**Lesson 1**

OOP: driven by modeling the code around objects

Objects have Fileds, and are capable of performing actions (methods)

Primitive variables – primitive datatypes e.g int,float etc

Objects are enhanced datatypes

the largest value for an integer is 2,147,483,647, so a 10 digit number that starts with a 9 for example cannot be stored in a single integer.

Access modifiers : private, public

Any helper methods or methods that are only to be used internally should be set to private, all other methods should be public

**Fields**

The fields of an object are all the data variables that make up that object. They are also sometimes referred to as **attributes** or **member variables**.

These fields are usually made up of primitive types like integers or characters, but they can also be objects themselves.

For example a book object may contain fields like title, author and numberOfPages.

Then a library object may contain a field named books that will store all book objects in an array.

**Accessing fields:**

Accessing a field in an object is done using the dot modifier ‘.’

For example, if we had an object called book that contains these fields:

String title;

String author;

**int** numberOfPages;

To access the title field you would use

book.title

This expression is just like any other string, which means you can either store it in a string variable:

String myBookTitle = book.title;

Or use it directly as a string itself and perform operations like printing it:

System.out.println(book.title);

**Setting Fields**

You can also change a field’s value. Say you want to set the number of pages in a book to 234 pages:

book.numOfPages = 234;

# Methods

You might have also noticed that running actions in objects look very much like calling a function. That’s because that’s exactly what it is.

Methods in Java are functions that belong to a particular object. When we get to creating our own object types later in this lesson we will be creating methods the same way we used to created functions.

### Calling a method

To use a method you call it (just like calling a function). This is also done using the dot modifier .

Methods, just like any function can also take in arguments. For Example: Assume that our book object has a method called setBookmark that takes the page number as a parameter:

**void** **setBookmark**(**int** pageNum);

If you wanted to set a bookmark at page 12, you can call the method and pass in the page number as an argument:

book.setBookmark(12);

## Summary

**Fields** and **Methods** together are what make an object useful, fields store the object's data while methods perform actions to use or modify those data.

However some objects might have no fields and are just made up of a bunch of methods that perform various actions.

Other objects might only have fields that act as a way to organize storing data but not include any methods!

## Next Step

Now that we’ve seen how to use objects and access their fields as well as call their methods, let’s set up your computer so you can start using objects straight away.

**Classes** and **Objects** are two different terms and should not be used interchangeably, they can sometimes seem like they both refer to the same thing but each has a different meaning.

Here's a comparison that illustrates when to use which:

|  |  | **Class** | **Object** |
| --- | --- | --- | --- |
| **What:** |  | A Data Type | A Variable |
| **Where:** |  | Has its own file | Scattered around the project |
| **Why:** |  | Defines the structure | Used to implement to logic |
| **Naming convention:** |  | CamelCase (starts with an upper case) | camelCase (starts with a lower case) |
| **Examples:** |  | Country | australia |
|  |  | Book | lordOfTheRings |
|  |  | Pokemon | pikachu |

In summary, **objects** are to **Classes** what **variables** are to **Data types**.

# Strings

You've probably already noticed that (unlike all primitive types) Strings start with an upper case 'S'! That's because a String is in fact a class and not a primitive type

A String variable is made up of an array of characters (char []) as its field, but being an object means that it also offers some powerful methods like length() that counts and returns the number of characters in that array, and equals(String s) that compares the characters in this string with another string.

# Everything is an object in Java

Because Java is an OOP language, it includes classes that simply wrap around all the primitive types themselves to offer some extra functionality through their methods:

| **Class** | **Primitive type** |
| --- | --- |
| Integer | int |
| Long | long |
| Double | double |
| Character | char |
| String | char[] |

Each of those classes is made up of the corresponding primitive type as its field, but usually also comes with some powerful methods.

It also allows you to forget about primitive types and treat everything in Java as an object. However, it is still recommended to use primitive types when writing a simple piece of code.

**The main method**

A Java program can be as small as a single class, but usually a single program will be made up of tens or even hundreds of classes!

A good Java program is one that divides the logic appropriately so that each class ends up containing everything related to that class, and nothing more!

Classes would be calling each other's methods and updating their fields to make up the logic of the entire program all together!

BUT, where should the program start from exactly? In other words, if a method can call another method and that method can call another, which method will start this sequence the very first time?

The answer is the main method! It looks like this:

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

*// Start my program here*

}

Let's break it down:

* **public**: Means you can run this method from anywhere in your Java program (we will talk more about public and private methods later
* **static**: Means it doesn't need an object to run, which is why the computer starts with this method before even creating any objects (we will also talk more about static methods later on)
* **void**: Means the main method doesn't return anything, it just runs when the program starts, and once it's done the program terminates
* **main**: Is the name of the method
* **String [] args**: Is the input parameter (array of strings) which we will cover how to use it later in this lesson as well!

This main method is the starting point for any Java program, when a computer runs a Java program, it looks for that main method and runs it.

Inside it you can create objects and call methods to run other parts of your code. And then when the main method ends the program terminates.

If this main method doesn't exist, or if there's more than one, the Java program won't be able to run at all!

The main method can belong to any class, or you can create a specific class just for that main method which is what most people do.

Let's have a look at an example next.

# Create your first method

Ok, now let's add the main method to our Main class.

Open the Main class and inside the class curly bracket start typing the definition of the main method:

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

}

Then, inside the main method, let's print a welcoming message "Hello world!"

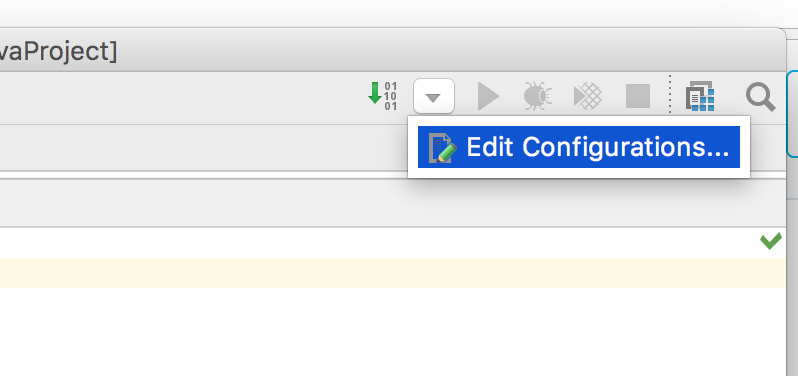
**public** **static** **void** **main**(String [] args){

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

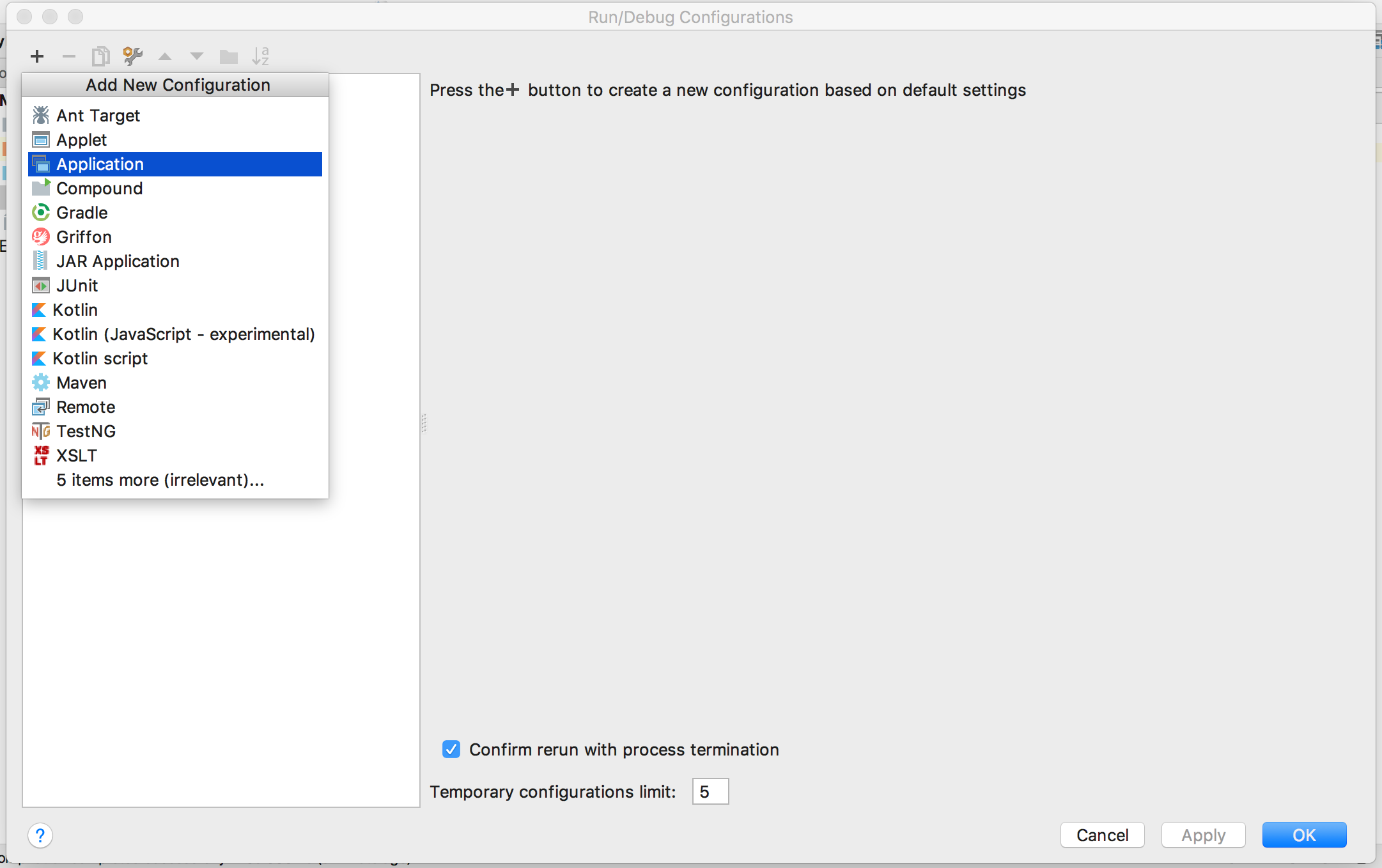
}

If the top right "run" button is not active, then you'll need to set up the configuration.

Click on the drop down menu button right next to the "run" button at the top right corner of your IDE, and then select **Edit Configuration**

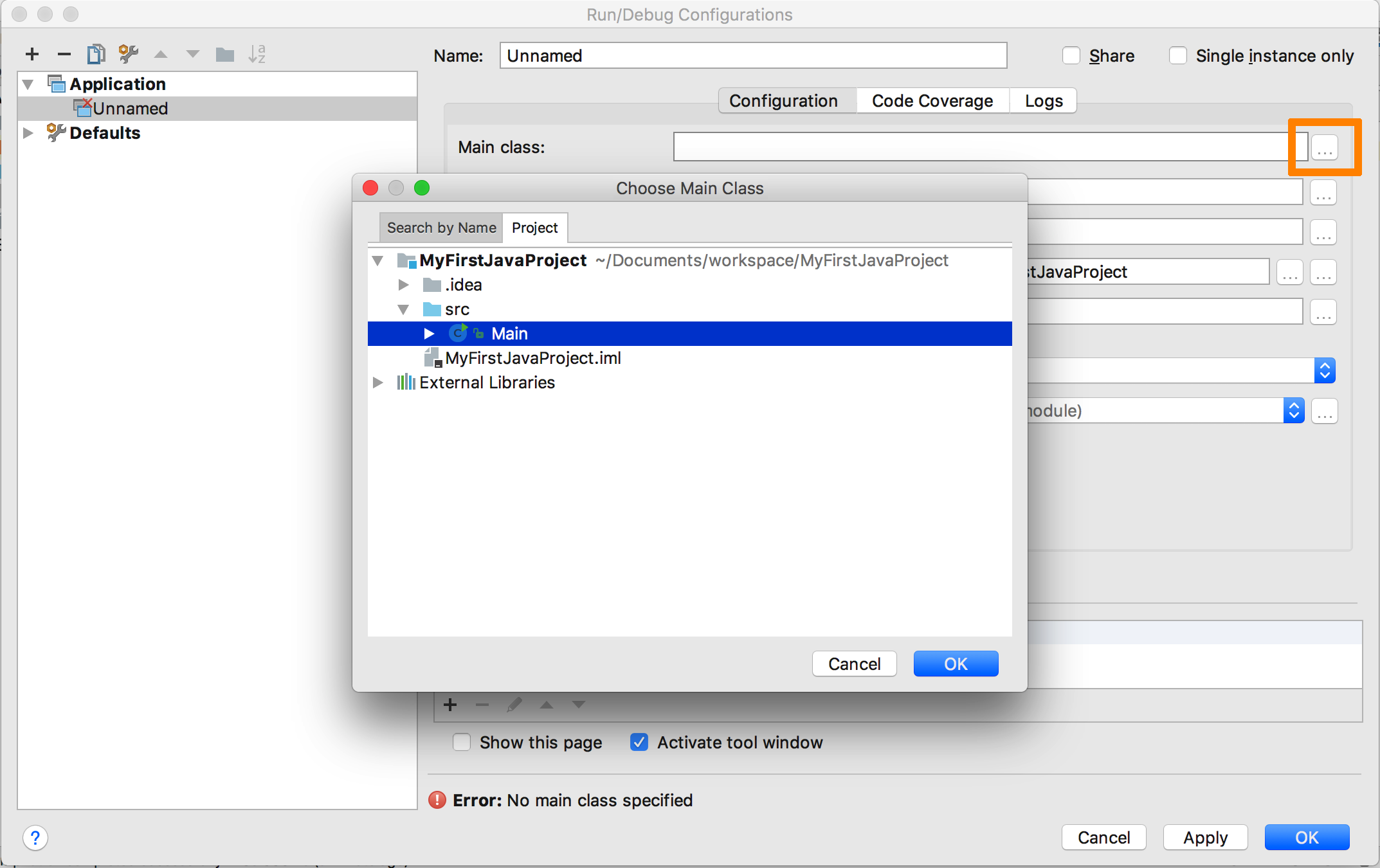
**[[](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)**

Then click on the + sign at the top left corner and select Application

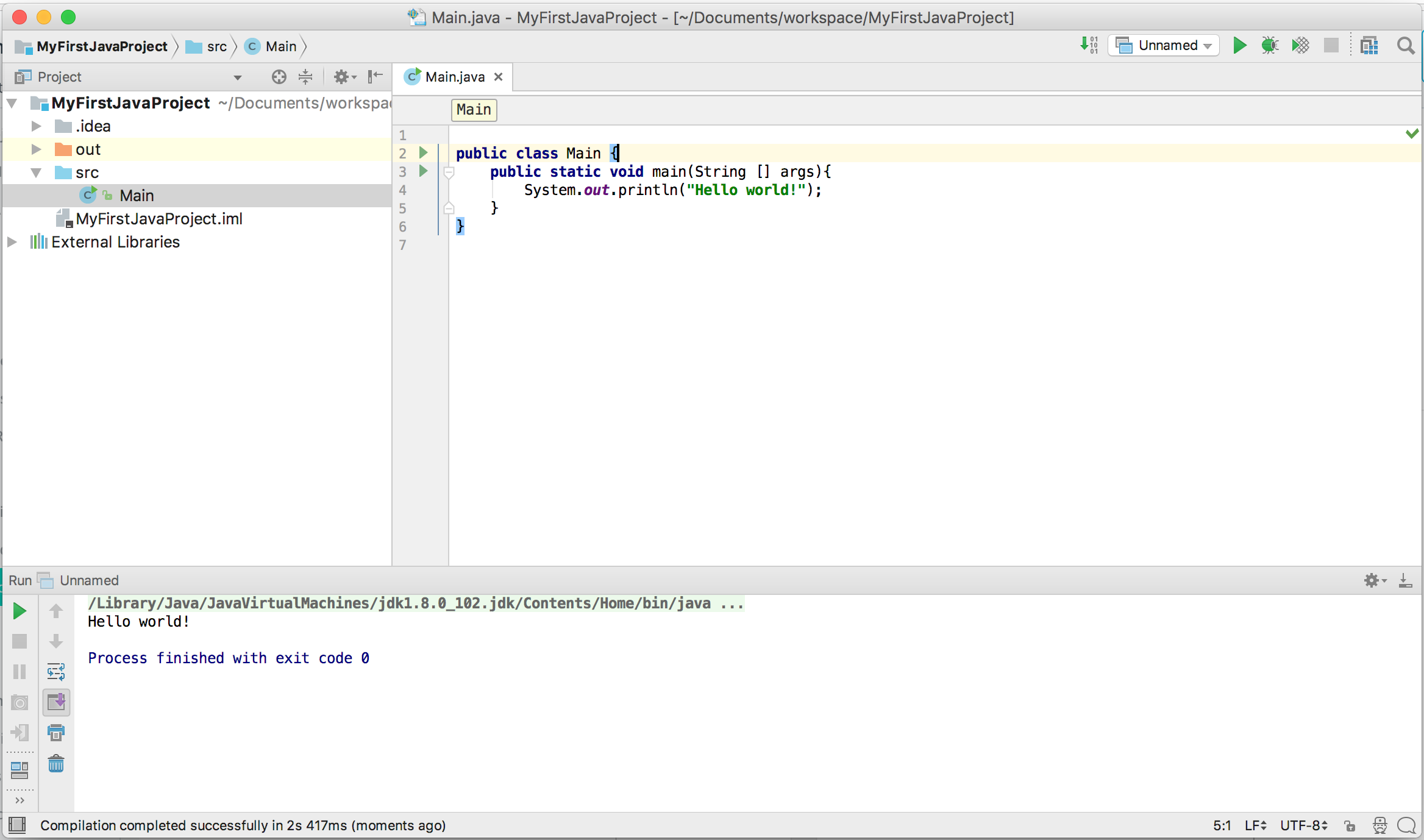
**[[](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)**

Then, you'll need to select the Main class that contains the main method for the IDE to know where to start.

To do so click on the three dots ... next to the Main class edit and then browse to the **Project** tab and select the **Main** class we just created.

**[[](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)**

Once you click OK and Apply your changes, the project is now configured and ready to run!

**[[](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)**

When you run the project you can see that it's first compiling the code and then (if no errors exists) it will run and show the output in this bottom panel down here!

You can see that it has indeed printed the welcoming message "Hello World!"

Great!

# Constructors

Constructors are special types of methods that are responsible for creating and initializing an object of that class.

### Creating a constructor

Creating a constructor is very much like creating a method, except that:

1. Constructors don't have any return types
2. Constructors have the same name as the class itself

They can however take input parameters like a normal method, and you are allowed to create multiple constructors with different input parameters.

Here's an example of a simple constructor for a class called Game

**class** **Game**{

...

*// Constructor*

Game(){

*// Initialization code goes here*

}

...

}

#### Default constructor

A Default constructor is one that doesn't take any input parameters at all!

It's optional, which means if you don't create a default constructor, Java will automatically assume there's one by default that doesn't really do anything.

However, if the class has fields that need to be initialized before the object can be used, then you should create one that does so.

For example, assume we have a class Game that has an integer member field score, we'd like to make sure that any object of type Game will start with the score value set to 0. To do so, we need to create a default constructor that will initialize the mScorefield

**class** **Game**{

**int** mScore;

*// Default constructor*

Game(){

*// Initialize the score here*

mScore = 0;

}

}

#### Parameterized constructor

As we've mentioned earlier, a constructor can also take input parameters.

Let's assume that some games start with a positive score value and not just 0, that means we need another constructor that takes an integer parameter as an input, and uses that to initialize the score variable.

**class** **Game**{

**int** score;

*// Default constructor*

Game(){

score = 0;

}

*// Constructor by starting score value*

Game(**int** startingScore){

score = startingScore;

}

}

### Accessing a constructor

Unlike normal methods, constructors cannot be called using the dot . modifier, instead, every time you create an object variable of a class type the appropriate constructor is called. Let's see how:

##### The new keyword

To create an object of a certain class, you will need to use the new keyword followed by the constructor you want to use, for example:

Game tetris = **new** Game();

This will create an object called tetris using the default constructor (i.e. tetris will have an initial score value of 0)

To create a game that is initialized with a different starting score you can use the second constructor:

Game darts = **new** Game(501);

##### The null keyword

If you do not initialize an object using the new keyword then its value will be set to something called null. null simply refers to an empty (uninitialized) object. nullobjects have no fields or methods, and if you try to access a null object's field or call its method you will get a runtime error.

In some cases, you might want to explicitly set an object to null to indicate that such object is invalid or yet to be set. You can do so using the assignment operation:

Game darts = **null**;

### Why multiple constructors?

You might be wondering why do we still need to keep the default constructor now that we have another constructor that can create a game object with any starting score value (including 0)?

Good point, however, it's considered a good practice to always include a default constructor that initializes all the fields with values that correspond to typical scenarios. Then you can add extra parameterized constructors that allow for more customization when dealing with less common cases.

### But you said the default constructor is optional!

As we've mentioned earlier, you have the option to not create any constructors at all! The class will still be valid and you will be able to create objects using the same syntax of a default constructor. Exactly as if you had created an empty default constructor.

However, this privilege goes away once you create any constructor of your own! Which means if you create a parameterized constructor and want to also have a default constructor, you will have to create that default constructor yourself as well.

# Self Reference

Sometimes you'll need to refer to an object within one of its methods or constructors, to do so you can use the keyword this.

**this** is a reference to the current object — the object whose method or constructor is being called. You can refer to any field of the current object from within a method or a constructor by using this.

## Using this with a Field

The most common reason for using the this keyword is because a field has the same name as a parameter in the method or constructor

For example, if a Position class was written like this

**class** **Position** {

**int** row = 0;

**int** column = 0;

*//constructor*

Position(**int** r, **int** c) {

row = r;

column = c;

}

}

A more readable way would be to use the same names (row & column) for the constructor parameters which means you will have to use the this keyword to seperate between the fields and the paramters:

**class** **Position** {

**int** row = 0;

**int** column = 0;

*//constructor*

Position(**int** row, **int** column) {

**this**.row = row;

**this**.column = column;

}

}

In the second snippet, the constructor Position accepts the parameters row and column, but the class Position also includes two fields with the exact same name.

Using this.row compared to row means that we are referring to the **field** named row rather than the input parameter.

There are plenty more uses for the keyword this that you can check out [**here**](https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html), but they are slightly outside the scope of this course.

# The Contacts Manager

Assume you're writing a Java program that's responsible for storing and managing all your friends' contact information.

We'll start by creating a class that's responsible for storing all contact information of a single person, it will look something like this:

**class** **Contact**{

String name;

String email;

String phoneNumber;

}

All fields, no methods, since a contact object itself won't be "doing" much actions itself in the scope of this program, it's merely a slightly more advanced data type that can store a few strings in 1 variable.

**Note:** Noticed how we used a String to store the phone number instead of using int! Can you think of a reason why?

### QUIZ QUESTION

Why is it a good idea to use a String variable to store a phone number than using an integer:

(There may be more than 1 correct answer)

* 

Because you can't compare integers

* Because phone numbers start with 0's and integers can't store leading 0's
* Because the largest integer is smaller than a typical phone number
* 

Because a String requires less memory than an integer

SUBMIT

Next, let's create the class that will be in charge of adding and searching for contacts. Since it will be managing all the contacts, I'll call it ContactsManager:

**class** **ContactsManager** {

}

This class will be storing the contacts in an array, which means one of its fields will be an array of Contacts, another field will be an int representing the number of friends added to the array, this int will help us know where in the array was the last contact added so we can continue to add more contacts into the array later as we will see.

This is what the class will look like after adding the fields

**class** **ContactsManager** {

*// Fields:*

Contact [] myFriends;

**int** friendsCount;

}

Okay, now let's create a default constructor that will initialize those fields.

**class** **ContactsManager** {

*// Fields:*

Contact [] myFriends;

**int** friendsCount;

*// Constructor:*

ContactsManager(){

**this**.friendsCount = 0;

**this**.myFriends = **new** Contact[500];

}

}

The friendsCount starts from 0 and will increment every time we add a new contact later.

The Contact array myFriends (just like any array) needs to be initialized using the newkeyword and we chose to reserve enough space in the array to store up to 500 contacts.

Next, let's start adding methods to the ContactsManager class that allows adding and searching for contacts in the array.

# The ContactsManager class methods

The first method we will create in the ContactsManager class is the addContactmethod which will add a Contact object to the Contact array myFriends:

**void** **addContact**(Contact contact){

myFriends[friendsCount] = contact;

friendsCount++;

}

The method addContact takes a Contact object as an input parameter, and uses the friendsCount value to fill that slot in the array with the contact that was passed into the method.

Then, since we need to move that counter to point to the following slot in the array, we increment friendsCount using the increment operation ++

Now, let's add another method called searchContact that will search through the array using a name String and return a Contact object once a match is found:

Contact **searchContact**(String searchName){

**for**(**int** i=0; i<friendsCount; i++){

**if**(myFriends[i].name.equals(searchName)){

**return** myFriends[i];

}

}

**return** **null**;

}

This method loops over the array, and for each element myFriends[i] it compares the name field to the searchName value using this if statment:

**if**(myFriends[i].name.equals(searchName))

This if statement will evaluate to true if the searchName is equal to the name field in the Contact stored in myFriends[i]

If it was a match, the loop will return the matching Contact object myFriends[i]. Otherwise. it will return null indicating that it could not find that contact.

Putting all this together, our ContactsManager class will look like this:

**class** **ContactsManager** {

*// Fields:*

Contact [] myFriends;

**int** friendsCount;

*// Constructor:*

ContactsManager(){

friendsCount = 0;

myFriends = **new** Contact[500];

}

*// Methods:*

**void** **addContact**(Contact contact){

myFriends[friendsCount] = contact;

friendsCount++;

}

Contact **searchContact**(String searchName){

**for**(**int** i=0; i<friendsCount; i++){

**if**(myFriends[i].name.equals(searchName)){

**return** myFriends[i];

}

}

**return** **null**;

}

}

To be able to run this program, we need the main method, so let's create another class called Main that will hold this method:

**class** **Main** {

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

ContactManager myContactManager = **new** ContactManager();

}

}

This means that once this program runs, the main method will start which will create the ContactManager object myContactManager and thus ready to be used.

However, if you go ahead and run this program nothing will appear because we we haven't created the logic to ask the user for adding or searching contacts yet.

Later on in this course, we will see how to read input from the user to make this program more powerful.

**Lesson 2**

# File Scanner

Another way of accepting runtime input is through files, these files can be plain text files that the user creates with a very basic text editor (e.g. notepad on windows or TextEdit on macs).

A good example would be a Java program that loads a list of expenses from a text file (or excel sheet) and after some calculations prints a report of the total amount, average spendings, largest purchase etc.

To read a text file in Java you can also use the same Scanner class we used to read command line inputs, but instead of passing System.in as the argument you pass a File object which you can create by typing in the file name:

File file = **new** File("expenses.txt");

Scanner fileScanner = **new** Scanner(file);

Once the file scanner has been created, you read lines the same way we did earlier.

But since you would most likely want to load the entire file at once, you can check if the file still has more lines using hasNextLine method and then use this loop to read everything:

**while** (input.hasNextLine()) {

String line = input.nextLine();

*// Use that line to do any calculations, processing, etc ..*

}

**Command line arguments**

There's one more way a Java program can accept input from the user, and that is *before* they actually run the program!

Remember the declaration of the main method:

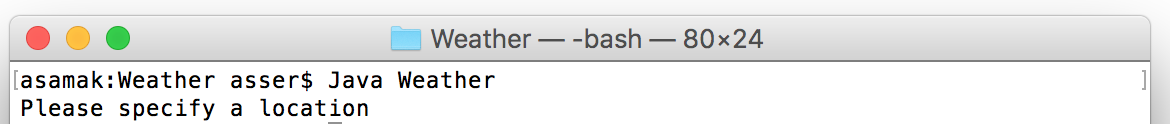
**public** **static** **void** **main**(String args[]){

}

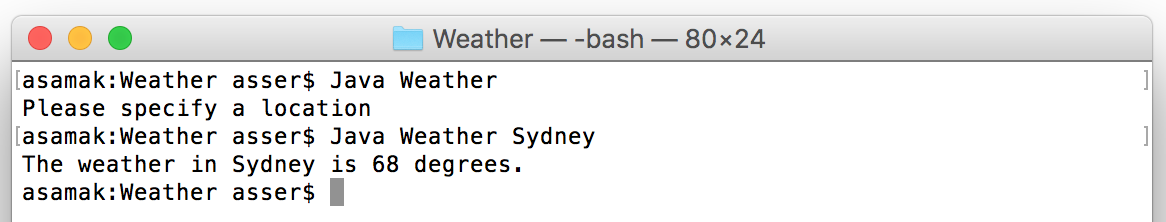
Notice that the method accepts a String array called args[] as an input parameter, but we never explicitly call the main method ourselves! So what is this String array and where does its value ever come from?

If you end up running the program from the command line, anything you type after the program name is considered an input argument.

For example, if we had a Java program called **weather** that prints today's weather, running it from the command line is as simple as typing in the program name:

**[[](https://classroom.udacity.com/courses/ud283/lessons/297a7f29-2c0d-4e79-863b-d7c83b4026c1/concepts/81702c78-2d94-4ee4-a945-bc05c4e66ce6)](https://classroom.udacity.com/courses/ud283/lessons/297a7f29-2c0d-4e79-863b-d7c83b4026c1/concepts/81702c78-2d94-4ee4-a945-bc05c4e66ce6)**

If we wanted the program to be more customizable, we could set it up to accept a city input and print the weather there. So to get the weather in Sydney you can type:

**[[](https://classroom.udacity.com/courses/ud283/lessons/297a7f29-2c0d-4e79-863b-d7c83b4026c1/concepts/81702c78-2d94-4ee4-a945-bc05c4e66ce6)](https://classroom.udacity.com/courses/ud283/lessons/297a7f29-2c0d-4e79-863b-d7c83b4026c1/concepts/81702c78-2d94-4ee4-a945-bc05c4e66ce6)**

The way this works is through the String [] args that's passed to the main method, which means inside the main method, the first String in that String array argscontains the value "Sydney".

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

**if**(args.length==0) {

System.out.println("Please specify a location");

}

**else** {

String location = args[0];

**int** temperature = 60 + (**int**)(Math.random()\*10);

System.out.println("The weather in "+location+" is "+ temperature);

}

}

You can loop through the args array and collect as many arguments as you want.

Feel free to read more information on how to read and use the [**command line arguments**](https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html)

Now it's time to try all of these input types in our project

# Catching exceptions

Inside the catch block you have the choice of either handling the situation quietly (like printing an error message or showing a warning popup)

**try**{

openFile("somefile.txt");

} **catch**(FileNotFoundException exception) {

*// Handle the situation by letting the user know what happened*

System.out.println("Cannot find that file");

}

OR you can elude the situation and just re-throw the exception:

**try**{

openFile("somefile.txt");

} **catch**(FileNotFoundException exception) {

*// Running away from the responsibility*

**throw** exception;

}

However, re-throwing the exceptions means that whoever is calling "this" method will now have to surround it with another try-catch block and do the same!

## Multiple catch statements

Since a try block can include more than one statement, and methods can actually throw more than one type of exceptions, you sometimes end up having to cater for different types of exceptions at the same time:

**try**{

openFile("somefile.txt");

array[index]++;

} **catch**(FileNotFoundException exception) {

*// Handle all the possible file-not-found-related issues here*

} **catch**(IndexOutOfBoundsException exception) {

*// Handle all the possible index-out-of-bounds-related issues here*

}

You can have as many catch statements as you need until you cover all possible Exception types that could be thrown inside the try statement.

## Catching all exceptions

Another option is to simply catch ALL exception types by catching the general type Exception, this means that whatever exception is thrown within this try-catch block, it will be caught and handled in this catch statement

**try**{

openFile("somefile.txt");

array[index]++;

} **catch**(Exception exception) {

*// Handle all the possible exceptions here*

}