IT 311 OBJECT ORIENTED PROGRAMMING I

Prepared by: Assist. Lect. Mohammad Salim Al-Othman

Department of INFORMATION TECHNOLOGY,

2018-2019 Fall Semester

FACULTY OF SCIENCE

ISHIK UNIVERSITY

Course materials are mainly based on

(Tutorialspoint.com, Udacity.com, Docs.oracle.com/javase)

For more details refer to this textbook:

(Object-Oriented Second Edition Programming and Java by Danny Poo)

October 2018

Contents

Part One: The World of Objects

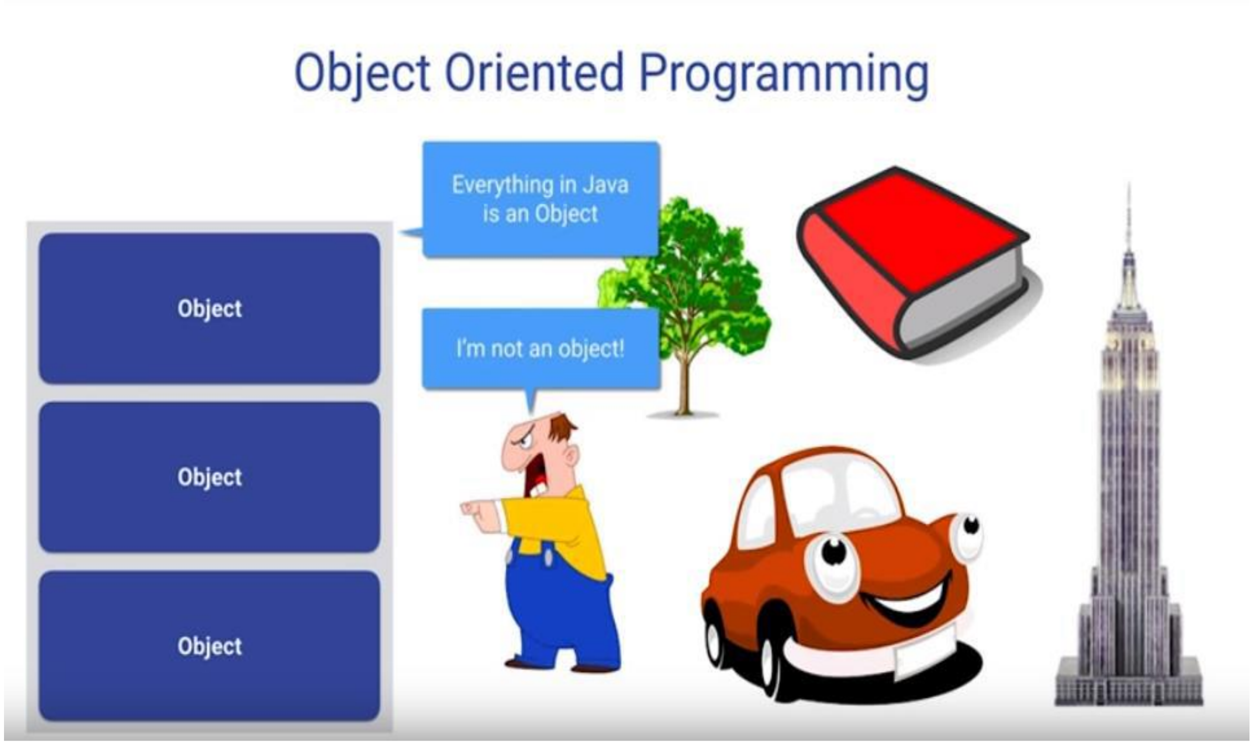
This lesson provides an introduction to object-oriented programming, and also explains how to use classes and objects in Java.

Before you start this course we recommend you first go through introduction about Java (Variables , loops , methods) and familiarize yourself with everything in it. This course will cover a variety of slightly advanced topics in Java:

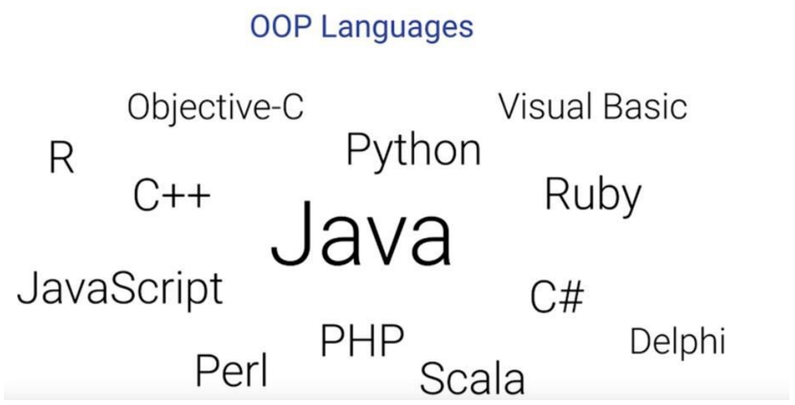
- **Object Oriented Programming**
- **Interacting with the user**
- **OOP concepts like "Inheritance" and more**
- **Collections to store a group of objects**

What's OOP

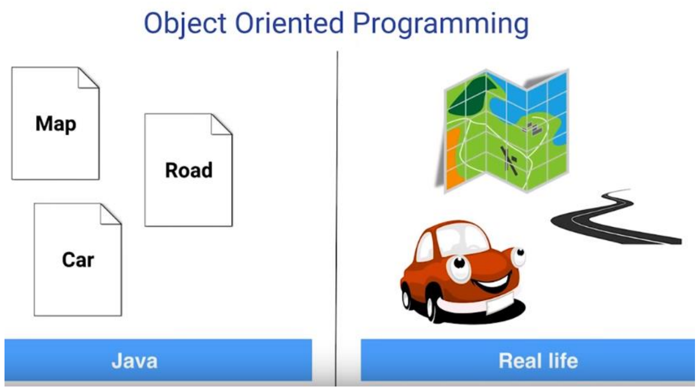
OOP is a type of programming that is driven by modeling your code around building units called objects, each object as the name implies represents a real-life object in the world around us like a building a book a tree a car even a person.



Nowadays almost every modern programming language you've heard of is an object-oriented including our Java.

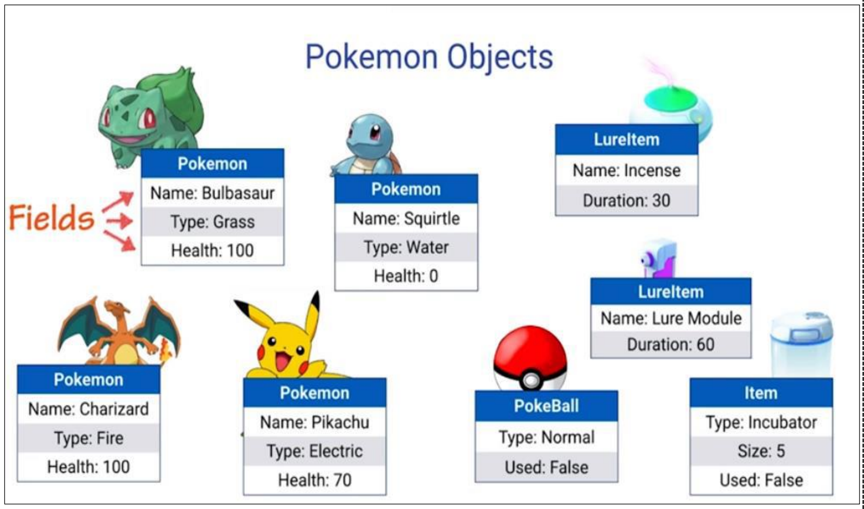


The idea is when you're coding you want to solve a real-world problem and modeling your code to match what you're trying to solve makes sense.

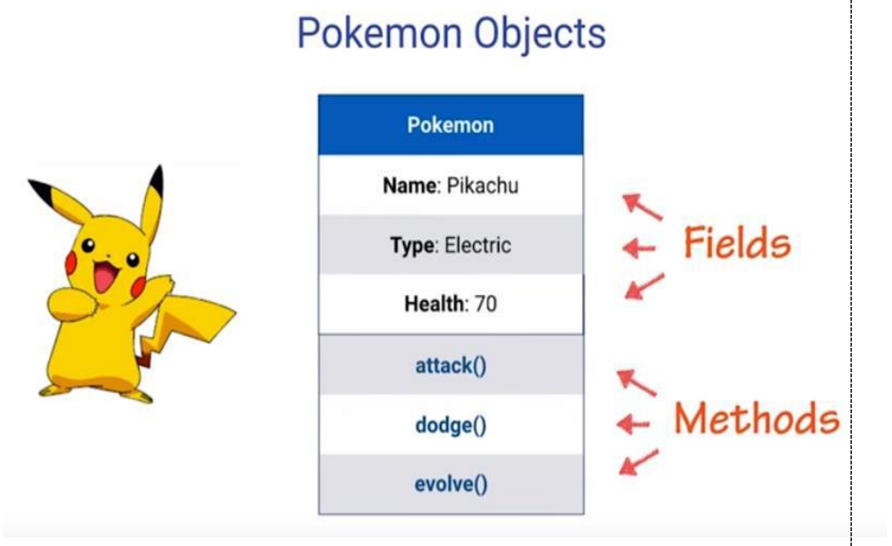


For example, assume you're building a Pokemon game in Java, then your code mostly will have an object for each Pokemon characters. As well as Pokemon objects, an object for each item he can carry and so on.

Each object is responsible for holding the data that describes itself. The Pokemon object for example has a name , a type , and a number indicating health level. These are referred to as **Fields** and along with having fields , objects are also able to perform actions.

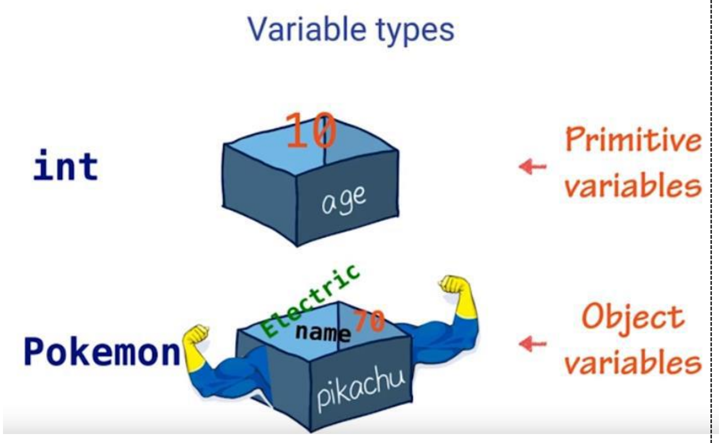


A Pokemon for example can Attack, doge and evolve. These actions in Java are referred to as **Methods**.



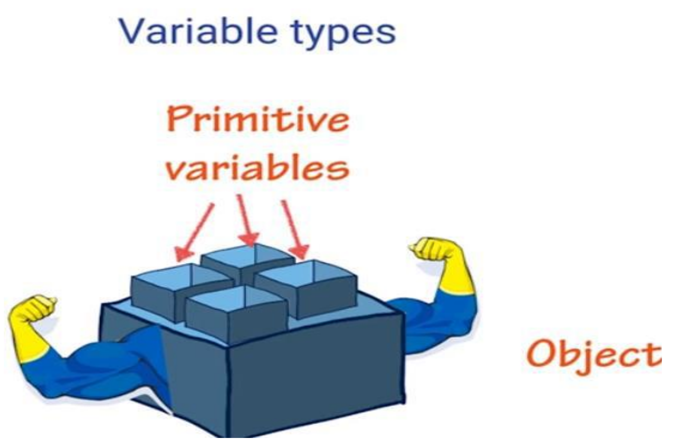
Now the question what does that example go to do with Java?

Just like creating variables of basic data types like **integers** and **doubles** otherwise known as **Primitive** variables, an object is nothing more than an enhanced data type that you get to design by yourself. An **integer** for example can only store a single number. But a Pokemon object can store the name, type, how fights won and much more all in one single variable and it can perform actions using **Methods**.



So why people are still using Primitive variables now?

As you might know you can't really write any Java code without them, since object themselves are made up of those primitive variables.



To summarize why we use objects ?

- Objects combine variables together to make your code meaningful.
- Code becomes organized and easier to understand.
- Maintaining your code becomes much simpler.
- Java won't work without objects!

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.

Integrated Development Environment

What is an IDE

To be able to create and run any code in Java (and pretty much any programming language), you will need 2 main things:

1. A helpful text editor that highlights keywords with different colors and autocompletes code.
2. A compiler that converts your Java code into computer code (known as **bytecode**) that can be understood by computers and hence run properly.
   An IDE (which stands for **Integrated Development Environment**) combines both of those amongst other features like highlighting errors and potential bugs.

   An IDE will help you power through creating any project in almost any programming language.

Choosing an IDE

There are plenty of options out there, and choosing one is usually based the programming language you're using as well as your personal preference. Here's a list of the most commonly used **Java IDEs**:

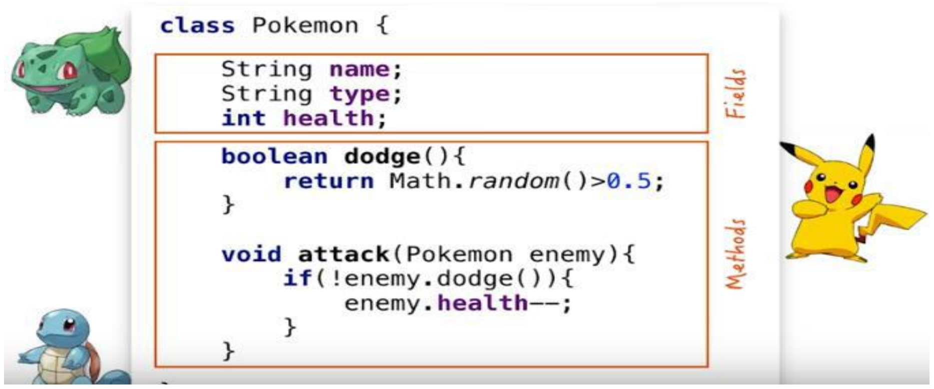
1. IntelliJ
2. NetBeans
3. Eclipse
4. Android Studio (based on IntelliJ)

Objects & Classes

Objects are so powerful they can hold up lots of different data types into that one variable and they can perform actions using methods. So, they keep our code well-organized. But how we as developers get to design our own objects?

We do so by creating classes, a class simply put it is just an object type. It’s the blueprint that defines what the object should looks like. An object on the other hand is the actual entity that is created from the class.

In other words, a class is where you would list all the fields and implement all the methods when defining what the object should look like. A Pokemon class would list that a Pokemon has to have a name, type and a value for itself and it can also perform actions like doge and attack.

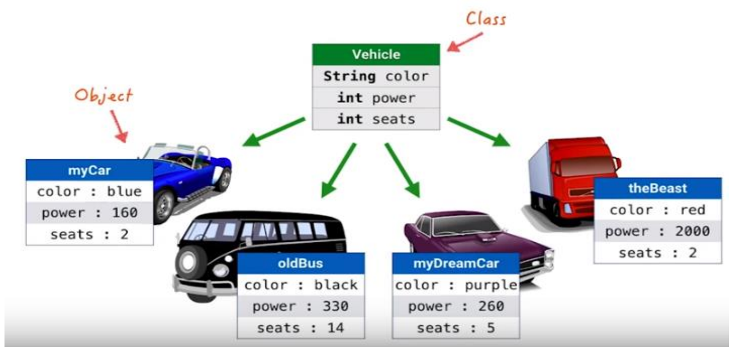


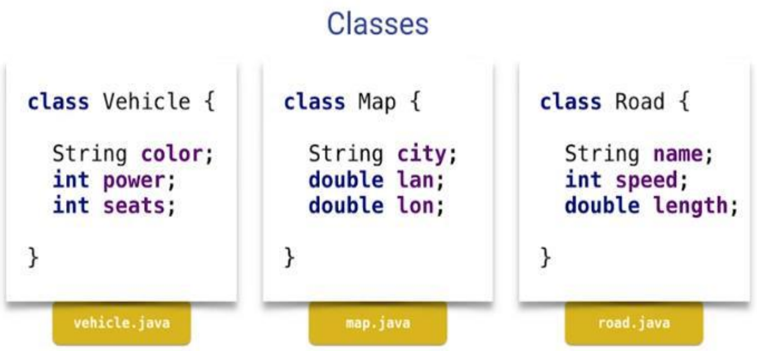
```
class Pokemon {
    String name;
    String type;
    int health;

    boolean dodge(){
        return Math.random()>0.5;
    }

    void attack(Pokemon enemy){
        if(!enemy.dodge()){
            enemy.health--;
        }
    }
```

Another example, if you wanted to define a class that's called vehicle that has the fields: color horsepower, and number of seats. My car would be a single object of that class. And inside that object, it would typically have those fields set to specific values. Like my car is blue, has the horsepower of 160 and the seats are two. And, you can create as many objects as you want of that class.

Each object would have a set of different values for those same fields. But, they all have to include a color, the number of seats and the horsepower as defined by the class. In Java, each class should be created in its own file. Each of those files has the extension.java.

And a typical Java program is nothing more than a bunch of those Java files or classes interacting with each other. No Java code can live anywhere outside a class, which means that any variable has to be a field inside some class and any code or logic has to be inside some




Classes

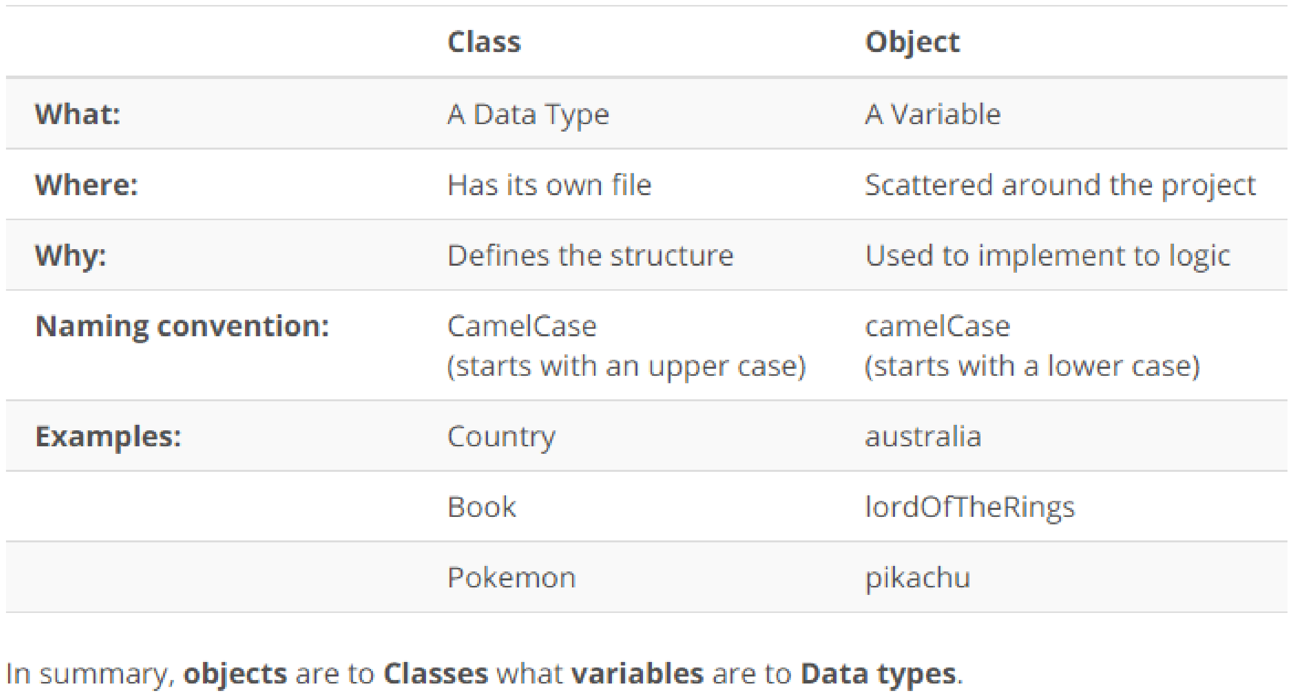
```
class Vehicle {

  String color;
  int power;
  int seats;

}
```

```
class Map {

  String city;
  double lan;
  double lon;

}
```

```
class Road {

  String name;
  int speed;
  double length;

}
```

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**.

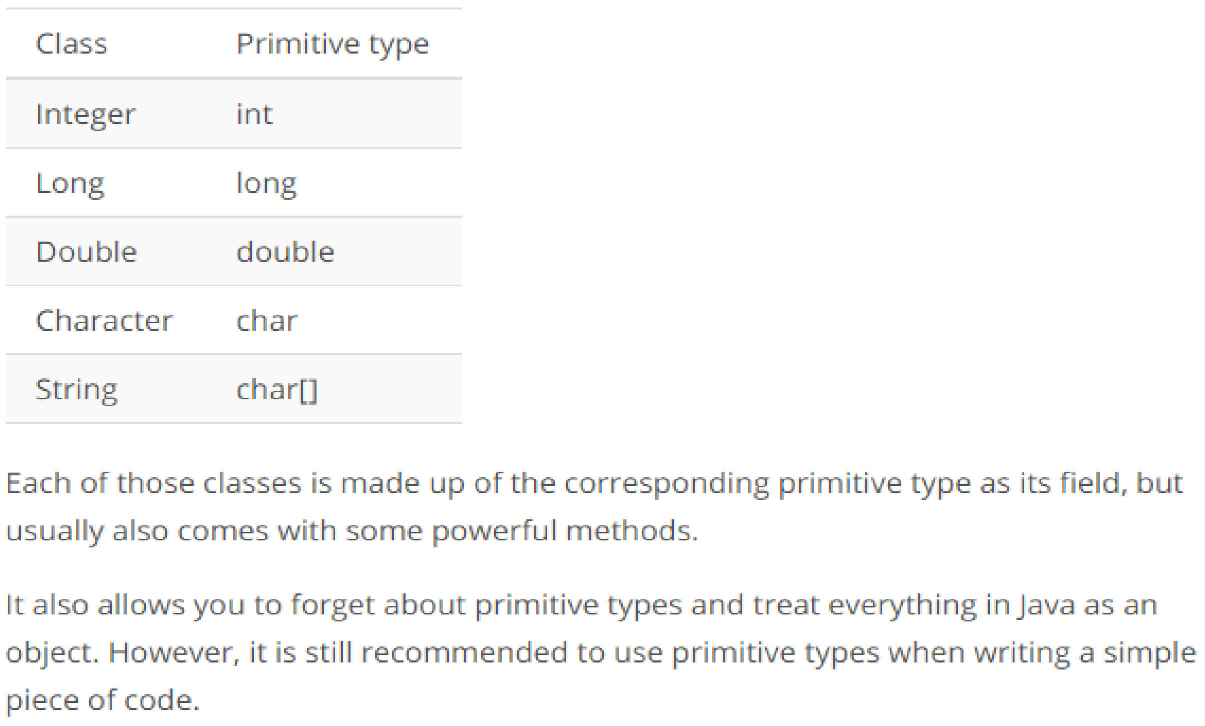
Strings

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.

Creating Classes

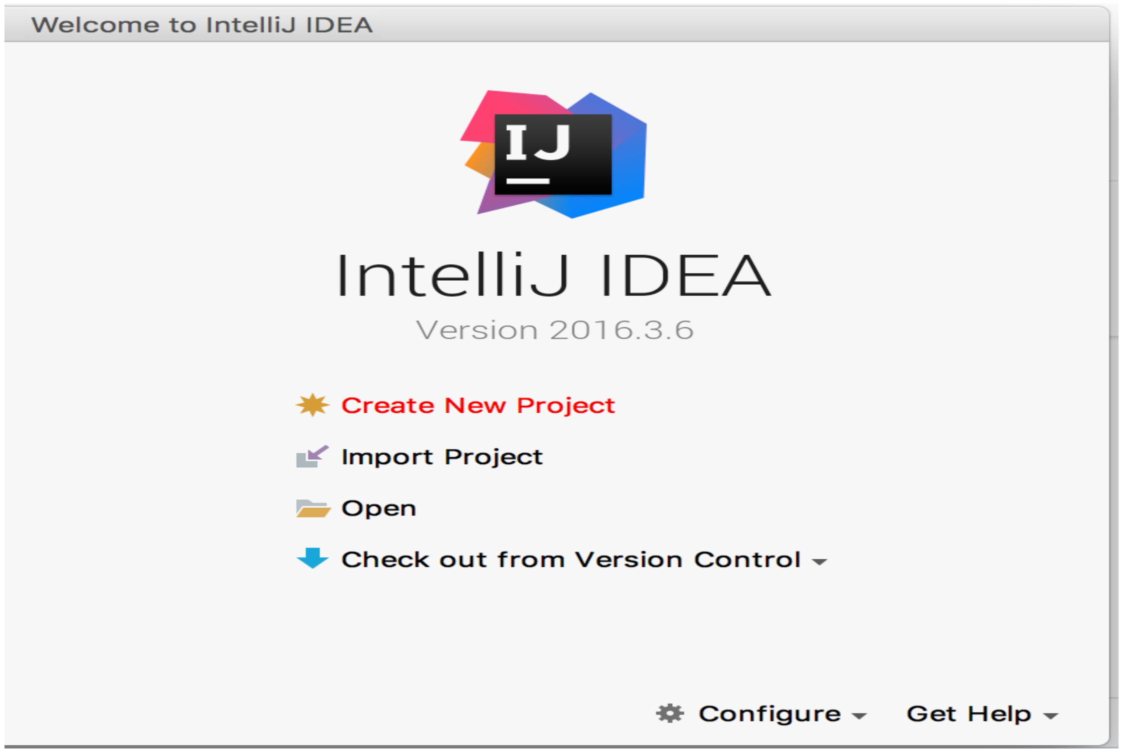
You now know that classes in Java are simply plain text files with an extension.java And a Java project is simply a folder that contains a bunch of those Java files. In fact, you can actually create an entire Java project with nothing more than a basic text editor like a notepad and a compiler that runs on the command line.

That's pretty much all you need. But nowadays, there are plenty of really cool tools called IDEs that make this development experience much more pleasant. IntelliJ is one of those. So, for the rest of this course, we will be showing you how to create in-projects in intelliJ.

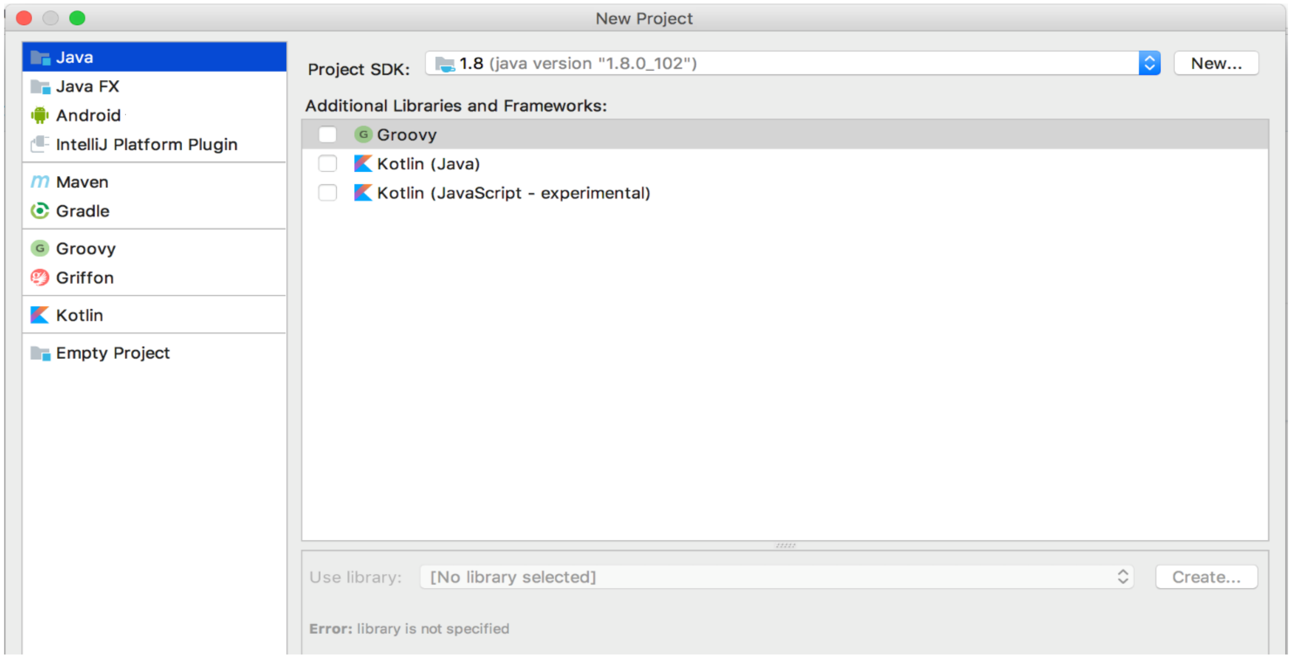
However, if you are learning Java to create Android projects and already have Android Studio installed, feel free to use that instead. They look very much alike and you will still be able to easily follow along with the videos. And if you have another IDE like Eclipse or NetBeans, you can continue to use those as well. But we do strongly recommend installing IntelliJ to avoid running into issues that were not covered here.

Creating a new Java project in IntelliJ

First thing, let's run IntelliJ and select **Create New Project**



Then make sure the Project SDK has **java version 1.8** selected there. If it's not in the menu options, then make sure you have the latest Java JDK installed.



New Project

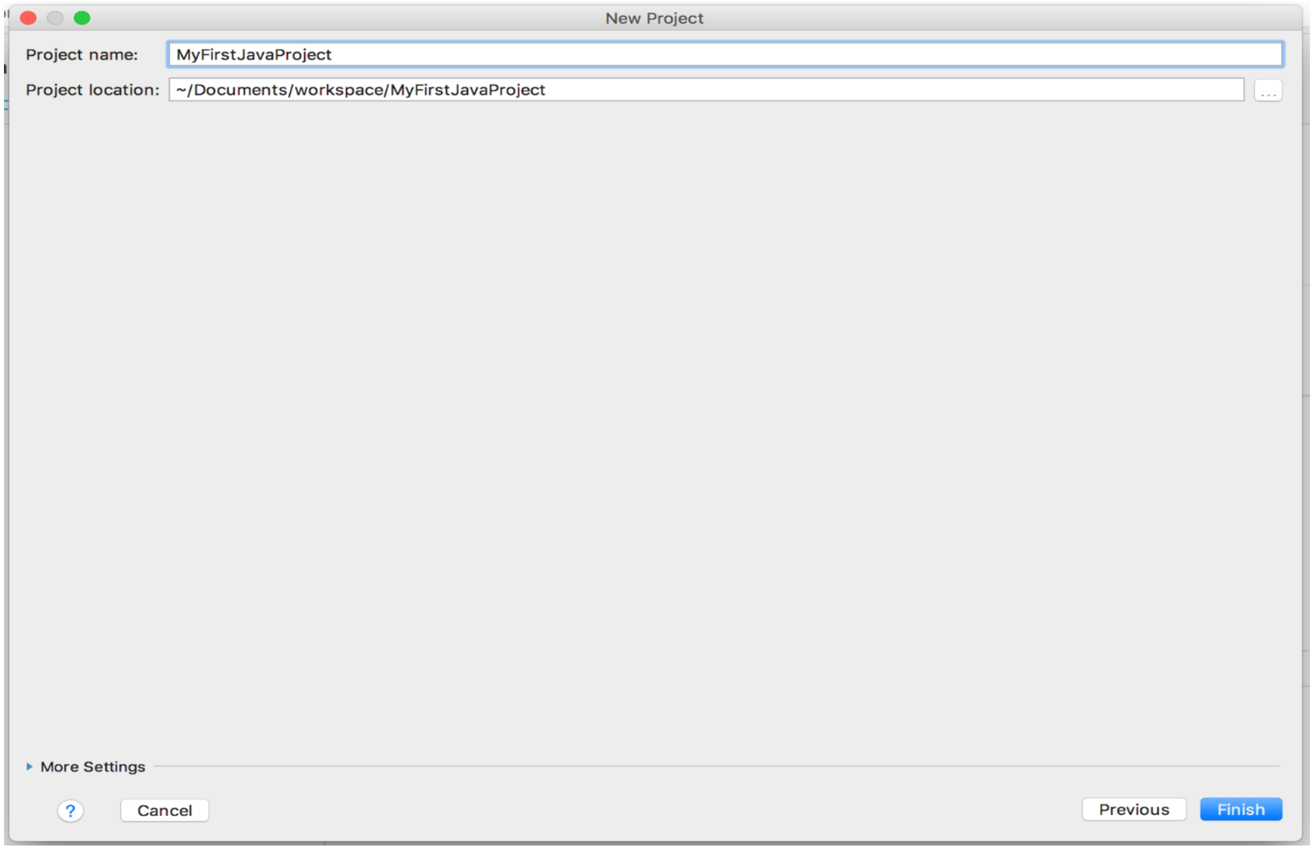
Project SDK: 1.8 (java version "1.8.0_102")

Additional Libraries and Frameworks:
- [ ] Groovy
- [ ] Kotlin (Java)
- [ ] Kotlin (JavaScript - experimental)

Use library: [No library selected]

Error: library is not specified

Use the default settings and move to the next screen, now give your new project a name, let's call it **MyFirstJavaProject**



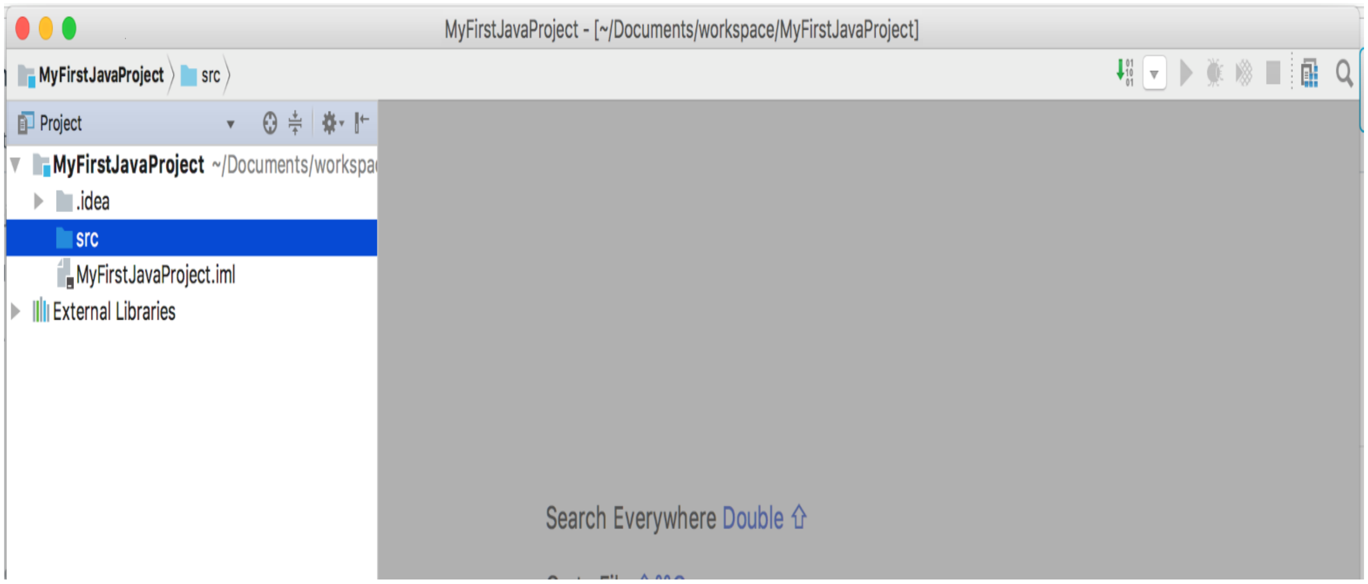
New Project

Project name: MyFirstJavaProject

Project location: ~/Documents/workspace/MyFirstJavaProject

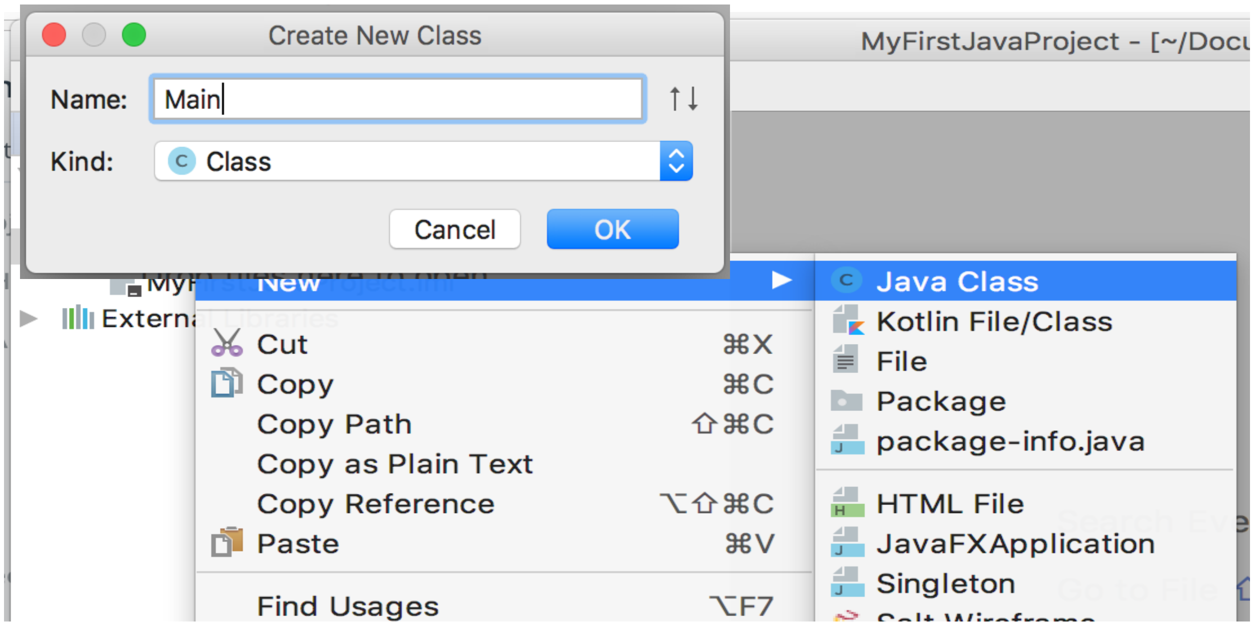
Congratulations! Now you have an empty Java Project!

On the left side you can see the folder structure of the project, the most important folder of all is the **src** folder, inside we will keep all our Java files.


MyFirstJavaProject - [~/Documents/workspace/MyFirstJavaProject]
MyFirstJavaProject
src
Project
MyFirstJavaProject ~/Documents/workspa
.idea
src
MyFirstJavaProject.iml
External Libraries
Search Everywhere Double ⇧

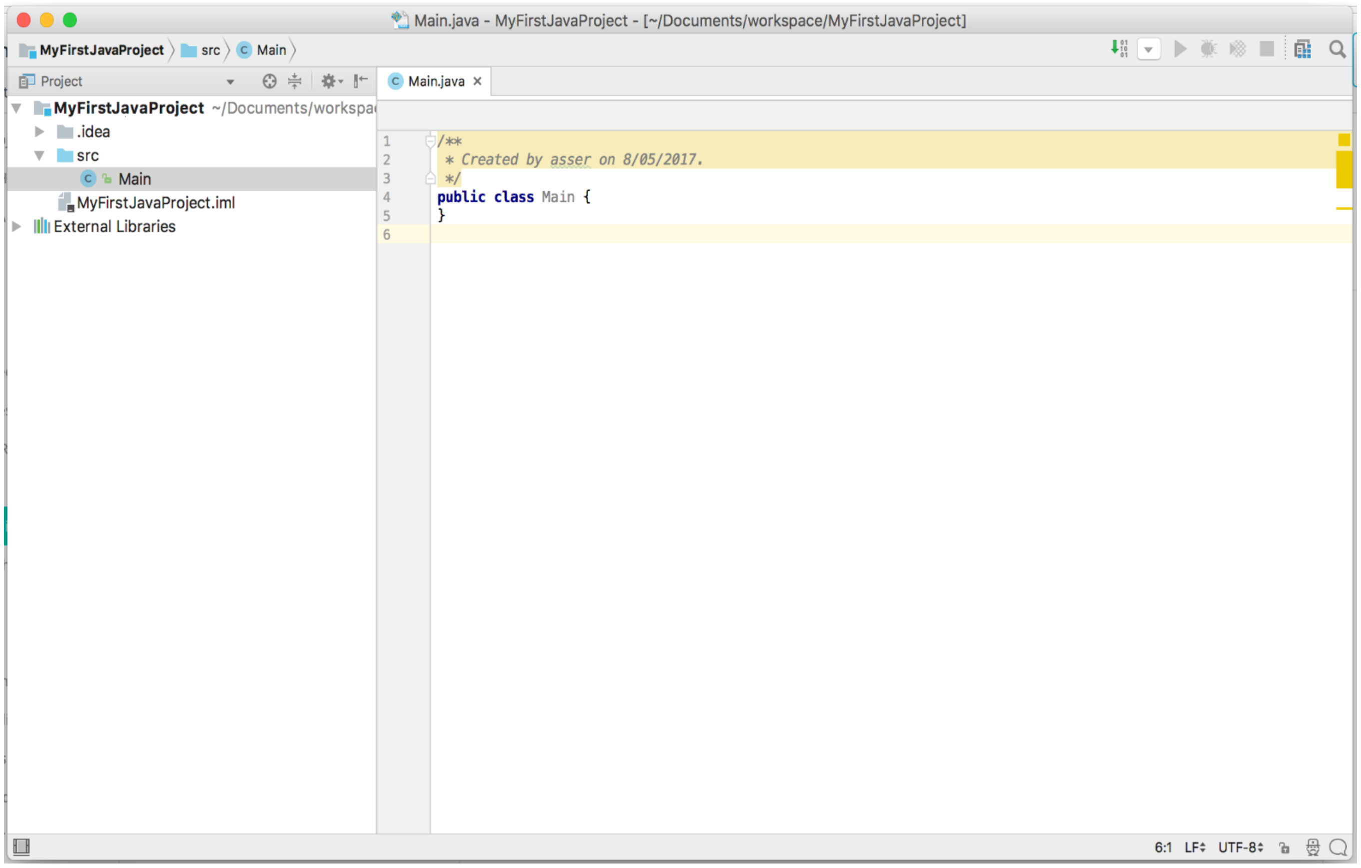
Creating a class in IntelliJ

Now to create our first class, right-click on this **src** folder and select **New** and then **Java Class**
Give your class a name, let's call it **Main** since that will be our main class



And voila! We now have an actual class file inside our project.

You can see that IntelliJ has automatically created a file inside the **src** folder with the same name of the class **Main**.



Main.java - MyFirstJavaProject - [~/Documents/workspace/MyFirstJavaProject]
MyFirstJavaProject
src
Main
Project
Main.java
MyFirstJavaProject ~/Documents/workspa
.idea
src
Main
MyFirstJavaProject.iml
External Libraries
6:1 LF÷ UTF-8÷

```
/**
 * Created by asser on 8/05/2017.
 */
public class Main {
}
```

Now that we have our first class created, let's move to the next part where we get to add methods and fields to that class.

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:

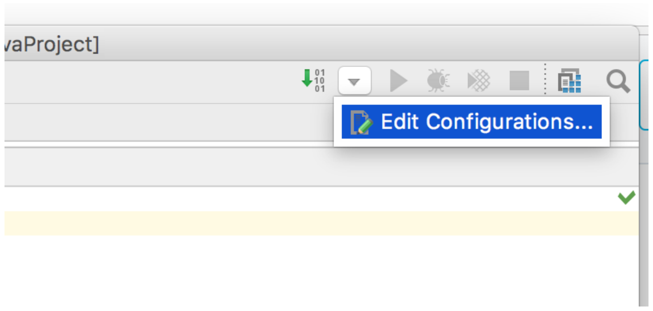
```
public static void main(String [] args){
}
```

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

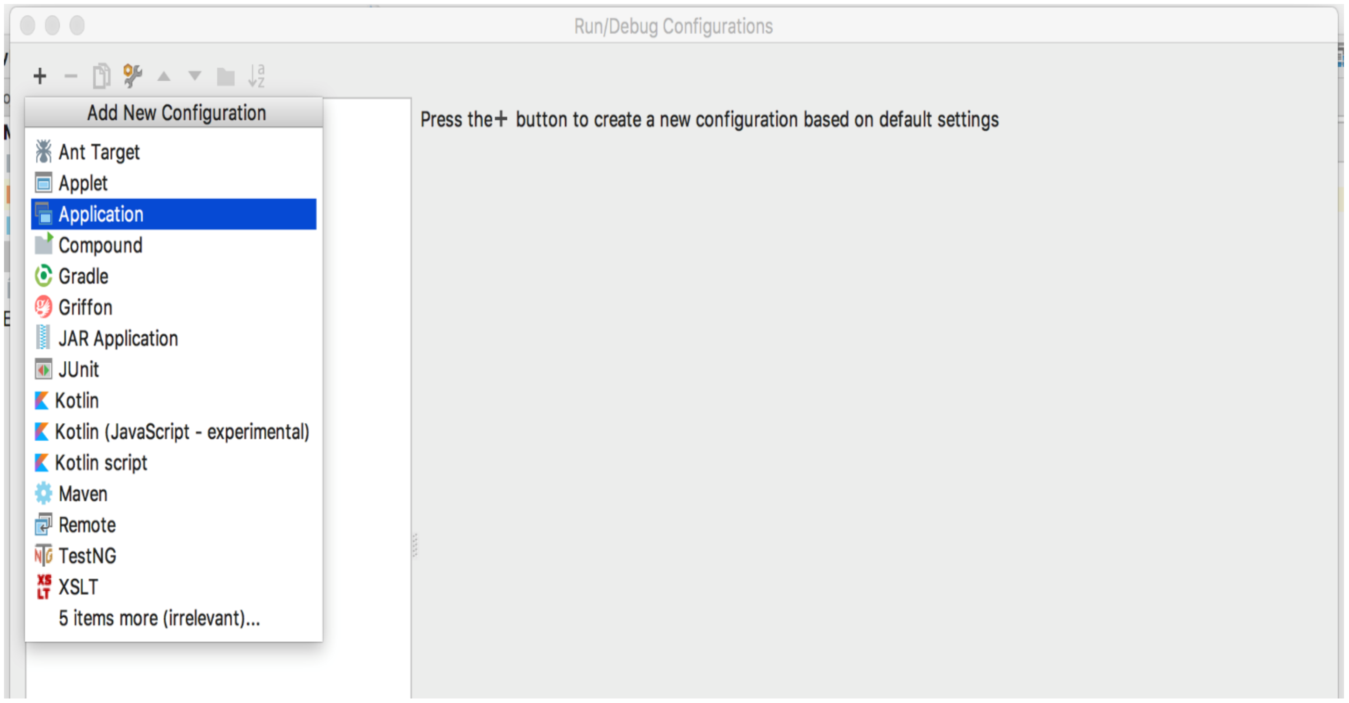
```
public static void main(String [] args){
  System.out.println("Hello world!");
}
```

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**



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

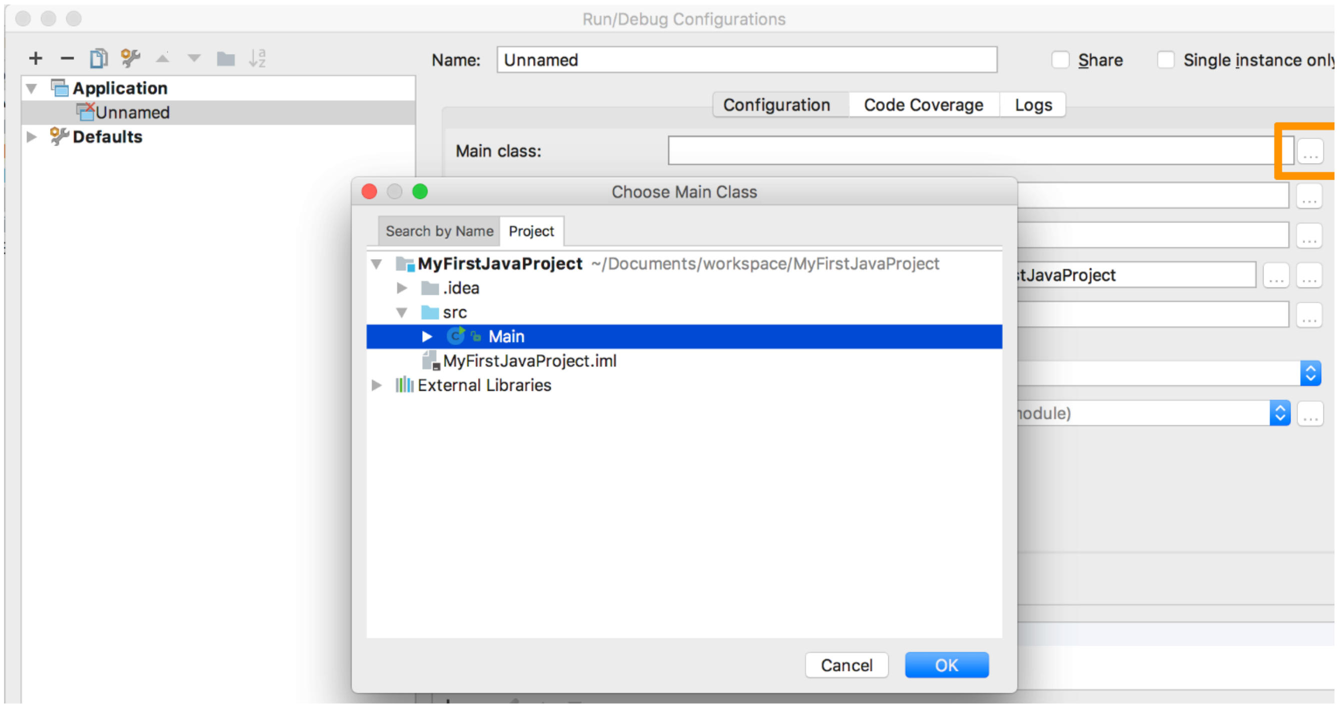

Run/Debug Configurations

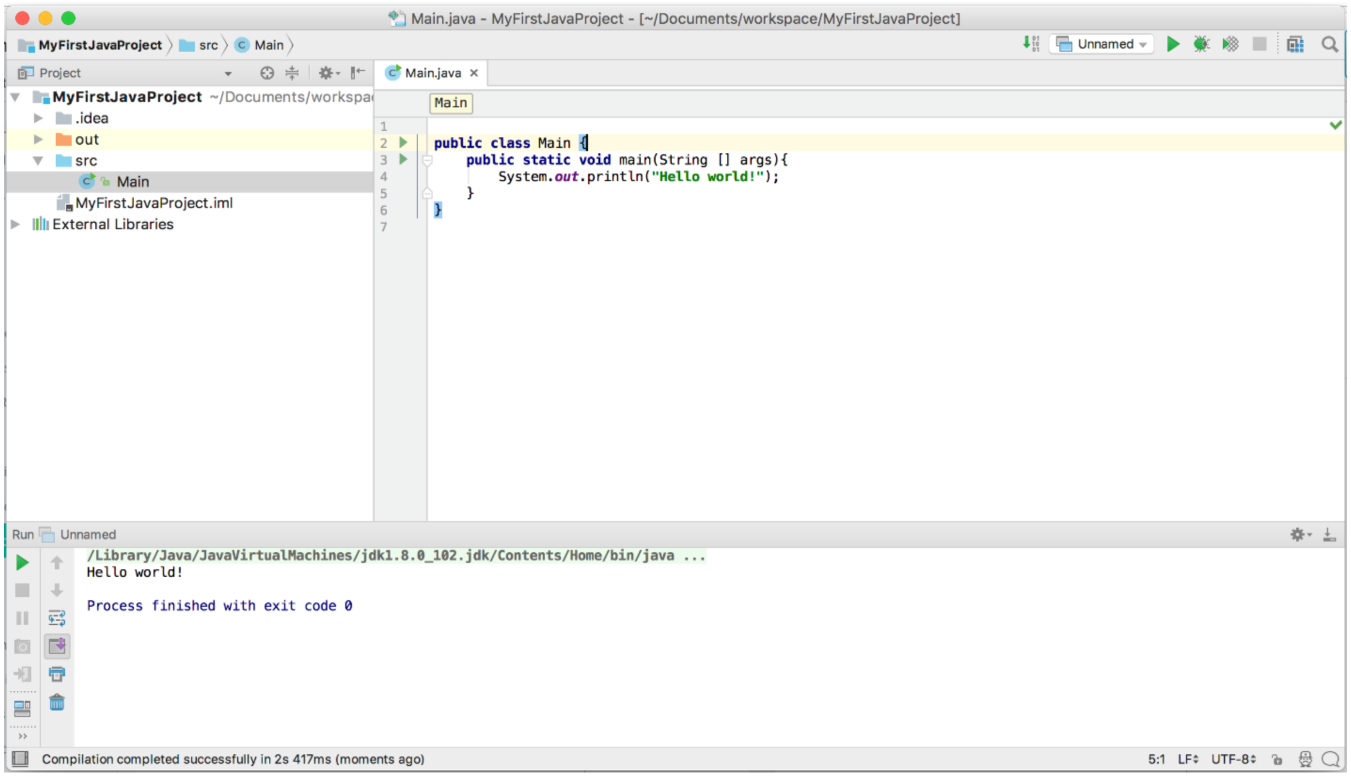
Press the+ button to create a new configuration based on default settings

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.

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

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!




Main.java - MyFirstJavaProject - [~/Documents/workspace/MyFirstJavaProject]
MyFirstJavaProject
src
Main
Unnamed
Project
MyFirstJavaProject ~/Documents/workspa
.idea
out
src
Main
MyFirstJavaProject.iml
External Libraries
Main.java
Run
Unnamed
Compilation completed successfully in 2s 417ms (moments ago)
5:1 LF÷ UTF-8÷

```java
public class Main {
    public static void main(String [] args){
        System.out.println("Hello world!");
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/bin/java ...
Hello world!

Process finished with exit code 0
```

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 `mScore`field

```
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. `null`objects 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 we 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.

Example: The Contacts Manager

The Contacts Manager

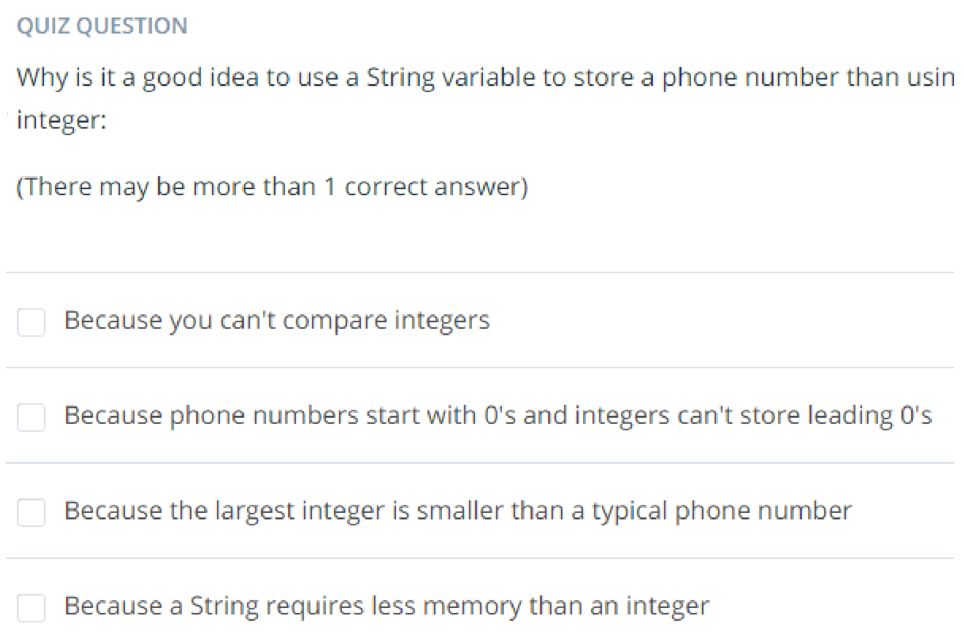
Assume you're writing a Java program that's responsible for storing and managing all your friends' contact information.

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 usin 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

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 new keyword 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 `addContact`method 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.

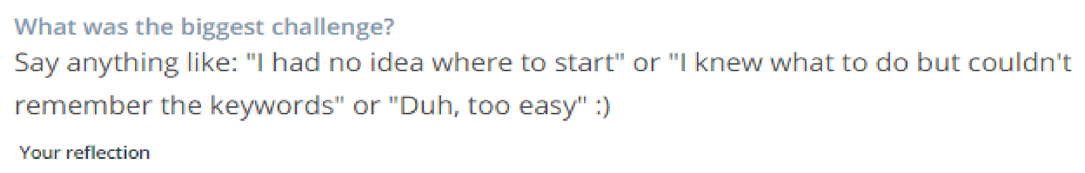
Programming Quiz

Now it's your turn, go ahead and create the ContactsManager program on your computer, then in the main method write the following:

1. Create a **ContactsManager** object called myContactsManager using the default constructor (we've already done so in the previous page)
2. Create a new **Contact** variable
   - Set the `name` to a friend's name
   - Set the `phoneNumber` field to their phone number
3. Call the `addContact` method in myContactsManager to add that contact
4. Repeat steps 2 and 3 for 4 more contacts
5. Search for a contact using the method `searchContact` and print out their phone number.

Once you're done, go ahead and submit a comment on what was your biggest challenge completing this exercise. If you are completely stuck check the solution in the next page.

Solution: Contacts Manager



What was the biggest challenge?

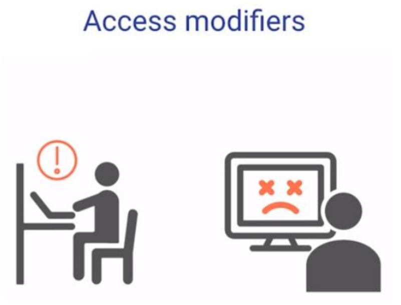
Say anything like: "I had no idea where to start" or "I knew what to do but couldn't remember the keywords" or "Duh, too easy" :)

Your reflection

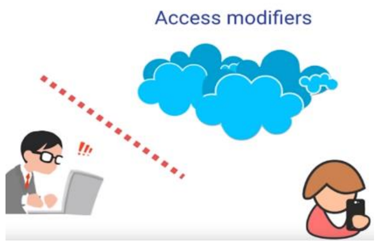
Here's a sample solution on how to use the ContactsManager object in the main method:

```
public static void main(String [] args){
  // Create the ContactsManager object
  ContactsManager myContactsManager = new ContactsManager();
  // Create a new Contact object for James
  Contact friendJames = new Contact();
  // Set the fields
  friendJames.name = "James";
  friendJames.phoneNumber = "0012223333";
  // Add James Contact to the ContactsManager
  myContactsManager.addContact(friendJames);
  // Create a new Contact object for Cezanne
  Contact friendCezanne = new Contact();
  // Set the fields
  friendCezanne.name = "Cezanne";
  friendCezanne.phoneNumber = "987654321";
  // Add Cezanne Contact to the ContactsManager
  myContactsManager.addContact(friendCezanne);
  // Create a new Contact object for Jessica
  Contact friendJessica = new Contact();
  // Set the fields
  friendJessica.name = "Jessica";
  friendJessica.phoneNumber = "5554440001";
  // Add Jessica Contact to the ContactsManager
  myContactsManager.addContact(friendJessica);

  // Now let's try to search of a contact and display their phone number
  Contact result = myContactsManager.searchContact("Jessica");
  System.out.println(result.phoneNumber);

}
```

Access Modifiers

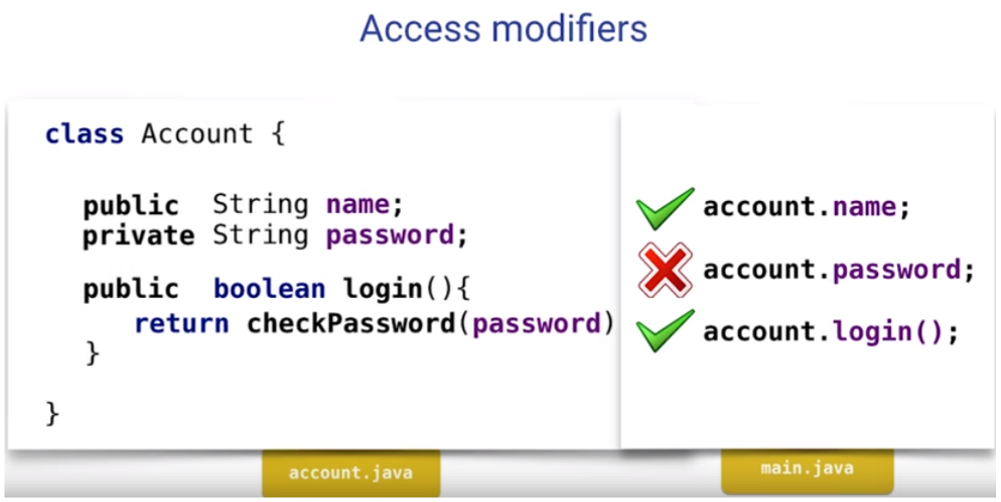


Let's face it, we all make mistakes and an innocent mistake in some program can lead to catastrophic results. So, it is imperative that we protect ourselves from our own.



A well-designed Java code is one that wouldn't even allow you to create bugs by mistake. One thing that can help is using the correct **access modifiers**. Think of it as if you're uploading photos to the cloud. Some of them you'd like to make **public** and share with others, while other photos are more of a personal nature and you'd like to keep them **private**.

Well, in Java, you can label a **field** or a **method** with either **public or private**. Making a field or a method **public**, means that other classes can access it. **Private** ones are meant to be kept hidden from the rest of your program and can only be used inside the class that created it.



Access modifiers

```
class Account {

   public  String name;
   private String password;

   public  boolean login(){
      return checkPassword(password)
   }

}
```

```
account.name;
account.password;
account.login();
```

Simply, adding the keyword **private** or **public** right before declaring the field or the method defines where it can be accessed. Let's have a look at some of the examples.

Fields (public vs private)

To label a field as private or public simply add the modifier just before the field type when declaring it:

```
public int score;
private String password;
```

You always have the final call on which fields you'd want to make public vs private, and it always depends on the purpose of the field as well as the overall design of your code.

However, it's strongly recommended in Java to label ALL fields as **private**:
For example , when defining a Book class, instead of saying:

```
class Book{
  String title;
  String author;
}
```

A proper way would be to define everything private, and only initialize them in the construtor.

```
class Book{
  private String title;
  private String author;
  public Book(String title, String author){
    this.title = title;
    this.author = author;
  }
}
```

This way you can guarantee that once a book object has been created, the title and author will never change!

But sometimes you need to have fields that can be modified by other classes.

For example, if we wanted to keep track of whether a Book is being borrowed or not, you can add a public boolean field to do so:

```
public class Book{
  private String title;
  private String author;
  public boolean isBorrowed;
  public Book(String title, String author){
    this.title = title;
    this.author = author;
  }
}
```

This will work, since you will be able to do something like this from anywhere in your project:

```
book.isBorrowed = true;
```

However, it's still slightly risky, and you could end up mistakingly setting the boolean to true when you only meant to check if it was true or false!

A better design would be to declare that field as private and then create public methods that return the value of such hidden field (known as *getters*) as well as public methods that provide a way to set or change its value (known as *setters*)

```
public class Book{
  private String title;
  private String author;
  private boolean isBorrowed;
  public Book(String title, String author){
    this.title = title;
    this.author = author;
  }
  public void borrowBook(){
    isBorrowed = true;
  }
  public void returnBook(){
    isBorrowed = false;
  }
  public boolean isBookBorrowed(){
    return isBorrowed;
  }
}
```

Setting the isBorrowed field as private will prevent you from mistakenly changing its value somewhere in the code, because the only way to change it now is to call either borrowBook() or returnBook() which is much more explicit.
And to be able to read the value of isBorrowed, we've created a getter method called isBookBorrowed() that is public and simply returns the value of isBorrowed

Summary

- Always try to declare all fields as **private**
- Create a constructor that accepts those private fields as inputs
- Create a public method that *sets* each private field, this way you will *know* when you are changing a field. These methods are called **setters**
- Create a public method that *returns* each private field, so you can read the value without mistakingly changing it. These methods are called **getters**

Methods (public vs private)

With methods, it's common to have a mix of private and public methods.

The private methods are usually known as **helper methods**, since they can only be seen and called by the same class, they are usually there to organize your code and keep it simple and more readable.
The public methods are the actual actions that the class can perform and are pretty much what the rest of the program can see and call.

Here's an example of when you'd want to use public methods vs private methods

```
class Person{
  private String userName;
  private String SSN;
  private String getId(){
    return SSN + "-" + userName;
  }
  public String getUserName(){
    return userName;
  }
  public boolean isSamePerson(Person p){
    if(p.getId().equals(this.getId()){
      return true;
    }
    else{
      return false;
    }
  }
}
```

The class Person has both its fields set to **private** because if they were public then any other class will be able to change such sensitive information. Setting them to private means that only methods and constructors inside this class can do so!
The method getId() was also set to private so that no other class can know the social security number of any person.
However, we were still able to use that method internally when comparing this person with another person object in the isSamePerson(Person p) method.
This means that any other class can only call getUserName or isSamePerson and will seem as if these are the only 2 methods provided by the class Personژ

Public classes

Even classes can be labeled as **public** or **private**! And even though you are allowed to label a class as **private**, it requires you to know about **nested classes** (https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html) which is not covered in this course.
So for now, make sure you label all your classes **public**.

What if I don't use any label at all?

We've been doing that so far anyway! What does that mean?

It's not recommended to do so, but if you do, it will default to something called "package public" which means it's as if you've labeled them **public** but only to the classes that are in the same package/folder. We will learn more about packages later. But again, it's always a good idea to label everything explicitly.

But but but

... wait a second! Since I have access to all the code, can't I just go back and change all the private methods and fields and make everything public and control the universe!

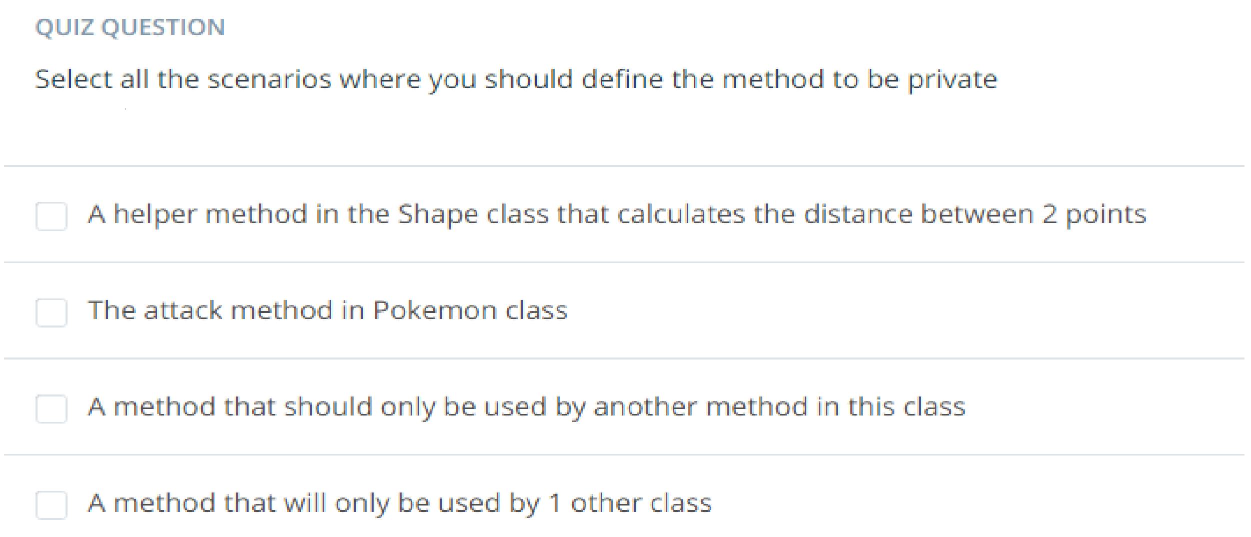
Yes, but remember that this is for your own good. You're trying to design your code in a way that will prevent you from doing things you don't want to happen! You're also most likely going to be working with a team of other developers and setting the correct access modifiers helps communicate with everyone the intended use of each part of your project.

Summary

To summarize, it's recommended to:

- Set all your classes to **public**
- Set all your fields to **private**
- Set any of your methods to **public** that are considered actions
- Set any of your methods to **private** that are considered helper methods

Quiz : Access Modifiers



QUIZ QUESTION

Select all the scenarios where you should define the method to be private

- ☐ A helper method in the Shape class that calculates the distance between 2 points
- ☐ The attack method in Pokemon class
- ☐ A method that should only be used by another method in this class
- ☐ A method that will only be used by 1 other class

Conclusion:

- By now you know a fair bit about object-oriented programming and how to create classes and objects, also how to use them to design better programs.

- You've also learned that any class is made up of constructors, fields, and methods and you know how to hide or share any parts of a class using access modifiers by sending them to either public or private.

In the next lesson we'll start building more interactive programs by *accepting and handling user input*. We'll also get to write code that reads and writes directly into text files which can be stored permanently on your computer.

Part Two: User Interaction

Reading User Input

To build a useful job application you'll want to make it as interactive and fun as possible. That means allowing the user to provide information at runtime. For example, for a context manager application it has some useful methods but to use them we had to write all the code in the main method including all your friends' contact details.

I don't know why anybody would want to do that. I sure wouldn't. We certainly shouldn't be asking users to write Java code and recompile it every time they want to make a change. That would be crazy.

Instead Java allows you to accept input from the user while the program is running. That means we can write our main method in such a way that ask the user to input their friends' names and phone numbers and chat IDs and all that cool stuff and simply pass that information on to be stored. Let's have a look at how to do that in Java.

Runtime Inputs

There are **4** different ways a Java program can read input from the user:

1. Runtime input
2. Files
3. Command line arguments
4. Graphical User Interface (GUI)

We will be covering the first **3 types** in details this lesson, and you will need to use all of them to complete the project.

GUI

GUI (Graphical User Interface) is a totally different story and won't be covered in this course. In short, it allows you to create buttons and move images and colors. It certainly makes any application you build look much cooler, so it's worth checking out.

Java for Android

If you are planning to use Java mainly to build Android applications then don't worry about learning any GUI yet. Android has its own layout system and it's really easy to learn.

Input Scanner

The most flexible and common way to read an input from a user is by asking them to type in something and wait till they respond.

Just like we've seen how this command will print **"Hello World"** to the command line as an output:

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

You can also ask the user to type in a message and then your Java program can read it into a variable and use it.

This is done using a Java class called Scanner.
First, to be able to access this class, you have to point your program to the java.util library that includes the Scanner class. You do that by typing this at the very top of the file

```
import java.util.Scanner;
```

We will talk more about including packages and libraries later on in this course.

A Scanner allows the program to read any data type from a particular input, if we create the scanner object like this:

```
Scanner scanner = new Scanner(System.in);
```

Then the scanner will be reading from the System's input (hence System.in) which is basically the command line.
It will continue to read whatever the user is typing until they hit "enter" then the program continues to execute.

Once the scanner object has been created, you can use it to read a String, an integer or an entire line.

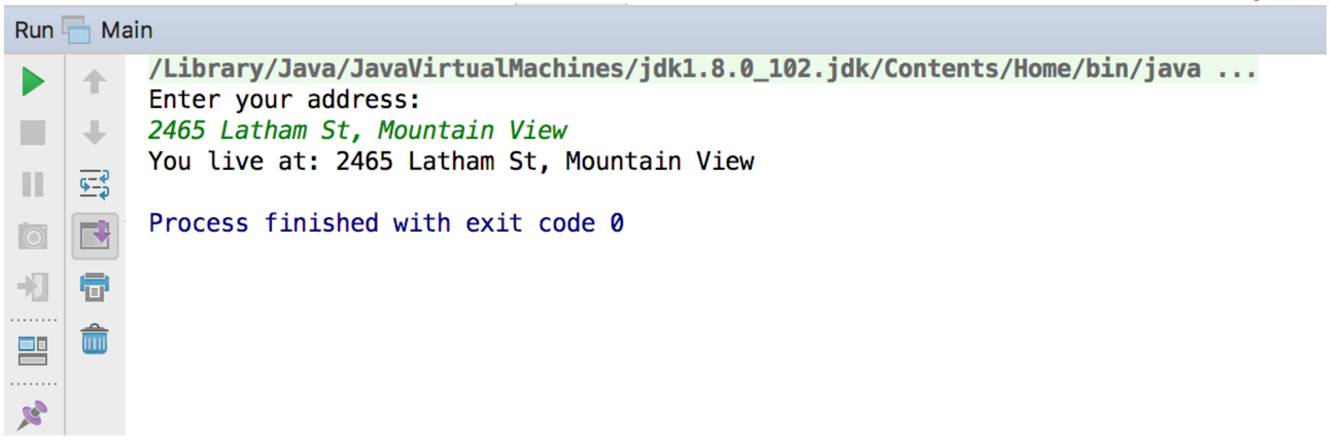
Calling the method nextLine() in that scanner object will return a String that contains everything the user has typed in before they hit "enter".

```
scanner.nextLine();
```

For example:

```
System.out.println("Enter your address: ");
Scanner scanner = new Scanner(System.in);
String address = scanner.nextLine();
System.out.println("You live at: " + address);
```

The above code will wait until the user types in their address, then stores it into the variable address and then prints it back to the user.
Go ahead and try it out yourself! You should get an output that looks like this:



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/bin/java ...
Enter your address:
2465 Latham St, Mountain View
You live at: 2465 Latham St, Mountain View

Process finished with exit code 0
```

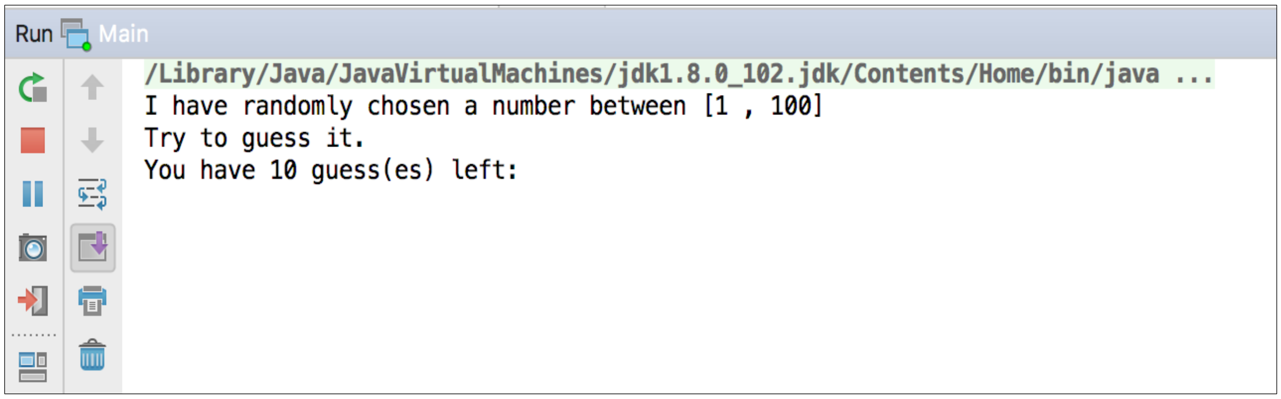
If you want to read a number into an integer variable instead of the entire line:

```
System.out.println("Enter your grade: ");
Scanner scanner = new Scanner(System.in);
int grade = scanner.nextInt();
if(grade > 90){
  System.out.println("Wow! you did well!");
}else{
  System.out.println("Not bad, but you can do better next time!");
}
```

Guess the number game

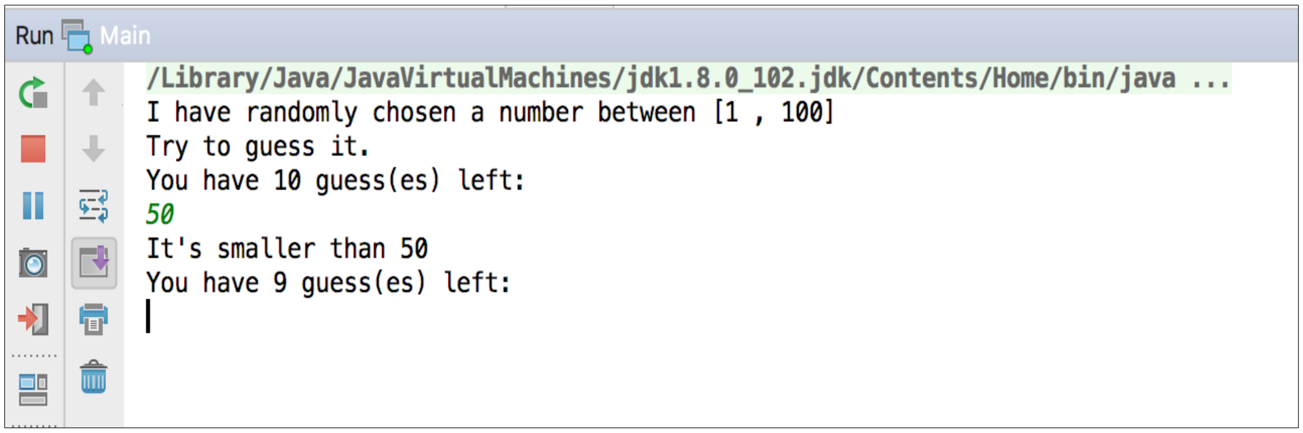
In this example, we will get to use the input scanner to build a guessing game where the computer will generate a random number between 1-100, and the user gets 10 guesses to find out what that number is.

When the game first starts it prints a few sentences explaining what's going on:



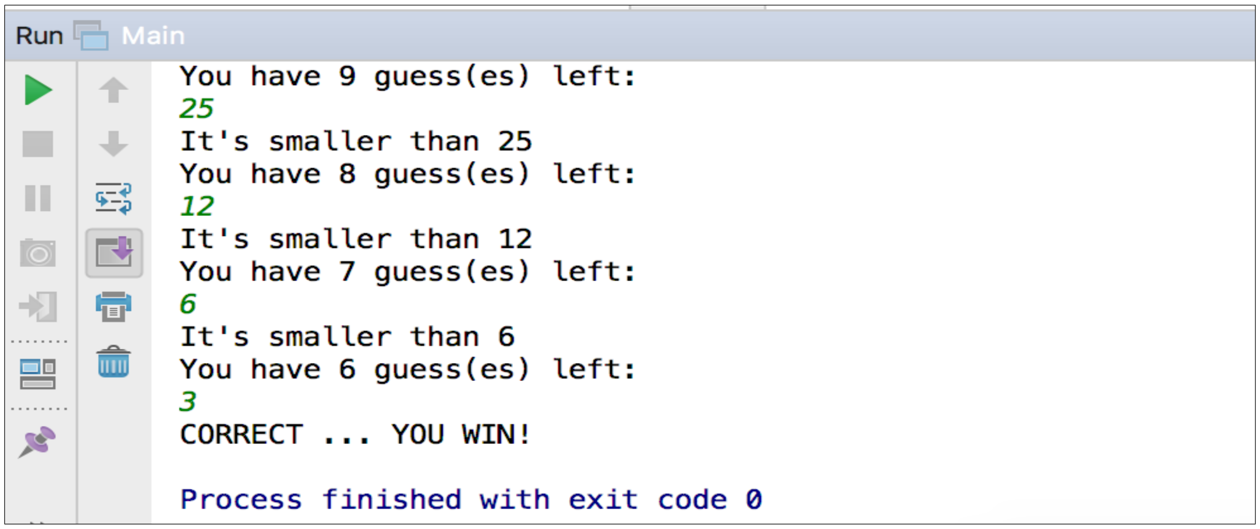
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/bin/java ...
I have randomly chosen a number between [1 , 100]
Try to guess it.
You have 10 guess(es) left:
```

Then asks the user to guess the number. Once the user types in a number and hits enter, the game will compare that guessed number with the random number it had generated and tell the user if it's smaller or larger, then they get to guess again.



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/bin/java ...
I have randomly chosen a number between [1 , 100]
Try to guess it.
You have 10 guess(es) left:
50
It's smaller than 50
You have 9 guess(es) left:
```

If the user manages to guess the number before they run out of guesses they win. Otherwise they lose!



```
You have 9 guess(es) left:
25
It's smaller than 25
You have 8 guess(es) left:
12
It's smaller than 12
You have 7 guess(es) left:
6
It's smaller than 6
You have 6 guess(es) left:
3
CORRECT ... YOU WIN!

Process finished with exit code 0
```

Let's walk through how to create this game step by step ...

Building the game

In this example we'll use the input scanner to build a guessing game. The computer will generate a random number between 0 and 100 the user gets guesses to find what that number is.

When the game first starts, it prints a few sentences explaining what's going on then it asks the user to guess the number. Once the user types a number and hits enter, the game will compare that guessed number to the random number that it generated and it will tell the user if it's smaller, larger or if it's the correct number. If the user manages to guess the number before they run out of guesses, they win. Otherwise, they lose.

Okay, the first thing is to create a new project in IntelliJ, and let's call it Number Game. And in the source directory, we'll create a new class called Number Game that will include the main method and pretty much all the code. Okay, the first thing we need to do is import the scanner class from Java.util.

This will be how we get input from the user and let's create that main method. Let's start by randomly selecting a number between 1 and 100 To do so, we'll use Math.random and multiply it by 100 The reason why we have to multiply it by 100 is because Math.random creates a floating point number between zero and one. To make sure that our number is not zero at any point, we need to add one to whatever the output is. And let's put some parentheses just to make sure everything is done in the right order. And lastly, this int here takes this floating point number that's generated from this portion and throws away the fractional part. Next, let's print a message to the user explaining the game. And for sanity check, let's run it at this point and also see what's being generated for the random number. That time it printed out, it's selected . Let's see they selected .

OK. Since the user only has guesses before they lose, let's create a loop that counts down from 10 all the way to zero. And inside that loop, the first thing we'll do is show the user how many guesses they have left. And if we run this again it will just print a count down of how many guesses you have left.

Well, let's add a message here to say a- choose again, so that they know they need to insert something.

So next, let's create a scanner to read that actual user input. And inside the loop, we'll store that in a variable called guess using scanner.nextInt. And what this does is it takes the things that you type in and it tries to convert that into an integer. And just to check, let's test that. And we can see here that it's actually reading in our guesses. And if we were to type in something that can't be turned into an integer that's when we'll get to error.

But let's assume that our users are going to follow instructions and only type in things that actually can be evaluated to numbers. Once we've read the user's guess and stirred it into the variable guess, we can now compare it to the random number that we generated earlier. If the random number was smaller than the user's guess, we want to tell them so. I'm using string concatenation here to display the value of the guess variable in this output string.

Let's next check to see if the random number is greater than what the user guessed. And in that case, we'll output a message telling them that the random number is greater than the number they guessed. For each guess there's another condition that we haven't handled yet. That's the one that checks to see if the random number is equal to the guessed number. And in that case, we want to signal to the system that the user has won.

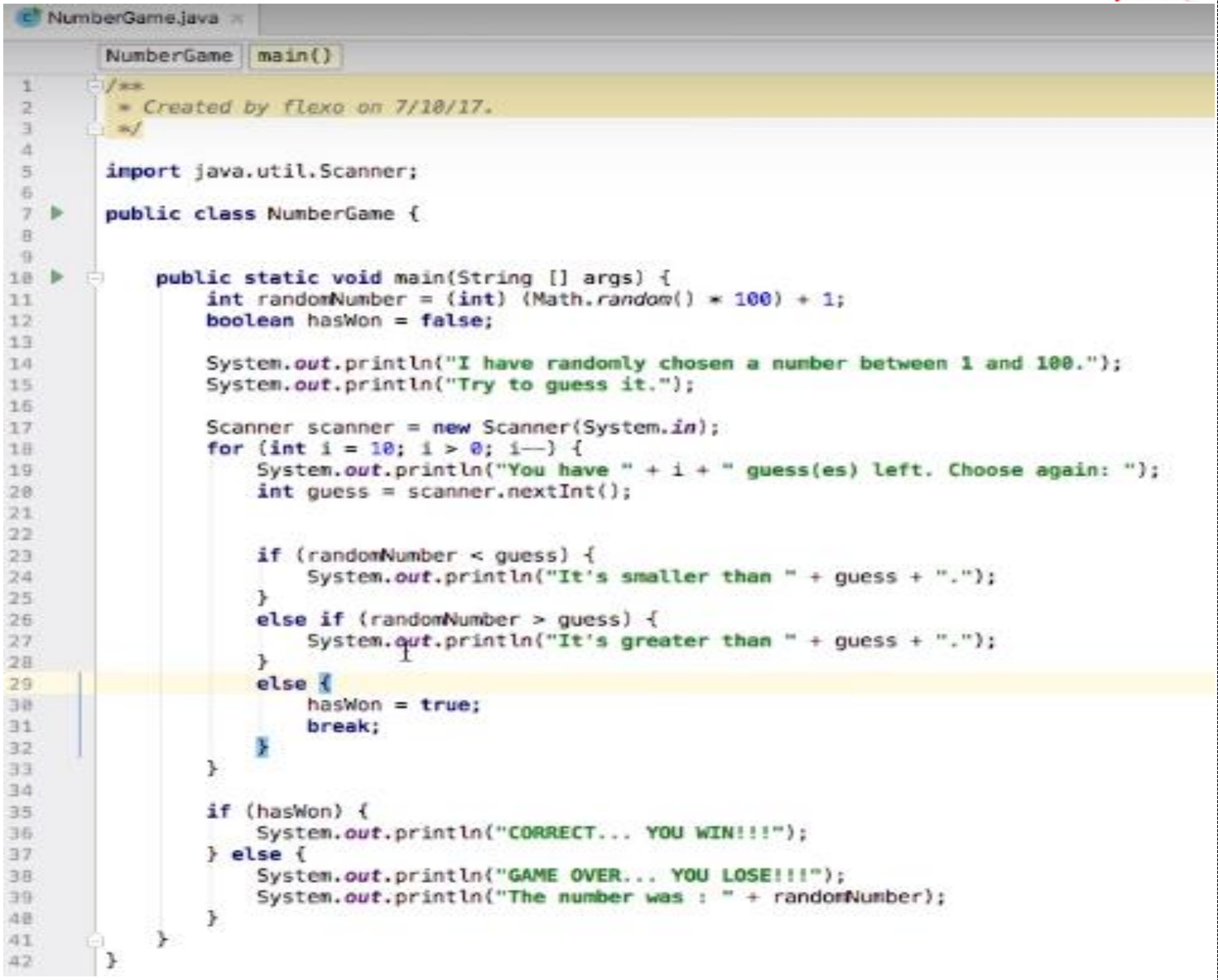
So let's create another variable,a boolean variable has one and let's initially set that to false. And here we set it to true. And because a person has won already, we want to break out of that loop so that it doesn't prompt them to guess again.

Now that we've completed our loop we need to check if they actually won or if they ran out of guesses. If has won is set to true and we can do these two different ways. We can just use has won, because it is a boolean variable or we could do the more verbose has won equals true. Both of them work. This form is just a little bit more concise. So if they have won we print out the message, "Correct. You win!!!" else, it prints out, "Game over, you lose!" and tells the user what the random number was.And let's try that out.

Let's say 50 and it's greater than 50 . Let's do it 75 , it's greater than 75 but smaller than 90 so it's between 75 and 90 . It's greater than 85 . So it's between 85 and 90 . It's 87. It's greater than 87 . Between 87 and 90 It’s 89. There you go.

All right, so our code runs correctly but there are a couple of different things that we can do to make this code better. So in our loop we have separate if statements for if the random number is less than a guess, if it's greater than the guess and if it equals the guess.

For any number, it can't be this, all three of these states are multiples of these states. So if our random number is 15 , and the guess is one, only one of these is going to be correct. So to avoid having each of these if statements be checked each time, we can use else statements so, put else if, so if it is smaller, it stops running here. And then, here we'll just do else because if the number is neither less than or greater than the random number it has to equal the random number.



```java
/**
 * Created by flexo on 7/10/17.
 */

import java.util.Scanner;

public class NumberGame {


    public static void main(String [] args) {
        int randomNumber = (int) (Math.random() * 100) + 1;
        boolean hasWon = false;

        System.out.println("I have randomly chosen a number between 1 and 100.");
        System.out.println("Try to guess it.");

        Scanner scanner = new Scanner(System.in);
        for (int i = 10; i > 0; i--) {
            System.out.println("You have " + i + " guess(es) left. Choose again: ");
            int guess = scanner.nextInt();


            if (randomNumber < guess) {
                System.out.println("It's smaller than " + guess + ".");
            }
            else if (randomNumber > guess) {
                System.out.println("It's greater than " + guess + ".");
            }
            else {
                hasWon = true;
                break;
            }
        }

        if (hasWon) {
            System.out.println("CORRECT... YOU WIN!!!");
        } else {
            System.out.println("GAME OVER... YOU LOSE!!!");
            System.out.println("The number was : " + randomNumber);
        }
    }
}
```

Interesting fact

Given 10 chances to guess a number between 1 and 100 (with "smaller than" or "greater than" feedback), is more than enough for you to ALWAYS win!

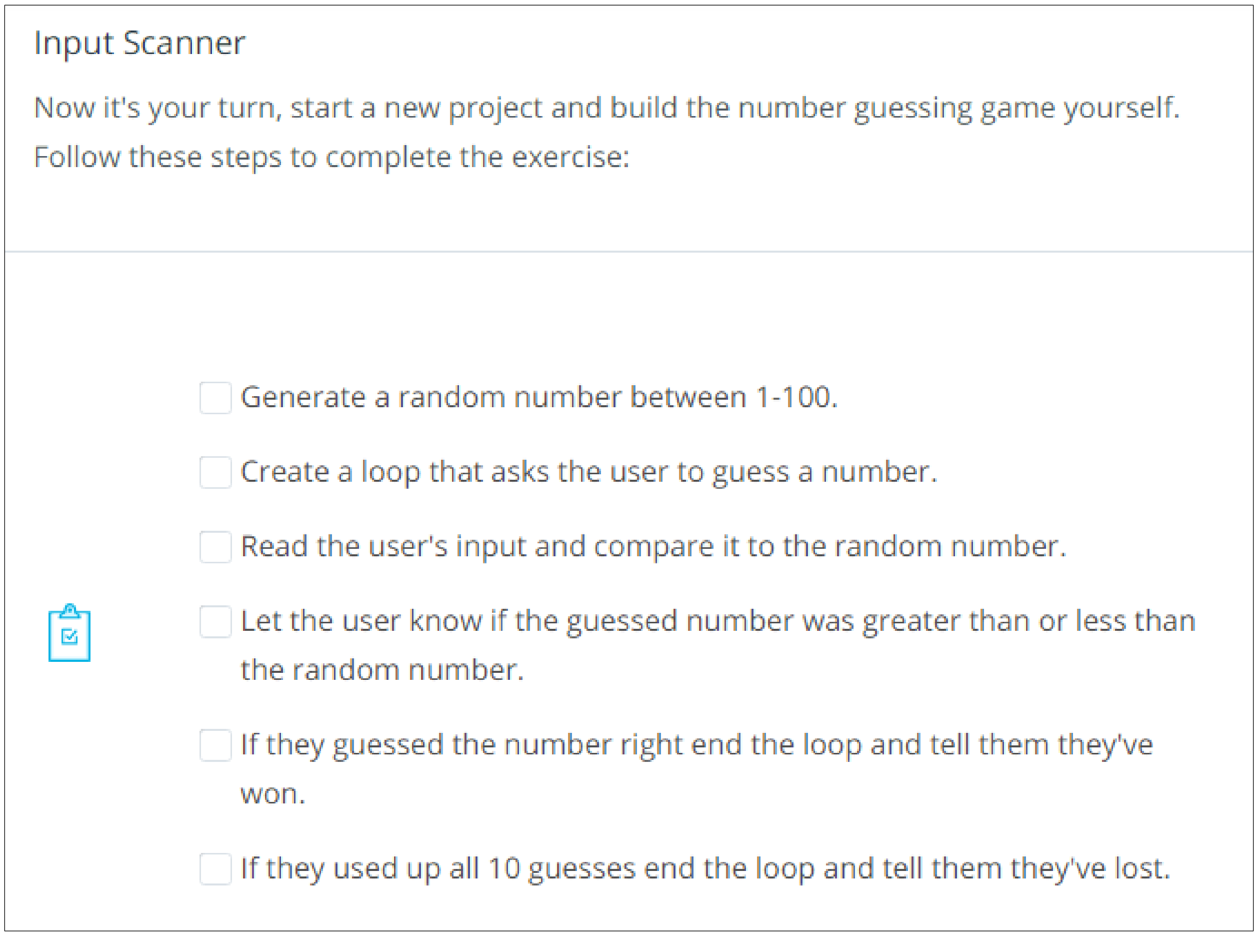
The trick is to use something called the **Binary Search Algorithm** It's a very clever search technique where you cut your search range by half every time you make a new guess:

- You start with the range **[1 - 100]**

- Always start your first guess as **50** (midpoint between 1 and 100)
- If the random number was **smaller**, then it must be between **[1 - 50]**
- If the random number was **greater**, then it must be between **[50 - 100]**
- Repeat the same technique by guessing the midpoint of the new range:

- If the new range is **[1 - 50]** then guess **25**
- If the new range is **[50 - 100]** then guess **75**
- And so on, until you get it right.
  In fact, you will only need **7** guesses at most (for the range 1-100).
  Binary Search is a very popular algorithm used by computer scientists all the time. Go ahead and try this strategy while playing the game ;)

Exercise: Input Scanner



Input Scanner

Now it's your turn, start a new project and build the number guessing game yourself. Follow these steps to complete the exercise:

☐ Generate a random number between 1-100.

☐ Create a loop that asks the user to guess a number.

☐ Read the user's input and compare it to the random number.

☐ Let the user know if the guessed number was greater than or less than the random number.

☐ If they guessed the number right end the loop and tell them they've won.

☐ If they used up all 10 guesses end the loop and tell them they've lost.

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 ..
}
```

Word Count

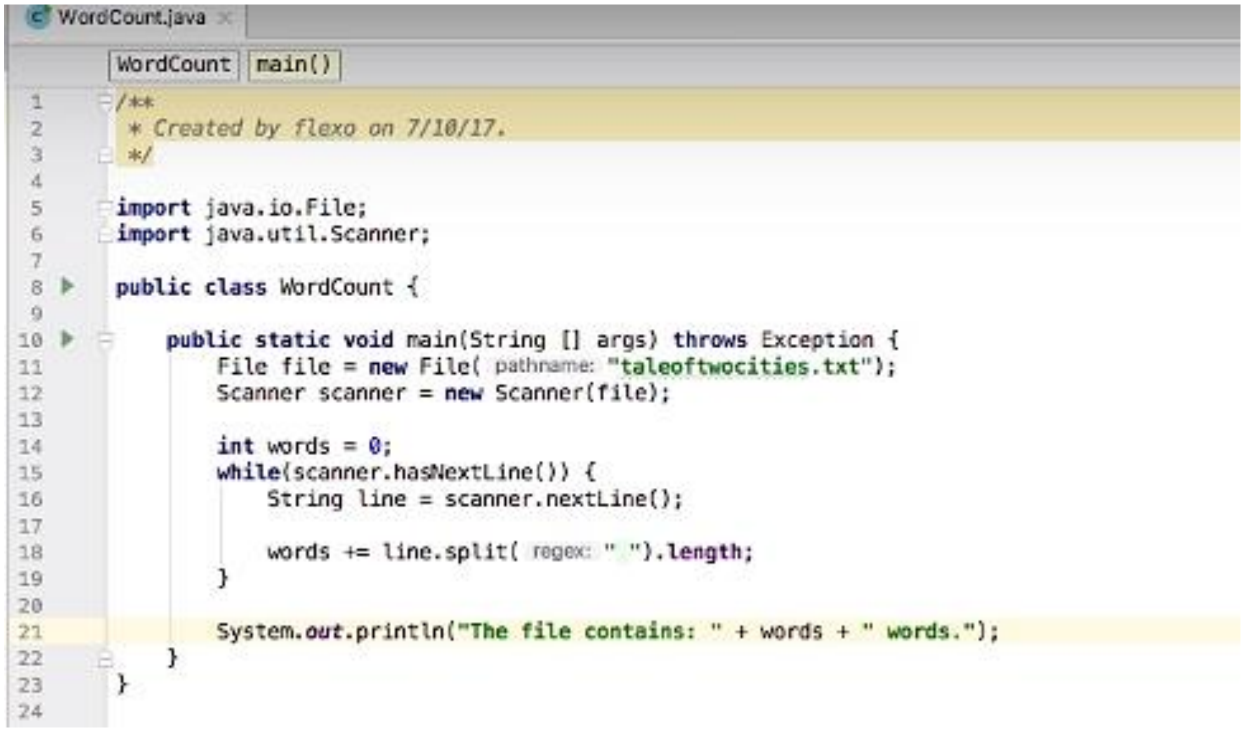
In this example, we'll use the file scanner to count the number of words in a text file. Let's start by downloading a text file. Project Gutenberg (www.gutenberg.org) offers a collection of public domain novels and books that you can use and download for free. Feel free to download your favorite book but I'm going to use "A Tale of Two Cities by Charles Dickens. Make sure you download the plain text version.

Now that we have a text file, let's create a new project in IntelliJ to count the number of words in it. So we'll create new project and just leave it as a Java project. And let's call it WordCount. And let's create a Java class also called WordCount. We'll need to import two packages at this time. One for the Scanner like we did in the previous exercise and another for the File class that we'll use to open and read the file's contents. And let's create our main method. Before we open the text file, we'll need to actually add it to this project. Let's start by creating a file object with the name File. And for the parameter, we're going to type in the name of the text file that we just added to our projects, taleoftwocities.txt.

When you open a file, you might get a file exception if there is a problem with reading it or if the file isn't there. So we'll need to account for that in our main method by saying that this function can throw an exception. Now that we have a file object, we can use a scanner object to read its contents. Notice that I've passed the file object to the scanner constructor instead of system.in like we did before.

This is because we want the scanner to scan the text file and not the user's input. And let's go ahead and run this. This isn't going to print out anything but it's a good exercise to run your code to make sure that you're on the right path. Cool. We don't have any error messages, so we know that everything is working properly.

Just for fun, let's change the name of the file to something that we know doesn't exist. And as we can see, it throws an exception, a file not found exception, that it couldn't find a file with that name.



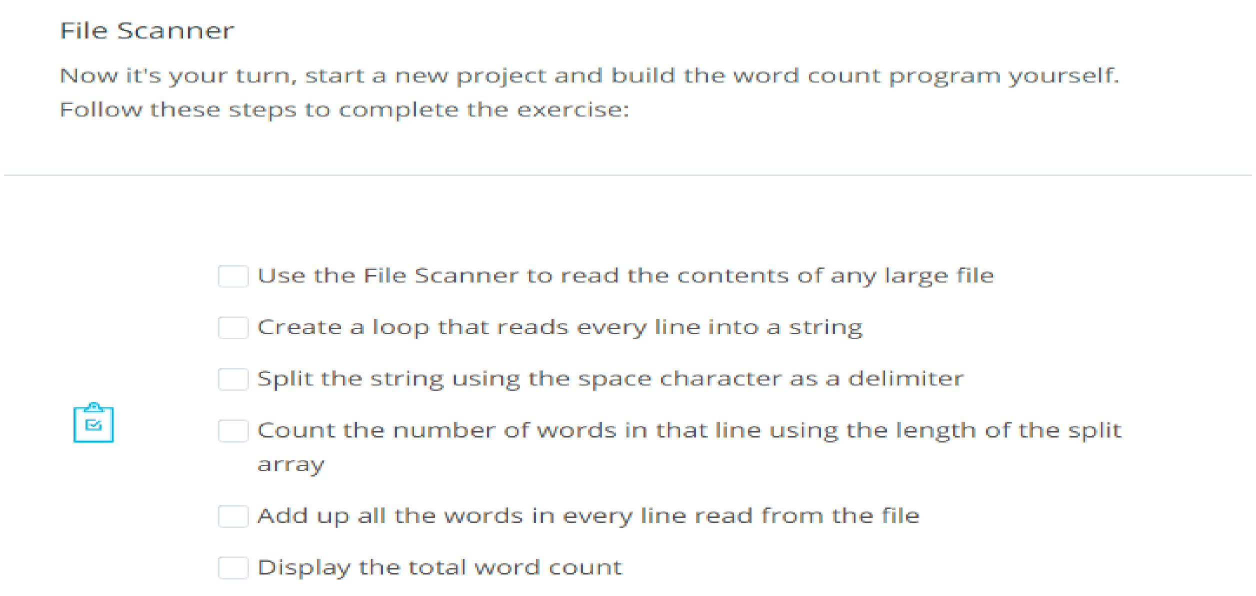
```java
/**
 * Created by flexo on 7/10/17.
 */

import java.io.File;
import java.util.Scanner;

public class WordCount {

    public static void main(String [] args) throws Exception {
        File file = new File( pathname: "taleoftwocities.txt");
        Scanner scanner = new Scanner(file);

        int words = 0;
        while(scanner.hasNextLine()) {
            String line = scanner.nextLine();

            words += line.split( regex: " ").length;
        }

        System.out.println("The file contains: " + words + " words.");
    }
}

```

But let's go ahead and fix that. Once we have the scanner object, we can start reading the file line-by-line using the nextLine and hasNextLine methods. And we want the scanner to loop through the whole file. So we'll need to put this in a while loop. Now you probably remember that the while loop expects whatever expression is placed between the parentheses to evaluate to a boolean. hasNextLine returns true or false. So it satisfies that already.

It would be a little bit more redundant to check explicitly if it equals true but that's also a valid code. But that's a little bit more code than you need to do, and IntelliJ will give you a message to simplify it. So inside of our while loop, we'll look at the contents of each line. So let's create a variable called Line that stores scanner nextLine which return to string. To count the number of words in that line, we'll want to split it, and we'll want to split it on the spaces. So if we stop right here and run our code again, each line of output gives us the array object that was created from splitting the line. But that doesn't really give us that much information, so let's look at the length. And now, it will output how many words are in each line.

But instead of each line, we want to do this for the whole file. So, we'll need to create another variable called words or word count and initialize that to zero to store a word count. Inside the while loop, instead of printing out each time that we check the number of words in a line, we're going to add them to the word variable, and add a message printing out the number of words in that file. And it says that A Tale of Two Cities has Keep in mind that our definition of a word is simply anything that's separated by a space. So the total count could be slightly different from other word count shown on text editors, for example, but it should be pretty close.

Exercise: File Scanner



File Scanner

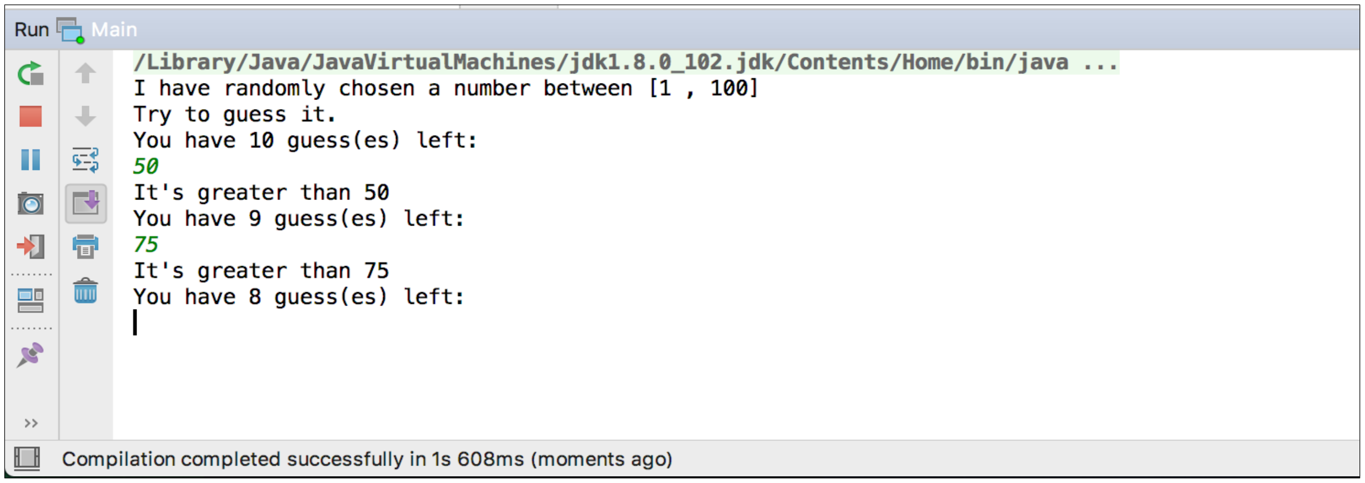
Now it's your turn, start a new project and build the word count program yourself. Follow these steps to complete the exercise:

- ☐ Use the File Scanner to read the contents of any large file
- ☐ Create a loop that reads every line into a string
- ☐ Split the string using the space character as a delimiter
- ☐ Count the number of words in that line using the length of the split array
- ☐ Add up all the words in every line read from the file
- ☐ Display the total word count

Using the terminal

Running a program in IntelliJ

So far we've been running our Java projects directly from IntelliJ. The output gets displayed in the output pane at the bottom.



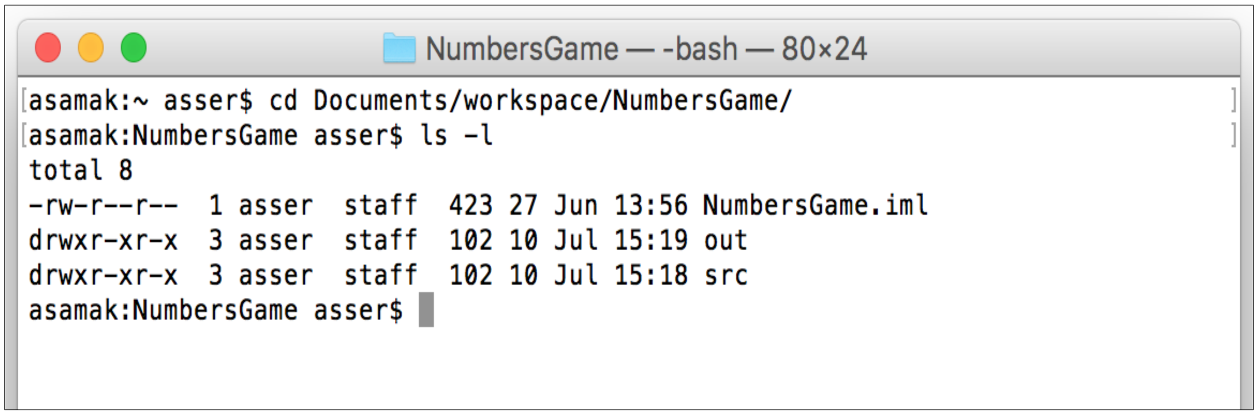
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/bin/java ...
I have randomly chosen a number between [1 , 100]
Try to guess it.
You have 10 guess(es) left:
50
It's greater than 50
You have 9 guess(es) left:
75
It's greater than 75
You have 8 guess(es) left:
```

Compilation completed successfully in 1s 608ms (moments ago)

You can also run any Java program from outside IntelliJ using a command line tool as follows:

Running a program in Terminal (mac)

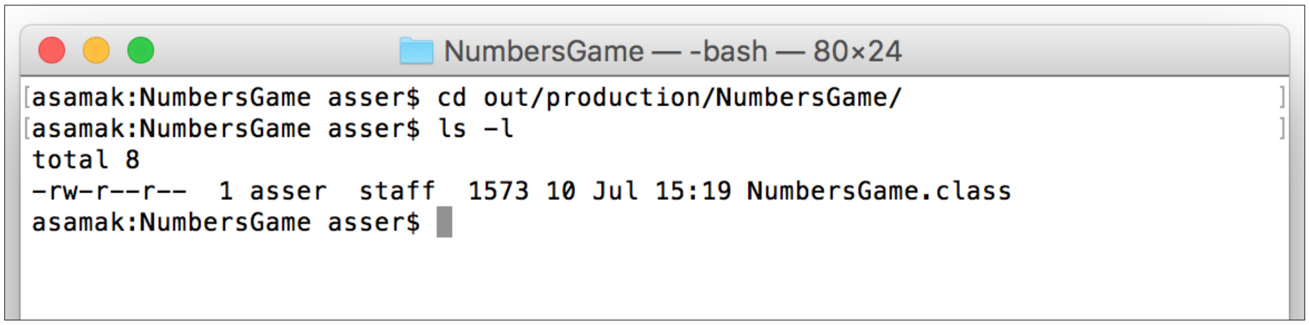
Open the terminal and browse to the location of the IntelliJ project folder:



```
asamak:~ asser$ cd Documents/workspace/NumbersGame/
asamak:NumbersGame asser$ ls -l
total 8
-rw-r--r--  1 asser  staff  423 27 Jun 13:56 NumbersGame.iml
drwxr-xr-x  3 asser  staff  102 10 Jul 15:19 out
drwxr-xr-x  3 asser  staff  102 10 Jul 15:18 src
asamak:NumbersGame asser$
```

There should be a `src` folder that contains all the .java source files (your code). If you've already compiled the project using IntelliJ there should also be an `out` folder that contains a `.class` file that corresponds to every `.java` file in the `src` folder.

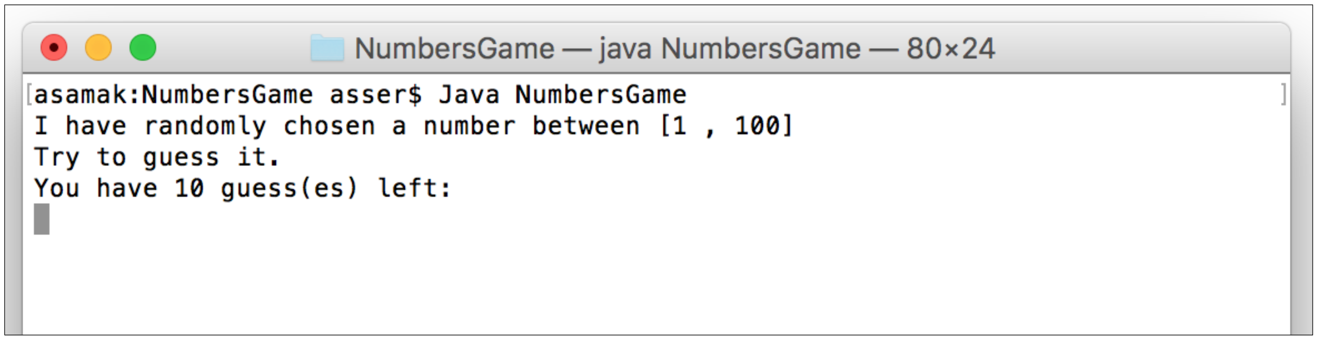
Navigate into that `out` folder then open the `production` folder and the `NumbersGame` folder:



```
asamak:NumbersGame asser$ cd out/production/NumbersGame/
asamak:NumbersGame asser$ ls -l
total 8
-rw-r--r--  1 asser  staff  1573 10 Jul 15:19 NumbersGame.class
asamak:NumbersGame asser$ 
```

These .class files are compiled and ready to run.
To run the class that contains the main method, type in

Java NumbersGame
This will start the program displaying the output in the same window and will wait for the user to enter their input there as well.



```
asamak:NumbersGame asser$ Java NumbersGame
I have randomly chosen a number between [1 , 100]
Try to guess it.
You have 10 guess(es) left:
```

Running a program via the command line (Windows)

It's pretty much the same steps as above, however you might need to make sure that Windows can find the Java compiler and interpreter:

1. Select Start -> Computer -> System Properties -> Advanced system settings -> Environment Variables -> System variables -> PATH.
2. Find out which jdk version you have installed by navigating to C:\Program Files\Java and check which folders are there.
3. Add C:\Program Files\Java\jdk???\bin; to the beginning of the PATH variable.

- Replace the ??? with the folder name from step 2

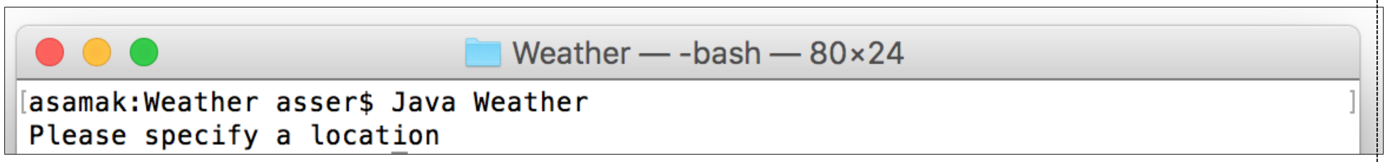
Command line arguments

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[]){
}
```

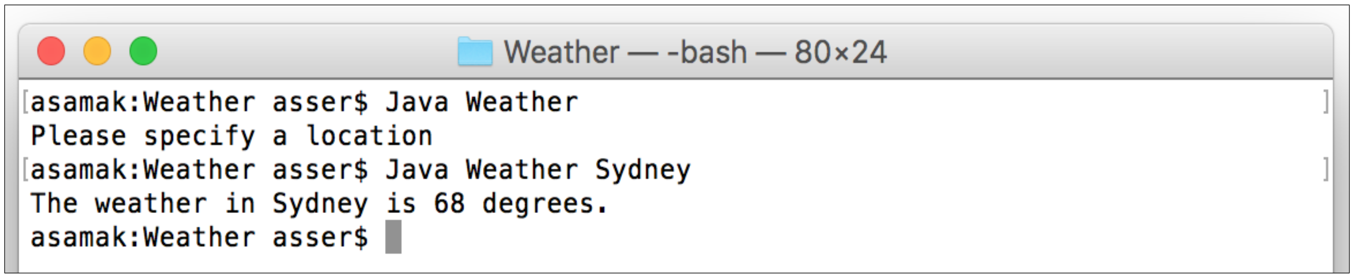
Notice that the method accepts a String array called args[] as an input parameter, but we never explicitly call the main method ourselves! So what is this String array and where does its value ever come from?
If you end up running the program from the command line, anything you type after the program 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:



```
asamak:Weather asser$ Java Weather
Please specify a location
```

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:



```
asamak:Weather asser$ Java Weather
Please specify a location
asamak:Weather asser$ Java Weather Sydney
The weather in Sydney is 68 degrees.
asamak:Weather asser$
```

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

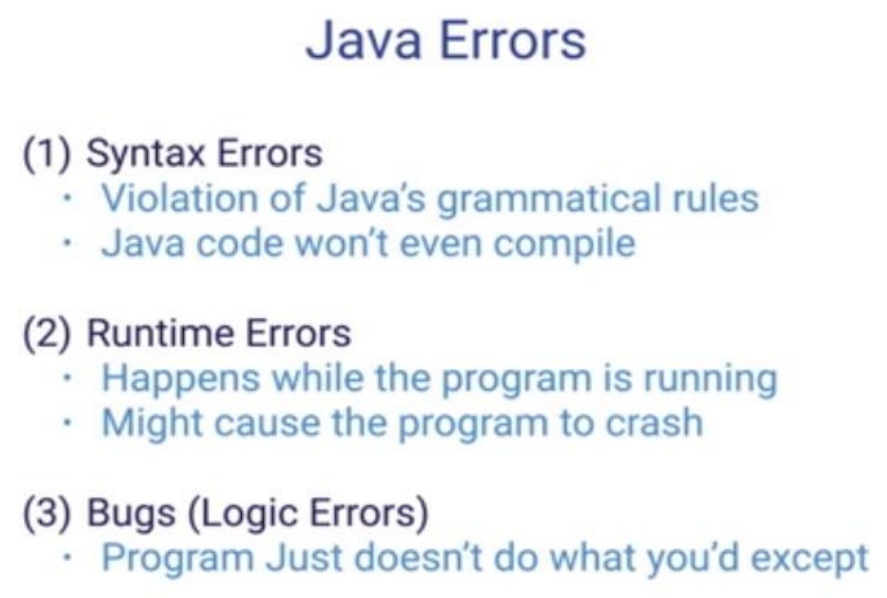
```
public static void main(String [] args){
  if(args.length==0) {
    System.out.println("Please specify a location");
  }
  else {
    String location = args[0];
    int temperature = 60 + (int)(Math.random()*10);
    System.out.println("The weather in "+location+" is "+ temperature);
     }
}
```

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

Exceptions

As a programmer, you will most certainly face these three types of errors: syntax errors, runtime errors, and bugs. We've already covered syntax errors in the previous course, and we've also seen how to find bugs and fix them, but we've only briefly touched on what runtime errors mean and how to handle them. So, now it's time to go deeper into the runtime errors.



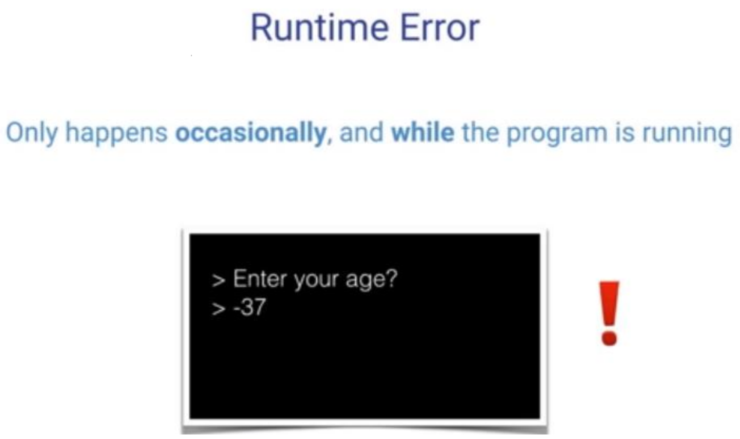
Java Errors

(1) Syntax Errors
- Violation of Java's grammatical rules
- Java code won't even compile

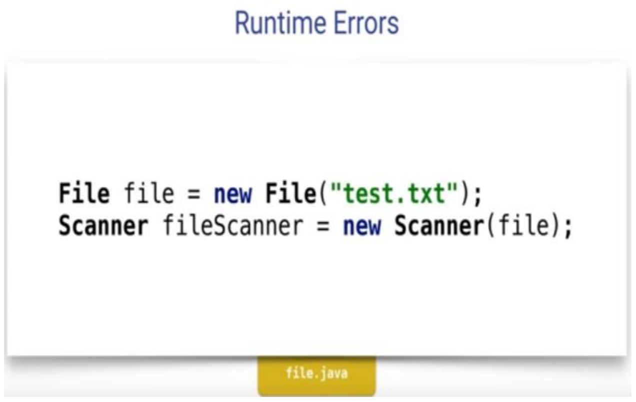
(2) Runtime Errors
- Happens while the program is running
- Might cause the program to crash

(3) Bugs (Logic Errors)
- Program Just doesn't do what you'd except

A runtime error is an error that only happens sometimes while the program is running. It's usually caused by issues like user entering an invalid input or trying to open a file that doesn't exist.

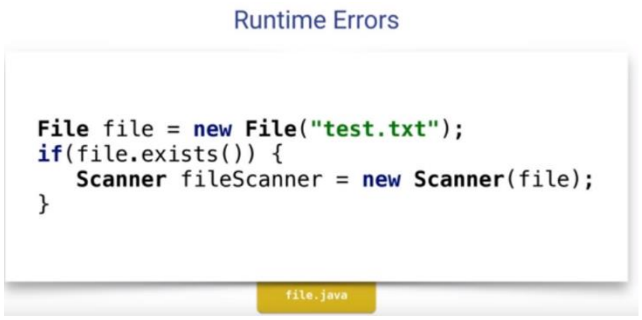

Runtime Error
Only happens **occasionally**, and **while** the program is running

```
> Enter your age?
> -37
```



```java
File file = new File("test.txt");
Scanner fileScanner = new Scanner(file);
```

A good Java program should check that any operation is valid before it tries to do it, like check that the file actually exists before opening it. If you don't, not only will the operation fail, but it might actually cause the entire program to crash.



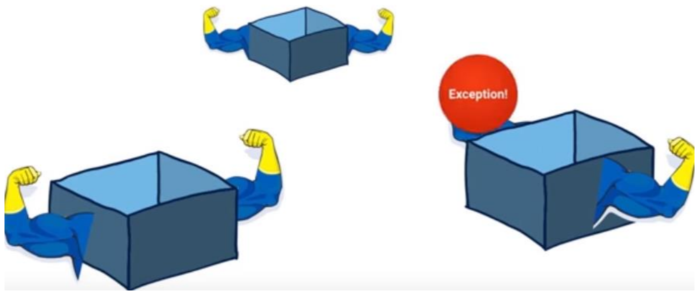
```java
File file = new File("test.txt");
if(file.exists()) {
   Scanner fileScanner = new Scanner(file);
}
```

That's why in Java, most common runtime errors are formalized into something called **exceptions**.

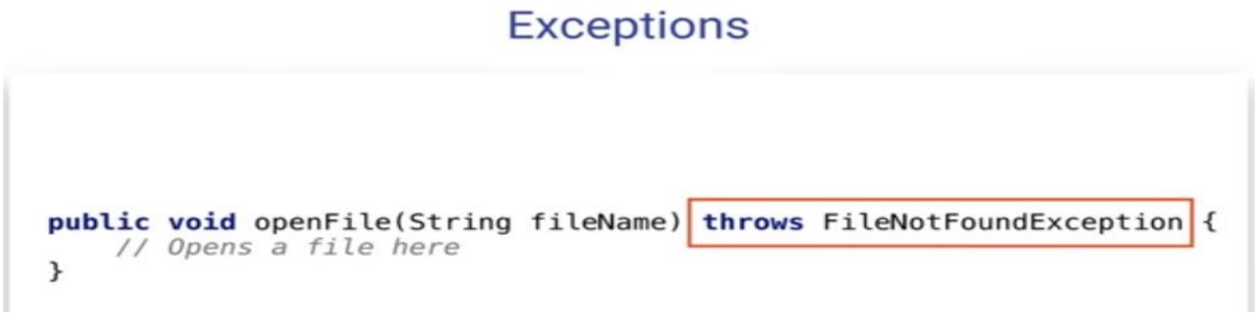
An **exception** is almost like a formal definition of a potential problem. For example, a very popular exception is called FileNotFoundException that appears whenever you try to open a file that doesn't exist.

But how do these exceptions appear and where do they come from?

Exceptions are thrown around between methods. It all starts when a method tries to perform an operation that is invalid. When it realizes that it can't, it creates an exception object of the relevant exception class, and throws it to whomever catches it. Then those who catch it can either re-throw it again or simply handle it gracefully.



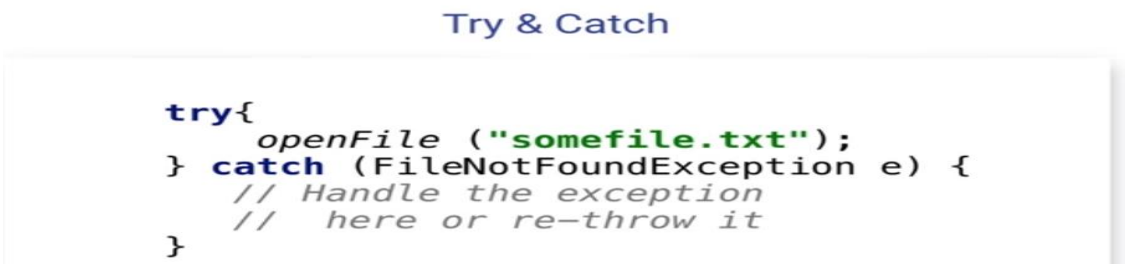
Methods typically communicate with each other using input parameters and returning output results. The way they communicate exceptions with each other is by throwing them. If a method has the potential of running into an invalid situation like opening a file that might not exist, it has to state that it might throw an exception. This is done by adding the throws keyword followed by the exception type when declaring that method.



Exceptions

```
public void openFile(String fileName) throws FileNotFoundException {
    // Opens a file here
}
```

Once you've declared that a method throws an exception, Java forces you to surround that method with a try clause every time you try to call it.

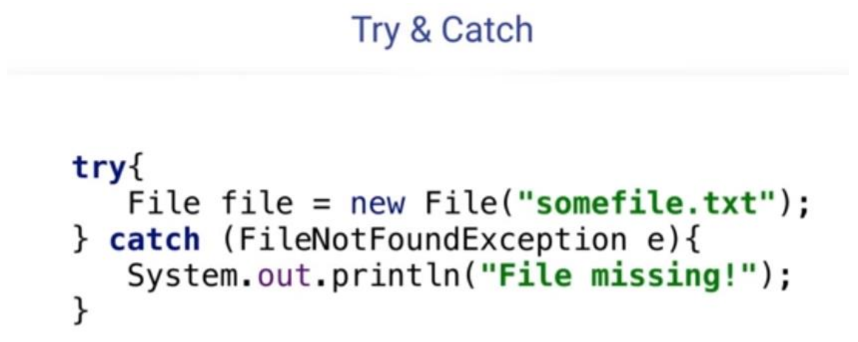


Try & Catch

```
try{
    openFile ("somefile.txt");
} catch (FileNotFoundException e) {
   // Handle the exception
   //  here or re-throw it
}
```

This simply means that you are aware that this method might throw an exception at any time. To handle that exception, you will also need to follow it with a catch block. Whenever that exception is thrown, the code would jump right into that catch block and run whatever's inside it.

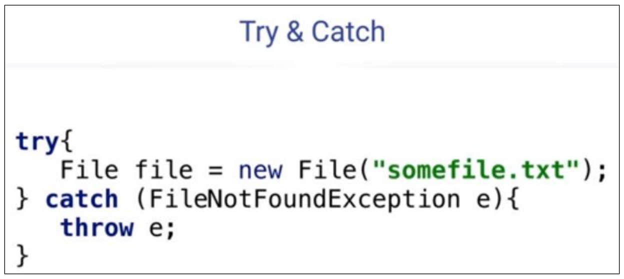
Most exceptions are already built in Java libraries like the file class that were used earlier. So, usually, all you have to do is just surround those methods with a try catch block, and handle the exception anyway you want. The most common way to handle an exception is to print a message to the user, explaining what went wrong and asked them to maybe try again or check their setup.



Try & Catch

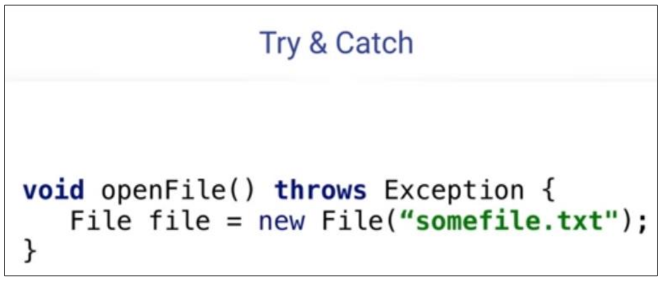
```
try{
   File file = new File("somefile.txt");
} catch (FileNotFoundException e){
   System.out.println("File missing!");
}
```

You may also choose to re-throw that exception or even not surround it with a try catch at all, and instead declare that this method itself throws an exception.



Try & Catch

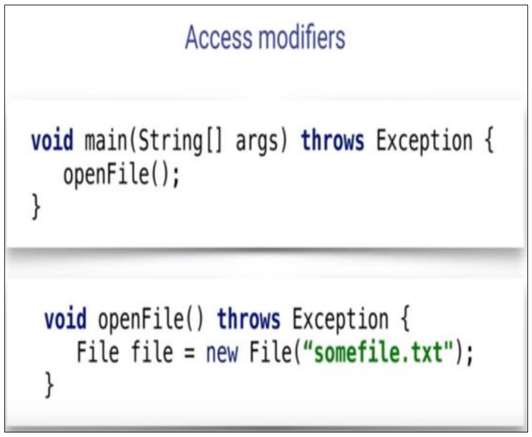
```
try{
   File file = new File("somefile.txt");
} catch (FileNotFoundException e){
   throw e;
}
```



Try & Catch

```
void openFile() throws Exception {
   File file = new File("somefile.txt");
}
```

You can actually continue to do so all the way up to the main method and declare the main method to throw an exception itself. However, that will mean that if such exception does get thrown, it will end up all the way back to the user causing the program to end with an unhandled exception method, not the most user-friendly experience I bet. So, it is always a good idea to handle any possible exception somewhere inside your code.



Access modifiers

```
void main(String[] args) throws Exception {
   openFile();
}
```

```
void openFile() throws Exception {
   File file = new File("somefile.txt");
}
```

Handling exceptions

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

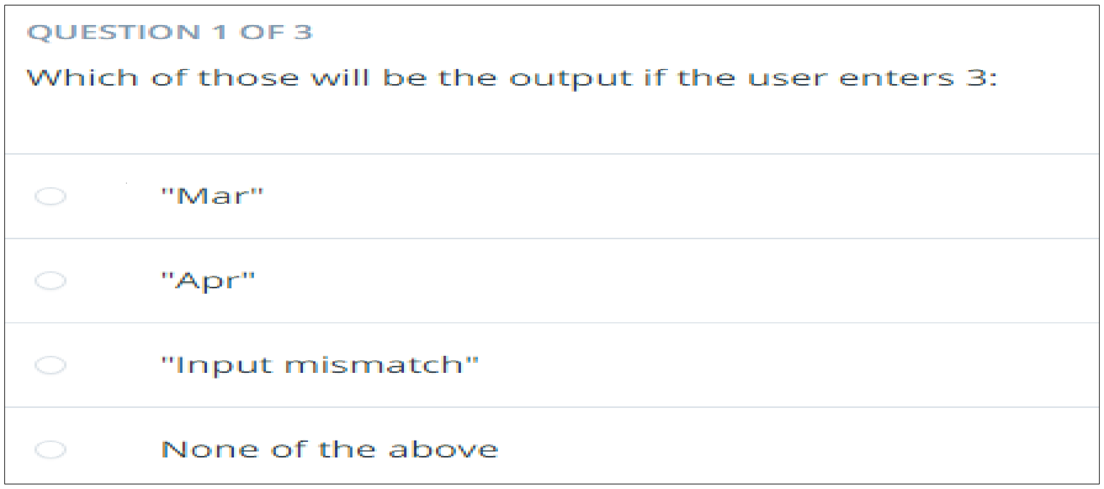
```
try{
  openFile("somefile.txt");
  array[index]++;
} catch(Exception exception) {
  // Handle all the possible exceptions here
}
```

Quiz: Exceptions

This code was meant to ask the user for a month number and print out the month's short name that corresponds to that number!

Read the code carefully and then try to answer the questions below.

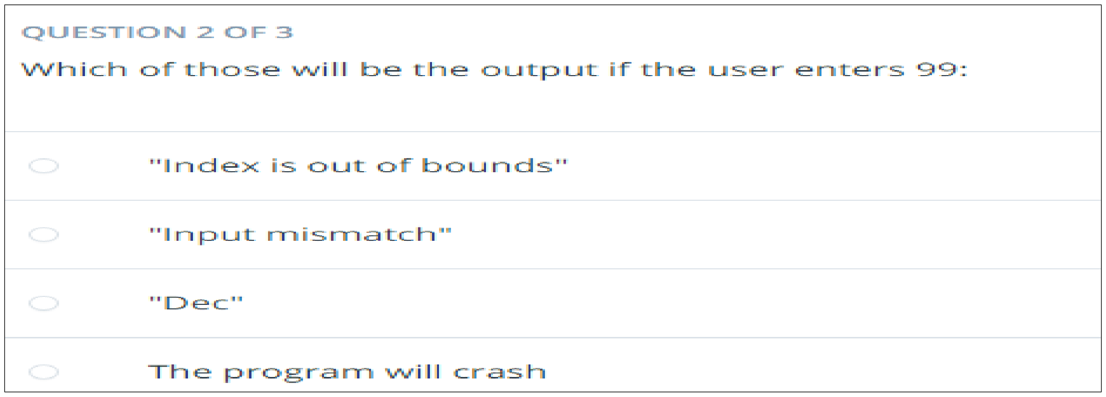
```
public static void main(String[] args) {
   String[] months = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
   Scanner scanner = new Scanner(System.in);
   try {
      int month = scanner.nextInt();
      System.out.print(months[month]);
   } catch (IndexOutOfBoundsException exception) {
      System.out.print("Index is out of bounds");
   } catch (InputMismatchException exception) {
      System.out.print("Input mismatch");          } }
```



QUESTION 1 OF 3

Which of those will be the output if the user enters 3:

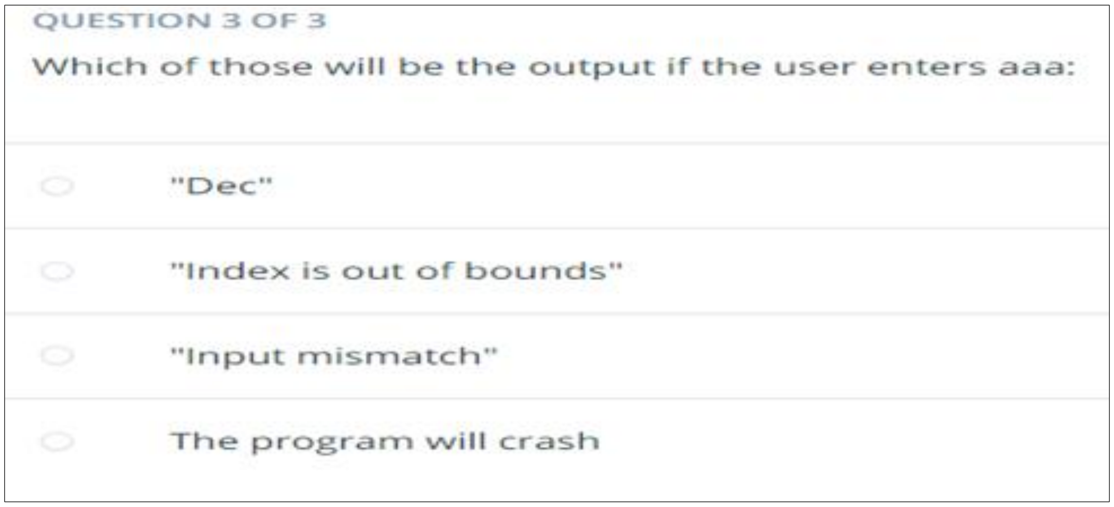
- "Mar"
- "Apr"
- "Input mismatch"
- None of the above



QUESTION 2 OF 3

Which of those will be the output if the user enters 99:

- "Index is out of bounds"
- "Input mismatch"
- "Dec"
- The program will crash



QUESTION 3 OF 3

Which of those will be the output if the user enters aaa:

- "Dec"
- "Index is out of bounds"
- "Input mismatch"
- The program will crash

Part 3: Project 2: Guess The Movie

Ok, it's time to build your own project in Java, this time you'll be completing a game where the player gets to guess the movie name given the number of letters in it (pretty much like hangman but with movies)!

The rules are simple, the computer randomly picks a movie title, and shows you how many letters it's made up of. Your goal is to try to figure out the movie by guessing one letter at a time.

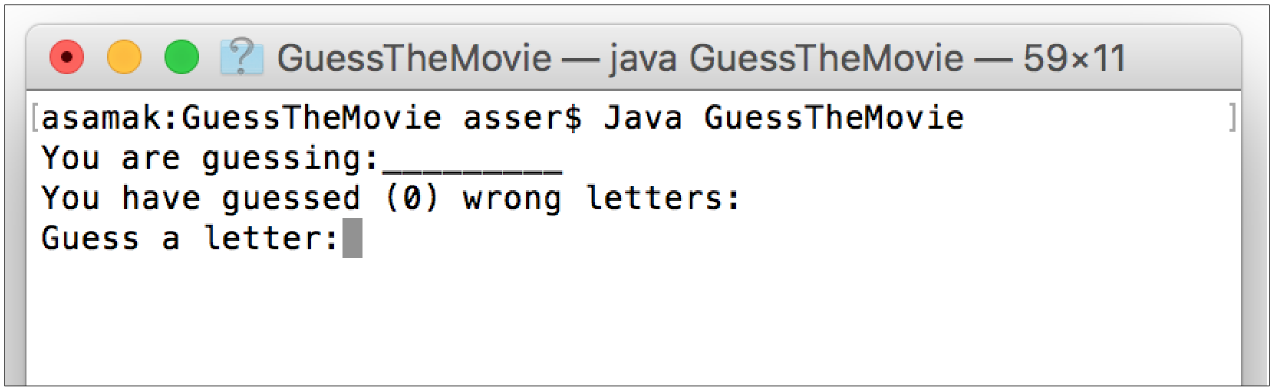
If a letter is indeed in the title the computer will reveal its correct position in the word, if not, you lose a point. If you lose 10 points, game over!

BUT the more correct letters you guess the more obvious the movie becomes and at a certain point you should be able to figure it out.

The program will randomly pick a movie title from a text file that contains a large list of movies.

You can download a sample text file to play with from the resources tab or create your own list of movie titles.

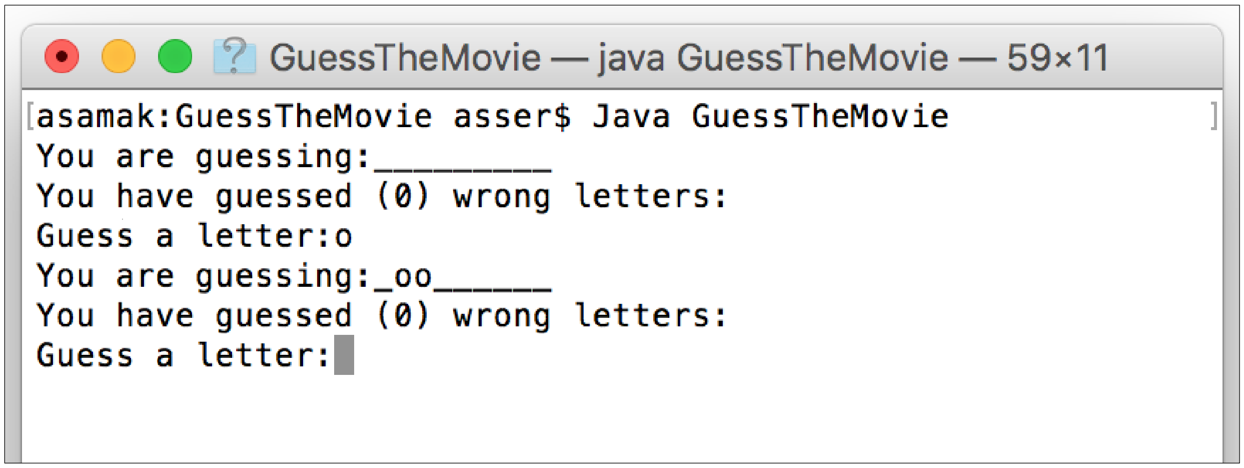
Once the computer picks a random title, it will display underscores "_" in place of the real letters, thereby only giving away the number of letters in the movie title.



```
asamak:GuessTheMovie asser$ Java GuessTheMovie
You are guessing:_________
You have guessed (0) wrong letters:
Guess a letter:
```

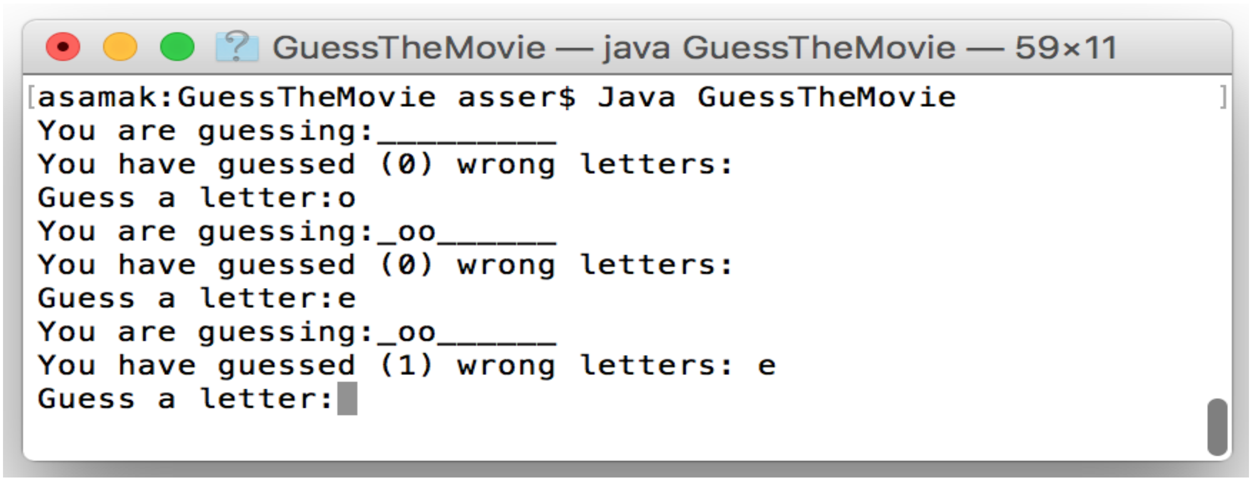
Then it will wait for the player to enter their first letter guess.

If the letter was indeed in the word, the underscores "_" that match that letter will be replaced with the correct letter revealing how many letters have matched their guess and where they are.



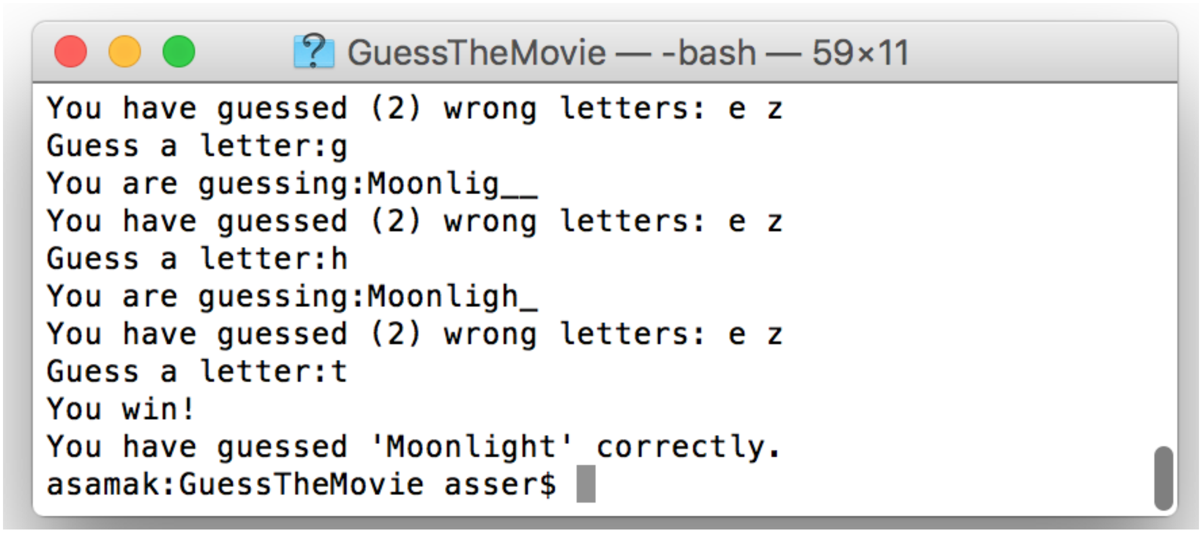
```
asamak:GuessTheMovie asser$ Java GuessTheMovie
You are guessing:_________
You have guessed (0) wrong letters:
Guess a letter:o
You are guessing:_oo______
You have guessed (0) wrong letters:
Guess a letter:
```

If their guess isn't in the movie title, then the output will display the same output as the previous step as well as any letters that have been previously guessed wrong.



```
asamak:GuessTheMovie asser$ Java GuessTheMovie
You are guessing:_________
You have guessed (0) wrong letters:
Guess a letter:o
You are guessing:_oo______
You have guessed (0) wrong letters:
Guess a letter:e
You are guessing:_oo______
You have guessed (1) wrong letters: e
Guess a letter:
```

Eventually, if the player manages to guess all the letters in the movie title correctly before they lost 10 points, they win



```
You have guessed (2) wrong letters: e z
Guess a letter:g
You are guessing:Moonlig__
You have guessed (2) wrong letters: e z
Guess a letter:h
You are guessing:Moonligh_
You have guessed (2) wrong letters: e z
Guess a letter:t
You win!
You have guessed 'Moonlight' correctly.
asamak:GuessTheMovie asser$
```

Everything you need to know to be able to build this game should be covered in the previous lessons, but of course that doesn't mean it has to be easy! It's ok to get stuck and it's absolutely normal for things to not work from the first time.

Just take it step by step, build a small part of the game first, test it and make sure it works and then continue to add more to it.

Aaaand, whenever you reach a roadblock, head to the forum straight away, there are tons of other students and mentors there that will be more than happy to help.

Download the movie list from **here** (https://bit.ly/2Nbprvg), and start coding. Good luck :)

Hints to help building Guess The Movie game

Game play hints

In English, the top 5 frequency of letters is e t a o i. It can help you play this game after you finish it.
It's an important study in Cryptanalysis. More info about this, please read **Letter frequency from Wikipedia**(https://en.wikipedia.org/wiki/Letter_frequency).

Use classes

This program will have more code than all of the exercises we've previously covered, so it's a good idea to divide your code into classes instead of writing everything in 1 class

A simple design would be to have at least one more class called Game that will include methods responsible for handling a single guess or displaying the hidden movie title etc.
Then have another class that contains the main method and controls the logic of reading the user's input and calling the methods in the Game class

Build it step by step

Don't rush into building the entire game at once, start small, for example:

1. Write some code that will simply read the movie list and display the whole list.
2. Then add to your code to randomly pick one movie and display that title only.
3. Then convert its letters to underscores (_) and display that instead, and so on.
4. Once you've got that part done start reading the user's input and search for it in the title.
5. Work on revealing the correct letters and displaying them.
6. Add the logic to keep track of wrong letters so they don't lose points for guessing the same letter twice.

7. After that, you can keep track of how many wrong guesses and end the game if they lose.
8. Finally, detect when they have guessed all the letters and let them know they've won!
   You can also start by hard coding a single movie title in the code instead of randomly picking one from the file, then add the file reading functionality at the end.

Test often

Every time you add new code that does something new, test it.

The best way to do so is to use `System.out.println()` everytime you add new functionality to test the output of that part.
Make sure when testing to try all possible cases that you can think of (what if the user tries to guess a space character? what if they type in a number? etc)

If you test often while building your code you will end up with fewer bugs as you get closer to finishing it.

String methods

Check out all the powerful methods that Java has already written for you **here**(https://docs.oracle.com/javase/7/docs/api/java/lang/String.html).
Knowing the capabilities of your programming language can save you hours and even days of re-writing code that already exists

For example:

To find if a letter exists in a String, instead of creating a loop to compare each character you can use the `indexOf()` method which returns the position of such character in the String.

Part 4: Inheritance

There's More to OOP

As the title suggests, there's more to OOP than just dividing your code into classes. OOP is all about designing your code in a way that's easy to understand but more importantly, easy to extend and add more features to.

To understand how to design a good Java program, you need to know some of the core principles of object-oriented programming. One of them is called **encapsulation**, *which means that each class is just like a capsule that contains everything it needs and nothing more.*

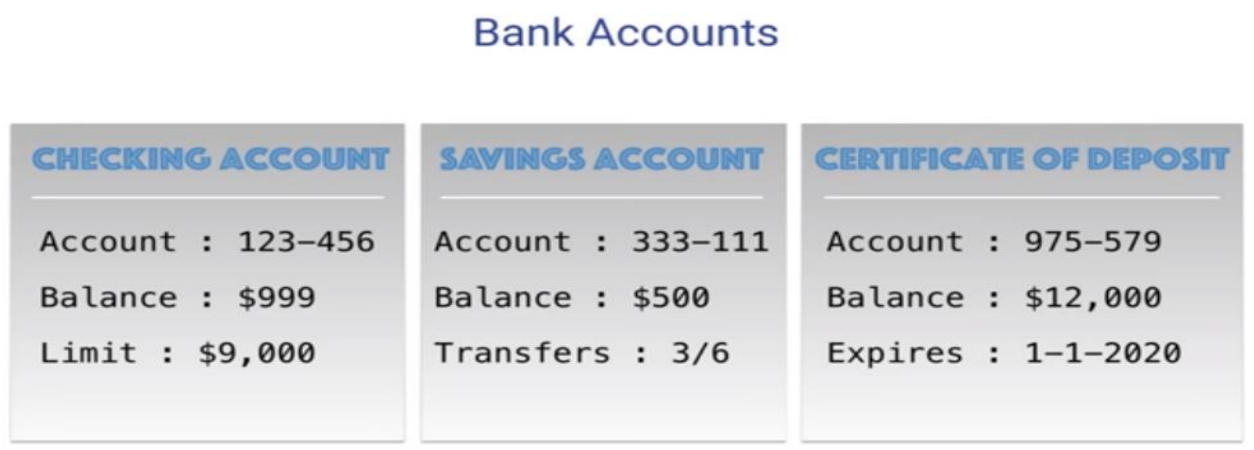
Another important concept is **polymorphism**. *A pretty complicated word, as you can tell, that basically means multiple shapes or forms. Polymorphism defines how Java objects can have multiple identities. That way, you can group different objects as if they're all the same type under certain conditions*.

But perhaps, the most significant of all is **inheritance**. **Inheritance** in Java is just similar to inheritance in real life. *It means passing down traits or characteristics from a parent to their child, like eye and hair color features or facial features*.

Classes can not only use other classes, but they can also inherit from them and extend their capabilities. Inheritance is crucial in designing a good Java program because *it saves you time in writing code* and *keeps things consistent and well-organized*. So, let's see what it's all about and how to use it.

Inheritance

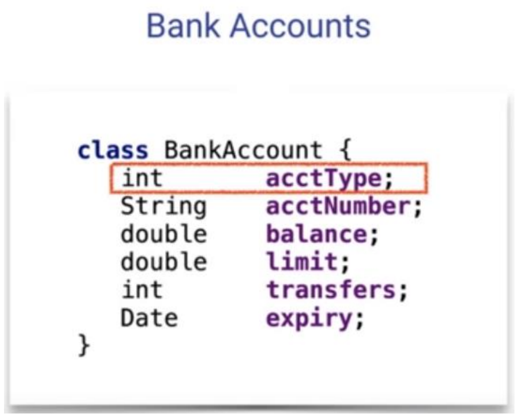
Assume you're building a Java program that manages your bank accounts. You have three different types of accounts and each has its own properties. They all share some similar information like the account number and the balance in them, but they all have different attributes as well. A typical checking account, for example, may have a credit limit. A savings account may limit the number of withdrawals to say six withdrawals per month. A certificate of deposit doesn't even allow any withdrawals until after a certain expiry date.



Bank Accounts

CHECKING ACCOUNT	SAVINGS ACCOUNT	CERTIFICATE OF DEPOSIT
Account : 123-456	Account : 333-111	Account : 975-579
Balance : $999	Balance : $500	Balance : $12,000
Limit : $9,000	Transfers : 3/6	Expires : 1-1-2020

If you want to implement that in Java, you could basically implement everything in one single class, called bank account, and maybe use a field called account type

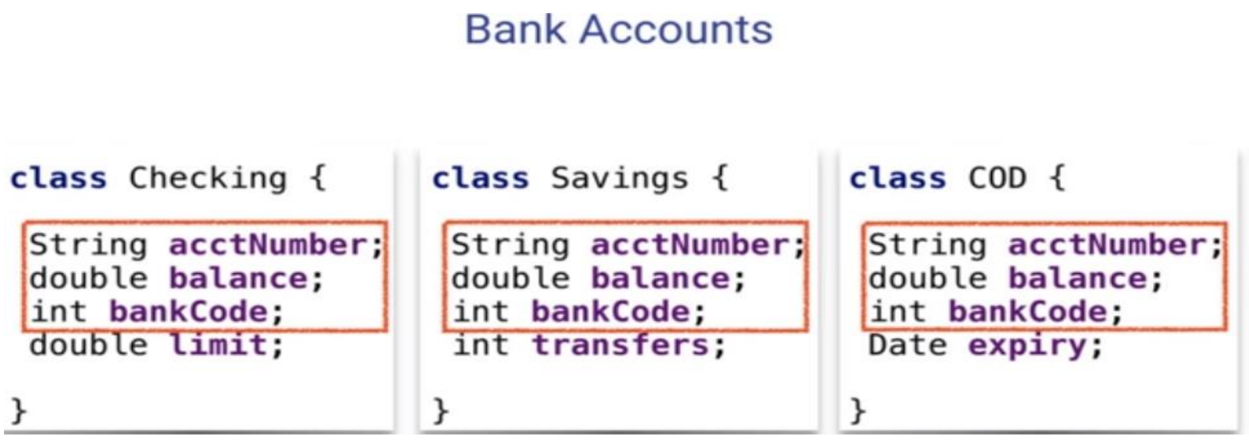


Bank Accounts

```
class BankAccount {
   int        acctType;
   String     acctNumber;
   double     balance;
   double     limit;
   int        transfers;
   Date       expiry;
}
```

that identifies which type each object is but that means you'll have to include everything in that class , and hence in each object that comes from it which means that sometimes it will make sense to have an expiry date for a savings account or a withdrawal limit for a checking account.

Another option is to create a class for each account type. That way, each class would contain, only the fields and methods that make sense for that class. But this also means that all this common area is repeated throughout the three classes. If we decide to also include the bank code, for example, we'll have to change that in all three classes. It might not seem too much here but in a production sized Java project this simple change could be a nightmare.


Bank Accounts

```
class Checking {
  String acctNumber;
  double balance;
  int bankCode;
  double limit;

}
```

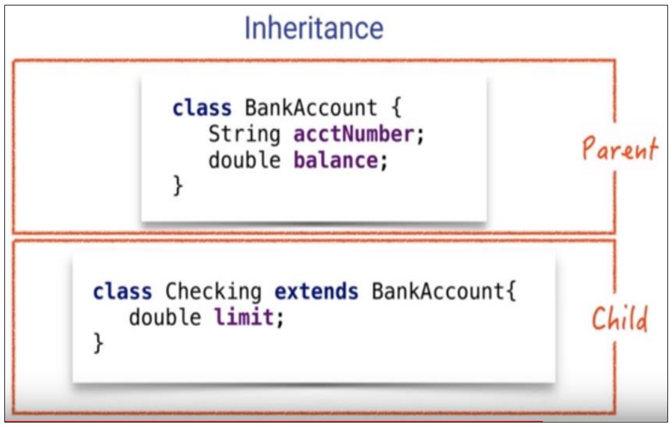
```
class Savings {
  String acctNumber;
  double balance;
  int bankCode;
  int transfers;

}
```

```
class COD {
  String acctNumber;
  double balance;
  int bankCode;
  Date expiry;

}
```

So, can we actually do any better?

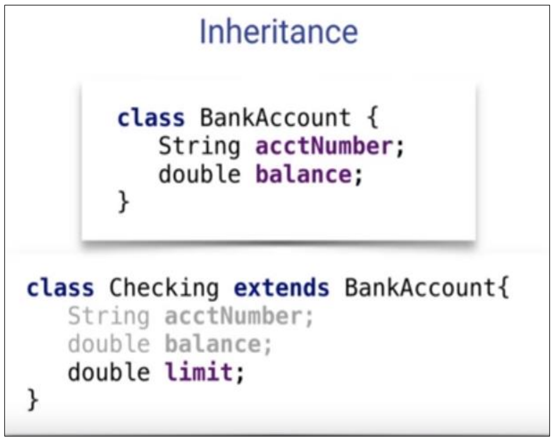
Yes. Java allows classes to extend other classes and build on top of them. This is called inheritance. Let's see how to use that here. If we start by creating a basic bank account class that contains all the common fields and methods we can then create a class for, each different type and point them to extend that basic class.

Each would have its own unique fields and methods while they all share the same basic attributes. To implement this in Java, you'll start by creating the bank account class and then create another class and point it to extend that basic class by adding the phrase , extends and then the class name



```
class BankAccount {
   String acctNumber;
   double balance;
}
```

```
class Checking extends BankAccount{
   double limit;
}
```


Inheritance

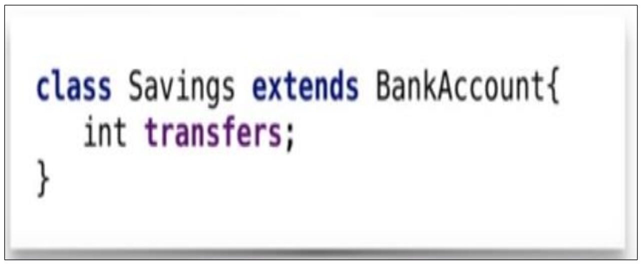
```
class BankAccount {
   String acctNumber;
   double balance;
}
```

```
class Checking extends BankAccount{
   String acctNumber;
   double balance;
   double limit;
}
```

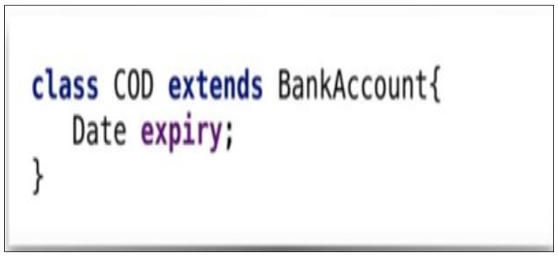
which in our case is bank account. The class that you're extending from is referred to as the parent class. The one that is extending that parent is known as the child class.

Just by extending a class everything that is included in that parent class is now part of that child class as well. Even though it doesn't appear in the code, all the fields and methods that belong to the parent class, in this case the bank account , are as if they were listed in the child class itself.

This way, we can then create another class for the savings account and have its own attribute in it and then another class for the certificate of deposit which also extends the bank account and have its own attribute in there as well.



```java
class Savings extends BankAccount{
   int transfers;
}
```



```java
class COD extends BankAccount{
   Date expiry;
}
```

You can see from that example that using inheritance has allowed us to minimize repeating any code while still having the flexibility and the good design of separate account classes.

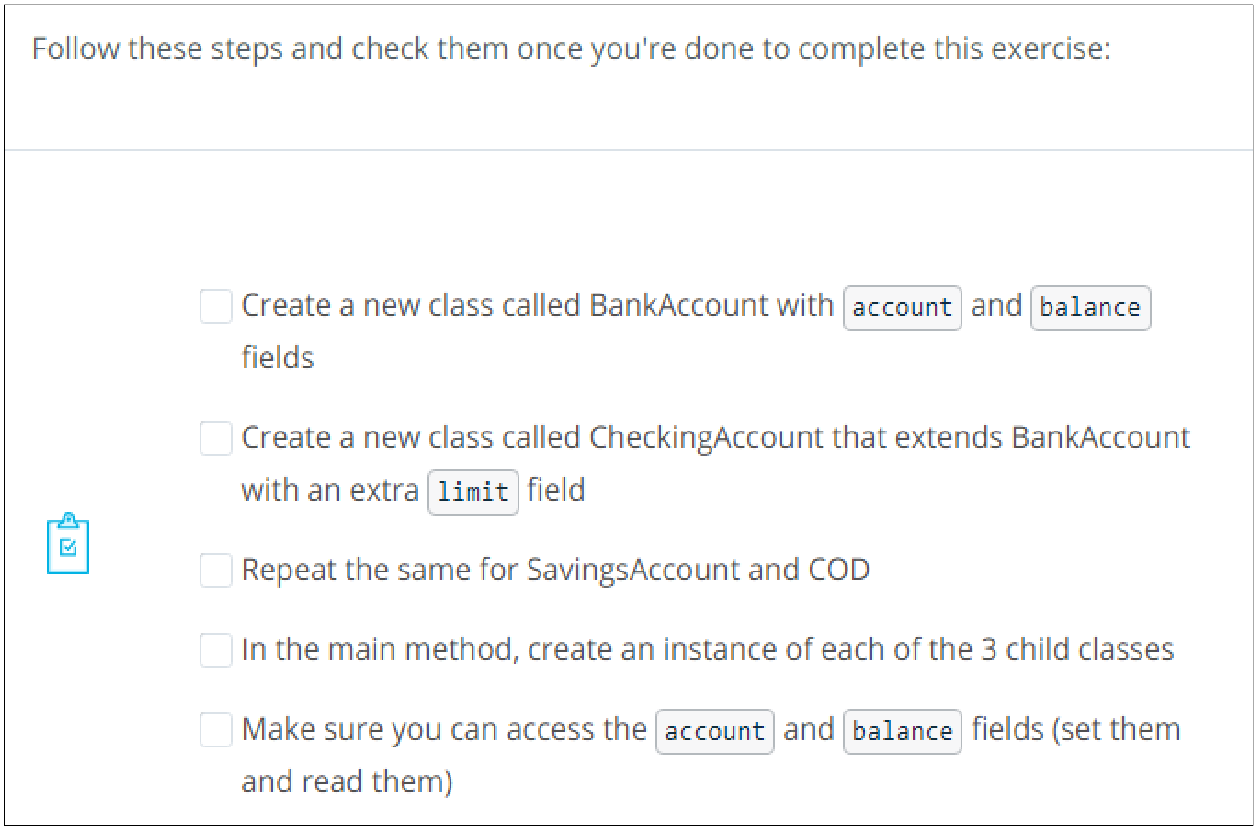
Exercise: Designing the BankManager application

Now it's your turn, open IntelliJ and start a new project called BankManager that will contain the following classes:

1. CheckingAccount
2. SavingsAccount
3. CertificateOfDeposit

Make sure they all extend from the same class called BankAccount that includes all the common fields.

Don't worry about implementing any of the methods for now, just leave them all blank. This exercise is mainly focusing on designing the code rather than writing any logic code.



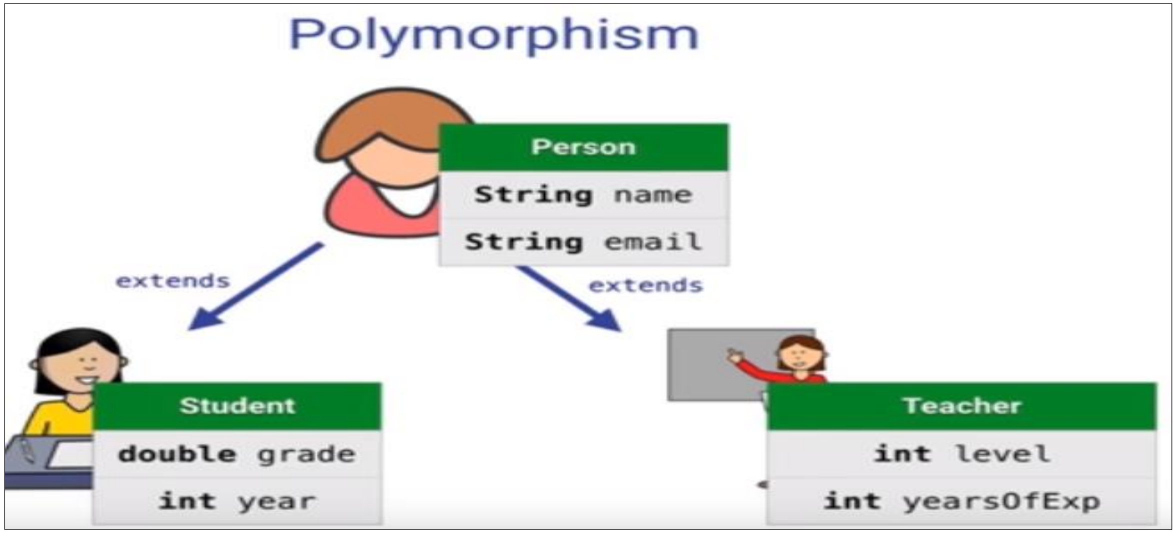
Follow these steps and check them once you're done to complete this exercise:

- ☐ Create a new class called BankAccount with `account` and `balance` fields
- ☐ Create a new class called CheckingAccount that extends BankAccount with an extra `limit` field
- ☐ Repeat the same for SavingsAccount and COD
- ☐ In the main method, create an instance of each of the 3 child classes
- ☐ Make sure you can access the `account` and `balance` fields (set them and read them)

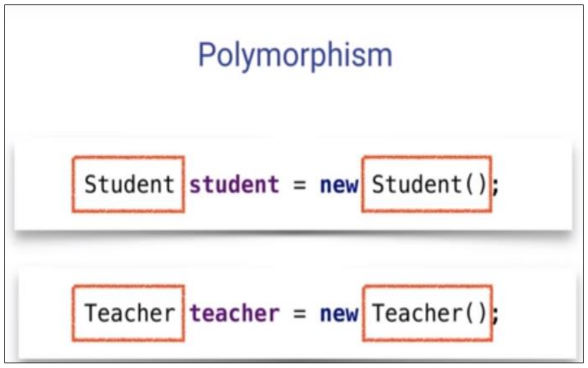
Polymorphism

Polymorphism literally means *something having multiple shapes or forms*. In object-oriented world, inheritance has allowed object to become polymorphic. Because when an object extends another class, it not only becomes its own type but also the type of its parent.

Imagine a class called person, that has a name and an email as fields. Now, if we have two more classes, teacher and student that inherit from that class using the extends keyword then each of those classes would automatically contain those fields along with any extra fields each of them has.



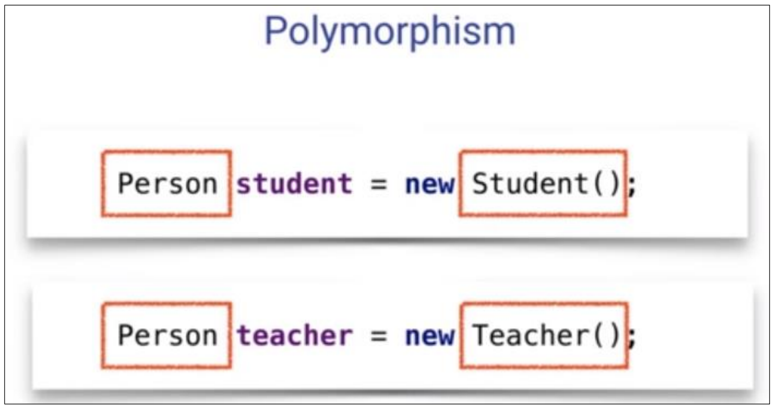
Perfect. So, if you were to create an object of type student, you can start declaring it using the student type and initialize it using the student constructor. And for the teacher you could declare it using the teacher type and initialize it using the teacher constructor.


Polymorphism

```
Student student = new Student();
```

```
Teacher teacher = new Teacher();
```

But, remember that both teacher and student are extending the class person. So, they are by default of type person as well, as the defined type of each. In fact, you can declare both the student and the teacher as of type person and still initialize each of them using their own constructor. This flexibility allows you to treat children objects as if they were the types of their parent and still use all the functionality inside the child class itself.



```
Person student = new Student();
Person teacher = new Teacher();
```

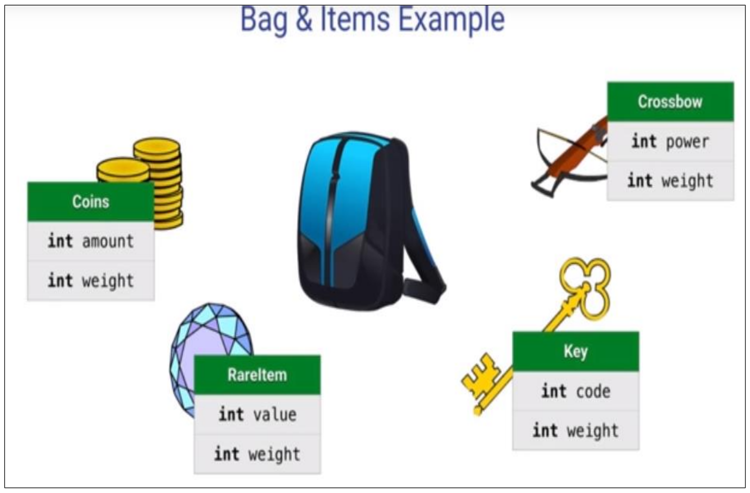
Okay, but is that really that helpful?

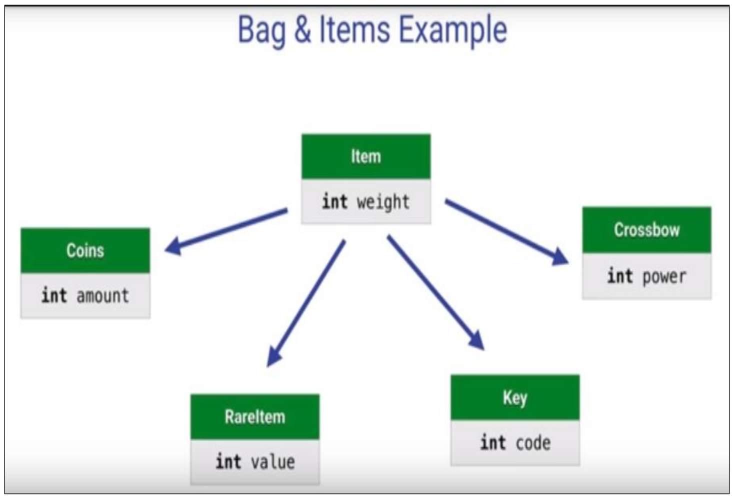
Yes. Throughout the remaining of this lesson, you'll see why. But for now, let's look at this scenario.

Imagine, you're working with a team of Java developers building a game that involves a bag of items that they can carry with them. However, the bag can only carry a total weight of kilograms.

You are the developer that is responsible for implementing the logic, that checks if you can add another item to the bag or not? While the other team members are still coming up with what these items will be.

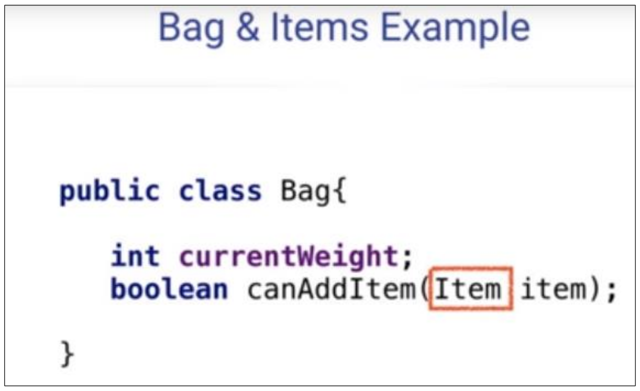
So far, they've created a bunch of different classes for each of the items. A crossbow, a key, a rare item, and some coins. But because they're good Java developers, instead of including the integer weight attributes in each of those classes, they've decided to extend a single class called item, which includes that attribute.





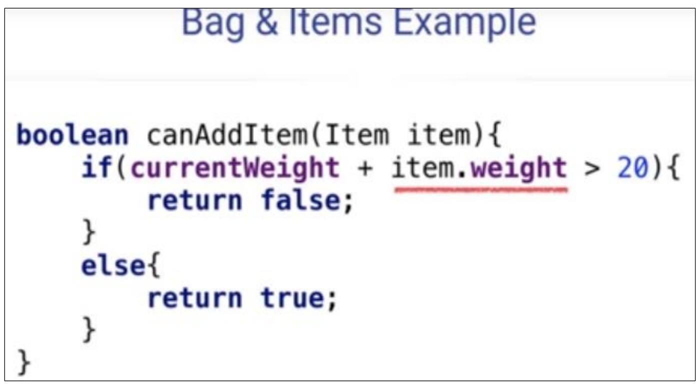
And then, each of the children classes can have its own extra attribute. But, how is that going to help you?

Well, when you get to implement the class bag which includes the method, can add item, you can define the input parameter to be of type item. And this will magically work for any of the child classes that extend from the class item. Which means you only need to implement this method once. And in that implementation, all you have to do is access the item.weight attribute. And because all the children classes that extend from item will include that attribute by default, this method will just magically work for any of the existing items. *These 2 examples of bag.java class.*



Bag & Items Example

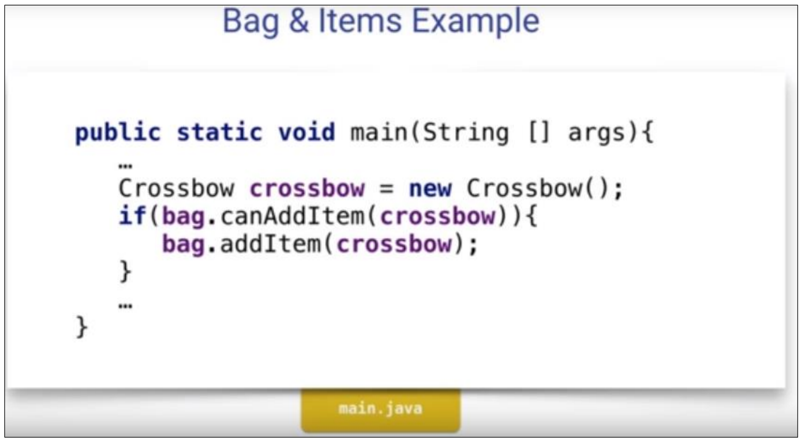
```java
public class Bag{

    int currentWeight;
    boolean canAddItem(Item item);

}
```



Bag & Items Example

```java
boolean canAddItem(Item item){
    if(currentWeight + item.weight > 20){
        return false;
    }
    else{
        return true;
    }
}
```

So, let's say in the main method you decide to declare a variable of type cross-bow, and then you can pass that cross-bow variable into the can add item method of the bag class just directly without even casting it to the item class. And because, cross-bow extends the item class it would definitely have the weight attribute inside it.

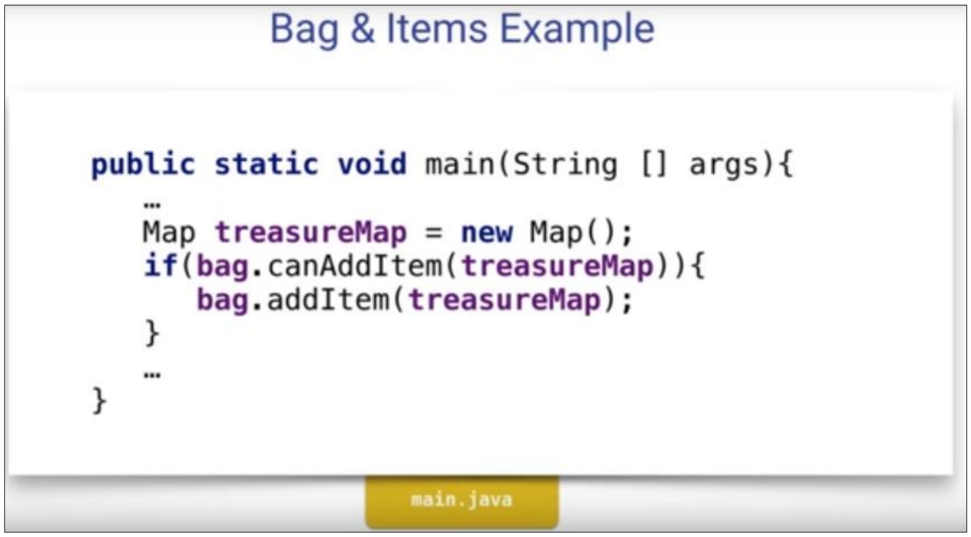


Bag & Items Example

```
public static void main(String [] args){
   …
   Crossbow crossbow = new Crossbow();
   if(bag.canAddItem(crossbow)){
      bag.addItem(crossbow);
   }
   …
}
```

main.java

Not only that, but imagine sometime in the future they come up with this new item class called Map, which also extends from the item class. You can in fact pass in that variable directly as well without changing anything in the can add item method. Amazing, isn't it? Well, that's just part of what polymorphism allows you to do in Java.



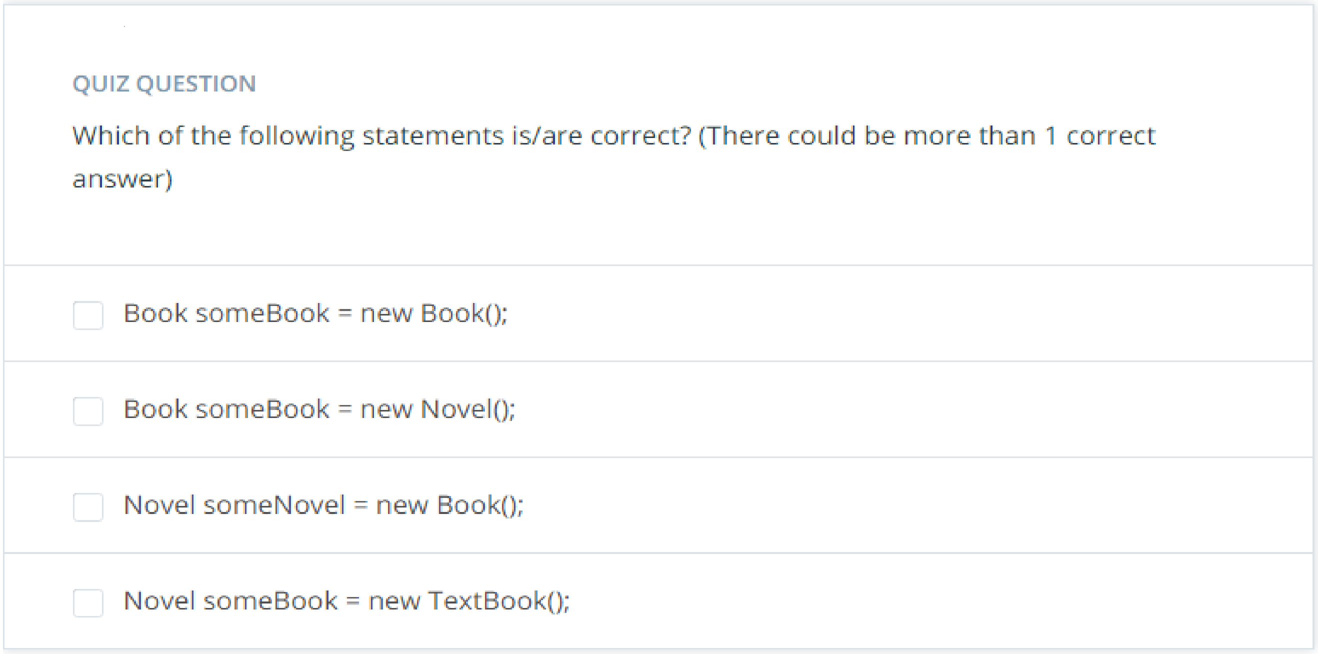
Bag & Items Example

```
public static void main(String [] args){
   …
   Map treasureMap = new Map();
   if(bag.canAddItem(treasureMap)){
      bag.addItem(treasureMap);
   }
   …
}
```

main.java

Quiz: Polymorphism

You are given a class `Book` that has the fields `title` and `numberOfPages`, as well as 2 more classes `Novel` and `TextBook` that extend class `Book`:



QUIZ QUESTION

Which of the following statements is/are correct? (There could be more than 1 correct answer)

☐ Book someBook = new Book();

☐ Book someBook = new Novel();

☐ Novel someNovel = new Book();

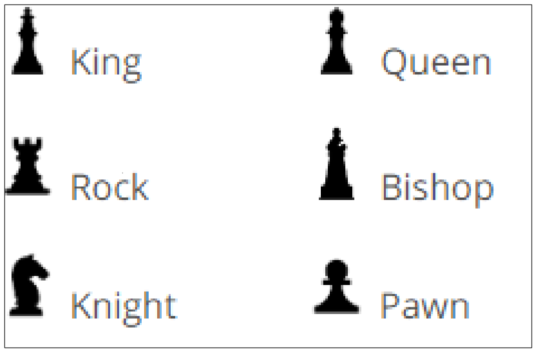
☐ Novel someBook = new TextBook();

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.

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



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.

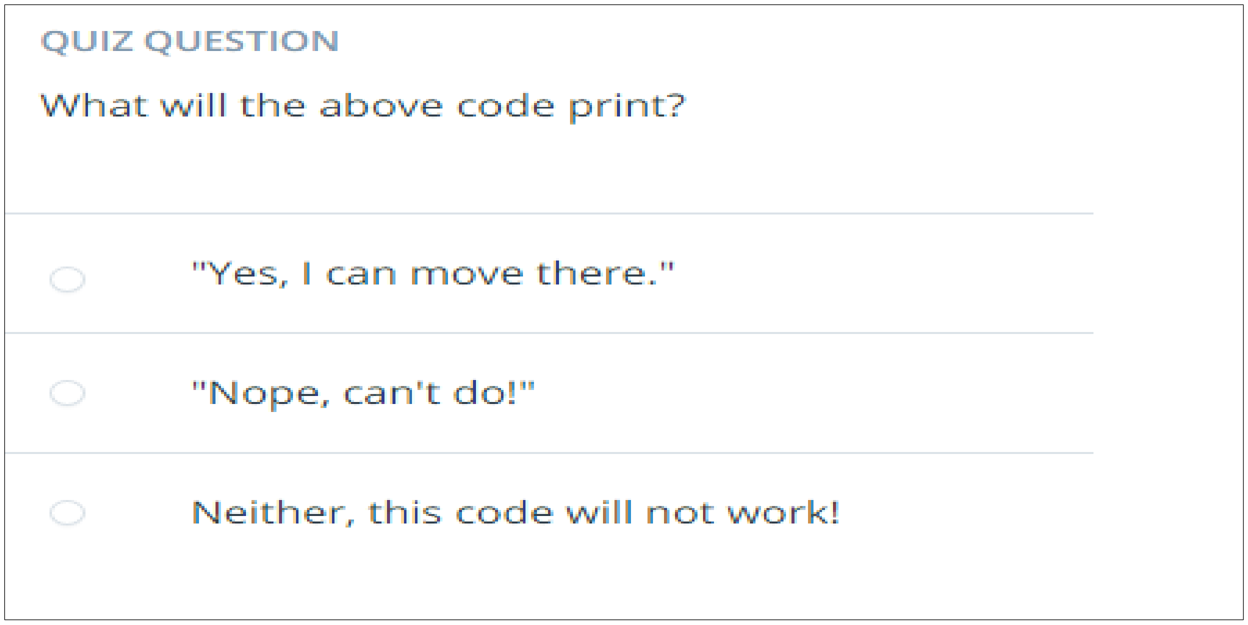
So, a good idea would be to include a method called isValidMove inside the Piececlass that takes a potential new position and decides if that position is within the bounds of the chess board.

```
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


QUIZ QUESTION

What will the above code print?

- "Yes, I can move there."
- "Nope, can't do!"
- Neither, this code will not work!

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?

Overriding methods

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.

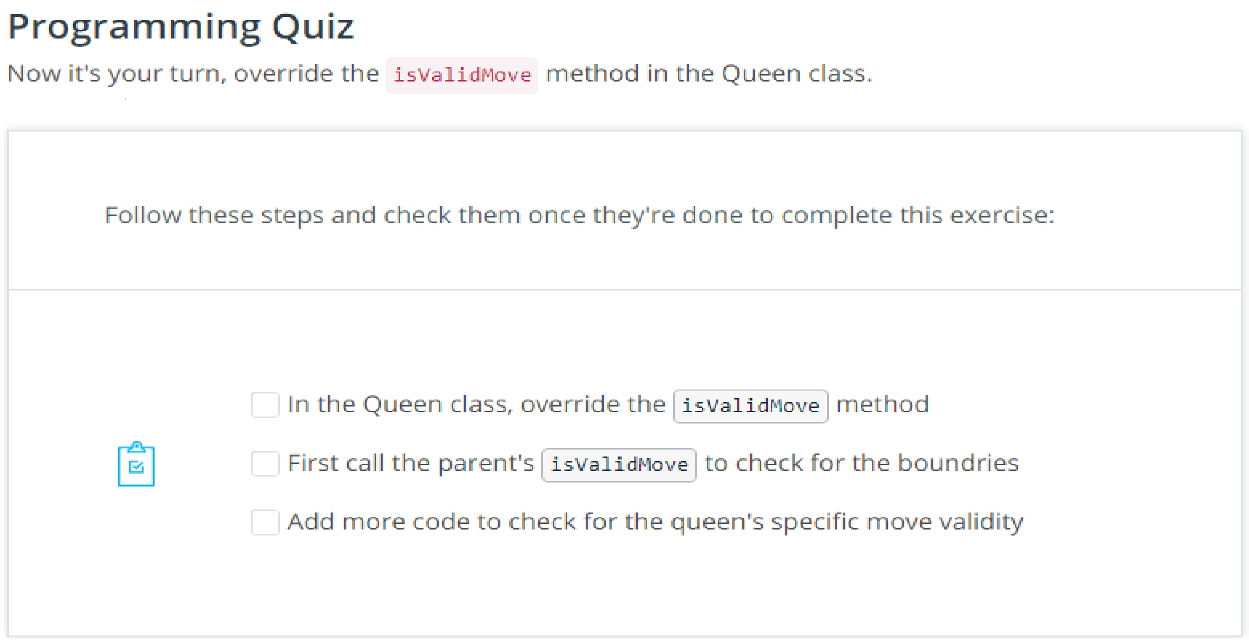
Remember that this.position.column and this.position.row are the local fields of the Rock class holding the current position of that piece.
We can also do the same for all the other piece types, like for example the Bishop class would have slightly different implementation:

```
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.

Programming Quiz: Override



Programming Quiz

Now it's your turn, override the `isValidMove` method in the Queen class.

Follow these steps and check them once they're done to complete this exercise:

- [ ] In the Queen class, override the `isValidMove` method
- [ ] First call the parent's `isValidMove` to check for the boundries
- [ ] Add more code to check for the queen's specific move validity

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:

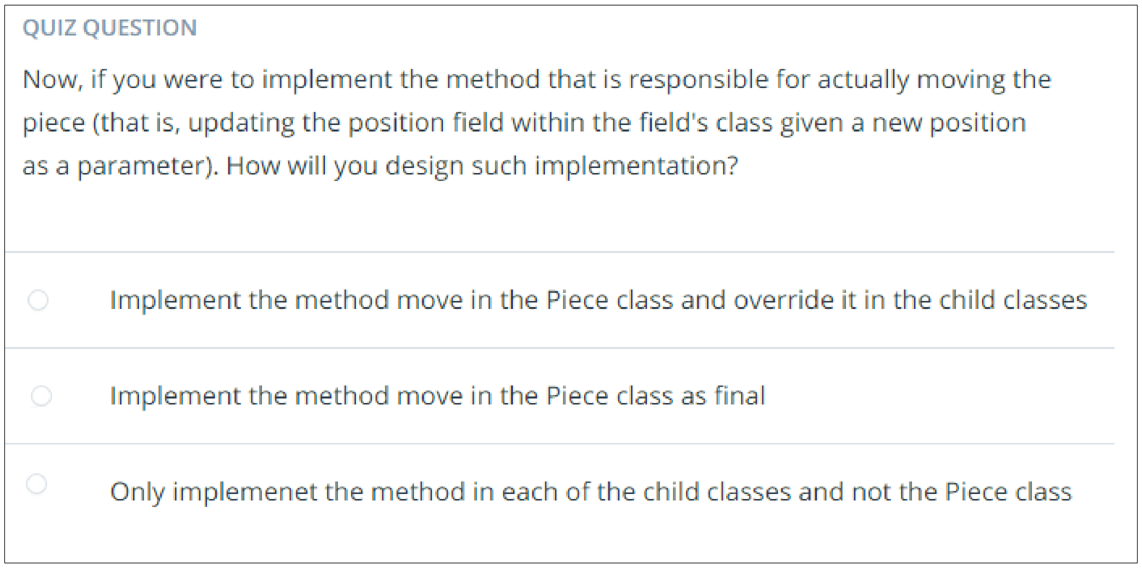
```
class Rock extends Piece{
  boolean isValidMove(Position newPosition){
    // First call the parent's method to check for the board bounds
    if(!super.isValidMove(position)){
      return false;
    }
    // If we passed the first test then check for the specific rock movement
    if(newPosition.column == 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.

QUIZ: Move Method



QUIZ QUESTION

Now, if you were to implement the method that is responsible for actually moving the piece (that is, updating the position field within the field's class given a new position as a parameter). How will you design such implementation?

- ○ Implement the method move in the Piece class and override it in the child classes
- ○ Implement the method move in the Piece class as final
- ○ Only implemenet the method in each of the child classes and not the Piece class

Multiple Inheritance

You have seen how extending a class can be so powerful. However, there is one major limitation in Java. class can only extend one single class. In other words, a class can only have one parent. That's because multiple inheritance can cause ambiguity if the parents had similar methods. Java's solution for this is **interfaces**. So, let's have a look at what an interface is and how to use one.

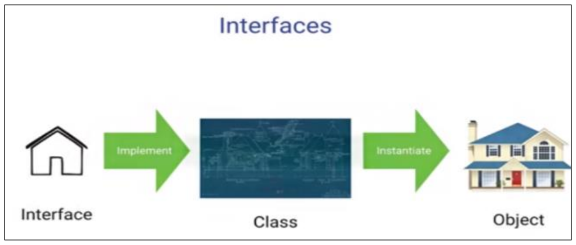
Extending a class is extremely helpful in many occasions:

1. Extending the capability of a class without making any changes to it.
2. Sharing some common code between variations of that class.
3. Leveraging polymorphism to treat different classes as if they were the same. However, there is 1 major limitation in Java: A class cannot extend more than 1 class (i.e. multiple inheritance is not allowed in Java).

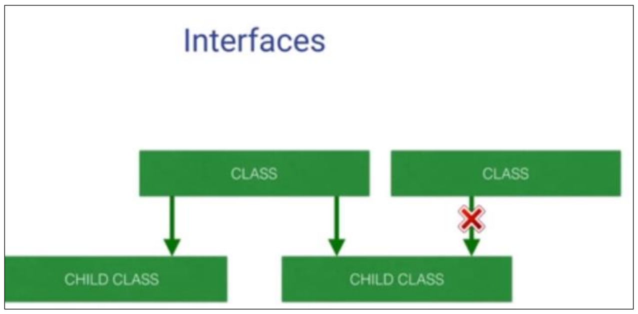
Why? Because multiple inheritance could cause ambiguity if the parents had similar methods. If you'd like to know more about such cases check out the popular example known as **The Diamond Problem**(https://bit.ly/2y0OQSE). Java's solution to the multiple inheritance problem is **Interfaces**.

Interfaces

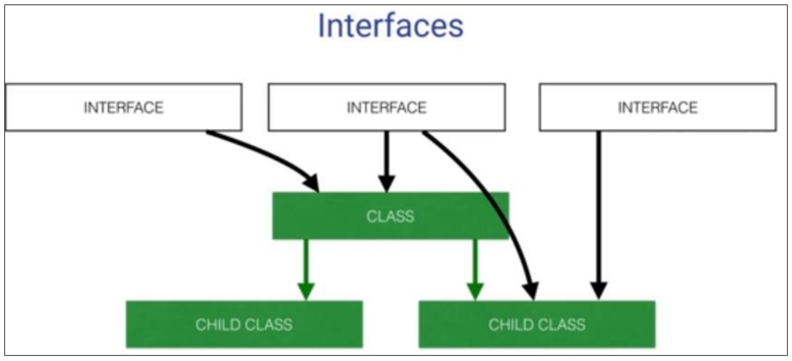
An interface is like a facade or an outline of some imaginary class. Its sole purpose is to be inherited by some other class. It only defines what needs to be done, but not how to do it. In other words, the interface would list the methods that need to be included in the class but no implementation code whatsoever. The implementations of these methods are the responsibility of the classes implementing that interface. Once you implement that interface in a class, you can then start instantiating objects as usual.



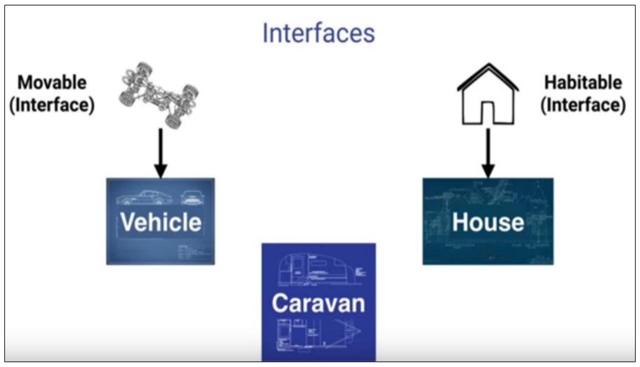
The reason Java introduced interfaces is because of the multiple inheritance problem. A single class can be extended by multiple classes. But a child class is not allowed to extend more than one parent class.

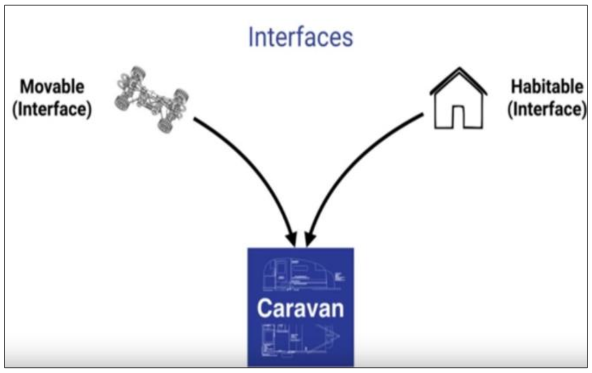


Interfaces, however, don't have that restriction, which means that a single class can implement multiple interfaces allowing for a more flexible design but without the ambiguity problem of the multiple inheritance.

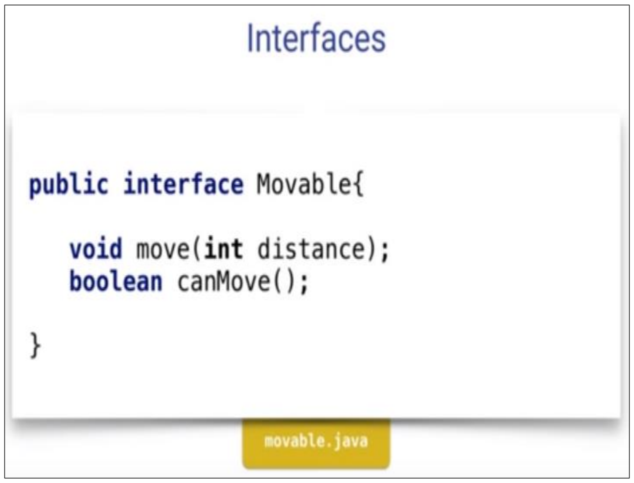


Let's have a look at an example. Imagine you're responsible of implementing a class called Caravan. We know that a caravan is half vehicle, half house. But if you had a class for a vehicle and a class for a house, we know that you're not allowed to extend both classes at the same time. A good solution would be to introduce interfaces. A good interface would be the movable interface, for example, which would define the methods that any class that moves should include, like the vehicle class, for example. Another good interface would be called the habitable interface, which would also define all the methods that are to be included for any habitable class.



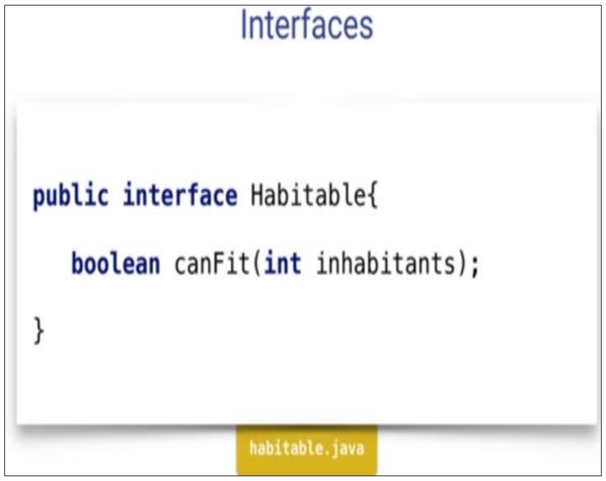


Once we define those two interfaces, we could then implement both of them in the Caravan class at the same time. Let's have a look at the Java code for that. Creating an interface in Java is very similar to creating a class. Simply swap the keyword class with interface. Inside the interface, as you can see, we have listed the method signatures but without any implementation code because that's the responsibility of the class that will be implementing that interface.
The same for the habitable interface, which in our case here only includes the canFit method.


Interfaces

```
public interface Movable{

   void move(int distance);
   boolean canMove();

}
```

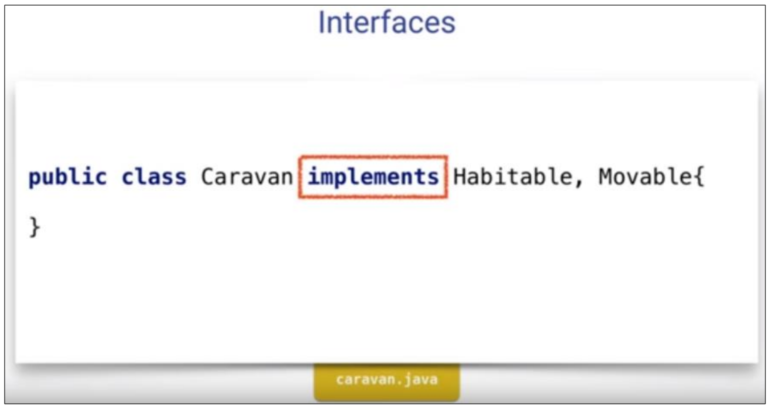
movable.java


Interfaces

```
public interface Habitable{

   boolean canFit(int inhabitants);

}
```

habitable.java

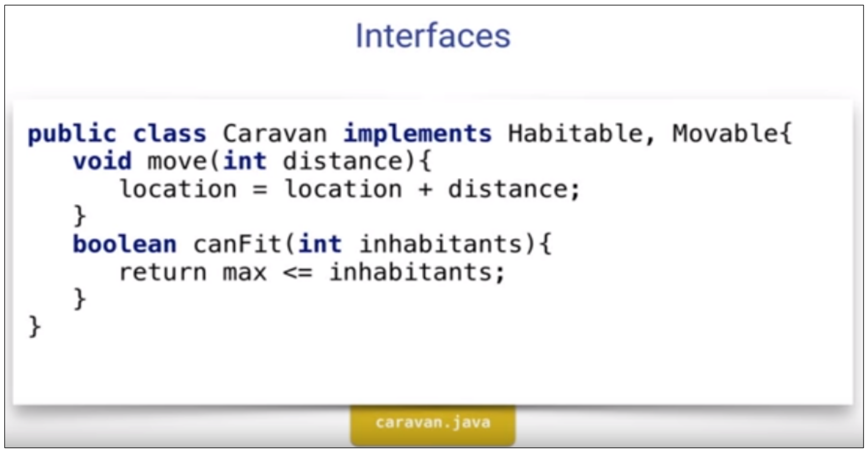
Once we create both of those interfaces, we can then start implementing our Caravan class, which implements both the habitable and movable interfaces. Notice that I've used the keyword, implements, here compared to the keyword, extends. When inheriting from classes, implements is the keyword to use when inheriting from interfaces.


Interfaces

```java
public class Caravan implements Habitable, Movable{
}
```

caravan.java

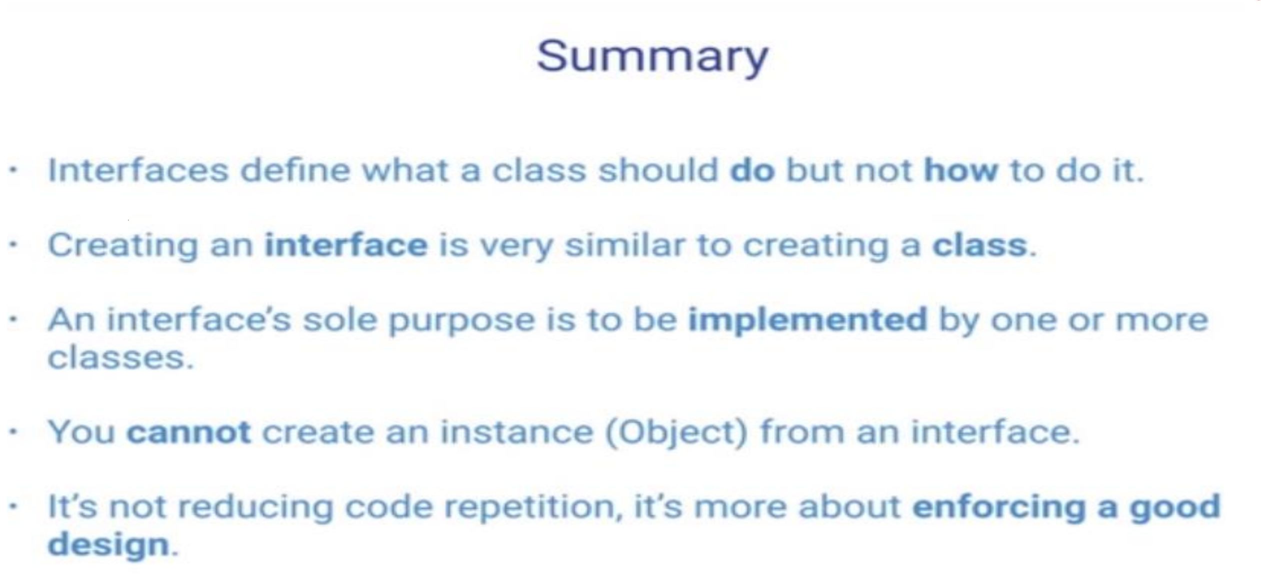
Once we implement those interfaces, we have to implement the code for every single method included in both those interfaces like the move method which was included in the moveable interface and the canFit method which was included in the habitable interface. If there's any method in any of those interfaces that was not implemented in the Caravan class, that will show a compilation error. Notice that inheriting interfaces hasn't really saved us from rewriting any code like we've done before with extended classes.


Interfaces

```java
public class Caravan implements Habitable, Movable{
    void move(int distance){
        location = location + distance;
    }
    boolean canFit(int inhabitants){
        return max <= inhabitants;
    }
}
```

caravan.java

That's because interfaces aren't there to help us reduce code as much as enforce a good design. Creating interfaces forces any class that will implement it to have to implement a certain number of methods. This means that later on, if you have a look at any class that implements a certain interface without looking at the code of that class, you can guarantee that it will include all these methods that are in that interface.



Summary

- Interfaces define what a class should **do** but not **how** to do it.
- Creating an **interface** is very similar to creating a **class**.
- An interface's sole purpose is to be **implemented** by one or more classes.
- You **cannot** create an instance (Object) from an interface.
- It's not reducing code repetition, it's more about **enforcing a good design**.

Comparable Interface

A very popular interface in Java is the **Comparable Interface** (https://bit.ly/2NW6GRO).
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 ...

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**:

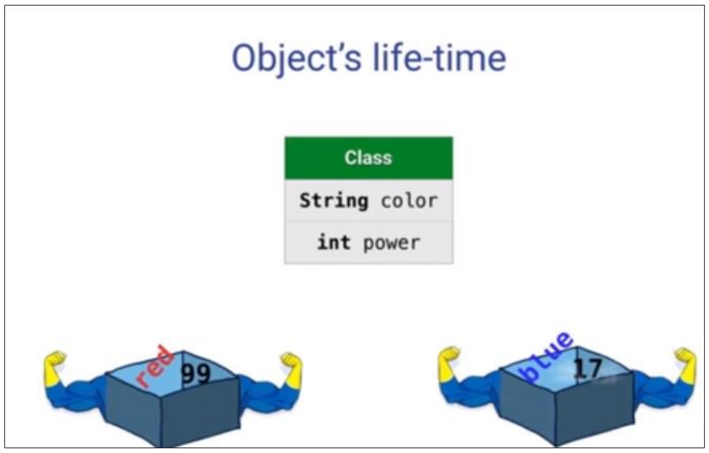
```
public class MathLib{
  public final double PI = 3.14;
}
```

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).

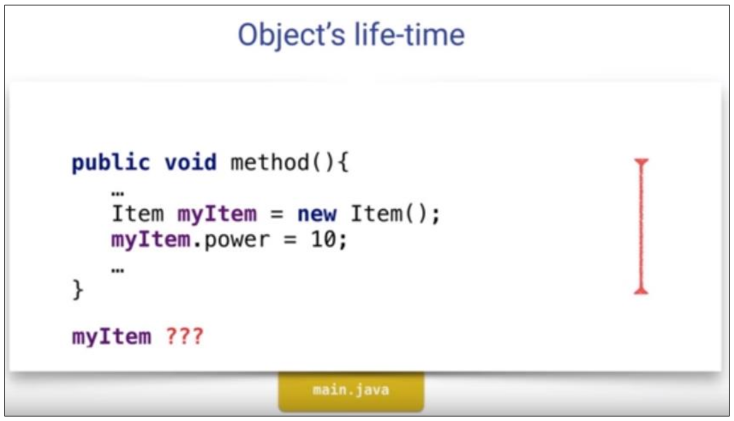
```
public static void main(String [] args){
  MathLib mathlib = new MathLib();
  mathlib.PI = 0; // This is not allowed and will show a compiler error!
}
```

Static

Objects that are created from a class don't really last forever. Typically, you'd create an object from a class, fill its fields with some values, and maybe create another object and fill its fields with some different values, but then eventually, both those objects will get destroyed, including every single value stored in those fields.

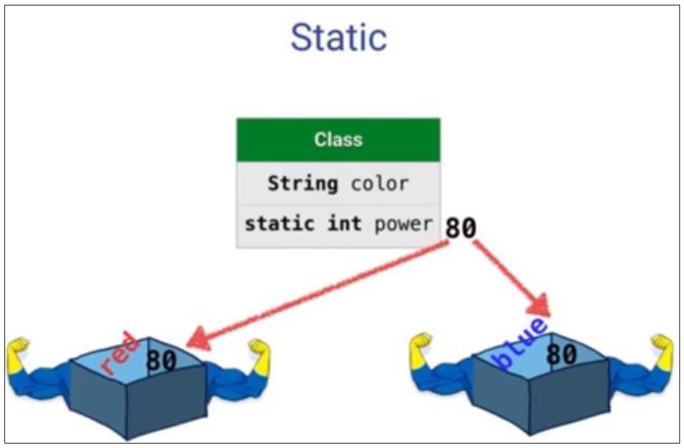


Typically, that would happen whenever the scope of that object ends. For example, here, inside this method, I've created the variable myItem, which is an object of the type class Item. Once this method ends, this myItem variable doesn't exist anymore, including all the values of all the field inside it.


Object's life-time

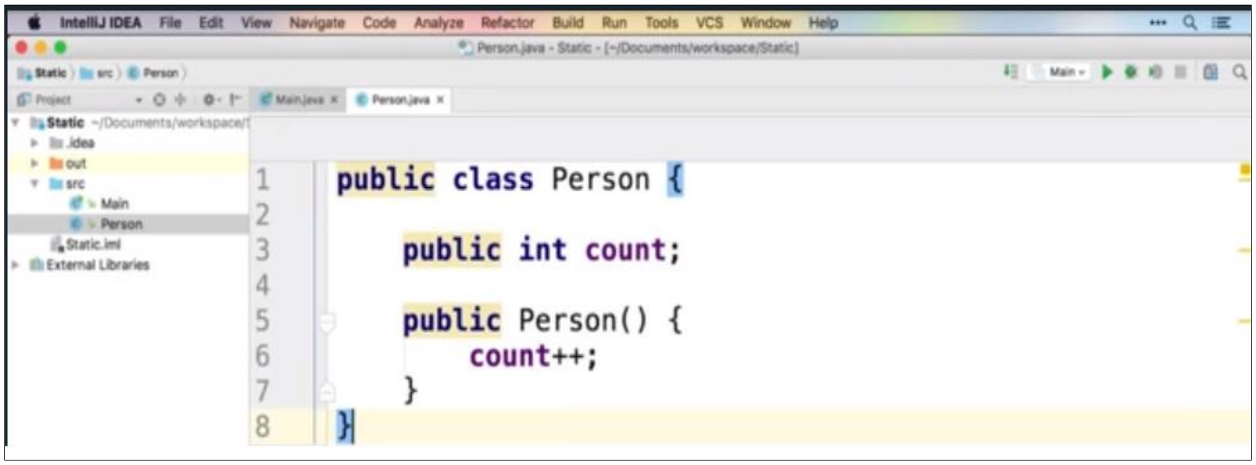
```
public void method(){
   …
   Item myItem = new Item();
   myItem.power = 10;
   …
}

myItem ???
```

This kind of makes sense because you don't really need the values of a field inside an object that you can't even access anymore. However, in some rare occasions, you might want to store the value of a certain field even if there are no objects for that class. In that case, you need to add the keyword static when declaring this field. Declaring a field as static means that these values are no longer stored within the object itself but within the class instead.



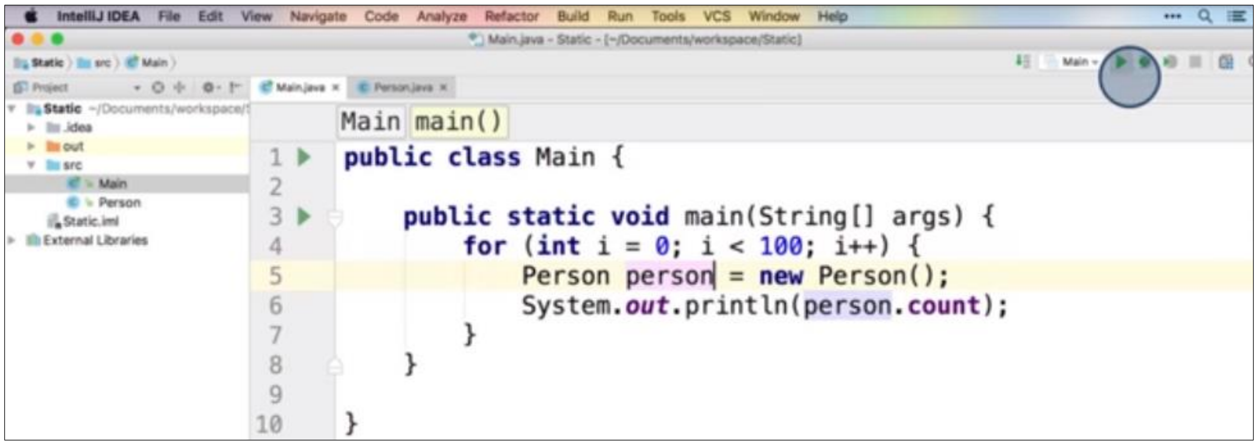
Meaning that all objects of the class will share that same exact value. And then even if every single object of the class has been destroyed, the value is still stored within the class. If you decide to create a new object of that same class, then it will end up using the same value that was stored in the class. Notice however, that static here doesn't mean that the value doesn't change. In fact, if that value does change, it will update it in every single object of that class again. Now because static fields belong to classes instead of object, Java allows you to access a static field directly from the class instead of having to create an object of that class. For example here, I can access the power field from the class item directly and set it to a value.

Let's have a look at a coding example. In this example, I've created a class called Person, and I'm trying to keep count of every single object that was created from the class. So, I've added this public int field called count, and inside the constructor, which is just the default constructor, I'm incrementing that count by one, which means that every time I create a new object of Person, it will add one to that count variable.



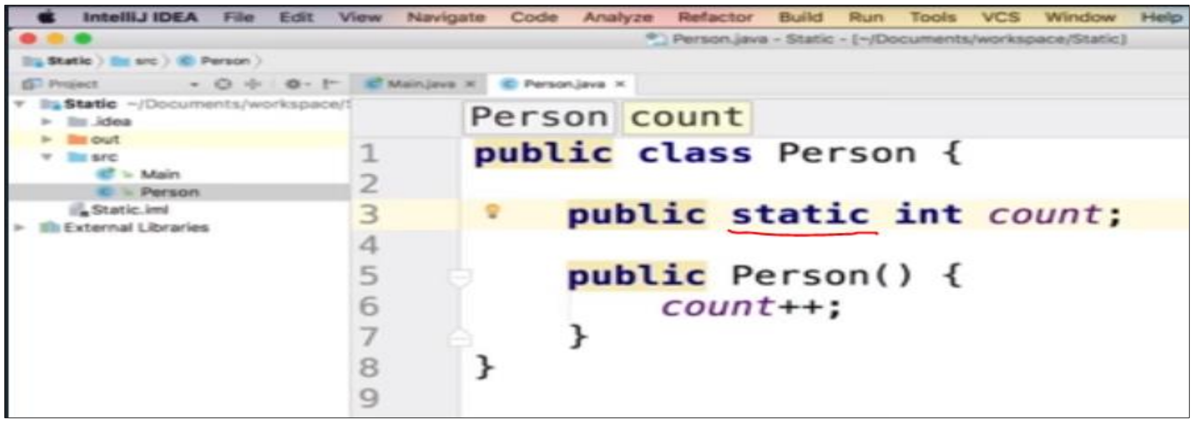
```
public class Person {

    public int count;

    public Person() {
        count++;
    }
}
```

Now if you go to the main method, you could see that I've created this loop that starts from zero and counts all the way to, and inside that loop, I'm creating a new instance of that class Person. And every time I create an object, I print out the count value installed inside it. So, when I run this code, I would expect that every time I'm creating a new Person object, it's adding one to the counter, and by the time we reach to the hundredth Person, the count value would be. So, I'm expecting to see the numbers between one and a hundred. However, if I do run this code, it seems like I'm just getting a bunch of ones, which means that this count variable is not being updated at all.



```
public class Main {

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            Person person = new Person();
            System.out.println(person.count);
        }
    }

}
```

The explanation for that is pretty simple. If we go back to the Person class, you'd notice that I haven't declared this count variable as static, which means that it belongs to the object not the class. And because I'm creating a new object every time, it means I'm creating a new variable called count and I'm only adding one to it. So, every single object would have its own count variable that has the value one. However, if I do add the static keyword to our count variable here, this means that this variable now belongs to the class rather than the object. And every time I'm calling the constructor, it's adding one to the same variable count.



```
public class Person {

    public static int count;

    public Person() {
        count++;
    }
}
```

So if I run this now, I should get all the numbers between one and 100 like I expected earlier. Counting the instances that were created from a class is one of the very common use cases for using the static keyword. So why don't you go ahead and try this yourself as well.

Static Methods -1

The static keyword can also be used to describe methods, allowing you to simply call the method from the class rather than having to create an object first and then calling the method. Static methods, however, have limited capabilities since it can't access non-static fields in the object anymore.
But before we get too much into static methods, go ahead and try out the static fields yourself.

So, when should we declare fields or methods to be static and when should we not?

The short answer is in most cases you would want the variables and methods to belong to a certain object rather than the entire class, which means most of the time you won't declare them as static. However, if you end up creating a class that provides some sort of functionality rather than have a state of its own, then that's a perfect case to use static for almost all of its methods and fields.

For example, remember the Math class that we used to generate random numbers? It turns out that Math is nothing more than a class with a bunch of static methods like random() and others. Because it doesn't really make sense to create an object called math1 and another called math2, all Maths are the same and hence we can simply use the class itself to call its methods directly, and that's why static was a good choice here.

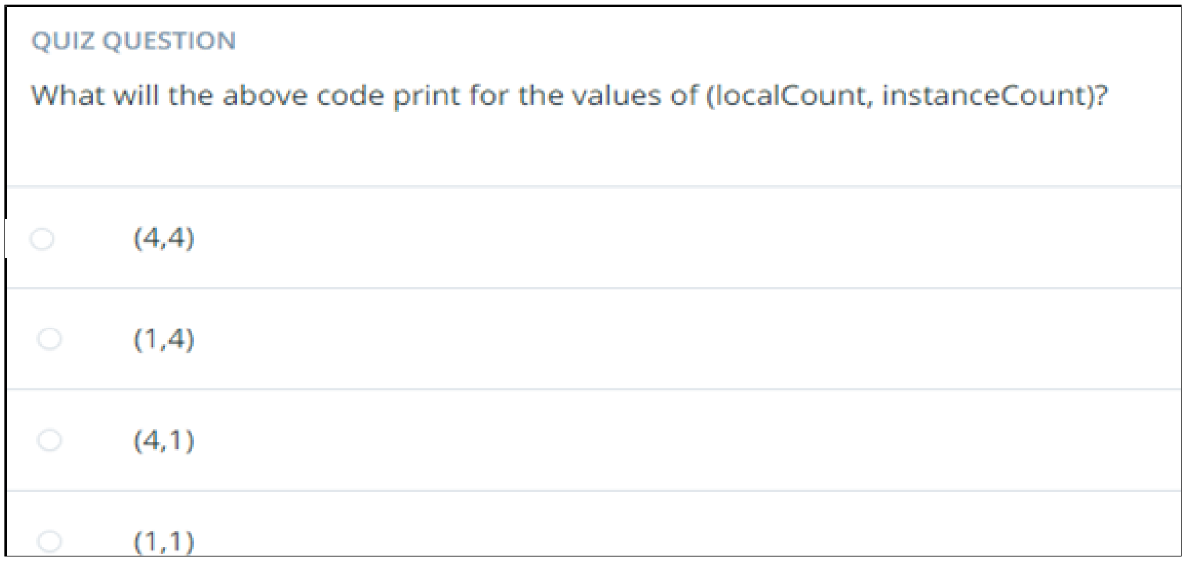
Quiz: Try it out yourself

Create a new project in IntelliJ , and make a new class called Person that contains a static counter and another non-static counter. Increment both counters in the constructor:

```java
public class Person {
   public static int instanceCount;
   public int localCount;
   public Person(){
      instanceCount++;
      localCount++;
   }
}
```

Then, in the main function, create multiple instances of the class Person and check out what the values are for each counter.

```java
public static void main(String[] args) {
  Person person1 = new Person();
  Person person2 = new Person();
  Person person3 = new Person();
  Person person4 = new Person();
  // Print the values of both counters
  System.out.println("(" + person4.localCount + "," + Person.instanceCount + ")");
}
```



QUIZ QUESTION

What will the above code print for the values of (localCount, instanceCount)?

○ (4,4)

○ (1,4)

○ (4,1)

○ (1,1)

Static Methods - 2

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);
```

Summary

Well done. I hope you've learned a few things this lesson.

- ✓ You've seen how to use Inheritance to extend classes and add more functionality to them.
- ✓ Then we've talked about overriding methods and modifying existing methods that were inherited from a parent.
- ✓ We've used the super keyword to access parent fields and methods directly from a child.
- ✓ And then we've seen how to use interfaces that will help us make our code design even better.
- ✓ And finally, we learned that final protects methods from being overridden and fields from being modified and how static methods and fields belong to classes rather than objects.

In the next lesson you'll get to use a very important component of Java called Collections. If you're familiar with arrays it's actually very similar but way more powerful. So, let's get to it.