Java Basics Iteration 3

Previous Iteration	Wiki Home	Forums
Iteration 2	Wiki Home Page	Forums

Previous Iteration	tWiki Home	Next Iteration
Iteration 2	tWiki Home Page	Iteration 4

Table of Contents

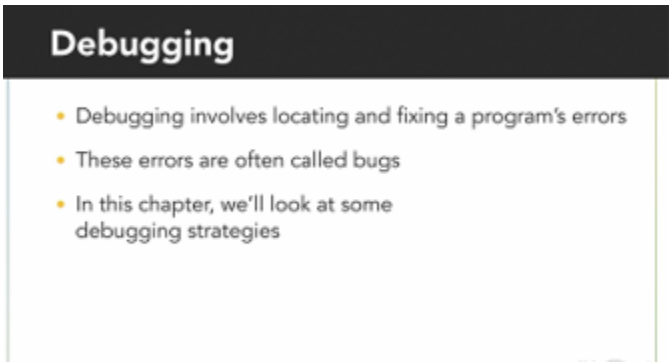
Iteration Overview

During this iteration, students will learn how to debug syntax and logical errors and how to use functions, return types, and parameters in Java. For a project, you will create a function that calculates an employee's salary, specifically by defining various functions, adding parameters, adding return types, and then calling them in your main function.

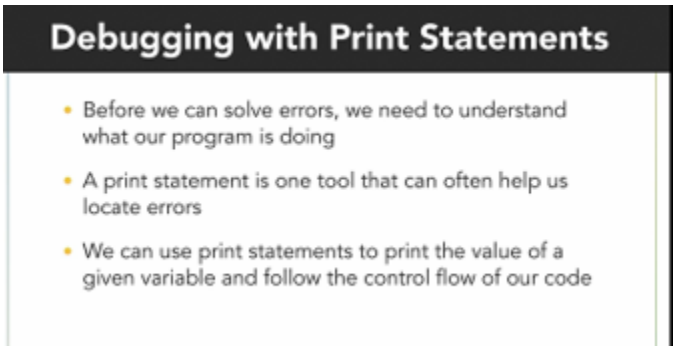
Save screenshots of your work along the way to show the Topic Advisor during your demo at the end of this iteration.

Debugging syntax and logical errors

Writing code sometimes we can make a mistake and get an error. These errors can cause our code to give the wrong output or crash our program entirely. And you've probably already experienced a little bit of this in the course so far. You may have heard of the term debugging before. This process involves locating and correcting code errors in your program. We often call these errors bugs, because computer scientist pioneer Grace Hopper coined the term bug when she found a moth causing an error in her early computer.



In this iteration, we'll look at some strategies for how to debug your programs and how to fix some common errors. Before we can solve errors, we have to understand how our program is executing code so that we can find them. One way we can understand how our program is working more fully and locate an error is by using print statements.

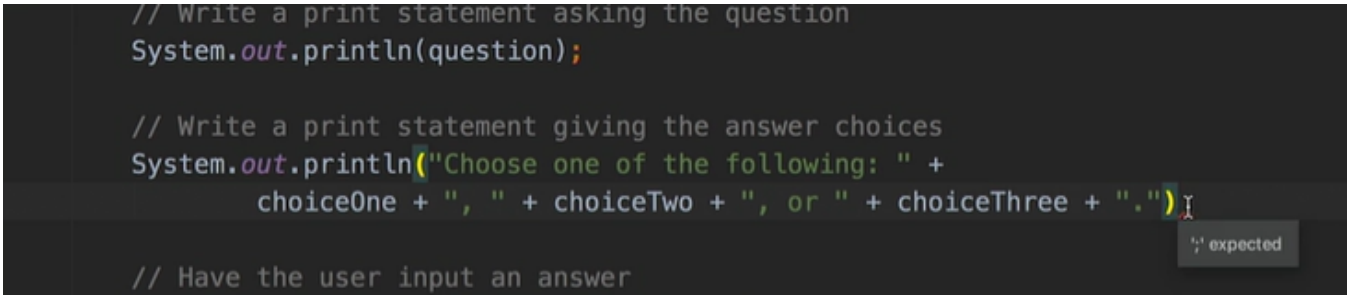


In this lesson, we'll take a look at our multiple choice program from the last iteration but with a few errors introduced. The first errors in the program we going to look at are syntax errors, or syntax bugs. Syntax errors often cause your program to fail before it's even executed because the computer has to understand your code in order to run it. This means if something's misspelled, or a certain symbol is missing, the program won't run, because your code is not in the right format.

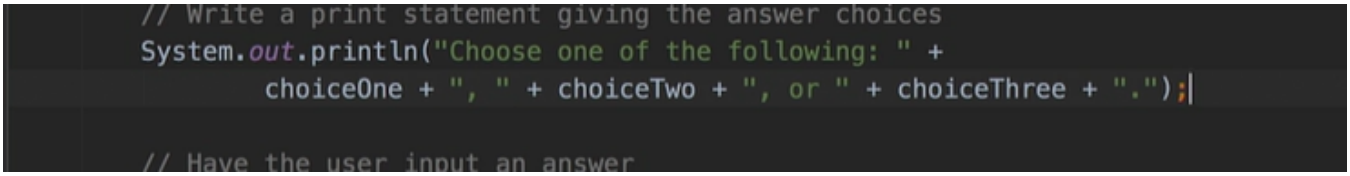
So, let's find some syntax errors in this program.

```
import java.util.Scanner;

public class Main {

    public static void main(String args[]) {
        String question = "What is the largest planet in our solar system?";
        String choiceOne = "earth";
        String choiceTwo = "jupiter";
        String choiceThree = "saturn";

        String correctAnswe = choiceThree;

        // Write a print statement asking the question
        System.out.println(question);
```

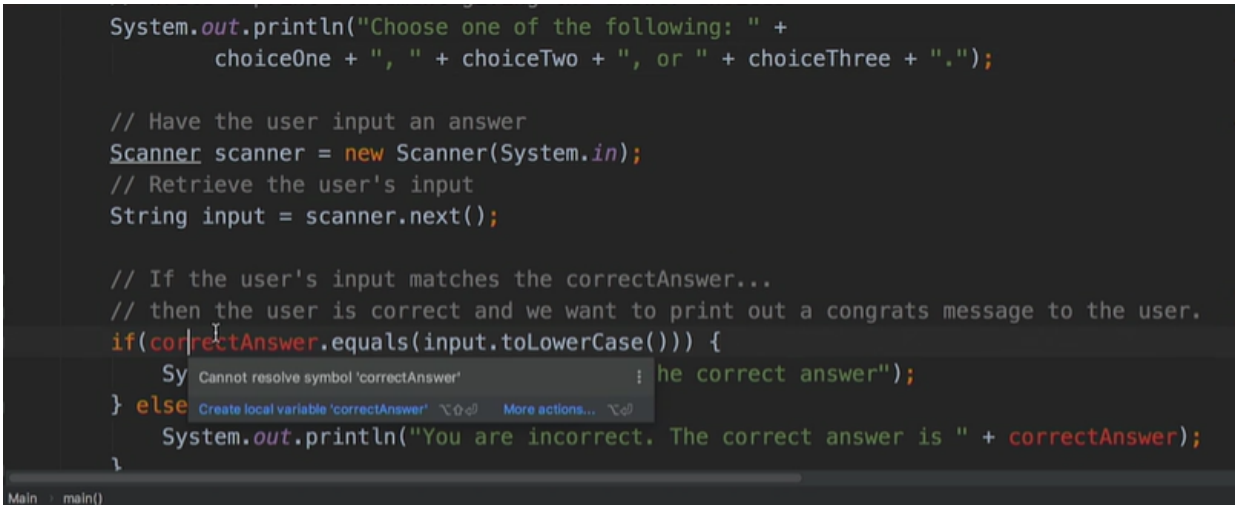
These can be easier to find with an IDE because an IDE tells you what line has the error. It may not be that specific line you have to change, but it's probably something around it. Let's take a look! Here on the right, we have some red symbols, and so if we click on the first one, here we have an error that says, "expected semicolon." This is a syntax error,



and all we have to do to fix it is to add the semicolon, since we were just missing a semicolon.



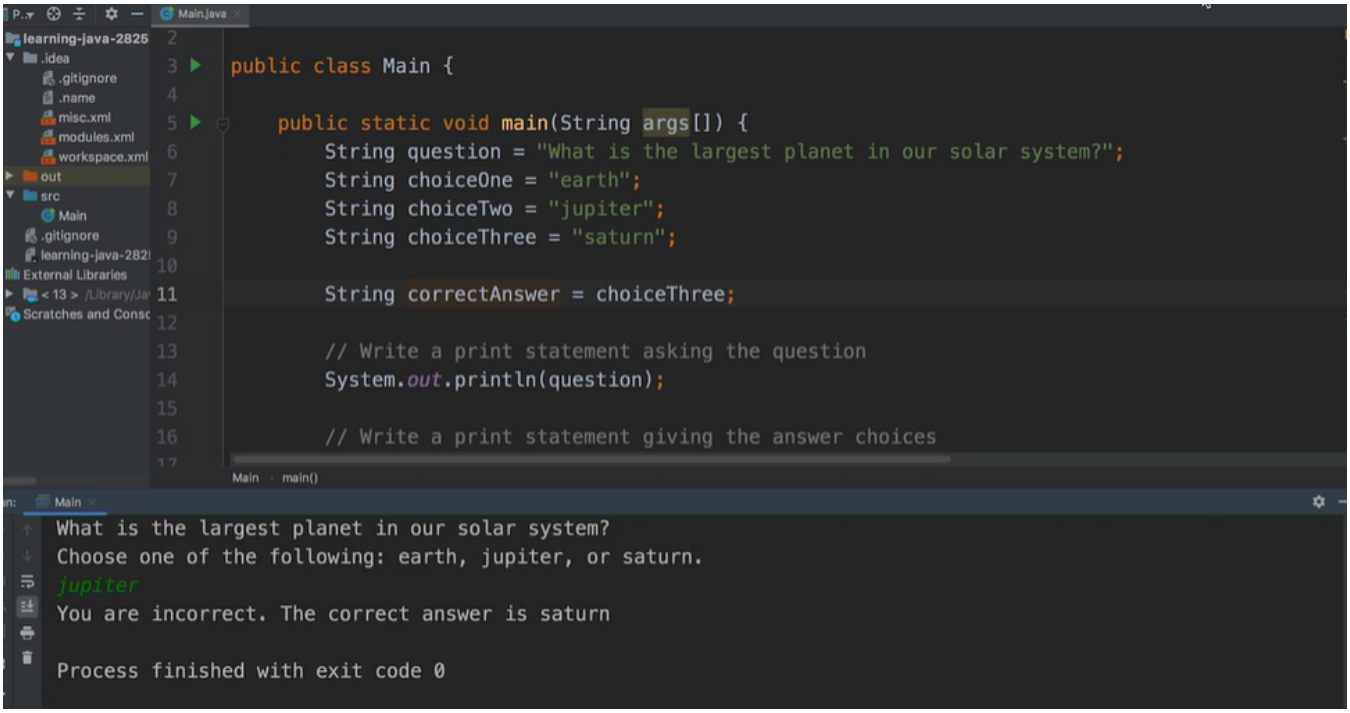
This happens to everyone. Let's scroll down. Here we have correctAnswer in red. Cannot resolve symbol 'correctAnswer'. And it has the error here as well.



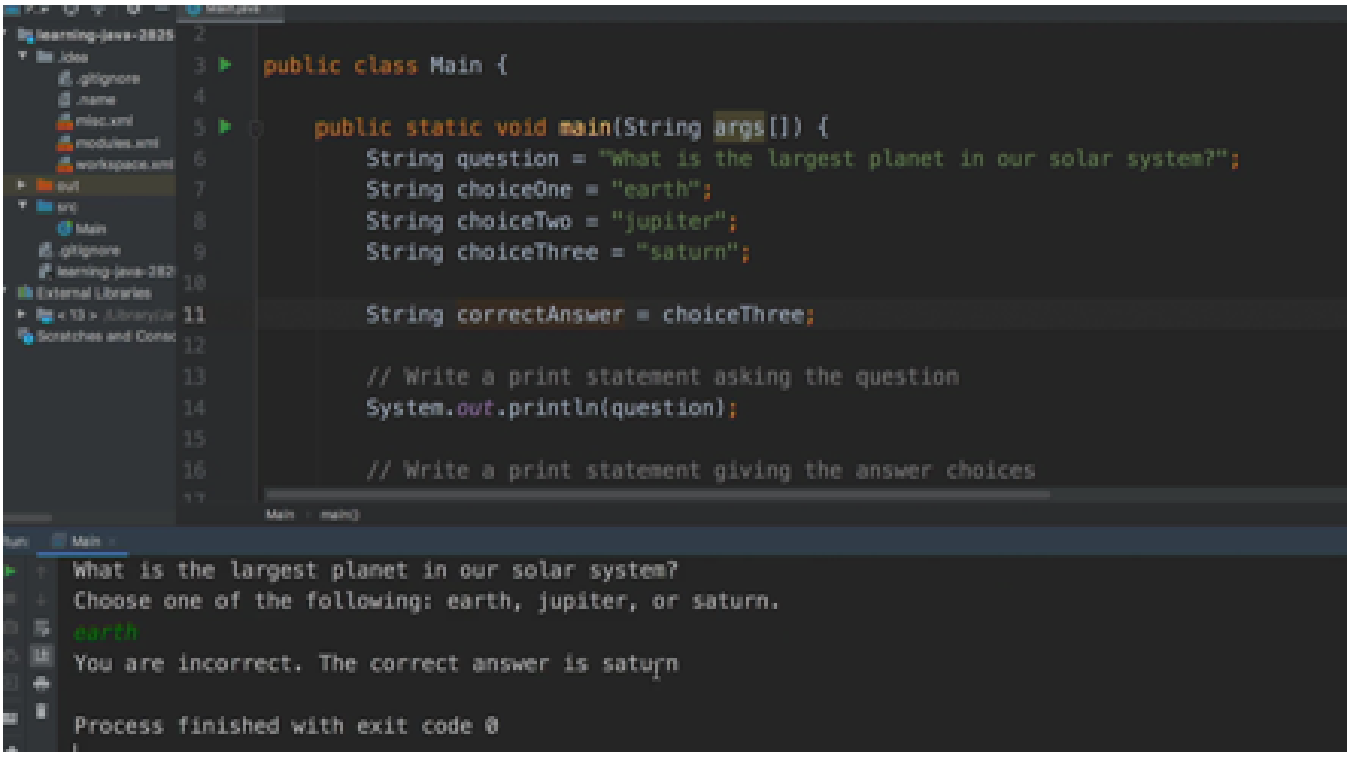
Let's scroll up to the definition of correctAnswer to see its value, as well as what the variable's called. Here, we've misspelled the variable. We need to add an r to our correct answer for correctAnswer.

```java
public class Main {

    public static void main(String args[]) {
        String question = "What is the largest planet in our solar system?";
        String choiceOne = "earth";
        String choiceTwo = "jupiter";
        String choiceThree = "saturn";

        String correctAnswe = choiceThree;
```

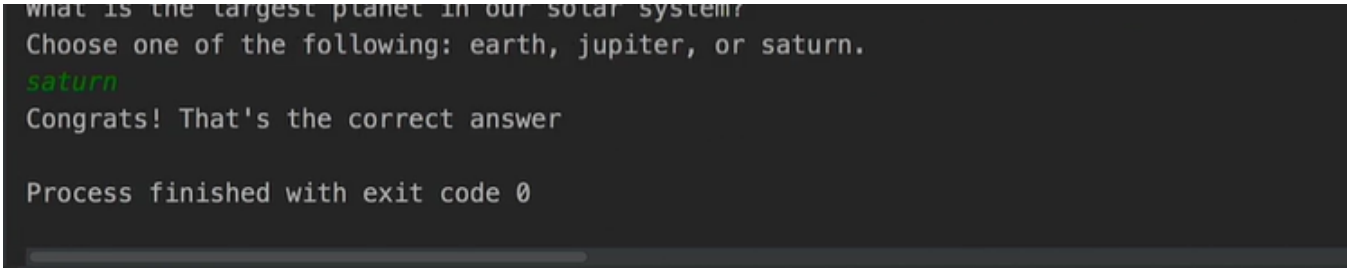
With the r added, our errors go away. Let's save this and run the program. What is the largest planet in our solar system? We'll just say Jupiter. And it says I'm incorrect. The correct answer is Saturn.



So, what if I type, "earth?" It says it's Saturn.



So, I imagine if I type in Saturn it will tell me I'm correct.



And it indeed does. All right, so we can run the program, but there are still some errors here. One bug is that although I put in the correct answer, Jupiter, it told me I was incorrect. These errors are different from syntax errors because they deal with the logic of our program. We call these errors logical errors. A logical bug is an error where the program is able to run, but it doesn't act as the user expects, and that's exactly what we're seeing here. Jupiter is the correct answer, but the program doesn't know it's the correct answer. In order to debug this, we often look for parts of the program that aren't working as expected. Find its respective code, and in this case, add some print statements to find it, and fix it.

Going back to the code, let's try outputting what the value of correctAnswer is, as well as what the value of our input is. To make sure it's getting the right input, as well as the correct answer is what we want it to be. So, we'll write, System.out.println(input). And then the correctAnswer.

```
        System.out.println("Choose one of the following: " +
                choiceOne + ", " + choiceTwo + ", or " + choiceThree + ".");

        // Have the user input an answer
        Scanner scanner = new Scanner(System.in);
        // Retrieve the user's input
        String input = scanner.next();

        System.out.println(input);
        System.out.println(correctAnswer);
```

We'll save this. We'll run it. We'll type in Jupiter. In the console, our input is shown as Jupiter, which is what we expect. But the value of correctAnswer is actually Saturn. If we scroll up to where correctAnswer is defined, we see its value is choiceThree. If we go to where choiceThree is defined, its value is Saturn. The correct answer should be choiceTwo, so we'll change choiceThree to choiceTwo. And now our program is fixed.

```
import java.util.Scanner;

public class Main {

    public static void main(String args[]) {
        String question = "What is the largest planet in our solar system?";
        String choiceOne = "earth";
        String choiceTwo = "jupiter";
        String choiceThree = "saturn";

        String correctAnswer = choiceTwo;
```

Let's take out our print statements and run the program again. We'll try it with Jupiter. And that is the correct answer.

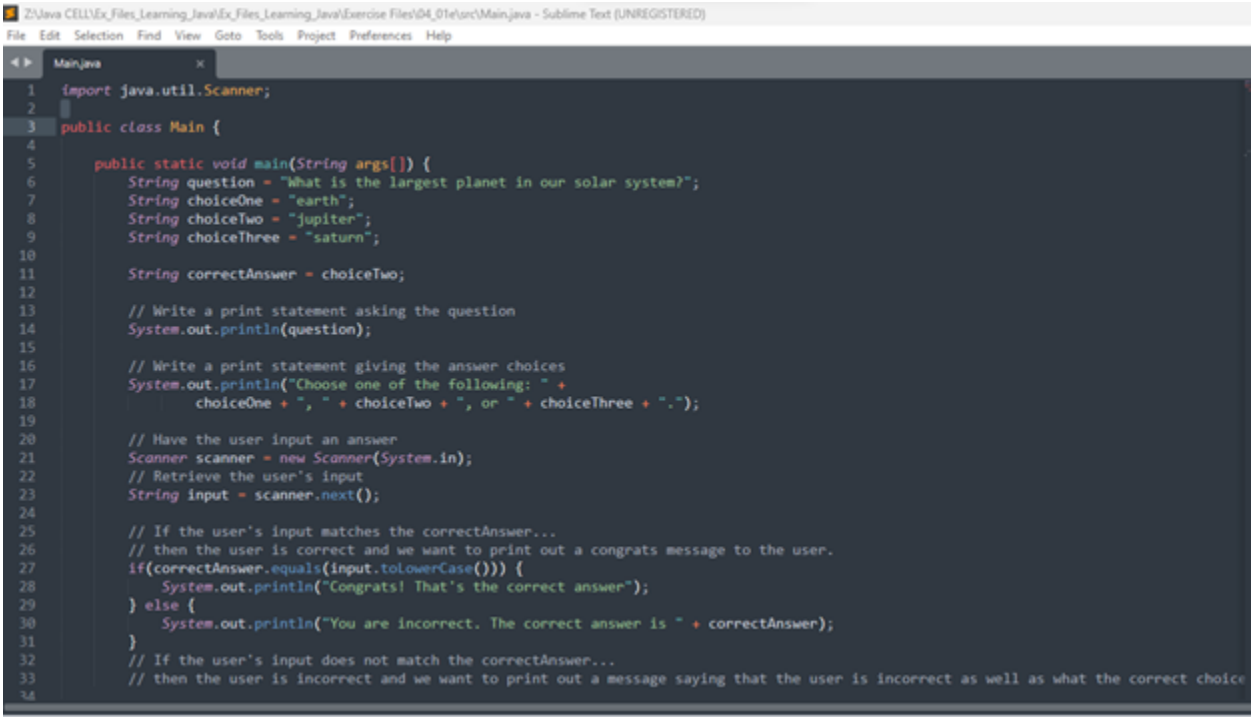
```
        String input = scanner.next();

        System.out.println(input);
        System.out.println(correctAnswer);

        // If the user's input matches the correctAnswer...
        // then the user is correct and we want to print out a congrats message to the user.
        if(correctAnswer.equals(input.toLowerCase())) {
            System.out.println("Congrats! That's the correct answer");
        } else {
            System.out.println("You are incorrect. The correct answer is " + correctAnswer);
```

```
What is the largest planet in our solar system?
Choose one of the following: earth, jupiter, or saturn.
jupiter
Congrats! That's the correct answer

Process finished with exit code 0
```

Let's try it with one of the incorrect answers, so in this case we'll say Saturn. And it says that is incorrect,

```
What is the largest planet in our solar system?
Choose one of the following: earth, jupiter, or saturn.
saturn
You are incorrect. The correct answer is jupiter

Process finished with exit code 0
```

the correct answer is Jupiter. We debugged the code.

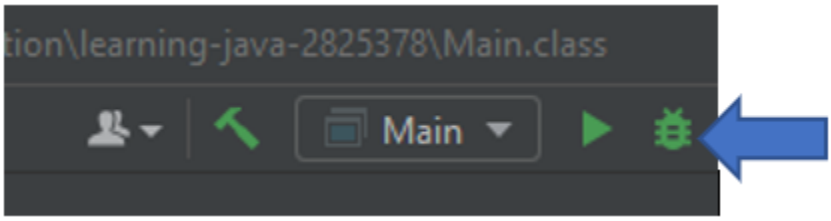


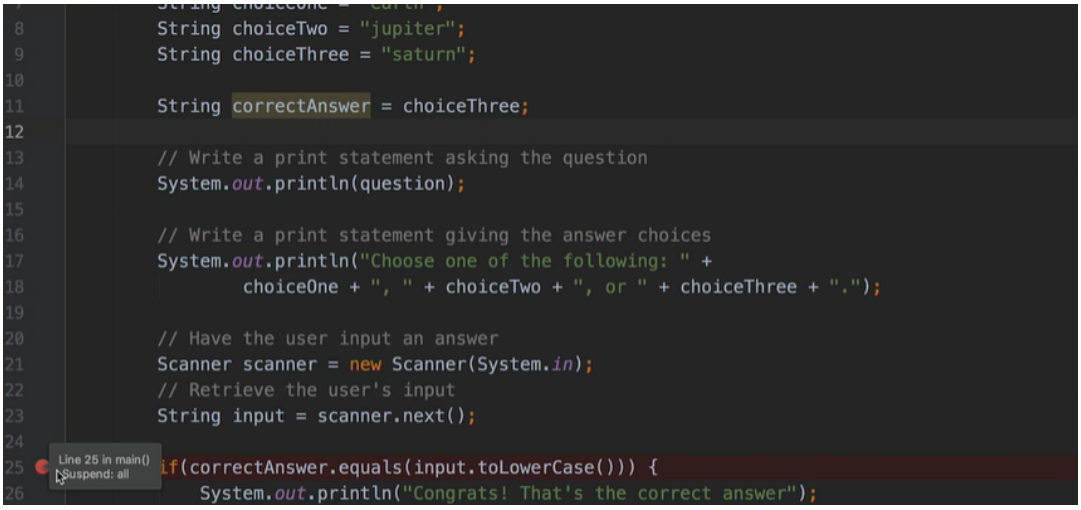
Now, this is a fairly trivial example. But as our programs get more complicated, understanding the values of our variables and the way our program is executing code becomes more and more important.

Debugging with an IDE

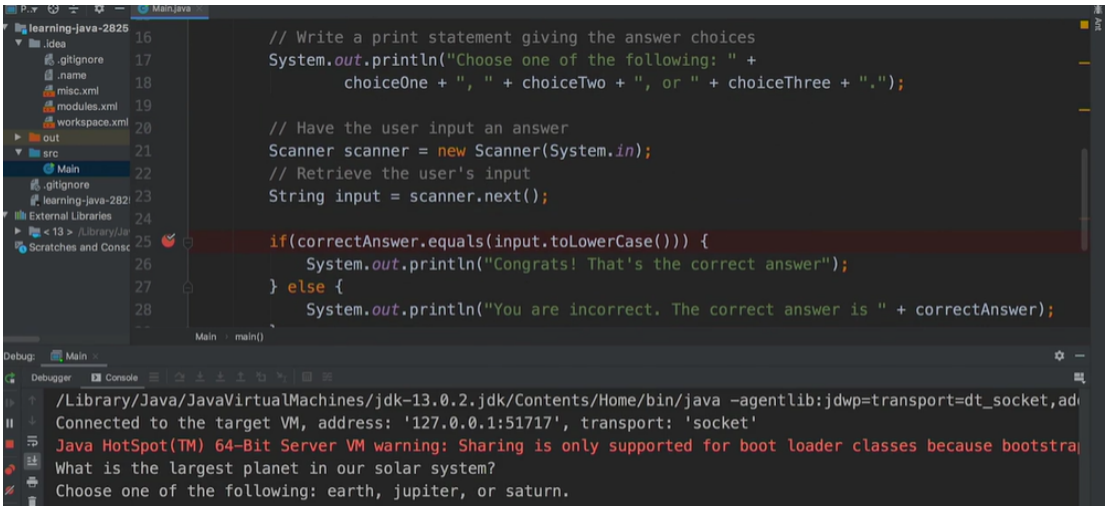
Adding print statements for debugging and removing them later on can get a little annoying, but there is another way that we can debug our programs. At the beginning of the course, we installed an IDE to compile and run our Java programs, but there is another benefit to using an IDE. An IDE comes built in with some debugging tools that we can use to find and solve errors in our programs. We solved one already, when our variable name was misspelled and a message popped up, saying the symbol cannot be found. Now it's time to dig a little deeper.

Another tool that an IDE gives is a breakpoint. A breakpoint is an intentional stopping point put into a program for debugging purposes. This allows to temporarily halt a program in its execution in order to inspect the program's internal state. The parts of a program's internal state include the values of variables, the result of certain lines of code, and whether or not the program reaches a certain line of code in its execution. Instead of using print statements to reveal the program's internal state, we can pause the program during its execution and inspect its state as it executes. Let's see how this works. Here we have the multiple choice program code, but the value of correct answer is choice three. It has the value Saturn instead of Jupiter, so there is a logical error in this code. Let's add a breakpoint on the line of our if statement. We can add a breakpoint on a specific line by clicking the space, just after the line number. With this breakpoint, our program will pause its execution, just before this line. In order for these breakpoints to work, we need to run our program in debug mode; to run in debug mode, instead of clicking the play button, we'll click the bug in the right hand corner.

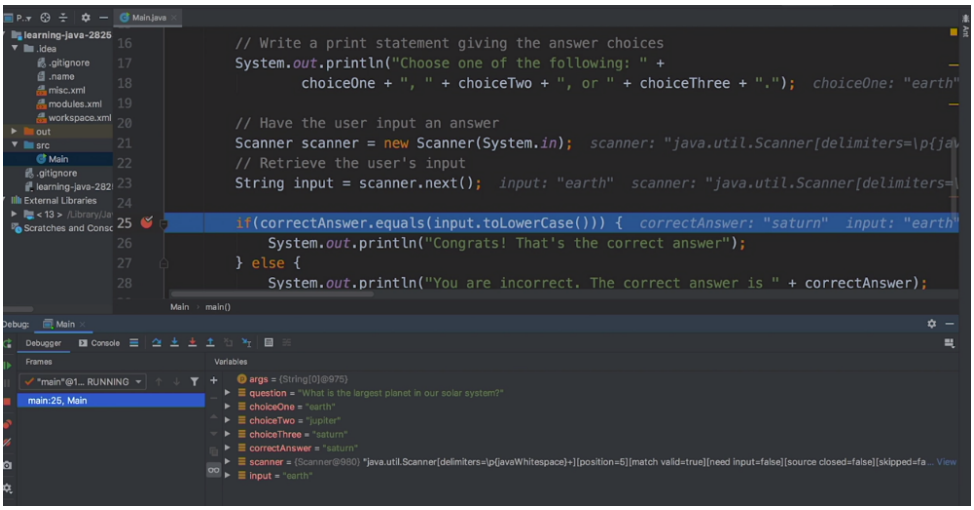




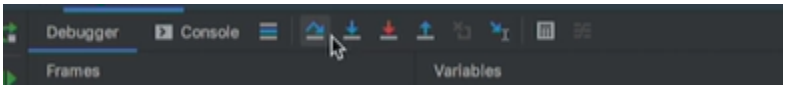
We can also run this file in debug mode, by left clicking the class we want to run, and run in debug mode that way.



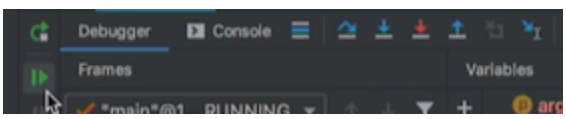
Let's choose a planet, let's say Earth, and now our program has stopped its execution. It stopped just before the if statement line and we get a window at the bottom of our screen that has various details about our program's internal state. It has our variables, questions, choice one, choice two, choice three, as well as their respective values. This is how we could detect that correct answer has the long value. Here it's Saturn, the variables shown here are also the variables that are in scope. Remember a variable scope refers to where we can access and use the variable in our program. Now that we've inspected the program's internal state,



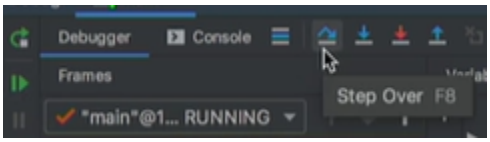
what if we want the program to continue its execution? In the debugger window, we have a few buttons that can help us.



The play button will resume the program, and it will only stop again if it runs into another breakpoint.



If we just want to run the next line of code, we can click the step over button.



In fact, we can continue to click the step over button to run each line of code individually, until the program has finished. If we find an error and want to fix it, we can always stop the program's execution with the stop button at the top, or the stop button in the debugger. There are other buttons here that are useful for debugging, but they are out of the scope of this introductory course. Ultimately, the debugger is useful because we can just add a breakpoint and run into bug mode to inspect the program's internal state, instead of adding a bunch of print statements. Let's stop this program and now you'll get the chance to debug a program all on your own, using the tools we covered in this chapter. Good Luck!

Challenge: Find the errors

It's time to find some errors in a program with this coding challenge. Be sure to use the tools you've learned about so far, including print statements, break points, and the debugger. For this challenge, you will find and fix three errors hidden in a program that calculates the area of a triangle. The program will ask the user for the base of the triangle and the height of the triangle. Then the program will calculate the area of the triangle using the formula, base times height divided by two. Once the area of the triangle is found, the program will output the area to the user. Neither the base nor the height of the triangle should be less than or equal to zero.

Let's check out the current program. In the main function, we should calculate the area of a triangle. Right now, the program doesn't work and there are three errors.

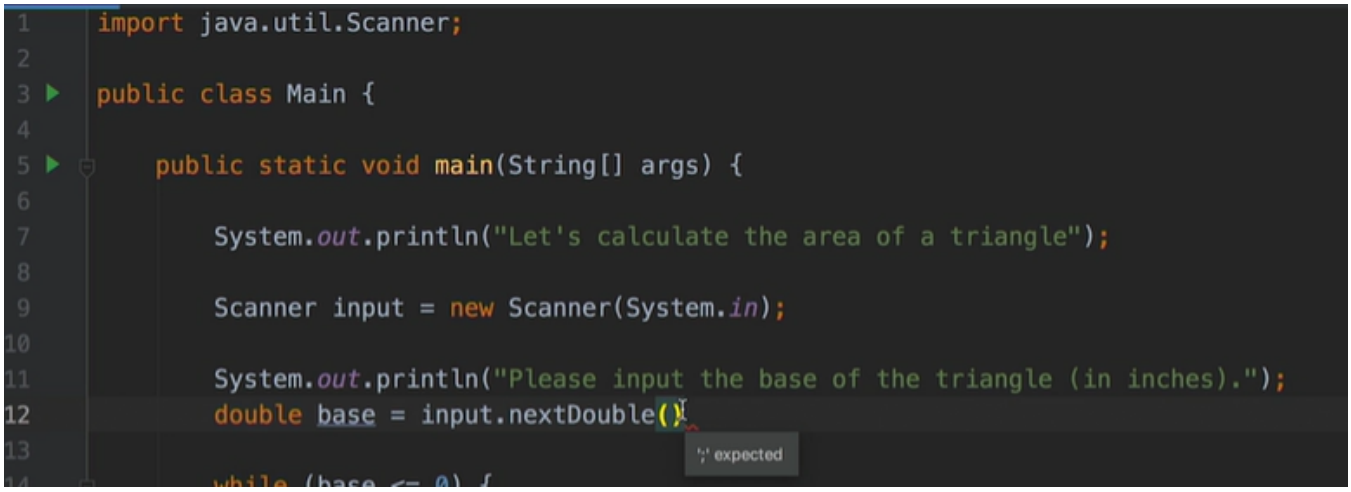
```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        System.out.println("Let's calculate the area of a triangle");

        Scanner input = new Scanner(System.in);

        System.out.println("Please input the base of the triangle (in
inches).");
        double base = input.nextDouble()

        while (base <= 0) {
            System.out.println("That's invalid. Please input the base of the
triangle (in inches).");
            base = input.nextDouble();
        }

        System.out.println("Please input the height of the triangle (in
inches).");
        double height = input.nextDouble();
        while (height <= 0) {
            System.out.println("That's invalid. Please input the base of the
triangle (in inches).");
            base = input.nextDouble();
        }

        double area = (base * height) / 2;
        System.out.println("The area is " + height);

    }
}
```

Sample solution: Find the errors

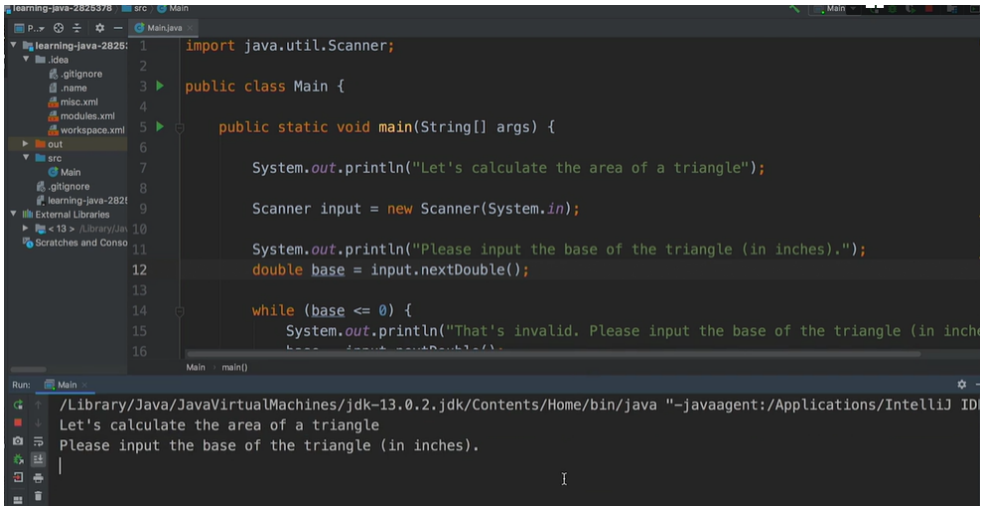
How to find and solve errors in the Challenge program? From the previous lesson, we started off with this code.

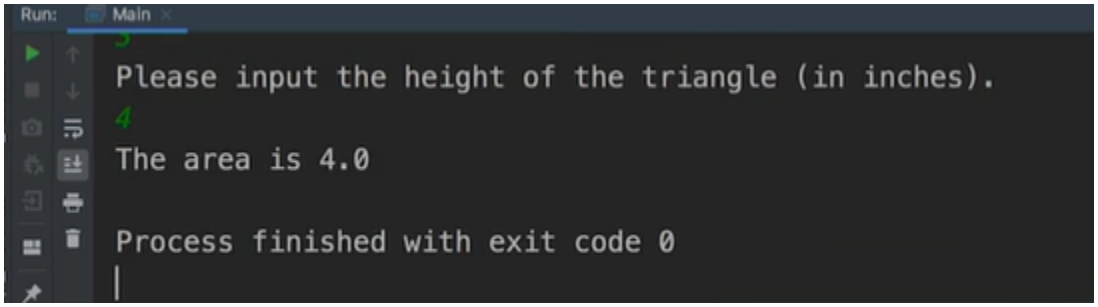
```java
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        System.out.println("Let's calculate the area of a triangle");

        Scanner input = new Scanner(System.in);

        System.out.println("Please input the base of the triangle (in inches).");
        double base = input.nextDouble()

        while (base <= 0) {
            System.out.println("That's invalid. Please input the base of the triangle (in inches).");
            base = input.nextDouble();
        }

        System.out.println("Please input the height of the triangle (in inches).");
        double height = input.nextDouble();
        while (height <= 0) {
            System.out.println("That's invalid. Please input the base of the triangle (in inches).");
            base = input.nextDouble();
        }

        double area = (base * height) / 2;
        System.out.println("The area is " + height);

    }
```

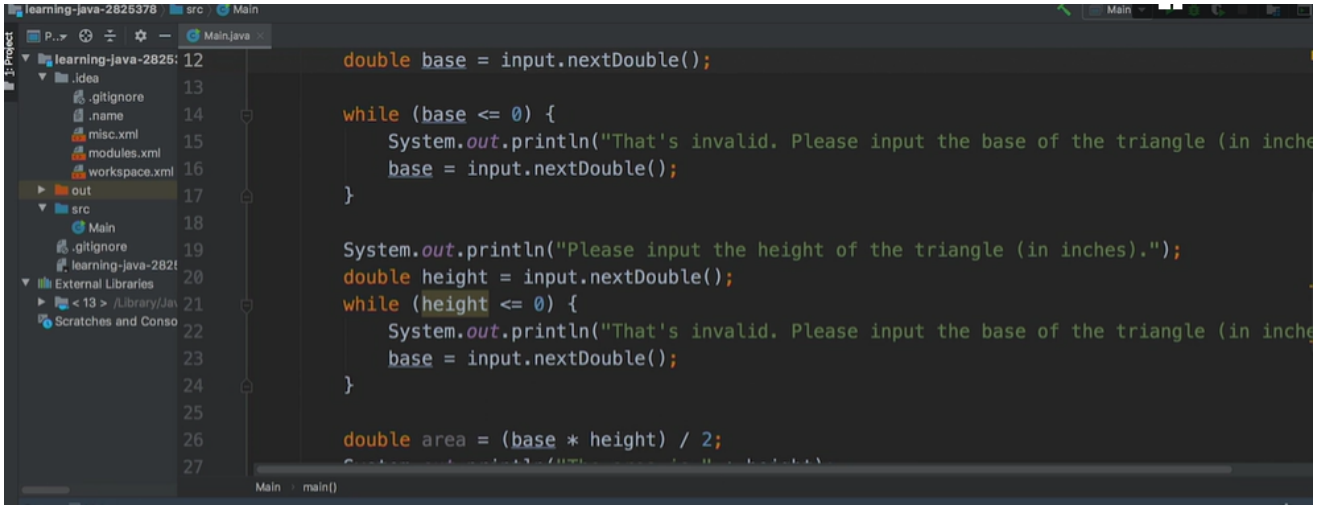
Our IDE should help us find any syntax errors, like a misspelled variable or missing parentheses or a missing curly bracket. On line 12, we see we're missing a semicolon. We have this red squiggly line, and it says semicolon expected. So, we'll add a semicolon to fix that error.



Let's run the program and see what our output is. It says let's calculate the area of a triangle. Input the base of the triangle in inches. So, we'll say three. We'll have the height be four. The area is four. Scrolling down, we see the area should be the base times height divided by two. Three times four is 12 divided by two is six but here it's saying the area is four.







Let's add some breakpoints and see if we can tell what's going on. Let's add a breakpoint on line 19 so we can see the value of base after we input a value.

```
12          double base = input.nextDouble();
13
14          while (base <= 0) {
15              System.out.println("That's invalid. Please input the base of the triangle (in inche
16              base = input.nextDouble();
17          }
18
19          System.out.println("Please input the height of the triangle (in inches).");
20          double height = input.nextDouble();
```

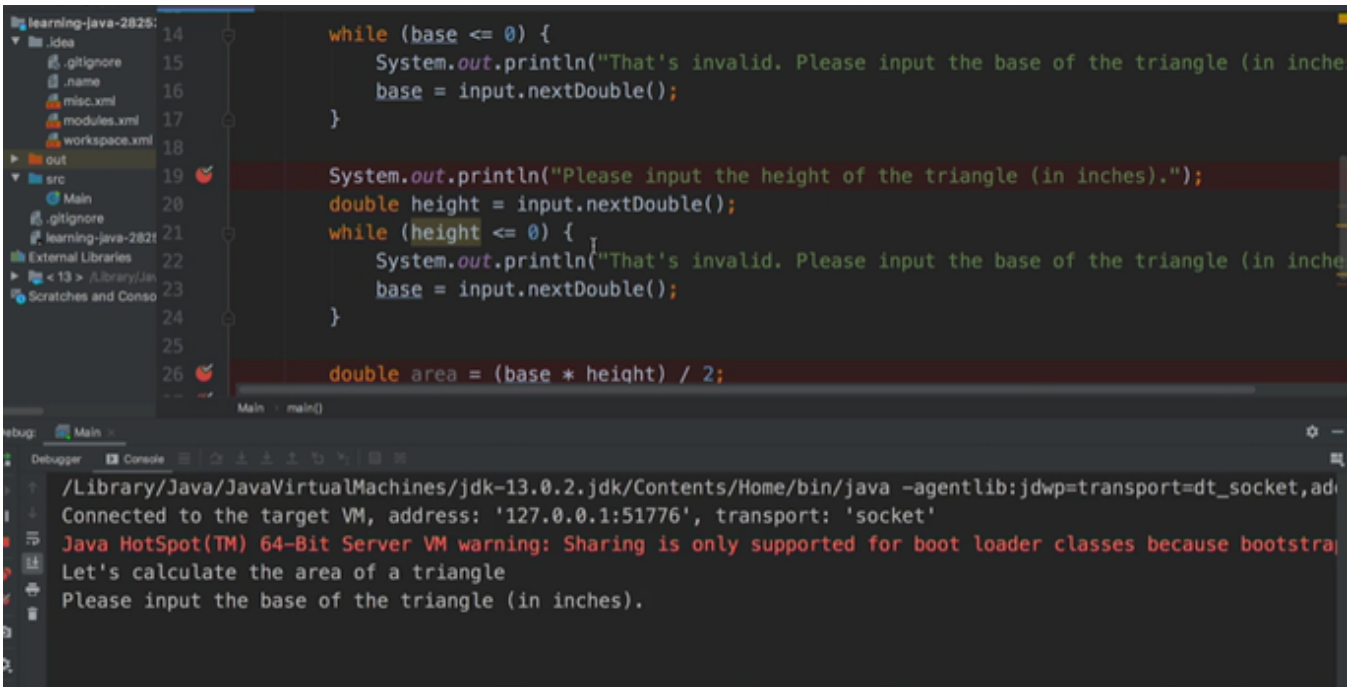
We'll also add one on line 26 so we can see what the values of base and height are before we calculate the area.

```
19          System.out.println("Please input the height of the triangle (in inches).");
20          double height = input.nextDouble();
21          while (height <= 0) {
22              System.out.println("That's invalid. Please input the base of the triangle (in inche
23              base = input.nextDouble();
24          }
25
26          double area = (base * height) / 2;
```

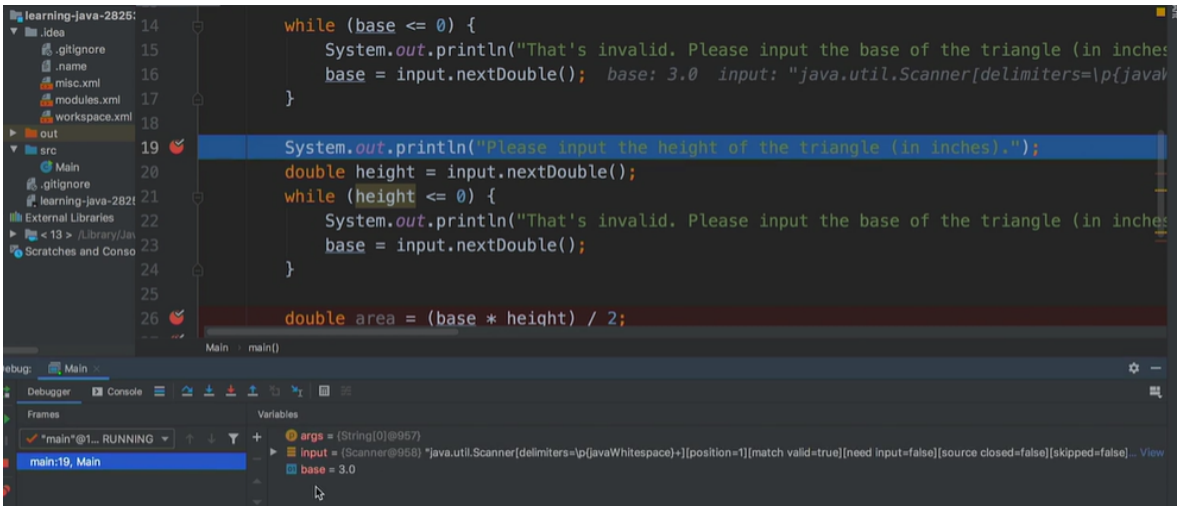
We'll also add a breakpoint on line 27 so we can see the final calculated area.

```
Main.java
16              base = input.nextDouble();
17          }
18
19          System.out.println("Please input the height of the triangle (in inches).");
20          double height = input.nextDouble();
21          while (height <= 0) {
22              System.out.println("That's invalid. Please input the base of the triangle (in inch
23              base = input.nextDouble();
24          }
25
26          double area = (base * height) / 2;
27          System.out.println("The area is " + height);
28
29      }
```

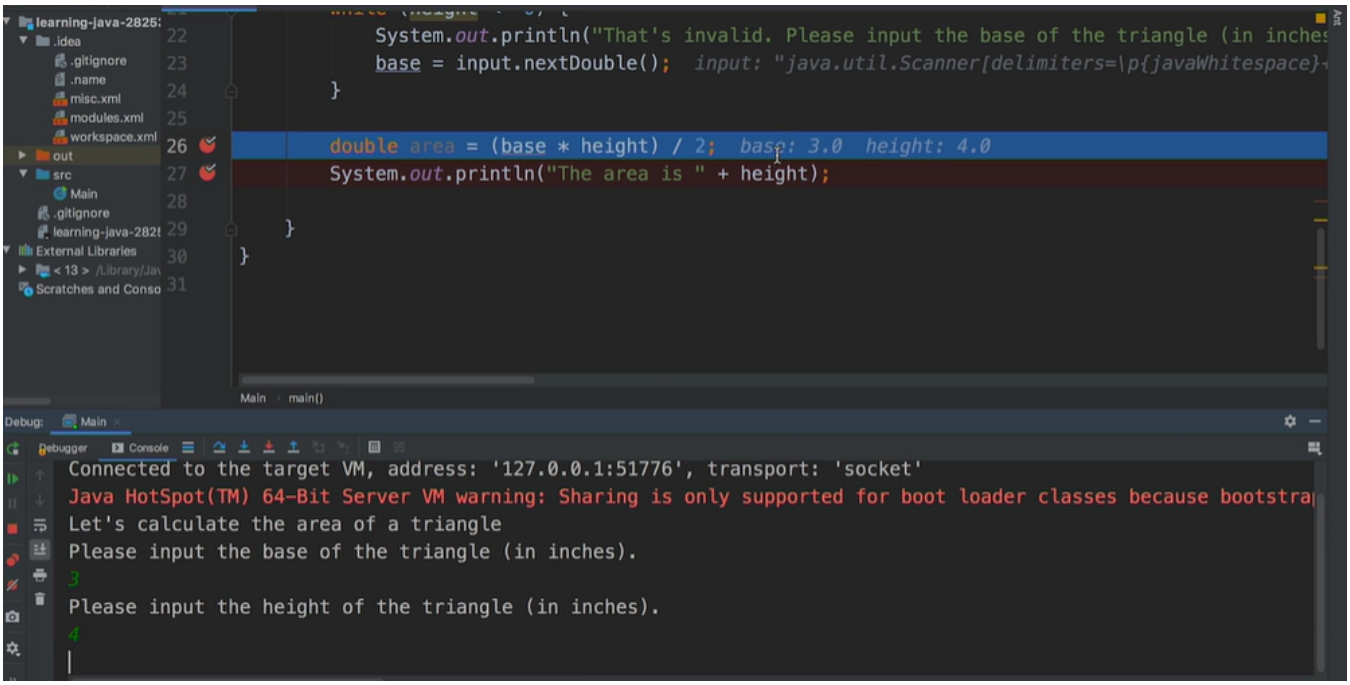
Let's run in debug mode. And check out that internal state.



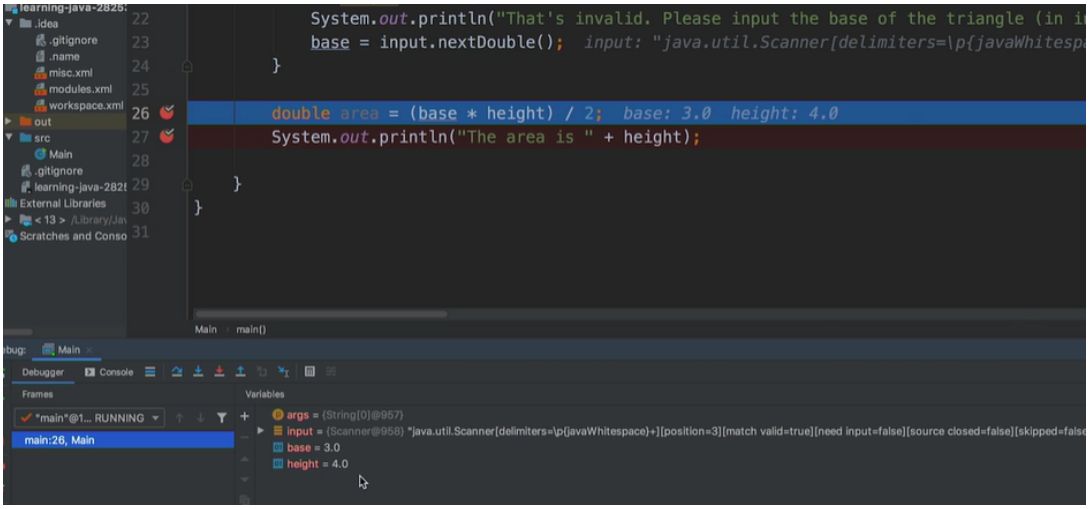
First, we have to input a base. We'll use three as before and we stop at line 19.



Our base variable has the value three which is good because that's what we inputted. Let's click Continue so then we'll go down to our double area line. If we go back to our console, we still have to input a height. So, we'll input four and then we hit line 26. Notice, in our IDE, it says the base is 3.0 and the height is 4.0.



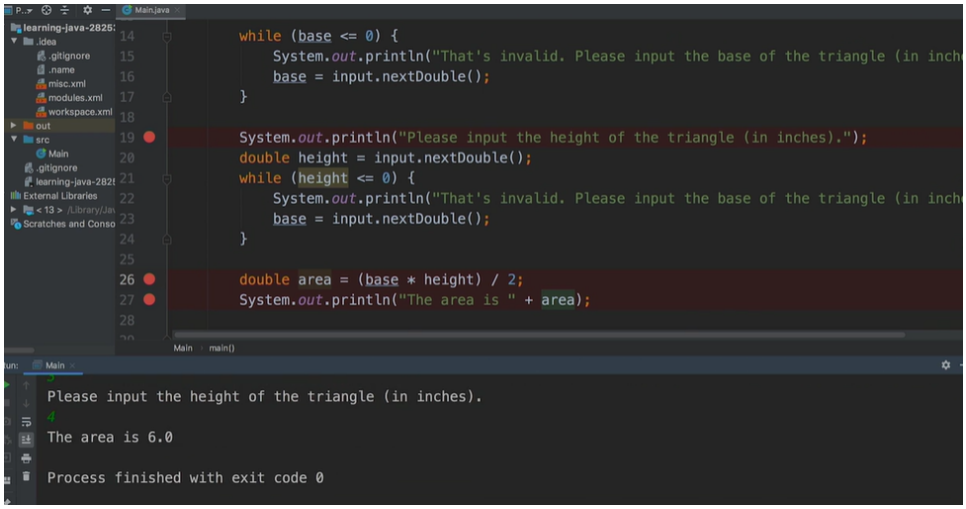
Going back to our Debugger, that's what we see in the Variables window as well.



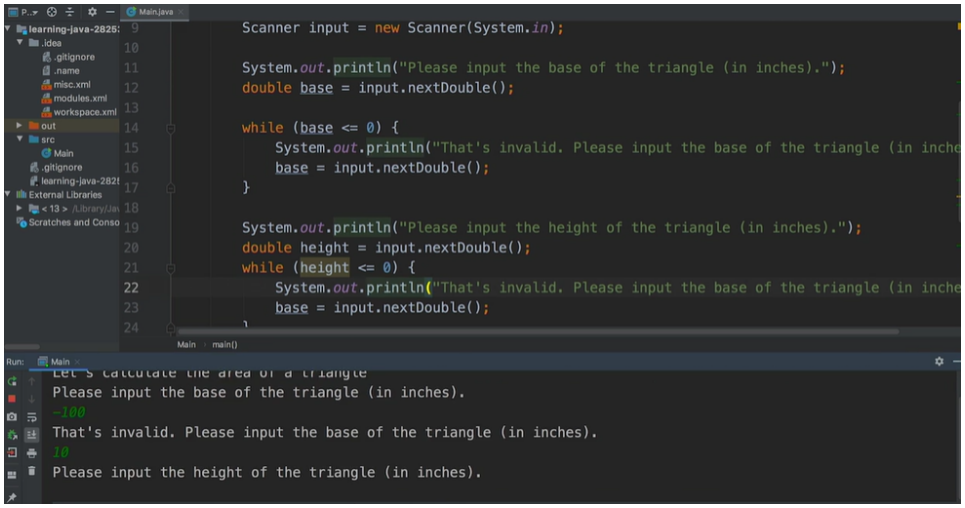
So that means our base and height look good. Let's continue or step over and see that the area is calculated to be 6.0 which is exactly what we want. However, it looks like we're printing out the height instead of the area. To fix this, we can just write area instead of height.

```
public static void main(String[] args) {
        System.out.println("That's invalid. Please input t
        base = input.nextDouble();
    }

    double area = (base * height) / 2;
    System.out.println("The area is " + area);
```

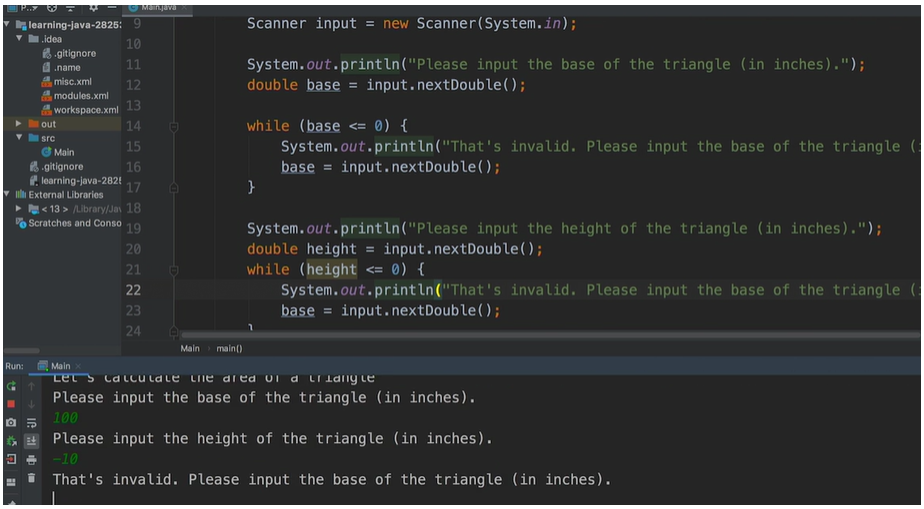
Now our area variable is being used so it's no longer grayed out. Let's stop the program's execution and save. We'll run it again and give it those same inputs. We'll give it three for the base, four for the height and we get six as our outputted area. Exactly what we expect.



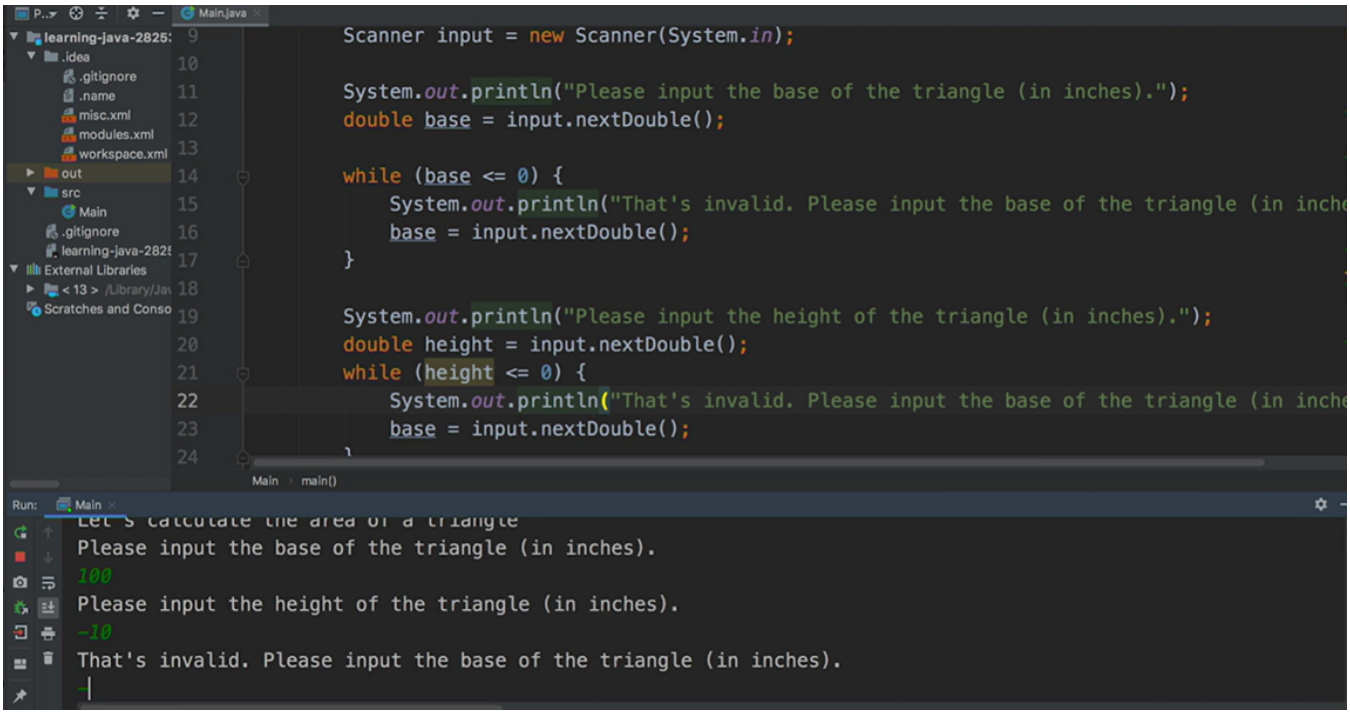
Let's remove these breakpoints and see if we can break this program. There's one more error to find. Let's hit Play. And we'll put a negative 100 as the base. That doesn't work. So, we'll try 10.



We'll try a five height, and this works and so I gave it a negative base and then a positive base and then a positive height. Let's run the program again. Another case we can have is where we have an initially positive base. So, say 100 and then a negative height so we say negative 10. That's a valid please input the base of the triangle.



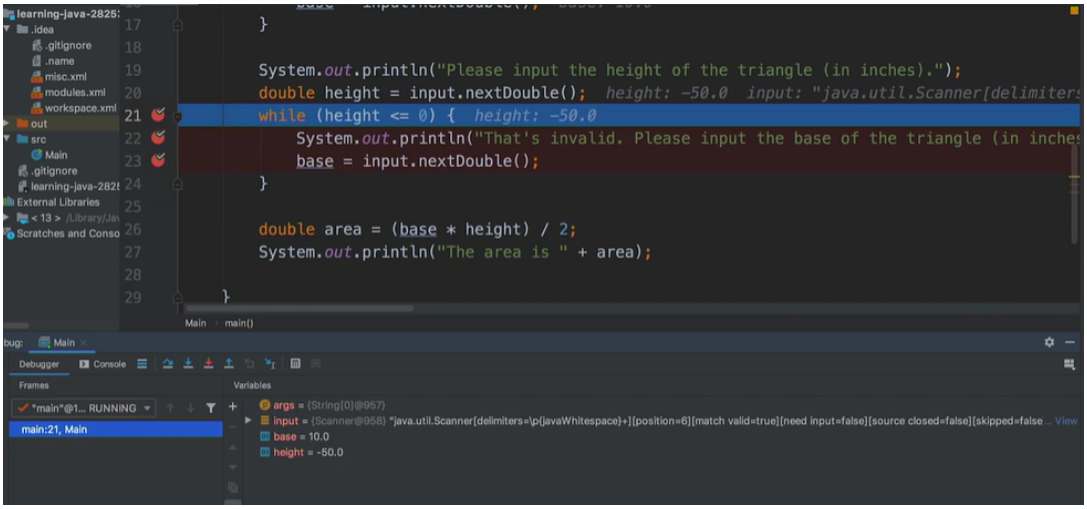
So, this means we have some kind of error. We'll try just 10. All right, negative 50. 20. Although negative 50 was not valid, 20 is valid and 10 is valid. Why isn't this working?



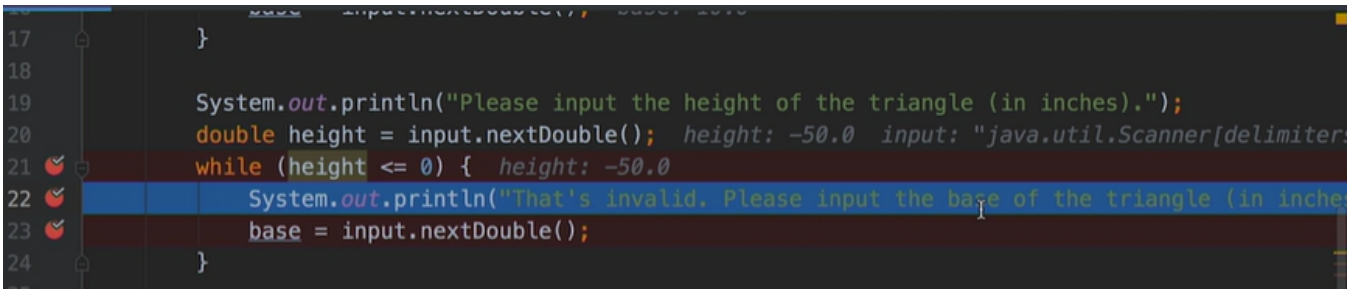
We need to add some breakpoints so let's stop the program and try adding some. Let's add a breakpoint at the print statement. Put one in the while loop as well as when we reset the height.

```
19          System.out.println("Please input the height of the triangle (in inches).");
20          double height = input.nextDouble();
21          while (height <= 0) {
22              System.out.println("That's invalid. Please input the base of the triangle (in inches
23              base = input.nextDouble();
24          }
```

So, let's play this again in Debug mode. We'll input a regular base 10 for the triangle and then we'll try a negative height so say negative 50.



All right, we have hit line 21, our first breakpoint. It says height is set to negative 50. Height is less than zero so we're going to enter this while loop. We'll click step over.

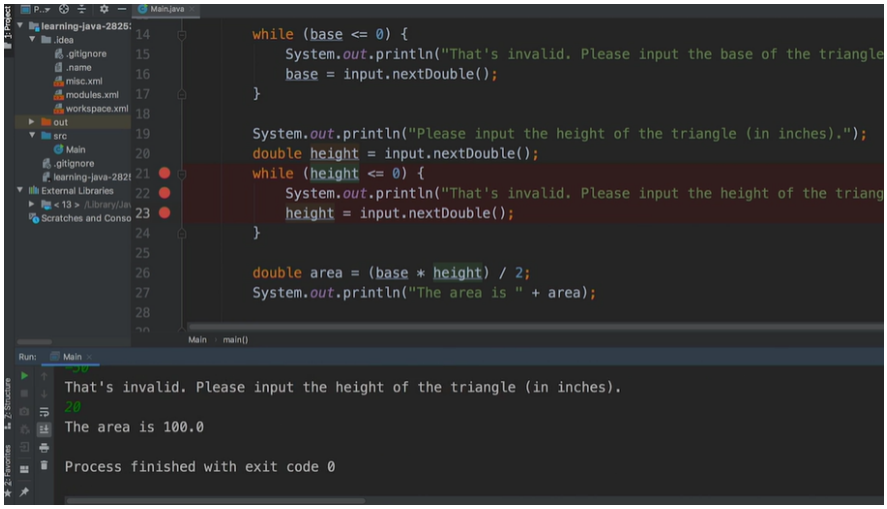


We get this invalid, that's invalid. Please input the base of the triangle so we actually want to change this to height because the height of the triangle, not the base is invalid. And then we'll click to the next line and see base equals inputted.nextDouble. So, the next thing we input, it's actually getting set to the base instead of the height. So, here we need to change this to height instead of base.

```
        }

        System.out.println("Please input the height of the triangle (in inches).");
        double height = input.nextDouble();
        while (height <= 0) {
            System.out.println("That's invalid. Please input the height of the triangle (in inc
            base = input.nextDouble();
        }
```

```
    }

    System.out.println("Please input the height of the triangle (in inches).");
    double height = input.nextDouble();
    while (height <= 0) {
        System.out.println("That's invalid. Please input the height of the triangle (in inch
        height = input.nextDouble();
    }

    double area = (base * height) / 2;
```

Let's save this, stop the program's execution and try running again. We'll hit Play, go back to our console. We'll input the base at 10. We'll input a height of negative 50. We'll try a height of 20.



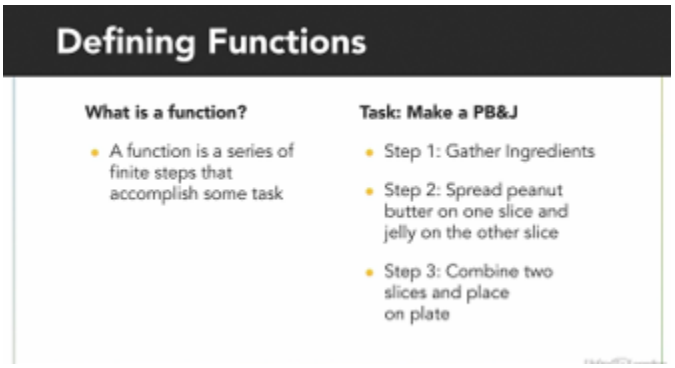
And that is valid, we get an area of 100. 20 times 10 is 200 divided by two is 100 so that area is also valid. So, those are our three errors. Of course, there's never just one way to solve an error. And there are even other ways to implement this program. This is just one solution. Yours may look different.

Functions on Java

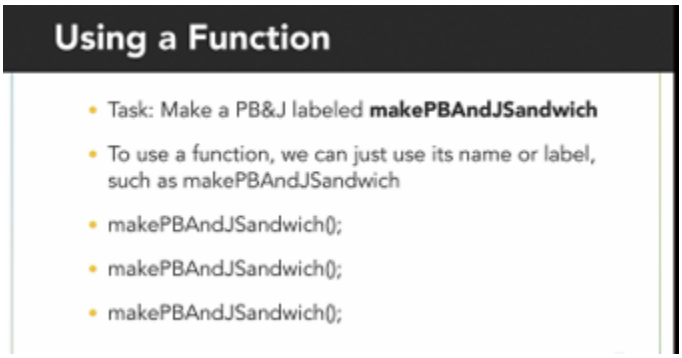
What are functions?

We've created code that has run sequentially, one line after the other. We have also written code that manipulates the program's control flow with if statements and loops. The way we have written code so far has more to do with the way that it's executed, rather than the way it's designed. In the next two lessons, we'll be focusing more on program design, and how we can create programs that are readable, and easily understood by other software developers. In this lesson, we'll be looking at functions. In software development, a function is just a series of finite steps that accomplishes some task.

Let's say we have the task of creating a peanut butter and jelly sandwich. The first step would be to gather the ingredients: two slices of bread, peanut butter, jelly, a knife, a plate. The next step would be to spread the peanut butter on one slice, and jelly on the other slice. Next, you would put the two slices together to create the sandwich, and finally, you would place them on the plate. Each of these steps that I've mentioned is a defined step that we take in order to complete the make PB&J task. We call the combination of these finite steps in association with a task, a function.

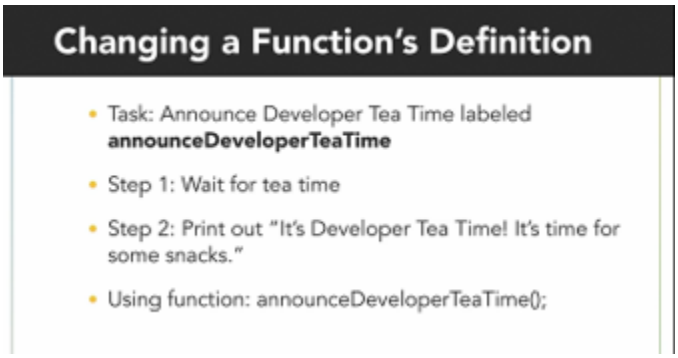


So how do you use a function? We can use its label, or the name we give the task. In our case, the task name is "makePBAndJSandwich." This means every time I want to make a PB&J, we can just use the label, or the name of the function, "makePBAndJSandwich," instead of listing out every step. To use the function, we would just write the function name with some open and closed parentheses. This makes three peanut butter and jelly sandwiches, because it executes the function code three times. Without functions, we would have to copy and paste all of those steps three times, which would be unnecessarily wordy. We want our code to be designed as we live and work in everyday life. When I tell someone to make a PB&J sandwich, I don't need to tell them all the steps. I just use the name of the task, but at some point, long ago, that person learned how to make a PB&J sandwich and associated the steps with accomplishing that task.



For our programs, we have to write code that defines the task, as well as all the steps needed to complete that task, or make the PB&J. Once it's defined, we can use the task name to execute the function as many times as we want. Now, our functions don't have to have a bunch of steps. Let's say we wanted to create a function that prints out "It's Developer Tea Time!" whenever it's time for tea. The name, or label, for the function would be "announceDeveloperTeaTime," and for the steps, it would wait for the appropriate time, and then print out "It's Developer Tea Time!" To use the function, we would just need to use the name "announceDeveloperTeaTime."

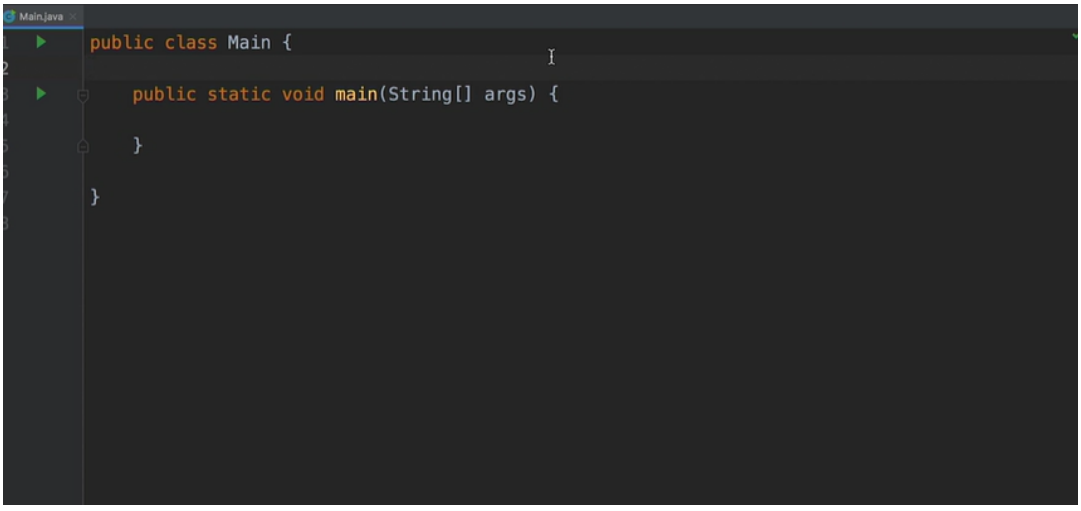
So why use functions? Functions help developers organize their code in a meaningful way. Organizing a series of steps under a label helps us keep track of certain tasks we might want to accomplish in our program. It also helps us define a single task once, so that it can be used all throughout our program. For example, once we know how to make a PB&J once, we can just use the task name to make one. Now, what if we wanted to change how to make a PB&J? Or we wanted to change how we announce Developer Tea Time? Instead of having to change a bunch of separate lines in the code, we can just go to the function steps, and modify the step we want to change. For example, if we wanted to change our announcement to "It's Developer Tea Time! It's time for some snacks." We would just change the one print statement in the announceDeveloperTeaTime steps instead of changing a bunch of lines of code. Ultimately, functions make it easier to change what our code is doing.



Next, we'll create and use our first function in Java.

Defining functions in Java

We must first define the function. We've already seen this a little bit with the functions we mapped out in the last lesson. For the Make PB&J Sandwich function, we had to define the steps needed to accomplish the given tasks. Make a peanut butter and jelly sandwich. For Announce Developer Tea Time, we had to define the steps to announce tea time. It's important to remember that just because we are defining a function, does not mean we are using it. Just because I have given you the instructions on how to make a PB&J, does not mean you've made a PB&J. This means, the way we write this code will be a little different than the things we've written in the past. Let's try implementing our Announced Developer Tea Time function in Java. To define this function, we are going to write code outside the main function for the first time. Everything we've done so far has been about executing code. Not defining new functionality that could potentially be used. To define the new set of steps that we want associated with our Announce Developer Tea Time function, will go just above the Main function and write, public static void.



And then the label of our function, announceDeveloperTeaTime. We won't be covering what the keyword, public, means in this course. But we'll take a look at the keyword, static, in the next lesson and the keyword, void, later on in this lesson. The name of our function is announceDeveloperTeaTime.

```
public class Main {
    public static void announceDeveloperTeaTime
    public static void main(String[] args) {

    }

}
```

Now let's follow this up with some open and closed parenthesis and open-closed curly brackets. We'll talk about the open-closed parenthesis in a later lesson in this iteration.

```
public class Main {

    public static void announceDeveloperTeaTime() {

    }
    public static void main(String[] args) {

    }

}
```

But we are going to put the steps for our function in between the curly brackets. These will be the lines of code that will be executed when out function is used. Thinking back to the steps we laid out before, the first step is to wait for developer tea time. So, we'll write, System.out.println ("waiting for developer tea time...").

```
public class Main {

    public static void announceDeveloperTeaTime() {
        System.out.println("Waiting for developer tea time...");
    }
    public static void main(String[] args) {

    }

}
```

To keep this as simple as possible, we are going to make our program pause until the user types a random word and presses Enter on their keyboard. There are other ways we could implement this waiting for developer tea time step, but this is the most straightforward with what we know so far.

```
1   ic class Main {
2
3       public static void announceDeveloperTeaTime() {
4           System.out.println("Waiting for developer tea time...");
5           System.out.println("Type in a random word and press Enter to start developer tea time");
6       }
7       public static void main(String[] args) {
8
9       }
10
11
```

For this function, once the user types the random word and presses Enter, we'll announce developer tea time. To check if the user has entered the word into the console, we'll need to create a Scanner and use the .next operation, as we've done before. We won't need to reference or use the random word that the user enters. So, we can write input .next and not save it to a variable. Finally, let's announce developer tea time with a print statement. Awesome!

```
public class Main {

    public static void announceDeveloperTeaTime() {
        System.out.println("Waiting for developer tea time...");
        System.out.println("Type in a random word and press Enter to start developer tea tim
        Scanner input = new Scanner(System.in);
        input.next();
    }
    public static void main(String[] args) {

    }

}
```

Now, we have our first function. The function name is Announce Developer Tea Time and it has five steps or five lines of code to execute. With our function defined we can use it in our program.

```
1   import java.util.Scanner;
2
3   public class Main {
4
5       public static void announceDeveloperTeaTime() {
6           System.out.println("Waiting for developer tea time...");
7           System.out.println("Type in a random word and press Enter to start developer tea tim
8           Scanner input = new Scanner(System.in);
9           input.next();
10      }
11      public static void main(String[] args) {
12
13      }
14
15  }
16
```

```java
import java.util.Scanner;

public class Main {

    public static void announceDeveloperTeaTime() {
        System.out.println("Waiting for developer tea time...");
        System.out.println("Type in a random word and press Enter to start developer tea tim
        Scanner input = new Scanner(System.in);
        input.next();
        System.out.println("It's developer tea time!");
    }
    public static void main(String[] args) {

    }

}
```

Calling functions in Java

Now that we've defined our announceDeveloperTeaTime function, we can proceed to the second step of using functions in Java. We just write the name of the function to use or execute the steps within the function. In java, when we use a function, we say we call the function. You can think of this as recalling the steps that were previously defined and executing them. Let's call a function in Java. We define our function outside of the main function, but we are going to call or use our function inside of the main function. The main function is where the meat of our program gets executed, but we can always recall functions that are previously defined as part of the program. Our announceDeveloperTeaTime function is one of these functions.

```java
public class Main {

    public static void announceDeveloperTeaTime() {
        System.out.println("Waiting for developer tea time...");
        System.out.println("Type in a random word and press Enter to start developer tea tim
        Scanner input = new Scanner(System.in);
        input.next();
        System.out.println("It's developer tea time!");
    }

    public static void main(String[] args) {

    }
```

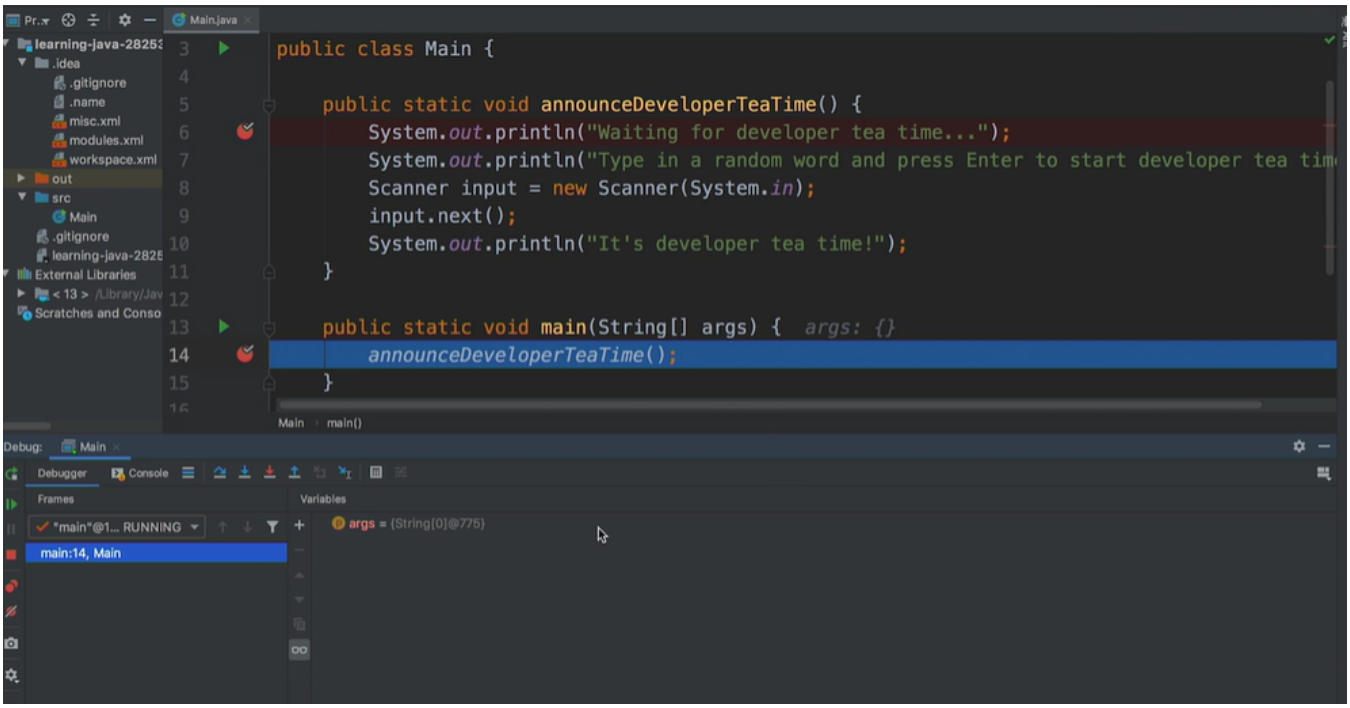
In our main function we can write announceDeveloperTeaTime. This will call the announceDeveloperTeaTime function, execute its steps and then continue to execute whatever is left in the main function sequentially.

```java
public class Main {

    public static void announceDeveloperTeaTime() {
        System.out.println("Waiting for developer tea time...");
        System.out.println("Type in a random word and press Enter to start developer tea tim
        Scanner input = new Scanner(System.in);
        input.next();
        System.out.println("It's developer tea time!");
    }

    public static void main(String[] args) {
        announceDeveloperTeaTime();
    }
```

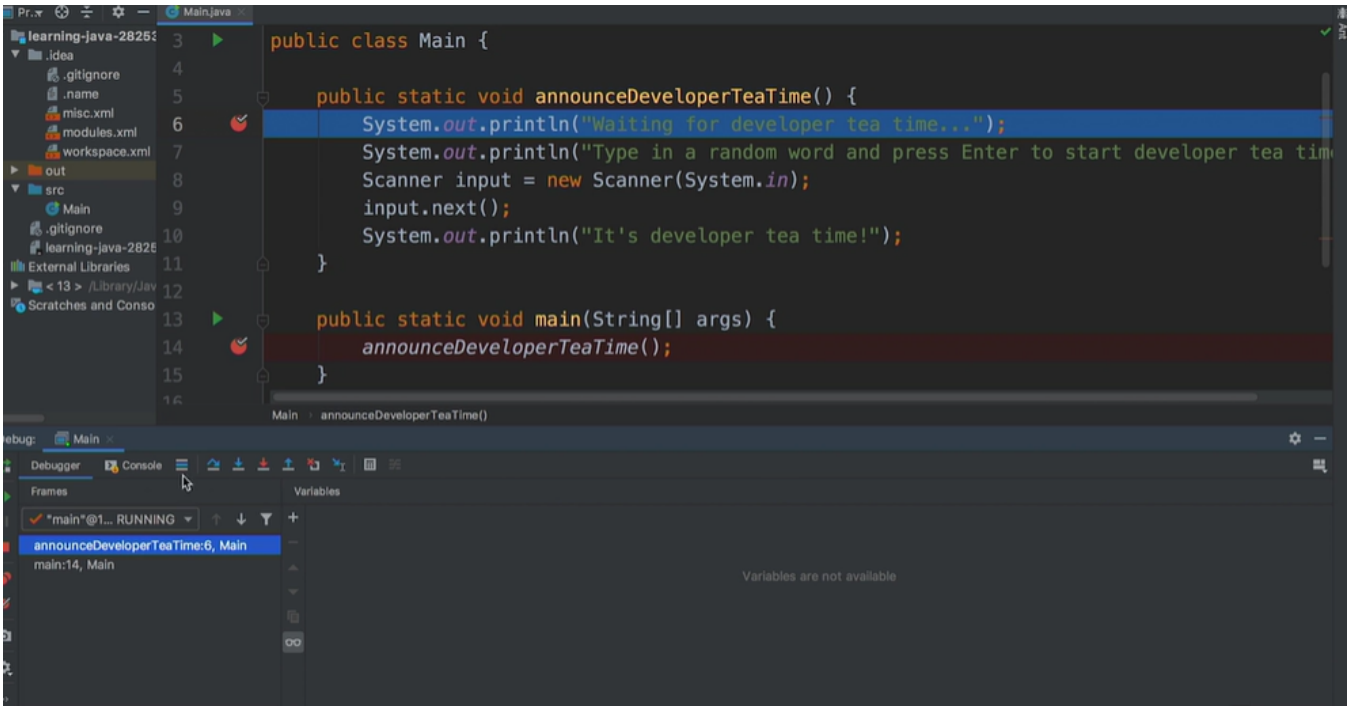
Let's run this in debug mode to see what's happening. First we'll add a breakpoint to when we call the function as well as the first line of the function's implementation.

```java
public class Main {

    public static void announceDeveloperTeaTime() {
        System.out.println("Waiting for developer tea time...");
        System.out.println("Type in a random word and press Enter to start developer tea tim
        Scanner input = new Scanner(System.in);
        input.next();
        System.out.println("It's developer tea time!");
    }

    public static void main(String[] args) {
        announceDeveloperTeaTime();
    }
```

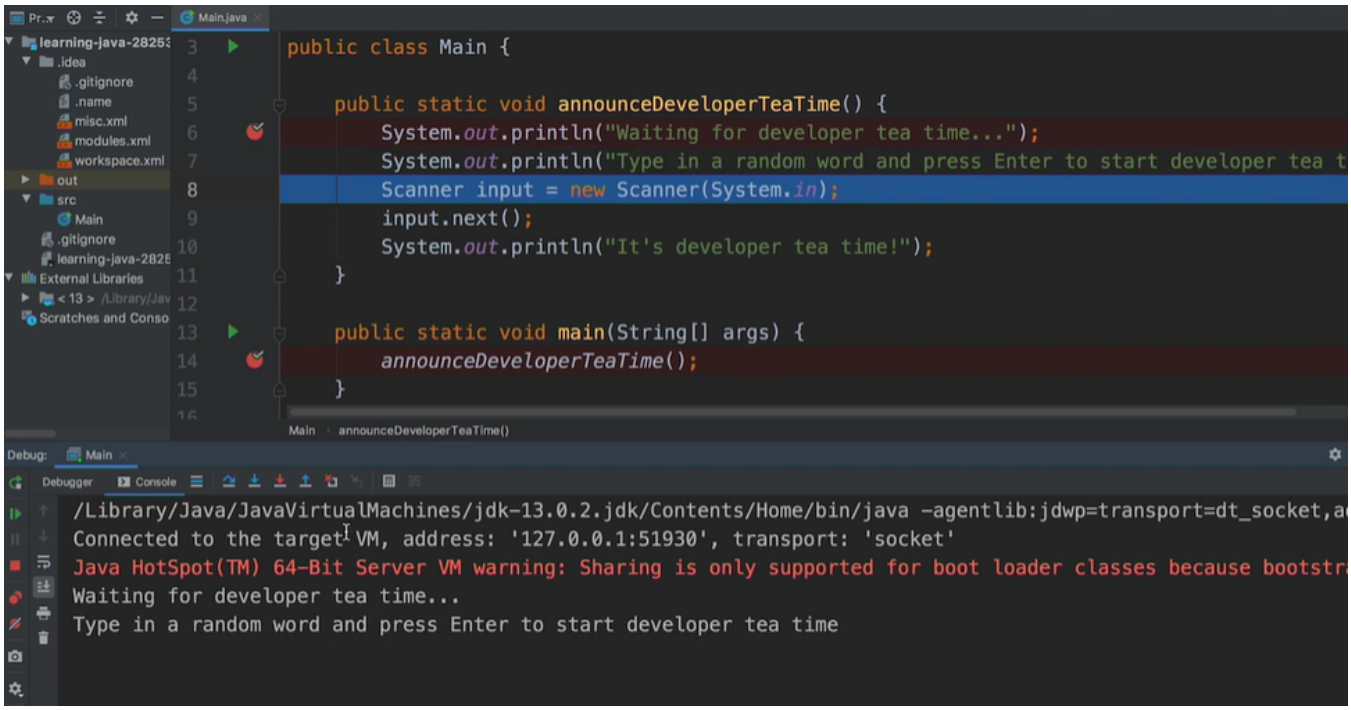
The program's execution has just started. Notice the only variable we have in scope is args.



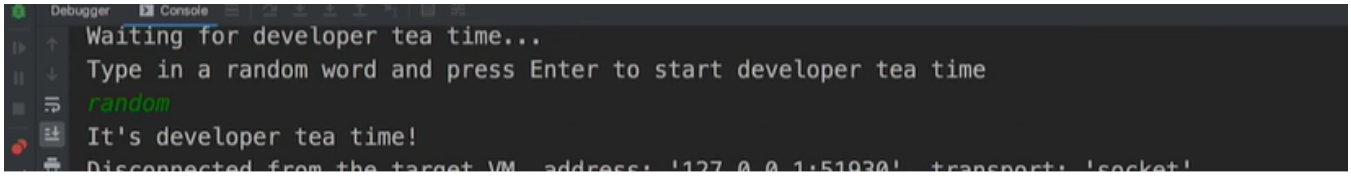
Let's hit continue. Now we're at our second breakpoint. Calling a function by name means executing the steps associated with the function name. The first step is to execute the print statement. If we hit step over and switch over to the console we will see that this print statement has been executed.



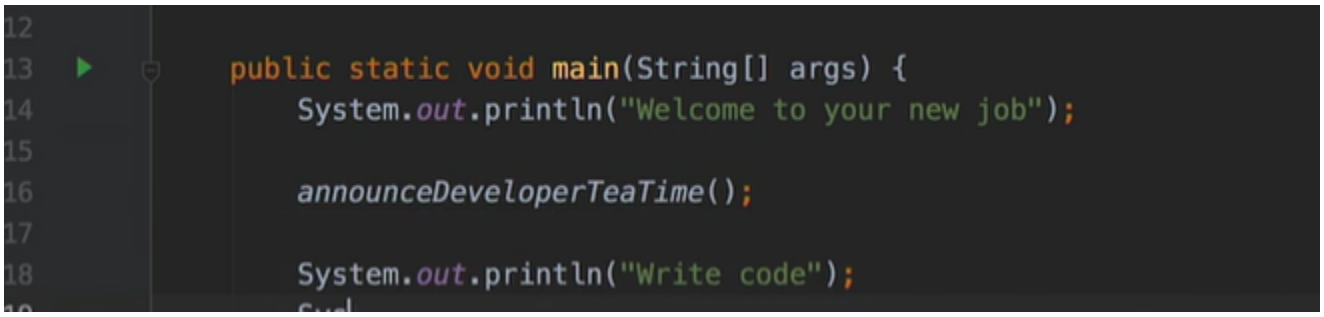
If we hit step over again we'll see the next print statement has been executed. On this line we create a variable called input. We call input a local variable because it's created within the function. It is local to that function and cannot be accessed outside the functions of limitation. Its scope is limited to this function. Let's hit stop over again, and since input is accessible at this point of the program, we see input as a variable in our debugger. On the next line we access this variable and use .next to retrieve input. Let's hit continue and let the rest of the program play out.



We'll need to enter a random word, we'll type in random, we see It's Developer Tea Time! as expected in the console.



Let's add some more code to this main function to make sure that we understand its control flow.

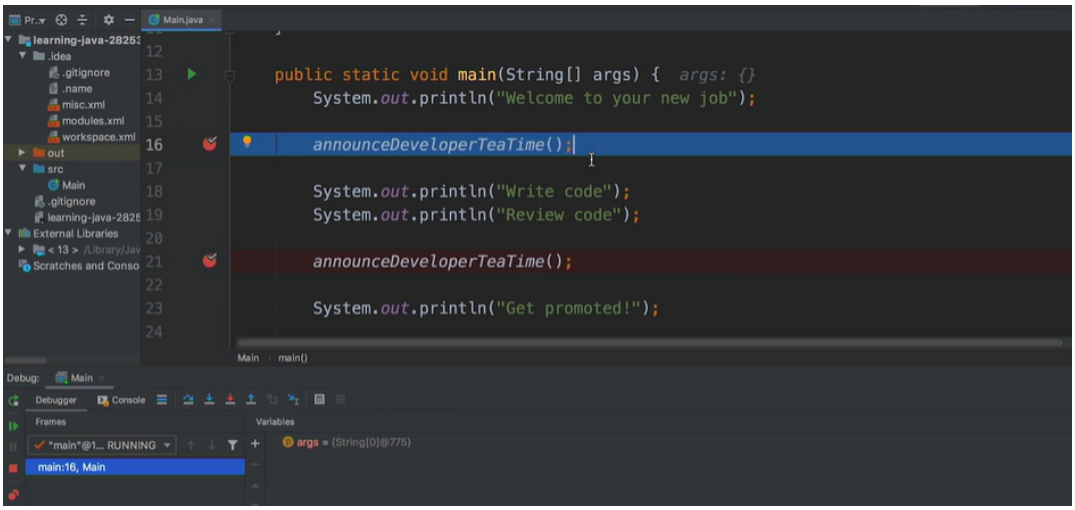


```
12
13      public static void main(String[] args) {
14          System.out.println("Welcome to your new job");
15
16          announceDeveloperTeaTime();
17
18          System.out.println("Write code");
19          System.out.println("Review code");
20
21          announceDeveloperTeaTime();
22
23          System.out.println("Get promoted!");
24
```

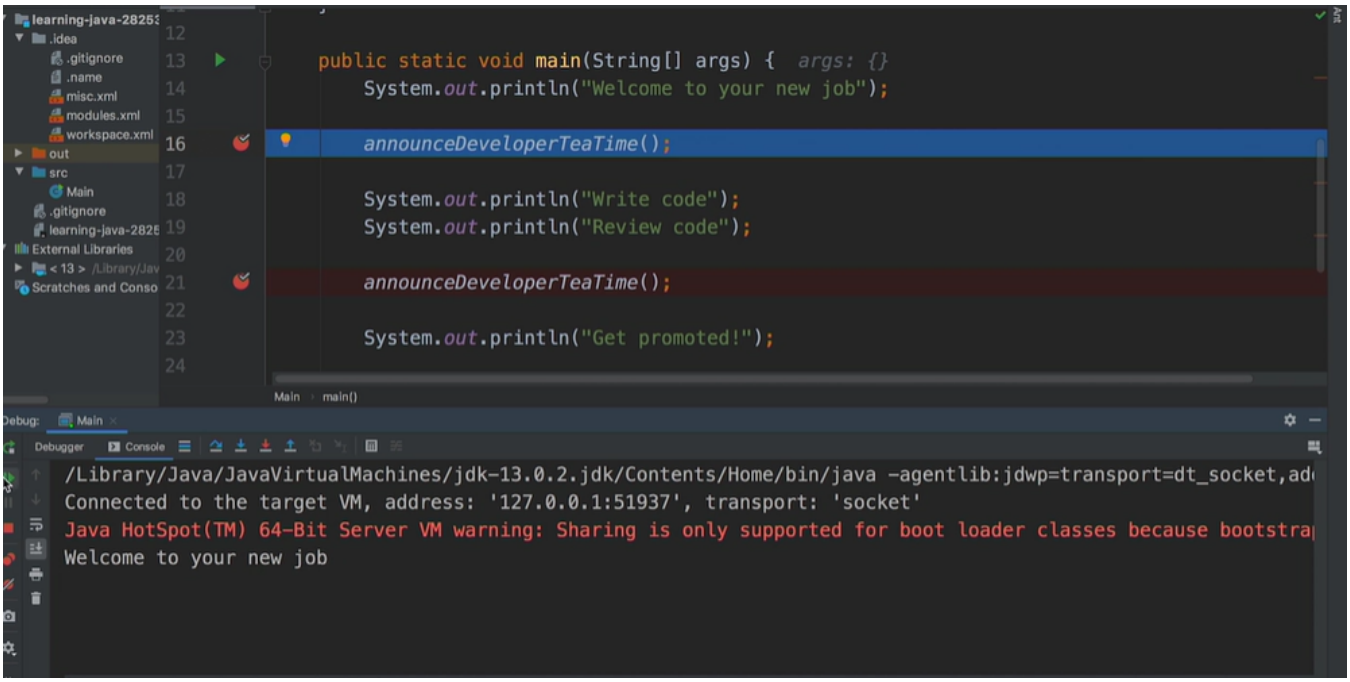
Let's add a breakpoint every time we call announceDeveloperTeaTime.

```
3   public class Main {
4
5       public static void announceDeveloperTeaTime() {
6           System.out.println("Waiting for developer tea time...");
7           System.out.println("Type in a random word and press Enter to start developer tea tim
8           Scanner input = new Scanner(System.in);
9           input.next();
10          System.out.println("It's developer tea time!");
11      }
12
13      public static void main(String[] args) {
14          System.out.println("Welcome to your new job");
15
16
```

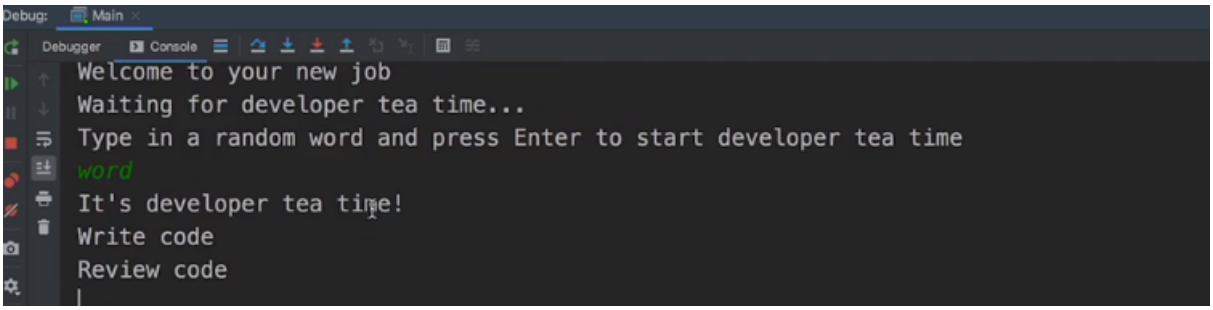
We'll keep the breakpoint at the implementation level as well. We'll run in debug mode and the first breakpoint we hit is our first call of announceDeveloperTeaTime.



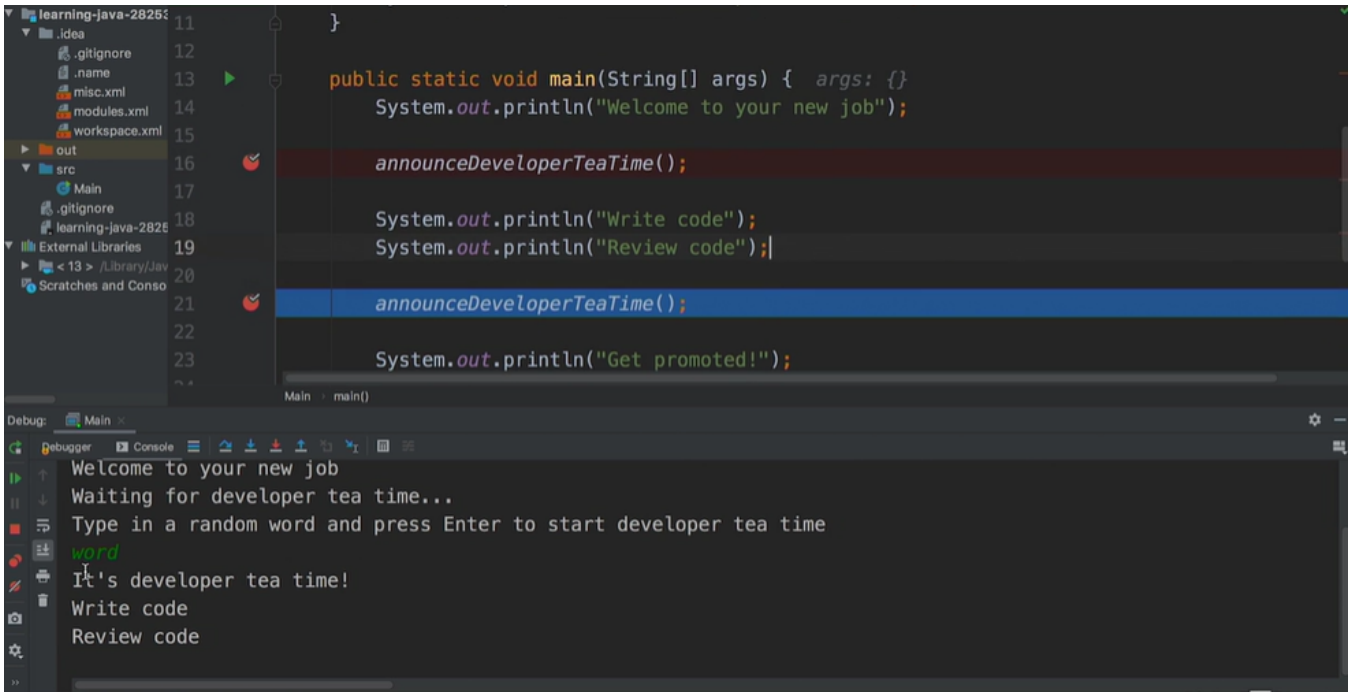
Going to our console we see Welcome to your new job has already been printed.



Let's hit continue. This is expected. We're walking through the steps of announceDeveloperTeaTime. We'll step over each of these steps and the print statements appear in the console. We'll enter our random word and continue onto the next breakpoint.

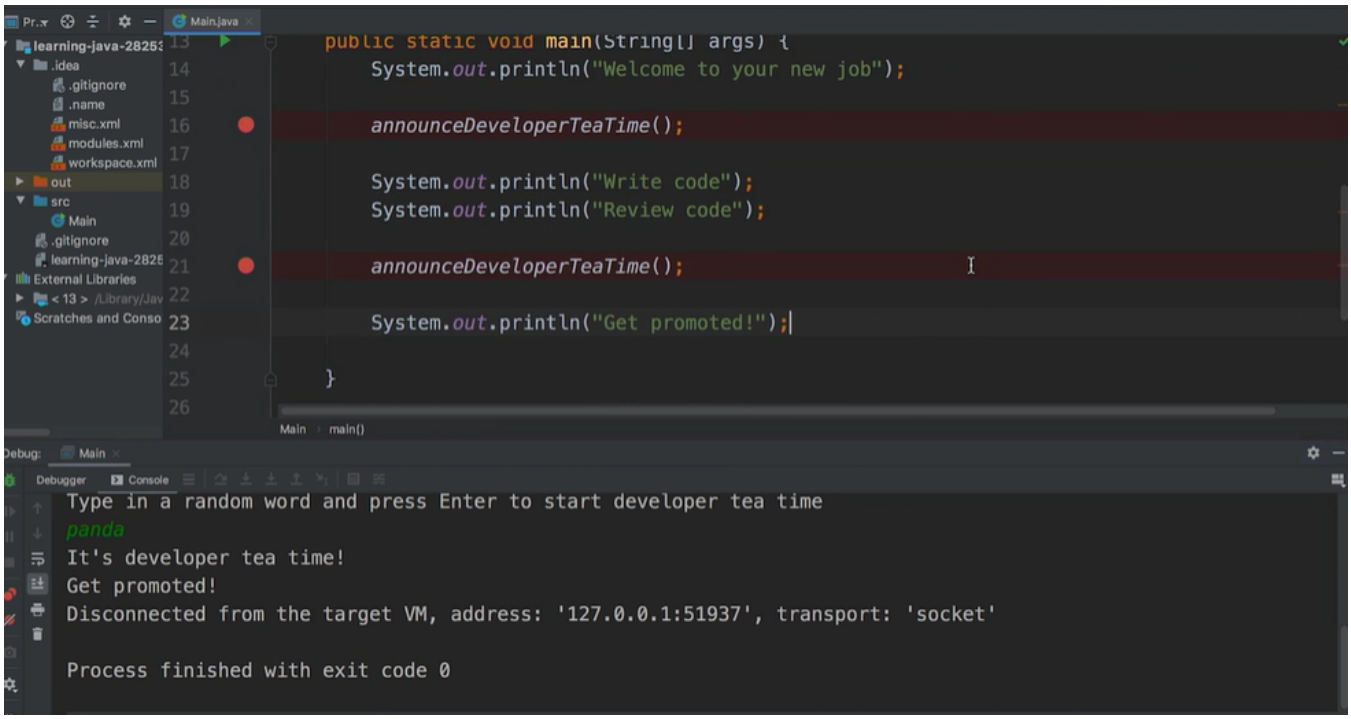


We see It's developer tea time! from the announceDeveloperTeaTime implementation but now we see write code, review code.



These live in our main function after the first announceDeveloperTeaTime call. Now it's time for the second call of our function. Let's hit continue and we go right back into those steps. Each print statement we print and then it waits for our input, in this case we'll say panda.

We'll hit continue. We announceDeveloperTeaTime again and then we get promoted which is the last line of our main function.



Sample input - We just called our first function. Now you might be thinking that this announceDeveloperTeaTime function looks pretty similar to the main function and that's because they're both functions. Whenever we hit the play button and execute a program the main function is automatically called and used.

Every time we've been writing code we've been defining the main function that's automatically called at execution. It's all coming together. In the next lesson we'll make a new function that's a little more complex.

Parameters in Java

The announceDeveloperTeaTime function had no inputs. It was just a series of steps that the program followed every time the function was called. The output was always the same. There was nothing dynamic about it. What if I wanted the output to be different depending on what was inputted into the function? For example, let's say I wanted to calculate the total cost of a meal including tip and tax. To calculate this, we would first figure out the tip by multiplying the tip rate with the listed price of the meal. We would figure out the tax by multiplying the tax rate with the listed price of the meal as well. Then we would add the tip, tax, and listed price of the meal and output the result. These are finite steps that we can label calculateTotalMealPrice. Of course, almost every meal you order will have a different listed price. And sometimes you might want to tip a different rate depending on what service you get. Sales tax rates also differ depending on what state or country you are in. How can we account for this in our function?

```
1  public class Main {
2
3      public static void calculateTotalMealPrice() {
4      }
5
6      public static void main(String[] args) {
7      }
8
9  }
10
```

What value could we possibly give listed meal price, tip rate, and tax rate if they change every time? In this lesson we are going to create a function with inputs. Adding inputs is going to allow us to insert a specific listed meal price, tip break, and tax rate into the function every time we call the function. The listed meal price, tip rate, and tax rate inputs will be defined in the function definition. But the values of these inputs will be assigned when we use the function or call the function.

Let's try implementing this in Java. Our function starts off with public static void and uses the function name calculateTotalMealPrice. Using our formula from before, we'll calculate the tip by multiplying the tip rate with the listed meal price. We'll calculate the tax by multiplying tax rate with listed meal price. We'll calculate the result by adding together listed meal price, tip, and tax. Then we'll output the result to the console.

```
public class Main {

    2 usages
    public static void calculateTotalMealPrice(double listedMealPrice,
                                                double tipRate,
                                                double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        System.out.println("Your total meal price is " + result);
    }

    public static void main(String[] args) {
```

You might notice that the listed meal price, tip rate, and tax rate are red.

```
public class Main {

    public static void calculateTotalMealPrice() {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        System.out.println("Your total meal price is " + result);
    }

    public static void main(String[] args) {
    }

}
```

And that's because we haven't defined those as inputs to our function. To define an input, we add code between the parentheses. Our first input will be listedMealPrice. So, we'll write the data type that listed meal price should have, in this case double and then the name of our input, listedMealPrice. We just defined our first input to the calculateTotalMealPrice function. And notice listedMealPrice is no longer red. Let's do this for tip rate and tax rate. Both of these will be doubles.

```
public class Main {

    public static void calculateTotalMealPrice(double listedMealPrice,
                                                double tipRate,
                                                double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        System.out.println("Your total meal price is " + result);
    }

    public static void main(String[] args) {
    }
```

Thinking back to our discussion of scope these inputs are only accessible within the function's implementation. We'll give these inputs a value later on, but since they are created here as a part of the function's definition they can only be used within the function.

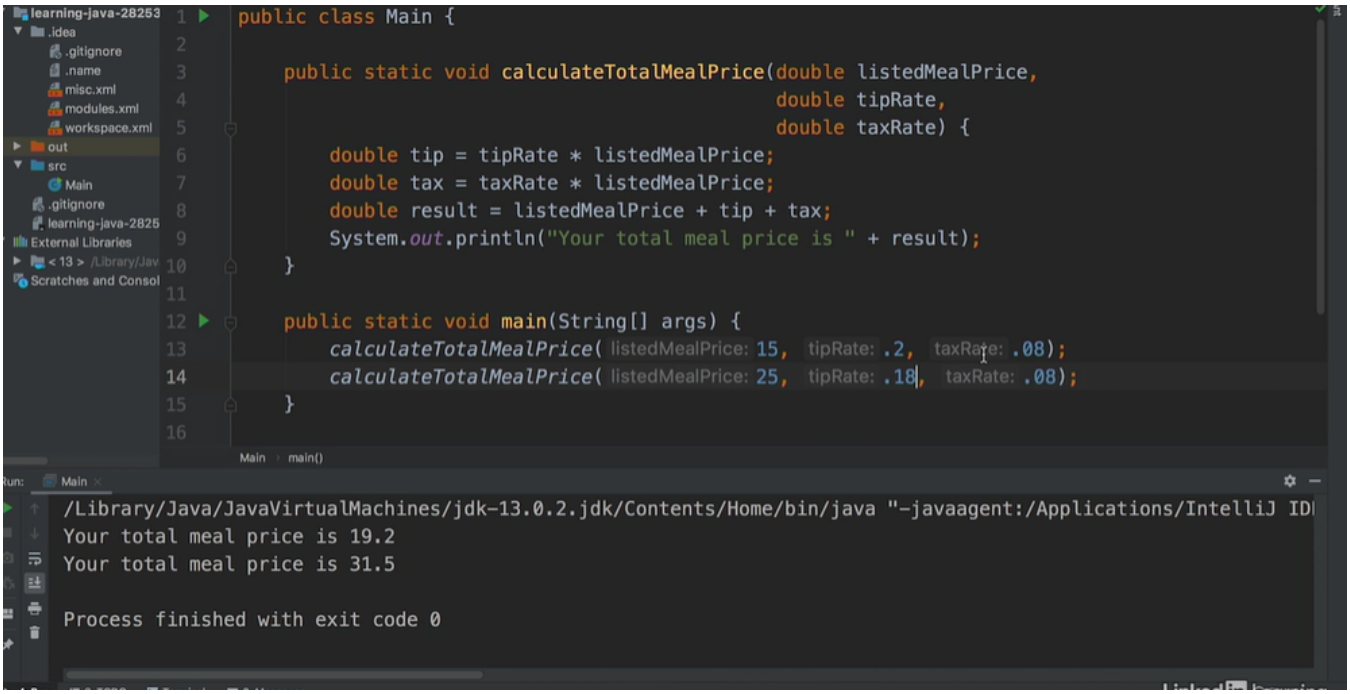
Now that our inputs are defined, let's add a call to calculateTotalMealPrice. If we try calling this function like our announceDeveloperTeaTime function, it won't work. Unlike announceDeveloperTeaTime we defined inputs to this function, and we have to give each of those inputs a value when we call the function. The first input to our function is listed meal price. We can give this input the value 15 by adding 15 between the parentheses. But we still have an error. That's because we need to add a value for each input not just the first. We also have the tip rate and tax rate. We'll add .2, 20% for the tip and .08 for the tax rate, 8%. It's important to remember that the order does matter here.

```java
public class Main {

    2 usages
    public static void calculateTotalMealPrice(double listedMealPrice,
                                                double tipRate,
                                                double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        System.out.println("Your total meal price is " + result);
    }

    public static void main(String[] args) {
        calculateTotalMealPrice( listedMealPrice: 15,  tipRate: .20,  taxRate: .08);
```

So, the first input defined in the function will get the first value in the parentheses and so on. Let's run the code and see what happens. And we get an output. Your total meal cost is 19.2. Let's try changing the values of the inputs to this function. Instead of 15, we'll say the listed meal price is 25. We'll also say we only want to tip 18% or .18. The tax rate will stay the same

```java
public class Main {

    2 usages
    public static void calculateTotalMealPrice(double listedMealPrice,
                                                double tipRate,
                                                double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        System.out.println("Your total meal price is " + result);
    }

    public static void main(String[] args) {
        calculateTotalMealPrice( listedMealPrice: 15,  tipRate: .20,  taxRate: .08);
        calculateTotalMealPrice( listedMealPrice: 25,  tipRate: .18,  taxRate: .08);
    }

}
```

Let's run the code again and the second output says the meal price is 31.5.



Changing the inputs to our calculateTotalMealPrice function affected the outputs. This is the first time we've used a function with different input values. In the next lesson we'll continue to make calculateTotalMealPrice more sophisticated.

Return types in Java

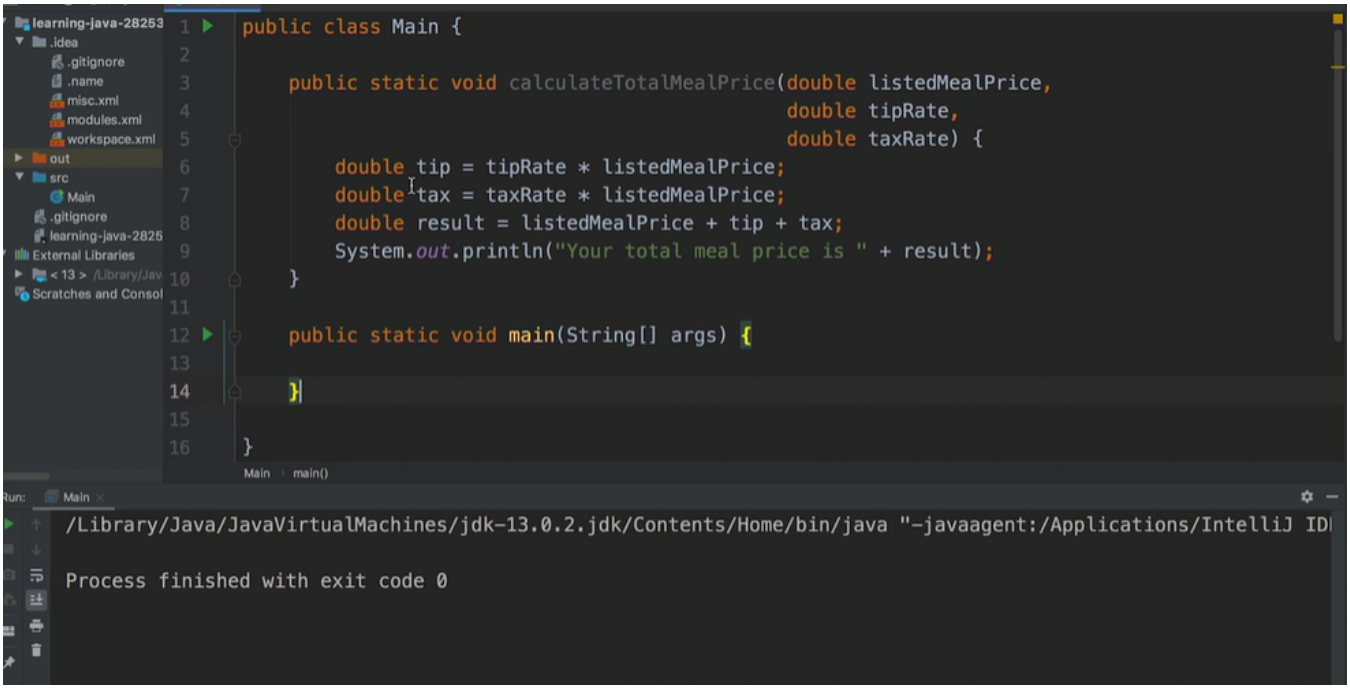
The only way we've outputted data is System.out.println, and this really outputs data to the user. We've gathered input from the user in the past with the scanner, and in the last lesson, we learned how to input data into functions using parameters. Now we're going to learn how to output data from functions using return types. We'll talk more about return types in a second, but imagine you go to a restaurant with four of your friends, and you all order similarly priced meals and want to tip and tax the same amount. The listed meal price comes out to be $100. You want to tip 20%, or .20, as a group, and the sales tax is .08, 8%. With the function you created, we can calculate the total meal cost, which will come out to 128, and that would get printed out to the console.

```
public class Main {

    public static double calculateTotalMealPrice(double listedMealPrice,
                                                  double tipRate,
                                                  double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        return result;
    }
```

Now what if we wanted to split the bill among me and my friends? The total meal cost is 128, but I want to know what each person owes on the bill. We could add a parameter to our function that represents the number of people at the meal and divide the final result by that number, but then the function name doesn't really make sense, because the printed-out statement should represent the total meal cost, not my portion. What I really want to do is somehow get access to the data outputted by the calculateTotalMealPrice function and manipulate it some more. I want to divide it by five. We can do this with return types. For the functions we've created, we've always prefixed them with public static void. Void is actually the return type.

```java
public class Main {

    public static double calculateTotalMealPrice(double listedMealPrice,
                                                  double tipRate,
                                                  double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        return result;
    }

    public static void main(String[] args) {
```

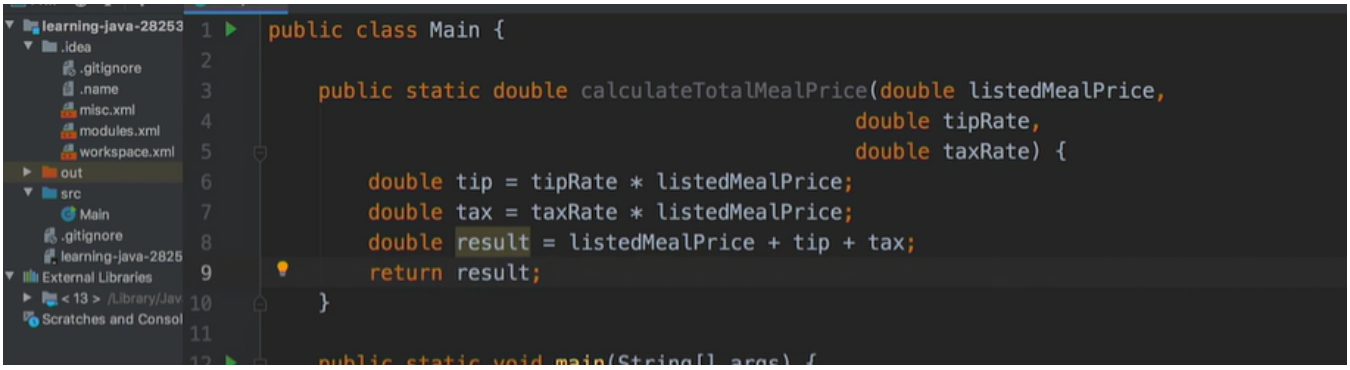
Every function in Java returns some value or nothing, and every function we've created so far has returned void, or nothing. For the calculateTotalMealPrice function, we want to return the value of the result variable. The result variable is a double, so we'll change void to double to make double our return type.



With the double return type, we have to make our function return a double value. We can do this using the return keyword. We'll write return and then the value we want the function to return, in this case, the value stored inside of result. Awesome. Our function returns the calculateTotalMealPrice result. Now we can access and save the result in a remain function. How do we save the result? Just like we save any value in Java. We can create a variable. First, we'll call the calculateTotalMealPrice function with 100, 0.2, and 0.08 as values which will map to listedMealPrice, tipRate, and taxRate.

```java
public class Main {

    public static double calculateTotalMealPrice(double listedMealPrice,
                                                  double tipRate,
                                                  double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        return result;
    }

    public static void main(String[] args) {
        double groupTotalMealPrice = calculateTotalMealPrice( listedMealPrice: 100,  tipRate: .2,  taxRate: .08);
        System.out.println(groupTotalMealPrice);
```

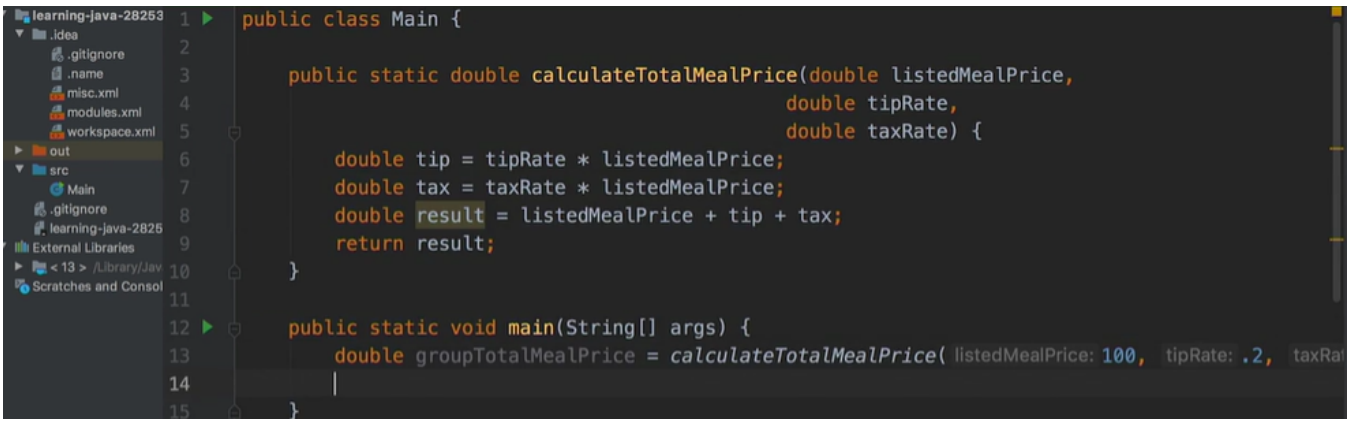
```java
public class Main {

    public static double calculateTotalMealPrice(double listedMealPrice,
                                                  double tipRate,
                                                  double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        System.out.println("Your total meal price is " + result);
    }
```



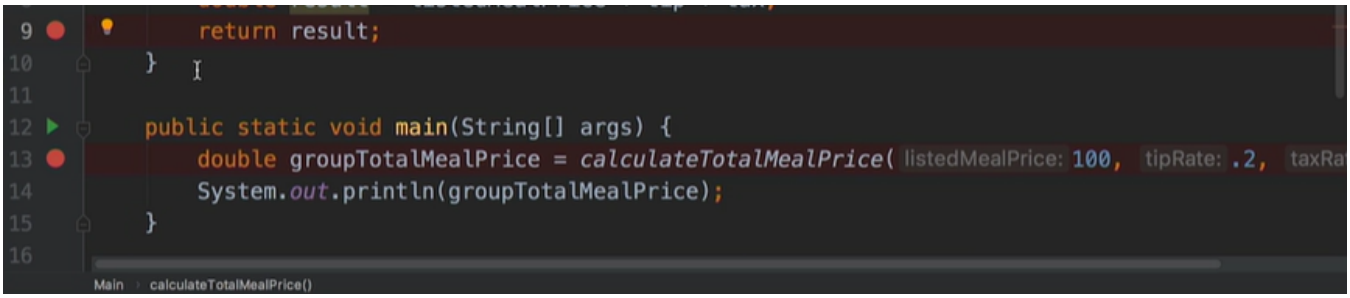
Then we'll save the output in a variable called groupTotalMealCost. This variable will be a double, because the return type to the calculateTotalMealPrice function is a double.

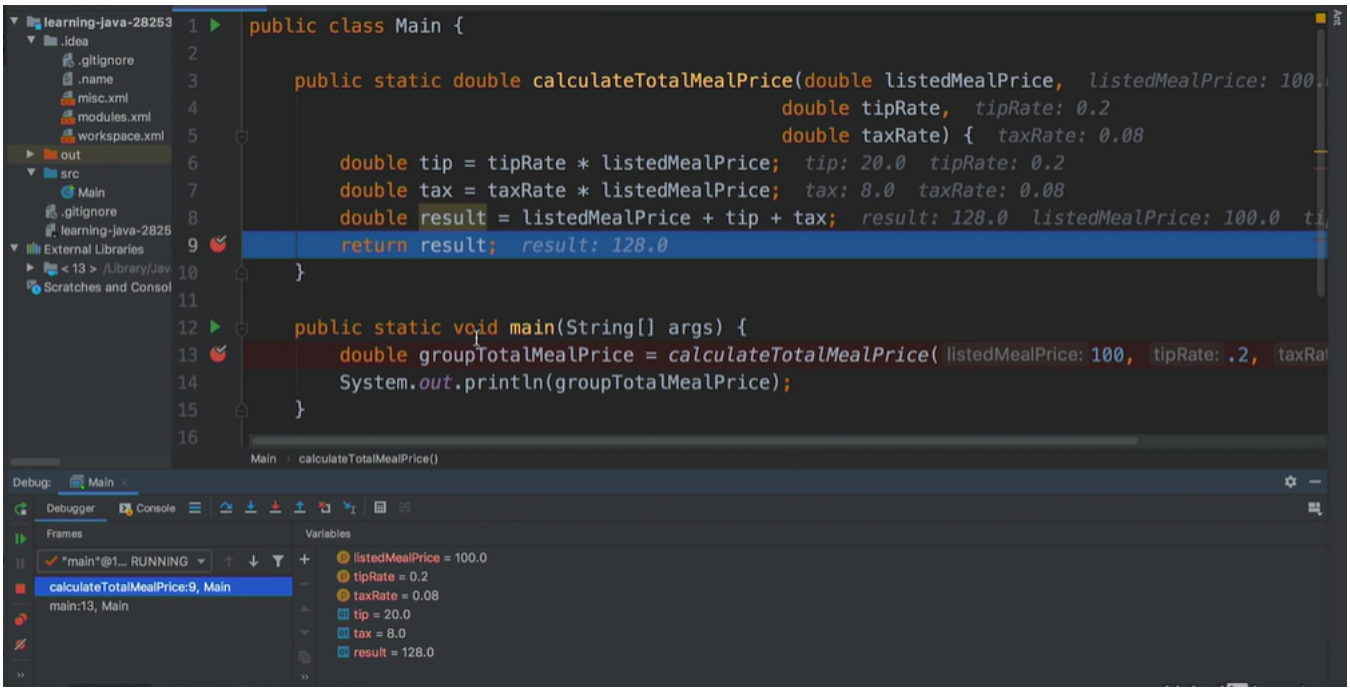
```java
public class Main {

    public static double calculateTotalMealPrice(double listedMealPrice,
                                                  double tipRate,
                                                  double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        System.out.println("Your total meal price is " + result);
    }

    public static void main(String[] args) {

    }

}
```

groupTotalMealPrice will have the same value as result.

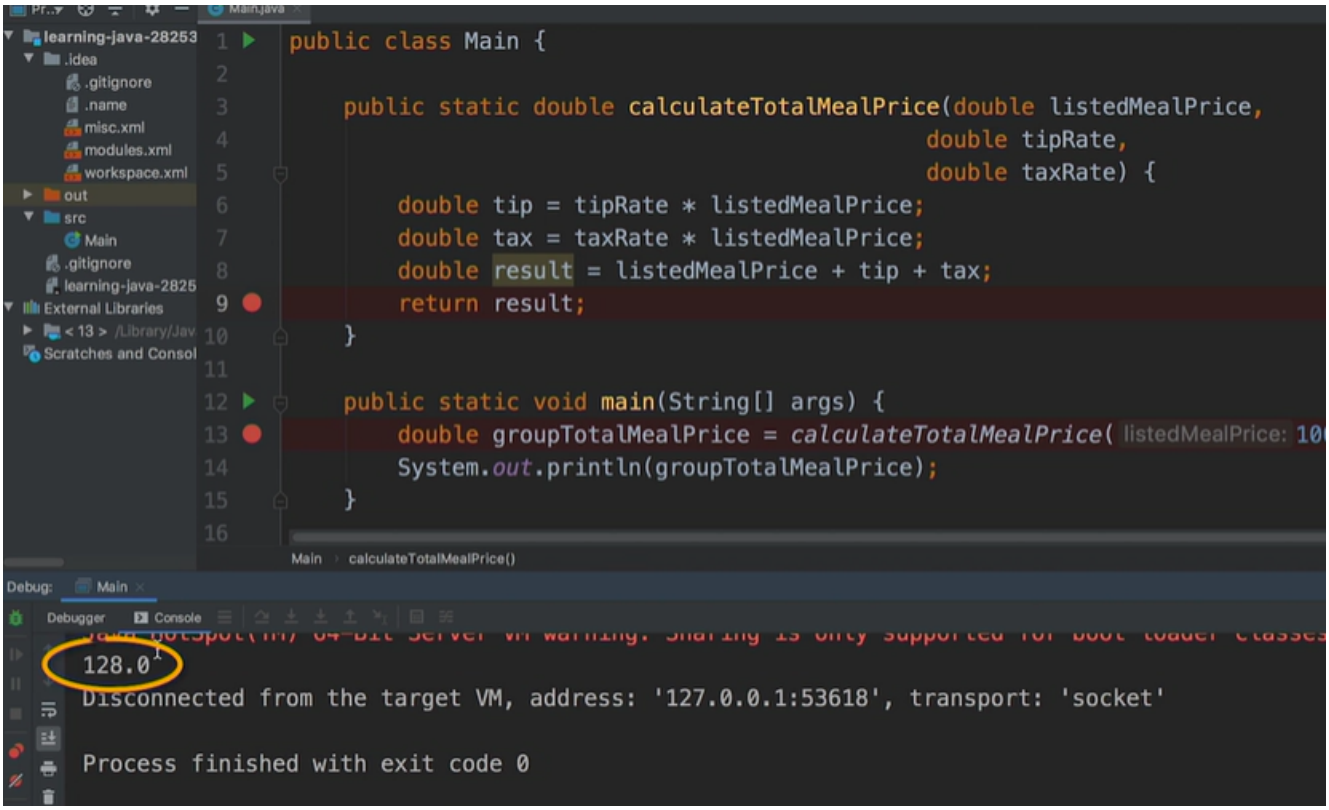


Let's print groupTotalMealPrice out to the console, add some break points, and run the code in debug mode. We'll add one break point at the function call and another break point when the result is returned.





Let's use the debugger. Here, we're right before the function call. The function will be called, and the result will be saved in groupTotalMealPrice. We'll hit Continue. Inside the function, we have 100 for our listedMealPrice, 0.2 for the tipRate, and 0.08 for the taxRate. Result has the value 128.



And if we hit Play again, we should see 128 in our console. And there it is. Returning a value allows us to take a value of a variable within a function and allow it to be accessed at a different scope of our program. Before, variables and values created within a function could only be accessed and used within that function. With return types, we can calculate a given value and make it accessible to a different part of our program. Now the ultimate goal was to figure out how much each person in the group owes if the meal is split. There are five people in the group, so we can divide the groupTotalMealPrice by five and save it in a variable called individualTotalMealPrice.

```
                                            double tipRate,
                                            double taxRate) {
        double tip = tipRate * listedMealPrice;
        double tax = taxRate * listedMealPrice;
        double result = listedMealPrice + tip + tax;
        return result;
    }

    public static void main(String[] args) {
        double groupTotalMealPrice = calculateTotalMealPrice( listedMealPrice: 100,  tipRate: .2
        System.out.println(groupTotalMealPrice);

        double individualMealPrice = groupTotalMealPrice / 5;
        System.out.println(individualMealPrice);
    }
```

Let's run the code and see what the individual price is. It's 25.6.

```
/Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java "-javaagent:/Applications/IntelliJ ID
128.0
25.6

Process finished with exit code 0
```

We just made our function a little more sophisticated by adding a return type. In this case, our return type happened to be a double, but it could be a Boolean, int, char, string, or any other data type that we've covered we've been defining and using our own custom functions, but there are functions that are already defined in Java, and we can just use them.

Using built-in functions in Java

We've been creating our functions from scratch. We've defined various functions, added parameters, added return types, and then called them in our main function. These functions are called user-defined functions. We are defining the finite steps, naming the function appropriately, and then calling it in our code. There are many functions that developers want to use, but not necessarily define themselves, and as a courtesy, the creators of Java have defined some of them for us. We just have to call the function by the name that's already defined. We don't have to define it.

One function that we've been using a lot that's built-in is println. We never defined system.out.println, but we can call it because it's a built-in function to Java. Back at the beginning of the course, we talked about operations on data types, operations we can run on strings, operations we can run on ints, and many of these operations are really functions. The .equals operation that we've run in the past is really a string function. The value we've added between the parentheses is the string we want to compare the original string to. The way we call a built-in function might seem a little different than a user-defined function, and that's because it is. We are using the dot operator to access many of these built-in functions. For system.out.println, we have to access out from system and then the println function from out, but the main takeaway here is sometimes we'll define our custom functions and sometimes we'll use built-in functions.

So, how do we find out about all of these built-in functions? Let's say you want to create a custom function for exponentiation, something that will take a given number to the second power, or the fifth power. This is a common task and there might be something in Java that already does this. So, you'll Google around and try to find some examples of a built-in function that already does exponentiation or an example of a user-defined function. Let's try doing that now.

All right, here's one, math.pow,

Math pow() method in Java with Example - GeeksforGeeks

and there are two inputs. The first input is for the base and the second input is for the exponent.

Syntax:

```
public static double pow(double a, double b)
Parameter:
a : this parameter is the base
b : this parameter is the exponent.
Return :
This method returns a^b.
```

We take the base to the power of the exponent, hence the function name pow. The way we access it is through math using the dot operator.

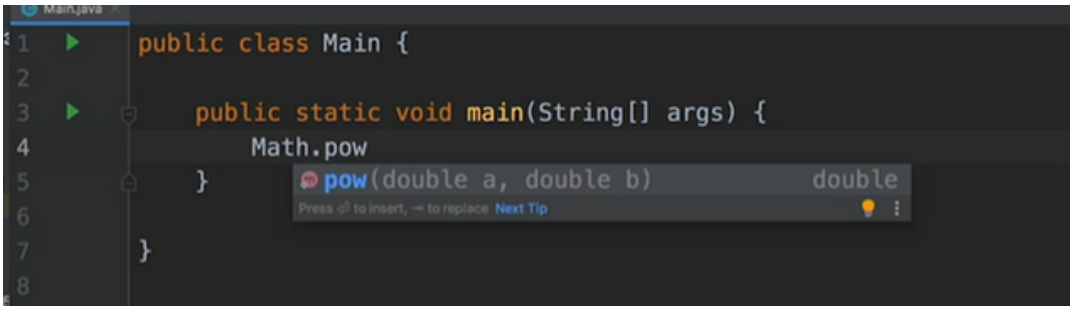
Read Discuss

```java
// Java program to demonstrate working
// of java.lang.Math.pow() method

import java.lang.Math;

class Gfg {

    // driver code
    public static void main(String args[])
    {
        double a = 30;
        double b = 2;
        System.out.println(Math.pow(a, b));

        a = 3;
        b = 4;
        System.out.println(Math.pow(a, b));

        a = 2;
        b = 6;
        System.out.println(Math.pow(a, b));
    }
}
```

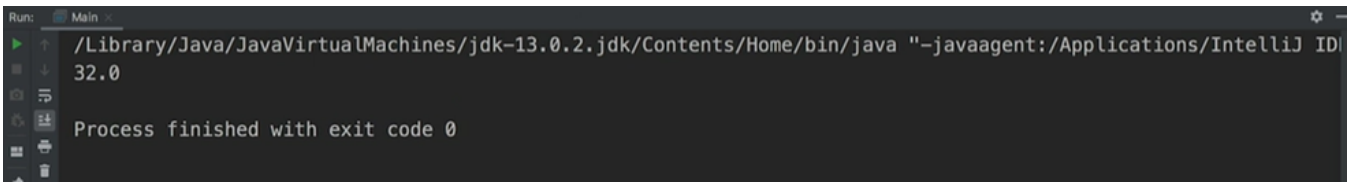
Let's try writing this in our code.

```
public class Main {

    public static void main(String[] args) {
```

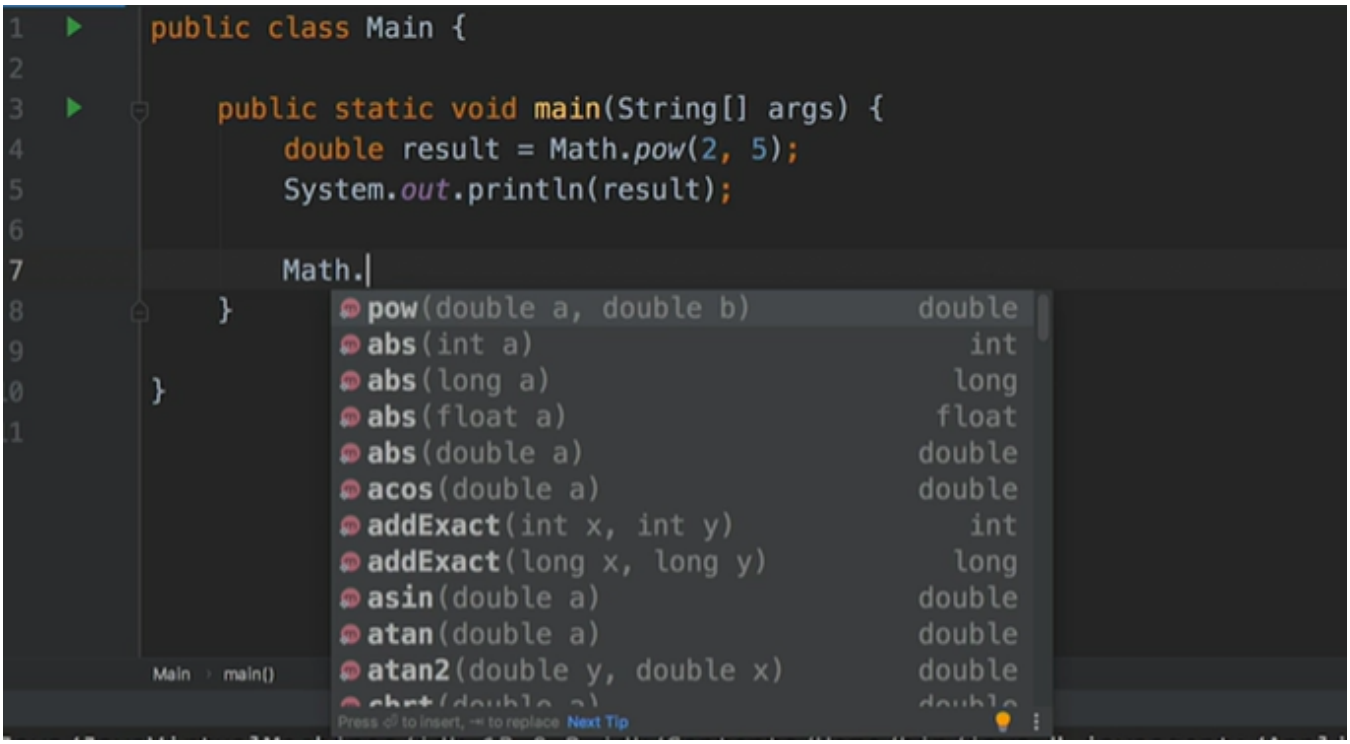
We'll write math.pow and you'll notice that some information about the function pops up in our IDE. We see that it has two parameters that are both doubles and it returns a double. We'll add two as the base and five as our exponent.



We'll save it in a variable called result and then print it to the console. We'll run the program, and we get 32, which is two to the power of five.



Now, if we type math., we can see there are a lot of different mathematical functions that we can call already built into Java.



Math is a part of the Java standard library, which contains all of the different built-in functions we can access. Math is just one component containing a series of built-in functions. So, what's the benefit of using built-in functions? You don't have to write the implementation of the function yourself. The function is already defined, and you can just call it. All you have to know about the function is what it takes as input and what the expected output is based on what the function does. Usually, if there is a built-in function for the thing you want to accomplish, you should use it instead of creating your own custom function. Built-in functions have been tested a lot by the creators of the programming language. They are basically guaranteed to do what they are described to do. Next, you'll test your functions in a coding challenge.

Sample solution: Salary calculator

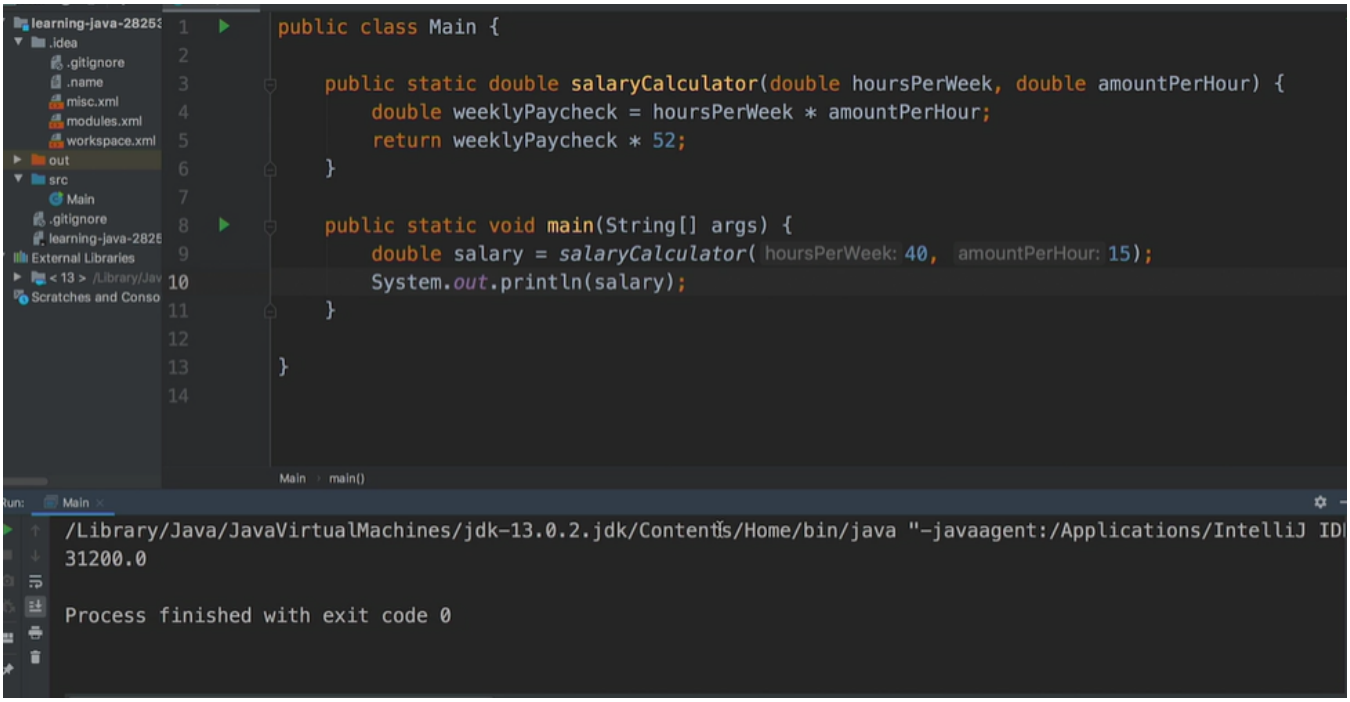
One way - we can create a function that calculates an employee's salary in java.

```
public class Main {
    public Main() {
    }

    public static double salaryCalculator(double hoursPerWeek, double amountPerHour, int vacationDays) {
        if (hoursPerWeek < 0.0) {
            return -1.0;
        } else if (amountPerHour < 0.0) {
            return -1.0;
        } else {
            double weeklyPaycheck = hoursPerWeek * amountPerHour;
            double unpaidTime = (double)vacationDays * amountPerHour * 8.0;
            return weeklyPaycheck * 52.0 - unpaidTime;
        }
    }
```

For the function, we return a double, but you could return an INT, as long as it's numerical. We also make both of our parameters doubles as well. You could make them INTs, but it limits your functionality. In the implementation between the curly brackets, the first thing we do is calculate the weekly paycheck. We multiply the number of hours worked per week, by the amount the employee makes per hour.

```java
public class Main {

    public static double salaryCalculator(double hoursPerWeek, double amountPerHour) {
        double weeklyPaycheck = hoursPerWeek * amountPerHour;
        return weeklyPaycheck * 52;
    }

    public static void main(String[] args) {
        double salary = salaryCalculator( hoursPerWeek: 40,  amountPerHour: 15);
        System.out.println(salary);
    }

}
```

In the next step, we return the yearly salary by multiplying the weekly paycheck times 52. In this case, we are assuming the employee works every week during the year. To call the function, we use the function's name, "Salary Calculator". In this example, we say, "The employee works 40 hours per week and makes $15 an hour." We save the result of this function call, in a variable called "Salary", which we make a double. After the salary is calculated, we print it out to the user. Let's run this program, and we see 31200.0 in the console.



Now what happens if we put in a negative number of hours per week, Or a negative amount per hour? That's going to cause a problem, so we should do some error checking inside the salary calculator. We'll write, "If hours per week is less than zero", then we want to return some value that says, "Hey your input's invalid." In this case, we'll return negative one. Now let's say, hours per week is valid, but amount per hour is not valid, it's less than zero. We'll want to check that as well. So, we'll check amount per hour, is it less than zero? If so, we'll return negative one saying this input is invalid.

For the bonus, you can add another parameter, called vacation days. This would be an INT or double. It just depends on if you want to allow your employees to take half days for vacation. In this case, we'll make ours an INT. To account for the vacation days, we'll create a variable called "Unpaid Time", and set it to the value of amount per hour times eight. This accounts for how much the employee should be paid less per year, based on one vacation day, however since we have a vacation days variable, we can multiply this by vacation days to see how much less the employee should be paid per year.

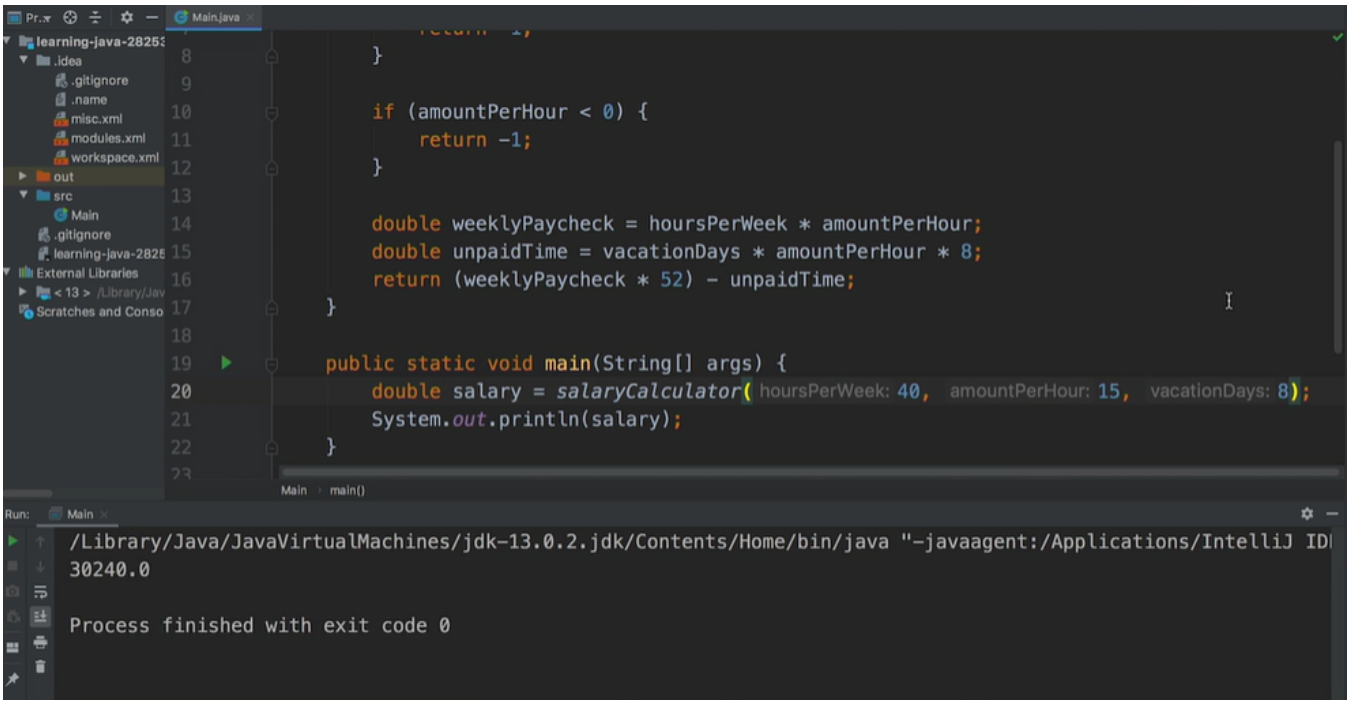
```java
public class Main {

    public static double salaryCalculator(double hoursPerWeek,
                                          double amountPerHour,
                                          int vacationDays) {
        if (hoursPerWeek < 0) {
            return -1;
        }

        if (amountPerHour < 0) {
            return -1;
        }

        double weeklyPaycheck = hoursPerWeek * amountPerHour;
        double unpaidTime = vacationDays * amountPerHour * 8;
        return weeklyPaycheck * 52;
```

Now that we have the unpaid time set, we can subtract it from the overall salary. We'll put some parenthesis around weekly paycheck times 52 and then subtract the unpaid time that the employee took off.

```java
    }

    public static void main(String[] args) {
        double salary = salaryCalculator( hoursPerWeek: 40,  amountPerHour: 15,  vacationDays: 8);
        System.out.println(salary);
    }
```

Main › main()

Let's edit how we call the function and run this program again. In this case, we'll say, "The employee took eight vacation days.". With eight vacation days, the employee only makes $30,240. The program works!



Demo

Schedule a demo with your Topic Advisor (TA) to show the results of your labs.

Next Steps

- Schedule a demo with a Topic Advisor
- Check your CELL group schedule in SharePoint
- Continue to collaborate with others and be active in MS Teams.
- If you noticed any errors on this page, please let the CELL Program Team know by sending an email to the Technology Learning Team mailbox: g31488@att.com
- Navigate to Iteration 4 and continue your CELL journey.

Previous Iteration	tWiki Home	Next Iteration
Iteration 2	tWiki Home Page	Iteration 4

Previous Iteration	Wiki Home	Forums
Iteration 2	Wiki Home Page	Forums