CP1 Extras

By Peter Mitchell

This document can be found to read at: http://bit.ly/CP1Extras_S2

You can find the code examples on GitHub at: https://github.com/Squirrelbear/CP1Extras

This document is prepared by a tutor as additional material to help you with additional examples to get through the topic Computer Programming 1. It is not official material for the topic and are the work of Peter Mitchell independently and are my own thoughts. Make sure you follow the official material in the topic and do not rely entirely on this document.

This document covers briefly examples that may be of use to students for CP1 and will be added to weekly if there are new common questions, or a useful example I feel is relevant. Please let me know any error you find or possible additions you would like to see examples of.

Contents

1 Change List

Below are changes to this document so you can see new things at a glance. I will be updating the content with additional examples over the course of the semester. You can also request specific examples and I may prepare them to help out.

2021/07/30: Initial version of document set up ready for the semester.

2021/08/11: Added Notes for Practical 3 to 9.

2 The Practicals

This section covers a summary of what is required for each practical along with any import notes on what you need to consider when completing the practicals.

2.1 Due Dates

Checkpoints	1–5	6–10	11–15	16–20	21–25	26–30	31–35	36–40	41–45	46–50
Practical	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Due by the end of week	3	4	5	6	7	8	9	10	11	12

2.2 Practical Structure

Practicals take two different forms. The first practical has 5 checkpoints that are all defined in the Practical 1 PDF. For the rest of the practicals, you will complete what are called Drill Exercises to unlock the practical. You need to score at least 2.5 out of 4 in the Drill Exercises to reveal the files to start the rest of the practical. The Drill Exercise is counted as the first checkpoint for that (e.g., 6, 11, 16, 21, 26, 31, 36, 41, and 46). And then the remaining four checkpoints for each practical are in a PDF. Most have bonus tasks at the end that are not worth marks but are worth completing if you want to learn the most from this topic.

For some examples to help with the Drill questions look at the following sections. The first two are useful for the first drills you will experience when unlocking Practical 2. And the example with using methods will be relevant when you reach later weeks nearly halfway through the topic.

- 5.1 Example of Input/Output
- 5.2 Example of Drill Exercise
- 5.10 Writing Drills with Methods

2.3 Practical 1

The following describe what you should be working on for Practical 1. If you have additional questions, feel free to ask the tutor during your session.

2.3.1 Tasks

1. Install JDK16 and IntelliJ. You can find the links and instructions for downloading and installing these on FLO.
2. Download the Practical 1 start files in Assessment Hub -> Practical 1 Files
3. Extract the ZIP files to a location where you want to store all your practical files.
4. The PDF that is in the Practical 1 Files folder explains what is required for Checkpoint 1 to 5 and should be focus of your first week practical.
5. In addition to the instructions seen in the PDF you can utilise the TestPractical.java file provided with the practical to test your output. See the section specifically about this later on.

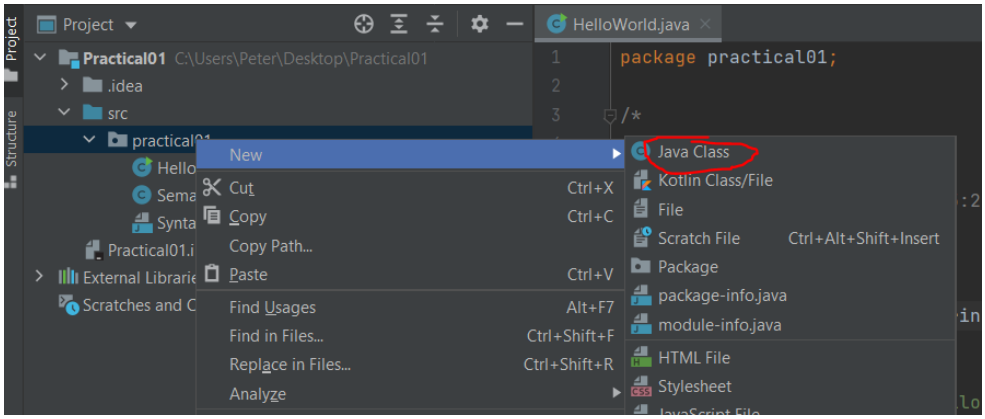
6. After you complete the tasks, you can upload a ZIP file containing your completed tasks. If you have issues with this or run into other problems some of the common issues are shown in the Troubleshooting Common Issues section of this document.

2.3.2 Important Notes

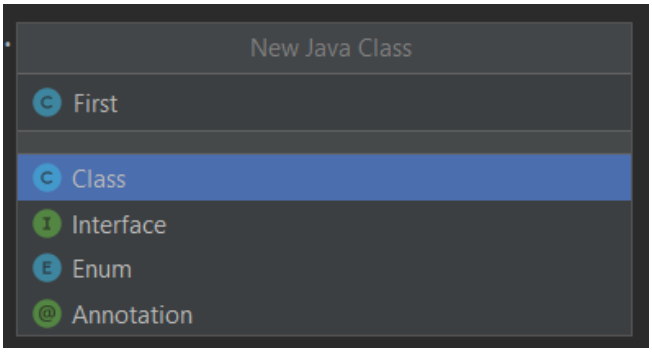
- You can see the expected output for each checkpoint in the TestCases folder. Make sure you check out the "Using TestPractical.java" tool included with the start files but looking at the output will give you an idea of what it should look like for each checkpoint.
- In Checkpoint 5 when writing First.java, you do need to write all four lines of text it shows in the question. The 42 and 8 numbers should not be written as 42 and 8 in Strings. You need to calculate these numbers using numbers such that you perform the 7 multiplied by 6 and that results in 42 printed out. In regard to making the quotation marks appear, you need to look up what escape sequences are.

2.3.3 Checkpoint 5 How to Create a New Java Class

To create a new Java Class for Checkpoint 5 you can simply right click on either the src folder or Practical01 folder under it. Then select New->Java Class.



This will open a small popup where you can type in the name of your new Class. In the task it asks you to call this "First". Then just hit Enter. This will create your file and you can add in a main method.



When completing this take make sure you include all four lines of text it tells you to print out and ensure you have maths calculating the 42 and 8 instead of just writing those in a String. The rest can be inside Strings.

2.4 Practical 2

2.4.1 Tasks

1. Complete the Drill Exercise to Unlock Practical 2 (this is Checkpoint 6).
    a. You will need at least 2.5 out of 4 marks to complete it. You can restart as many times as you wish, and it will give you new random questions each time. The first time you submit an attempt it will not deduct marks for the question. Every additional attempt that is not correct reduces the possible marks for that question by 10% (0.1).
    b. You can find very useful examples showing how to deal with input and output relevant to these in section 5.1 and 5.2 that show an example of how to read in an age input, modify the number and then output a result.
2. Once the Drill is complete, the Practical 2 folder will appear just under where the Drill was in the assessment hub.
    a. Extract the IntelliJ project and follow instructions in the PDF.
3. Once you are finished with the practical you can submit it to the submission box on FLO.

2.4.2 Important Notes

- For Checkpoint 10: There are a number of ways you can solve this problem. I would recommend you look at the code example in 5.3 showing an example of how you can separate numbers into individual digits and then you can reassemble those numbers to reverse the number.

2.5 Practical 3

2.5.1 Tasks

1. Complete the Drill Exercise to Unlock Practical 3 (this is Checkpoint 11).
2. Once the Drill is complete, the Practical 3 folder will appear just under where the Drill was in the assessment hub.
3. Complete the practical and submit to FLO.

2.5.2 Important Notes

- In Checkpoint 12 it is important to understand you can use the && or || to combine multiple conditions together inside an if statement.
- You can write if statements in the structure. Where you can have any number of else ifs and up to one else. All conditions are skipped if one is true.

```
if(condition) {

} else if(condition) {

} else {

}
```

- For Checkpoint 14 (the BMI task), you may want to look at the examples of Formatting with Math.round and DecimalFormat in section 5.5 of this document to understand how to show only one decimal place.
- For Checkpoint 15 to pass the checkpoint you MUST handle the special cases. When ANY of the four numbers are 0, that number should appear as "no" instead. You also need to correctly handle the plural cases. You can do this by breaking up the message into parts using print instead of println. And then using an if/else if/else structure to check for each variable being 0, 1, or 2+ to output the correct output.

  The test cases in TestPractical test a variety of cases, you should be able to pass all of these for full marks for the Checkpoint.

  You can find an example of breaking up a message using an if and else in the Boolean Example in section 5.4 of this document.

2.6 Practical 4

2.6.1 Tasks

1. Complete the Drill Exercise to Unlock Practical 4 (this is Checkpoint 16).
2. Once the Drill is complete, the Practical 4 folder will appear just under where the Drill was in the assessment hub.
3. Complete the practical and submit to FLO.
4. The first checkpoint does not require you to change much code. It is mostly about understanding what the code it gives you initially does.

2.6.2 Important Notes

- You do not need to create separate files for every checkpoint. It is additive and if you feel you want to keep parts as they change, you can comment out the code.
- If you are not sure of what a constructor, getter, or setter looks like you can look at an example of a class in this document in section 5.6 where a SaleItem class is shown.
- If you want to keep multiple checkpoints working at the same time you can either look at the recommendations for how you should write your BoatMaker in TestPractical, or the tips in my section of this document titled "How to Approach Checkpoints with Multiple Parts" in section 5.7.
- If you are stuck working out what to do when attempting the extension task the concepts you are looking for are linked lists. This extension task (not worth marks) is significantly harder compared to other checkpoints. You can find a related example showing the type of approach you can use on my GitHub in ExampleLinkedClass.java mimicking the Boat class and ExampleLinkedClassTest.java showing the approach you can use for the main method. https://github.com/Squirrelbear/CP1Extras

2.7 Practical 5

2.7.1 Tasks

1. Complete the Drill Exercise to Unlock Practical 5 (this is Checkpoint 21).
2. Once the Drill is complete, the Practical 5 folder will appear just under where the Drill was in the assessment hub.
3. Complete the practical and submit to FLO.

2.7.2 Important Notes

- For Checkpoint 25, you may want to look at sections in this document in sections 5.8 and 5.9 where you will find examples related to “Types of Loops and When to Use Them”, and a “Switch Statement Example”.
- Also, in relation to Checkpoint 25, it is NOT asking you to modify the original string. The question is simply asking you to print out the string one character at a time either printing out the modified character or if there is not a specified replacement you would print the original character.
- You will likely either this week, or next, encounter Drill exercises that are dealing with methods instead of just writing code directly with no method. You can find an example to help with this in section 5.10 where you will find an example of “Writing Drills with Methods”.

2.8 Practical 6

2.8.1 Tasks

1. Complete the Drill Exercise to Unlock Practical 6 (this is Checkpoint 26).
2. Once the Drill is complete, the Practical 6 folder will appear just under where the Drill was in the assessment hub.
3. Complete the practical and submit to FLO.

2.9 Practical 7

2.9.1 Tasks

1. Complete the Drill Exercise to Unlock Practical 7 (this is Checkpoint 31).
2. Once the Drill is complete, the Practical 7 folder will appear just under where the Drill was in the assessment hub.
3. Complete the practical and submit to FLO.

2.9.2 Important Notes

- You can find an example of various array related content in section 5.11 of this document titled “Arrays Example”.
- I would strongly recommend either separating your tasks for this practical into methods for each checkpoint or commenting out clearly the work for each one since you are making significant changes between the last two tasks.

2.10 Practical 8

2.10.1 Tasks

4. Complete the Drill Exercise to Unlock Practical 8 (this is Checkpoint 36).
5. Once the Drill is complete, the Practical 8 folder will appear just under where the Drill was in the assessment hub.
6. Complete the practical and submit to FLO.

2.11 Practical 9

2.11.1 Tasks

7. Complete the Drill Exercise to Unlock Practical 9 (this is Checkpoint 41).
8. Once the Drill is complete, the Practical 9 folder will appear just under where the Drill was in the assessment hub.
9. Complete the practical and submit to FLO.

2.12 Practical 10

Not available yet. TODO

3 Using the TestPractical.java Tool

The Practical Evaluator utility is a new feature that has been added to the practicals this semester developed recently by Peter Mitchell. The tool allows you to easily compare your programs output for each checkpoint against one or more test cases. You will find the required components with the start files for practical 1 to 9. The included parts for this are:

- PracticalEvaluator.java: This file does all the work. You can look in this if you are interested in learning how it works, but you can ignore this file if you wish.
- TestCases folder: This folder contains all the test cases with files labelled A_B.in and A_B.out. Where A is the checkpoint number, and B is the test case number. There will always be a .out file for each test case, but the .in file only exists if there is input for that checkpoint.
- TestPractical.java: This file is the one you mostly need to care about for each practical. Each practical has a slightly modified file with code already there commented out ready for you to use for performing the tests. In some cases, there are also comments that list out limitations where the test cases will not work.

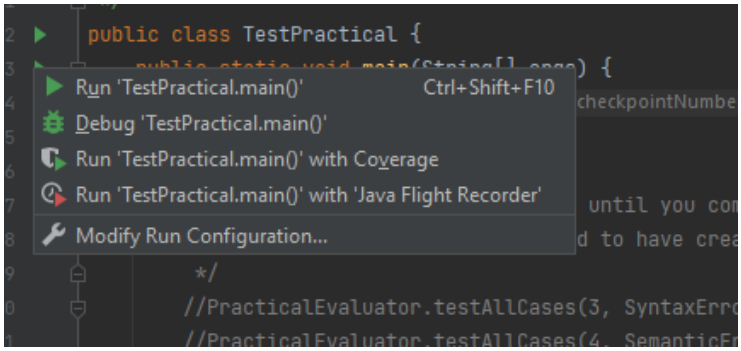
Running the TestPractical.java file will execute all the test cases you leave uncommented in its main method. You can choose to test single checkpoints, or even all of the checkpoints at once you are done. By using this tool, you can quickly evaluate for yourself that the output is as expected.

If you happen to run into any big problems with the tool, have suggestions for what would help you use it more effectively, or would otherwise like to provide feedback on it you can send me an email at: peter.mitchell@flinders.edu.au

3.1 Example for Checkpoint 2: Hello World

The Hello World example is being used here because it should already be testable without any changes or writing of code. This shows how it should look when you pass the test case. The first version of the example will show the default output you should see when you run the TestPractical.java and then an example where a typo has been introduced.

To run the TestPractical.java file, I would recommend you run it by using either the shortcut for running current file, or by using the green arrow next to the main method. On Windows you can see the shortcut in the image is Ctrl+Shift+F10. You will see the shortcut for Mac when you click the green arrow.



3.1.1 Example where Everything is Correct

Checkpoint 2 Test Number: 1

Expected output:

Hello World

End expected output.

Actual Output:

Hello World

End of Actual Output

Validating Checkpoint 2 Using Input Variation: 1

Finished Validation of Checkpoint 2.

Correct Lines: 1, Incorrect: 0, Incorrect Extra: 0, Unnecessary Blanks: 0, Missing Lines: 0

Validation PASSED!

All Checkpoint 2 tests completed. 1 of 1 successfully passed.

3.1.2 Breakdown of the Correct Output Example

1. You can see at the top it starts with “Checkpoint 2 Test Number: 1”, this tells you that the text following is related to this single test case identified in the file “2_1.out”.
2. Next you can see a section that begins with “Expected output:” and ending with “End expected output.”. The text shown between these two lines is the exact output for the test case. You should be trying to match this output exactly. This information is read in from the test case file “2_1.out”.
3. The next section starts with “Action Output:” and ends with “End of Actual Output”. The content between these two is the output produced from running your program.
4. The last section starts with “Validating Checkpoint 2 Using Input Variation: 1” and ending with “Finished Validation of Checkpoint 2.” There is nothing between these two because it passed the test. If there were incorrect lines, extra lines, or other unexpected differences found they would be listed here. This is summarised just after with showing the number of different occurrences of validated lines.
5. The test case finishes with “Validation PASSED!” or “Validation did not pass all expectations.” and after all the test cases (there can be more than one), a summary shows the number of total cases that passed.

3.1.3 Example where a Typo is Introduced

For this example, the output line is modified with a comma in the middle.

```
System.out.println("Hello, World");
```

The output when running the same tests against this would be.

```
Checkpoint 2 Test Number: 1

Expected output:
Hello World
End expected output.

Actual Output:
Hello, World
End of Actual Output

Validating Checkpoint 2 Using Input Variation: 1
Line mismatch. Expected on line 1:
Hello World
Got:
Hello, World
Finished Validation of Checkpoint 2.
Correct Lines: 0, Incorrect: 1, Incorrect Extra: 0, Unnecessary Blanks: 0, Missing Lines: 0
Validation did not pass all expectations.

All Checkpoint 2 tests completed. 0 of 1 successfully passed.
```

You can see that now in the validating section it shows where the mismatch has occurred by highlighting the correct text in green up until it is invalid and then appears as red. The lines at the end also appear in red to indicate there were errors. It was not shown in the first correct version of running it, but these would appear in green if they passed.

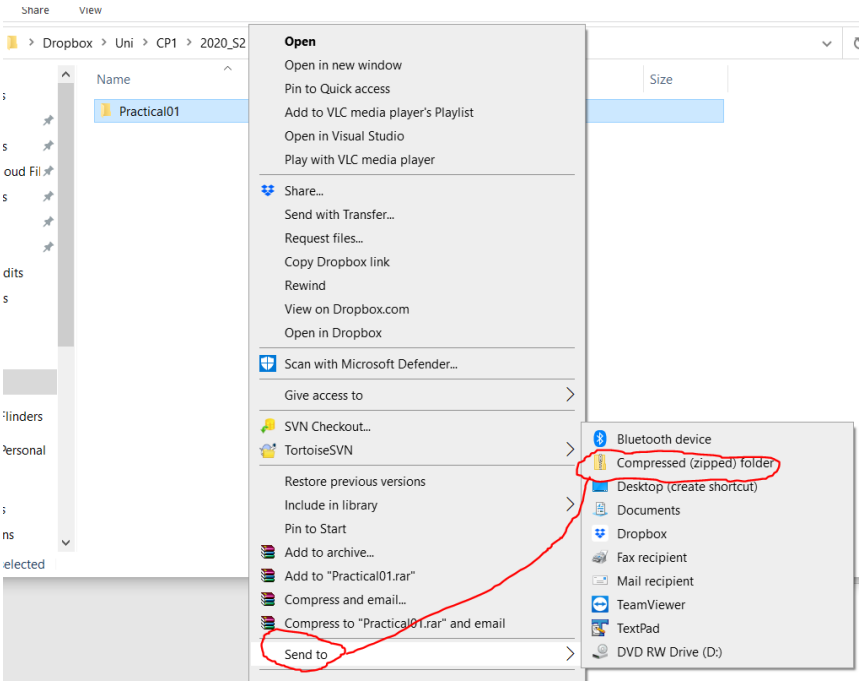
4 Troubleshooting Common Issues

4.1 Submitting Checkpoints via FLO

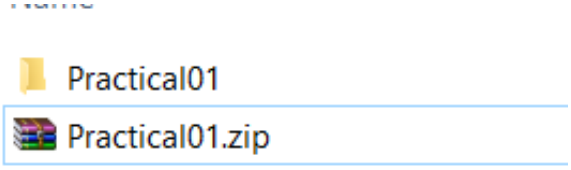
For online students you have the option of submitting your checkpoints by putting your project in a ZIP file and uploading it to FLO or showing your code during the practical time. You can submit your completed checkpoints at any time during the week to the submission boxes on FLO. You can follow the steps below for submitting your checkpoints via FLO.

Step 1) When you intend to submit your checkpoints via FLO try to write comments into your code to explain what your code does. You do not need a comment for every line, but some general explanation to show that you understand the code is acceptable.

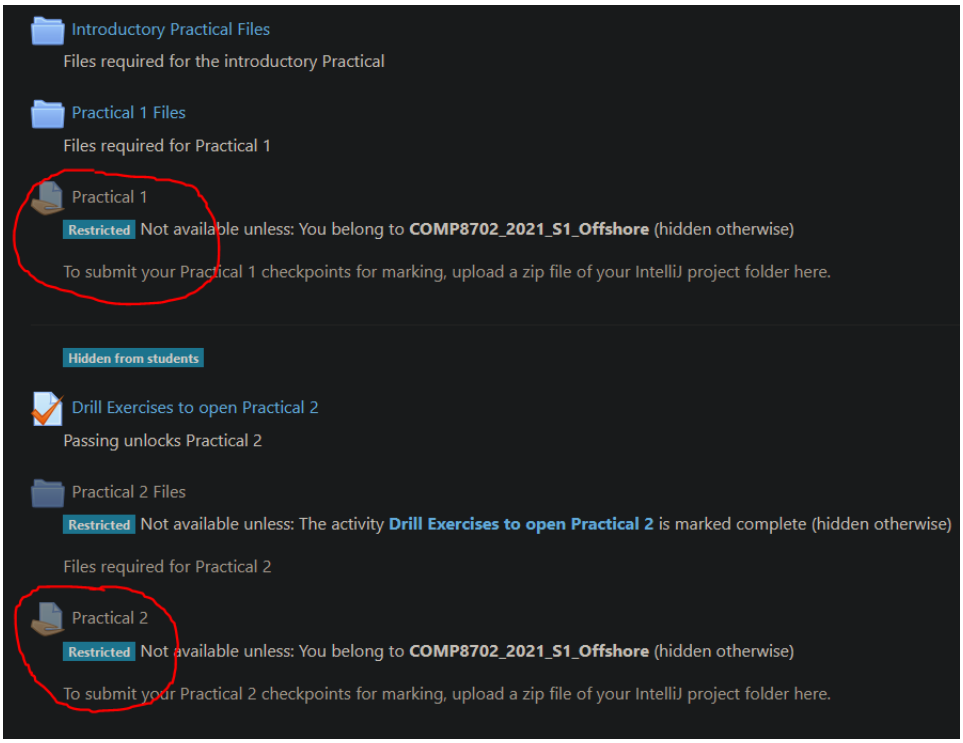
Step 2) Make sure you select the correct folder to submit. The folder you should submit is the one above the “src” folder. Then you can right click on the folder. Select “Send To” -> “Compressed (zipped) folder”. You can see this in the image below. You will probably have a lot less options in your menu.



Step 3) Once completed correct you should have a “.zip” in addition to your folder line in the image below. I would recommend opening your ZIP file and making sure all the files are as you expect inside. To avoid accidentally uploading just the start files or a wrong project.



Step 4) Next you can open FLO and locate the submission box for the associated practical you are trying to submit. These will be located just under the folder for that practical. As seen circled below.



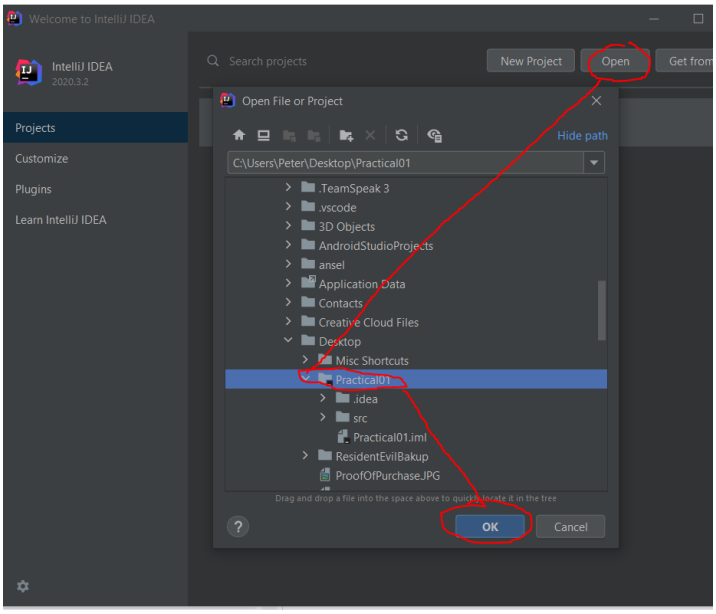
Step 5) Upload your file and at this point you can choose to either submit straight away. Or if you choose to leave it as a draft you can ask during the practical to check it for you before submitting it.

Step 6) Sometime after you have uploaded your practical, the practicals will be marked sometimes with feedback if applicable. This will likely normally happen during the Friday practical session.

4.2 Troubleshooting IntelliJ Project not allowing Run or Build

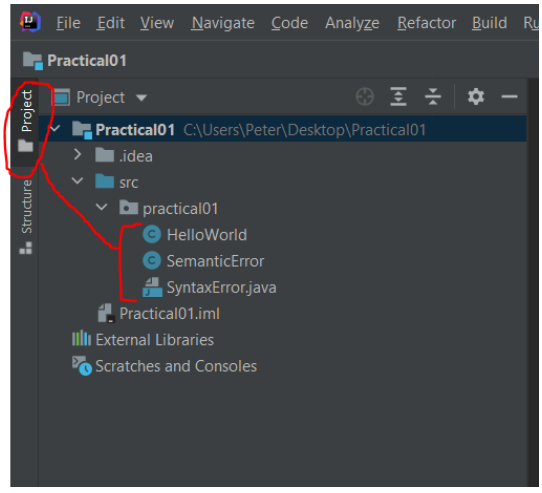
4.2.1 Solution 1: Incorrect Folder Opened

The most common error is opening the wrong folder. If you open the project from the wrong folder it will still let you do it, but the project will not be something you can run. To fix this you can just reopen the project from the correct folder. By either clicking File->Open or File->Close Project and then clicking “Open” again. When you are opening it should look like the following:



You can see the “Practical01” folder being opened has the “src” folder below it. Make sure this is the case. You will also see a slightly different folder icon when the folder is also a project.

When the project has opened you should see the following. If you don’t, simply click the “Project” tab that is circled on the left. And then expand “Practical01” with the arrow, then the “src” folder with another arrow. “HelloWorld” is the first file you will use for checkpoints in the PDF.



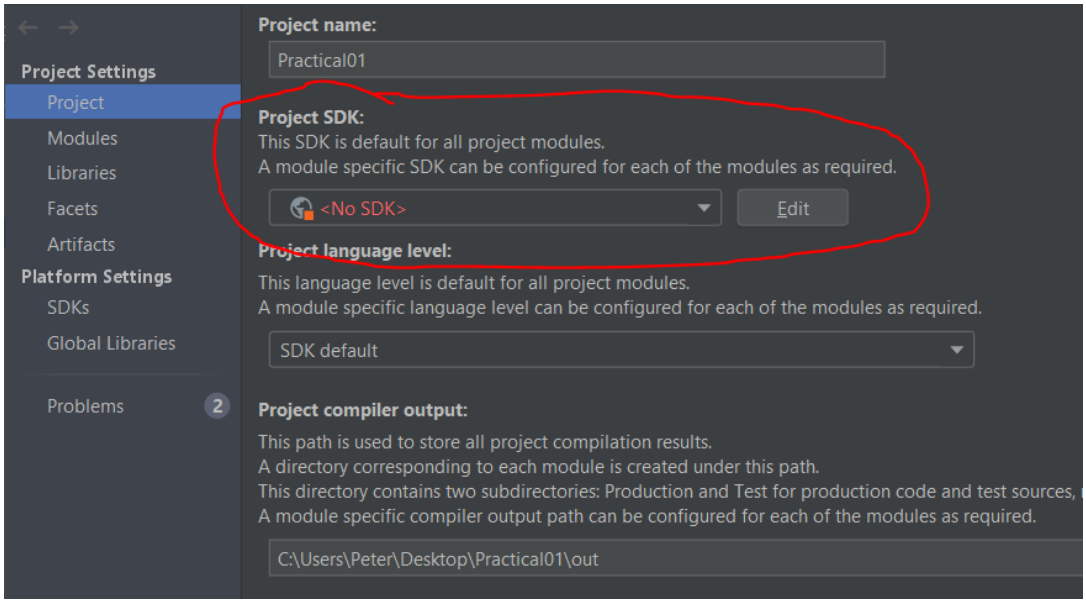
4.2.2 Solution 2: JDK Not Found

There are two situations this may be the case. The first is you are just using an older version of the JDK (something that is not JDK 15). You will see something like the below probably.

On the right side there is a “Configure” button. Click that button and select an existing JDK on your system. You may need to click on Add JDK if it does not show up. If for some reason you can’t find any it may be the case that you did not install the JDK and should follow the steps from the introductory practical to go and install JDK 15.

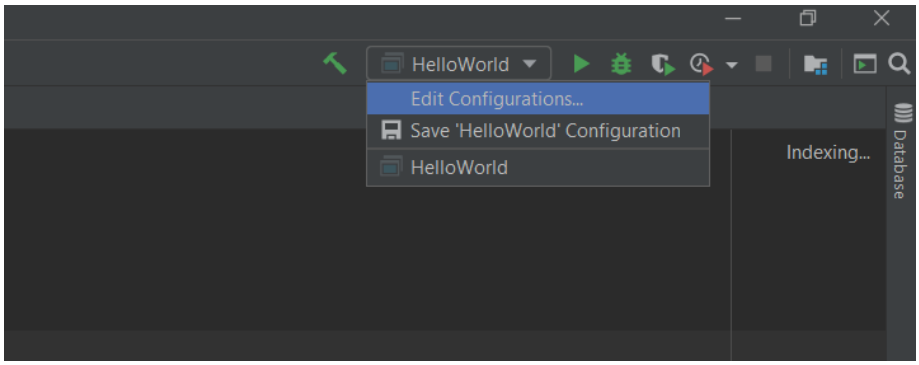
If for some reason you need to change the version of JDK after it has already been configured, you can follow these steps:

Step 1) Open File->Project Structure

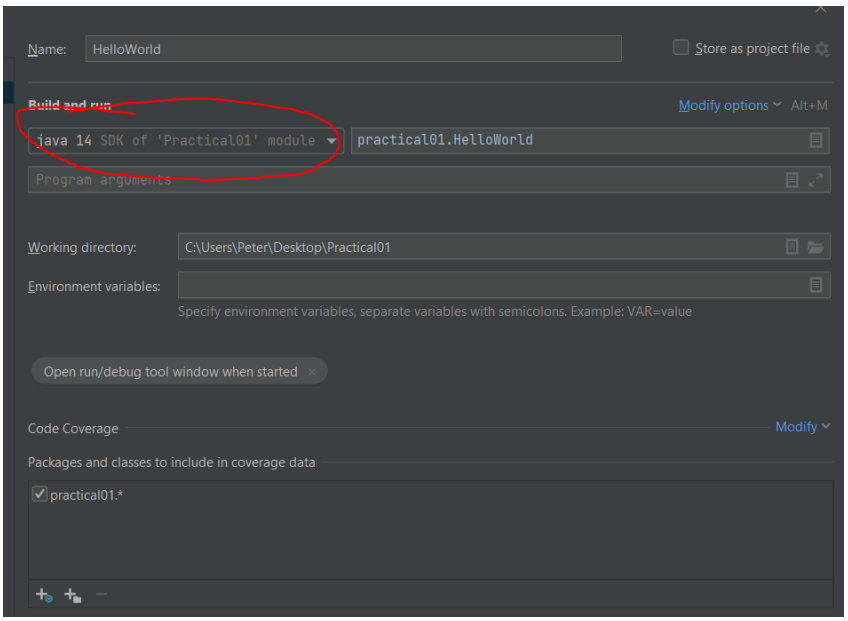


Step 2) In the clicked dropdown you can select your JDK. This will normally fix everything, but if not, you can continue on with the next couple of steps.

Step 3) In the top right of your IDE you will find the dropdown shown below. Click on Edit Configurations.

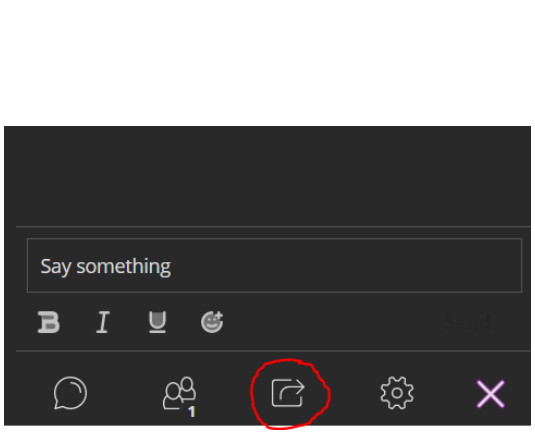


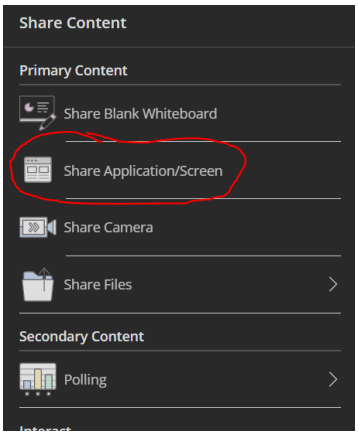
Step 4) Make sure the circled dropdown in the image below matches the save JDK version you selected in the previous steps. Then click OK.



4.3 How to share your screen in Collaborate

The circled icon below is the Share Screen button in Collaborate. Then a selection of options will appear, and you can select “Share Application/Screen”. Then you can either choose to share your whole screen or IntelliJ from the options it presents you.





5 Code Examples

5.1 Example of Input/Output

Below is an example of code that will take input from the user, calculate a result, and then output that result back to the user. Following this example is a brief explanation of each step.

```
import java.util.Scanner;

public class InputOutputExample {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter your age: ");
        int ageInput = scan.nextInt();

        int result = ageInput + 1;
        System.out.println("Next year you will be " + result
                           + " years old!");
    }
}
```

Step 1) The first statement you will see is the import statement that imports the Scanner library. This will need to be included any time you want to use a Scanner type object for reading input from the user. For the purpose of this example, it is assumed you understand that the main method must be nested inside a class. Importantly note that the name of the class must match the name of the .java file.

Step 2) Inside the main method the first line creates our Scanner. “Scanner scan” creates a variable of type Scanner (from our imported library) and calls this variable “scan”. The “=” operator then assigns this variable a reference to where the input will come from by creating a “new Scanner()” object using “System.in” as input. “System.in” in this case refers to the keyboard.

Step 3) Before allowing the user to enter a value we should tell the user what they are supposed to enter. In this case the user is prompted with a println statement that asks “Enter your age: “. You can shortcut to showing “System.out.println()” by typing in sout and then hitting tab. “println” will print the message out and then move to the next line. If you were to use “print” without the “ln” the input could be typed in on the same line as the.

Step 4) After telling the user what to enter we can make the program wait for the user to type in input by using our Scanner object. First, we declare a variable for this input to be stored in and then assign it a value from the user’s input. “int ageInput” declares a variable of type int (whole numbers) and calls it ageInput. Then the “=” assigns it the value from “scan.nextInt()”. “scan.nextInt()” uses our Scanner object to wait for the user to enter some input and then press their enter key. Note that this method does allow you to enter anything and does not check if you entered a number, there are other ways this is handled later in the topic.

Step 5) After receiving the input from the user the next step is to calculate our result. This is stored in a new variable to clearly represent its purpose. “int result” similar to the previous int variable declaration creates a variable of type int and calls it “result”. Then we assign it a

value based on that of our other variable. “ageInput + 1” will take the value from our age Input and add one to it. Eg, if the input were 5 it would now be 6 in result.

Step 6) The last step is to output the result. This last println statement demonstrates how you can show variables as part of a message being printed out. You simply need to have any elements you wish to print joined with a + operator. This will join the string values and any variables you may wish to print out. As you can see in this example it is possible to spread statements over multiple lines if necessary. The “+” operators will continue joining everything together over multiple lines. This does not add extra lines to the output.

5.2 Example of Drill Exercise

Let us say for example that the “Example of Input/Output” code was the solution to a Drill exercise; how might this look and how would you go about putting the code into Coderunner.

Example Question:

Write a program that will tell the user what their age is next year.

Input	**Output**
5	Enter your age: Next year you will be 6 years old!

As you can see there are three parts to the question. The text at the top and then an “Input” example and an “Output” example. This is just one possible value it will test your program with. You should expect multiple different values to be tested. For example, you could not just print out “6” all the time. You will also notice that the output does not show the input, but you can see where it would have been typed in. In this case if you were to run the program using this input it may look something like the below:

```
Enter your age:
5
Next year you will be 6 years old!
```

If our code from the Input/Output Example was the solution for this task, the code we would paste into Coderunner would be the following:

```
Scanner scan = new Scanner(System.in);
System.out.println("Enter your age: ");
int ageInput = scan.nextInt();

int result = ageInput + 1;
System.out.println("Next year you will be " + result
        + " years old!");
```

You will notice that the import statement, the class, the main method, and other surrounding braces “{“ have been excluded. When putting your code into CodeRunner for the Drill exercises you will only place in the code that is functional. Note importantly that in later weeks you will be asked to write code for methods or classes. In these cases, you will be provided a method or class structure and that will also be part of what goes in the CodeRunner window.

5.3 Separating Numbers into Individual Digits

There are a few different ways you can do this. The first examples will show how you can separate into a variable for each digit if you already know how many digits there are. The other examples will show how you can iterate through individual digits by using a loop. Loops will be covered in later weeks. The last example shows how you could separate out a single digit without going through all the steps to reach it.

```
import java.util.Scanner;

public class SplitNumberExample {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter a 5 digit number: ");
        int input = 12345; //scan.nextInt(); // Change to use input if you want

        // Example using 12345 as input separating into a variable for each digit
        int fifthDigit = input % 10; // fifthDigit = 5
        input = input / 10;          // input = 1234
        int fourthDigit = input % 10; // fourthDigit = 4
        input = input / 10;           // input = 123
        int thirdDigit = input % 10; // thirdDigit = 3
        input = input / 10;          // input = 12
        int secondDigit = input % 10; // secondDigit = 2
        input = input / 10;           // input = 1
        int firstDigit = input;       // Could skip this step and just use "input"

        System.out.println("The odd numbered digits are: ");
        System.out.println(fifthDigit + ", " + thirdDigit + ", " + firstDigit);
        // 5, 3, 1

        // Example using a loop to do the same thing:
        input = 12345;
        String result = ""; // String to gradually store the result in
        for(int i = 5; i >= 1; i--) { // step from digit 5 down to digit 1
            if(i % 2 != 0) { // if the index is odd
                result += (input % 10) + ", ";
            }
            input = input / 10;
        }
        System.out.println(result); // 5, 3, 1,

        // How about looping through all numbers regardless of length?
        input = 1234567;
        result = ""; // Result to store the number separated in order
        while(input > 0) {
            result = (input % 10) + ", " + result;
            input = input / 10;
        }
        // One problem with this is any number that ends with 0s will not work!
        System.out.println(result); // 1, 2, 3, 4, 5, 6, 7,

        // How about if you just want to know a specific digit?
        input = 1234567;
        System.out.println("Second Digit is: " + ((input / 100000) % 10));
        // Simply divide by 1 followed by 0s
        // for every number after the one you want, then % 10.
        System.out.println("Fifth Digit is: " + ((input / 100) % 10));
    }
}
```

5.4 Example of Using Booleans for Output

This example demonstrates two different ways you could use boolean type variables. The first example shows that you can print out true or false based on either an expression or the result of that expression stored in a variable. The second example demonstrates how you can chain parts of messages together based on the values of variables. You can use this approach for the Biscuits practical task.

```
import java.util.Scanner;

public class BooleanExample {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a number: ");
        int input = scan.nextInt();
        boolean result = (input == 42);
        System.out.println("The input is the answer to everything: " + result);
        /* Outputs either:
         The input is the answer to everything: true
         or:
         The input is the answer to everything: false
         */

        // Alternative with if statements
        System.out.print("You entered the answer to ");
        if(input == 42) {
            System.out.println("everything!");
        } else {
            System.out.println("nothing important.");
        }
    }
}
```

5.5 Example of Formatting with Math.round and DecimalFormat

The following example shows how you can use either Math.round or DecimalFormat to round numbers in Java. To put it simply you should use the DecimalFormat when you are printing out the value in a println method call. And you should use Math.round when you are continuing to use the number in some way (in the case of Drills this is typically where you are returning the number from a method).

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class FormattingExample {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter an Integer: ");
        double input = scan.nextDouble();

        /* Example using Math.round
            Use this when you want to keep the value as a number.*/
        double exampleOne = Math.round(input * 10) / 10.0;
        System.out.println("One Decimal Place: " + exampleOne);

        double exampleTwo = Math.round(input * 100) / 100.0;
        System.out.println("Two Decimal Places: " + exampleTwo);

        double exampleThree = Math.round(input * 1000) / 1000.0;
        System.out.println("Three Decimal Places: " + exampleThree);

        /* Example using DecimalFormat
            Use this when outputting as part of a println statement. */
        // Always show 1 decimal place
        DecimalFormat formatOne = new DecimalFormat("0.0");
        // Always show 2 decimal places
        DecimalFormat formatTwo = new DecimalFormat("0.00");
        // Up to 2 decimal places
        DecimalFormat formatThree = new DecimalFormat("0.##");

       System.out.println("One decimal place always: " + formatOne.format(input));
       System.out.println("Two decimal places always: " + formatTwo.format(input));
       System.out.println("Up to 2 decimal places: " + formatThree.format(input));

        // Extra examples. The first will make the decimal part disappear entirely
  System.out.println("Up to 2 decimal places (12.0): " + formatThree.format(12.0));
  System.out.println("Up to 2 decimal places (12.4): " + formatThree.format(12.4));
  System.out.println("Up to 2 decimal places (12.456): "
                     + formatThree.format(12.456));
    }
}
```

5.6 Example of a Class (SaleItem)

The following SaleItem class demonstrates examples of private instance variables, a default constructor, method overloading with another constructor, getters/setters for the instance variables, print methods, and a method that uses the class to create a custom result.

```
import java.text.DecimalFormat;

public class SaleItem {
    private String name;
    private double cost;

    // Default Constructor
    public SaleItem() {
        name = "NAME_NOT_SET";
        cost = 0;
    }

    // Constructor with both parameters
    public SaleItem(String name, double cost) {
        this.name = name;
        this.cost = cost;
    }

    // Setter for the name instance variable
    public void setName(String name) {
        this.name = name;
    }

    // Getter for the name instance variable
    public String getName() {
        return name;
    }

    // Setter for the cost instance variable
    public void setCost(double cost) {
        this.cost = cost;
    }

    // Getter for the cost instance variable
    public double getCost() {
        return cost;
    }

    // Print the sale item
    public void print() {
        DecimalFormat fmt = new DecimalFormat("0.00");
        System.out.println(name + " costs $" + fmt.format(cost));
    }

    // The toString method is called when you try to print an object
    public String toString() {
        DecimalFormat fmt = new DecimalFormat("0.00");
        return name + " costs $" + fmt.format(cost);
    }

    // Takes discount as a value like 0.2 to represent 20% off
    public double getDiscountedCost(double discount) {
        return cost * (1-discount);
    }
}
```

5.7 How to Approach Checkpoints with Multiple Parts

Using the SaleItem class as an example, this class may have been developed over multiple checkpoints. This means each checkpoint will have its own tasks and expected output. You can preserve and separate out this in a very sensible way by following the example below. By placing each checkpoint's test code in a separate public void method, you can call it from the main method as shown. Sometimes lines of code will stop working as you move to later checkpoints, those can just be commented out.

I would strongly recommend using this approach to keep your code organised and this lets you check that previous checkpoints still work after making changes.

```
import java.text.DecimalFormat;
public class SaleItemExample {
    public static void main(String[] args) {
        SaleItemExample = new SaleItemExample();
        saleItemExample.task1();
        saleItemExample.task2();
        saleItemExample.task3();
    }

    public void task1() {
        System.out.println("Task 1:");
        SaleItem chocolate = new SaleItem();
        // Can comment out invalid lines after making changes
        // In this case once the variables become private they
        // are no longer valid.
        //chocolate.name = "Chocolate";
        //chocolate.cost = 9.5;
        chocolate.print();
    }

    public void task2() {
        System.out.println("\nTask 2:");
        SaleItem chocolate = new SaleItem("Chocolate", 9.5);
        SaleItem bread = new SaleItem("Bread", 3.5);
        chocolate.print();
        bread.print();
        chocolate.setCost(8.5);
        chocolate.print();
    }

    public void task3() {
        System.out.println("\nTask 3:");
        SaleItem chocolate = new SaleItem("Chocolate", 9.5);
        System.out.println(chocolate); // This will call the toString() method
        double discountedPrice = chocolate.getDiscountedCost(0.4);
        DecimalFormat fmt = new DecimalFormat("0.00");
        System.out.println("Price at 40% off: $" + fmt.format(discountedPrice));
    }
}
```

Output:

```
Task 1:
NAME_NOT_SET costs $0.00

Task 2:
Chocolate costs $9.50
Bread costs $3.50
Chocolate costs $8.50

Task 3:
Chocolate costs $9.50
Price at 40% off: $5.70
```

5.8 Types of Loops and When to Use Them

There are four types of loops: for loops, while loops, do while loops, and for-each loops. The for loop is good when you know a fixed number of iterations need to happen. The while loop is good when you do not know how many times something needs to repeat, and it could be 0 or more times. The do while loop is the same as a while loop except it will run the code inside 1 or more times. A for-each loop can be used when you do not care what the index is in the array. If you need to know the index use a regular for loop instead. The examples below show how each may look.

```
import java.util.Scanner;

public class LoopExample {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        /*  for(init; condition; increment)
            int i = 0; Initialise
            i < 5;  Condition. (Loop while true)
            i++     Increment (end of every loop) */
        for(int i = 0; i < 5; i++) {
            System.out.print(i + " ");
        }
        System.out.println();

        // while(condition) loop
        System.out.print("Eat cake? ");
        String input = scan.nextLine();
        while(!input.equalsIgnoreCase("yes")) {
            System.out.print("Surely you want to eat cake? ");
            input = scan.nextLine();
        }
        System.out.println("Excellent! Much cake will be eaten.");

        // do while loop
        do {
            System.out.print("What noise does a dog make? ");
            input = scan.nextLine();
        } while(!input.equalsIgnoreCase("woof"));

        // for-each loop: for(Type varName : array)
        String[] colours = new String[] { "Red", "Green", "Blue" };
        // For each colour in colours
        for(String colour : colours) {
            System.out.println(colour);
        }
    }
}
```

Output:

```
0 1 2 3 4
Eat cake? no
Surely you want to eat cake? no
Surely you want to eat cake? yes
Excellent! Much cake will be eaten.
What noise does a dog make? bork
What noise does a dog make? meow
What noise does a dog make? woof
Red
Green
Blue
```

5.9 Switch Statement Example

The example below shows an example of a switch statement using char types. You will normally use either int or char types with switch statements. Try out the code if you are not sure what it will do with the characters y, b, 1, 2, and any other character.

```
import java.util.Scanner;

public class SwitchExample {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter some text: ");
        String input = scan.nextLine();
        // switch based on the last character's value
        char lastChar = input.charAt(input.length()-1);
        switch(lastChar) {
            case 'y':
                System.out.println("Why so serious?");
                break;
            case 'b':
                System.out.println("Be happy!");
                break;
            case '1':
            case '2':
                System.out.println("12121212");
                break;
            default:
                System.out.println(lastChar + " was not important.");
                break;
        }
    }
}
```

5.10 Writing Drills with Methods

If you are reading this, you have probably reached Drill exercise questions that involve writing methods and may wonder how you can test your methods. This section will show an example question, what the solution may look like, and a good way to represent testing your code in IntelliJ. The input box for methods is a bit different for methods and classes as it will shows how the method would be called and then what should be printed out as a result.

Example Drill Question:

Write a method to return a list of even numbers between a start number and an end number. This should be returned as a String with commas after each number.

Input	Output
System.out.println(evenNumbersBetween(1,10));	2,4,6,8,10

Start code: (this is the code it would give you in the box to show you the expected method definition)

public String evenNumbersBetween(int start, int end) {

}

You could efficiently solve and test this question using the following code. You will see it tests multiple different examples with the testEvenNumbersBetween() method. You can write a pair of methods like this for every Drill question you attempt. One with examples to test, and the other as the Drill you are solving.

```
public class DrillMethodExample {
    public static void main(String[] args) {
        DrillMethodExample drillMethodExample = new DrillMethodExample();
        drillMethodExample.testEvenNumbersBetween();
    }

    public void testEvenNumbersBetween() {
        // 2,4,6,8,10,
        System.out.println(evenNumbersBetween(1,10));

        // -2,0,2,4,
        System.out.println(evenNumbersBetween(-3,4));

        // Empty String expected
        System.out.println(evenNumbersBetween(1,1));
    }

    public String evenNumbersBetween(int start, int end) {
        String result = "";
        for(int i = start; i <= end; i++) {
            if(i % 2 == 0) {
                result += i + ",";
            }
        }
        return result;
    }
}
```

5.11 Arrays Example

The following code shows three separate examples using arrays. Demonstrating how to create arrays with a specific size, with an initialiser list, or based on the length of another array. And then setting values, printing values, and calling methods based on elements in the array.

```
public class ArraysExample {
    public static void main(String[] args) {
        // Create an array with a specific size and fill with 5
        int[] numbers = new int[5];
        for(int i = 0; i < numbers.length; i++) {
            numbers[i] = 5;
        }

        // Create an array with an initailiser list and print values
        String[] words = {"Example", "Words", "Here"};
        for(int i = 0; i < words.length; i++) {
            System.out.println(words[i]);
        }

        // Create an array based on the size of another array and fill with values
        int[] numberLetters = new int[words.length];
        for(int i = 0; i < numberLetters.length; i++) {
            numberLetters[i] = words[i].length();
        }

        for(int i = 0; i < words.length; i++) {
            System.out.println("The word \"" + words[i]
                               + "\" has length: " + numberLetters[i]);
        }
    }
}
```

6 Other General Topics

6.1 Naming Things (Variables/Methods/Classes)

General Naming

- Keep names short and descriptive!

Naming Variables

- Try to use descriptive nouns consisting of 1 to 3 words to tell the reader what it is.
- Use Camel case: exampleVariable, anotherExampleVariable.
    - Starts with a lower-case word and capitalise the start of every other word.
- Constant variables (declared with “final” keyword) should be all caps with underscores between words like MAX_COUNT.
- Avoid single letters except in a few situations:
    - It makes sense when you have a coordinate for x, y, or z.
    - You are using it for a loop with i, j, or k. (i is often used to represent index).

Naming Methods

- Use Camel case like with variables but try to include a verb describing the action. For example, “getValue()” or “print()” indicate the action that will be performed.

Naming Classes

- Start all words with a capital. Eg, MyExampleClass. This makes them visually distinct from variables and methods when you are reading code.

6.2 Tips for Writing Useful Comments

Writing good comments does not actually start with the comments themselves. It starts with the code. Choosing to use informative concise variable names and methods will cut down a significant amount of the mental load when reading code. Often to the point where you do not necessarily need many comments. The following list of tips may help you in thinking about how to write your code better and make it clearer to the reader.

- Variable names should have a clear purpose that is typically a noun that concisely gives you an idea of the type of object it will contain.
- Methods should have a name that describes what they are doing with a verb. Methods should normally perform a specific job and not be too broad.
- If you have well written code with clear variable names and methods the code should be at least somewhat understandable as you read through.
- Having a comment for every single line is excessive and will probably make it harder to understand instead of easier.
- Separate your code with good white space into functionally similar blocks. (E.g., getting the input, doing something with the input, providing the output).
- Once you have code in smaller blocks separated by a line you can determine whether there is anything potentially unclear about the code in that block. If there might be any doubt you can write a concise comment over 1 to 2 lines that describes the code that follows.
- Often if your methods are small, they may not need comments inside the methods themselves if you follow standards with providing documentation of the methods and variables.
- In Java, this documentation is called Javadoc. Every method, class, and instance variable can have a comment. This comment should briefly describe the purpose of that element. For methods with return types, it can include a line indicating what the method returns. For methods with parameters, you can specific a message for each parameter.
- All the game example code I have provided on GitHub shows examples of Javadoc commenting and appropriate additional comments where necessary inside methods. I would recommend browsing through some of the code to see how the comments appear if you are interested in viewing examples of good comments.

6.3 Debugging Strategies

As you write more and more code you will run into situations where different types of errors. This mostly applies to situations where you are encountering logical errors and do not understand why your code is doing something. It could also be the case though that you are getting crashes from run time errors due to something unexpected. This short section will briefly discuss how you could go about solving the issue. After showing two strategies there is an example using both.

To solve any sort of bug in you code there are a few things you need to know. What is happening? Where is it happening? Why is it happening? By evaluating each of these you can try to narrow down the code that is causing the issue. To approach this, you can use a combination of multiple techniques, if necessary, but for the purpose of this I will briefly cover two common strategies.

6.3.1 Debugging Strategy One: Print Stuff Out

Sometimes there is no need to try and overcomplicate the process of analysis and just start to print stuff out. Some examples of what you might do with this are as follow.

- Print out a message to indicate "I have reached here" to visibly see output. Often you can use this if you are unsure whether code is running. By putting a message in there or even in multiple different places you can see which messages show up. If it is never showing a message you expect to see, this might mean that you have not called that code from anywhere, or perhaps you have a condition preventing it from branching correctly.
- Print out the values of variables. When you know all the code is running correctly you may not be sure if the values of your variables are correct for a calculation. Often you may want to check the output by putting in some debug lines of output that print out as many variables as necessary to check something. For example, if you are doing a calculation and are unsure if the calculation is being performed correctly you might output all the inputs for the calculation and the result. This means you can verify by reading the output to check that the values are as expected. This can also be used to check what a method is outputting if it is a mystery method, and you are not sure what it is doing. By looking at the output of a method you can get some idea of what to expect from it.
- Log the printed output to files if there is a lot of it. Sometimes you may have a very large data set and become overwhelmed trying to read through all the data. If it is becoming too much you could look at writing all that debug output to a file and then having it to review. This does also mean you will not be able to see it in real time normally. You do need to carefully consider whether you will be able to understand the point in the program where the log is coming from and what it means.

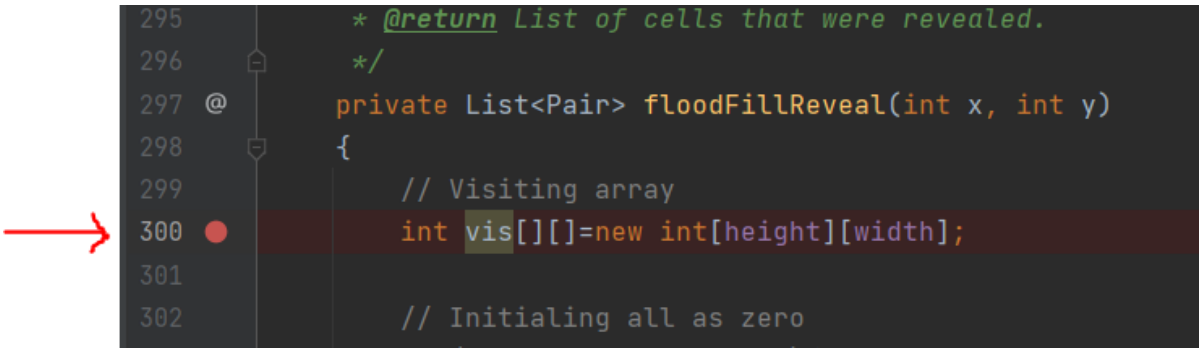
The problem you will find with hard coding this sort of debugging into your code is that sometimes you will forget to remove the messages. To some extent you can mitigate this issue by setting up a variable or similar technique where you can turn debugging on and off. This way you can write an if statement to only show the debug information when debugging

is on. If you are just testing with debug messages to find a problem and then removing those messages just after that will also solve this problem.

6.3.2 Debug Strategy Two: Use Breakpoints

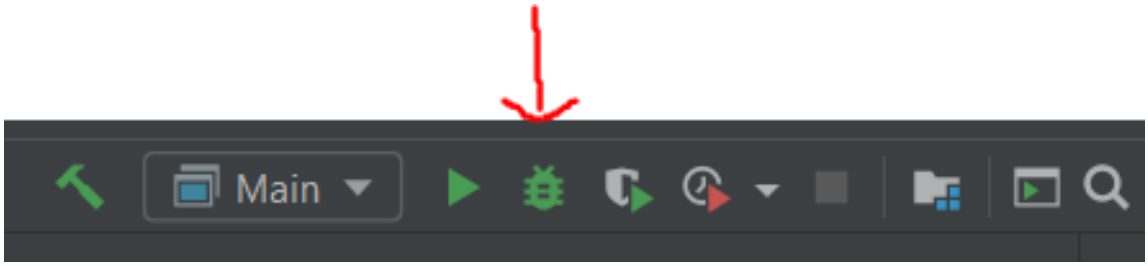
Most IDEs like IntelliJ have what are called Breakpoints. These are wonderful things that let you pause your program when you reach any point you choose so you can view the values of variables and step through a single variable at a time. This is an alternative approach to those presented with printing all your debug information out where you can step through and view the details as you go. The following is an example of stepping through how you would go about using Breakpoints inside IntelliJ. For this example, the code being observed is the floodFillReveal method in Board.java of the Minesweeper game.

Step 1) Set one or more breakpoints. You have probably accidentally done this at some point and not understood what it meant. If you click in the margin next to the line number of any code it will place a red dot like in the image below. Clicking just next to the 300 will put that red dot there and add a red shade behind that line of code.

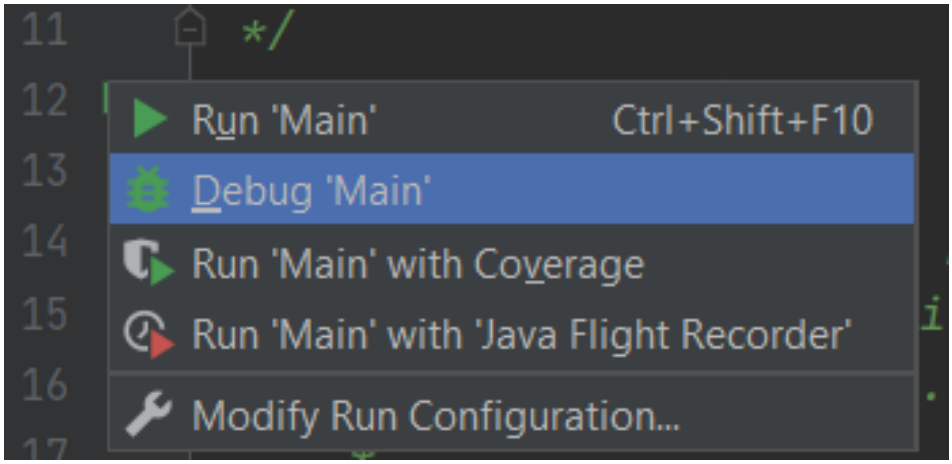


This indicates is that when the program is running this is the line where it should pause execution so you can view everything about the current state of the program at that point. You can add as many different breakpoints as you want, but normally setting them at places relevant to where you think a problem may be occurring. You can move them during execution and add/remove as many as you need.

Step 2) Once you have set any breakpoints you want to use to make them pause execution of your program you will need to use the little bug icon shown below. This will run your code with the debugger enabled. If you were to run your code with the play button it will just ignore breakpoints.

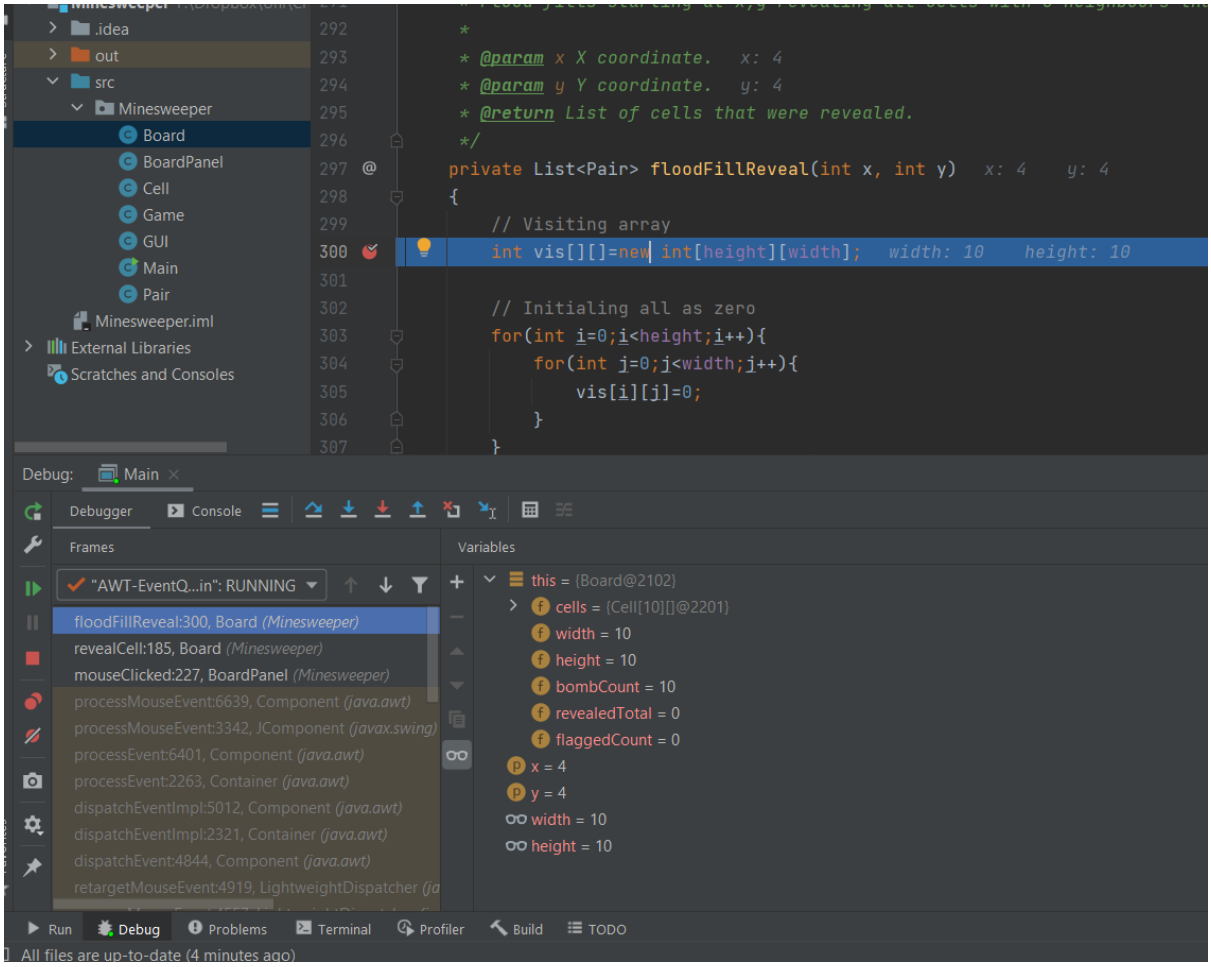


Alternatively, you could find this in the little green arrow at your main method instead as seen below.

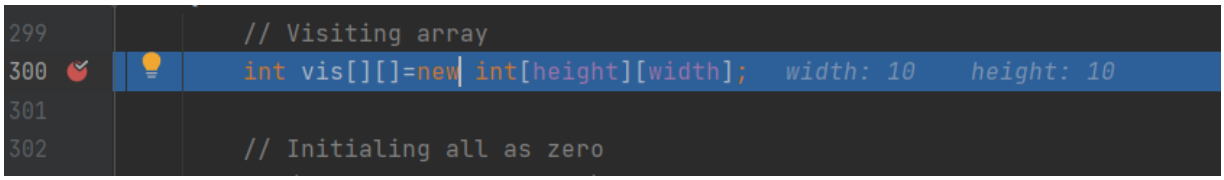


Once you run your code the debugger will stop any time it reaches a breakpoint. In this example it will be when the floodFillReveal() method is called.

Step 3) Once you hit a breakpoint you will have an additional debugger window appear at the bottom of your screen. The full view of this is shown below. I will walk through what each part of this is and what it means to you as a person doing debugging.

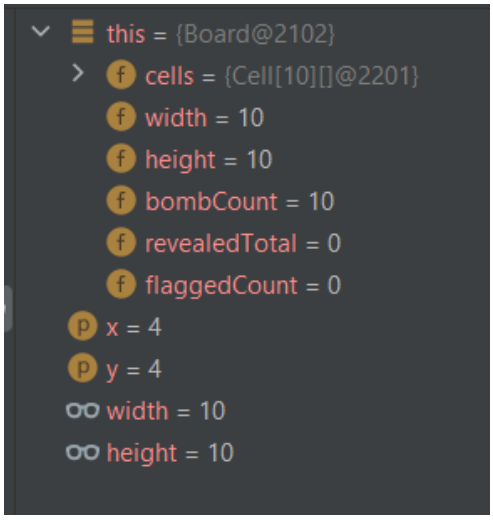


1. You will notice that the line is now highlighted showing the current line you have pause at in the program. This will change as you continue to move around in the program based on using the different tools it provides.

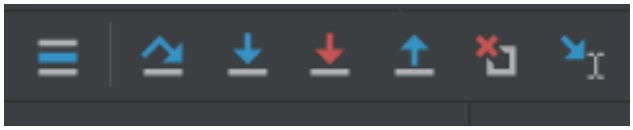


   As you can see in the above example image of this, the values for width and height are shown in grey off to the right of the line of code. This allows you to quickly see the values being passed to anything. You will notice this type of information appears in other places where it has already been evaluated like on the method declaration line where it says x:4 y:4.

2. Down the bottom in the middle panel, you will see something like what is shown below. As you can see it shows all the values of the instance variables stored inside the Board class. You could click on any of the little arrows to expand the individual cells out. This is useful if you need to look at the values nested in any way. It gives you a broad view of all your variables available in the current context.



3. The next thing you may want to do is move around inside the code. You obviously cannot go backwards in executing code, but there are a few ways you can go forward. Just above the previous dialog you will see the following. The list following summarises what each button does.



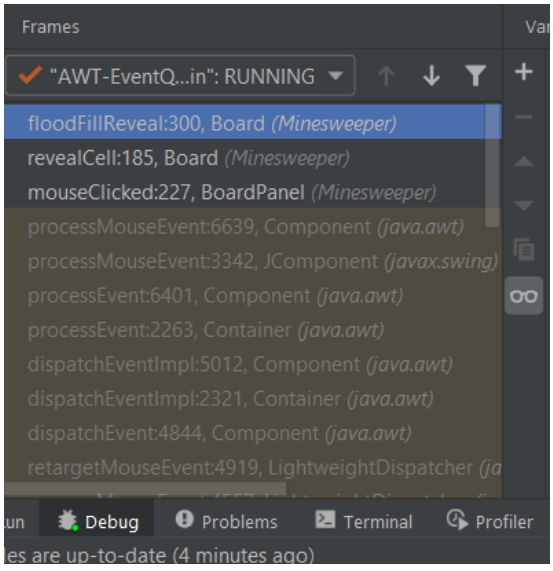
   a. The first icon with three lines will return you to the current line of execution if you ever become lost and need to find it again.
   b. The second with an angled arrow performs a “Step Over”. What this means is that it will step forward a line. If that line contains a method or code that is executed

somewhere else, it will not follow the execution to that point and will stay in the current method. You will normally use this to step through until you find something suspicious. And then use the “Step Into” to check in more detail.

c. The third with a blue arrow pointing down will perform a “Step Into”. What this means is that it will do the same thing as “Step Over” in going to the next line, but if that line contains a method it will step inside that method so you can continue stepping through one line at a time wherever the code goes.
d. The fourth with a red arrow pointing down is a very similar variation of “Step Into”.
e. The fifth with a blue arrow pointing up performs a “Step Out”. This makes the code in the current method finish and takes you up to the place where the current method was called from.
f. The sixth is not important for now.
g. The seventh as an arrow pointing at a cursor is for making the code continue running until it reaches the position of your cursor. So that you do not have to continually step just one line at a time in longer code.

By combining these you can step through each line of code that is necessary and you will see the values changing based on results of the code in either the list of all variables shown in point 2, or the inline values shown within the code.

4. The next part to look at is what is called the call stack. This is the sequence of methods that were called to reach the current point of execution. You will see in the image below that it shows the stack from the current point you are at and goes backwards through all the elements. You can see that floodFillReveal() was called from revealCell() which was called from the mouseClicked() method. This may be useful for checking the path that methods were called through. You can see there are many lower down the list. It will grey out all the method calls that are not part of code in your own code. In this case if you read through what they are you will see they are related to the events being triggered in the window by the AWT library. Generally, if they are greyed out most of the time it will not be relevant to your debugging.



5. The last part is the panel of buttons seen below. The dot points are for each of the shown buttons in the bottom right of the IntelliJ client. The most important button is probably the resume running button since that will let you stop going line by line and resume back to running as normal. At least until the next breakpoint is hit.

- Restart running your application.
- Modify run configurations.

- Resume running (use if paused by a breakpoint or after you pause).
- Pause (lets you pause whenever you want and start stepping through)
- Stop running.

- View Breakpoints (shows a dialog with all set breakpoints).
- Mute all breakpoints (makes debugger ignore them).

So now you have been introduced to most of the tools used in breakpoints for IntelliJ. Many other IDEs use similar types of functionality for breakpoints, so it is worth becoming familiar with it. Once you have learned to use breakpoints you can step through parts of your code with ease to check they are working properly. When you are stepping line by line and watching the values change it will improve your ability to evaluate what could be the source of a logical error.