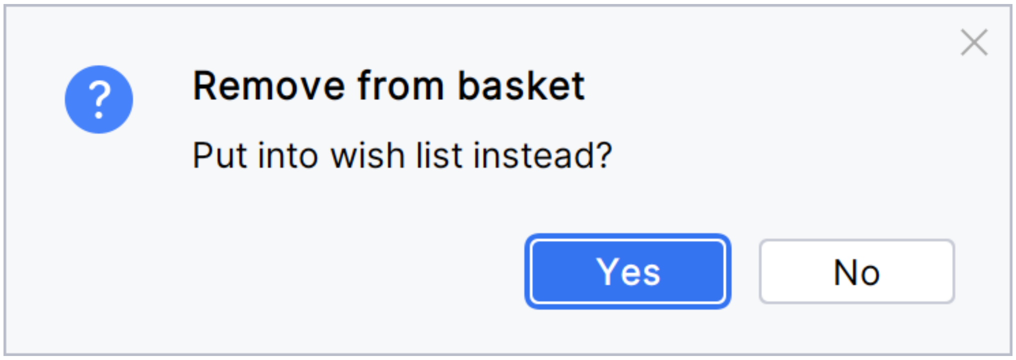


Figure 5.2 Confirmation window with two possible outcomes.

In the code you decided to return a `boolean` value, like this:

```
boolean shouldMoveToWishList() {
  ...
  // Show dialog and return "true" if user pressed "Yes"
}
```

Later you realize that at this point the user may reconsider the removal and want to cancel the whole operation. So, you are going to add a “Cancel” button to the dialog, as shown in figure 5.3.

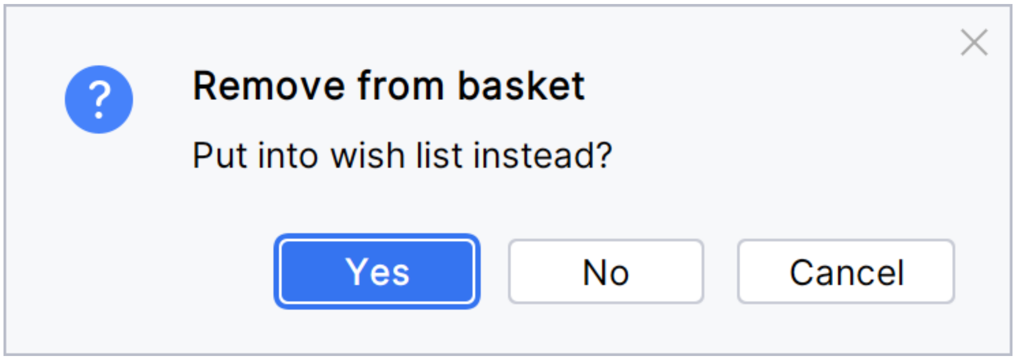


Figure 5.3 Confirmation window with three possible outcomes.

Now, the `shouldMoveToWishList()` method should return three different values, and it might be tempting to change the return type to an object `Boolean` and use null for “Cancel”:

```
Boolean shouldMoveToWishList() {
  ...
  // Show dialog and return "true" if user pressed "Yes",
  // "false" if user pressed "No" and "null" if user pressed "Cancel"
}
```

People do this because it looks like an easiest approach. You don't need to change much at the call sites after that. However, this is quite dangerous. Suppose that the method is used like this:

```
if (shouldMoveToWishList()) {
  addItemToWishList(item);
}
deleteItemFromBasket(item);
```

After updating to a `Boolean` type, you won't have a compilation error here, so you may easily forget to refactor this code. Now, when the user presses Cancel, automatic unboxing occurs at the `if` statement condition, and you'll get a `NullPointerException`.

It would be much better to introduce an enum and put all three values there. You may either declare a more generic enum, which can be reused in other contexts:

```
enum UserAnswer {
  YES, NO, CANCEL
}
```

Alternatively, you may prefer a specific enum for this particular scenario:

```
enum ItemRemovalRequest {
  REMOVE_FROM_BASKET,
  MOVE_TO_WISH_LIST,
  CANCEL
}
```

In this case, a compilation error will force you to fix the code at the call site. With enums, you can use `switch` statement instead of `if` statement:

```
switch (shouldMoveToWishList()) {
  case REMOVE_FROM_BASKET -> deleteItemFromBasket(item);
  case MOVE_TO_WISH_LIST -> {
    addItemToWishList(item);
    deleteItemFromBasket(item);
  }
  case CANCEL -> { /* do nothing */ }
}
```

Now the possible outcomes are much clearer, and readers of the code don't need to guess what null means in this particular context. Another advantage of enum is that you can add more options there in the future, if necessary.

Ways to avoid this mistake:

- Use defensive checks via `requireNonNull` and similar methods at your API boundaries if you don't expect nulls.
- It could also be reasonable to check that input collections contain no nulls inside, though this may imply a performance cost.
- Avoid returning nulls when your variable type is a collection, Stream, or Optional. Prefer returning an empty container instead.
- Avoid returning arrays, collections, or maps that contain null elements. This is a reasonable recommendation, even if you prefer a more null-friendly approach. The callers of your methods usually don't expect that a returned array or collection could contain nulls.
- If you encounter an error such as "file not found" and you cannot recover from it by yourself, it's better to throw an exception containing the detailed information instead of returning null.
- If you have an enum, consider introducing a special enum constant instead of returning null.
- Avoid boxed primitives like `Integer` or `Boolean`. Instead of using `Boolean`, declare an enum with three constants.

5.1.2 Using Optional instead of null

Some developers who want to avoid null, prefer to use the `Optional` type instead. I imagine the `Optional` variable as a box that could be either empty or contain a non-null value.

Note that the primary purpose of an empty `Optional` is to designate the absence of a value. For example, if you are writing a search method that could find nothing, you can wrap the result into `Optional`:

```
public Optional<User> findUser(String name) {
  // return Optional.empty() if not found
}
```

This way, you're clearly signaling to the API clients that the absent value is possible, and they must deal with it. The Optional type provides useful methods to transform the result in a fluent way. For example, assuming that the `User` class has a `String fullName()` method, you can safely use a call chain like this:

```
String fullName = findUser(name).map(User::fullName)
                                .orElse("(no such user)");
```

Here, we extract a full name, replacing it with "(no such user)" string if `findUser()` returned an empty `Optional`.

Using `Optional` as a return value is a good idea. However, it's inconvenient if you use `Optional` as a method parameter, as all the clients will be forced to wrap input values. Of course, you should never return `null` from a method whose return type is `Optional`. There are different opinions on whether it's okay to create fields of `Optional` type.

There are three standard ways to get an `Optional` object:

- `Optional.empty()` to get an empty `Optional`;
- `Optional.of(value)` to get a non-empty `Optional` throwing a `NullPointerException` if the value is null.
- `Optional.ofNullable(value)` to get an empty `Optional` if the value is null and non-empty `Optional` containing value otherwise.

One should be careful with using the last method, as it's suitable only when the null value should be converted to the empty `Optional`. Some developers prefer using `ofNullable()` everywhere because it's "safer", as it never throws an exception. However, this safety is deceiving. If you already know that your value is never null, it's better to use `Optional.of()`. In this case, if you see an exception, you will immediately know that your knowledge was wrong, and there's probably a bug in the previous code. If you have used `ofNullable()`, you will silently get an empty `Optional`, so your program will not throw an exception but will behave incorrectly, and you may not notice this until much later.

5.1.3 Nullity annotations

One popular approach to minimize possible `NullPointerException` problems in Java is to use nullity annotations. You can use one of popular annotation packages listed in Chapter 1. If you don't like external dependencies, it's also possible to declare such annotations right in your project. Usually, static analysis tools allow you to configure custom nullity annotations.

The two most important annotations are usually spelled as `@Nullable` and `@NotNull` (or `@Nonnull`). The `@Nullable` annotation means that it should be expected that the annotated variable contains null or the annotated method returns null. In contrast, `@NotNull` means that the annotated value is never null. These annotations document the code and provide hints for static analysis. The most common problems reported by static analysis tools is dereferencing a nullable value without a null-check and checking a not-null value against null:

```
import org.jetbrains.annotations.*;

interface MyInterface {
  @NotNull String getNotNull();
  @Nullable String getNullable();

  default void test() {
    // Warning: condition is always false
    if (getNotNull() == null) {...}

    // Warning: trim() invocation may cause
    // NullPointerException
    System.out.println(getNullable().trim());
  }
}
```

While checking a not-null value against null might not look like a real problem, such warnings may help to remove the redundant code. Sometimes, it may point to the actual mistake. For example, if you are working with XML DOM API and want to read a value of an XML element attribute, you can find the `getAttribute()` method inside the `org.w3c.dom.Element` class.

It accepts the attribute name and returns the attribute value. It might be quite natural to expect that null will be returned if the specified attribute is missing. So, the following code might look correct:

```
String getName(Element element) {
  String name = element.getAttribute("name");
  if (name == null) {
    return "<default>";
  }
  return name;
}
```

However, this code is wrong. According to the `getAttribute` documentation, this method returns an empty string instead of null if the attribute is not found. If it's desired to distinguish between an empty attribute and an absent attribute, the `hasAttribute` method should be used. While the Java standard library is not annotated with any nullability annotations, some static analyzers like IntelliJ IDEA, SpotBugs, and PVS-Studio have so-called external annotations applied to the standard library. As a result, the analyzer knows that the `getAttribute` method never returns null and warns you about an always false condition `name == null`.

Another common source of similar mistakes is the Reflection API. There are many query methods like `Class.getMethod()`, `Class.getField()`, and so on. These methods never return null. Instead, they throw an exception like `NoSuchMethodException` in case if the corresponding object is not found. We often saw that developers check the result of this method against null and ignore the exception. Static analyzers may help to identify such problems.

Some annotation packages allow you to specify default nullity for a whole class or even package. It could be handy if your API is completely or mainly null-hostile. For example, the Eclipse JDT Annotation Package contains a `@NonNullByDefault` annotation. You can apply it to a class or a package and specify what kind of source code elements:

```
import org.eclipse.jdt.annotation.*;

@NonNullByDefault({DefaultLocation.PARAMETER,
                   DefaultLocation.RETURN_TYPE})
public interface MyInterface {
  String processString(String input);
}
```

Here, it's specified that all the method parameters and method return values in `MyInterface` are non-null by default. In particular, the `processString()` method does not accept null as a parameter and never returns null as a return value. Using such an annotation can greatly simplify nullity specification.

Sometimes, it's useful when a static analyzer knows not only whether the method accepts null but also what exactly will happen if null is passed. For example, the `Class.isInstance()` standard library method always returns `false` when a `null` argument is passed. If the analyzer knows this, it may behave better. Consider the following code example:

```
default void processObject(@Nullable Object obj,
                           @NotNull Class<?> cls) {
  if (!cls.isInstance(obj)) return;
  int hash = obj.hashCode();
  // process hash
}
```

As `obj` is marked as `@Nullable`, and there's no explicit null-check, a naïve analyzer might issue a warning at the `obj.hashCode()` call that it may lead to a `NullPointerException`. However, this is in fact impossible. If `obj` is null then the result of the `isInstance()` call is false and we don't reach the `obj.hashCode()` call. To specify this knowledge about methods behavior, IntelliJ IDEA uses the contract annotation applicable to methods. This annotation is available in the JetBrains annotations package. For the `isInstance()` method, the contract could be specified as follows:

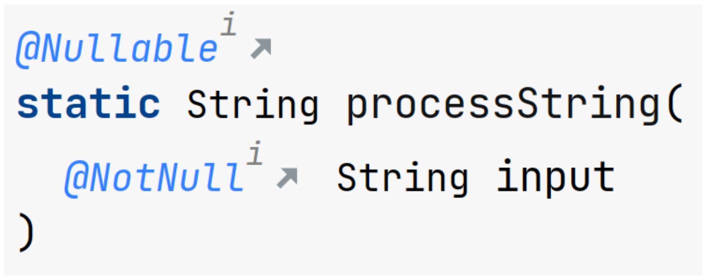
```
@Contract("null -> false")
```

This means "if the method argument is null then its result value is false". IntelliJ IDEA understands such annotations and takes them into account during the analysis, so you won't see a false-positive warning in the code example above. As with nullity annotations, standard library is pre-annotated with contracts. For your code, you can use the contract annotation explicitly.

Even if nullity annotations are used in the project, often developers have to deal with unannotated code. This could be legacy parts written before nullity annotations were introduced, or modules imported from another projects. IntelliJ IDEA may still help in this case, as it tries to infer nullity annotation automatically. For example, you may write the following method:

```
static String processString(String input) {
  String result = input.trim();
  if (result.isEmpty()) return null;
  return result;
}
```

If you open auto-generated documentation for this method, using the "Quick documentation" shortcut, you'll see the annotations marked with 'i' letter (figure 5.4).



```
@Nullable^i ↗
static String processString(
  @NotNull^i ↗ String input
)
```

Figure 5.4 Inferred annotations in IntelliJ IDEA are marked with a superscript 'i'

This means that the annotation is inferred automatically: IntelliJ IDEA has checked the source code of your method and determined the nullity of the parameter and the return value. Unfortunately, this works only in rare cases, as IntelliJ IDEA must analyze your code online while you continue editing it, so it's pretty limited in terms of the CPU time it can spend on this analysis.

IntelliJ IDEA also provides a separate "Infer Nullity" action (available under the Code | Analyze Code menu). This is a more advanced mechanism to annotate your project or its parts automatically. Still, it's not ideal. There are more advanced static analyzers like Infer that can perform deep interprocedural analysis to find non-trivial potential `NullPointerException` problems.

5.2 Mistake #42. IndexOutOfBoundsException

The good thing about Java is that it does bounds checks for you. You cannot accidentally read or write beyond the array length: you will get an `ArrayIndexOutOfBoundsException` from the virtual machine instead of a silently malfunctioning program. The same principle is followed in the standard library. If you try to access string characters beyond the string length (e. g., via the `charAt` or `substring` methods) or list elements beyond the list size, you'll also get an appropriate exception like `StringIndexOutOfBoundsException` or simply `IndexOutOfBoundsException`.

However, these exceptions are intended only to identify bugs in your program and prevent worse consequences like data corruption, incorrect behavior, or security holes. In fact, you still need to do bounds checks by yourself. Various flavors of `IndexOutOfBoundsException` are quite a common problem in Java programs.

For example, suppose that we want to check whether the first character of the string is a digit. It's easy to forget a bounds check in this code:

```
void processString(String s) {
  if (Character.isDigit(s.charAt(0))) {
    // process strings starting with digit
  }
}
```

This code behaves well until the empty string is supplied as the input. I have encountered similar bugs with arrays and lists as well.

Sometimes a bounds check is present but it's implemented incorrectly. I've seen three typical problems with it:

- The bounds check is performed after the index access, probably due to refactoring or a copy-paste error. For example, here, it was intended to check `index < data.length` before `data[index] >= 0`:

```
void processArrayElement(int[] data, int index) {
  if (data[index] >= 0 && index < data.length) {
    ...
  }
}
```

- Relation is accidentally flipped. For example, here `data.length > index` was intended:

```
void processInput(int[] data, int index) {
  if (data.length < index && data[index] >= 0) {
    ...
  }
}
```

- Off-by-one error: there's a corner case when the index is exactly equal to the array, list, or string length. Such an index is out of bounds, but sometimes developers don't pay attention to this case. For example, here it was intended to exit the method if the `index >= data.length`:

```
void processInput(int[] data, int index) {
  if (index > data.length) return;
  if (data[index] >= 0) {
    ...
  }
}
```

Static analysis

The first two cases are detected by IntelliJ IDEA static analyzer, unless a complex expression is used as the index. Don't forget to help the static analyzer by extracting the repeating complex expression to intermediate variables. The third problem can be detected by PVS-Studio analyzer (diagnostic name: "V6025. Possibly index is out of bound").

Things become trickier if you need to perform a bounds check before accessing a contiguous range of indexes. A common example is filling part of an array, given a starting offset and number of elements to fill. Normally, you should check that the starting offset and count are non-negative and the array size is at least equal to the sum of the starting offset and count. Additional care should be taken, as sum of offset and count may overflow, so simple check like `offset + count <= length` might not be enough.

In some cases, you just need to throw an exception if the supplied offset and count are incorrect, without doing anything else. For example, you may need this if you implement a three-argument `InputStream.read` method. In this case, it's convenient to use the `checkFromIndexSize` method, which is included in the `java.util.Objects` class since Java 9:

```
class MyInputStream extends InputStream {
  @Override
  public int read(byte[] b, int off, int len) {
    Objects.checkFromIndexSize(off, len, b.length);
    ...
  }
}
```

There are also the methods `checkIndex` and `checkFromToIndex`, which could be helpful. For example, you can use them if you implement your own `List`:

```
class MyList<T> implements List<T> {
  public T get(int index) {
    Objects.checkIndex(index, size());
    ...
  }

  public T set(int index, T element) {
    Objects.checkIndex(index, size());
    ...
  }

  public List<T> subList(int fromIndex, int toIndex) {
    Objects.checkFromToIndex(fromIndex, toIndex, size());
    ...
  }
  ...
}
```

Note, however, that these check-methods are not always suitable. For example, if you implement `List.add(index, element)`, the index is allowed to be equal to the list size (but not bigger), so by using `checkIndex` you'll violate the interface contract.

Static analysis

Some static analyzers support annotations to specify the allowed range of method parameters. Sometimes, it helps to catch the problem during the development, before you actually hit the exception. For example, Checker framework annotations package contains `@NonNegative` annotation which denotes that the method parameter cannot be negative. So, if you use a method parameter as index, you can annotate it:

```
int getValue(@NonNegative int index) {
    return myData[index];
}
```

In this case, static analyzer may warn you if you call this method with a negative argument. JetBrains annotation package provides a `@Range` annotation that allows to specify explicitly the lower and upper bounds. An equivalent for `@NonNegative` for `int` parameter would be `@Range(from = 0, to = Integer.MAX_VALUE)`.

Ways to avoid this mistake:

- If you accept an index value from the clients, check its bounds immediately. The sooner the mistake is discovered the better.
- Be careful when using numeric comparisons with indexes. It's easy to mix up the <, <=, >, and >= operations. Substitute some example values in your mind and think about which result your condition should have. For example, if the size of the array or list is 5 and the index is 4, should it be true or false? What if both index and size are equal to 5?
- Consider using range annotations to document your code better and help static analysis tools.

5.3 Mistake #43. ClassCastException

There are two main sources of a `ClassCastException` in Java. The first one is explicit type cast expression and the second one is an implicit cast added when a generic type is instantiated.

5.3.1 Explicit cast

To avoid `ClassCastException` on explicit cast, it's recommended to protect every explicit cast with the corresponding `instanceof` check:

```
if (obj instanceof MyType) {
    MyType myType = (MyType) obj;
    // use myType
}
```

This code pattern is so important that since Java 16, the `instanceof` expression was enhanced, and you can simplify it:

```
if (obj instanceof MyType myType) {
    // use myType
}
```

The `MyType myType` clause is called a 'pattern'. You can learn more about patterns in the Java 16 language specification or various tutorials. The important thing here is that you can avoid explicit cast and protect yourself from `ClassCastException`, as the pattern-based `instanceof` expression never throws one.

Unfortunately, many projects still cannot migrate to Java 16+ and even if they can tons of code was written in older Java versions where you should spell the same type in `instanceof`, cast expression, and a variable type (unless you are using the `var` keyword). As usual, bugs like repetitions, so if you need to repeat something, there are chances that things will go wrong. Sometimes, these types differ due to a typo or a mistake introduced during refactoring.

Another way to check the type before cast is calling `getClass()`. This code pattern appears often in `equals()` method implementations, especially when the class is not final:

```
public class Point {
  final int x, y;

  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Point point = (Point) o;
    return x == point.x && y == point.y;
  }
}
```

Here, the cast is also safe, because the `getClass()` called on `this` object is either `Point` or its subclass. A possible mistake in this case appears when a developer copies the `equals` method implementation from another similar class without updating the cast type. To avoid such a problem, do not copy an `equals()` method from another class, even if that class is very similar. Instead, use IDE features to automatically generate the `equals()` implementation from scratch.

However, most of `ClassCastException` problems appear when the cast is performed without the previous `instanceof` or `getClass()` check, when developer is completely sure that the cast will succeed.

One particularly bad case is casting the method parameter unconditionally upon the method entrance:

```
public void process(Object obj) {
  String str = (String) obj;
  // Use str
}
```

In this case, while the method declares the `Object` parameter type, in fact it accepts `String` parameters only. As a result, we cannot rely on the compiler's type checks and should be extremely careful to pass a value of the proper type. It would be much better to change the method declaration:

```
public void process(String str) {
  // Use str
}
```

Sometimes, compatibility concerns arise. There are probably third-party clients that already call `process(Object)` and you cannot update them immediately. In this case, it's a good idea to declare a deprecated overload:

```
/**
 * @param obj object to process (must be a String)
 * @deprecated use {@link #process(String)} instead.
 */
@Deprecated
public void process(Object obj) {
  process((String) obj);
}

public void process(String str) {
  // Use str
}
```

Now, while older clients will work as before, you will have a deprecation warning at the call sites where non-string arguments are used. Eventually all the clients will update and the deprecated overload can be removed.

The problem becomes more complicated when your method overrides or implements the method from a superclass or interface. Sometimes it's impossible to modify the supertype and you will have to accept more generic type. Still, you can at least localize the cast in the intermediate method and produce a better error message in case an unexpected object is supplied. For example:

```
@Override
public void process(Object obj) {
  if (!(obj instanceof String)) {
    throw new IllegalArgumentException(
                 "Unexpected object passed: " + obj);
  }
  process((String) obj);
}

private void process(String str) {
  // Use str
}
```

Here, we must accept an `Object` according to the interface we're implementing. However, we guard the cast with a condition where we can throw a custom exception. Here we added a string representation of a supplied object, but in real code you can provide more context to help debug the unexpected value and find out where it comes from.

If you can modify the supertype but it's hard to adapt all the inheritors, you can consider declaring a generic parameter. E. g., suppose you had an interface like this:

```
interface Processor {
    void process(Object obj);
}
```

If your particular implementation of `StringProcessor` accepts only the `String` parameter, you can declare a type parameter on the interface:

```
interface Processor<T> {
    void process(T obj);
}
```

Of course, this would require subsequent refactoring of all inheritors and users of this interface, as raw type warnings will appear everywhere. However, this can be done incrementally, as the code will still compile even with raw types. You may declare `StringProcessor implements Processor<String>` and other implementations `implement Processor<Object>`. It's also possible that, during the refactoring, you will discover more implementations that work with a specific subtype only, so the type parameter will be useful in other places as well.

Static analysis

There's a "Cast conflicts with instanceof" inspection in IntelliJ IDEA (turned off by default) that can detect if you have a cast under an `instanceof` check and the cast type differs from `instanceof` type. While it occasionally produces false warnings, it might be helpful. If the cast type is obviously incompatible with the `instanceof` type, "Nullability and data flow problems" inspection may also warn you.

SpotBugs also has a bunch of warnings including "Impossible cast" and "Impossible downcast" for cases where the cast is completely impossible. Other warnings like "Unchecked/unconfirmed cast" and "Unchecked/unconfirmed cast of return value from method" report cases when a cast can fail under certain conditions. However, these warnings are noisy and may produce false-positives.

Another useful inspection in IntelliJ IDEA is called "Too weak variable type leads to unnecessary cast". It reports type casts that could be removed if you use more concrete variable type instead. For example:

```
List<String> list = List.of("a", "b", "c", "d");
for (Object s : list) {
    System.out.println(((String) s).repeat(2));
}
```

Here, you can declare the variable 's' as `String` and remove the cast completely, and inspection suggests you should do so. This code may not cause `ClassCastException` in its current shape, but who knows what will happen later. If you can avoid cast, then it's better to avoid it and rely on static type checking instead. So it's perfectly reasonable to fix all the warnings of this inspection, especially taking into account that the automated quick-fix is provided.

Ways to avoid this mistake:

- Since Java 16, it's strongly encouraged to prefer pattern matching with `instanceof` syntax to blend `instanceof` and cast operations together. This way you mention the target type only once, avoiding copy-paste mistakes.
- When implementing the `equals()` method, don't copy the implementation from another class. Instead, use IDE capabilities to generate this method from scratch. This way you'll avoid copy-paste mistakes such as forgetting to update the cast type.
- Avoid unguarded casts without a previous `instanceof` check. If you are sure that your variable always contains an object of a specific type, then you should probably update the declared variable type to that specific type. If it comes from the method parameter or the method return type, consider updating them as well and remove the cast if possible. This way, an object of incompatible type could not be stored there at all, as this will be a compilation error.
- Use code template generation mechanisms to write `instanceof`-cast code sequences automatically. IntelliJ IDEA has the 'inst' live template for this purpose. Just type 'inst' in the editor and press Enter. It will ask you to select the variable and the target type. The rest will be generated automatically.

5.3.2 Generic types and implicit casts

Another common source of `ClassCastException` is an implicit cast when a generic type is instantiated. In this case, you have no cast expression in the source code which could be confusing. For example, consider the following method:

```
void printFirstItem(List<String> list) {
  String s = list.get(0);
  System.out.println(s);
}
```

The `List` class is declared as `List<E>` and the `get()` method returns the `E` type. However, as generics are erased at runtime, from the virtual machine point of view, there is no `E` type, and the `get()` method simply returns an `Object`. To convert the `Object` type to `String`, a Java compiler inserts an implicit cast, desugaring the method to the following:

```
void printFirstItem(List list) {
  String s = (String) list.get(0);
  System.out.println(s);
}
```

This is how the method looks to the virtual machine. It works with the raw type `List`, but every time you call a method that returns its type parameter (like the `get()` method), the implicit cast operation is performed. This cast might fail if the original list was not actually a list of strings. Such situation may happen if you have an unchecked cast or unchecked call somewhere at the `printFirstItem` call site. For example:

```
List<String> list = new ArrayList<>();
List<?> objects = list;
((List<Object>) objects).add(1); // unchecked cast
printFirstItem(list);
```

Here, we first assign `list` to a variable `objects` of the more generic type `List<?>`, which is allowed. However, after that we perform an unchecked cast of `objects` to `List<Object>`. The compiler issues an "unchecked cast" warning here which means that it won't be checked at runtime whether such a cast is actually possible. In fact, as types are erased at runtime, the cast to `List<Object>` disappears in the bytecode completely. We just add a number to the list, and nobody stops us. This essentially creates a time bomb: now the list is broken but the program continues executing as if everything is correct. The bomb explodes later, when the `printFirstItem()` method is executed and we try to get the string.

As you can see, an unchecked cast may produce a problem much later, so it's better to avoid it. I cannot advice to remove them completely, as in rare cases they are inevitable. However, in many cases you can avoid an unchecked cast. For example, imagine that you have two methods like this:

```
List<String> getList() { … }
void processList(List<CharSequence> list) { … }
```

Now you want to pass the result of `getList()` to the `processList()`. Even though `String` implements the `CharSequence` interface, as a general rule it's not safe to pass the

List<String> where List<CharSequence> is expected. That's because, if processList() modifies the list, then it may add another implementation of CharSequence (for example, StringBuilder) to the list, so List<String> will contain not only strings.

However, you may know in advance that processList() will not modify the list at all. In this case, you know that passing List<String> to processList() is safe. However, you cannot simply call processList(getList()), as it will be a compilation error. To fix the compilation error, you may consider adding an upcast to List<?> and then an unchecked cast:

```
processList((List<CharSequence>)(List<?>)getList());
```

But it still produces a warning. How to convince Java that it's safe? A good solution here is to declare the processList() method using a wildcard type with ? extends:

```
List<String> getList() { … }
void processList(List<? extends CharSequence> list) { … }
```

Now, processList() says that it accepts a List of any particular CharSequence. This signature still allows for querying list elements inside the processList(), but now you cannot add new elements (except nulls) without unchecked casts. This is fine, as you weren't going to add anything.

Sometimes you cannot modify the signature of the method. It could be a library method, or the signature could be enforced by a parent interface. In this case, you can still fix the problem at the call site without an unchecked cast:

```
processList(Collections.unmodifiableList(getList()));
```

The unmodifiableList() library method has a nice feature. Aside from wrapping your list in an unmodifiable wrapper, it can also adapt the list type. If you have custom generic types, don't hesitate to write read-only or write-only adapters.

Another way to step on such a problem is using a raw type. A raw type is a parameterized type which is used without any parameters, like simply List instead of List<?> or List<String>. Their main purpose is the ability to gradually migrate your code base from Java 1.4 to Java 1.5, something you should not have to worry about these days. However, as this syntax is allowed by Java compilers, people may use a raw type accidentally.

When a raw type is used, the compiler doesn't check anything about type parameter compatibility. It may issue warnings, but it will allow you to do anything that was allowed in Java 1.4, where no generic types existed. For example, the following code is perfectly compilable:

```
List list = new ArrayList<>(); // accidental raw type List
list.add(123);
printFirstItem(list);
```

There are compiler warnings about "unchecked calls" on the second and third lines, which means that the call is potentially unsafe and the compiler cannot guarantee the type safety.

Ways to avoid this problem

- Unchecked casts in Java can be occasionally useful, due to the imperfections of the Java type system. However, you must understand why you need an unchecked cast, why it's impossible to solve the problem in another way, and what consequences could result. It's better to encapsulate unchecked casts into well-tested utility methods or use existing methods for this purpose. Sometimes, adapter methods like `Collections.unmodifiableList()` may help to avoid unchecked casts.
- When declaring methods, consider using more generic types in method parameters involving wildcards like `? extends` (if you only need to read from the object) or `? super` (if you only need to write to the object). This approach is often called as "PECS", which stands for "producer – extends, consumer – super". It not only helps to avoid the unchecked casts but also documents your code better, as the user of your method will understand whether you are going to read or write.
- Avoid raw types as much as possible. When you use raw types, compiler checks become much weaker and you may easily screw the things up storing objects of incorrect type somewhere. In modern Java, there are only a few places where raw types could be useful, notable if you want to create an array of generic type. This is a bad idea anyway, it's better to use `ArrayList` instead.
- It's possible to ask the compiler to report raw types via the command line option `-Xlint:rawtypes`. You can use it to find raw types in your program and try to eliminate them.
- There are wrappers in the `Collections` utility class, like `checkedList()`, and `checkedMap()`, that allow you to actually control the type of elements added to the collection or map. Consider using them if you really care about the type safety of your collections. In this case, even an unchecked cast will not allow you to store an object of invalid type. For example, we can wrap the `ArrayList` from the `printFirstItem` sample with the `checkedList()`. In this case, an attempt to add a number to a list of string will cause an exception:

```
List<String> list = Collections.checkedList(
                      new ArrayList<>(), String.class);
List<?> objects = list;
((List<Object>) objects).add(1); // ClassCastException!
printFirstItem(list);
```

While using these wrappers everywhere is probably too pedantic, they are useful if you want to investigate where exactly an object of incorrect type appeared inside your collection. Just wrap the collection temporarily, run your program again, and you will see an exception at the element addition, rather than at the retrieval.

5.3.3 Different class loaders

The Java Virtual machine distinguishes classes by name and class loader. Usually, if you see a `ClassCastException`, this means that the class has a different name. However, it's also possible that the class has the same name but a different class loader (Figure 5.5).

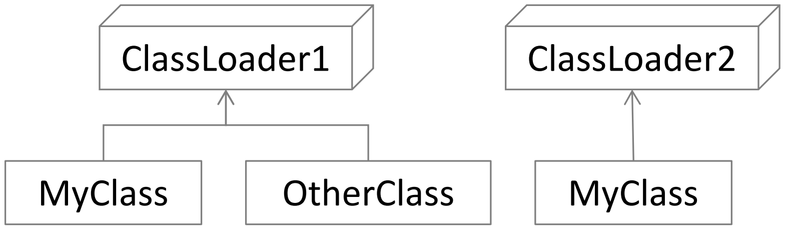


Figure 5.5 Classes with the same name can be loaded by different class loaders.

This doesn’t happen in simple Java programs, as such programs usually use the same class loader for all the classes. However, this may easily happen in more complex ones that have modular architecture or plugin support. In this case, it’s possible that the classes from different plugins or modules are loaded using different class loaders.

To illustrate what the possible problem looks like, let’s create a custom class loader. The easiest way is to use `URLClassLoader` and instruct it to load classes from the same location where actual classes of your application are located. This can be done in the following manner:

```
import java.net.*;

public class MyClass {
  public static void main(String[] args) throws Exception {
    URL url = MyClass.class.getResource(".");
    ClassLoader parent = System.class.getClassLoader();
    try (URLClassLoader newLoader =
           new URLClassLoader(new URL[]{url}, parent)) {
      Class<?> anotherMyClass = newLoader.loadClass("MyClass");
      Object o = anotherMyClass.getConstructor().newInstance();
      System.out.println("Class: " + o.getClass());
      System.out.println("Loader: " + o.getClass().getClassLoader());
      System.out.println("Instance of: " + (o instanceof MyClass));
      MyClass cls = (MyClass) o;
    }
  }
}
```

Note that there’s no package statement, and we load the class with the same name “MyClass” from the default package. This program produces the output like this:

```
Class: class MyClass
Loader: java.net.URLClassLoader@12edcd21
Instance of: false
Exception in thread "main" java.lang.ClassCastException: class MyClass cannot be cast to
       class MyClass (MyClass is in unnamed module of loader java.net.URLClassLoader
       @12edcd21; MyClass is in unnamed module of loader 'app')
       at MyClass.main(MyClass.java:13)
```

As you can see, we successfully loaded a class `MyClass` one more time, but it’s a different class with the same name. The `instanceof` check returns false, and the cast fails. It might look confusing to see the message like "class MyClass cannot be cast to class MyClass" but the additional explanation in the parentheses make it clear that the class loaders differ.

Ways to avoid this mistake

- Be careful when defining your own class loaders like `URLClassLoader` when you want to load some classes dynamically. In particular, pay attention to the parent class loader used. To make your class functional, it's necessary to load other classes as well, like superclasses and interfaces, and all of them will be loaded via your class loader. However, it always tries its parent class loader first, so all classes that can be loaded by the parent will return the parent. In the sample above, we used `System.class.getClassLoader()` as a parent class loader, so it could load standard library classes but could not load your application classes. Replace it with `MyClass.class.getClassLoader()`, and you'll be able to load existing classes of your application as well.
- If your application has a pluggable architecture, pay attention to plugin or module boundaries and be careful when passing objects between plugins. Even if two plugins depend on the same library, it's possible that the library classes are loaded by separate class loaders, to allow plugins to use different versions of the library for example. In this case, if you pass a library object to another plugin, you won't be able to cast it to the same type at destination.
- If you actually need to work with an object loaded by another class loader and have no clean options to resolve this, you can use the reflection. Just store that object as an Object type and call the methods reflectively, using API methods like `findMethod()` `invoke()`:

```
obj.getClass().getMethod("myMethod").invoke(obj);
```

Don't forget that such code is very fragile, as you don't have compile-time checks that the called method actually exists and has compatible parameters. If you use this approach, encapsulate all the reflection calls into separate methods, explain in the comments why it was done this way, and write unit tests to ensure that this approach actually works.

5.4 Mistake #44. StackOverflowError

Recursion is a useful programming technique that allows implementing many algorithms in cleaner way, compared to loops. This comes at a cost though. The recursive calls eat the stack of the thread and if the depth of the recursion is too big, you may get a `StackOverflowError` exception instead of the result of the computation. On popular JVM implementations, the maximal depth is not a constant: it depends on many things like whether the executed method is interpreted or JIT-compiled, and which tier of JIT-compilation was used. Also, it depends on how many parameters and local variables the recursive calls have. The same code may work correctly once but produce a `StackOverflowError` the next time, just because JIT was not fast enough to finish the compilation of the recursive method.

5.4.1 Deep but finite recursion

Some programming languages, especially functional programming languages, can automatically eliminate so-called tail recursion during compilation when your recursive call is the last action in the method. For example, JVM languages like Scala and Kotlin support this when compiling into JVM bytecode. In Kotlin, you should use `tailrec` function modifier[1]. In Scala, optimization is done automatically when possible, but you can assert it via `@tailrec` annotation[2], so compilation will fail if the recursion cannot be optimized. However, there's nothing like this in the Java compiler. Some virtual machines like IBM OpenJ9 may perform this optimization during JIT-compilation but it's not guaranteed.

On the other hand, tail recursion can always be mechanically rewritten to a loop right in the source code. IntelliJ IDEA provides a way to do this. There's an action "Replace tail recursion with iteration" which is available in the context actions menu (usually activated via Alt+Enter) when staying on the tail call. While the resulting code might not be so pretty, it will not suffer from possible stack overflow. For example, here's simple tail recursive implementation of an `indexOf` function on a string:

```
/**
 * @param s string to search in
 * @param start start offset
 * @param c symbol to search
 * @return position of symbol c inside string s
 * starting from start offset; -1 if not found
 */
static int indexOf(String s, int start, char c) {
  if (start >= s.length()) return -1;
  if (s.charAt(start) == c) return start;
  return indexOf(s, start + 1, c);
}
```

Running this method on very long strings containing for example 10,000 characters may produce a `StackOverflowError`. Applying the "Replace tail recursion with iteration" action will convert it into a loop:

```
static int indexOf(String s, int start, char c) {
  while (true) {
    if (start >= s.length()) return -1;
    if (s.charAt(start) == c) return start;
    start = start + 1;
  }
}
```

This implementation may not look so nice, but now it can process strings of arbitrary size. We recommend rewriting tail recursion with loops, either automatically or manually, unless you are completely sure that the maximal depth of your recursive calls does not exceed several hundreds.

Unfortunately, not every recursive call is a tail call, so it cannot always be optimized. Still, it's always possible to rewrite it using a loop. However, to store the state of intermediate

[1] https://kotlinlang.org/docs/functions.html#tail-recursive-functions
[2] https://www.scala-lang.org/api/2.13.x/scala/annotation/tailrec.html

calls, you'll need to use a side collection in the program heap. Usually, `ArrayDeque` collection fits well to emulate the stack calls.

As an example, let's consider a simple recursive tree traversal search algorithm that cannot be expressed via tail recursion. Let's assume that we have tree nodes that contain names and children (figure 5.6) and a method to find a node by name among the descendants, returning null if not found:

```
record Node(String name, Node... children) {
  Node find(String name) {
    if (name().equals(name)) return this;
    for (Node child : children()) {
      Node node = child.find(name);
      if (node != null) return node;
    }
    return null;
  }
}
```

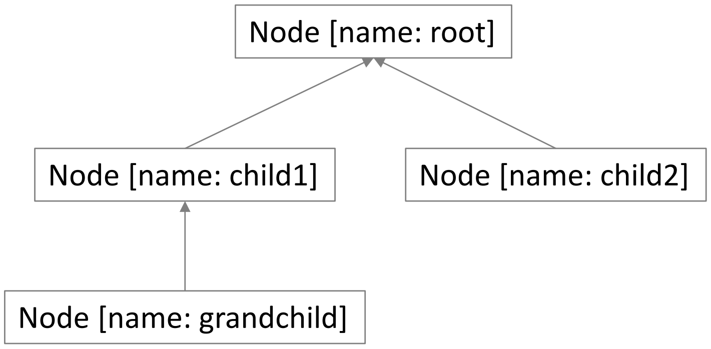


Figure 5.6 Simple tree of named nodes.

This method is simple and clear, and, in many cases, it would be ok to use it. For example, the following code creates a tree like the one shown in figure 5.6 and successfully finds the grandchild code:

```
Node root = new Node("root",
        new Node("child1", new Node("grandchild")),
        new Node("child2"));
System.out.println(root.find("grandchild"));
```

However, if you expect to have very deep trees, it would be better to get rid of recursion. All you need is to maintain a list of nodes to visit in a collection.

```
Node find(String name) {
  Deque<Node> stack = new ArrayDeque<>();
  stack.add(this);
  while (!stack.isEmpty()) {
    Node node = stack.poll();
    if (node.name().equals(name)) return node;
    Collections.addAll(stack, node.children());
  }
  return null;
}
```

The method becomes a little bit longer and probably harder to understand, but now you are limited by heap size instead of the stack size, which allows you to traverse much deeper trees.

Sometimes, the state to maintain between calls may consist of several variables, so you'll need to declare an additional data structure to maintain the state. Java records could be handy in this case. And unfortunately, it's not uncommon for recursive calls to contain many intermediate frames. For example, method A may call method B, which calls method C, which calls method A again. In this case, it could be quite non-trivial to refactor the code and avoid the recursion.

Ways to avoid this mistake:

- When making recursive calls, always consider whether it's possible that the stack capacity will be exceeded. Normally, the virtual machine can handle something between 1000 and 10,000 recursive calls, depending on many things like whether how many parameters and local variables your methods have, whether they were interpreted or JIT-compiled, and so on. So if you expect that the depth of calls will approach a thousand, it's better to play safe and use heap instead of stack space.
- Avoid implementing standard algorithms manually. In many cases, recursion is used to implement one of a few common algorithms, like graph search or tree traversal. Use well tested library methods instead, which avoid recursion already.

5.4.2 Infinite recursion

In the previous example, recursive calls are intended, and recursion is finite, but its depth exceeds the virtual machine stack. It can happen, though, that the recursion is truly infinite: regardless of the provided stack capacity, it will run out eventually. The difference is illustrated on figure 5.7. Finite recursion changes the program state, and eventually reaches the exit condition. On the other hand, infinite recursion does not change the program state, except consuming more space in the stack. It's also possible that the state is changed but the recursion termination condition is never satisfied. In this case rewriting the recursion with a loop will not help, as the loop will be infinite as well. It simply means that the program has a bug that should be fixed.

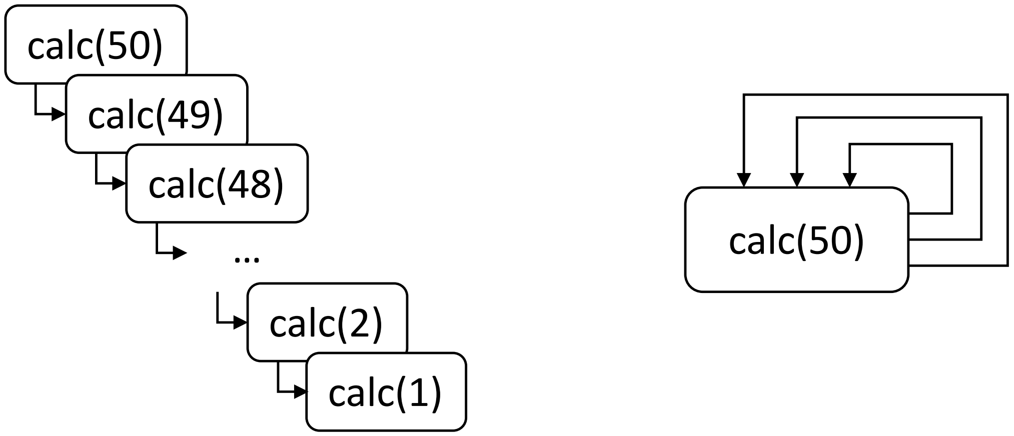


Figure 5.7 Finite recursion (left) and infinite recursion (right).

A simple case of accidental infinite recursion I have observed many times is when it was intended to delegate the call. For example, let’s assume that we have a UI button interface that handles the ‘pressed’ event:

```
interface Button {
  void onPressed();
}
Now, we have a wrapper that can disable the button:
class ButtonWrapper implements Button {
    private final Button delegate;
    private boolean enabled;

    ButtonWrapper(Button delegate) {
        this.delegate = delegate;
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    @Override
    public void onPressed() {
        if (!enabled) return;
        onPressed(); // delegate.onPressed() was intended
    }
}
```

Here, it was intended to call the `onPressed()` method on delegate, but the qualifier was mistakenly omitted. As a result, when the button is not disabled, we will have an infinite recursion.

In general, infinite recursion occurs when you call the same method with the same parameters, and the program state (values of variables and fields that are relevant to this part of the program) is also the same. So, when you make a call, the program returns to the original state but more stack space has been used.

One example of accidental state equality is recursive traversal of tree-like data structures. It's very simple to traverse a tree or a directed acyclic graph. All you need is to process the current node, then call yourself recursively for all the child nodes. However, if your data structure happens to contain a cycle, then stack overflow can easily happen. There are two possible cases: either having a cycle is legal but you weren't aware of this during traversal, or the cycle is introduced due to the earlier mistake in the program. In the former case, you should track already visited nodes to avoid visiting them twice. In the latter case, fixing the traversal is unnecessary. There's nothing wrong in recursive tree traversal, if we know in advance that cycles should be impossible. Instead, the earlier mistake should be found and fixed.

A simple example of such a recursive traversal can be found in standard library implementations of the list `hashCode()` method. If the list contains itself, its hash code cannot be calculated by specification. For example, the following code throws a `StackOverflowError`:

```
List<Object> list = new ArrayList<>();
list.add(list);
System.out.println(list.hashCode());
```

This is not a bug in the `hashCode()` implementation, as such a behavior is specified. In general, it's a bad idea to add a collection to itself.

Standard implementation of the `toString()` method has a simple protection against this, so the following code completes successfully:

```
List<Object> list = new ArrayList<>();
list.add(list);
System.out.println(list);
// [(this Collection)] is printed
```

However, the protection covers only the shortest possible cycle, when a collection is added directly to itself. If you create a longer cycle, `StackOverflowError` will still happen. E. g., the following code fails:

```
List<Object> list = new ArrayList<>();
list.add(List.of(list));
System.out.println(list);
```

A classic example of a data structure a program might traverse is a file system. It's a common task to traverse a subtree of directories and files and perform some action on them. Don't forget that file systems may contain cycles as well. If you follow symbolic links during traversal, it's very easy to create such a link that points to a parent directory. Even if not, some operating systems, like Linux, allow mounting file systems recursively, one into another (using the `mount --rbind` command). If you happen to traverse a directory that contains such a mount point, you may end up with stack overflow. Sometimes, it's acceptable to ignore this problem, especially if you traverse the directory that was previously created by your program automatically and you can assume that nobody has interfered with the content of that directory. However, if you traverse any custom user-specified directory, it’s probably a good idea to protect yourself from possible stack overflow.

As a Java IDE developer, I stepped on this problem when traversing a superclass hierarchy in Java programs during code analysis. In correct Java programs, a superclass cannot extend its own subclass, so if you follow the chain of superclasses, you’ll eventually come to `java.lang.Object`. However, it’s not guaranteed that a program opened in Java IDE is a correct Java program. It may contain compilation errors, but the analysis will still be executed. A naïve analyzer that doesn’t track already visited classes may easily end up with stack overflow on the following code:

```
class A extends B {}
class B extends A {}
```

Static analysis

Simple cases of infinite recursion can be detected by static analyzers. Pay attention to the corresponding warnings (e. g., the “Infinite recursion” inspection in IntelliJ IDEA, or the “InfiniteRecursion” bug pattern in Error Prone). They report rarely, but if they do report it’s almost always a real bug.

Ways to avoid this mistake:

- When traversing a tree-like data structure, think carefully whether it may contain cycles. Review how the data structure is created, check its specification. Probably cycle is a rare corner case, but if it’s possible, it’s better to avoid it.
- Reuse library methods to traverse the graph, don’t invent them by yourself. It’s often tempting to write a simple recursive procedure in-place. However, if you discover later that cycles are possible, it might be harder to fix it. Ready library methods may take care about tracking the visited elements, so you will have automatic protection against stack overflow.

5.5 Summary

- `NullPointerException` is a big problem which every Java programmer stumbles on. There are different techniques to minimize null-related mistakes. Some developers tend to avoid nulls completely, others control nullity using annotations.
- Use defensive checks at your API boundaries (public methods) to ensure that inputs are not nulls and indexes are within the expected range before saving them to fields. This will allow catching problems earlier.
- Casting a variable to another type without preceding `instanceof` check is very fragile, as you are losing the benefits provided by the static type system. If you are sure that the real variable type is also more concrete than the declared type, then change the declared type and remove the cast.

- Unchecked casts of generic type, will not cause `ClassCastException` immediately, because such a cast does not exist to the virtual machine due to type erasure. However, it may cause `ClassCastException` much later, as the compiler may insert implicit casts at the places where you use the generic type. This makes your code much harder to debug. Avoid unchecked casts when possible. Wildcard types or read-only and write-only wrappers may help you.
- Be careful when your application uses several class loaders, as classes loaded by different class loaders are different classes even though they have the same name.
- Try to avoid arbitrarily deep recursion in Java programs, as the Java stack is quite limited. Rewrite tail recursion with loops and use heap-based data-structures like `ArrayDeque` to handle non-tail recursion.
- When traversing data structures, pay attention to whether cycles are possible. If yes, it's better to memorize already visited elements to avoid infinite recursion.