**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!"

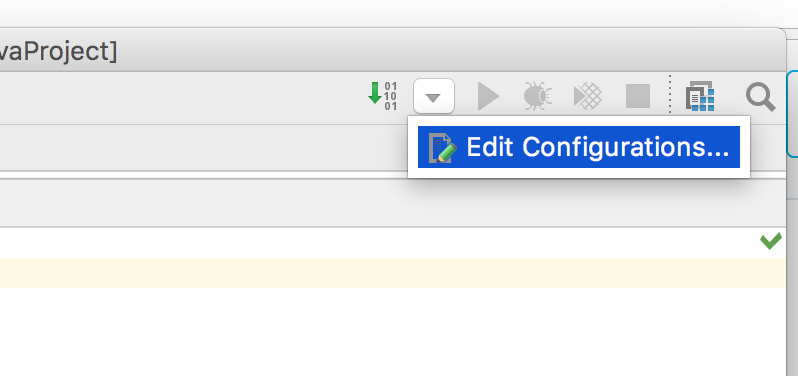
**public** **static** **void** **main**(String [] args){

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

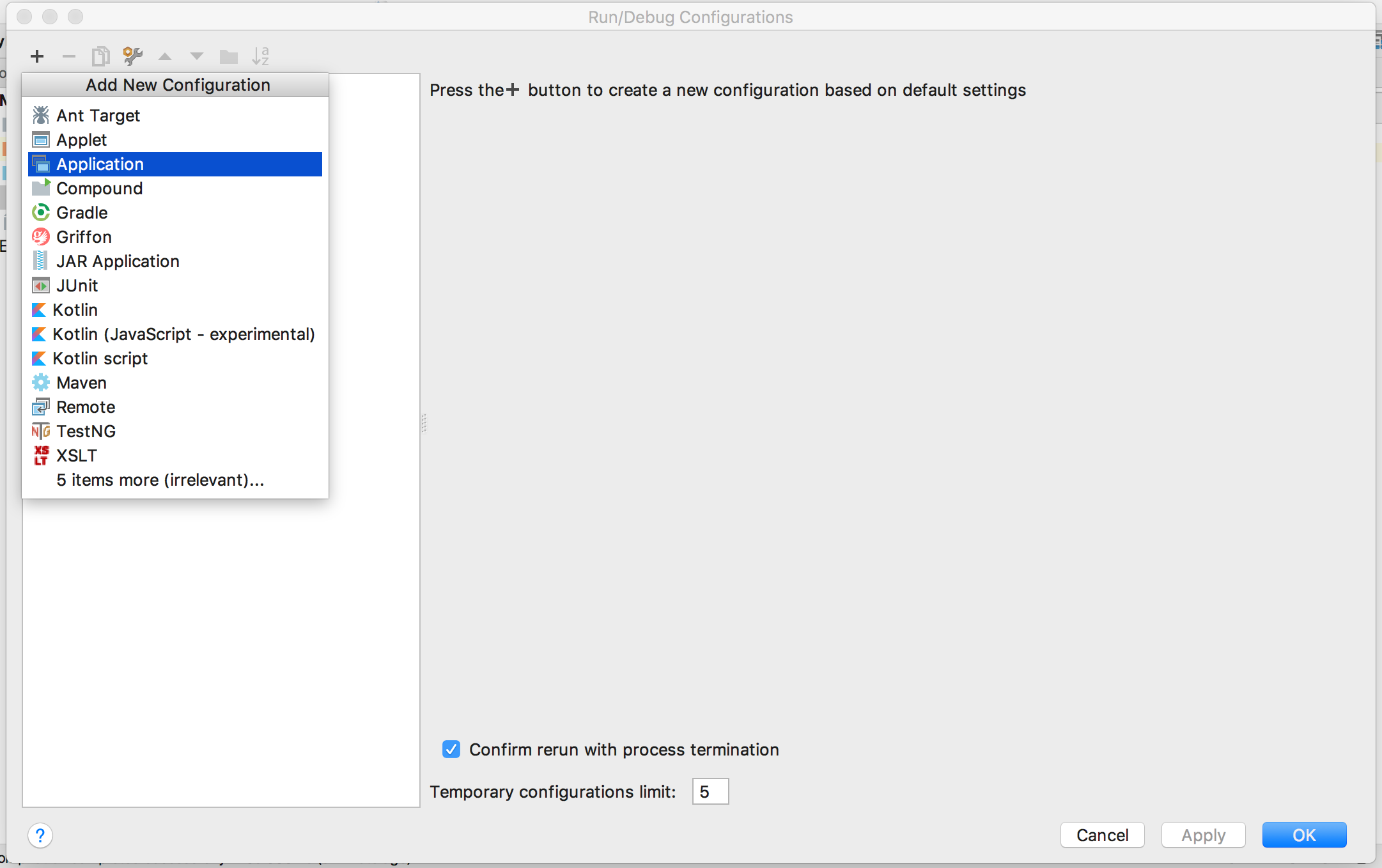
}

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**

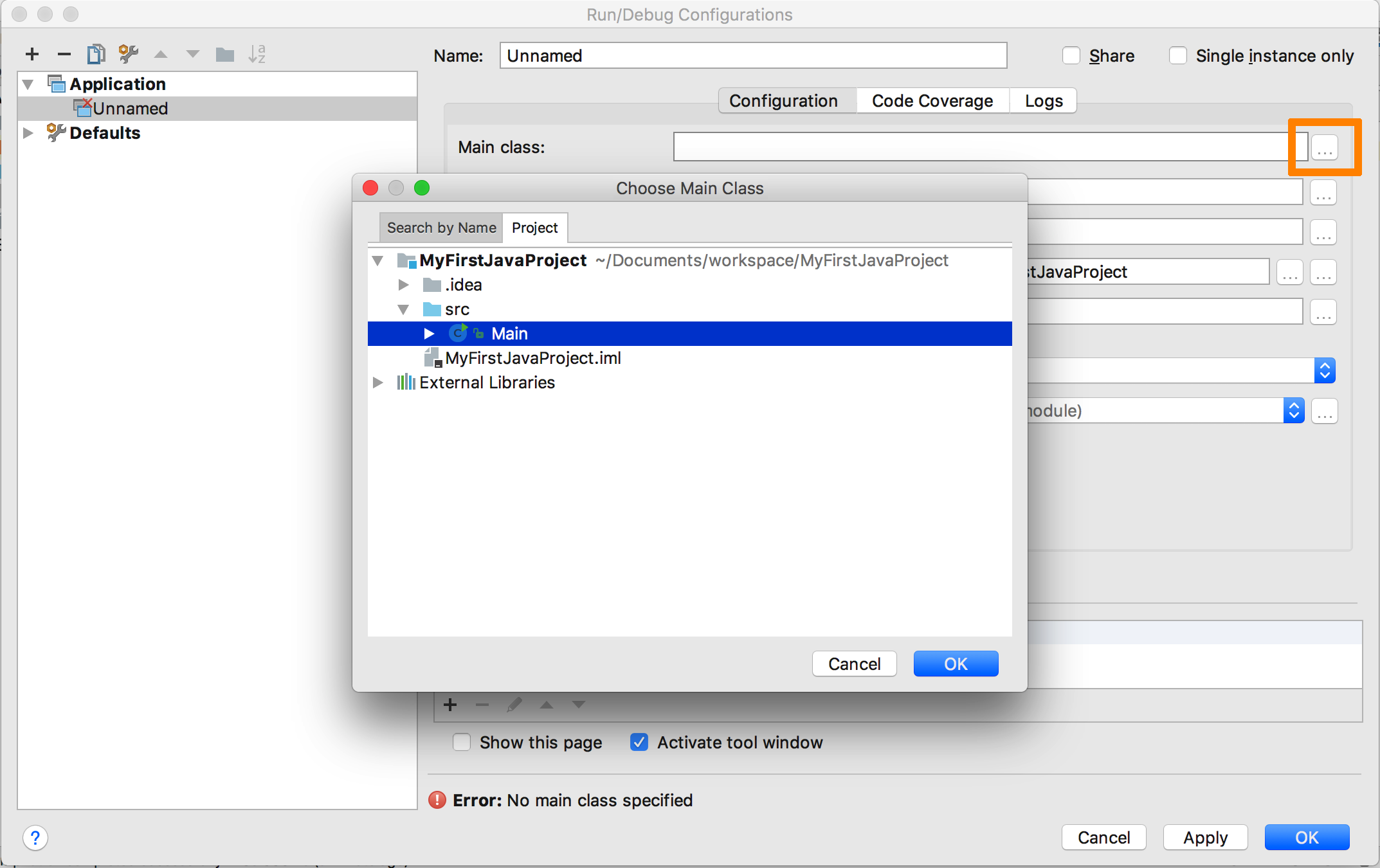
**[[](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)**

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

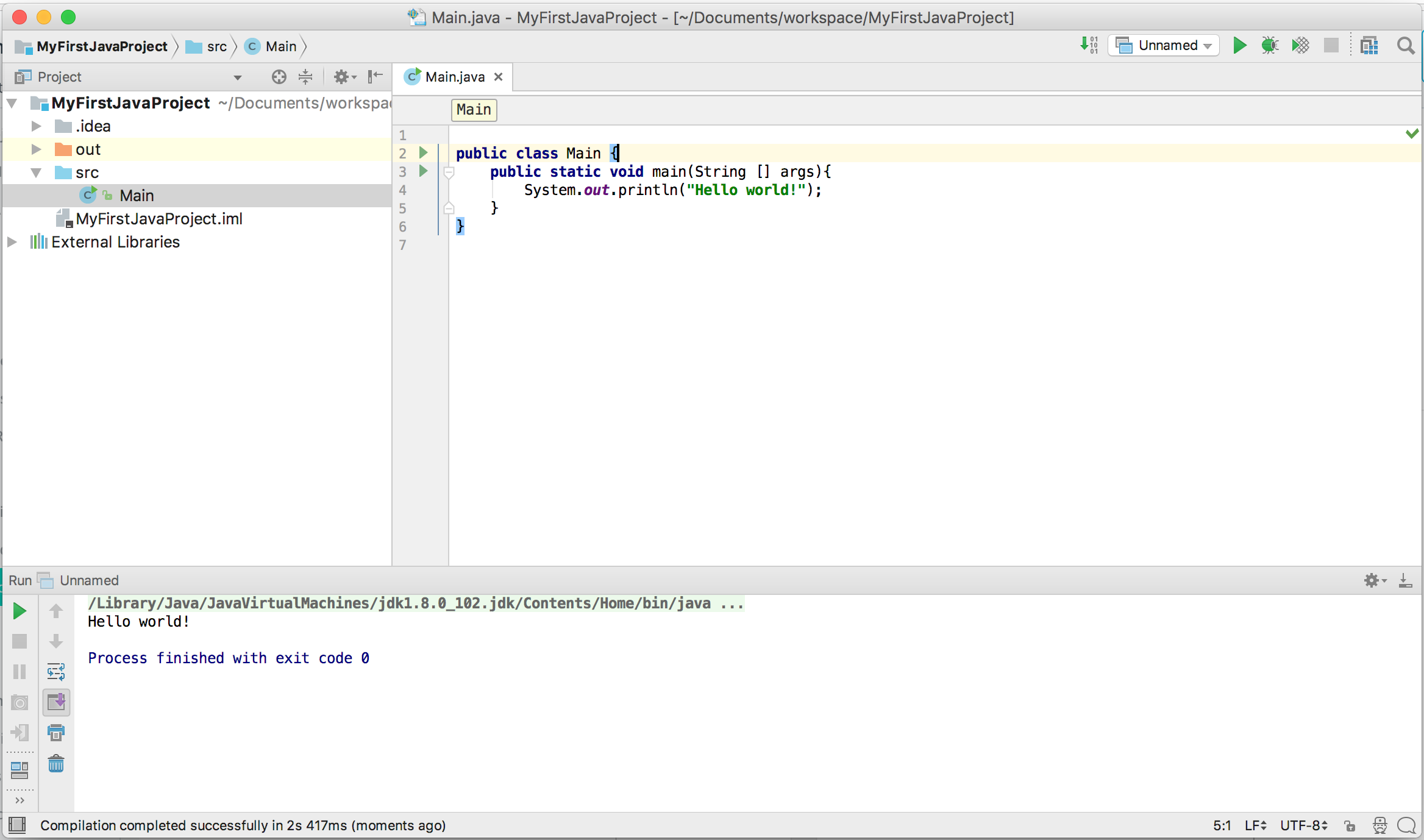
**[[](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)](https://classroom.udacity.com/courses/ud283/lessons/008b74dd-d786-4d22-84e6-ace8ae102ba4/concepts/68052328-6301-4b55-a8cb-b016cc601ecb)**

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:

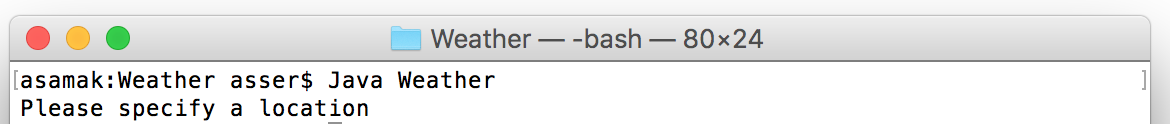
**public** **static** **void** **main**(String args[]){

}

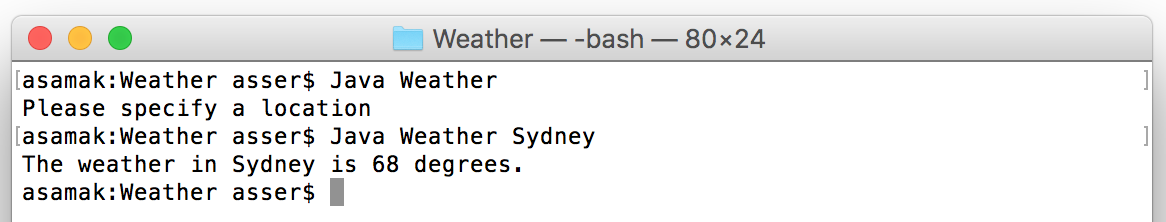
Notice that the method accepts a String array called args[] as an input parameter, but we never explicitly call the main method ourselves! So what is this String array and where does its value ever come from?

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*

}

**Lesson 4**

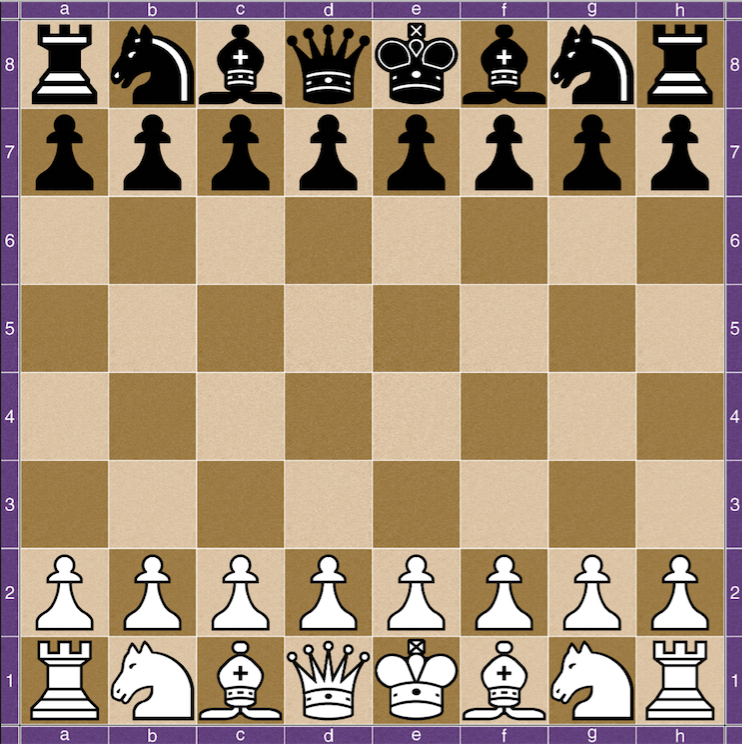
You are allowed to use the parent's data type when declaring a child variable. But to use the parent's constructor you will need to cast the variable before assigning it to the child's data type.

# The Chess Example

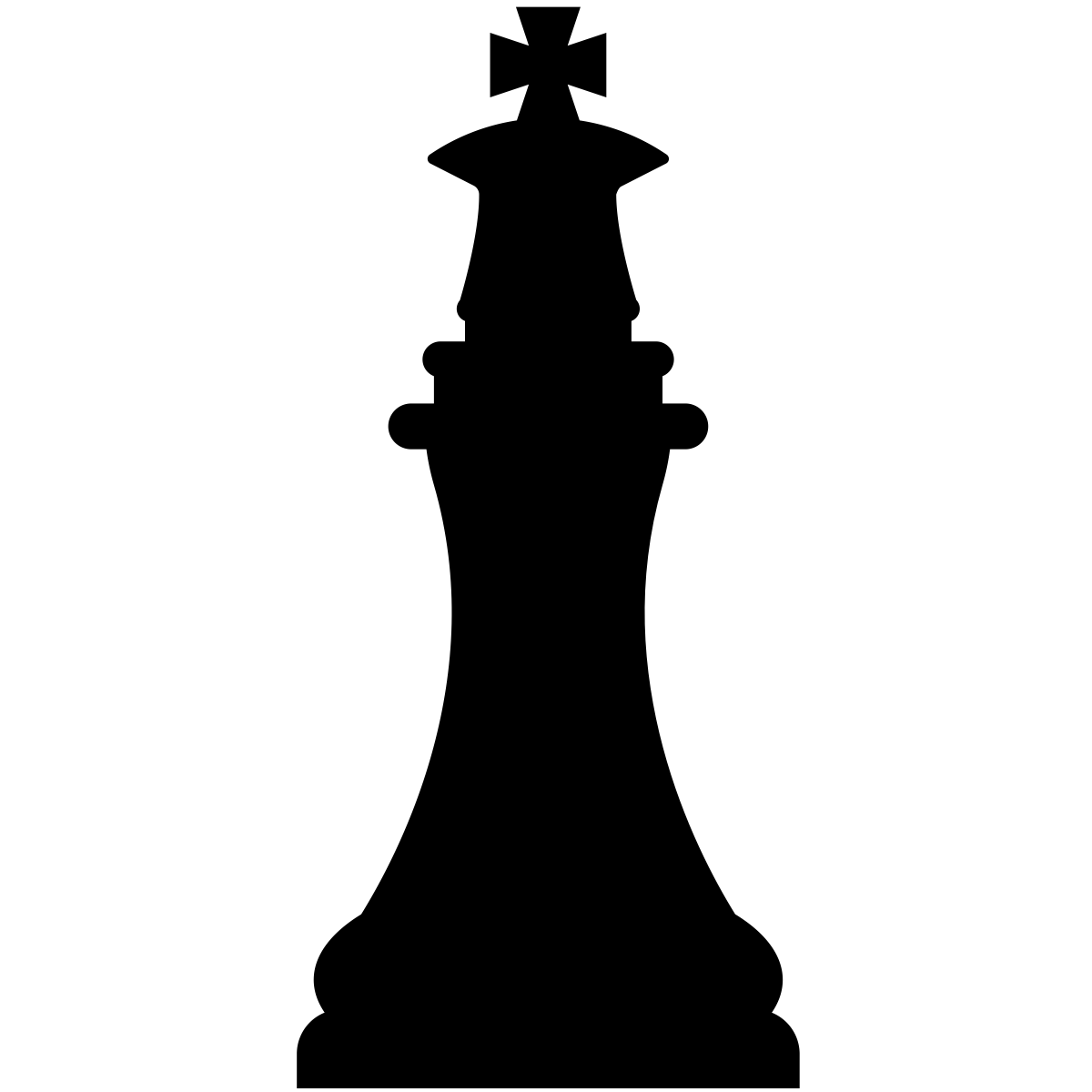
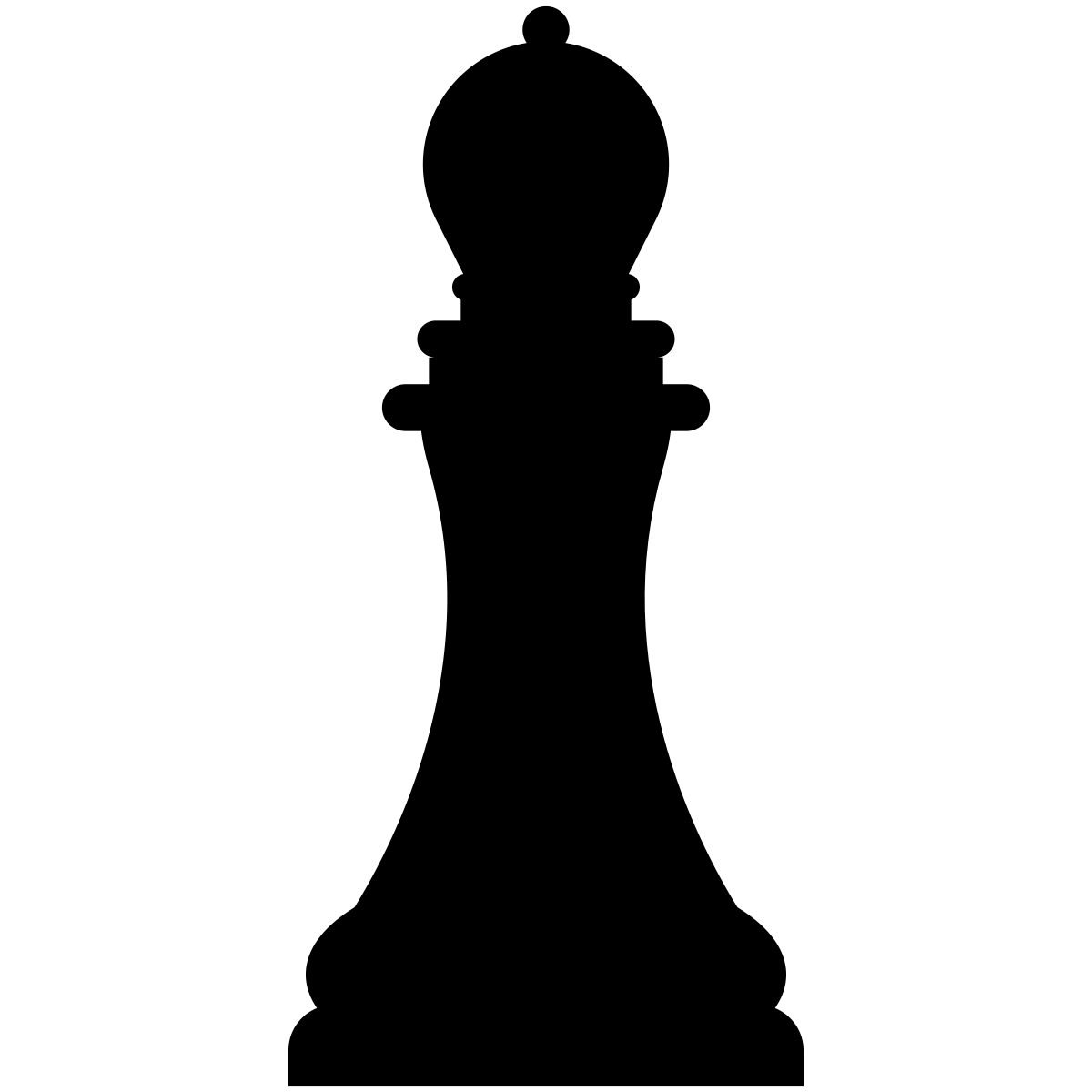
We've seen how Inheritance allows you to extend classes and add more functionality to them.

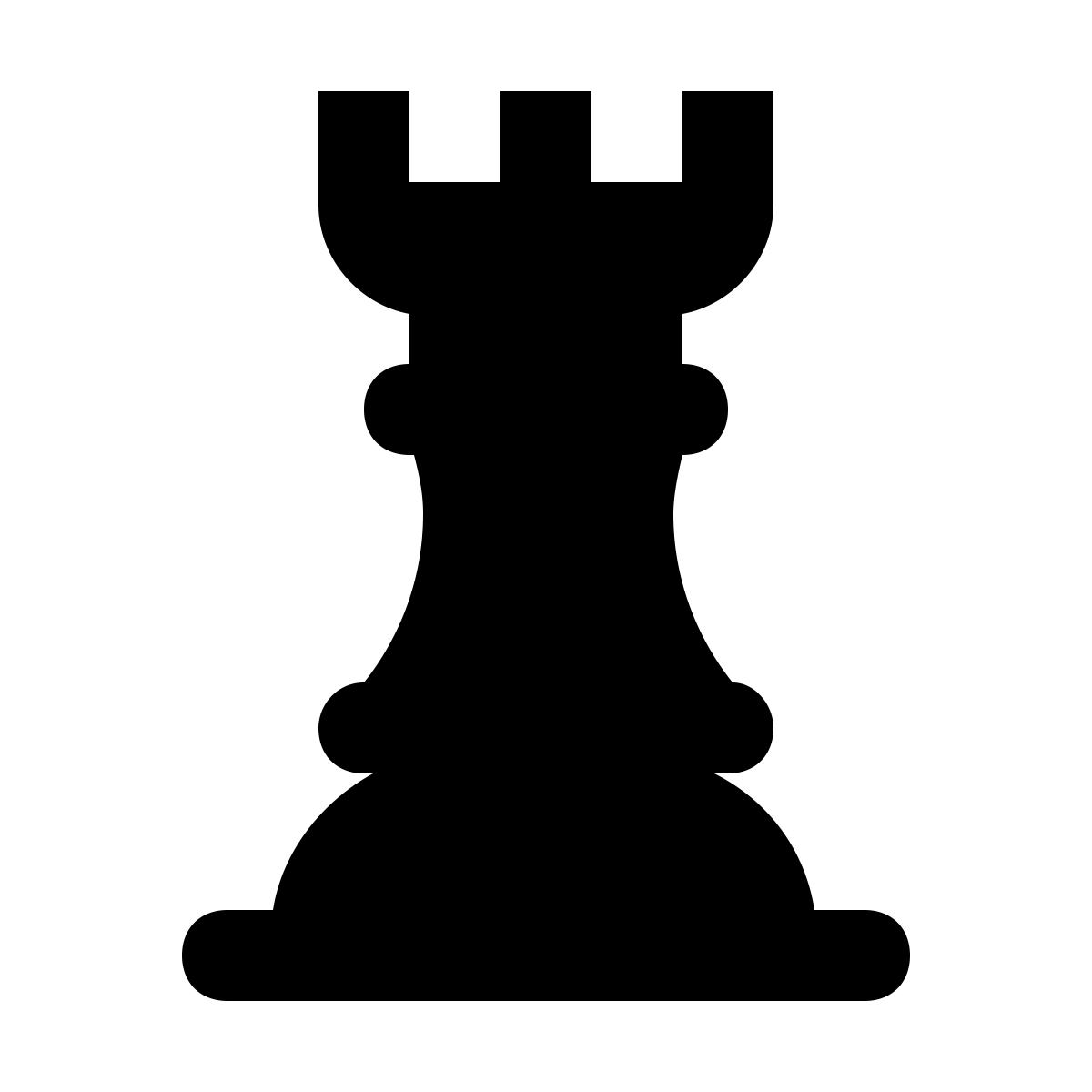
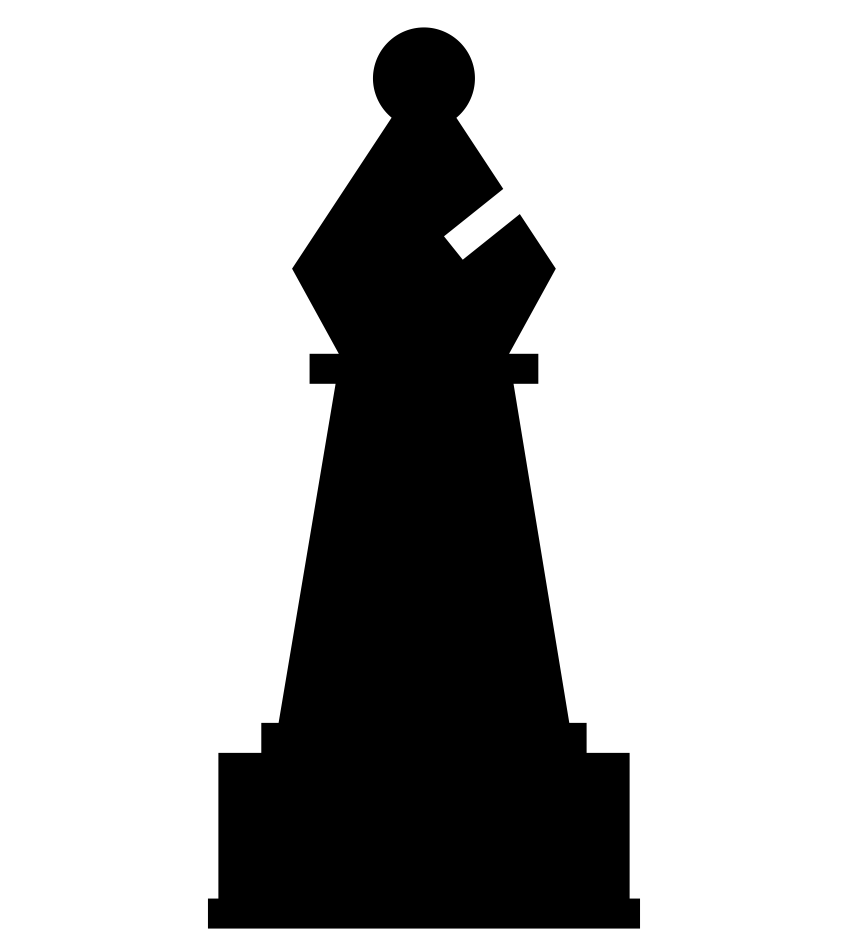
Sometimes you not only want to extend the functionality of a class, but also modify it slightly in the child class.

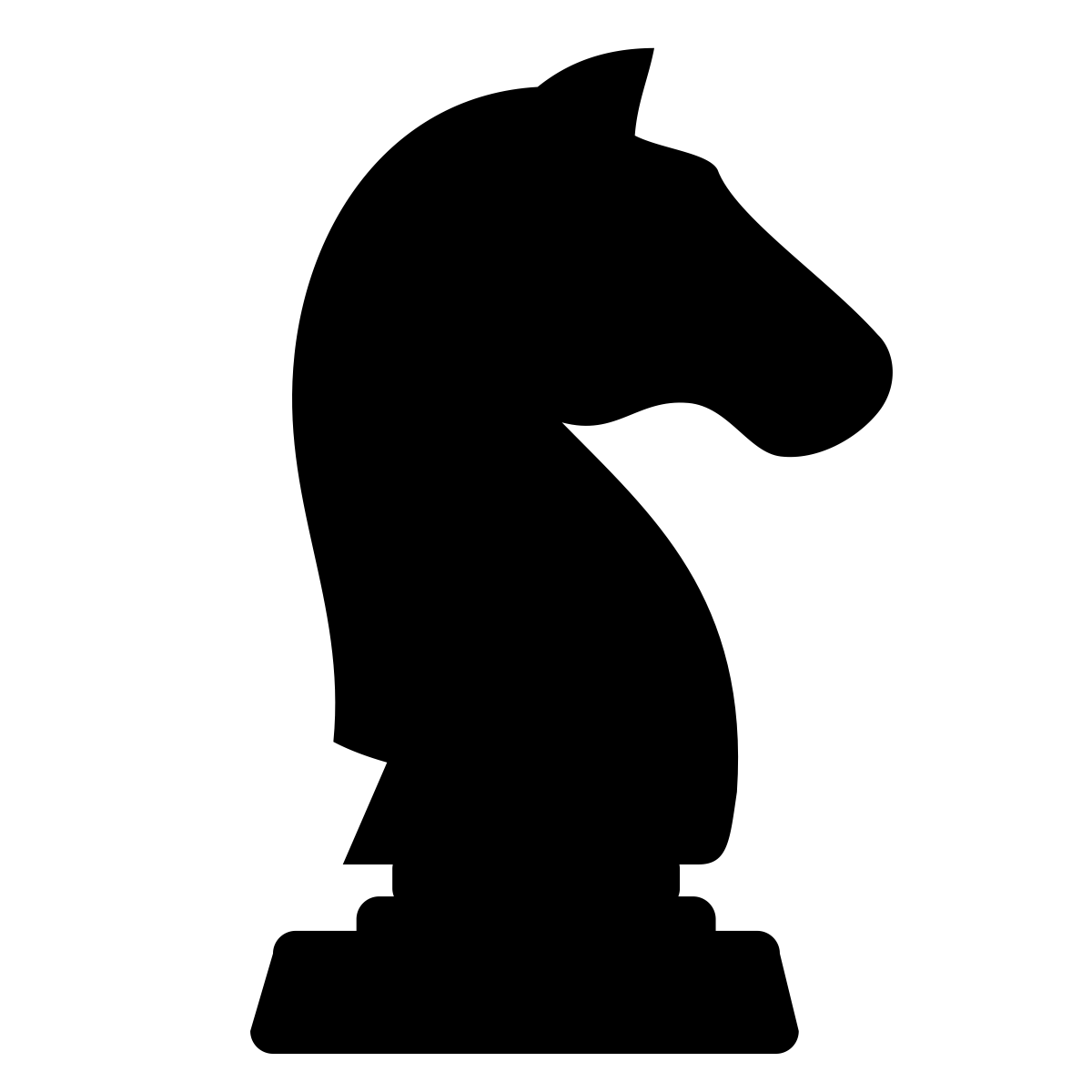
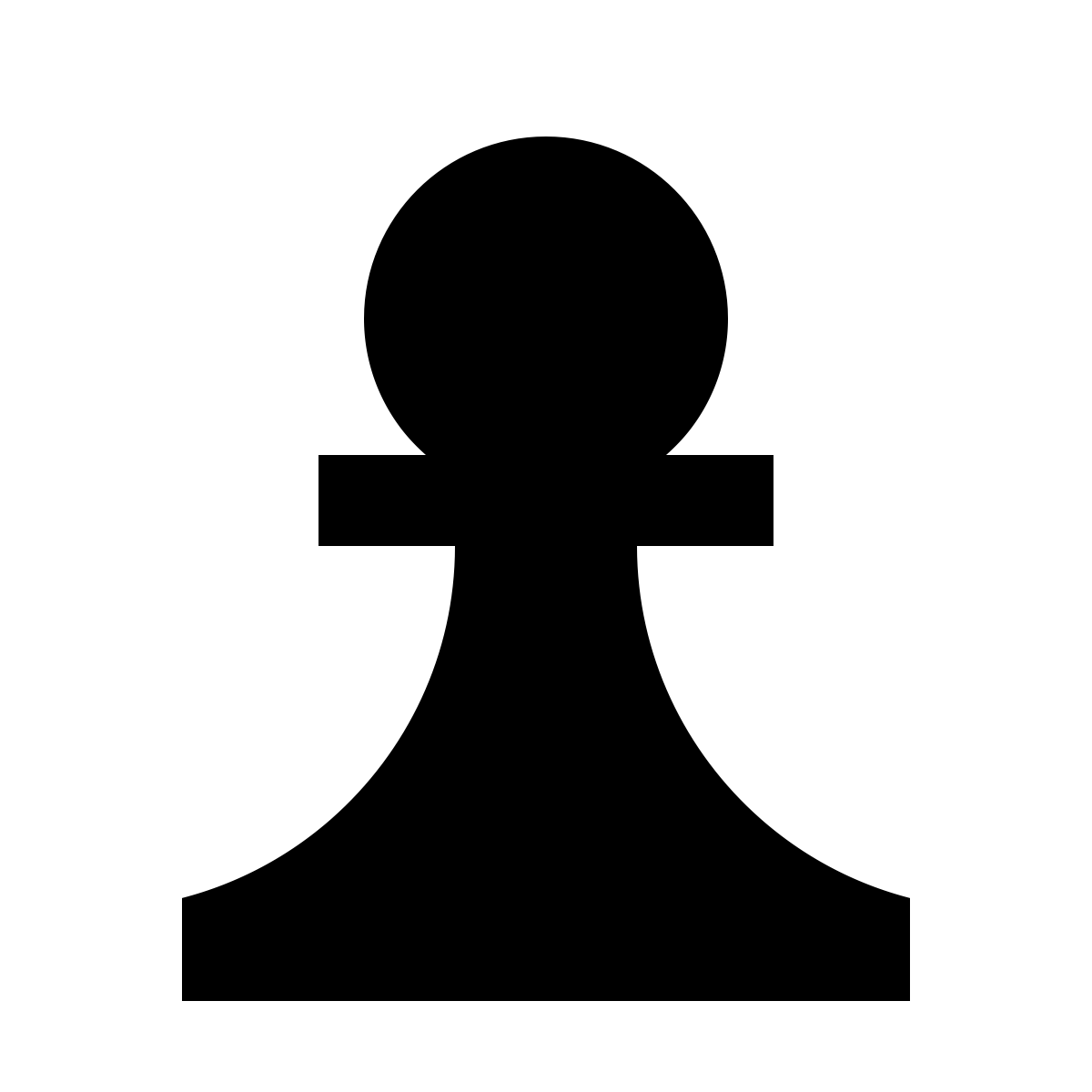
For example, say you're building a Java chess game.

**[[](https://classroom.udacity.com/courses/ud283/lessons/35032599-cea2-464c-bd4e-b42f71b04b42/concepts/f7288fe6-ba9a-43bb-b835-dfae293d6bea)](https://classroom.udacity.com/courses/ud283/lessons/35032599-cea2-464c-bd4e-b42f71b04b42/concepts/f7288fe6-ba9a-43bb-b835-dfae293d6bea)**

A good Java design will have a **class** for each piece type:

 King       Queen

 Rock         Bishop

 Knight      Pawn

And they should all inherit from a common base class: **Piece**

#### Why?

Because according to the concept of polymorphism, you could represent the chess-board as a 2D array of Piece objects, and then each cell in the 2D array can contain any of the child classes that extend the Piece class.

### Other classes

To store this 2D array we will need a class that represents the Game itself:

**class** **Game**{

Piece [][] board;

*// Constructor creates an empty board*

Game(){

board = **new** Piece[8][8];

}

}

And finally, a simple class called Position that has nothing more than a row value and a column value to represent a specific slot on the board.

**class** **Position**{

**int** row;

**int** column;

*// Constructor using row and column values*

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

**this**.row = r;

**this**.column = c;

}

}

That way, the Piece class can include a field variable of type Position that stores the current position of that piece on the board:

**class** **Piece**{

Position position;

}

Now, since all piece types inherit from the same parent class Piece, they will all share any methods declared in that class.

For example:

It will be useful to have a method that checks if a potential movement of a piece is a valid one. Even though each piece type has a unique movement capability, any piece (regardless of its type) has to - at the very least - stay within the bounds of the chess board.

So, a good idea would be to include a method called isValidMove inside the Piececlass that takes a potential new position and decides if that position is within the bounds of the chess board.

**class** **Piece**{

**boolean** **isValidMove**(Position newPosition){

**if**(newPosition.row>0 && newPosition.column>0

&& newPosition.row<8 && newPosition.column<8){

**return** **true**;

}

**else**{

**return** **false**;

}

}

}

Since each of the child classes inherits from that Piece class, each will automatically include this method, which means you can call it from any of those classes directly.

For example:

Queen queen = **new** Queen();

Position testPosition = **new** Position(3,10);

**if**(queen.isValidMove(testPosition)){

System.out.println("Yes, I can move there.");

}

**else**{

System.out.println("Nope, can't do!");

}

### QUIZ QUESTION

What will the above code print?

* 

"Yes, I can move there."

* "Nope, can't do!"
* 

Neither, this code will not work!

SUBMIT

What we've done so far can be considered as a good start for checking the validity of the movement of a piece on the board. However, each piece type has a different pattern of allowed movements, which means that each of those child classes needs to implement the isValidMove method differently!

Luckily, Java not only offers a way to inherit a method from a parent class but also modify it to build your own custom version of it.

Let's see how.

# Override

When a class extends another class, all public methods declared in that parent class are automatically included in the child class without you doing anything.

However, you are allowed to **override** any of those methods. Overriding basically means re-declaring them in the child class and then re-defining what they should do.

Going back to our chess example, assume you're implementing the isValidMovemethod in the Rock class.

The Rock class extends the Piece class that already includes a definition of the isValidMove method.

**class** **Piece**{

**boolean** **isValidMove**(Position newPosition){

**if**(newPosition.row>0 && newPosition.column>0

&& newPosition.row<8 && newPosition.column<8){

**return** **true**;

}

**else**{

**return** **false**;

}

}

}

Now let's implement a custom version of that method inside the Rock class:

**class** **Rock** **extends** **Piece**{

**boolean** **isValidMove**(Position newPosition){

**if**(newPosition.column == **this**.column || newPosition.row == **this**.row){

**return** **true**;

}

**else**{

**return** **false**;

}

}

}

Notice how both method declarations are identical, except that the implementation in the child class has different code customizing the validity check for the Rock piece. Basically it's checking that the new position of the rock has the same column value as the current position (which means it's trying to move up or down), or has the same row position which means it has moved sideways, both valid movements for a Rock piece.

Remember that this.position.column and this.position.row are the local fields of the Rock class holding the current position of that piece.

We can also do the same for all the other piece types, like for example the Bishop class would have slightly different implementation:

**class** **Bishop** **extends** **Piece**{

**boolean** **isValidMove**(Position newPosition){

**if**(Math.abs(newPosition.column - **this**.column) == Math.abs(newPosition.row - **this**.row)){

**return** **true**;

}

**else**{

**return** **false**;

}

}

}

For the Bishop, since it can only move diagonally, we'd want to check that the number of vertical steps is equal to the number of horizontal steps. That is, the difference between the current and new column positions is the same as the difference between the current and new row positions.

I've used Math.abs which takes the absolute value of that difference to always be a positive value, allowing this check to work for all 4 diagonal directions.

Perfect, now try to override this method for the Queen class, remember, a Queen can move diagonally or in straight lines.

# super

SUPER! Not only because you managed to solve that exercise, but "super" is actually another Java keyword that we will be using next!

It is important to note that once you override a method, you basically ignore everything that was in the parent class and instead have your own custom implementation in the child class (literally overwriting it)!

In our case, we don't want to throw away the parent implementation. We actually want to continue to use the original method, and ADD the extra checks for each child class individually.

This is where we get to use the "super" keyword!

You are allowed to re-use the parent method in the child class by using the "super" keyword, followed by a dot and then the method name:

**super**.isValidMove(position);

Using the keyword super here means that we want to run the actual method in the super (or parent) class from inside the implementation in "this" class.

Which means in each of the child classes, before you get to check the custom movement, you can check if super.isValidMove(position) has returned false. If so, then no need to do any more checks and immediately return false; otherwise, continue checking.

The new implementation for the Rock class will look like this:

**class** **Rock** **extends** **Piece**{

**boolean** **isValidMove**(Position newPosition){

*// First call the parent's method to check for the board bounds*

**if**(!**super**.isValidMove(position)){

**return** **false**;

}

*// If we passed the first test then check for the specific rock movement*

**if**(newPosition.column == **this**.column && newPosition.row == **this**.row){

**return** **true**;

}

**else**{

**return** **false**;

}

}

}

You can also use super() to call the parent's constructor. This is usually done when implementing the child's constructor. Typically you would want to first run everything in the parent's constructor then add more code in the child's constructor:

**class** **Rock** **extends** **Piece**{

*// default constructor*

**public** **Rock**(){

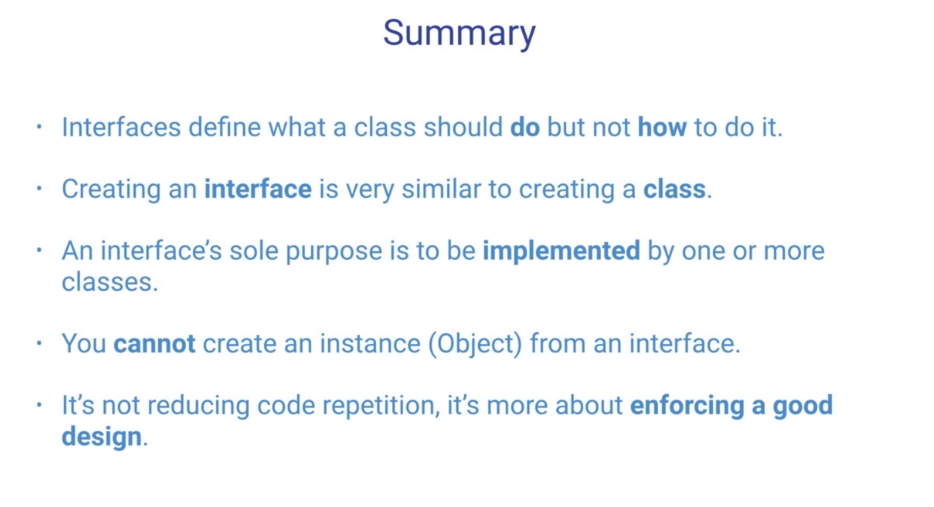
**super**(); *// this will call the parent's constructor*

**this**.name = "rock";

}

}

**Note:** If a child's constructor does not explicitly call the parent's constructor using super, the Java compiler automatically inserts a call to the default constructor of the parent class. If the parent class does not have a default constructor, you will get a compile-time error.



# The Comparable Interface

A very popular interface in Java is the [**Comparable Interface**](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html).

This interface includes a single method definition called compareTo which takes an object as an input parameter of the same type and compares both objects ("this" object against the input argument object).

The main purpose of this interface is to give any class a chance to describe how to compare 2 objects of that class against each other. This will be really handy when we get to sorting or searching for such objects of that type.

For example:

Assume you have a class that represents a book:

**public** **class** **Book**{

**int** numberOfPages;

String title;

String author;

}

And you are asked to implement the Comparable Interface so that you can sort the books according to the following criteria:

1. If a book has more pages than the other, then the book with the more pages goes first.
2. If both books have the same number of pages, then sort by the title alphabetically.
3. If both books have the same number of pages and the same title, then sort by the author alphabetically.

Before we start coding, let’s go through how the compareTo method should work:

The compareTo method takes a single input parameter (let's refer to it as the "specified object") and since this method belongs to an object itself (let's refer to it as "this object"), then the method simply compares the "specified" object against "this" object.

According to the documentation, there are 3 possible outcomes when comparing any 2 objects:

1. "This" object is considered less than the "specified" object
2. "This" object is considered equal to the "specified" object
3. "This" object is considered greater than the "specified" object

Respectively, for each of those cases, compareTo method should return:

1. A negative integer (any number less than 0)
2. zero (0)
3. A positive integer (any number greater than 0)

Ok, now that we've got everything well defined, let's start coding:

**public** **class** **Book** **implements** **Comparable**<**Book**>{

**public** **int** **compareTo**(Book specifiedBook) {

*// First check if they have different page counts*

**if**(**this**.numberOfPages != specifiedBook.numberOfPages){

*// this will return a negative value if this < specified but will return a positive value if this > specified*

**return** **this**.numberOfPages - specifiedBook.numberOfPages;

}

*// If page counts are identical then check if the titles are different*

**if**(!**this**.title.equals(specifiedBook.title){

**return** **this**.title.compareTo(specifiedBook.title);

}

*// If page titles are also identical then return the comparison of the authors*

**return** **this**.author.compareTo(specifiedBook.author);

}

}

# Final methods

OOP (Object Oriented Programming) is powerful - you can extend classes, add features to them and even override their methods to behave differently.

But, remember ...

**[[](https://classroom.udacity.com/courses/ud283/lessons/35032599-cea2-464c-bd4e-b42f71b04b42/concepts/1a102a3c-be87-4cde-8014-635daaa0791c)](https://classroom.udacity.com/courses/ud283/lessons/35032599-cea2-464c-bd4e-b42f71b04b42/concepts/1a102a3c-be87-4cde-8014-635daaa0791c)**

Being able to override any method could be dangerous. If someone creates a class with a certain method, they assume this method behaves in a certain way.

That's why, if you want to protect your method from being overridden in a child class you can prefix it with the keyword final.

A final method can still be accessed by the child class (if the permissions allow so) but cannot be overridden, hence you can guarantee that any final method will behave exactly like the parent's implementation.

Here's an example:

**public** **class** **Room**{

**private** **double** width;

**private** **double** height;

**public** **Room**(**double** width, **double** height){

**this**.width = width;

**this**.height =height;

}

**public** **final** **double** **getArea**(){

**return** width\*height;

}

}

Now if another class extends Room, no matter what type of room it is it shouldn't be allowed to override the method getArea because the area should always be calculated the same way:

**public** **class** **LivingRoom** **extends** **Room**{

*// The constructor simply calls the parent's constructor using super()*

**public** **LivingRoom**(**double** width, **double** height){

**super**(width,height);

}

*// Not allowed to override getArea() here*

}

But since the method is public, it means that it's also available in the child class:

LivingRoom myLivingRoom = **new** LivingRoom(5,3);

**double** area = myLivingRoom.getArea();

System.out.println(area);

The above code will work just fine, and the output will be **15.0** as expected!

# Final fields

The final keyword can also be used to describe fields. However, unlike with methods, a final field has nothing to do with inheritance!

A final field is simply a constant variable! In other words, a variable that is only to be set once and is not allowed to change ever again!

A good example of a final field is defining math constants, like **PI**:

**public** **class** **MathLib**{

**public** **final** **double** PI = 3.14;

}

This basically means that even though the field is public, you are not allowed to change the value of PI anywhere (inside or outside of this class).

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

MathLib mathlib = **new** MathLib();

mathlib.PI = 0; *// This is not allowed and will show a compiler error!*

}

# Static Methods

Just like static fields, static methods also belong to the class rather than the object.

It's ideally used to create a method that doesn't need to access any fields in the object, in other words, a method that is a standalone function.

A static method takes input arguments and returns a result based only on those input values and nothing else.

Not needing any field values makes it easy for a method to be attached to the class definition and not an individual object since it doesn't care about the values of any of the fields.

However, a static method can still access static fields, that's because static fields also belong to the class and are shared amongst all objects of that class.

Here's an example of a calculator implementation with some static methods:

**public** **class** **Calculator** {

**public** **static** **int** **add**(**int** a, **int** b) {

**return** a + b;

}

**public** **static** **int** **subtract**(**int** a, **int** b) {

**return** a - b;

}

}

Since both add and subtract don't need any object-specific values, they can be declared static as seen above and hence you can call them directly using the class name Calculator without the need to create an object variable at all:

Calculator.add(5,10);

# Lists

A List in Java is an interface that behaves very similarly to an array

* It's an ordered collection (also known as a sequence).
* The user of this interface has precise control over where each item is inserted in the list.
* The user can access items by their integer index (position in the list).
* The user can search for items in the list by looping over the items in it.

In fact, the most common class that implements the List interface uses an array internally and is even called ArrayList

# ArrayList

An ArrayList is a class that implements the interface List. It's simply a wrapper around an array, but provides really powerful methods that make dealing with the array much simpler.

**Note:** An item in an ArrayList is known as an element

Let's have a look at some of the ArrayList's methods:

* **add(E element):** Appends the specified element to the end of this list.
* **add(int index, E element):** Appends the specified element to the specified index of this list.
* **get(int index):** Returns the element at the specified position in this list.
* **contains(Object o):** Returns true if this list contains the specified element.
* **remove(int index):** Removes the element at the specified position in this list.
* **size():** Returns the number of elements in the list.

For the full list of methods check out the documentation [**here**](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)

To create and initialize an ArrayList:

ArrayList grades = **new** ArrayList();

Then you can add elements by calling the add() method:

grades.add(100);

grades.add(97);

grades.add(85);

To access the first item in the list:

grades.get(0)

This will return the value stored at index 0 (in this case: "100")

At any point, you can check the size of an array using the size() method:

System.out.println(grades.size().toString());

This will print the number of elements in the list (in this case: "3")

You can also remove items by index:

grades.remove(0);

Which will remove the first item in the list with index=0 ("100"), and then shift the other 2 items ("97" and "85") to have the indices (0 and 1) respectively

You can even clear the entire list by calling the clear() method:

grades.clear();

Some of these methods save us from writing extra code (loop and if checks) that we used to do with arrays. That's because the ArrayList already has such code implemented internally.

For example, when adding an element to the end of an array you will need to keep track of the index pointing to that last slot, and continue to increment that index every time you add a new item. But with ArrayLists, all you have to do is call the method add() with the item and it will take care of finding the correct index and inserting it in the right place.

Also, notice that we don't need to specify the initial size of an ArrayList:

ArrayList myArrayList = **new** ArrayList();

Unlike what we used to do with arrays:

**int**[] myArray = **new** **int**[100];

This is because the ArrayList class takes care of resizing the internal array every time it runs out of space, which is all done behind the scenes so you don’t have to worry about implementing any of that.

# Loops & Collections

Just like with arrays, the best way to access each and every element in an ArrayList is to create a loop and use the loop counter as an index:

*// First get the size of the array list*

**int** size = list.size();

*// Start the loop*

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

System.out.println(list.get(i));

}

The above will get each element at index "i" and print it.

There's another type of loop that's basically a shorthand for loops that doesn't require you to specify a loop counter variable nor an index variable.

It consists of 2 parts, declaring the item variable followed by a colon : then the ArrayList variable (or any collection type):

*// iterate via "New way to loop"*

**for** (String item : list) {

System.out.println(item);

}

This basically reads: For each item in the list, print that item! Pretty simple!

### Advanced topic:

There's another way to loop over collections, however, it's a more advanced technique that should only be used in certain collections where the 2 loops above are too slow, like a LinkedList. For more information check out this [**link**](https://www.tutorialspoint.com/java/java_using_iterator.htm)

# Stacks

The Stack collection represents a last-in-first-out (LIFO) stack of objects.

It means that the last item added (pushed) to a stack is the first item removed (popped) from that stack.

Just like a stack of plates: The last plate I add on will always be the first plate I take off.

**[[](https://classroom.udacity.com/courses/ud283/lessons/d9b6371c-f3bb-4ebf-95dd-7c43d80b89f0/concepts/d1e2d14b-f869-4bff-a9d9-6ff84116a66c)](https://classroom.udacity.com/courses/ud283/lessons/d9b6371c-f3bb-4ebf-95dd-7c43d80b89f0/concepts/d1e2d14b-f869-4bff-a9d9-6ff84116a66c)**

The Stack class includes these five methods:

* **push(E item):** Adds an item onto the top of this stack
* **pop():** Removes the object at the top of this stack and returns that object
* **peek():** Returns the object at the top without removing it from the stack.
* **empty():** Checks if this stack is empty.
* **search(Object o):** Searches for an object in this stack and returns its position.

Just like the ArrayList, the Stack also internally uses an array that stores everything in order.

An example of when a stack is useful would be when developing something like an email system: Once the email server receives a new email it should add this email to the top of the stack of emails so that the user will read the latest email first!

Here's a simple example of how to use a Stack:

Stack newsFeed = **new** Stack();

newsFeed.push("Morning news");

newsFeed.push("Afternoon news");

newsFeed.push("Evening news");

String breakingNews = (String) newsFeed.pop();

System.out.println(breakingNews);

Ok, great, now what happens if we call these 2 lines again right after the code above?

String moreNews = (String) newsFeed.pop();

System.out.println(moreNews);

If you want to access the top of the stack **without** removing it, then call peek instead of pop

String peekNews = (String) newsFeed.peek();

#### Note

You might be wondering why the need to cast the return type of pop() and peek()using (String)?

That's because these methods actually return the type Object and not String, simply because it has no idea what you've inserted and what type it is!

We will learn more about this when we get to the Generics section of this lesson.

# Queue

Another very common collection type used in Java is the Queue. As the name suggests it resembles the queue or line of people one after the other.

Unlike the stack, it's a First-In-First-Out (FIFO) data type where the first element added to the queue is the first element to be accessed or removed.

**[[](https://classroom.udacity.com/courses/ud283/lessons/d9b6371c-f3bb-4ebf-95dd-7c43d80b89f0/concepts/d5e4bf98-bd9a-4512-aea0-e31314a7b18e)](https://classroom.udacity.com/courses/ud283/lessons/d9b6371c-f3bb-4ebf-95dd-7c43d80b89f0/concepts/d5e4bf98-bd9a-4512-aea0-e31314a7b18e)**

The Queue is only an Interface and not a Class by itself, however, it defines 2 important methods for all classes that do implement the Queue interface.

* **add(E element):** Inserts the specified element into this queue
* **poll():** Retrieves and removes the head of this queue

A special type of Queues is known as Deque which is a double-ended queue. Meaning that you can add or remove elements from either end of a Deque (Front or End).

Along with the 2 Queue methods, a Deque also offers these methods:

* **addFirst(E element):** Inserts the specified element at the front of this deque
* **addLast(E element):** Inserts the specified element at the end of this deque
* **pollFirst():** Retrieves and removes the first element of this deque
* **pollLast():** Retrieves and removes the last element of this deque

Java also provides a few classes that implement the Queue Interface, perhaps the most popular of all is the LinkedList

Here's an example on how to create and use a LinkedList object:

Queue orders = **new** LinkedList();

orders.add("Order1");

orders.add("Order2");

orders.add("Order3");

System.out.print(orders.poll());

System.out.print(orders.poll());

System.out.print(orders.poll());

# Generics

In a nutshell, Generics enable classes to accept parameters when defining them, much like the more familiar parameters used in method declarations.

Defining a type parameter for a class provides a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

## Generics in Collections

For example: ArrayLists use Generics to allow you to specify the data type of the elements you're intending to add into that ArrayList.

The way to do so is by defining that data type between <> when declaring the ArrayList variable:

ArrayList<String> listOfStrings = **new** ArrayList();

This is explicitly saying that you want to create an ArrayList of Strings, and hence the compiler will only allow Strings to be inserted into this ArrayList and will show you an error if you try to insert something else.

## Generics eliminate the need for casting

Remember how we had to cast the return value of a pop() method when we were using a collection like the Stack earlier?

Well, because of generics you can now specify the datatype of the stack or queue or any collection when declaring it, thus eliminating the need to cast any return types:

The following code snippet without generics requires casting:

List list = **new** ArrayList();

list.add("hello");

String s = (String) list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = **new** ArrayList<String>();

list.add("hello");

String s = list.get(0); *// no cast*

## Define your own Generic Types

And finally, you can create your own generic type by declaring a generic parameter when defining your class. This will not be covered in this lesson, but if you want to know more about how to do so check out the documentation [**here**](https://docs.oracle.com/javase/tutorial/java/generics/types.html)

# ArrayList methods

You can search for an item in an ArrayList the same way we've done with arrays before, all you have to do is create a loop that checks every item in the ArrayList and compares it to the one you are looking for.

For example, looking for the String "Sydney" in the ArrayList cities will look like this:

**for**(**int** i=0; i<cities.size(); i++){

**if**(cities.get(i).equals("Sydney")){

**return** **true**;

}

}

Or even use the shorthand loop that we mentioned earlier:

**for**(String city : cities){

**if**(city.equals("Sydney")){

**return** **true**;

}

}

Moreover, ArrayList comes with the power of being a class that contains a bunch of useful methods, including:

**int** **indexOf**(Object o)

This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

So instead of having to create a loop and search "Sydney" one by one you can replace all that with this 1 line of code:

cities.indexOf("Sydney");

If the above returned -1 then Sydney is not in the list, if it returned any positive value than that will be the index of the String "Sydney".

Another really cool method is the remove() method.

The remove() method not only handles searching for and removing the item from the list but also shifts all the following items to fill up the gap created by removing that item.

Check out this [**documentation page**](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html) for more information about all the methods in the ArrayList class.

# Optimization

Understanding how different data structures work in Java will help you become a great software developer. The reason it is important to use the correct data structure for a variable or a collection is: **performance**.

A single program can be implemented in so many different ways, but only some will run smoothly and fast enough that users will want to continue using it.

For example:

We've seen how to search an ArrayList of Strings for a particular String inside, we know that we need to use a loop to compare that String with every other String in the list (even when using the *indexOf* method, internally it will still use a loop to search for your item).

It may not seem to take that long when you try it on your computer, but if that ArrayList contained millions of items (a very common list size in Java btw), it will take a noticeable time to find the item you are looking for.

Users usually consider a program "non-responding" if it takes more than 10 seconds to complete an operation! Which means if the search itself took that long, no-one will ever use your program again!

# HashMaps

A HashMap is another type of collection that was introduced in Java to speed up the search process in an ArrayList.

In some sense, it's just another collection of items (Strings or Integers or any other Object), but the way it stores those items is unique.

HashMaps allow you to store a key for every item you want to add. This key has to be unique for the entire list, very much like the index of a typical array, except that this key can be any Object of any Type!

The point is to be able to find an item in this collection instantly (without having to loop through all the items inside) and hence save that precious run-time.

Let's have a look at an example:

Consider you have a class called Book that contains every detail about such book:

**public** **class** **Book**{

String title;

String author;

**int** numOfPages;

**int** publishedYear;

**int** edition;

String ISBN;

}

If you were to create a Library class that will simulate a virtual library of all the books that exist in the world (~130 Million) you can easily create an ArrayList of Books, and fill it up with all the book details that you may have.

**public** **class** **Library**{

ArrayList<Book> allBooks;

}

Now, to search for a book by its ISBN (String) you will need to create a loop to compare the ISBN of each book with the one you are looking for:

Book **findBookByISBN**(String isbn){

**for**(Book book : library.allBooks){

**if**(book.ISBN.equals(isbn)){

**return** book;

}

}

}

A way more optimal solution is to use a HashMap instead of ArrayLists. To use HashMaps, you will need to import it at the very top of your Java file:

**import** java.util.HashMap;

This is how you declare a HashMap:

**public** **class** **Library**{

HashMap<String, Book> allBooks;

}

The above declaration means that we are creating a collection of Books with a key of type String.

To initialize this hash map, use the default constructor like so:

allBooks = **new** HashMap<String, Book>();

Then, to add items to the HashMap:

Book taleOfTwoCities = **new** Book();

allBooks.put("9781539687399", taleOfTwoCities);

To search for a book using its ISBN:

Book **findBookByISBN**(String isbn){

Book book = allBooks.get(isbn);

**return** book;

}

The above code will use the String key (ISBN) to find the book **instantly** in the entire collection of ~130 Million books leading to a much better performing Java program.