Working with IntelliJ – Part 2

Student Workbook

Table of Contents

Module 1

IntelliJ Code Generation

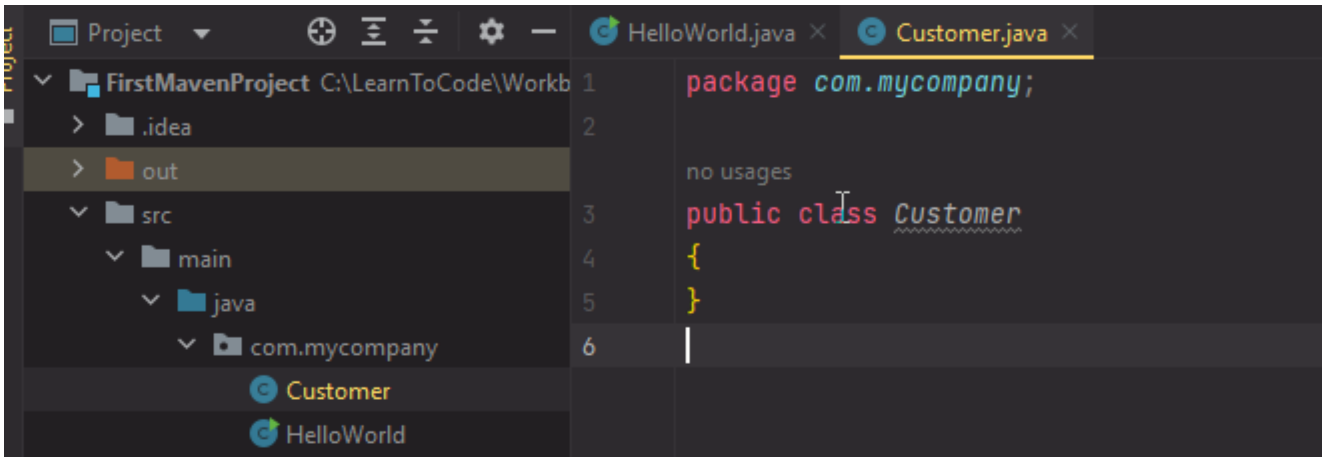
Section 1–1

Generating Code

Creating a New Non-Runnable Class

- Now that you got the hang of creating a class, you can create as many classes as you'd like.

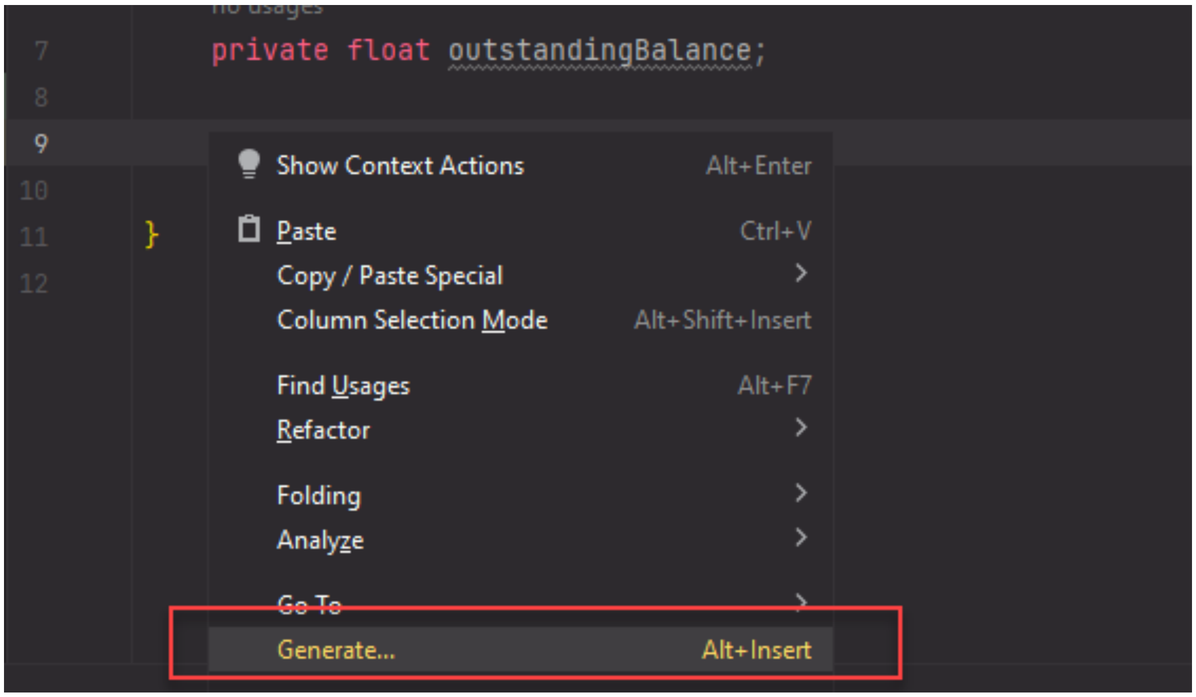
- Select the package again and create a new class. In this example, we'll create new class called **Customer** with private member variables.



```
package com.mycompany;

no usages
public class Customer
{
}
```

Using IntelliJ to Generate Constructors

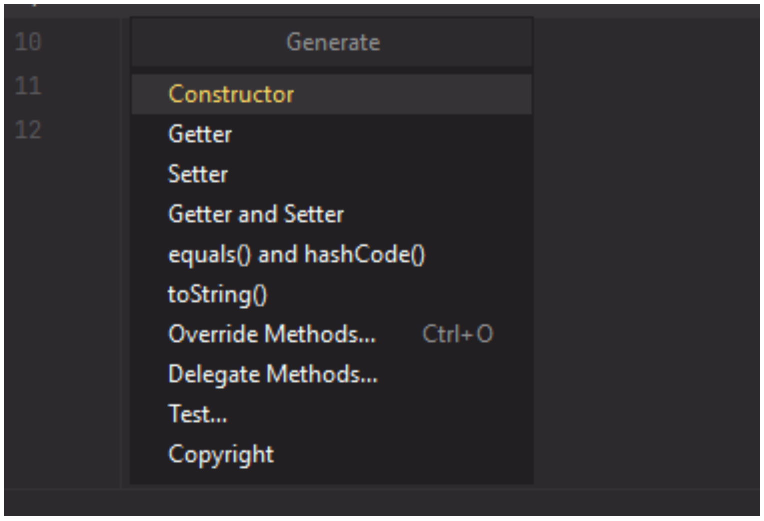
- **To generate a fully developed class, you can create private member variables representing the internal state or properties of your object.**
- **For example, a** `Customer` **would have an** `id`**, a** `name`**, an** `outstandingBalance`**, etc.**
- **To automatically generate constructors, getters and setters, toString, equals, and hashCode you can right click any empty line in your class where you wish to create your code and select Generate**



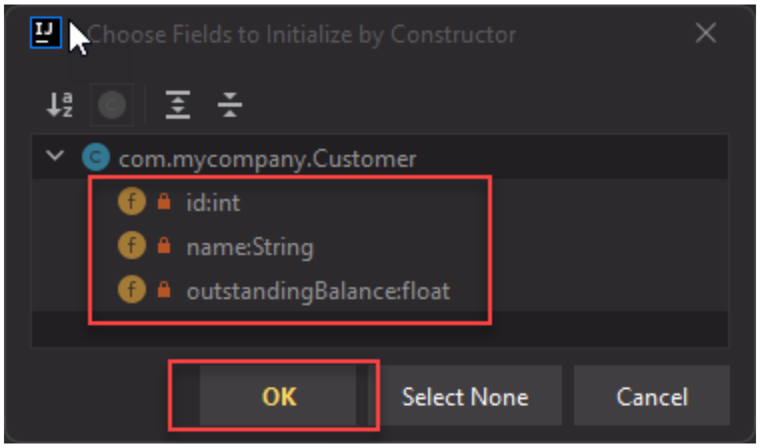
```
no usages
    private float outstandingBalance;

}
```

- **You have a wide variety of option to then choose from**

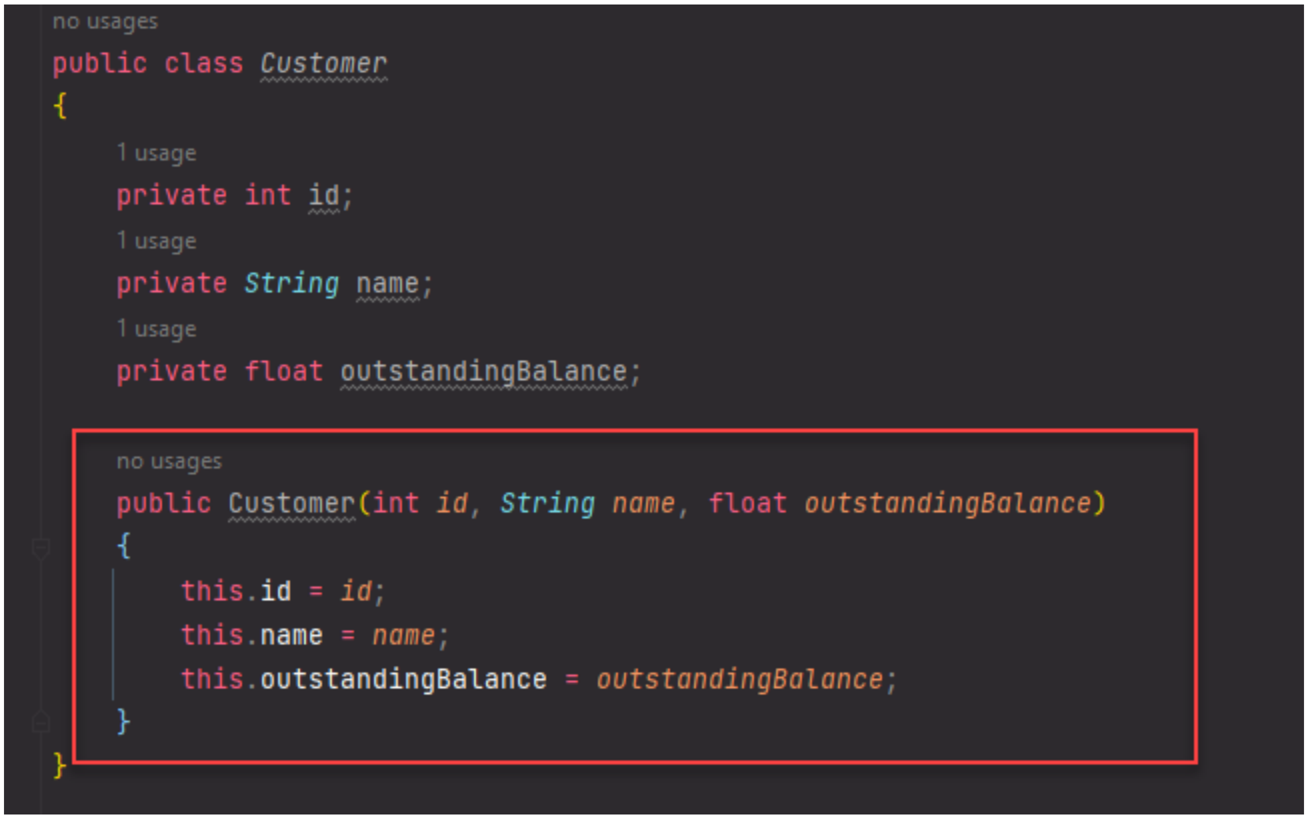
- **Select Constructor**



- **Highlight ALL fields to select them and click OK**
  - Ctrl-click to select multiple fields



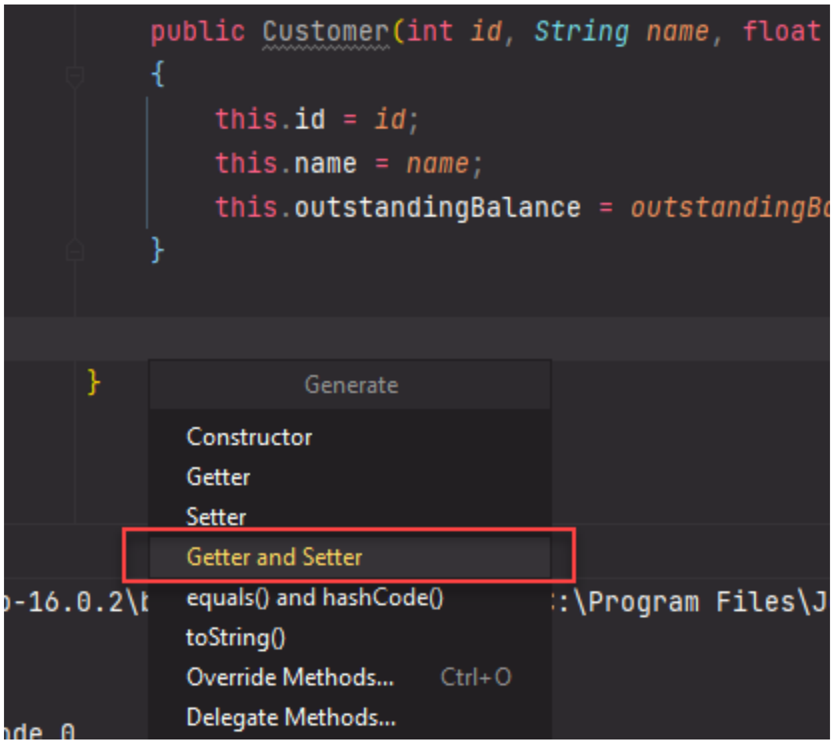
- **This will create the constructor for you**



```
no usages
public class Customer
{
    1 usage
    private int id;
    1 usage
    private String name;
    1 usage
    private float outstandingBalance;

    no usages
    public Customer(int id, String name, float outstandingBalance)
    {
        this.id = id;
        this.name = name;
        this.outstandingBalance = outstandingBalance;
    }
}
```

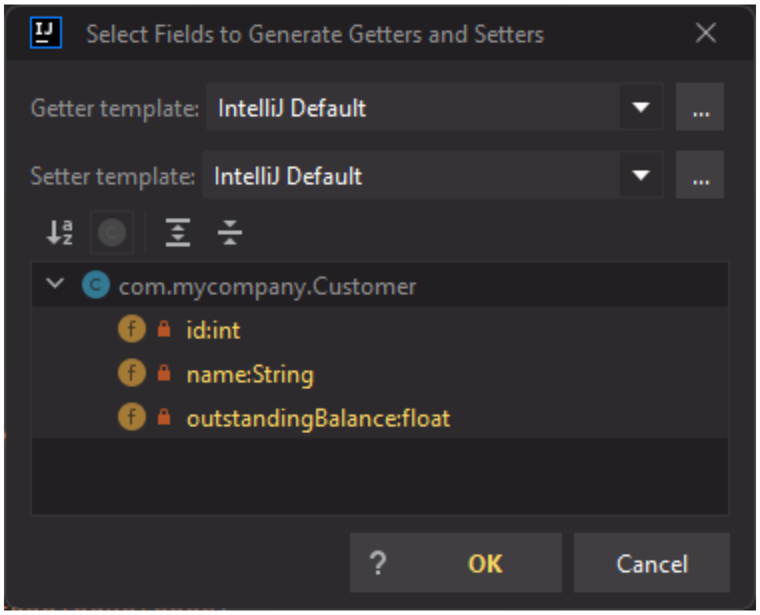
Using IntelliJ to Create Getters and Setters

- Using the same technique by right clicking on a blank line, you can select **Generate** and choose **Getter and Setter**



```
public Customer(int id, String name, float
{
    this.id = id;
    this.name = name;
    this.outstandingBalance = outstandingBa
}
```

- **Select the fields that you wish to create a getter and setter for. Then click OK**

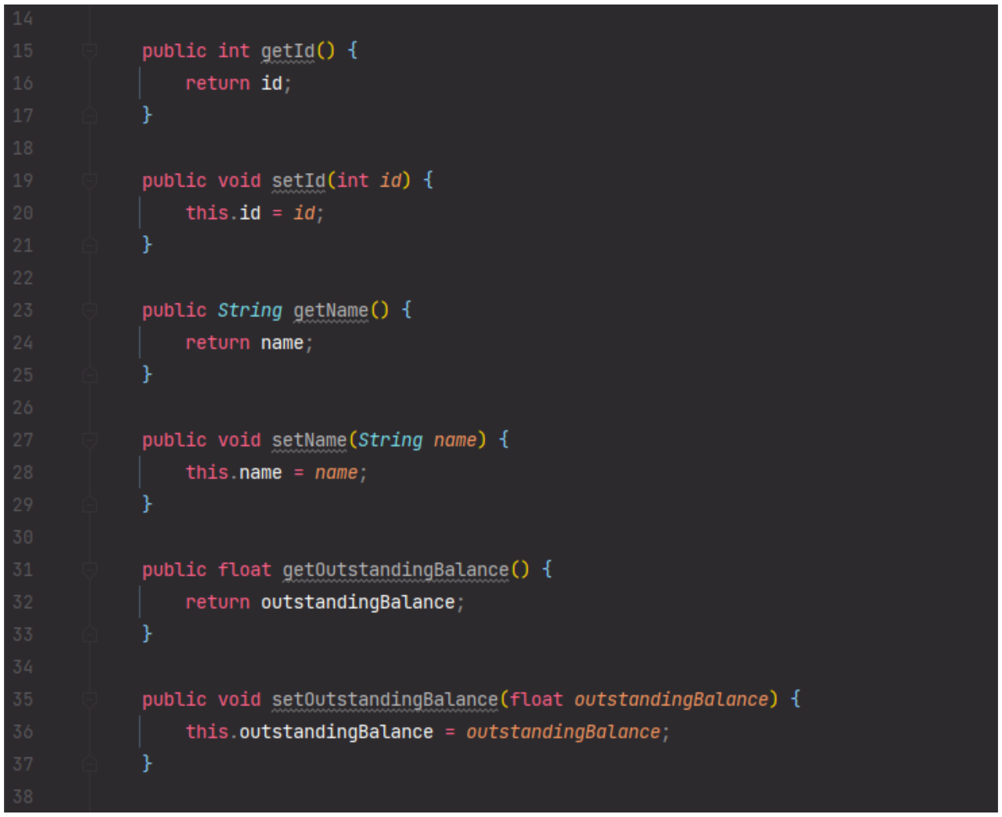


Getter template: IntelliJ Default

Setter template: IntelliJ Default

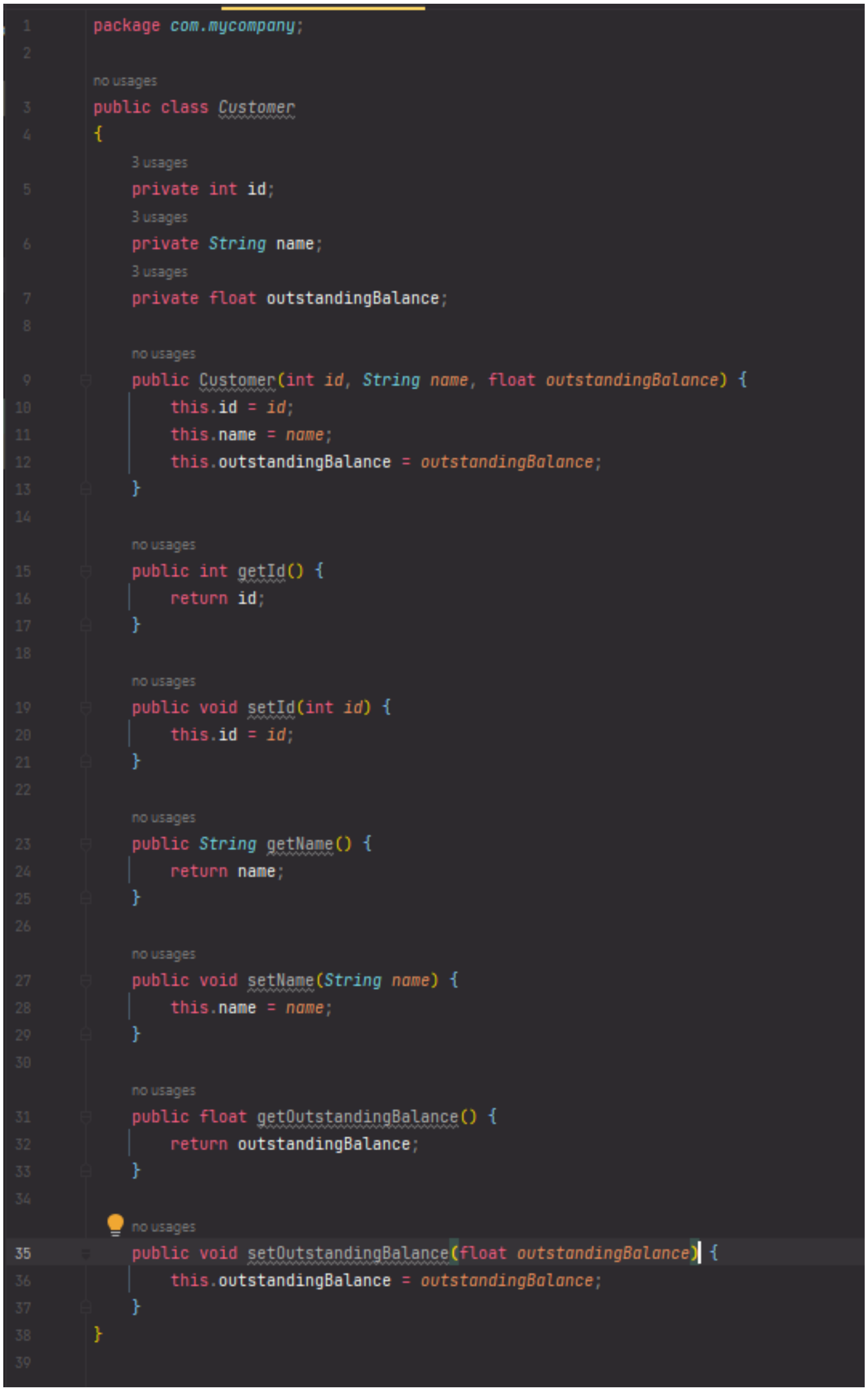
com.mycompany.Customer
- id:int
- name:String
- outstandingBalance:float

- **This will generate your getters and setters**



```java

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public float getOutstandingBalance() {
        return outstandingBalance;
    }

    public void setOutstandingBalance(float outstandingBalance) {
        this.outstandingBalance = outstandingBalance;
    }

```

Viewing your Work so far

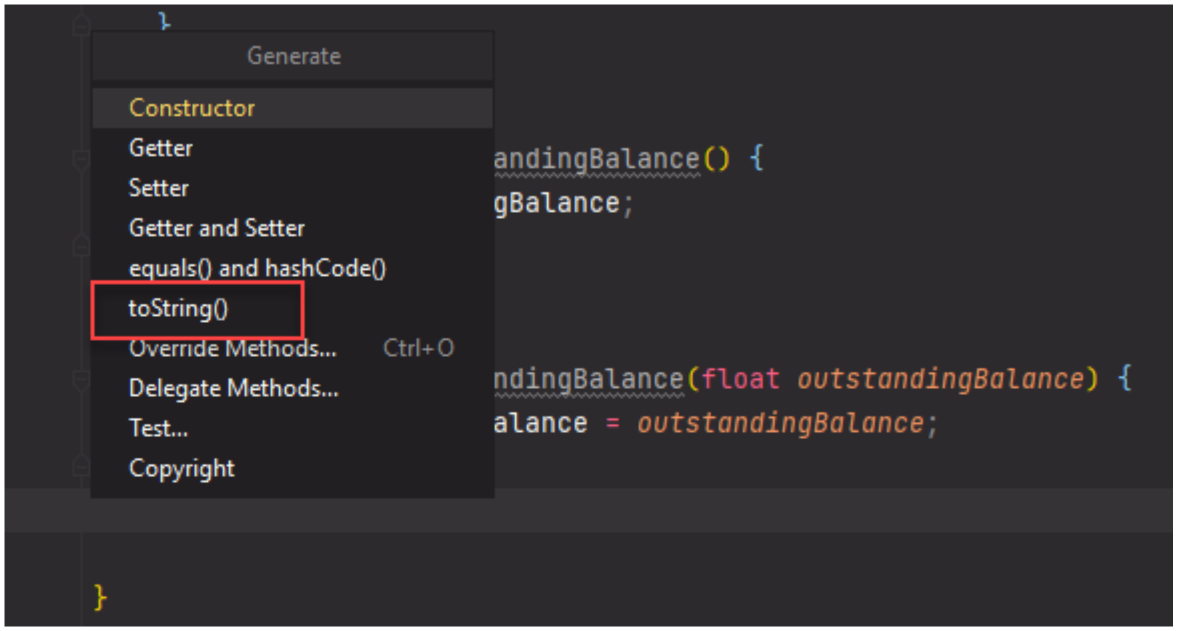


```java
package com.mycompany;

no usages
public class Customer
{
    3 usages
    private int id;
    3 usages
    private String name;
    3 usages
    private float outstandingBalance;

    no usages
    public Customer(int id, String name, float outstandingBalance) {
        this.id = id;
        this.name = name;
        this.outstandingBalance = outstandingBalance;
    }

    no usages
    public int getId() {
        return id;
    }

    no usages
    public void setId(int id) {
        this.id = id;
    }

    no usages
    public String getName() {
        return name;
    }

    no usages
    public void setName(String name) {
        this.name = name;
    }

    no usages
    public float getOutstandingBalance() {
        return outstandingBalance;
    }

    no usages
    public void setOutstandingBalance(float outstandingBalance) {
        this.outstandingBalance = outstandingBalance;
    }
}
```

Using IntelliJ to Generate a toString()

- We can now generate a toString implementation for your class much in the same way you created your constructor and getters and setters.

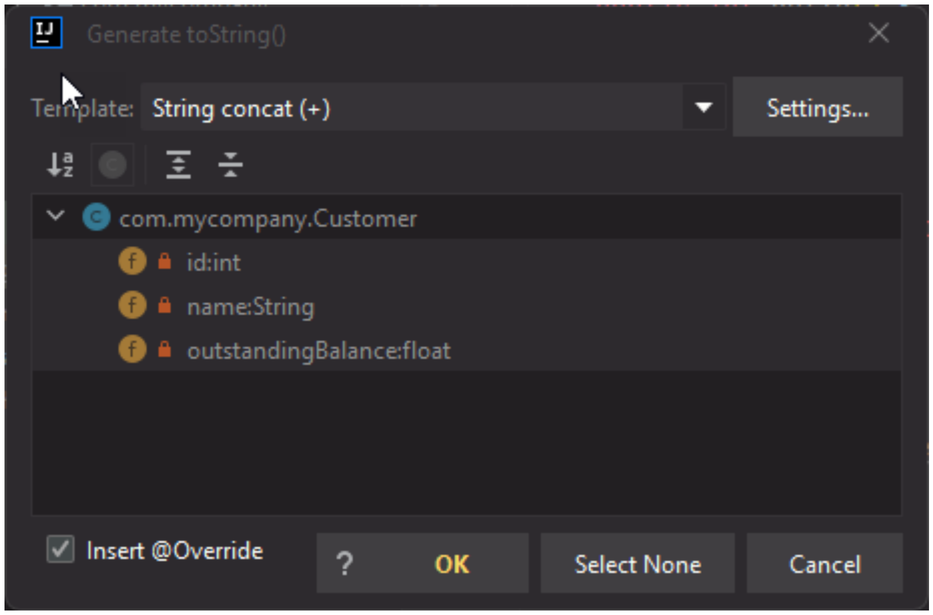
- Right click where you want your code to be located and select **Generate -> toString()**



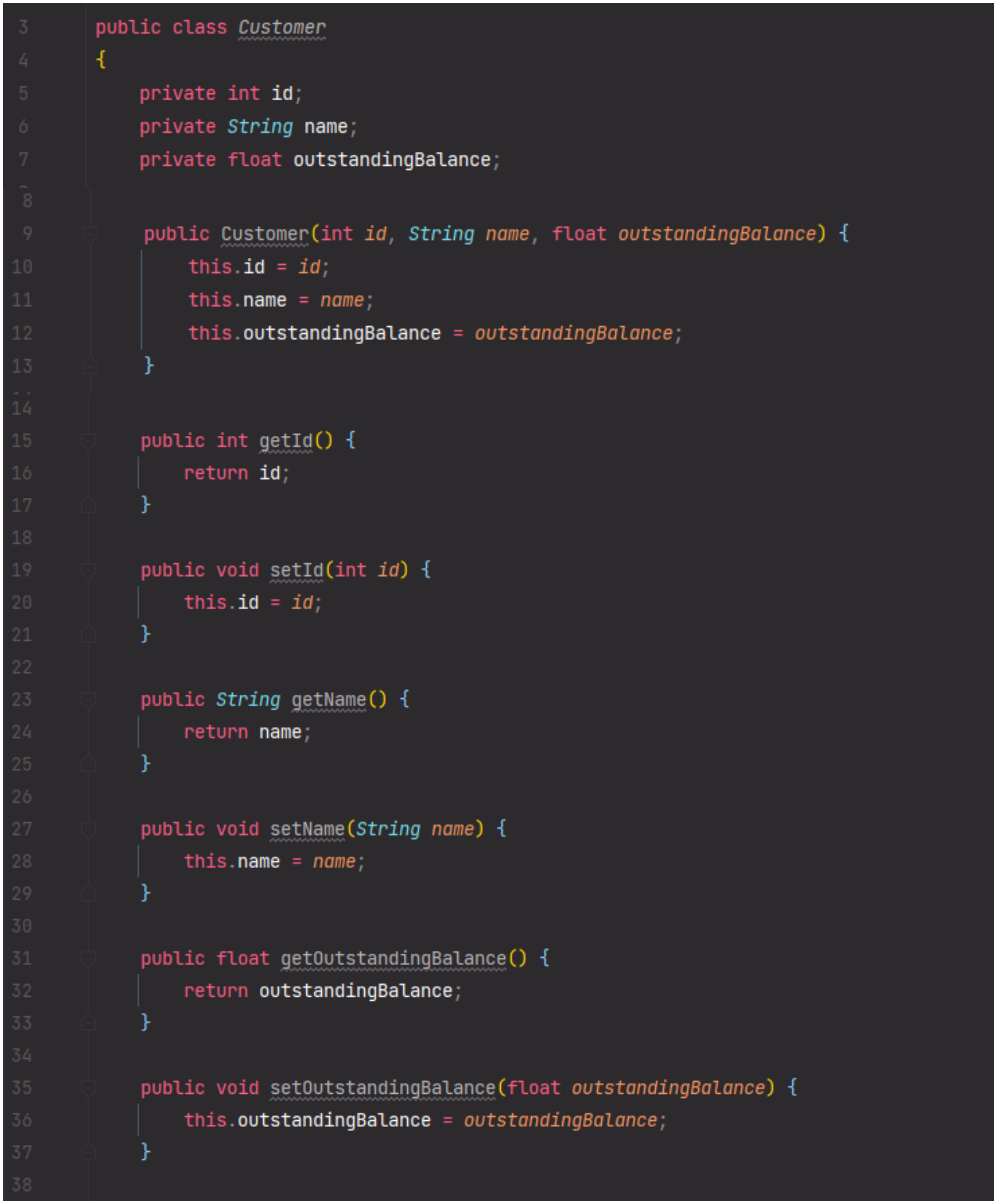
```
    }
    andingBalance() {
    gBalance;

    ndingBalance(float outstandingBalance) {
    alance = outstandingBalance;

}
```

Selecting what goes into your toString()

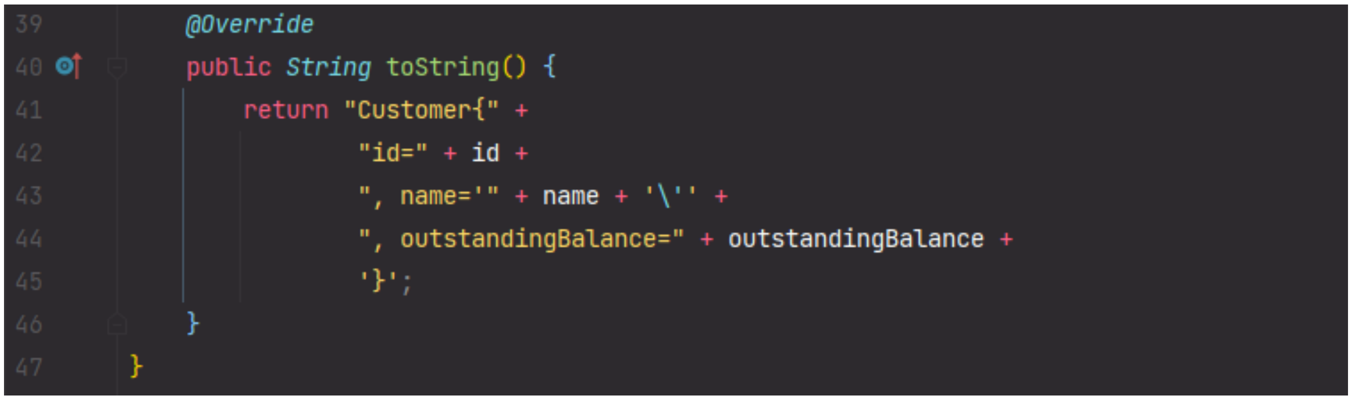
- **In the dialog box, select which fields are to be used when composing your `toString()` implementation.**



Admiring your Code

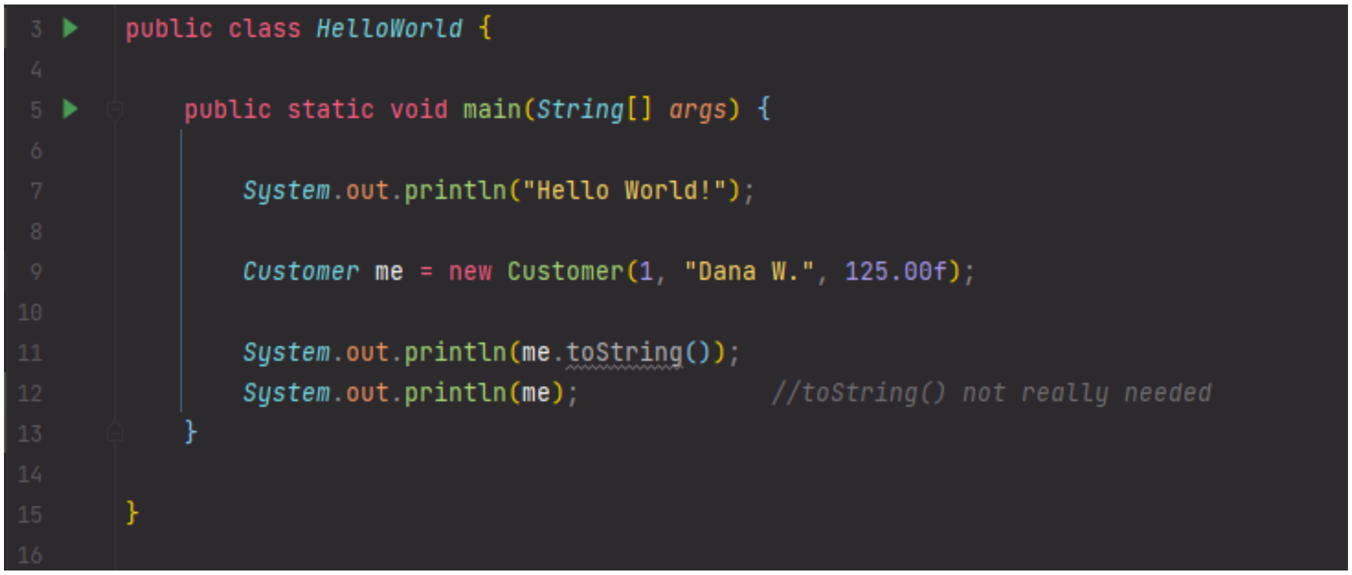


```
public class Customer
{
    private int id;
    private String name;
    private float outstandingBalance;

     public Customer(int id, String name, float outstandingBalance) {
         this.id = id;
         this.name = name;
         this.outstandingBalance = outstandingBalance;
     }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public float getOutstandingBalance() {
        return outstandingBalance;
    }

    public void setOutstandingBalance(float outstandingBalance) {
        this.outstandingBalance = outstandingBalance;
    }

```


```java
    @Override
    public String toString() {
        return "Customer{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", outstandingBalance=" + outstandingBalance +
                '}';
    }
}
```

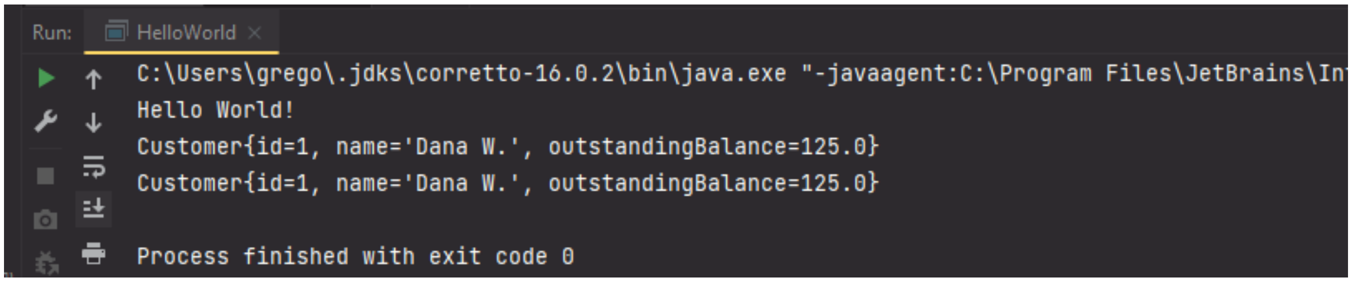
Using Customer in your Main Method

- **Since you have completed your `Customer` class, you can instantiate the Customer class within your main method**
  - The call to `toString()` is not required in the `println()` method call because it is called implicitly



```java
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World!");

        Customer me = new Customer(1, "Dana W.", 125.00f);

        System.out.println(me.toString());
        System.out.println(me);             //toString() not really needed
    }

}
```

- **Run the class as we have done before, and view the output from your main method**



```
C:\Users\grego\.jdks\corretto-16.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\In
Hello World!
Customer{id=1, name='Dana W.', outstandingBalance=125.0}
Customer{id=1, name='Dana W.', outstandingBalance=125.0}

Process finished with exit code 0
```

Exercises

EXERCISE 1

In this exercise, create a new project named CarInventory. Create it in the `workbook-2` folder. Make sure it is using Java 17!

Use your CarRental project as a guide. Re-create the `Vehicle` class by clicking File -> New -> Class.

In your `Vehicle` class, add the private class fields (`id`, `makeModel`, etc). Now use IntelliJ tools to:

1. create a parameterized constructor
2. create getters and setters for each property
3. code the `toString()` method

You can port over any remaining code you have by copying it to the clipboard and pasting it in the new IntelliJ project.

Check the Problems window to see if you have any errors. Once you have fixed them all, run the application by clicking the green arrow on the toolbar!

Finally… make sure each file is formatted and indented correctly by pressing **`CTRL+ALT+L`**

Commit and Push your code.

Section 1–2

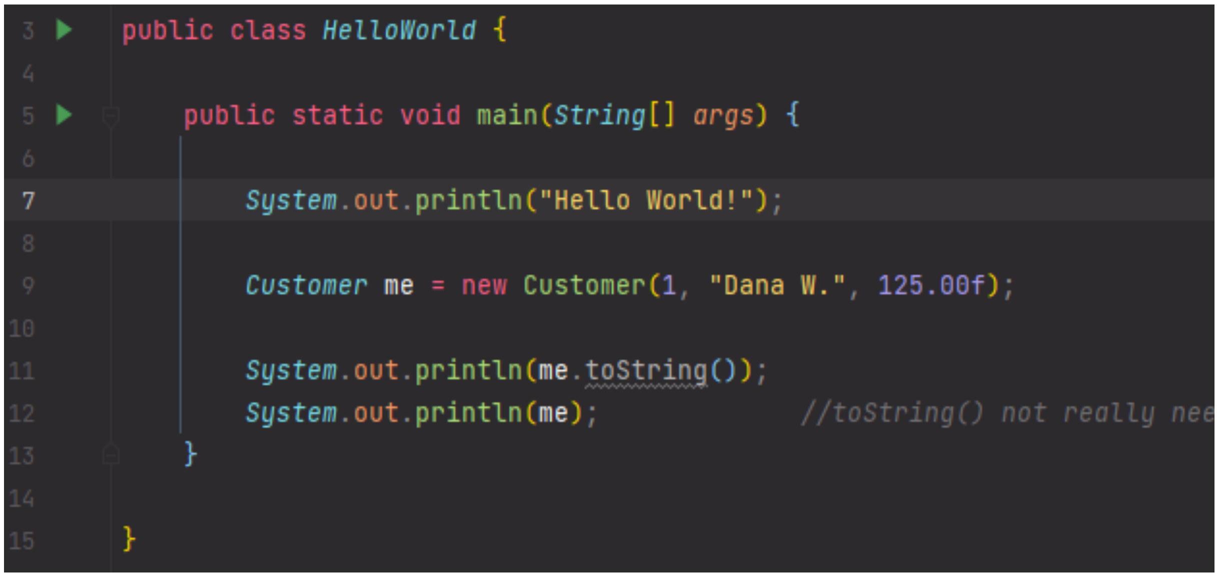
Building, Running and Debugging

Building a Project

- **IntelliJ saves all open files and builds the project each time it is run**
- **If there are any files with compile errors, the project will fail to compile and you will receive warning messages**
- **If you have multiple errors, begin by fixing the top error first**
  - Often fixing the first error will fix most or all other errors

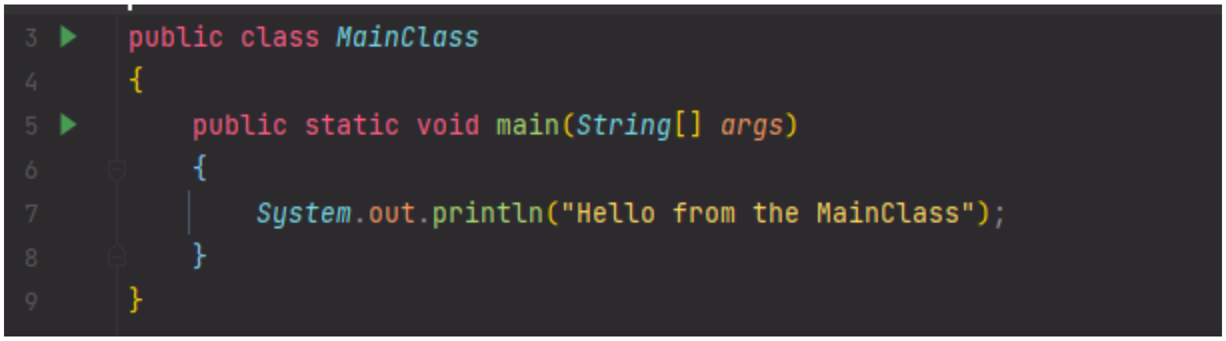
Running with a Run Configuration

- **There are some projects that you may have more than one class that has a main method which is runnable**
  - HelloWorld.java has a `static void main`



```java
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World!");

        Customer me = new Customer(1, "Dana W.", 125.00f);

        System.out.println(me.toString());
        System.out.println(me);             //toString() not really nee
    }

}
```

  - But now so does MainClass.java

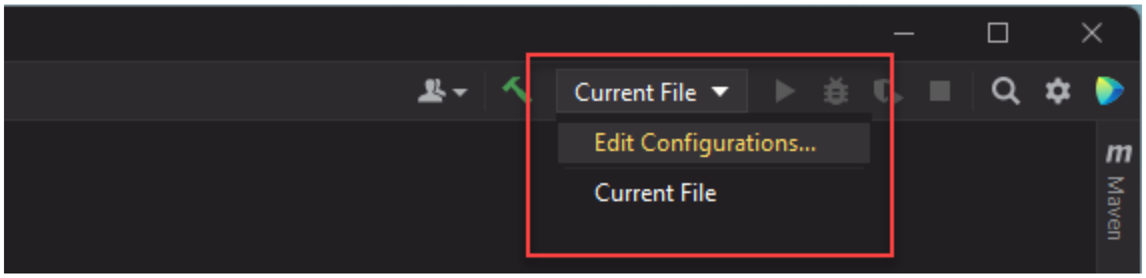


```java
public class MainClass
{
    public static void main(String[] args)
    {
        System.out.println("Hello from the MainClass");
    }
}
```

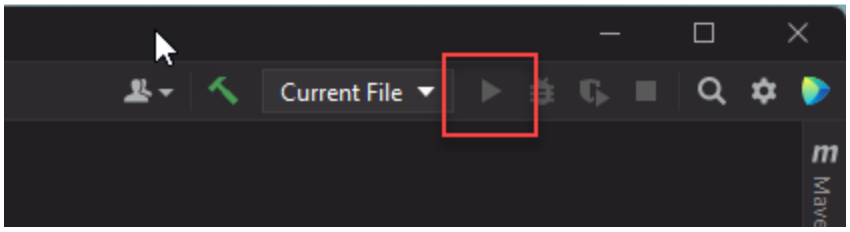
- **Only 1 main method can be the Entry Point each time you run your application**
  - One option is to click the Green Arrow next to the main method you want to run each time you run your application

- **To simplify running your application, you can create Run Configurations for your application**
  - In fact, when you click the Green Arrow to run your application, a Run Configuration is created for that Entry Point

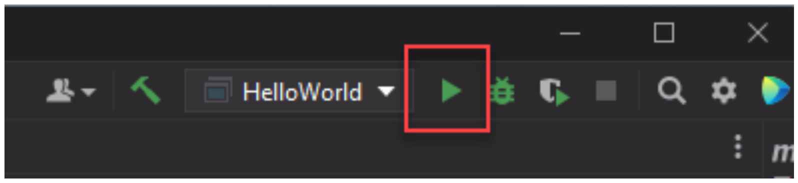
- **Configurations are managed in the top right corner of IntelliJ**



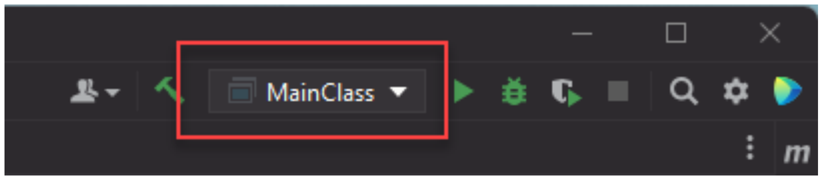
- **When you first open a project, no run configurations are set up**
  - Notice that the Play button is disabled



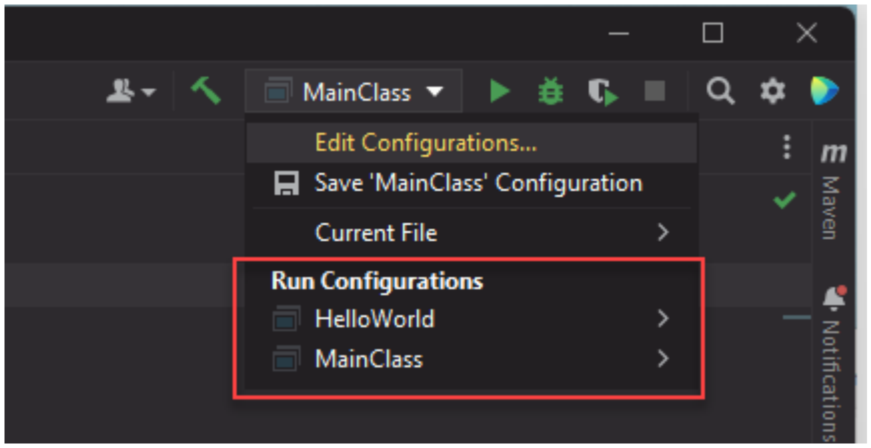
- **Once you click the Green Arrow of an entry point to run your application, that file becomes the default Run Configuration**
  - Now you do not need to run your application from the actual file, but you can run it from the Configuration menu



- **If you launch the application from another Entry Point then that configuration will become the default**

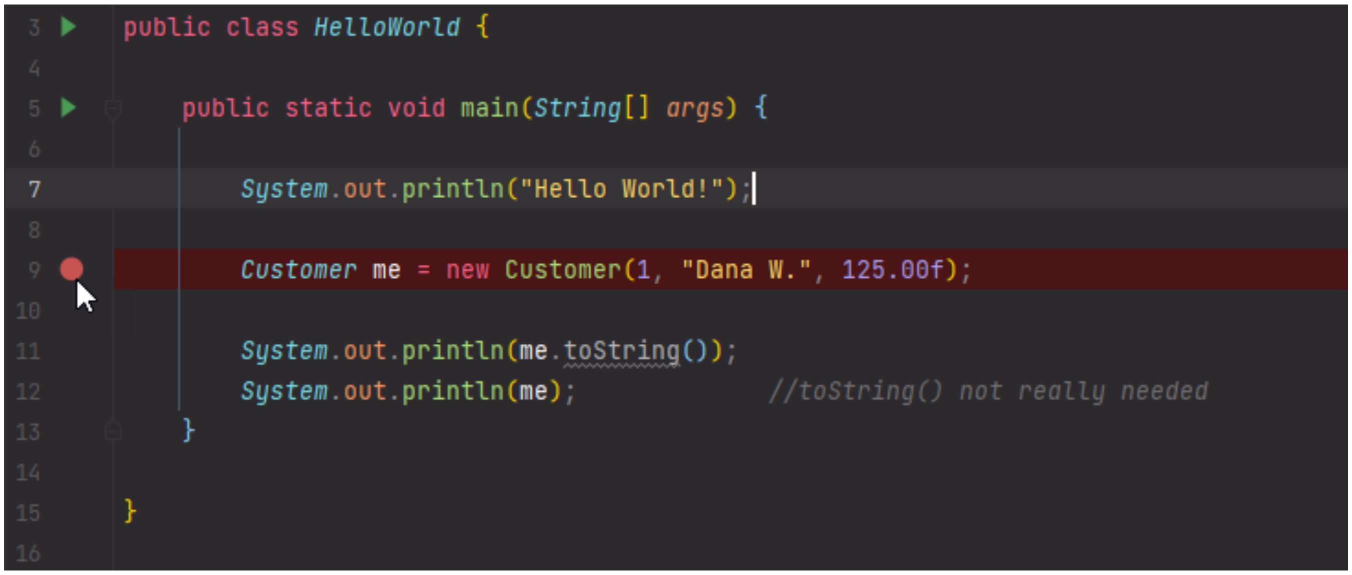


- **You can then select which Configuration you want to use when each time you run your application**



Setting a Breakpoint

- **To set a breakpoint, click in the gutter on the line you wish to break**
  - The gutter is to the right of the line number on the left hand side of your code



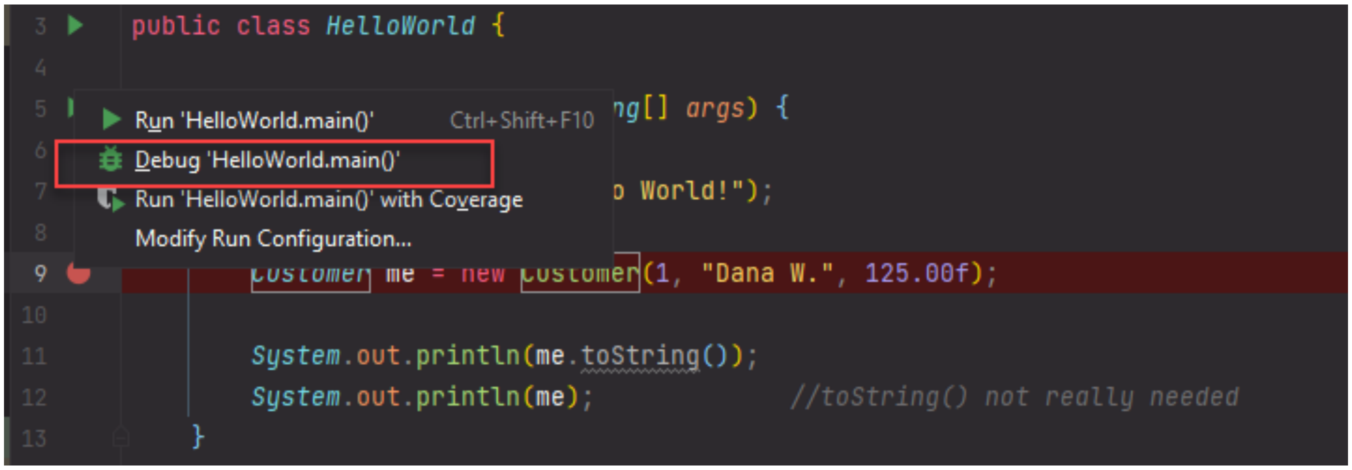
```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World!");

        Customer me = new Customer(1, "Dana W.", 125.00f);

        System.out.println(me.toString());
        System.out.println(me);            //toString() not really needed
    }

}
```

Why Set Breakpoints?

- Often your code will run, but the calculations are incorrect
- You need to quickly find the incorrect code so that we can fix it
- To do this we want to debug our code 1 line at a time and watch how each line of code affects our application
- A breakpoint allows us to specify which line of code should be the first line that we want to watch

Debugging a Project Overview

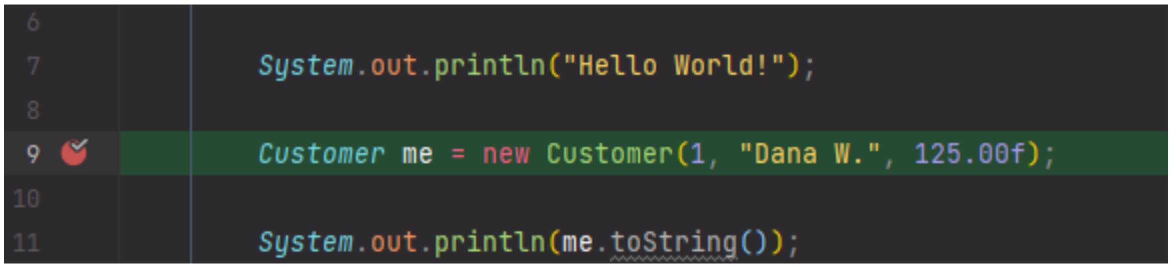
- **Debugging requires the following setup:**
  - Toggle on a breakpoint so the debugger can halt and you can review your code at that point
  - Run your project in debug mode



```
public class HelloWorld {

                             ng[] args) {

                             o World!");

        Customer me = new Customer(1, "Dana W.", 125.00f);

        System.out.println(me.toString());
        System.out.println(me);          //toString() not really needed
    }
```

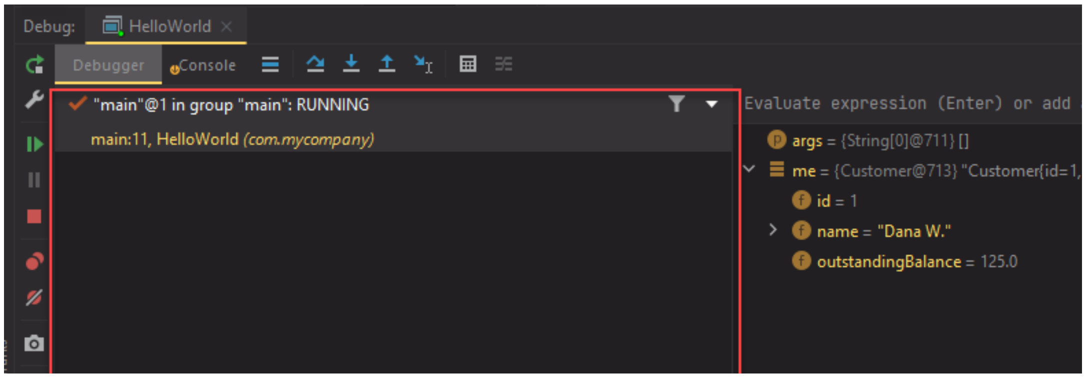
Run 'HelloWorld.main()' Ctrl+Shift+F10
Debug 'HelloWorld.main()'
Run 'HelloWorld.main()' with Coverage
Modify Run Configuration...

  - When the code gets to the breakpoint, the application pauses
  - You can see which line of code is waiting to be executed because it turns green



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

        Customer me = new Customer(1, "Dana W.", 125.00f);

        System.out.println(me.toString());
```

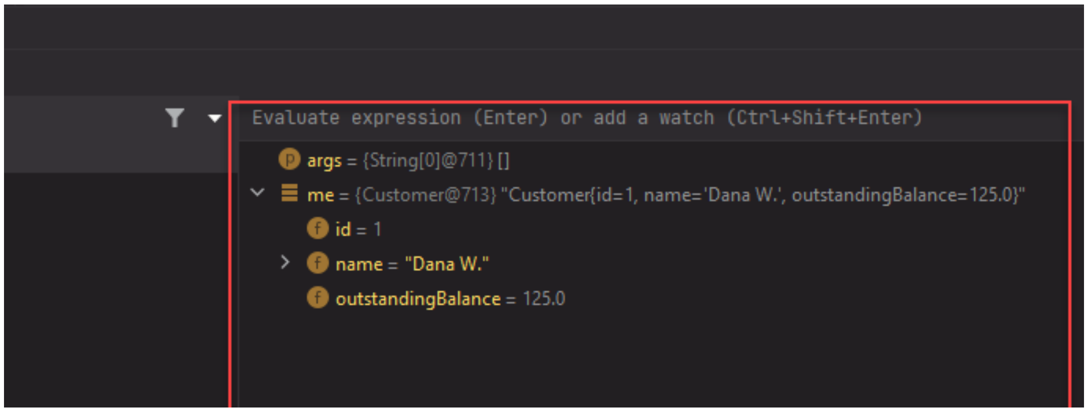
- **In Debug mode IntelliJ displays a few new windows and menus**
  - The Call Stack window on the left



```
"main"@1 in group "main": RUNNING
main:11, HelloWorld (com.mycompany)
```

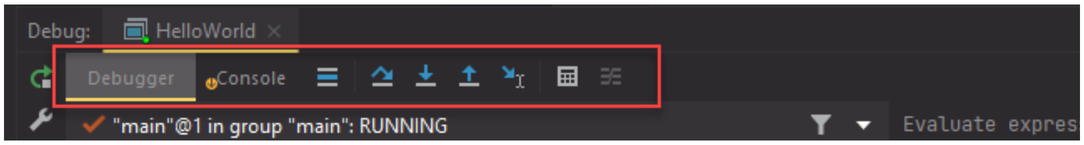
```
Evaluate expression (Enter) or add
args = {String[0]@711} []
me = {Customer@713} "Customer{id=1,
id = 1
name = "Dana W."
outstandingBalance = 125.0
```

  - The Watch window on the right



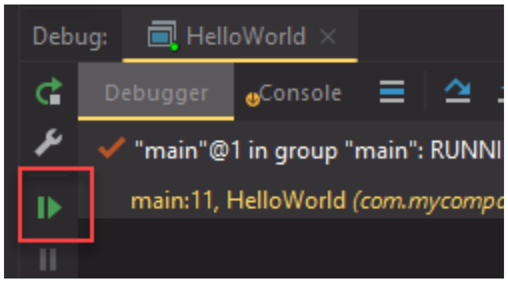
```
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
args = {String[0]@711} []
me = {Customer@713} "Customer{id=1, name='Dana W.', outstandingBalance=125.0}"
id = 1
name = "Dana W."
outstandingBalance = 125.0
```

  - A Debug Menu Bar at the top to step over, in and out of code



```
"main"@1 in group "main": RUNNING
```

  - A Continue button on the far left

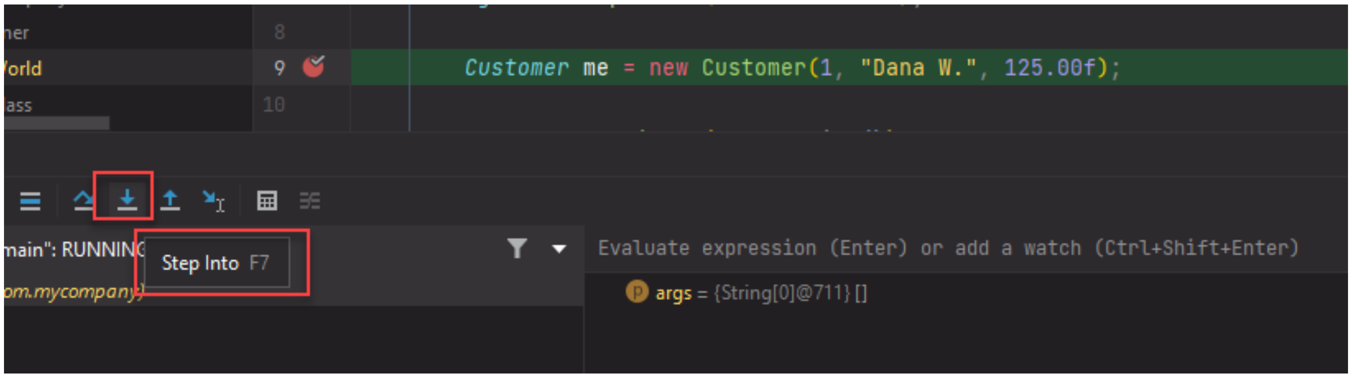


```
"main"@1 in group "main": RUNNI
main:11, HelloWorld (com.mycomp
```

Debugging Options

- **You have options to choose from when debugging:**
  - **Continue** - Continue with the rest of the debugging
  - **Pause** - Pause the debugging
  - **Stop** - Stop the debugging (the application ends)
  - **Disconnect** - Debugging is a process, you can disconnect from the debugging session (the application will continue to run)
  - **Step Over** - When you are on a method in the debugger, and you want to execute the method, but not step through it line by line
  - **Step Into** - When you are on a method in the debugger, and you want the debugger to step into the method
  - **Step Out** - Instruct the debugger to leave the function; all the way back out as if you are returning from the method

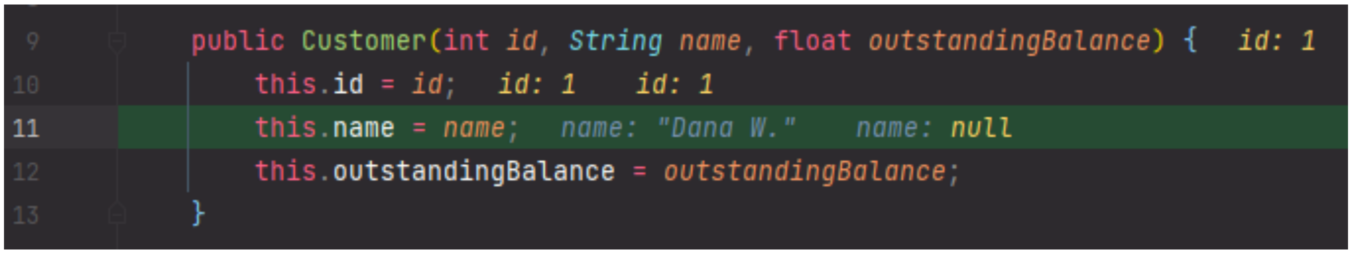
- **You can Step Into a function of constructor of the line being executed**



```
Customer me = new Customer(1, "Dana W.", 125.00f);
```

main": RUNNING
Step Into F7
om.mycompany
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
args = {String[0]@711} []

- **This results in stepping into the function and being able to watch each line of that function**

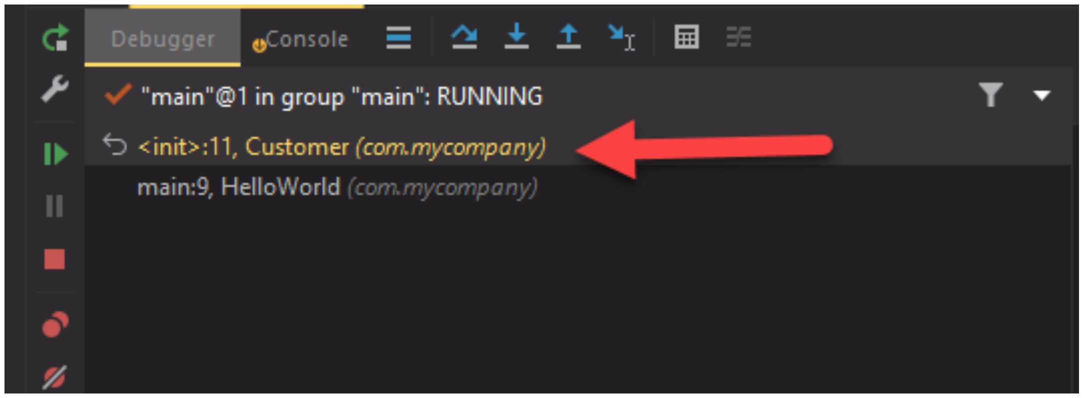


```
public Customer(int id, String name, float outstandingBalance) {   id: 1
    this.id = id;   id: 1    id: 1
    this.name = name;   name: "Dana W."    name: null
    this.outstandingBalance = outstandingBalance;
}
```

Viewing the Stack Frame

- When debugging, the left most window is the call stack frame

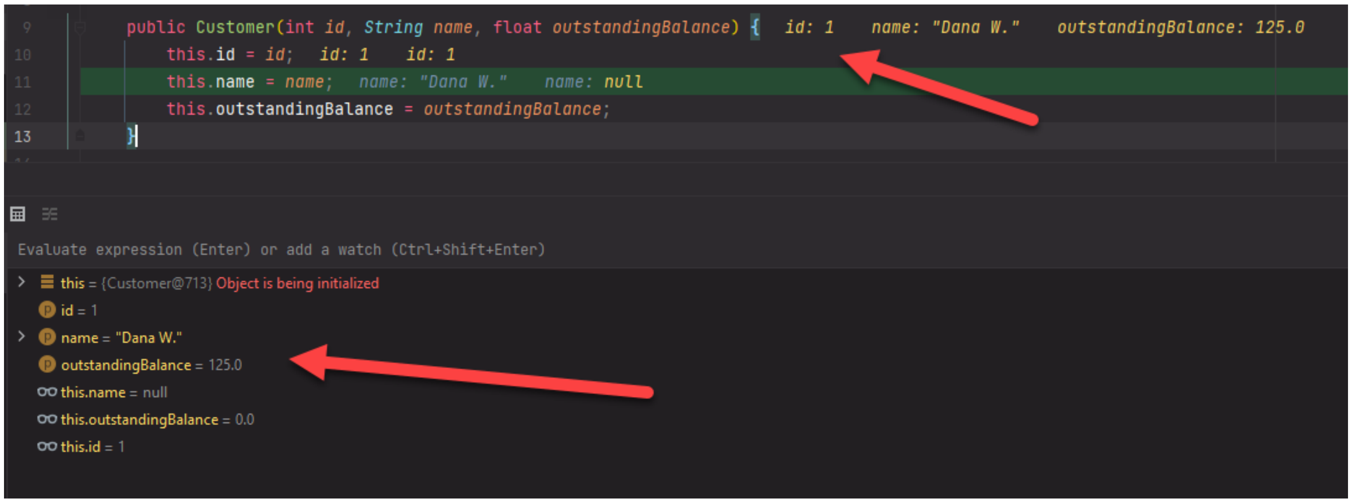
- Here you can view the call stack, to analyze what classes and methods you have visited to get to this point.



```
"main"@1 in group "main": RUNNING
<init>:11, Customer (com.mycompany)
main:9, HelloWorld (com.mycompany)
```

Viewing Variables during Debugging

- **In the right window you can view what variables are currently being used**
- **This gives you an opportunity to understand why certain decisions are being made within your code**
- **You can also hover a variable during your debugging session to view the value**
  - IntelliJ actually also displays the current values of each variable inline as you are debugging

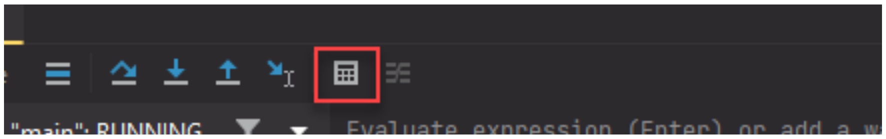


```
public Customer(int id, String name, float outstandingBalance) {   id: 1    name: "Dana W."    outstandingBalance: 125.0
    this.id = id;   id: 1    id: 1
    this.name = name;   name: "Dana W."    name: null
    this.outstandingBalance = outstandingBalance;
}
```

```
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
> this = {Customer@713} Object is being initialized
  id = 1
> name = "Dana W."
  outstandingBalance = 125.0
  this.name = null
  this.outstandingBalance = 0.0
  this.id = 1
```

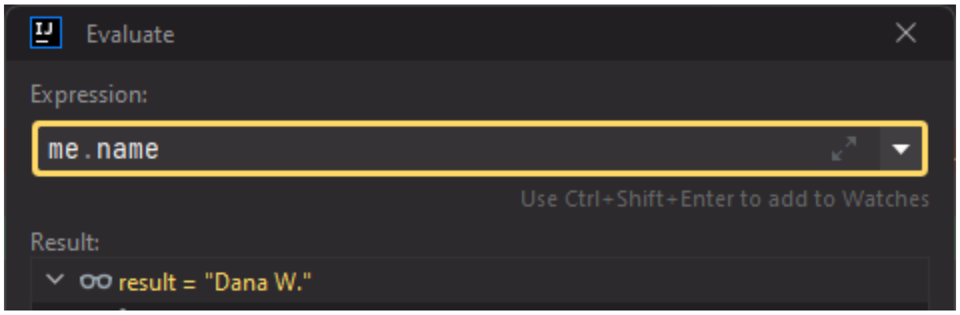
Viewing Expression During Debugging

- When debugging, you can enter expressions that you would like to have evaluated.
- You can add expressions at any time. This includes object references, or even mathematical expression
- Open the expressions window



```
"main": RUNNING        Evaluate expression (Enter) or add a w
```

- Add any expression you wish to evaluate



Evaluate

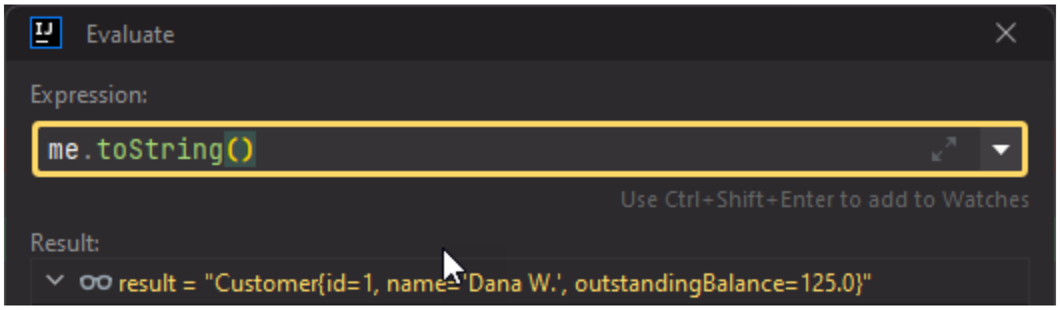
Expression:

```
me.name
```

Use Ctrl+Shift+Enter to add to Watches

Result:

```
result = "Dana W."
```



Evaluate

Expression:

```
me.toString()
```

Use Ctrl+Shift+Enter to add to Watches

Result:

```
result = "Customer{id=1, name='Dana W.', outstandingBalance=125.0}"
```

Exercises

The goal of this exercise is to get familiar with the debugging tools available in IntelliJ. When you are developing you will frequently run into logic errors, and you will need to determine what line(s) of code are incorrect. The debug tools are invaluable.

Whenever you run your application, you should get in the habit of using debug instead of run to test the code.

EXERCISE 2 (optional)

In this exercise, you will write code to see if a number is a palindrome. A palindromic number reads the same both ways.

The largest palindrome made from the product of two 2-digit numbers is 9009

- 91 × 99 = 9009 (9009 is the same backwards as forwards so it is a palindrome).

Create a new project in your `workbook-2` directory named PalindromeProduct

Your goal is to find the largest palindrome made from the product of two 3-digit numbers.

You'll need nested loops that each go from 1 to 999. Multiply the two loop counters together. Then convert your potential palindrome number to a `String` using `String.valueOf()`, make a copy of the String, and reverse it. If the two strings are the same, it's a palindrome!

The real trick is keeping only the largest palindrome that you find!

Whether you are successful or unsuccessful, using debugging techniques to view your code running step by step!

Commit and Push your code.