


Computer science → Programming languages → Java → Errorless code → Debugging

# Debugging techniques

Theory

Practice

 100% completed, 0 problems solved ▼

## Theory

🕒 7 minutes reading

Unskip this topic

Start practicing

Debugging is the process of finding and fixing bugs in a program. Some bugs, like those that prevent the program from compiling, can be fixed easily since the compiler or an IDE can tell you what's wrong. Other bugs are trickier and may require you to put a lot of effort into detecting them.

In this topic, we will consider the most popular ways that programmers use to debug a program:

- Logging
- Assertions
- Attaching a debugger

## §1. Logging/'printf' debugging

One way to track the changes in the program state is to insert additional print statements in the code. When executed, they will inform you about what's happening under the hood at runtime.

For example, you can insert a line just before a method returns that will print the return value to the console. This way, in addition to seeing the final result, you will also be able to understand what happens at a certain stage of data processing.







Let's look at the following code snippet, which hangs indefinitely when run:

```
1 class UnexpectedResults {
2     public static void main(String[] args) {
3         count(1, 10);
4     }
5
6     public static void count(int start, int to) {
7         while (start < to); {
8             System.out.println(start);
9             start++;
10        }
11    }
12 }
```

While the error is not hard to catch, we can still add some print statements that would clearly indicate where the hanging happens:

```
1 class UnexpectedResults {
2     public static void main(String[] args) {
3         System.out.println("main() started");
4         count(1, 10);
5         System.out.println("main() complete");
6     }
7
8     public static void count(int start, int to) {
9         System.out.println("count() started");
```

### 3 required topics

-   [Hierarchy of exceptions](#) ▼
-   [What is a bug](#) ▼
-   [Introduction to logging](#) ▼

### 3 dependent topics

- [Debugging simple constructs](#) ▼
- [Debugging](#) ▼
- [Logging with Logback](#) ▼

### Table of contents:

- [1 Debugging techniques](#)
- [§1. Logging/'printf' debugging](#)
- [§2. Assertions](#)
- [§3. Attaching a debugger](#)
- [§4. Conclusion](#)
- [Discussion](#)

```
1
0     while (start < to); {
1
1         System.out.println(start);
1
2         start++;
1
3     }
1
4     System.out.println("count() complete");
1
5 }
1
6 }
```

Now, instead of just hanging, the program will output:

```
1  main() started
2  count() started
```

This output shows us that the program reaches the start of the `count()` method, but never reaches its end, which means that the problem is in the `while` loop. After we take a closer look at the construct, we see that there is an extra semicolon.

Inserting print statements is the most basic way to debug your code, however, we provide it just so you know this technique. You should not use this method in real projects because modern debuggers can do the same in a much more convenient way and because you would not be able to do that everywhere. For example, if you want to get information from some library code, this would be a problem because you cannot modify compiled code.

Be patient, we will cover the nice way shortly.

## §2. Assertions

In order to detect bugs in the program at earlier development stages, you can use assertions. The assertion is a mechanism that monitors the program state, but unlike additional print statements, it terminates the program in a fail-fast manner when things go wrong.

Fail-fast is an approach when errors that could otherwise be non-fatal are forced to cause an immediate failure, thus making them visible.

You may wonder why one would want to crash the production code, and the answer is: one wouldn't. Assertions are meant for testing/debugging and should never be used in production code.

Let's take a look at the following program:

```
1  class BrokenInvariants {
2      public static void main(String[] args) {
3          Cat casper = new Cat("Casper", -1);
4      }
5  }
6
7  class Cat {
8      String name;
9      int age;
10
11     public Cat(String name, int age) {
12
13         this.name = name;
14
15         this.age = age;
16     }
17 }
```

```
1  
3     }  
1  
4 }
```

This code creates a `Cat` object. This would be fine if it wasn't for the negative age value that makes no sense. Naturally, in a more complex program, this may lead to various bugs. Such an object may be passed around for a long time before we see a problem, and when a problem arises, it is not always obvious what was the cause.

To prevent that from happening, we can use an assertion right in the `Cat` constructor:

```
1 public Cat(String name, int age) {  
2     assert (age >= 0) : "Invalid age";  
3     this.name = name;  
4     this.age = age;  
5 }
```

The part before the colon specifies the boolean expression that should be checked, and when it evaluates to `false`, an error is thrown. The part after it specifies the message that describes the error.

Now, if we run the code with the `-ea` flag ( `java -ea BrokenInvariants` ), the program will throw an error and terminate right in the `Cat` constructor:

```
1 Exception in thread "main" java.lang.AssertionError: Invalid age  
2 at Cat.<init>(scratch_1.java:11)  
3 at BrokenInvariants.main(scratch_1.java:3)
```

You may have noticed that we used the word *invariants* and are curious what it means. **Invariants** are constraints that must be met for a program to function properly. In the code above, positive `age` is an example of an invariant. Using a negative `age` is asking for a problem. Enforcing invariants is exactly why we need assertions.

We can also use assertions to check method preconditions and postconditions, that is conditions that must be met before or after a method is invoked.

The only and very important limitation that should be observed when using assertions is that they should never produce side effects and change the way a program operates. In other words, the assertion should not affect a program in any way other than throwing an error.

Below is an example of an assertion that violates this rule:

```
1 assert (age++ >= 0) : "Invalid age";  
2 this.name = name;  
3 this.age = age;
```

Besides checking the condition, the assertion also increments the `age` value, which means that when assertions are disabled, the program will operate differently, invalidating the results of testing and probably introducing new bugs. Clearly, you want to avoid such situations in production.

### §3. Attaching a debugger

A debugger is a tool that interferes with the normal program execution allowing you to get runtime information and test different scenarios to diagnose bugs. This is the most popular use of debuggers. However, when you grow more experienced with them, you'll see that they can be helpful in various situations, not necessarily related to bugs.

Modern debuggers provide a vast variety of tools that can be used to diagnose the most intricate failure conditions, so they definitely warrant a section of their own. In the next topics, we will get started with IntelliJ IDEA debugger and learn how to debug simple code.

### §4. Conclusion

In this topic, you have learned about different debugging techniques you can use to ensure that your code is error-free. The most basic method is inserting print statements to keep track of the values and the execution order of your program. Also, you may add assertions using the `assert` keyword to make potential hidden errors visible at an early stage of development. Such assertions may be enabled and disabled by adding or removing the `-ea` JVM flag. Finally, you may attach a debugger to the program to examine its internal state at runtime. These techniques will help you to detect and fix bugs in your code efficiently.

 Report a typo

141 users liked this piece of theory. 7 didn't like it. **What about you?**



Start practicing

Unskip this topic

[Comments \(9\)](#)

[Useful links \(4\)](#)

[Show discussion](#)



Tracks

Pricing

For organizations

About

Contribute

Careers

 Become beta tester

Be the first to see what's new

Terms   Support    

Made with  by Hyperskill and JetBrains