

Debugging in Java

Contents

Introduction	3
How to find compile time errors	4
Compile the code	4
other examples	7
IDE's Language Server Protocol	13
How to find runtime and logical errors	19
print statements	19
Unit testing or test cases	23
Debuggers	25
Basics of debuggers	25
Advanced debugging	38

Conclusion	39
Documentation	42

Introduction

We all write perfect code on first attempt, right?

Me: *Writes some code*

Software:



Figure 1: perfect code

In this tutorial, we will learn how to understand the errors in our code and how to fix them.

How to find compile time errors

Compile the code

The Java compiler has a great way of informing you of problems in your code. It will tell you the line number and the type of problem. It will also tell you if there are multiple problems on the same line.

Example:

```
class punchcard {  
    public static void main(String[] args) {  
        System.out.println("Hello World!")  
    }  
}
```

Output:

```
/home/suren/Templates/Java/test/test.java:3: error: ';' expected  
    System.out.println("Hello World!")  
                                ^  
1 error  
error: compilation failed  
  
shell returned 1
```

- Let's go over the output and understand what it means.

```
/home/suren/Templates/Java/test/test.java
```

This is the path to the file that has the error.

This helps you identify the file that has the error.

Look at the end of it to see the file name.

The file name is **test.java**.

- Let's continue with the output.

```
:3: error:
```

This part of the output tells you the line number of error.

You can see the number of line in the left of side of IDE's editor.

So you can go to the line number and see what the problem is.

The line number is **3**.

- Let's continue with the output.

```
'; ' expected
```

This part of the output tells you what the problem is.

In this case, it is telling you that you are missing a **semicolon**.

There are many other errors that you can get. Some of These errors are easily understandable and some of them are cryptic. :b

In a case you can't understand the error, you can search it on the **internet**. Remember, as a programmer, The internet is your best friend.

- Let's continue with the output.

```
System.out.println("Hello World!")  
-----^
```

This is an amazing feature of the Java compiler. It tells you the exact location of the error.

So no more wondering where the error is in a long line of code.

- Let's continue with the output.

```
1 error
```

This part of the output tells you the number of errors in your code.

Yes Java compiler can tell you if there are multiple errors in your code.

In this case, there is only one error.

- Let's continue with the output.

```
error: compilation failed
```

This part of the output tells you the type of error.

In this case, it is telling you that the compilation failed.

- Let's continue with the output.

```
shell returned 1
```

This line is not part of the Java compiler output.

This line is the output of the shell.

But it is still useful to know that the shell returned 1.

A return value of 0 means that the program ran successfully.

A return value of 1 means that the program failed to run.

Here value of 1 means that the program failed to compile.

other examples

Example:

```
class punchcard {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Output:

```
/home/suren/Templates/Java/test/test.java:5: error: reached end of file while parsing
}
^
1 error
error: compilation failed
```

file name: test.java

line number: 5

error: reached end of file while parsing

number of errors: 1

type of error: compilation failed

missing: }

Example:

```
class punchcard {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Output:

```
/home/suren/Templates/Java/test/test.java:3: error: ')' or ',' expected
    System.out.println("Hello World!");
                                         ^
1 error
error: compilation failed
```

file name: test.java

line number: 3

error: ')' or ',' expected

number of errors: 1

type of error: compilation failed

missing:)

Example:

```
class punchcard {
    public static void main(String[] args) {
        System.out.println("Hello World!");
        not_read_code_sitting_here;
    }
}
```

Output:

```
/home/suren/Templates/Java/test/test.java:4: error: not a statement  
        not_read_code_sitting_here;  
               ^  
1 error  
error: compilation failed
```

file name: **test.java**

line number: **4**

error: **not a statement**

number of errors: **1**

type of error: **compilation failed**

this code is not a valid statement.

Example:

```
class punchcard {  
    public static void main(String[] args) {  
        System.out.println("Hello World!")  
        System.out.println("Hello Second World!");  
    }  
}
```

Output:

```
/home/suren/Templates/Java/test/test.java:3: error: ';' expected
    System.out.println("Hello World!")
                                         ^
/home/suren/Templates/Java/test/test.java:4: error: unclosed string literal
    System.out.println("Hello Second World!");
                                         ^
2 errors
error: compilation failed
```

file name: test.java

line number: 3 and 4

error: ';' expected and unclosed string literal

number of errors: 2

type of error: compilation failed

missing: ; and "

He couldn't sleep for 2 days
because he missed her.

I couldn't sleep for 4 days
because I missed a stupid " ; " in
my code.

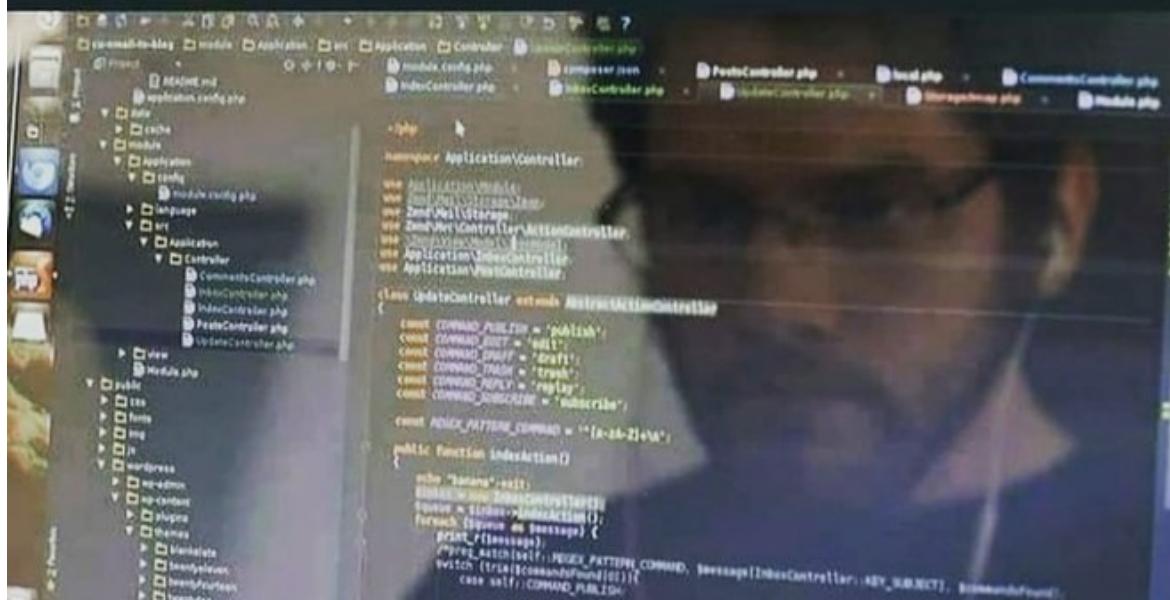


Figure 2: errors

IDE's Language Server Protocol

Most popular IDE's support the **Language Server Protocol**.

This feature allows you to see the errors in your code as you type.

With this feature, you don't have to compile your code to see the errors.

An LSP has many indicators to show you the errors in your code.

- Explorer Panel

Look at the explorer panel on the left side. There is a number next to the file name.

This number shows you the number of errors in the file.

- Mini Map

Look at the mini map on the top right side of Figure 3. There is a red line on the mini map.

This red line shows you the general location of the error within the file.

This can help to find the errors in a long file.

- Error Panel

In the bottom of the IDE, there is a panel. This panel has many useful tabs. Up until now, we have only used the **TERMINAL** tab.

Now, let's look at the **PROBLEMS** tab.

The screenshot shows a Java development environment with the following components:

- EXPLORER** sidebar: Lists files in the project, including `.vscode`, `digit.displayer.java`, `digit.displldyer.2.java`, `donut.java` (marked with a red error icon), `test.java` (selected and marked with a blue icon), and `test2.java`.
- CODE EDITOR**: The active file is `test.java`, containing the following Java code:

```
1 class punchcard {  
2     Run | Debug  
3         public static void main(String[] args) {  
4             System.out.println("Hello World!");  
5         }  
6 }
```
- PROBLEMS** panel: Shows one syntax error in `test.java`: "Syntax error, insert ";" to complete BlockStatements Java(1610612976) [Ln 3, Col 36]".
- NAVIGATION**: Includes links for `OUTLINE`, `TIMELINE`, `SONARLINT ISSUE LOCATIO...`, and `JAVA PROJECTS`.

Figure 3: basic IDE with LSP

This tab shows you the errors in your code in a list format.

You can click on the red cross-circle to go to the line that has the error.

You can copy the error message by right-clicking on the error that is in the list. This can be useful if you want to search the error on the internet.

At the end of the error message, there is a set of numbers.

[Ln 3, Col 36]

The ‘Ln’ tells you the line number of the error. (line 3) The ‘Col’ tells you the column number of the error. (column 36)

- Red underline

Look at the red underline in the Figure 3. Exactly under the ‘)’ in the line 3.

If you hover over the red underline, you will see the error message. Some IDE also show more information about the error. Look at the Figure 4.

There’s even a quick fix available for this error. The quick fix gives you a suggestion to fix the error. You might not need to get your hands dirty with the code.

The screenshot shows a Java code editor within an IDE interface. The code in the editor is:

```
1 class punchcard {  
2     public sta  
3         System.out.println("Hello World!);  
4     }  
5 }  
6
```

A tooltip is displayed over the closing parenthesis in the third line, indicating a LSP diagnostic:

String literal is not properly closed by a double-quote Java(1610612995)

Below the editor, the IDE's bottom navigation bar includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The PROBLEMS tab is selected, showing 6 issues. A specific issue is highlighted:

test.java 1 of 1 problem

String literal is not properly closed by a double-quote Java(1610612995)

Details for the first problem in the list:

PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL PORTS !*2.j... Showing 1 of 6 J test.java 1 1

String literal is not properly closed by a double-quote Java(1610612995) [Ln 3, Col 22]

The left sidebar contains the following sections:

- EXPLORER
- TEST
 - .vscode
 - digit.displayer.java
 - digit.displldyer.2.java
 - donut.java
- 2
- J test.java 1
- J test2.java 3
- > OUTLINE
- > TIMELINE
- > SONARLINT ISSUE LOCATIO...
- > JAVA PROJECTS

Figure 4: IDE with LSP and quick fix

With these tools we can easily find any runtime errors in code.

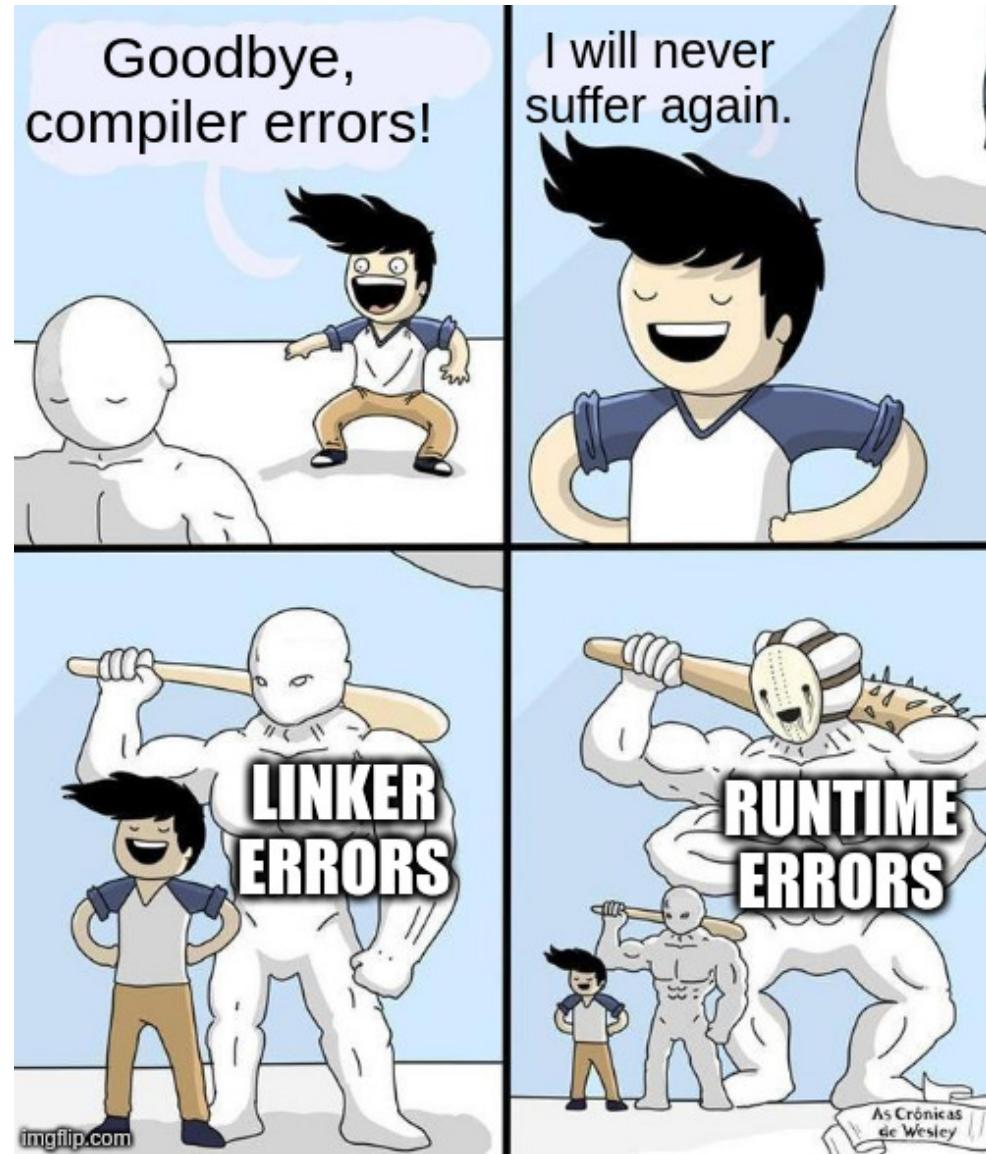


Figure 5: errors

How to find runtime and logical errors

print statements

As long as mankind learned the art of programming, they have used print statements to debug their code.

It's the least elegant way to debug your code, but it's the most effective way to debug small programs.

Look In the following code.

```
class punchcard {
    public static void main(String[] args) {
        int a = 5 + 7; // BUG
        int b = 10;
        int c = a + b;
        if (c == 15) {
            System.out.println("Success!");
        }
        else {
            System.out.println("Failure!");
        }
    }
}
```

Let's say we expected the value of **a** to be 5 and **b** to be 10. As a result, we expected the value of **c** to be 15. But for some reason, the value of **c** is not 15.

First thing we can do is to print the value of **c** to see what the value of **c** is.

```
class punchcard {  
    public static void main(String[] args) {  
        int a = 5 + 7; // BUG  
        int b = 10;  
        int c = a + b;  
        System.out.println(c); // BUG  
        if (c == 15) {  
            System.out.println("Success!");  
        }  
        else {  
            System.out.println("Failure!");  
        }  
    }  
}
```

Output:

```
22  
Failure!
```

Now we know that the value of **c** is 22. But we still don't know why the value of **c** is 22.

Let's print the value of **a** and **b** to see what the value of **a** and **b** are.

```
class punchcard {  
    public static void main(String[] args) {  
        int a = 5 + 7; // BUG  
        System.out.println(a); // BUG  
        int b = 10;  
        System.out.println(b); // BUG  
        int c = a + b;  
        System.out.println(c); // BUG  
        if (c == 15) {  
            System.out.println("Success!");  
        }  
        else {  
            System.out.println("Failure!");  
        }  
    }  
}
```

Output:

```
12  
10  
22  
Failure!
```

Gotcha! The value of **a** is 12. That's why the value of **c** is 22. We can fix the bug by changing the value of

a to 5.

Don't forget to remove the print statements after you fix the bug. Just to keep your code clean.

```
class punchcard {  
    public static void main(String[] args) {  
        int a = 5;  
        int b = 10;  
        int c = a + b;  
        if (c == 15) {  
            System.out.println("Success!");  
        }  
        else {  
            System.out.println("Failure!");  
        }  
    }  
}
```

Output:

Success!

Unit testing or test cases

What if the inputs are not as simple, or you might not know all the edge cases?

In this case, you can use unit testing to test your code. Unit testing is a way to test your code by writing code.

However, The topic of unit testing is beyond the scope of this tutorial. Besides, writing unit tests needs more knowledge of Java.

Unit test is not preferred for debugging small code.

Unit tests just inform the developer that an unexpected behavior has occurred. It gives no information about the cause of the problem.

But test cases are one of the crucial parts of big projects that are developed by multiple developers.

It ensures that the code is working as expected.

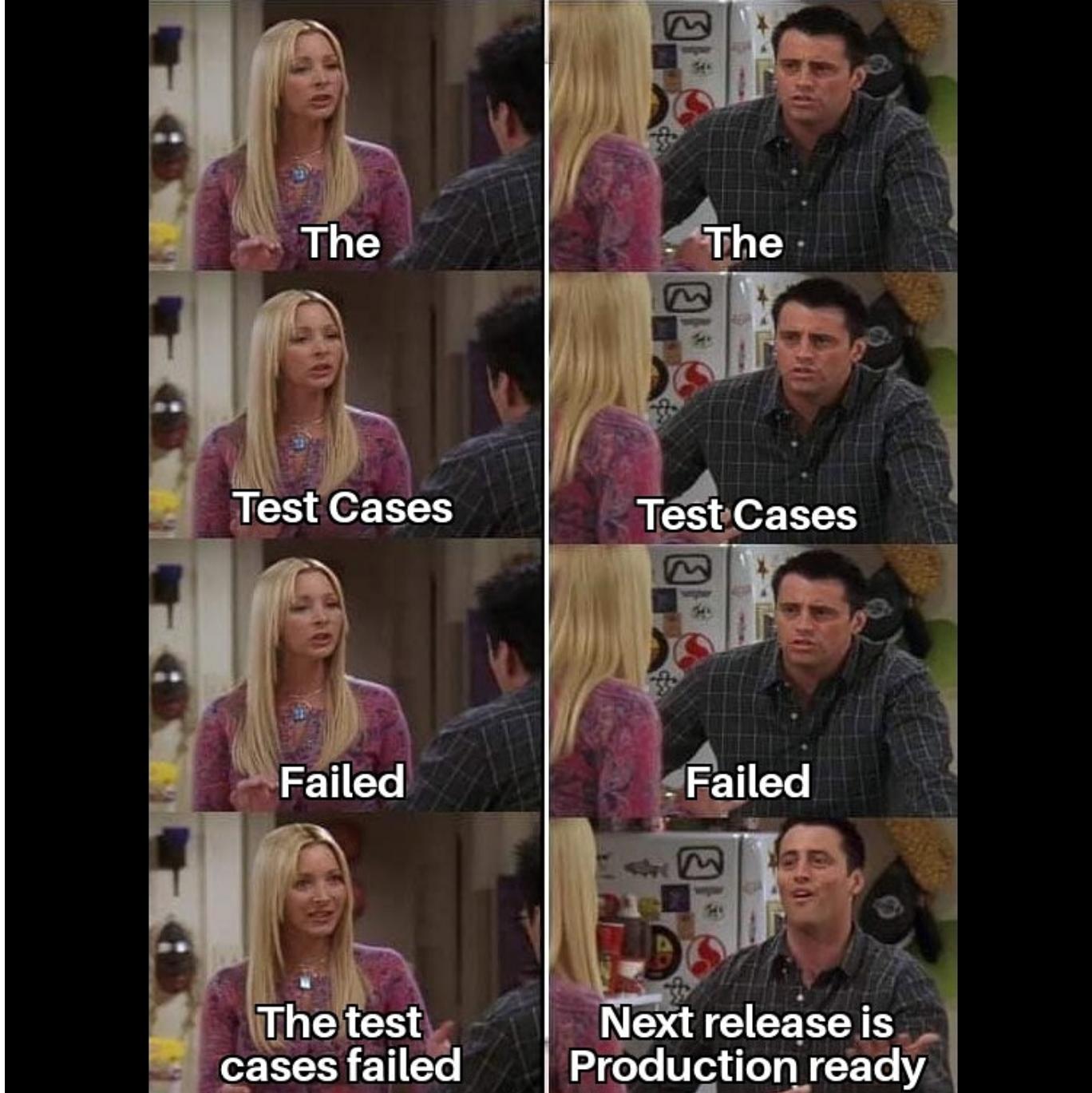


Figure 6: unit test
24

Debuggers

Here comes the most powerful tool that can save you hours of debugging.

Debuggers are tools that allow you to run your code line by line.

You can see the value of variables at each line of code.

Don't let Its name deceive you. Debuggers are not only used for debugging. Debuggers are great for:

1. **Understanding how an unfamiliar code works.** You can run the code line by line and see what each line of code does. Watching how values change.
2. **Finding the cause of a bug.** You can run the code line by line and see where the code is not working as expected. What value is not as expected.
3. **Testing your code.** You can run the code line by line and see if the code. There are times that a code works and outputs the correct result, but it's not the correct way to solve the problem. Debuggers can help you find these problems.

Basics of debuggers

First let's learn the bread and butter of debuggers. THE **breakpoints**.

- **Breakpoints**

Breakpoints are the points in your code that you want to stop the execution of the code.

You can set a breakpoint by clicking on the left side of the line number. Figure 6.

Look at this example.

The screenshot shows the VS Code interface with the following details:

- EXPLORER**: Shows a tree view with a node for **.vscode** and several Java files under **TEST**: **digit.displayer.java**, **digit.displldyer.2.java**, **donut.java** (count 2), **test.java** (count 4, currently selected), and **test2.java** (count 3).
- CODE EDITOR**: Displays the **test.java** file with the following code:

```
1 class punchcard {
2     public static void main(String[] args) {
3         int a = 5 + 7; // BUG
4         int b = 10;
5         int c = a + b;
6         if (c == 15) {
7             System.out.println("Success!");
8         } else {
9             System.out.println("Failure!");
10    }
11 }
```

A red circular icon with a white hand cursor is positioned next to the line number 5.
- TERMINAL**: Shows the output of running the **punchcard** class:

```
[suren@HaghDazhDefoov ~/Templates/Java/test]$ /usr/bin/env /usr/lib/jvm/java-21-openjdk/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/suren/.config/VS Codium/User/workspaceStorage/cf11f2490440d335edf9d9951da113b9/redhat.java/jdt_ws/test_c9889b72/bin punchcard
Picked up _JAVA_OPTIONS: -Djava.util.prefs.userRoot=/home/suren/.config/java
Failure!
[suren@HaghDazhDefoov ~/Templates/Java/test]$
```

Figure 7: breakpoint

Example:

```
class punchcard {  
    public static void main(String[] args) {  
        int a = 5 + 7; // BUG  
        int b = 10;  
        int c = a + b;  
        if (c == 15) {  
            System.out.println("Success!");  
        }  
        else {  
            System.out.println("Failure!");  
        }  
    }  
}
```

Let's say you want to see the value of **a** and **b** before the **if** statement. You can set a breakpoint on the line 5 and click on the **debug** button. Figure 8.

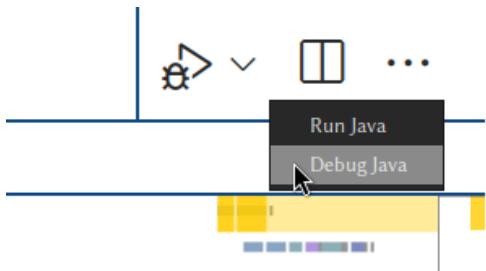


Figure 8: debug button

The screenshot shows a Java debugger interface with the following details:

- Top Bar:** Includes "RUN A...", "No Config", "..." (dropdown), and a toolbar with icons for Run, Stop, Step, Break, and Refresh.
- Code Editor:** Displays the file `test.java` at line 4, showing the code for a class named `punchcard`. The line `int c = a + b; a = 12, b = 10` is highlighted in yellow, indicating it is the current line of execution. A red circle with a dot is placed on the left margin next to line 5, marking a breakpoint.
- Variables View:** Shows the local variables `args`, `a`, and `b`. `args` is a String array of size 8. `a` is 12. `b` is 10.
- Watch View:** Empty.
- Call Stack:** Empty.
- Breakpoints View:** Shows three entries:
 - Uncaught Exceptions
 - Caught Exceptions
 - test.java
- Terminal:** Displays the command `t_ws/test_c9889b72/bin punchcard` and the output: "Picked up _JAVA_OPTIONS: -Djava.util.prefs.userRoot=/home/suren/.config/java".
- Bottom Bar:** Includes tabs for PROBLEMS (16), OUTPUT, DEBUG CONSOLE, TERMINAL (selected), and PORTS, along with a dropdown menu for Java Build Settings, Run configuration, and Debug configuration.

Figure 9: debugger

This is a debugging session. Figure 9.

There are many useful tools in the debugger. Let's go over them. One by one.

But first let's understand the flow of a java program.

As for now, all we need to know is within the `public static void main(String[] args)`, the code is executed line by line from top to bottom.

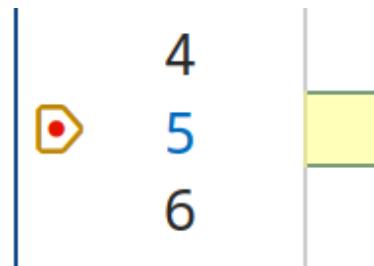


Figure 10: step indicator

Step Indicator This brown octagon-arrow shaped button is the **step indicator**. Figure 10.

This button shows the next line of code that will be executed. The code is paused on the line that is shown in the step indicator.

Now we can see all the information before the execution of `c = a + b;`.

Variables There are multiple ways to see the value of variables before stop of the flow. Figure 11.

- **Variables Panel**

✓ VARIABLES

✓ Local

```
args: String[0]@8  
a: 12  
b: 10
```

The screenshot shows a Java debugger interface. On the left, there's a 'VARIABLES' panel titled 'Local' with entries for 'args', 'a', and 'b'. The main area shows a Java file named 'test.java' with the following code:

```
J test.java > punchcard > main(String[])
1 chcard {
Run | Debug
2 public static void main(String[] args) { args = String[0]@8
3     int a = 5 + 7; // BUG a = 12
4     int b = 10; b = 10
5     int c = a + b; a = 12, b = 10
6     if (c == 15) {
```

A yellow box highlights the assignment 'a = 12'. A blue box highlights the assignment 'b = 10'. A red box highlights the assignment 'c = a + b; a = 12, b = 10'. A green box highlights the condition 'if (c == 15) {'. A small orange circle with a dot is positioned next to line 5.

Figure 11: variables

In left side, you can witness the variables panel. All the values before the stop of the flow are shown in this panel.

- Hover over

You can hover over the variable to see the value of the variable. Hover over the **a** in the line 3. Displays the value **12**.

- Inlay hints

The values for the variables are shown in the code, just after actual code. In a gray colored text

- Debug Console

You can call the value of a variable in the debug console. Figure 12.

There's no need to use the `System.out.println()` to see the value of a variable. Just write the variable name in the debug console prompt and press enter.

PROBLEMS

16

OUTPUT

DEBUG CONSOLE

TERMINAL

→ a

12

→ b

10

→ a+b - 7

15

> a+b

Figure 12: debug console

Debug Toolbar Let's add a breakpoint on line 10. We will make use of this second breakpoint later in this section.



Figure 13: debug toolbar

This is the debug toolbar. Figure 13.

From left to right:

- **Continue**

This button continues the execution of the code until the next breakpoint. Remember, the flow of execution is paused on the line that is indicated by the step indicator, So the `System.out.println("Failure!")` is not executed yet. Figure 14.

- **Step Over**

This button stops the flow of execution on the next line of code. No matter if there is a breakpoint on the next line or not.

So as an example you can put a breakpoint on line 3 and click on the **step over** button. The flow of execution will stop on line 4. Next button press will stop the flow of execution on line 5. Figure 15.

- **Step Into**

The screenshot shows the VS Code interface with the following details:

- Top Bar:** RUN A... (green play button), No Config, gear icon, three dots.
- Editor:** File test.java (4 lines). The code is:

```
1 class punchcard {  
2     public static void main(String[] args) { args = String  
3         int a = 5 + 7; // BUG a = 12  
4         int b = 10; b = 10  
5         int c = a + b; c = 22, a = 12, b = 10  
6         if (c == 15) { c = 22  
7             System.out.println("Success!");  
8         }  
9         else {  
10            System.out.println("Failure!");  
11        }  
12    }  
13 }
```
- Variables View:** Local variables: args: String[0]@8, a: 12, b: 10, c: 22.
- Watch View:** Watched variable: 10 (System.out.println("Failure!")).
- Breakpoints View:** Breakpoints for test.java: 5 (checked), 10 (checked).
- Bottom Bar:** PROBLEMS (16), OUTPUT, DEBUG CONSOLE (selected), TERMINAL, PORTS.
- Terminal:** Command-line output:

```
=y,address=localhost:44301 --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /home/suren/.config/VSCode/User/workspaceStorage/cf11f2490440d335edf9d9951da113b9/redhat.java/jdt_ws/test_c9889b72/bin punchcard  
Picked up _JAVA_OPTIONS: -Djava.util.prefs.userRoot=/home/suren/.config/java
```
- Right Sidebar:** Java Build St..., Run: punchcard, Debug: punchcard.

Figure 14: continue

J test.java 4 X

J test.java > punchcard > main(String[])

```
1 class punchcard {
    Run | Debug
  2     public static void main(String[] args) {
  3         int a = 5 + 7; // BUG a = 12
  4         int b = 10; b = 10
  5         int c = a + b; a = 12, b = 10
  6         if (c == 15) {
```

Figure 15: step over

This button action is similar to the **step over** button. The difference is out of the scope of this tutorial. (useful for debugging methods)

- **Step Out**

The action of this button is out of the scope of this tutorial. (useful for debugging methods)

- **Restart**

This button restarts the debugging session. It's useful if you want to debug the code multiple times.

Or you have changed the code, and you want to debug the code again.

- **Stop**

This button stops the debugging session. It also Disconnects the debugger, so you can go back to the normal mode of the IDE and run the code normally.

Breakpoints Normally, If you don't need a breakpoint anymore, you can remove it by clicking on the breakpoint, But there are times that you want to keep the breakpoint but you don't want it to stop the flow of execution. (for now)

In this case, you can disable the breakpoint. Figure 16.

✓ BREAKPOINTS



Uncaught Exceptions

Caught Exceptions

● test.java



Figure 16: disable breakpoint



Figure 17: Using Debugger?!

Advanced debugging

Debuggers have many more advanced features. But these features are out of the scope of this tutorial, However I will mention some of them here.

Conditional breakpoints Normally, the debugger stops where you asked for a breakpoint. If you are in a loop or have a clue what values trigger the problem, you don't want that. A conditional breakpoint allows you to add a bit of Java code to your breakpoint so it will only stop when that condition is true. This approach avoids having to hit resume a lot of times until you get to the value you care about.

This feature is especially useful for debugging loops.

Watchpoints Watchpoints are similar to breakpoints. But instead of stopping the flow of execution, they just print the value of the variable.

This is better than using print statements because you don't have to remove the watchpoint after you fix the bug.

Changing the value of variables You can Manually change the value of variables in the debugger and let the code continue the execution with the new value instead of the original value.

This helps you test potential fixes, without having to change the code.

Conclusion

Debugging is one of the most important skills of a programmer. It's a skill that you will use every day. It's a skill that you will use in every project.

Learn it, Use it, Master it.

And as always, Happy coding!

Don't be this guy!

Spending 4 hours fixing a bug and realizing
that the code you've been compiling was
not the code you've been fixing



Figure 18: The guy

Be this guy!

Programmers while reviewing the codes



Figure 19: The other guy

Documentation

Every IDE has its own debugger. So the documentation of the debugger is different for each IDE.

Head over to the documentation of your IDE to learn more about the debuggers.

- IntelliJ IDEA
- Eclipse
- NetBeans
- Visual Studio Code
- vim