Java

16 February 2021 19:36

Following Java Programming Masterclass for Software Developers (Udemy).
https://www.udemy.com/course/java-the-complete-java-developer-course/

Section 1 : Course Introduction
Section 2 : Software Tools Setup
Section 3 : First Steps

JRE : Java Runtime Environment enables java code execution.
JDK : Java Development Kit = It's a software that is used to create and run/execute Java Programs.
IDE : Integrated Development Environment. Edit, build, run debug

IDE -> JDK -> (Java App -> JRE -> Host Environment)

Keywords are case sensitive

```
public class Hello {
}
```

Public and class are two java keywords.

Method : Collection of statement that performs an operation.
Main method : Entry point of any java code

("Hello World") = This is literal string

Primitive Types : There are 8. => int, boolean, byte, char, short, long, float, double

Wrapper Classes : For int primitive type we have Integer Wrapper class in java

Size of Primitive Types and Width :

Byte = 8 bits, width = 8
Short = 16 bits, width = 16
Int = 32 bits, width = 32
Char = 16 bits, width = 16 (allow to store unicode characters) it can represent one of 65535 different types of characters.
Boolean = in java - True or False (In general - True or False, Yes or No, 1 or 0)

Casting :

Casting means to treat or convert a number from one type to another.
(byte) (myMinByteValue / 2); (use int by default)

Integer is the best data type for whole numbers generally.

Single and Double Precision :

Float : Single precision, 32 bits, width = 32
Double : double precision, 64 bits, width = 64

Strings : Datatype in java, not primitive type. It's actually a Class.

If we try to add a string with int number, java will int number as a string and make new string.
String in java are Immutable.

Operators : That perform specific Operations. Like +, -, *, /
Operand : Term used to describe any object that is manipulated by an **operator**.
Expression : Combination of Operators and Operand. Formed by combining variables, literals, method return values and operators.

Abbreviating Operators :

- a + 1 can be written as a++.
- a = a + 2 can be written as a += 2

Logical AND and Logical OR : && and || which operates on boolean operands

Ternary Operator ?: = Shortcut of if-then-else statement
Operator Precedence is also Important.

Section 6: OOP Part 1 - Classes, Constructors and Inheritance

Class : A Class is like an object constructor, or a "blueprint" for creating objects or prototype from which objects are created.
Every class we create is inherited from base java class.

Object : An object in Java is the physical as well as a logical entity.

Class_Name Object_Name = new Class_Name();

Put this before private data member to modify then in public function on concur same name convention. **Example :**

```
public class Car {

   private int doors;
   private int wheels;
   private String model;
   private String engine;
   private String colour;

   public void setModel(String model)
   {
```

Section 4 : Java Tutorial: Expressions, Statements, Code blocks, Methods and more

If, Else if, Else

Indentation is Important.

Methods : Functions

Use **DiffMerge** Software to compare 2 files or folders.

Method Overloading : Creating methods (functions) with same name but with different arguments/parameters.

Method Overloading is a feature that allows us to have more than one method with the same name, so long as we use different parameters.

Good practice to follow method overloading.
It improves core readability and re-usability.

Defining a Constant in Java : (Cannot be changed)
public static final String **INVALID_VALUE_MESSAGE** = "Invalid Value";

Section 5: Control Flow Statements

4 key Control flow Statements :

1) Switch statement
2) For statement
3) While statement
4) Do-while statement

Switch Statement :

```
int switchValue = 5;
      switch(switchValue)
      {
         case 1:
            System.out.println("Value was 1");
            break;
         case 2: case 3: case 4 :
            System.out.println("was a 2, or a 3, or a 4");
            System.out.println("Actually it was a " + switchValue);
            break;

         default:
            System.out.println("Was not 1 or 2 or 3 or 4");
            break; // technically don't need this break
      }
```

For Statement :

for(init; condition/termination, increment)

While Statement :

```
While(condition)
{
       // statements
}

int count = 0;
      while(count != 5)
      {
         System.out.println("Count value is " + count);
         count++
      }
```

Do While Statement :

```
Do
{
       // statements
       While(conditions);
}

count = 1;
      do
      {
         System.out.println("Count value is " + count);
         count++;
      }
      while (count != 6);
```

Continue :
Use continue to get back to next iteration. Code below continue of that iteration will not execute.

Break :
Exit the loop depending on the condition.

Reading User Input :

```
    private String engine;
    private String colour;

    public void setModel(String model)
    {
        this.model = model;
    }
}
```

Use like this in Main : porsche.setModel("Carrera");

Concept of Encapsulation : not allowing people to access the field directly. Can validate the data as well.

Default (Empty) Constructor always called when a new object is created.

Instead of calling set and get initially, there is another way of doing this when we are creating an object for the first time using a class, that's using constructors.

General rule : don't call setter from inside a construstor. Call an constructor from inside the constructor is fine.

Inheritance :
Super Keyword : Super means is to call the constructor that is for the class that we are extending from (superclass)

Reference vs **Object** vs **Instance** vs **Class** :

A **Class** is basically a blueprint(plan) for a house, using this we can build as many houses.

Each house we build (instantiate using **new** operator) is an **object** also known as **instance**.

Each house build has an address. If we want to tell somewhere where we live. This is known as **reference**. Can copy reference as many times, but it remain as one house.

We can pass **references** as **parameters** to **constructors** and **methods**.

In java we always have **references** to an **object** in memory, there is no way to access an **object** directly everything is done using a **reference**.

This vs super :

Super is used to access/call the parent class members (variables & methods)
Commonly used with method overriding, when we call a method with the same name from the parent class.

This is used to call the current class members (variables and methods).
Commonly used with constructors and setters, optionally in getters

This() call : to call a constructor from another overloaded constructor in the same class, must be first statement in constructor.

Super() call : only way to call a parent constructor. This calls parent constructor, , must be first statement in each constructor.

Note : A constructor can have a call to **super()** or **this()** but never both.

Methods Overriding vs Overloading :

Method overloading : provides 2 or more separate methods in a class with same name but different parameters. (java developers often refer overloading as Compile Time Polymorphism)

Overriding : means defining a method in a child class that already exist in the parent class with same signature (same name, same argument). Method overriding is also known as Runtime Polymorphism and Dynamic Method Dispatch.
@Override immediately above the method definition.
We can't override static methods only instance methods. And methods can be overridden only in child classes.

Static vs Instance Methods :

Static methods are declared using static modifier and can't access instance methods and instance variables directly.
Static methods don't require an instance to be created. Just type class name dot method name.

Instance methods belong to an instance of a class.
We have to instantiate the class first usually by new keyword

Static Variable share between all its instance. One instance change all will change.

Instance Variables belong to an instance of a class.

Main class automatically inherited from object class in java.

Section 9: Inner and Abstract Classes & Interfaces

Interfaces : So an interface, in java terms, specifies methods that a particular class implements the interface must implement.

Creating a java interface (file instead of class) where we define actual methods. For convention interface file name starts with capital I. Like IName.

```
public interface IName{
        // Methods Declaration just like .h file
        void methodOne();
```

Break :
Exit the loop depending on the condition.

Reading User Input :
```
Scanner scanner = new Scanner(System.in);
System.out.println("Enter your name: ");
String name = scanner.nextLine();

scanner.nextLine(); // handle next line character (enter key)

scanner.close();
```

Section 7: OOP Part 2 - Composition, Encapsulation, and Polymorphism

Car **is a** Vehicle (**Inheritance**)
Computer **has a** Motherboard (**Composition**)

Composition is creating object within objects.

Encapsulation : Hiding the data member of class by making them private so they can't be access from outside the class. And only be accessed using public method or getters, setter/constructor.

Polymorphism : It is really the method or the mechanism on object oriented programming, that allows actions to act differently based on the actual object that the action is being performed on.

If a function is not present in child class, then java will look into its base/parent class, if its exist it will automatically return that.

Section 8: Arrays, Java inbuilt Lists, Autoboxing and Unboxing

Arrays : A data structure that allows you to store multiple values of the same type into a single variable.

```
Int [] myIntArray;
myIntArray= new int[10];
```

Or

```
Int[] myIntArray= new int[10]
```

Or

```
Int[] myNumbers = {1, 2, 3, 4, 5};
```

Or

```
anotherArray = new int[] {1, 2, 3, 4, 5};

Saving value = myIntArray[5] = 50;

Double[] myDoubleArray = new double[10];
```

/r = move curser to next line.

Printing array : System.out.println("array= " + Arrays.toString(array));

Static is Important : That means we don't have to create a new object instance for this class

Array List : It is a resizable array. (Looks like vector (c++))

Defining a new Array List : (of String Type)
```
private ArrayList<String> myArrayList= new ArrayList<String>();

// This will not work (not for primitive add type)
private ArrayList<int> myArrayList= new ArrayList<int>();

Instead use this :
private ArrayList<Integer> myArrayList= new ArrayList<Integer>();
```

Iterator : Iterator<String> iterator = list.iterator();
```
while(iterator.hasNext())
{
        System.out.println(iterator.next());
}
```

Autoboxing :

Interger.valueOf(i); // converting the primitive type Integer to the Object wrapper, to the object in other words, that's autoboxing

Unboxing :

myIntValue.intValue(); // here we are suing .intValue so we are actually unboxing. We are convering it from the object, the object wrapper back into the primitive top of this case the int which is value

// Use psvm, shorcut for creating main() method declaration

Linked List : (It's doubly linked list)

interface file name starts with capital I. Like IName.

```
public interface IName{
        // Methods Declaration just like .h file
        void methodOne();
}
```

Don't write code in interface file, code will be written in java file.

```
public class Names implements Iname {
        // Should contain all methods (overridden) that are mentioned in Iname interface to have a
        valid class.

        @override
        Public void methodOne()
        {
                // code here
        }
}
```

Build in Interfaces by Java:

```
List<String> list = new ArrayList<>();
List<String> list = new LinkedList<>();
List<String> list = new Vector<>();
```

In Java, multiple inheritance is only available by implementing several interfaces.

Passing a int value to String List can be done by : "" + intValue;
Getting int value from string list : Integer.parseInt(StringValue);

Inner Classes :

In java, it's possible to nest a class into another class.

There are 4 types of nested classes:-
1) Static nested classes
2) Nonstatic nested class (we called than an inner class)
3) Local class (Inner class defined inside of a scope)
4) Anonymous class (Nested class without a class name)

Inner Classes Syntax:

```
public class mainClass {
……
……
        private class subClass {
        ……
        ……
        }
}
```

In Main:

```
mainClass mc1  = new mainClass(6);
mainClass.subClass sc1 = mc1.new subClass(1, 12.3);
```

Abstraction : Abstraction is when you define the required functionality for something without actually implementing the details.
We focused on what needs to done, not on how's it's to be done.

Interfaces are by definition in java Abstract.

Abstract class implementation:

```
public abstract class Animal {

   private String name;

   public Animal(String name) {
      this.name = name;
   }

   public abstract void eat();
   public abstract void breathe();

   public String getName() {
      return name;
   }
}

public class Dog extends Animal {

   public Dog(String name) {
      super(name);
   }

   @Override
   public void eat() {
      System.out.println(getName() + " is eating");
   }
```

Sub class must implement those methods (as overridden) that are in super class

=> We need to check about the relationships:
1) Is a,
2) Has a,

```
// Use psvm, shorcut for creating main() method declaration
```

Linked List : (It's doubly linked list)

```
LinkedList<String> list = new LinkedList<String>();
```
List Iterator : `ListIterator<String> listIterator = list.listIterator();`

listIterator.hasPrevious() and listIterator.previous() doesn't work directly. Need you use another variable to maintain going next or previous.

```
if(forward)
        {
                if(listIterator.hasPrevious())
                {
                   listIterator.previous();
                }
                forward = false;
        }
if(listIterator.hasPrevious())
             {
                System.out.println("Output: " + listIterator.previous().toString());
              }
             else
             {
                System.out.println("Output Else ");
                forward = true;
             }
If(!forward)     // same for hasNext()
```

Section 10: Java Generics

Generic Methods:-

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

We can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods –

-> All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

```
public class GenericMethodTest
{
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
     // Display array elements
     for(E element : inputArray) {
       System.out.printf("%s ", element);
     }
     System.out.println();
  }

public static void main(String args[]) {
     // Create arrays of Integer, Double and Character
     Integer[] intArray = { 1, 2, 3, 4, 5 };
     Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
     Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

     System.out.println("Array integerArray contains:");
     printArray(intArray);   // pass an Integer array

     System.out.println("\nArray doubleArray contains:");
     printArray(doubleArray);   // pass a Double array

     System.out.println("\nArray characterArray contains:");
     printArray(charArray);   // pass a Character array
  }
}
```

Bounded Type Parameters (Example) :

```
public class MaximumTest {
  // determines the largest of three Comparable objects

  public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
     T max = x;   // assume x is initially the largest

     if(y.compareTo(max) > 0) {
       max = y;   // y is the largest so far
     }

     if(z.compareTo(max) > 0) {
       max = z;   // z is the largest now
     }
     return max;   // returns the largest object
  }
```

Generic Classes :

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section

Sub class must implement those methods (as overridden) that are in super class

=> We need to check about the relationships:
1) Is a,
2) Has a,
3) Can a

Dog is an animal, Bird is an animal -> That make sense to inherited from animal class rather than implementing an actual animal interface.

Can fly birds, which means to implement interface.

Using both extend super call and interface:-
public abstract class Bird **extends** Animal **implements** CanFly
{……}

Implements used for interface and extends used for abstract classes.

If something is common to all, implement that in abstract class, as it's all methods are mandatory to be implemented in sub classes.

Interface cannot have constructors but abstract class have.

Section 11: Naming Conventions and Packages. static and final keywords

Naming Conventions :-

1) Packages : lower case, unique, not start with number, can starts with .com. Example - com.rj
2) Class Names - CamelCase, Nouns, Starts with Capital Letter (Examples - ArrayList, LinkedList)
3) Interface Names - CamelCase (Example - List, Comparable, Serializable)
4) Methods Names - mixedCase, Often verbs, (Example - size(), getName(), addPlayer())
5) Constants : ALL UPPER_CASE, separate words using underscore _, declared using the final keyword. (Examples - Static final int MAX_INT)
6) Variable names - mixedCase, Meaningful and Indicative, starts with lower case letter, do not use underscore _. (Examples - I, league, BoxLength)
7) Type Parameters - Single Character, capital letters (seen in generic section). Examples - E (Element), K (Key), T (Type), V (Value)

Packages :-

Makes easy to know where to find classes and interfaces that can provide the functions provided by the package.

We can create our own code package (saved as jar) and import it in another project and add the library after that we can use all the functions in our new project. It's like creating a library and then using it in other project directly.

Creating Your Package (Library) :-

Select Package (com.rj.whatever) -> File (Project Structure) -> Artifacts (Add - click on +) -> Select Jar then From modules with dependencies -> Click OK -> Check default location and click OK.

Click Build (Build Artifacts) -> Click on Build in Action menu

Jar file is successfully created just check in folder in directory out -> artifacts -> ProjectName_jar -> jar file is here.

Using Created Package (Library) in other Projects :-

File (Project Structure) -> Libraries (Add - click on +) -> Select Java (And select the Jar file) -> Click OK -> Click OK -> Click OK

Now it's added in this project, can be cross checked in External Libraries.

Scope :-
Actually refers to the visibility of a class, member or variable.

Calling base class method of same name exist also in inner class from inner class :
MainClass.this.functionName();

Access Modifiers :-

At Top Level :
only classes, interfaces and enums can exist at the top level, everything else must be included within one of these.

Public : the object is visible to all classes everywhere, whether they are in the same package or have imported the package containing the public classes.

Package-private: the object is only available within its own package (and its visible to every class within the same package). Package-private is specified by not specifying, i.e. it is the default if you do not specify public. There is not a "package-private" keyword.

Member Level :

Public :
Package-private:
Private : the object is only visible within the class it is declared. It is not visible anywhere else (including in subclasses of its class).
Protected : the object is visible anywhere in its own package (like package-private) but also in subclasses even if they are in another package.

All methods are public in interfaces because all interface methods are automatically public, so lack of an access modifier here does not imply package-private.

Generic Classes :

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

```
public class Box<T> {
  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    Box<String> stringBox = new Box<String>();

    integerBox.add(new Integer(10));
    stringBox.add(new String("Hello World"));

    System.out.printf("Integer Value :%d\n\n", integerBox.get());
    System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

Section 12: Java Collections

Use Java Collections:

```
Private Collection<Seat> seats = new ArrayList<>();
Private Collection<Seat> seats = new LinkedList<>();
Private Collection<Seat> seats = new HashSet<>();
Private Collection<Seat> seats = new LinkedHashSet<>();
Private Collection<Seat> seats = new ArrayList<>();
```

Shallow Copy : Taking a copy, a shallow copy of the same data. Both have same shared objects.

Deep Copy : As opposed to a shallow copy, a deep copy is a copy where the elements are not just references to the same element as in the original list, but are themselves copied.

Interfaces Java Docs : https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

Comparable and Comparator : The comparator interface defines a single method called compare. It's similar to comparable. The comparator interface defines a single method called compare. Unlike comparable the objects to be sorted don't have to implement comparator. Instead an object of type comparator can be created with a compare method that can sort the objects that we are interested in.

Maps : (Key, Value) Pair. For a particular key, there will be only one value and if we try to insert more it will be overridden.
https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

```
Map<String, String> languages = new HashMap<>();
Languages.put("first language", "hindi");
Languages.put("second language", "english");
```

To print the value :

```
System.out.println(languages.get("first language");
```

To check a key already exist or not :

```
if(languages.containsKey("first language"))
{ …… }
```

There is another methos called put if absent and that's only gonna add to the map if the key is not already present.

For printing :

```
for(String key: languages.keySet())
{
        System.out.println(key + " : " + languages.get(key));
}
```

There is no order in hashmap.

May more methods such as .remove, .replace, etc

Immutable Classes : are a great way to increase encapsulation.
1) Don't provide "setter" methods
2) Make all fields final and private.
3) Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final.

https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html

Sets & HashSet : Set cannot defined ordering number, and can't contain duplicates.
https://docs.oracle.com/javase/9/docs/technotes/guides/vm/performance-enhancements-7.html

Protected : the object is visible anywhere in its own package (like package-private) but also in subclasses even if they are in another package.

All methods are public in interfaces because all interface methods are automatically public, so lack of an access modifier here does not imply package private.

But it's not possible to have anything except public methods in an interface.

The lack of an access modifier means the default of package private except with interface methods and variables, which are always public.

Static :

Static methods can directly be accessed using ClassName.staticMethodName(); wihout using object name.
Static methods and fields belong to the class, not to instances of the class, and as a result, can be called by referencing the class name rather than a class instance.
That's main has to be Static.

Non static methods and fields cannot be referenced from a static context. All has to be static that's why in Main when we call any other method or field that also has to static. (in their own classs only)

So as a result, Java can't allow a static method to access non-static fields or methods because they don't exist when the static Methods called but we can call the static methods from non-static once with no problems.

But static methods can access not-static fields and methods from another class because it creats instances of a class in order to do so.

Final :

Used generally to define constant values. Not exactly constants because they can be modified but only once, and any modification must be performed before class constructor finished.

Marking a class final, we can prevent our class from being subclassed.

Adding final in super class methods, prevent sub classes to override that methods.
We cannot override final methods.

All static initialization block called first in the order they define, after that constructor is called.

as final.
https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html

Sets & HashSet : Set cannot defined ordering number, and can't contain duplicates.
https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html

HashSet - equals() and hashCode()
https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-

For overcoming object duplication in sets, we override the hashcode() method.

```
@Override
  public int hashCode() {
     System.out.println("hashcode called");
     return this.name.hashCode() + 57;    // 57 is just a random number.
  }
```

If we mark any method final, it means that method can't be overridden (in sub classes)

Sets - Symmetric & Asymmetric :
https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html

In Set theory 2 differences are defined : symmetric difference and asymmetric difference.
Java doesn't have method for calculating the symmetric difference possibly that's use less often.

The symmetric difference is the element that appear in one set or the other but not both so it can therefore as the union minus intersection

Linked HashMap/TreeMap : Store Data in alphabet order (sorted).

UnmodifiableMap : It's unmodified Map.

Section 13: JavaFX

JavaFX was designed with the MVC, or Model-View-Controller, pattern in mind.
Download JavaFX Plugin : https://gluonhq.com/products/javafx/

It's for building GUI desktop/mobile applications. (A GUI Toolkit)

(On the downside, Java web programming is just much more popular than desktop programming. Within desktop programming, Swing is still much more popular than JavaFX)

I am Not going through this section.

Section 14: Basic Input & Output including java.util

Exceptions : Try, Catch, Throw

1) LBYL : Look before You Leave

```
private static int divideLBYL(int x, int y) {
   if(y != 0) {
      return x / y;
   } else {
      return 0;
   }
}
```

2) EAFP : Easy to Ask Forgiveness Permission

```
private static int divideEAFP(int x, int y) {
   try {
      return x / y;
   } catch(ArithmeticException e) {
      return 0;
   }
}
```

Throwing an exception :-

```
try {
      return x / y;
   } catch(ArithmeticException e) {
      throw new ArithmeticException("attempt to divide by zero");
   }
```

Writing content - FileWriter class and Finally block :

```
try
{
      locFile = new FileWriter("locations.txt");
      for(Location location : locations.values())
      {
         locFile.write(location.getLocationID() + "," + location.getDescription() + "\n");
      }
}
catch(IOException e)
{
      System.out.println("In catch block");
      e.printStackTrace();
}
finally
{
```

Section 15: Concurrency in Java

A **Process** is a unit of execution that as has its own memory space.

If one java application is running and we run another one, each application has its own memory space of **heap.**

A **Thread** is a unit of execution within a process. Each process can have multiple threads.

Each thread has what's called a thread stack, which is the memory that only that thread can access.

So, every Java application runs as a single process, and each process can have multiple threads. Every process has a heap, and every thread has a thread stack.

Concurrency, which refers to an application doing more than one thing at a time. Basically Concurrency means that one task doesn't have to complete before another can start.

Run another thread :-

1st way of using Threads : Using Sub-classes the thread class

First Create a class :

```
public class AnotherThread extends Thread {

   @Override
   public void run()
   {
      System.out.println(ANSI_BLUE + "Hello from " + currentThread().getName());

   }
```

Then create an instance and start it :

```
        System.out.println("In catch block");
        e.printStackTrace();
}
finally
{
        System.out.println("in finally block");
        try
        {
           if(locFile != null)
           {
              System.out.println("Attempting to close locfile");
              locFile.close();
           }
        }
        catch(IOException e)
        {
           e.printStackTrace();
        }
}
```

https://docs.oracle.com/javase/7/docs/technotes/guides/language/try-with-resources.html

Java 7 added new feature :

```
try(FileWriter locFile = new FileWriter("locations.txt"))
{
        for(Location location : locations.values())
        {
           locFile.write(location.getLocationID() + "," + location.getDescription() + "\n");
        }
}
```

Reading data from a file :-

```
Scanner scanner = null;
      try
      {
        scanner = new Scanner(new FileReader("locations.txt"));
        scanner.useDelimiter(",");
        while(scanner.hasNextLine()) {
           int loc = scanner.nextInt();
           scanner.skip(scanner.delimiter());
           String description = scanner.nextLine();
           System.out.println("Imported loc: " + loc + ": " + description);
           Map<String, Integer> tempExit = new HashMap<>();
           locations.put(loc, new Location(loc, description, tempExit));
        }

      }

// Now read the exits
      try (BufferedReader dirFile = new BufferedReader(new FileReader("directions_big.txt"))) {
        String input;
        while((input = dirFile.readLine()) != null)
        { ……. }
```

The process of translating a data structure or object into a format that can be stored and recreated is called **Serialization**.

Java NIO Package : In Java 1.4, A new package was added to the Java SDK.
Called java.nio, the package was described as an improvement to Java I/O because the classes in the package perform I/O in a non-blocking manner.

// **Reading data**

```
try (BufferedReader dirFile = Files.newBufferedReader(dirPath))
{
        String input;
        while ((input = dirFile.readLine()) != null)
        {
                String[] data = input.split(",");
                int loc = Integer.parseInt(data[0]);
                String direction = data[1];
        }
}
catch (IOException e)
{
        e.printStackTrace();

}
```

Java NIO - Reading and Writing

```
 Path dataPath = FileSystems.getDefault().getPath("data.txt");
        Files.write(dataPath, "\nLine 5".getBytes("UTF-8"), StandardOpenOption.APPEND); //writing
         List<String> lines = Files.readAllLines(dataPath);
         for(String line : lines) {
            System.out.println(line);
         }
```

Writing data to a data.dat file :-

```
 try(FileOutputStream binFile = new FileOutputStream("data.dat");
         FileChannel binChannel = binFile.getChannel())
{

         byte[] outputBytes = "Hello World!".getBytes();
         ByteBuffer buffer = ByteBuffer.wrap(outputBytes);
         int numBytes = binChannel.write(buffer);
```

```
    }
```

Then create an instance and start it :

```
Thread anotherThread = new AnotherThread();
anotherThread.setName("== Another Thread ==");
anotherThread.start();
```

There is no order of thread, until we set their priority.

Creating Anonymous Class :-

```
new Thread(
 {
        public void run() {
           System.out.println(ANSI_GREEN + "Hello from the anonymous class thread");
        }
      }.start();
```

2nd way of using Threads : Using Runnable Interface

First Create a class :

```
public class MyRunnable implements Runnable {

   @Override
   public void run() {
      System.out.println(ANSI_RED + "Hello from MyRunnable's implementation of run()");
   }
}
```

Then create an instance and start it :

```
Thread myRunnableThread = new Thread(new MyRunnable());
myRunnableThread.start();
```

Creating Anonymous Class of Runnable:-

```
Thread myRunnableThread = new Thread(new MyRunnable() {
        @Override
        public void run() {
           System.out.println(ANSI_RED + "Hello from the anonymous class's implementation of
run()");
        }
      });

      myRunnableThread.start();
```

Most Developer use runnable as it's more flexible and recommended.

Don't call run method directly (Will call main run, not thread run), must call start method.

Sleeping a thread :-

```
try {
        Thread.sleep(3000);
      } catch(InterruptedException e) {
        System.out.println(ANSI_BLUE +  "Another thread woke me up");
      }
```

Interrupts : We interrupt a thread when we want it to stop what it was doing to do something else.

```
anotherThread.interrupt();
```

Joining threads :- When we join one thread to another thread and the join times out we have to handle that case in a real world application

```
anotherThread.join(); // written in public void run() {…}
```

Local Variables are stored in thread stack, that means that each thread has its own copy of a local variable in contrast the memory required to store an object instance value is allocated on the heap.

The process of controlling when threads execute code and therefore when they can access the heap is called **synchronization.** (To prevent race condition)
If a class has three synchronize methods then only one of these methods can ever run at a time and only on one thread.

To make synchronized :-

1) Add synchronized in function name :-
   public synchronized void doCountdown() { …. }
2) We can also synchronize a block of statements rather than an entire method :-
   synchronized(this) { … }

As a general rule it's easy to just remember not to use local variables to synchronize.

Critical section just refers to the code that's referencing a shared resource like a variable only one thread at a time should be able to execute a critical section.

Thread-Safe : When a class or method is thread-safe what that means is that the developer has synchronized all the critical sections within the code so that we as a developer don't have to worry about the thread interference.

Deadlocks, wait, notify and notifyAll Methods :-
https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList(java.util.List)

Thread Interference :-
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

```
        byte[] outputBytes = "Hello World!".getBytes();
        ByteBuffer buffer = ByteBuffer.wrap(outputBytes);
        int numBytes = binChannel.write(buffer);
        System.out.println("numBytes written was: " + numBytes);
.....
}
```

File System :-
Each folder, which is also referred to as a directory, is also a node in a path. And then, of course, theirs the file itself.

Absolute Path : Specify root mode (starting from which location)
Relative Path : Doesn't specify a root node, it doesn't contain enough information to identify the file.

Reading a file using a given Path :-
```
  private static void printFile(Path path)
  {
    try(BufferedReader fileReader = Files.newBufferedReader(path))
    {
      String line;
      while((line = fileReader.readLine()) != null) {
        System.out.println(line);
      }
    }
    catch(IOException e)
    {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
  }
```

Getting Path from a working project :-
```
Path path = FileSystems.getDefault().getPath("FileName.txt");
printFile(path);
```

For a random path :-
```
filePath = Paths.get("C:\\whole\\path\\fileName.txt");
printFile(path);
```

Get Current Path :-
```
filePath = Paths.get(".");
System.out.println(filePath.toAbsolutePath());
```

FileSystems.getDefault().getPath(".") is same as Paths.get(".");

Copy a File :-
```
Files.copy(sourceFilePath, copyFilePath);
```

Move a File :-
```
Files.move(fileToMove, destination);
```

Delete a File :-
```
Files.delete(fileToDelete);
Files.deleteIfExists(fileToDelete);
```

Create a File :-
```
Files.createFile(fileToCreate);
```

Create a Directory :-
```
Files.createDirectory(dirToCreate);
```

https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html
https://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/BasicFileAttributes.html

Read name of all files in a Directory:-
```
  Path directory = FileSystems.getDefault().getPath("DIrectoryName\\Folder");
    try (DirectoryStream<Path> contents = Files.newDirectoryStream(directory))
                                        // instead of \\ we can use File.separator
    {
      for (Path file : contents)
      {
        System.out.println(file.getFileName());
      }

    }
    catch (IOException | DirectoryIteratorException e)
    {
      System.out.println(e.getMessage());
    }
```

A filter can also be pass on Files.newDirectoryStream as another argument to filter all files in the directory.
https://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystem.html#getPathMatcher-java.lang.String-

Walk the Directory :- To read all the files and directory, files/directories that are itself in a directory can be read.

Take away :- It's to use java.nio when working with a file system. But when it comes to reading and writing file contents, sometimes java.io streams are still the better choice.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html#beforeExecute-java.lang.Thread-java.lang.Runnable-

Thread Interference :-
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Instead of using synchronization we can prevent thread interference using classes that implement the java.util.concurrent locks.lock interface

Use :-
```
ReentrantLock bufferLock = new ReentrantLock();
Replace synchronized block with :
bufferLock.lock();
..... block .....
bufferLock.unlock();
```

Put bufferLock.lock(); before try block and bufferLock.unlock(); in finally block.
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html

Thread Pools :-

Creating Executive Service :-

```
ExecutorService executorService = Executors.newFixedThreadPool(3); // where 3 = 3 number of threads.

New Thread(object).start() is replaced by executorService.execute(producer);

Also need to shut down the executive service using executorService.shutdown();
```

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

Thread safe means that we can be confident that our call to one of the class methods will complete before another thread can run a method in the class.

Deadlock : Deadlock occurs when two or more thread are blocked on locks and every thread that's blocked is holding a lock that another block thread wants.
For example : Thread 1 is holding Lock 1 and waiting to acquire Lock 2 but Thread 2 is holding Lock 2 and waiting to acquire Lock 1. Because all the threads holding the locks are blocked they will never release the locks they're holding and so none of the waiting threads will actually even run.

How to prevent Deadlock :-
1) Use 1 lock, but that's not possible as many applications use multiple locks.
2) If we made both treads obtain the locks in the same order, a deadlock can't occur.
3) Another solution would be to use a lock object rather than using synchronized blocks.

Starvation :- When starvation occurs it's not that threads will never progress because they'll never get a lock but that they rarely have the opportunity to run and progress. So starvation often occurs due to thread priority when we assign a high priority to a thread we are suggesting to the operating system that it should try and run the thread before other waiting threads.

Setting Priority :-
```
Thread t1 = new Thread(new Worker(ThreadColor.ANSI_RED), "Priority 10");
t1.setPriority(10);
t1.start();

private static class Worker implements Runnable { ..... }
```

It's just priority, it still totality depend on operating system to run threads.

Setting priorities can make starvation more likely to happen, So how to prevent it.
While we are dealing in deadlocks, the order in which locks are require was important, but the starvation which thread gets to run when a lock becomes available is important

Alternative to Synchronous blocks is : Fair Locks and Live Locks

ReentrantLock :-
```
private static ReentrantLock lock = new ReentrantLock(true); // true means it's first come first served - FIFO
```

The only thing fair lock guarantees is the first come first served ordering for getting the lock.
Secondly the try lock method doesn't honor the fairness settings so it will not be first come first served and lastly when using fair locks with a lot of threads keep in mind that performance will be impacted to ensure fairness, may things get slow down when there are lot of threads.

Using :-
```
Replace synchronized block with :-
lock.lock();
try {
            ..............................
             // execute critical section of code
finally {
             lock.unlock();
        }
```

Another problem we can have while working with threads is **live lock** which is similar to deadlock but instead of the threads been blocked they're actually constantly active and usually waiting for all the other threads to complete their tasks.

The next potential problem that can arise in a multi-threaded application is called a **slipped condition.** This is a specific type of race condition (aka thread interference).
It can occur when a thread can be suspended between reading a condition and acting on it.

Solution to slipped condition is the same as it is for any type of thread interference: use synchronized blocks or locks to synchronize the critical action of code.
If the code is already synchronized, then sometimes the placement of the synchronization may be causing the problem.
When using multiple locks, the order in which the locks can be acquired can also result in a slipped

Section 16: Lambda Expressions

Every lambda expressions got three parts :-
1) Argument list
2) Arrow token
3) Body

Because the compiler needs to match the lamnda expression to a method, lambda expressions can only be used with interfaces that contain only one method that has to be implemented. So these interfaces are also referred to as functional interfaces.

Using **lambda Expression to create a thread** :-

```
new Thread(()-> {
        System.out.println("Printing from the Runnable");
        System.out.println("Line 2");
        System.out.format("This is line %d\n", 3);
    }).start();
```

Interface Comparator <T>
https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Local variable has to be declared as Final when we use them within an anonymous call.
Because the local variable doesn't belong to the anonymous class instance.

Lambda expressions are treated as nested blocks. So there within the enclosing block scope so what that should mean is that were able to use the local variable.

When working with lambdas if we want to use the local variables in enclosing block they have to be effectively final so that the runtime will know what values to use when the lambda expression is evaluated.

Functional Programming :-
https://en.wikipedia.org/wiki/Functional_programming

Consumer Document :-
https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

Functional Interface & Predicated :-

Function :-
```
  private static void printEmployeesByAge(List<Employee> employees,
                                          String ageText,
                                          Predicate<Employee> ageCondition) {

    System.out.println(ageText);
    System.out.println("==================");
    for(Employee employee : employees) {
      if (ageCondition.test(employee)) {
        System.out.println(employee.getName());
      }
    }
  }
```

Calling from Main :-
```
printEmployeesByAge(employees, "Employees over 30", employee-> employee.getAge() > 30);
printEmployeesByAge(employees, "\nEmployees 30 and under", employee ->employee.getAge()
<= 30);
```

According to age condition it will print the employee names from employees list.

We can do the above same thing using anonymous class also. Like this :-
```
printEmployeesByAge(employees, "\nEmployees younger than 25", new Predicate<Employee>() {
        @Override
        public boolean test(Employee employee) {
          return employee.getAge() < 25;
        }
    });
```

IntPredicate greaterThan15 = i -> i > 15;

```
System.out.println(greaterThan15.test(10));      // output : false

IntPredicate lessThan100 = i -> i < 100;
System.out.println(greaterThan15.and(lessThan100).test(50));      // output : true
```

Function Lambda Expression for getting last from a list of employees :- (Employee is class)
```
Function<Employee, String> getLastName = (Employee employee)-> {
        return employee.getName().substring(employee.getName().indexOf(' ') + 1);
    };

String lastName = getLastName.apply(employees.get(1));
    System.out.println(lastName);
```

Upper case Employee first name :-
```
Function<Employee, String> upperCase = employee -> employee.getName().toUpperCase();
Function<String, String> firstName = name -> name.substring(0, name.indexOf(' '));
Function chainedFunction = upperCase.andThen(firstName);
System.out.println(chainedFunction.apply(employees.get(0)));
```

Package java.util.function :-
https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

condition.

Thread Issues :-

An **Atomic Action** can't be suspended in the middle of being executed. It either completes, or it doesn't happen at all. Once a thread starts to run an atomic action, we can be confident that it can't be suspended until it has completed the action.

Atomic Action are as follows :-
1) Reading and writing reference variables. Statement myObject1 = myObject2
2) Reading and writing primitive variables, except those of type long and double.
    a. myInt = 10 is atomic action, but myDouble = 1.234 is not. (But we can use the AtomicLong and AtomicDouble classes to make theses operations atomic)
3) Reading and writing all variables declared volatile.
    a. Public volatile int counter;
    b. When we use a volatile variable, the JVM writed the value back to main memory immediately after a thread updates the value in its CPU cache.
    c. It also guarantees that every time a variable reads from a volatile, it will get the latest value.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html

JavaFX Background Tasks :-
https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/Task.html

If a variable is a local variable, it's already threadsafe. Local variables are stored on the thread stack, so each thread will have its own copy. Threads won't interface with each other when it comes to setting and getting its value.

When dealing with deadlock situation, look for the following :-
1) Is a set of locks being obtained in a different order by multiple threads. If so, can we force all threads to obtain the locks in the same order?
2) Are we over synchronizing the code?
3) Can we rewrite the code to break any circular call patterns?
4) Would using RecentrantLock object help?

Section 17: Regular Expressions

Regular expressions are way to describe a string or a pattern. Regular expressions are often used to search strings for a specific pattern or to validate that user input matched a specific pattern.

```
String alphanumeric = "abcDeeeF12Ghhiiiijkl99z";
System.out.println(alphanumeric.replaceAll(".","Y"));      // will replace all character of string with Y

System.out.println(alphanumeric.replaceAll("^abcDeee", "YYY"));   // will only replace starting occurance of abcDeee with YYY

System.out.println(alphanumeric.matches("^abcDeee"));   // will return false, return true only when whole string is matched and should be same

System.out.println(alphanumeric.replaceAll("ijkl99z$", "THE END"));   // will replace last of ijkl99z with THE END (opposite of ^)

System.out.println(alphanumeric.replaceAll("[aei]", "X"));   // all occurance of a, e, I will replace with X

System.out.println(alphanumeric.replaceAll("[aei][Fj]", "X"));   // going to perform a replacement if of the three letters A, E, I is actually followed by F or a  J

System.out.println(newAlphanumeric.replaceAll("[^ej]", "X"));   // will replace every letter with X except e and j

System.out.println(newAlphanumeric.replaceAll("[a-fA-F3-8]", "X"));   // can use - character to specify the range.

System.out.println(newAlphanumeric.replaceAll("(?i)[a-f3-8]", "X"));   // using (?i) we can turn of case sensitivity

System.out.println(newAlphanumeric.replaceAll("\\d", "X"));   // replace all digits with X

System.out.println(newAlphanumeric.replaceAll("\\D", "X"));   // replace all non-digits with X

System.out.println(hasWhitespace.replaceAll("\\s", ""));   // with remove all spaces, tabs and new line

System.out.println(hasWhitespace.replaceAll("\t", "X"));   // will replace all tabs

System.out.println(hasWhitespace.replaceAll("\\S", ""));   // will replace all non-whitespace characters

System.out.println(newAlphanumeric.replaceAll("\\w", "X"));   // will replace a-z, A-Z, 0-9 and _

System.out.println(hasWhitespace.replaceAll("\\b", "X"));   // each word will surrounded by X, which means hello string will be XhelloX.
```

Quantifiers : It specifies how often an element in a regular expression can occur.

```
System.out.println(chainedFunction.apply(employees.get(0)));
```

Package java.util.function :-
https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Stream :-
In practice a stream is a set of object references. The stream method which was added to the collections class of object references. The stream method which was added to the collections class in Java 8 creates a stream from a collection. Now each object references in the stream corresponds to an object in the collection and the ordering of the object reference matched the ordering of the collection.
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Example :-
```
List<String> someBingoNumbers = Arrays.asList(
        "N40", "N36",
        "B12", "B6",
        "G53", "G49", "G60", "G50", "g64",
        "I26", "I17", "I29",
        "O71");

someBingoNumbers
        .stream()
        .map(String::toUpperCase)  // :: colon colon notation is called method reference
        .filter(s->s.startsWith("G"))
        .sorted()
        .forEach(System.out::println);
```

Method References :-
https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

```
Stream<String> ioNumberStream = Stream.of("I26", "I17", "I29", "O71");
Stream<String> inNumberStream = Stream.of("N40", "N36", "I26", "I17", "I29", "O71");
Stream<String> concatStream = Stream.concat(ioNumberStream, inNumberStream);
System.out.println("----------------------");
System.out.println(concatStream
        .distinct()
        .peek(System.out::println)
        .count());
```

We may want to map a single object more than one object and we can use a **flat map method** to actually achieve that. The method accepts a function that returns a stream value so we can pass an object as the function argument and returns a string containing several objects which that were effectively mapping one object too many.

```
public class Employee {
   private String name;
   private int age;
……..
}

public class Department {
   private String name;
   private List<Employee> employees;
…….
}
departments.stream()
        .flatMap(department -> department.getEmployees().stream())
        .forEach(System.out::println);
```

Interface Callable<V> :-
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html

Interface Comparator<T> :-
https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

The stream chains are evaluated lazily. Nothing happens until a terminal operation is added to the chain. At that point, the chain is executed.

Interface Stream<T> :-
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Section 19: Databases

Database : the container for all the data that you store.
Database Dictionary : provides a comprehensive list of the structure and types of data in the database.

Table : a collection of related data help in the database.
Field : the basic unit of data in a table.
Column : is another name for field. This can be confusing, but relational databases existed before spreadsheets.
Row : a single set of data containing all the columns in the table. Rows are also called records.
File Flat database stores all data in a single table. This results in a lot of duplication.

Splitting the data is knows as **Normalization**. **Database normalization** is basically the process of removing redundant duplicated and irrelevant data from the tables and the more that this is done the higher the level of normalization.
If we look into a normalization we'll find that we can go up to level 6. 6 normal form but in most

```
System.out.println(hasWhitespace.replaceAll("\\b", "X"));   // each word will surrounded by X, which means hello string will be XhelloX.
```

Quantifiers : It specifies how often an element in a regular expression can occur.

Using "^abcDeee" we can use this "^abcDe{3}" or "^abcDe+" (followed with e) or "^abcDe*" (followed by any character after abcD) or "^abcDe{2,5}"

```
.replaceAll("h+i*j", "Y"))   // replacing all occurrences of h followed by any number of I's followed by at least one j with Y.
```

Class Pattern :-
https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Using Matcher Method :-

```
 StringBuilder htmlText = new StringBuilder("<h1>My Heading</h1>");
htmlText.append("<h2>Sub-heading</h2>");
htmlText.append("<p>This is a paragraph about something.</p>");
htmlText.append("<p>This is another paragraph about something else.</p>");
htmlText.append("<h2>Summary</h2>");
htmlText.append("<p>Here is the summary.</p>");

String h2Pattern = ".*<h2>.*";   // . will match every character and star means zero or more
Pattern pattern = Pattern.compile(h2Pattern);
Matcher matcher = pattern.matcher(htmlText);
System.out.println(matcher.matches());   // will return true or false if pattern is exist or not
```

How to count number of occurrence:-

```
matcher.reset();        // matcher only can used once, so reset it and String h2Pattern = "<h2>";
      int count = 0;
      while(matcher.find()) {
         count++;
        }
```

Class Matcher :-
https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html

Using Pattern and Matcher :-
```
String challenge11 = "{0, 2}, {0, 5}, {1, 3}, {2, 4}";
      Pattern pattern11 = Pattern.compile("\\{(.+?)\\}");
      Matcher matcher11 = pattern11.matcher(challenge11);
      while(matcher11.find()) {
         System.out.println("Occurrence: " + matcher11.group(1));
      }

String challenge12 = "11111";
String challenge13 = "11111-1111";
System.out.println(challenge12.matches("^\\d{5}(-\\d{4})?$"));
System.out.println(challenge13.matches("^\\d{5}(-\\d{4})?$"));
```

Section 18: Debugging and Unit Testing

Debugging :-
Debugging can be done using **breakpoints** and **watchpoints**. Makes it very easier to detect what's the problem is in the program.

We can use a **watchpoint** to stop execution whenever the value of an **expression changes**, without having to **predict a particular place** where this may happen.

A **breakpoint** indicates a line of code or program at which you want the **execution of an application to pause**, a **watchpoint** indicates a **data** item whose **change in value** causes the execution of your application to pause.

Unit Testing :-
One of the important phases in the application development lifecycle is the testing phase.
There are several types of testing, one of them is unit testing which is usually done by developer or development team. When working with java a unit usually refers to a method.

J unit is a popular testing framework that we can use to run unit tests (Can run automated tests on our code).

Parameterized Testing :-
To run bunch of tests on the same method

Section 20: Java Networking Programming

Splitting the data is knows as **Normalization**. **Database normalization** is basically the process of removing redundant duplicated and irrelevant data from the tables and the more that this is done the higher the level of normalization.
If we look into a normalization we'll find that we can go up to level 6. 6 normal form but in most practical application it's rare to go beyond the 3rd level.

A **View** is a selection of rows and columns, possible from more than one table.

SQLite Demo Commands:-

.headers on // to see column names while printing the data

CREATE TABLE contacts (name text, phone integer, email text);
INSERT into contacts (name, phone, email) values('RJ', 34234234, 'rj@email.com');
SELECT * FROM contacts;
SELECT name, phone, email FROM contacts;
INSERT into contacts VALUES("R", 23434234234, "R@email.com");
INSERT into contacts (name, phone) values('K', 34234);

A **sqlite command** can end without semicolon and starts with . (dot)
A **sqlite statement** must end with semicolon

.backup testbackup // will backup current database
UPDATE contacts **SET** email="st@email.com"; // will update all email in database, need to very careful while using UPDATE command.
.restore testbackup // restoring backup database
UPDATE contacts SET email="st@email.com" WHERE name = "K";
SELECT * from contacts WHERE name="RJ";
SELECT name, email from contacts WHERE name="RJ";
DELETE FROM contacts WHERE phone="34234";

.tables : will list all the tables in database
.schema : print out the structure of your table
.dump : gives us the sequence statement for creating the table but all the inserts necessary to populate it with the data that's in it. So it wraps the whole thing and what's called a transaction. WE will get begin transactions and commits. Can be used to copy and paste the output from dumped into our code.
.exit or .quit : exit the sqlite shell

A **Key** in a table is in index which provides a way to really speed up searches and joins on a column. Now when columns are indexed they can be searched much faster than if they are not.
Basically index columns are sorted so that they can be searched through much faster.

Relation Database : ordering of the rows is undefined. They are very similar to java maps or to set. In fact relational database theory is heavily based on set theory.
By defining a key, we are making the data ordered on that column or group or columns and searches etc. work far more efficiently as a result of doing that.
There can be lots of keys on a table but there can only be one primary key. Primary key must be unique.

SQLite Autoincrement :-
https://www.sqlite.org/autoinc.html

.schema
CREATE TABLE songs (_id INTEGER PRIMARY KEY, track INTEGER, title TEXT NOT NULL, album INTEGER);
CREATE TABLE albums (_id INTEGER PRIMARY KEY, name TEXT NOT NULL, artist INTEGER);
CREATE TABLE artists (_id INTEGER PRIMARY KEY, name TEXT NOT NULL);

SELECT * FROM albums ORDER BY name; // lower case will show sorted after upper case
SELECT * FROM albums ORDER BY name COLLATE NOCASE; // can ignore case using NOCASE
SELECT * FROM albums ORDER BY name COLLATE NOCASE DESC; // sort in descending order (ASC for ascending)
SELECT * FROM albums ORDER BY artist, name COLLATE NOCASE; // sort first by album ID and then by album name
SELECT * FROM songs ORDER BY album, track;

Joining the Tables :-
SELECT songs.track, songs.title, albums.name FROM songs JOIN albums ON songs.album = albums._id;
SELECT songs.track, songs.title, albums.name FROM songs INNER JOIN albums ON songs.album = albums._id ORDER BY albums.name, songs.track;
SELECT albums.name, songs.track, songs.title FROM songs INNER JOIN albums ON songs.album = albums._id ORDER BY albums.name, songs.track;

Joining 3 Tables :
SELECT artists.name, albums.name, songs.track, songs.title FROM songs
  ...> INNER JOIN albums ON songs.album = albums._id
  ...> INNER JOIN artists ON albums.artist = artists._id
  ...> ORDER BY artists.name, albums.name, songs.track;

ORDER also matters :-
SELECT artists.name, albums.name, songs.track, songs.title FROM songs
INNER JOIN albums ON songs.album = albums._id
INNER JOIN artists ON albums.artist = artists._id
WHERE albums.name = "Doolittle"
ORDER BY artists.name, albums.name, songs.track;

LIKE and Wild Card Searched and Character (%) :-
SELECT artists.name, albums.name, songs.track, songs.title FROM songs
INNER JOIN albums ON songs.album = albums._id
INNER JOIN artists ON albums.artist = artists._id
WHERE songs.title **LIKE** "%doctor%"
ORDER BY artists.name, albums.name, songs.track;

Creating a VIEW :-

Section 20: Java Networking Programming

Network :- A network is a system of computers connected together so they can share resources and communicate with each other.

Networking :- Networking refers to how the connected computers communicate.

Host : When discussing networking, a machine is usually referred to as a **host**.

Computer on a network (which includes the internet) communicate with each other using a **transport protocol**.

Each application that needs data from the network is assigned a **port** (this includes clients connecting to a server that's on the same machine). When data arrives, the port number is used to router the data to the application that's waiting for it.

IP stands for **Internet Protocol**. WE heard the term **TCP/IP**. This refers to using the TCP protocol with IP addresses, which doesn't necessarily mean the host is connected to the internet. Two applications running on the same host can use TCP/IP to communicate with each other. When the client and server are on the same host, usually the IP address **127.0.0.1**, which is referred to as **localhost**, is used to identify the host.

TCP, which stands for **Transmission Control Protocol**, establishes a two-way connection between hosts, and this connection is reliable in the sense that the two hosts talk to each other. When used with Internet addresses, you get TCP/IP, which uses the client/server model.

When communicating using TCP/IP, the sequence of events is as follows :-
1) The client open a connection to the server
2) The clients sends a request to the server
3) The server sends a response to the client
4) The client closes the connection to the server.

Steps 2 and 3 may be repeated multiple times before the connection is closed.

When using low-level networking API, we will use **Sockets** to establish connections, send request, and receive responses. A socket is one end-point of two-way connection. The client will have a socket, and the server will have a socket.
When we have multiple clients connecting to the same server, we will use the same port number, but each client will have its own socket. We will use the **Socket Class** for the client socket, and the **ServerSocket class** for the server's socket.

Server Code :
```
try(ServerSocket serverSocket = new ServerSocket(5000))
{
        Socket socket = serverSocket.accept();
        System.out.println("Client Connected");

....
}
```

Client Code :

```
 try (Socket socket = new Socket("localhost", 5000))
{
        ......
}
```

When using TCP, some handshaking has to take place between the server and the client. So the client has to connect to the server, and the server has to accept that connection. So the client sends a request and the server sends a response. It's a two way connection, and there's tight coupling between the client and the server. Now the TCP protocol also performs error checking and will resend message that don't make it to the server. It's reliable.

When using UDP, there's no handshaking at all, and the destination host, which may or may not be a server, doesn't actually send any responses to the message sender. So we use it when we don't need a reliable connection or a two way connection or when speed is essential.
UDP uses datagrams, and a datagram is a self-contained message, and it's not guaranteed to arrive at its destination. UDP is often used for time-sensitive communication and when losing he odd message or packet here or there won't matter.

URIs, URLs and URNs :-
https://www.w3.org/TR/uri-clarification/

When working with the java.net package, a **URI** is an identifier that might not provide enough information to access the resource it identifies. A **URL** is an identifier that includes information about how to access the resource it identifies.
URI can specify a relative path, but a **URL** has to be an absolute path, because when we use the **URL**, it has to contain enough information to locate and access the resource it identifies.

A **Scheme** is the part of a URI or URL that appears before colon.
For example, "http", "file" and "ftp" are all **schemes**.

When working with **Low-level API**, we used the following classes: **Socket**, **ServerSocket**, and **DatagramSocket**.
When working with the **High-Level API**, we will use the following classes: **URI**, **URL**, **URLConnection**, and **HttpURLConnection**.

A **URI** can contain **Nine Components :-**
1) Scheme
2) Scheme-specific part
3) Authority
4) User-info
5) Host
6) Port
7) Path

```
INNER JOIN artists ON albums.artist = artists._id
WHERE songs.title LIKE "%doctor%"
ORDER BY artists.name, albums.name, songs.track;
```

Creating a VIEW :-

```
CREATE VIEW IF NOT EXISTS artist_list AS
SELECT artists.name, albums.name, songs.track, songs.title FROM songs
INNER JOIN albums ON songs.album = albums._id
INNER JOIN artists ON albums.artist = artists._id
ORDER BY artists.name, albums.name, songs.track;
```

Cross check using .schema command. Can use those view as any other table.

Deleting a VIEW using DROP command :-

```
DROP VIEW astist_list;    // we can also delete table using DROP, but deleting a view didn't acceft
```

as we have creating it for our convenience.

We can also rename column name while creating a View (So the name don't clash):-

```
CREATE VIEW artist_list AS
SELECT artists.name AS artist, albums.name AS album, songs.track, songs.title FROM songs
INNER JOIN albums ON songs.album = albums._id
INNER JOIN artists ON albums.artist = artists._id
ORDER BY artists.name, albums.name, songs.track;

DELETE FROM songs WHERE track < 50;
SELECT * FROM songs WHERE track <> 71    // <> = !=
SELECT COUNT(*) FROM songs;

UPDATE artists SET name = "One Kitten" WHERE artists.name = "Mehitabel";
SELECT * FROM artists WHERE artists.name = "One Kitten";
SELECT count(title) From artist_list WHERE artist = "Aerosmith";
SELECT DISTINCT title From artist_list WHERE artist = "Aerosmith" ORDER BY title;    // for no
```

duplicates

```
SELECT COUNT(DISTINCT title) From artist_list WHERE artist = "Aerosmith";    // counting distinct
```

title

```
SELECT COUNT(DISTINCT album) From artist_list WHERE artist = "Aerosmith";
```

Download SQLite JDBC :- (Download jar file)
https://github.com/xerial/sqlite-jdbc/releases

2 Ways to create database from java :-

1) Using Driver Manager
2) Using Data source Object (For more advance)

Creating a Table using Driver Manager :-

```
Connection conn = DriverManager.getConnection("jdbc:sqlite:C:\\Users\\rajatkumar\\Documents
\\IdeaProjects\\Java-Programming-Course\\TestDB\\testjava.db");

Statement statement = conn.createStatement();
statement.execute("CREATE TABLE contacts (name TEXT, phone INTEGER, email TEXT)");

statement.execute("CREATE TABLE IF NOT EXISTS contacts " +
                  " (name TEXT, phone INTEGER, email TEXT)");
```

Inserting the data in table :-

```
statement.execute("INSERT INTO contacts (name, phone, email) " +
                  "VALUES('Joe', 45632, 'joe@anywhere.com')");
```

Printing the data :-

```
statement.execute("SELECT * FROM contacts");
ResultSet results = statement.getResultSet();

//  ResultSet results = statement..execute("SELECT * FROM contacts");
while(results.next())
{
    System.out.println(results.getString("name") + " " +
    results.getInt("phone") + " " +
    results.getString("email"));
}

public static final String CONNECTION_STRING = "jdbc:sqlite:C:\\Users\\rajatkumar\\Documents
\\IdeaProjects\\Java-Programming-Course\\Music\\" + DB_NAME;
```

SQLite will create an empty database if it didn't find a data base file in mentioned path.

Column indices are one based (Indexing)

JDBC don't have .schema command support. But we can use **ResultSetMetaData** to get query about number of columns and names of it in a table. Column number starts from 1 not 0.
Interface ResultSetMetaData :-
https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html

Use songArtists.isEmpty()) instead of songArtists == null. Better Way.

Should use " = ?" **Prepared Statements** instead of regular to avoid **SQL injection Attacks**. It will substitute the value and will treat it as a single literal and won't interpreted as sql.

PreparedStatments :- It's good practice to use PreparedStatments because of the potential performance benefit, and because they protect the database against SQL injection attacks.

We do the following to use a PreparedStatement :

1) Declare a constant for the SQL statement that contains the placeholders.
2) Create a PreparedStatetement instance using
    a. Connection.prepareStatement(sqlStmtString)
3) When we're ready to perform the query (or the insert, update, delete), we call the
3) Authority
4) User-info
5) Host
6) Port
7) Path
8) Query
9) Fragment

URI - Uniform Resource Identifier :-
https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
Scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]

A URIs that specify a scheme are called **Absolute URIs**.
A URI doesn't specify the scheme, it's called a **Relative URI**.

The URI doesn't have to be valid to work with it, It only has to be valid when you wanna convert it an absolute location (by converting to url using : URL url = uri.toURL();)

Methods : GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE, CONNECT : -
https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

List of HTTP header fields :-
https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

Download Apache HttpClient :-
http://hc.apache.org/downgloads.cgi

Section 21: Java 9 Module System

Java 9 introduced a new program component known as a module. You can think of a Java application as collection of **Modules**. (Project Jigsaw)

A **Module** is a named collection of data and code.

Every module comes with a module descriptor file that describes the module and contains metadata about the module.
The module descriptor file is always located directly at the module root folder, and always has the name **module-info.java**.

There are 2 types of modules : **Normal Modules** and **Open Modules**.

A **Normal Module**, without the open modifier, grants access at compile time and run time to types in only those packages which are explicitly exported.

An **Open Module,** with the open modifier, grants access at compile time to types in only those packages which are explicitly exported, but grants access at run time to types in all its packages, as if all packages had been exported.

Project Jigsaw has the following primary goals :-

1) Scalable Platform
2) Security and Maintainability
3) Improved Application Performance
4) Easier Developer Experience

Project Jigsaw defines two types of modules :-
Namely :

1) **Named Modules -** don't officially exist in JDK 9 - We just use the term normal for a named module that is not an automatic module (have module descriptor file i.e. module-info.java)
2) **Unnamed Modules** - does not have a name and it is not declared.

An **Automatic Module** is created after adding a JAR file to the module path (doesn't have module descriptor file i.e. module-info.java)

Basic Modules don't officially exist in JDK 9 - We just use the term basic for a module that is not an open Module.

Aggregator Modules - These exist for convenience. Usually they have no code of their own, they just a module descriptor. They collect and export the contents of other modules, this is the reason why they are named as an aggregator.

A **Module Path** is used by the compiler to find and resolve modules. Every Module from a module path needs to have a module declaration (module-info.java).

A **Class Path** represents a sequence of JAR files.

Section 22: Migrating Java Projects to Java 9

Need to Structuring the Project

We do the following to use a PreparedStatement :

1) Declare a constant for the SQL statement that contains the placeholders.
2) Create a PreparedStatetement instance using
   a. Connection.prepareStatement(sqlStmtString)
3) When we're ready to perform the query (or the insert, update, delete), we call the appropriate setter methods to set the placeholders to the values we want to use in the statement
4) We run the statement using PreparedStatements.execute() or PreparedStatement.executeQuery()
5) We process the results the same way we do when using a regular old statement.

Transactions :- A transaction is a sequence of SQL statements that are treated as a single logical unit. If any of the statement fail, the results of any previous statements in the transaction can be rolled back, or just not saved. It's as if they never happened.

Database transactions must be **ACID** compliant.

1) **Atomicity** - If a series of SQL statements change the database, then either all the changes are committed, or none of them are.
2) **Consistency** - Before a transaction begins, the database is in a valid state. When it completes, the database is still in a valid state.
3) **Isolation** - Until the changes committed by a transaction are completed, they won't be visible to other connections. Transactions can't depend on each other.
4) **Durability** -Once the changes performed by a transaction are committed to the database, they're permanent. If an application then crashes or the database server goes down (in the case of a client/server database like MYSQL), the changes made by the transition are still there when the application runs again, or the database comes back up.

Essentially Transactions ensure the integrity of the data within a Database.

SQLite uses transactions by default, and auto-commits by default.
But we change it using :-

```
conn.setAutoCommit(false);
If(condition)
{
        conn.commit();
}
catch(Exception e) {....}
finally {
         try {
            System.out.println("Resetting default commit behavior");
            conn.setAutoCommit(true);
                }
                catch(Exception e) { .....}
}
```

Can also create a GUI Program using JavaFX and connect SQLite database using JDBC.

Section 24: Archived Videos

This section consist of Old JavaFX Introduction Video for JDK 8 and Old JavaFX Code Vs FXml Video for JDK 8.

Java applications can obviously have graphical user interface. JavaFX is being used here and that's a set of API's that can be used to build interfaces in Java .
It essentially a set of Java packages and it's the successor to swing which was Java's UI toolkit.

We can create UI components using code as well as using fxml. Using fxml is best practice.

Section 25: Extra Information - Source code, and other stuff

Single Source code download | https://learnprogramming.academy/

Source Code for All Programs :-

Source Code for ...

Free Course and Programming Guide :-

Free Course and Progr...

Section 22: Migrating Java Projects to Java 9

Need to Structuring the Project

https://jdk.java.net/9/ :-

JDK 9 Releases

JDK 9 has been superseded. Please visit jdk.java.net for the current version.
Older releases, **which do not include the most up to date security vulnerability fixes and are no longer recommended for use in production**, remain available in the OpenJDK Archive.

Section 23: Course Remaster in Progress

A **keyword** is one of the around 57 words in the Java language that have a specific meaning. Time to time new keywords are added in a new version of Java.

List of Java keywords :-
https://en.wikipedia.org/wiki/List_of_Java_keywords

Google Java Style Guide :-
https://google.github.io/styleguide/javaguide.html

Java Operator Precedence Table :-
http://www.cs.bilkent.edu.tr/~guvenir/courses/CS101/op_precedence.html

Statements :- Statements can be compared to sentences in natural languages. A statement form a complete unit of execution.

There are **Three** types of Statements in Java :-

1) The Declaration Statement (int finalScore = 50;)
2) The Expression Statements
   a. Assignments expressions (groupAge = 5, salary*=2)
   b. Increment ++ or decrement -- operator (num++;, result--;)
   c. Methods calls (System.out.println("Testing");
   d. Object creation expressions (Car ownerCar = new Car();)
3) The Control Flow Statement (if-then-else, looping statements, branching statements (break, continue, return)).

Google Java Style Guide :-
https://google.github.io/styleguide/javaguide.html

Methods : Methods are a way to organize your code to reduce duplication and make your code easier to maintain.

The general format of defining a method in Java is :

```
<MODIFIER> <RETURN TYPE> methodName (<PARAMETERS>) {
.....
}
```