Contents

[**Keyboard Shortcuts** 1](#_Toc174628331)

[**CORE JAVA** 1](#_Toc174628332)

[**Primitive Data Types** 1](#_Toc174628333)

[**Arrays** 2](#_Toc174628334)

[**Multidimensional Arrays** 2](#_Toc174628335)

[**Jagged Arrays** 3](#_Toc174628336)

[**Creating an array of objects** 3](#_Toc174628337)

[**Copying Elements From one element to another** 4](#_Toc174628338)

[**Enhanced For Loop/ foreach loop\*** 4](#_Toc174628339)

[**String is not a primitive data type. It is a class** 5](#_Toc174628340)

[**To Create mutable (changable) strings, we use either “StringBuffer” or “StringBuilder”** 5](#_Toc174628341)

[**Static Keyword** 6](#_Toc174628342)

[**Encapsulation** 7](#_Toc174628343)

[**Constructors** 8](#_Toc174628344)

[**Inheritance (this() and super())** 8](#_Toc174628345)

[**Anonymous Objects** 10](#_Toc174628346)

[**Method Overriding** 10](#_Toc174628347)

[**Packages** 11](#_Toc174628348)

[**Access Modifiers** 11](#_Toc174628349)

[**Polymorphism** 13](#_Toc174628350)

[**Dynamic Method Dispatching** 13](#_Toc174628351)

[**Final Keyword** 14](#_Toc174628352)

[**Wrapper Classes.** 14](#_Toc174628353)

[**Abstract Keyword** 15](#_Toc174628354)

[**Inner Class** 16](#_Toc174628355)

[**Anonymous Inner Classes** 17](#_Toc174628356)

[**Interfaces** 19](#_Toc174628357)

[***lambda expressions***. 21](#_Toc174628358)

[**Enums (Enumerations)** 22](#_Toc174628359)

[**Annotations** 24](#_Toc174628360)

[**Exceptions** 25](#_Toc174628361)

[**Hierarchy of Exceptions** 25](#_Toc174628362)

[**“Throw” Keyword** 27](#_Toc174628363)

[**Custom Exceptions** 27](#_Toc174628364)

[**“Throws” Keyword** 28](#_Toc174628365)

[**Accepting User Input** 30](#_Toc174628366)

[**Try … Finally/ Try With Resources** 31](#_Toc174628367)

[**Threads** 32](#_Toc174628368)

[**Extend “Thread” class** 32](#_Toc174628369)

[**Implementing “Runnable” Interface** 34](#_Toc174628370)

[**Using lambda expressions (Anonymous Inner Classes)** 36](#_Toc174628371)

[**Race Conditions** 38](#_Toc174628372)

[**Thread States** 39](#_Toc174628373)

[**collection API/ Collection/ Collections** 40](#_Toc174628374)

[**Collection examples:** 42](#_Toc174628375)

[**Maps examples:** 44](#_Toc174628376)

[**Sorting Collections Using Custom Logic** 47](#_Toc174628377)

[**ForEach and Consumer** 51](#_Toc174628378)

[**Stream API** 52](#_Toc174628379)

[**Filter Method** 54](#_Toc174628380)

[**Map Method** 55](#_Toc174628381)

[**Reduce Method** 56](#_Toc174628382)

[**Sealed Classes** 56](#_Toc174628383)

[**Sealed Interfaces** 57](#_Toc174628384)

[**Record Classes** 58](#_Toc174628385)

[**JUNIT** 61](#_Toc174628386)

[**JUNIT5** 62](#_Toc174628387)

[**Junit5 Features** 63](#_Toc174628388)

[**Junit5 Annotations/ Test Case Life Cycle** 65](#_Toc174628389)

[**Test-Driven Development** 67](#_Toc174628390)

[**DATA STRUCTURES AND ALGORITHMS** 68](#_Toc174628391)

[**Algorithms** 68](#_Toc174628392)

[**Searching Algorithms** 68](#_Toc174628393)

[**Linear Search Algorithm** 69](#_Toc174628394)

[**Binary Search Algorithm** 69](#_Toc174628395)

[**Sorting Algorithms** 70](#_Toc174628396)

[**Bubble Sort Algorithm** 70](#_Toc174628397)

[**Selection Sort Algorithm** 71](#_Toc174628398)

[**Insertion Sort** 74](#_Toc174628399)

[**Quick Sort** 75](#_Toc174628400)

[**Merge Sort** 76](#_Toc174628401)

[**Tips for Determining if your algorithm is good** 80](#_Toc174628402)

[**Data Structures** 80](#_Toc174628403)

[**Linked List** 80](#_Toc174628404)

[**Custom-made Linked List** 81](#_Toc174628405)

[**Stack** 83](#_Toc174628406)

[**Custom-made Stacks** 84](#_Toc174628407)

[**Fixed-Sized Stack** 84](#_Toc174628408)

[**Dynamically-Sized Stack** 85](#_Toc174628409)

[**Queue** 88](#_Toc174628410)

[**Custom-made Queue** 89](#_Toc174628411)

[**Binary Search Tree** 91](#_Toc174628412)

[**Custom-made binary search tree:** 92](#_Toc174628413)

[**JDBC (JAVA DATABASE CONNECTIVITY)** 95](#_Toc174628414)

[**Procedure For Connection A Java Application To A Database** 95](#_Toc174628415)

[**Data Access Model With JDBC** 98](#_Toc174628416)

[**SERVLETS AND JAVA SERVER PAGES (JSP)** 102](#_Toc174628417)

[**Servlets** 102](#_Toc174628418)

[**Procedure for creating your first Java Servlets project** 102](#_Toc174628419)

[**Calling a servlet from another servlet** 105](#_Toc174628420)

[**RequestDispatcher** 105](#_Toc174628421)

[**sendRedirect** 107](#_Toc174628422)

[**sendRedirect with URL Rewriting** 107](#_Toc174628423)

[**sendRedirect using Session** 108](#_Toc174628424)

[**sendRedirect using Cookies** 109](#_Toc174628425)

[**ServletContext and ServletConfig** 110](#_Toc174628426)

[**Servlet Annotation Configuration** 112](#_Toc174628427)

[**Java Server Pages (JSP)** 112](#_Toc174628428)

[**Types of tags in JSP** 114](#_Toc174628429)

[**a)** **Directive** 114](#_Toc174628430)

[**i)** **page** 114](#_Toc174628431)

[**ii)** **include** 115](#_Toc174628432)

[**iii)** **taglibs** 115](#_Toc174628433)

[**b)** **Declaration** 115](#_Toc174628434)

[**c)** **Scriplet** 115](#_Toc174628435)

[**d)** **Expression** 115](#_Toc174628436)

[**Implicit Objects in JSP** 116](#_Toc174628437)

[**Connecting a database to a JSP file using JDBC** 117](#_Toc174628438)

[**MVC Architecture** 118](#_Toc174628439)

[**Expression Language in JSP (JSP EL)** 118](#_Toc174628440)

[**JSTL (JavaServer Pages Standard Tag Library)** 119](#_Toc174628441)

[**Procedure for getting started with JSTL** 120](#_Toc174628442)

[**JSTL Nuances** 120](#_Toc174628443)

[**JSTL SQL Tags** 123](#_Toc174628444)

[**JSTL Function tags** 124](#_Toc174628445)

[**Filters** 125](#_Toc174628446)

[**Logging In/Out Management using Sessions** 128](#_Toc174628447)

[**MAVEN** 132](#_Toc174628448)

[**Maven Example Project – File Uploads** 133](#_Toc174628449)

[**HIBERNATE** 136](#_Toc174628450)

[**Sample Hibernate Project** 136](#_Toc174628451)

[**Annotations in Hibernate** 140](#_Toc174628452)

[**Fetching Data using Hibernate** 142](#_Toc174628453)

[**Embeddables in Hibernate** 143](#_Toc174628454)

[**Mapping Relations In Hibernate** 146](#_Toc174628455)

[**a)** **One-To-One Relationships** 146](#_Toc174628456)

[**b)** **One-To-Many Relationships** 149](#_Toc174628457)

[**c)** **Many-To-Many Relationships** 154](#_Toc174628458)

**Acronymns**

**POJO** – Plain Old Java Object

# **Keyboard Shortcuts**

**psvm + tab** – Generate the “public static void main“ method in intellij.

**sout + tab** – Generate “System.out.println()” method in intellij.

**sysout + ctrl + space** - Generate “System.out.println()” method in eclipse.

**Ctrl + shift + o** – To automatically import all the unknown classes in eclipse

# **CORE JAVA**

## **Primitive Data Types**

long num = 1798798l; // You have to add the “l” (for “lucky”) as the end of integer values because the default data type for integers is “int”

float num = 5.6f; // You have to add the “f” at the end when using floats because the default data type for floating point values is “double”.

int num = 0b101; // This assigns the binary number “5” to the variable.

int num = 0x7E; // This assigns the hexadecimal number “126” to the variable.

int num = 10\_000\_000; // The underscores, ”\_”, help separate values but the value is stored as a regular number, “1000000”

double num = 12E10; // This assigns the exponent of 12 raised to the power 10 to the variable.

char c = 'e';

c++;         // This increment the character value to 'f'

System.out.println(c);

int a = 257;

byte b = (byte) a;

System.out.println(b);          // Returns "1" as it performs 257%256 (256 is the range of data type byte (-127,127))

byte a = 30;

byte b = 10;

byte resultB = a \* b;               // Type mismatch: cannot convert from int to byte

int result = a \* b;                 // Type Promotion

int a = 4;

int b = 5;

int c = 6;

int d = 7;

boolean result;

// result = a<b | c<d;

result = a<b || c<d;            // Similar to above, except this one allows for short circuiting

int i = 6;

do {

  System.out.println("Hi: " + i);

  i++;

} while(i < 5);         // The semicolon is necessary

* Class instances (objects) are stored in the heap memory and their references (addresses) stored in the stack memory of the methods that use them.
* Each method in a program has its own stack memory.
* These objects in the heap store the global variables as well as the method declarations

## **Arrays**

Arrays are stored in the heap memory.

int nums1[] = new int[4]; // Default value of 0

int nums[] = {1,2,3,4};         // Array initialisation using "{}" rather than "[]"

### **Multidimensional Arrays**

int nums[][] = new int[3][4];

for(int i=0; i<nums.length; i++) {

    for(int j=0; j<nums[i].length; j++) {

        nums[i][j] = (int)(Math.random()\*100);

    }

}

// New looping syntax for a multi-dimensional array

for(int n[] : nums) {       // The first variable is an array

    for(int m : n) {        // The second one is the element in the array being looped over

        System.out.print(m + " ");

    }

    System.out.println();

}

### **Jagged Arrays**

int nums[][] = new int[3][];

nums[0] = new int[3];       // Jagged Array

nums[1] = new int[4];

nums[2] = new int[2];

for(int i=0; i<nums.length; i++) {

  for(int j=0; j<nums[i].length; j++) {

     nums[i][j] = (int)(Math.random()\*100);

  }

}

### **Creating an array of objects**

class Student {

    int regNo;

    String name;

    int marks;

    public static void main(String args[]) {

        Student s1 = new Student();

        s1.regNo = 1;

        s1.name = "Daniel";

        s1.marks = 66;

        Student s2 = new Student();

        s2.regNo = 1;

        s2.name = "Victor";

        s2.marks = 77;

        Student s3 = new Student();

        s3.regNo = 1;

        s3.name = "Stellah";

        s3.marks = 63;

        Student students[] = new Student[3];        // An array of objects

        students[0] = s1;

        students[1] = s2;

        students[2] = s3;

        for(int i=0; i<students.length; i++) {

            System.out.println(students[i].name + " - Reg No: " + students[i].regNo + students[i].marks);

        }

    }

}

### **Copying Elements From one element to another**

This can be done using a for loop. However, java comes with an inbuilt method that allows you to do the same:

System.arraycopy(originalArray, startIndex, destinationArray, startIndex, numberOfValuesToCopy);

## **Enhanced For Loop/ foreach loop\***

Syntax:

for(<type of variables in array> <name of the variable holding the element being lterated over> : <source array>)

int nums[] = new int[4];

nums[0] = 4;

nums[1] = 3;

nums[2] = 2;

nums[3] = 1;

for(int n: nums) {          // There is no counter. You only need to specify the variable name of the element being looped over and the source array.

   System.out.println(n);

}

for(Student stud: students) {

   System.out.println(stud.name + " - Reg No: " + stud.regNo + stud.marks);

}

## **String is not a primitive data type. It is a class**

String name = new String("Daniel");         // Object creating

String name1 = "Karongo";                   // Java Shorthand for the above

String name2 = "Karongo";                   // Does not create a new object but instead assigns the address of the already existing "Karongo", in the String Constant Pool, in the heap memory, to "name2". In that case, both "name1" and "name2" refer to the same object.

name1 = name1 + "Kungu";                    // This creates a new object with the value "Karongo Kungu" and assigns its address to "name1", overwriting the address of the original "Karongo" String literal. This "Karongo" is then liable for garbage collection

// This implies that “String” objects are immutable (non-changing)

## **To Create mutable (changable) strings, we use either “StringBuffer” or “StringBuilder”**

* “StringBuffer” is thread-safe whereas “StringBuilder” is not. Otherwise they work the same

StringBuffer name = new StringBuffer("Karongo");

StringBuffer name1 = new StringBuffer();

System.out.println(name.capacity());        // returns 23 (16 + length of "Karongo")

System.out.println(name1.capacity());       // returns 16 - the space alotted in case of reassignment

name.append("Daniel");                  // How you concatenate strings in StringBuffer

name1.append("Kungu");

name1.insert(0, "Daniel");       // How you concatenate string at a specific index

System.out.println(name1);

name.deleteCharAt(2);

System.out.println(name);                   // Returns "KarongoDaniel"

System.out.println(name.capacity());

System.out.println(name.length());          // The capacity does not change but the length does

name.setLength(10);                               / Removes the extra "el"

System.out.println(name);                   // Returns "KaongoDaniel"

System.out.println(name.capacity());

System.out.println(name.length());

## **Static Keyword**

This makes the member (variable/ method) associated with it to be assigned to the class itself rather than to the objects/instances of the class. E.g

class Mobile {

    String brand;

    int price;

    static String phoneType;      // All the students will share this as their "stdType" and changing this value will change it for all the class instances as well.

    static {                    // Called only once, when the class is loaded.

        phoneType = "Smartphone";     // Initialisation of the static variable

    }

    public Mobile(){                // Object Constructor (called every time there is a new object being created for the class)

        brand = "Samsung";

        price = 20000;

        System.out.println("In constructor");

    }

public static void showDetails(Mobile mobile) {

        System.out.println("Phone Type: " + phoneType);

        // System.out.println(brand + price);       // You cannot use instance variables directly in a static method

        // System.out.println(mobile.brand + mobile.price);     // You have to pass an object to a static method so as to access the instances variables

    }

}

class Demo {

    public static void main(String args[]) throws ClassNotFoundException {

        Class.forName("Mobile");        // This is used to explicitly load classes, "Mobile" in this case, because, even though a class is loaded once, it will not be loaded until at least one object is created or a static method is called.

//Class.forName(“Mobile”).newInstance(); - Does the same thing as above as well as create an object of the same class.

        // As a result, "System.out.println("In static method");" in the "Mobile", class "static" block will execute

    }

}

## **Encapsulation**

This is where the properties of a class are set to “private” so that the only way to access them is by using getters and setters.

class Human {

    private int age;

    private String name;

    public int getAge() {

        return age;

    }

    public void setAge(int age) {

        this.age = age;

    }

    // public void setAge(int age) {

    //     age = age;       This would modify the local variable rather than the instance variable, i.e. it would update the parameter variable with itself rather than update the variable of the object whouse setter is being called.

    // Preference is always given to local variables over instance variables.

    // }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

}

class Demo {

    public static void main(String args[]) throws ClassNotFoundException {

        Human hum1 = new Human();

        hum1.setAge(30);

        hum1.setName("David");

        System.out.println(hum1.getName() + " : " + hum1.getAge());

    }

}

## **Constructors**

* Constructors in java have the same name as the class.
* Every class has a default constructor.

public Human() { // default constructor

    }

    public Human(int age, String name) {        // Parameterised constructor/ Method overloading (same mothod name different parameters)

        this.age = age;

        this.name = name;

    }

## **Inheritance (this() and super())**

Every class’s default constructor has an unseen, unless explicitly written, “super()” method that is called every time an object is created. This method calls the default constructor of the parent class. At the same time, all classes in java extend a class called “Object”.

Also, every constructor, especially the non-default/parameterized constructors, can use the “this()” method. This “this()” calls the default constructor of the current class. For this reason, it becomes possible to call two constructors in the same class when creating a class instance of the said class.

class A {   // Extends "Object"

    public A() {

        System.out.println("In A");

    }

    public A(int n) {

        this();               // Calls the default constructor of A

        System.out.println("In A int");

    }

}

class B extends A {

    public B() {

        this(5);            // Calls the parameterised constructor of B

        System.out.println("In B");

    }

    public B(int n) {

        super(5);           // Calls the parameterised constructor of A

        System.out.println("In B int");

    }

}

public class Inheritance {

    public static void main(String[] args) {

        B bObj = new B();

    }

}

*PS D:\Projects\Java\Refresher> java Inheritance*

*In A*

*In A int*

*In B int*

*In B*

In java, multi-level inheritance is allowed, i.e., if class C extends class B, and B extends A, then C gets all the properties/methods of class A. (Transitive property allowed).

However multiple inheritance is not allowed, i.e., class C cannot extend both A and B, if the two are not “related” (none of them extends the other).

## **Anonymous Objects**

These are objects that have no reference variable in the stack memory, i.e. objects created without their addresses being in a variable.

There only problem is that you cannot reuse them. Because you cannot refer to them.

class Example {

    Example() {

        System.out.println("Object Created");

    }

    public void show() {

        System.out.println("Showing");

    }

}

class Anonymous {

    public static void main(String[] args) {

        new Example();

        new Example().show();

    }

}

*PS D:\Projects\Java\Refresher> java Anonymous*

*Object Created*

*Object Created*

*Showing*

## **Method Overriding**

This is where methods have the same name, parameter types, parameter numbers and return types and yet function independently. It happens during inheritance when the child class has its own method definition such that when that method is called, the child method definition is chosen over the parent’s definition

class A {

    public int add(int n1, int n2) {

        return n1 + n2;

    }

}

class B extends A{

    public int add(int n1, int n2) {

        return n1 + n2 + 1;

    }

}

public class MethodOverriding {

    public static void main(String[] args) {

        B b = new B();

        int r1 = b.add(3, 4);       // The add() method in the child class, B, is called

        System.out.println(r1);

    }

}

## **Packages**

These are the folders that separate files, class files, in java.

* A class defines its package on the first line of the file.
* You can have packages within packages.
* When using a class in another package, it is imported from its package, as *“import packageName.classNme”*
* Every class imports the package “java.lang.\*” by default, the package containing classes such as “System” in “System.out.println”. The asterisk, “\*”, in an import statement refers to all the FILES, not directories, within a package.
* When sharing packages as libraries worldwide, these need to have a unique name. One way to ensure this is by using the reverse of the domain name, say “com.google.Calculator”. A good place to see this publicly shared libraries/packages is [***https://mvnrepository.com/***](https://mvnrepository.com/)

## **Access Modifiers**

Variables, as well as methods, can have their areas of visibility be restricted. There are 4 access modifiers in java:

* Public – can be accessed from anywhere.
* Private – can only be accessed from within the class. All instance variables should be private.
* Default (No other access modifier is specified – can be accessed from different classes but within the same package.
* Protected – can be accessed from different classes and packages, but the class has to be a child class

Note:

When using the “protected” access modifier and child classes are outside the package:

* You will need to create an instance of the child class from within the child class itself and then access the protected property (variable)/method using that instance.

package Protected;

public class ProtectedTest {

    protected int marks = 70;

    protected void show() {

        System.out.println("Hello");

    }

}

The child class:

package other;

import Protected.ProtectedTest;

public class ModifierTester extends ProtectedTest {

    public static void main(String[] args) {

        ModifierTester mT = new ModifierTester();

        mT.show();

    }

}

* To compile the .java files from the terminal you will need to compile the two files together such as:

*PS D:\Projects\Java\Refresher>* ***javac Protected\ProtectedTest.java other\ModifierTester.java***

## **Polymorphism**

This refers to something having multiple implementations depending on how it is used. There are two kinds: compile-time (method overloading), run-time(method overriding).

## **Dynamic Method Dispatching**

class A {

    public void show() {

        System.out.println("In A show");

    }

}

class B extends A{

    // Method overriding

    public void show() {

        System.out.println("In B show");

    }

}

class C extends A{

    // Method overriding

    public void show() {

        System.out.println("In C show");

    }

}

public class DynamicMethodDispatching {

    public static void main(String[] args) {

        // Dynamic Method Dispatching. Run-time polymorhism. Note the type of the reference variable "obj" is "A", the parent class but the instance is being changed to the children classes.

        A obj = new A();

        obj.show();

        obj = new B();

        obj.show();

        obj = new C();

        obj.show();

    }

}

## **Final Keyword**

This keyword is used with variables, classes and methods. When used with:

* Variables – Its makes the variable a constant (it can not be reassigned a new value)
* Classes – It makes the class uninheritable, i.e., the class cannot be extended.
* Method – It prevents method overriding by the class’s subclasses.

## **Wrapper Classes.**

Java has primitive data types but some java-based frameworks may have restrictions that require the use of objects exclusively rather than primitives. For this reason, java has classes that correspond to the primitives. For example:

* int – Integer
* double – Double
* char – Character

These classes are what are called wrapper classes. The process of storing a primitive value in an object is called ***“boxing”*** whereas that for getting the primitive from the object is called ***“unboxing”***.

public class WrapperClasses {

    public static void main(String[] args) {

        int x = 10;

        // Integer y = new Integer(x);  // Boxing

        Integer y = x;                  // Auto-boxing

        // int z = y.intValue();        // Unboxing

        int z = y;                      // Auto-unboxing

String str = "12";

        int num = Integer.parseInt(str);

        System.out.println(num \* 2);

        System.out.println(z);

    }

}

## **Abstract Keyword**

This keyword has the following implications:

* When used with methods, it is used to only declare a method but not to define its implementation.
* A class with any abstract method must also be declared as abstract.
* You cannot create an object of an abstract class.
* \*Any class that extends an abstract class must implement all the method declarations of the parent class’s abstract methods.
* If a child class of an abstract parent class cannot implement all the abstract methods of the parent, it must also be declared as abstract.

abstract class Car {

    // Method declaration with no implemetation

    public abstract void drive();

    public abstract void fly();

    // An abstract Class can also have concrete (non-abstract) methods

    public void playMusic() {

        System.out.println("Play Music");

    }

}

abstract class Toyota extends Car {

    // The child class has not implemented all the abstract methods of Car and so must also be declared as abstract

    public void drive() {

        System.out.println("Driving");

    }

}

class Corolla extends Toyota {

    // The class implements the last remaining method declration of the parent (Car) class. It therefore doe not need to be declared as abstract.

    public void fly() {

        System.out.println("Flying");

    }

}

// The class can be declared abstract but not have any abstract methods

abstract class Robot {

    public void killPeople() {

        System.out.println("Killing people");

    }

}

public class AbstractClasses {

    public static void main(String[] args) {

        // Cannot create objects of abstract classes

        // Car car = new Car();

        // Toyotta toyotta = new Toyotta();

        // Robot robot = new Robot();

        Corolla corolla = new Corolla();

        corolla.drive();

        corolla.fly();

        corolla.playMusic();

    }

}

## **Inner Class**

A class can have a class nested inside it, in the same way that it can have variables and methods.

// This outer class cannot be declared as static

class A {

    public void show() {

        System.out.println("In A Show");

    }

    // To Access B we need an object of class A

    class B {

        public void configB() {

            System.out.println("In Config B");

        }

    }

    // To Access C we do not need an object of class A

    static class C {

        public void configC() {

            System.out.println("In Config C");

        }

    }

}

public class InnerClasses {

    public static void main(String[] args) {

        A obj = new A();

        obj.show();

        // The period (.) denotes that B is the inner class to A

        A.B obj1 = obj.new B();

        obj1.configB();

        // The object of A is not necessary when creating an object of C because C is static

        A.C obj2 = new A.C();

        obj2.configC();

    }

}

*PS D:\Projects\Java\Refresher>* ***javac InnerClasses.java***

*PS D:\Projects\Java\Refresher>* ***java InnerClasses***

***In A Show***

***In Config B***

***In Config C***

## **Anonymous Inner Classes**

Sometimes we may want to override a method within a class but only want to do that once or a similar small number of times. In this case extending the class and creating a new one is unnecessary. To replicate the method overriding we can use anonymous inner classes.

class A {

    public void show() {

        System.out.println("In A show");

    }

}

// Unnecessary

// class B extends A {

//     public void show() {

//         System.out.println("In new show");

//     }

// }

// You can also create an anonymous inner class of an abstract class.

abstract class B {

    public abstract void show();

    public abstract void config();

}

public class AnonymousInnerClasses {

    public static void main(String[] args) {

        A obj = new A() {

            public void show() {

                System.out.println("In New Show");

            }

        };

        obj.show();

        // B obj1 = new B();        You cannot instantiate an abstract class

        B obj1 = new B() {

            public void show(){

                System.out.println("In show of the abstract class's child");

            }

            public void config(){

                System.out.println("In config of the abstract class's child");

            }

        };

        obj1.show();

        obj1.config();

    }

}

*PS D:\Projects\Java\Refresher>* ***javac AnonymousInnerClasses.java***

*PS D:\Projects\Java\Refresher>* ***java AnonymousInnerClasses***

***In New Show***

***In show of the abstract class's child***

***In config of the abstract class's child***

Compilation of the above code results in the two class files: “AnonymousInnerClasses.class” and “AnonymousInnerClasses$1.class”, the latter being the class file for the anonymous inner class.

## **Interfaces**

An interface, like an abstract class, is used when you want to define an entity and specify its behavior (methods) but also not providing the implementation.

* All method declarations are “public” and “abstract” by default and these keywords can therefore be omitted.
* All variables within an interface must be initialized (since variables cannot be extended between classes, only methods).
* All variables within an interface are “final” and “static” by default.
* Classes implement interfaces.
* A class can implement more than one interface.
* Interfaces can also implement other interfaces.
* Like abstract classes, it is impossible to instantiate an interface.
* Like classes, it is possible to declare a variable of the interface’s type, but instantiate it with an object of a “child” class (a class implementing the interface).

// class -- class  -> 'extends'

// class -- interface  -> 'implements'

// interface -- interface  -> 'extends'

interface A {

    int age = 22;       //  "final" and "static" by default

    void show();

    void config();

}

interface B extends A {         // Interface B inherits both the "show()" and "config()" methods

    void render();

}

interface C {

    void display();

}

class D implements B, C {           // The class implementS both interfaces B and C making it necessary to implement all the methods, lest it be declared as "abstract"

    public void show() {

        System.out.println("Showing");

    }

    public void config() {

        System.out.println("Configuring");

    }

    public void render() {

        System.out.println("Rendering");

    }

    public void display() {

        System.out.println("Display");

    }

}

public class Interfaces {

    public static void main(String args[]) {

        A obj = new D();        // The reference is of type A, the interface, but the object is of the class D

        obj.config();

        obj.show();

        D obj1 = new D();       // Regular class instantiation

        obj1.config();

        obj1.show();

        obj1.render();

        obj1.display();

    }

}

**Types of Interfaces**

* Normal Interfaces

These have more than one method.

* Functional/ Single Abstract Method (SAM) Interfaces

These have only one method. They can be marked by an annotation “***@FunctionalInterface***” so that all who have access to the code know that it should have only once method, and return an error when more than one method is declared.

They also allow for

### ***lambda expressions***.

These are java 8+ shorthands that were introduced to make the code less verbose. There only work for functional interfaces. For example:

@FunctionalInterface

interface A {

    int add(int i, int j);

}

public class FunctionalInterfaces {

    public static void main(String[] args) {

        // A obj = new A() {

        //     public int add(int i, int j) {

        //         return i + j;

        //     }

        // };

        //  System.out.println(obj.add(3,4));

        // Equivalent to the code comment above. This kind of notation is called a "lambda expression"

        A obj = (i,j) -> i+j;

        System.out.println(obj.add(3,4));

    }

}

* Marker Interfaces

These have no method. Often used for serialization (where you want to save the objects to the hard disk) and deserialization (reading the contents of the saved file into the object). E.g. in video games to save the player progress.

## **Enums (Enumerations)**

These are classes used to list out things, often unchangeable, sort of like a constant array. However, their values are objects and the can be given attributes. E.g. An enum containing a list of the days of the week and each day having an attribute like the “level of traffic” say for a bank. This attribute would be linked to the objects via a parameterised constructor with the name of the enum as the name of the class and consequently the name of the constructor.

* Enums cannot be extended.
* Enums can also not be instantiated used the “new” keyword.

enum Status {

    Running, Pending, Failed, Success;

}

// class A extends Status {         Will not work

// }

enum Laptop {

    // The values/ Objects of the Class "Laptop"

    Lenovo(35000), Hp(30000), Acer(25000), Dell(33000), Compaq;

    private int price;

    //  The default constructor of the class. Used with "Compaq"

    private Laptop() {

        this.price = 28000;

        System.out.println("Default Constructor");

    }

    // Parameterised constructor

    private Laptop(int price) {

        this.price = price;

        System.out.println("Parameterised Constructor");

    }

    // Getter for the class "Laptop", because "price" is private

    public int getPrice() {

        return price;

    }

    // Setter for the class "Laptop", because "price" is private

    public void setPrice(int price) {

        this.price = price;

    }

}

public class Enums {

    public static void main(String[] args) {

        Status s = Status.Running;

        // System.out.println(s);       Same as "s.name()"

        System.out.println(s.name());                       // Returns "Running"

        // System.out.println(s.getClass().getName());

        System.out.println(s.ordinal());                    // Returns "0"

        System.out.println(s.equals(Status.Running));       // Returns "true"

        // if(s == Status.Success) {

        //     System.out.println("All done");

        // }

        switch(s) {

            case Running:

                System.out.println("In Progress");

                break;

            case Pending:

                System.out.println("Please wait");

                break;

            case Failed:

                System.out.println("Sorry. Unsuccessful");

                break;

            default:

                System.out.println("All done ");

                break;

        }

        Status[] ss = Status.values();

        for(Status iS: ss) {

            System.out.println(iS.name() + " : " + iS.ordinal());

        }

        Laptop[] laptops = Laptop.values();

        for(Laptop laptop: laptops) {

            System.out.println(laptop.name() + " : " + laptop.ordinal() + " : " + laptop.getPrice());

        }

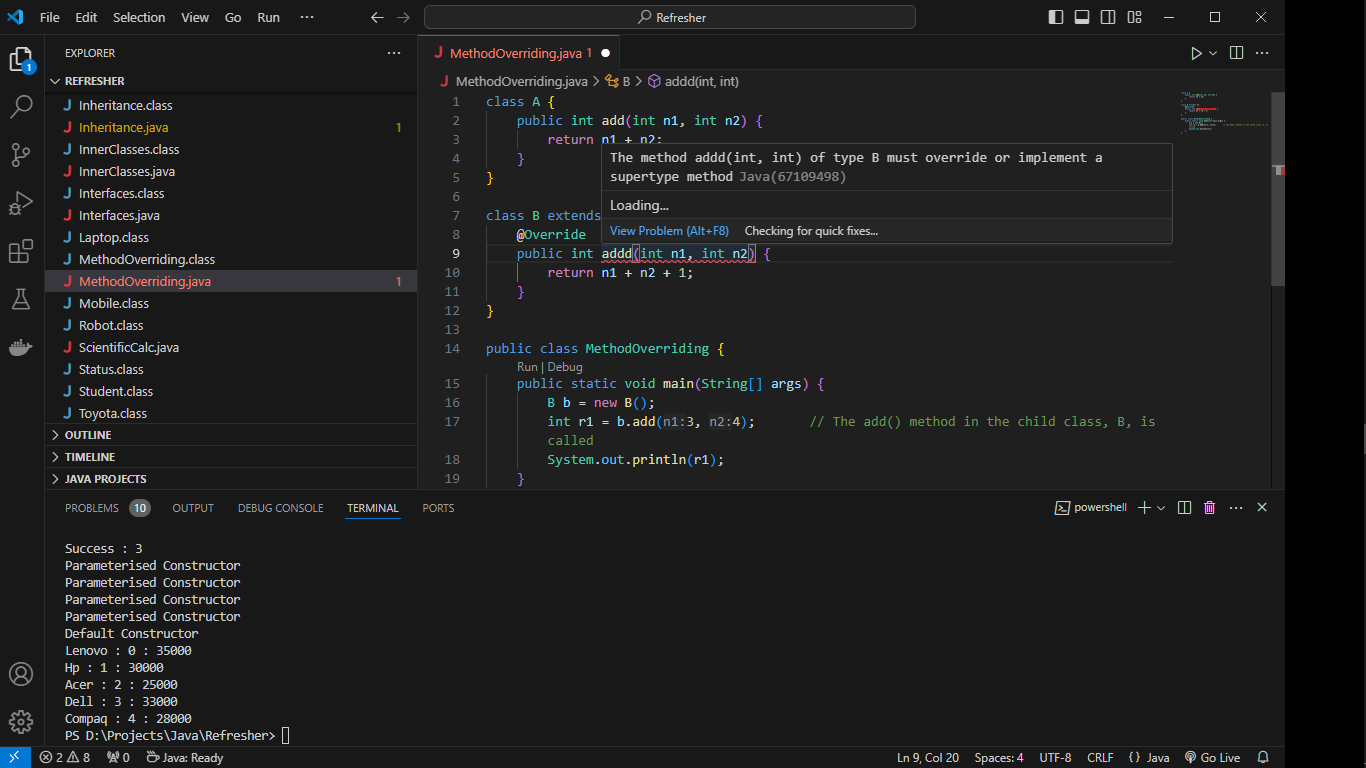
    }

}

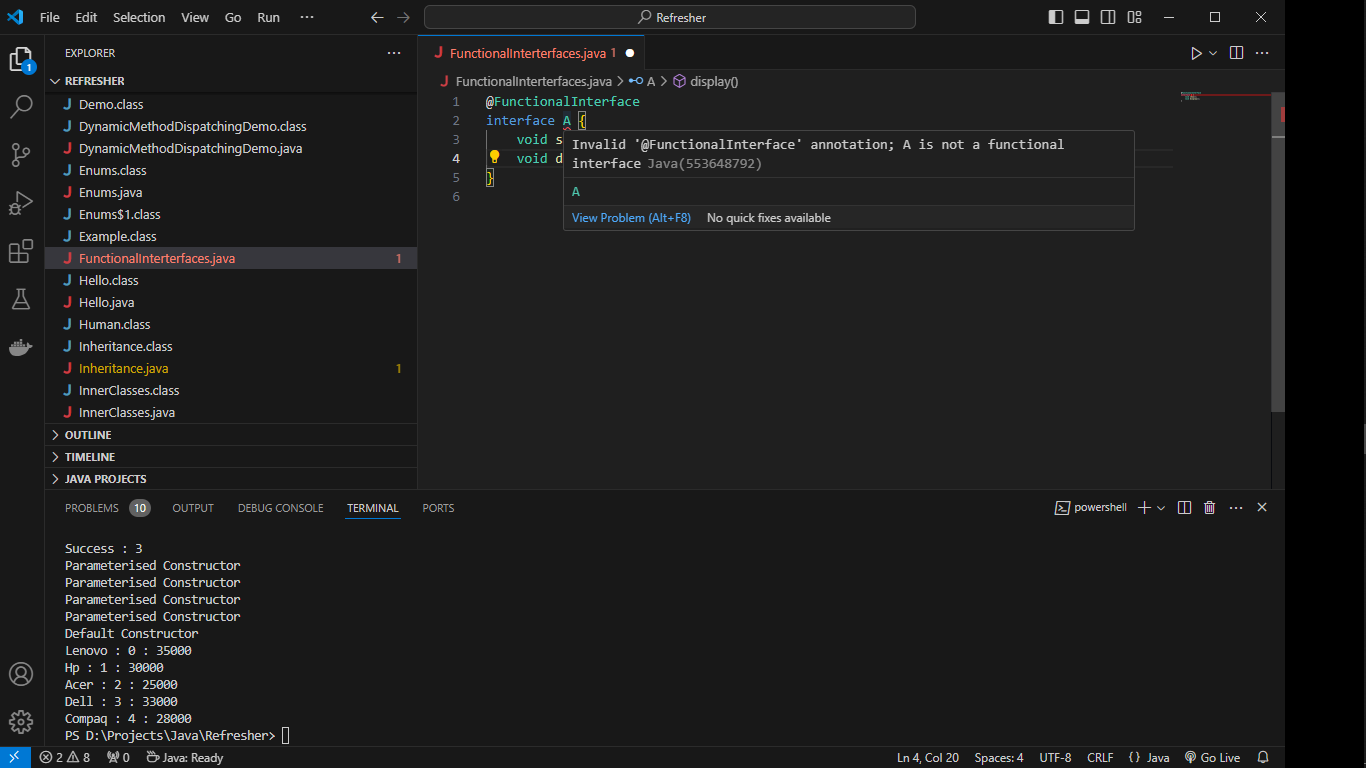
## **Annotations**

These are used to give the compiler some metadata.

* **@Override** – Denotes that the method under this line is an overriding of another in a parent class. E.g. When used and the method name does not match the parent’s method name, it returns an error on the child’s method name.



* ***@FunctionalInterface*** – Used with interface declarations to make sure that it is a functional interface so that all who have access to the code know that it should have only one method, and return an error when more than one method is declared.



## **Exceptions**

These are runtime errors. If left unhandled they terminate the program execution which is undesirable. To handle them, we wrap the *critical statements* within a “try” block and handle the error with the “catch” block.

This catch block is used to take the error, thrown by the compiler, and do something decided on by the developer. The errors can be of different types but all these types are objects of various classes.

For example, they may be of class:

* ArrayIndexOutOfBounds – thrown when try to access an element that is outside the length of the array you are specifying
* NullExceptionPointer – thrown when try to perform some operation that cannot take a “null” value as an operand.
* ArithmeticException – thrown when the numerical operation you are trying to perform cannot be done, e.g. diving a number by zero.
* .
* .
* .
* Exception
* Throwable

### **Hierarchy of Exceptions**

**Note:** Each type of exception should be handled in a different “catch” block. These exceptions have a hierarchy, with the higher classes recommended to be handled further down in the catch section of the code.

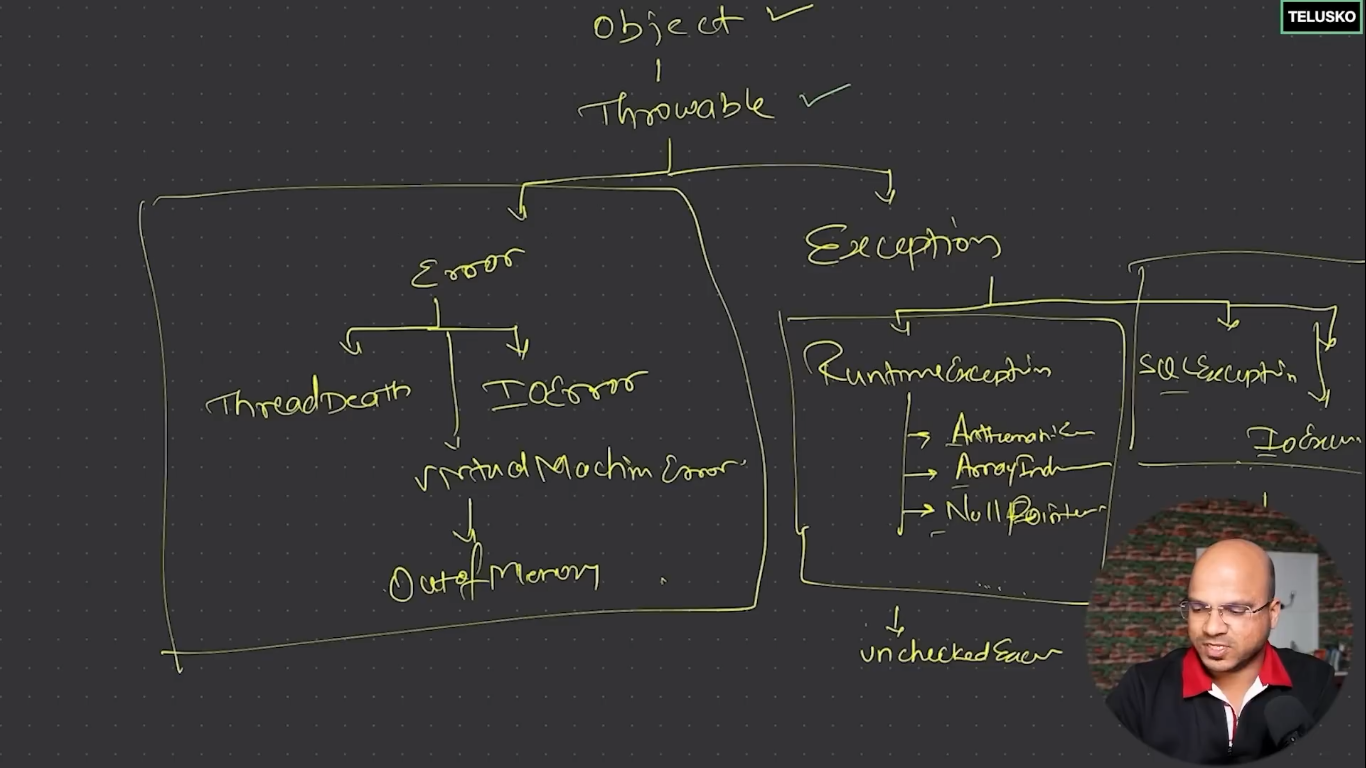


Figure 1: At 9:44:17 of tutorial

public class Exceptions {

    public static void main(String[] args) {

        int i = 2;

        int j = 0;

        int nums[] = {2,2,34,5};

        int num = 0;

        String str = null;

        try {

            j = 18/i;

            num = nums[3];

            num = str.length();

        } catch(ArithmeticException e) {

            System.out.println("Cannot divide by zero");

        } catch(ArrayIndexOutOfBoundsException e) {

            System.out.println("Array index specified is outside the length of the array");

        } catch(NullPointerException e) {

            System.out.println("You cannot perform that operation on a null value");

        } catch(Exception e) {

            System.out.println("Something else went wrong");

        } catch(Throwable e) {

            System.out.println("Something even more mysterious went wrong");

        }

        // Without the try - catch these statements would not execute

        System.out.println(j);

    }

}

### **“Throw” Keyword**

Sometimes we may to manually create an exception even if that particular kind of exceptions has not occurred at runtime. To do this we use the “throw” keyword.

public class Exceptions {

    public static void main(String[] args) {

        int i = 20;

        int j = 0;

        try {

            j = 18/i;

            if(i > 18) {

                throw new ArithmeticException("integer division does not allow for decimals. It returns 0 as the quotient if the divisor is greater than the dividend");

            }

        } catch(ArithmeticException e) {

            if(i > 18) {

                j = 18/1;

                System.out.println(e);

            } else {

                System.out.println("Cannot divide by zero");

            }

        }

        // Without the try - catch these statements would not execute

        System.out.println(j);

    }

}

### **Custom Exceptions**

Sometimes we may want to create our own exceptions. For example to handle the case above but with own custom exception class:

// Create the class and extend the "Exception" or "RuntimeException" class

class MyCustomException extends Exception {

    // Create a constructor to accept the error message

    MyCustomException(String errorMessage) {

        // Pass this error message to the parent class that then handles it.

        super(errorMessage);

    }

}

public class Exceptions {

    public static void main(String[] args) {

        int i = 20;

        int j = 0;

        try {

            j = 18/i;

            if(i > 18) {

                // Create a new object of the custom class and pass the message to the parameterised constructor

                throw new MyCustomException("integer division does not allow for decimals. It returns 0 as the quotient if the divisor is greater than the dividend");

            }

        } catch(ArithmeticException e) {

            System.out.println("Cannot divide by zero");

        }

        // Catch the error thrown

        catch(MyCustomException e) {

            System.out.println(e);

        }

        // Without the try - catch these statements would not execute

        System.out.println(j);

    }

}

### **“Throws” Keyword**

This keyword is used to pass the responsibility of handling the runtime exception created to the function higher in the stack, i.e. the function calling the one where the exception has occurred.

class Calc {

    // Throws the exception instead of surrounding "int res = i / j;" with a try catch block

    public static int divide(int i, int j) throws ArithmeticException {

        int res = i / j;

        return res;

    }

    // Throws the exception instead of surrounding "int num = nums[4];" with a try catch block

    public static int getArrayElement(int nums[]) throws ArrayIndexOutOfBoundsException {

        int num = nums[4];

        return num;

    }

}

public class ThrowsKeyword {

    // It is discouraged to use "throws" exceptions from the main method because this is then handled by the JVM which will stop execution

    public static void main(String[] args) {

        int i = 5;

        int j = 0;

        int nums[] = {1,2,3,4};

        try {

            int quotient = Calc.divide(i,j);

            int elem = Calc.getArrayElement(nums);

        }

        // Handling the exceptions thrown from the functions "divide()" and "getArrayElement"

        catch (ArithmeticException e) {

            System.out.println(e);

            e.printStackTrace();            // Displays the hierarchy of methods involved.

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println(e);

        }

        System.out.println("After Error Handling");

    }

}

## **Accepting User Input**

There are three options of accepting user input from the terminal:

* Using “System.in.read()”
  + returns the ASCII code for the character you have entered.
  + One character at a time.
* Using “BufferReader” and “Integer.parseInt(bf.readLine())”
  + this is a resource which needs to be closed as a resource can only be used by one person at a time.
  + Accepts multiple character inputs.
  + “bf.readLine()” returns a string and hence the “Integer.parseInt()”
* Using “Scanner”
  + Introduced in java 1.5
  + Accepts multiple character inputs.
  + Accepts multiple input sources, terminal, files etc.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Scanner;

public class UserInput {

    public static void main(String[] args) {

        System.out.println("Enter a number: ");

        try {

            // Using System.in

            int num = System.in.read();          // Returns the ASCII code the input

            System.out.println(num - 48);

            // Using BufferedReader

            InputStreamReader in = new InputStreamReader(System.in);

            BufferedReader bf = new BufferedReader(in);

            // num = bf.read();             // Also returns the ASCII

            num = Integer.parseInt(bf.readLine());

            System.out.println(num);

            bf.close();

            // Using Scanner

            Scanner sc = new Scanner(System.in);

            num = sc.nextInt();

            System.out.println(num);

            sc.close();

        } catch(IOException e) {

            System.out.println(e);

            e.printStackTrace();

        }

    }

}

## **Try … Finally/ Try With Resources**

“finally” is a block that is used in conjunction with “try” when you want the statements within to be executed regardless of whether an exception from the “try” block occurs or not.

It is mainly used to close resources such as “BufferedReader”.

An alternative notation of the same is to declare the resources inside opening and closing parentheses on the same line as the “try” statement, e.g. try(…..declare resource here…..) {…}. Here, once all the statements in the try block are executed, the resource will be closed automatically.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Scanner;

public class UserInput {

    public static void main(String[] args) throws IOException {

        System.out.println("Enter a number: ");

        // Using "finally"

        BufferedReader bf = null;

        try {

            InputStreamReader in = new InputStreamReader(System.in);

            bf = new BufferedReader(in);

            // num = bf.read();             // Also returns the ASCII

            int num = Integer.parseInt(bf.readLine());

            System.out.println(num);

        } catch(IOException e) {

            System.out.println(e);

            e.printStackTrace();

        } finally {

            bf.close();

        }

        // Using "try()"/ "try with resources"

        try(BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));) {

            int num = Integer.parseInt(bf.readLine());

            System.out.println(num);

        } catch(IOException e) {

            System.out.println(e);

            e.printStackTrace();

        }

    }

}

## **Threads**

These are the smallest units of software that the Operating System can work with. For example, in a code editor, one thread could be responsible for intellisense, another for error reporting, and another for displaying the code being typed out. Threads, by the same software, share resources and can be run in parallel (concurrently).

There are several methods of creating threads in java:

### **Extend “Thread” class**

// Turns the class intances into threads rather than normal objects

class A extends Thread{

    // Default first method called when a thread is started

    public void run() {

        for(int i=0; i<100; i++){

            System.out.println("Hi");

            try {

                // Tells/Suggests to the scheduler to wait before continuing with execution

                Thread.sleep(10);

            } catch (InterruptedException e) {

                // TODO Auto-generated catch block

                e.printStackTrace();

            }

        }

    }

}

// Turns the class intances into threads rather than normal objects

class B extends Thread{

    // Default first method called when a thread is started

    public void run() {

        for(int i=0; i<100; i++){

            System.out.println("Hello");

            try {

                // Tells/Suggests to the scheduler to wait before continuing with execution

                Thread.sleep(10);

            } catch (InterruptedException e) {

                // TODO Auto-generated catch block

                e.printStackTrace();

            }

        }

    }

}

public class Threads {

    public static void main(String[] args) {

        A obj1 = new A();

        B obj2 = new B();

        // Starts the threads

// obj1.setPriority(Thread.MAX\_PRIORITY);

        System.out.println(obj1.getPriority());

        obj1.start();

        // Suggests that the scheduler waits before continuing

        try {

            Thread.sleep(10);

        } catch (InterruptedException e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

// obj2.setPriority(Thread.MAX\_PRIORITY);

        System.out.println(obj2.getPriority());

        obj2.start();

    }

}

// Note: Different schedulers use different algorithms and so the "Thread.sleep()" is a suggestion whose implementation varies from OS to OS.

// All a programmer can do is try to optimize the running of these threads but not determine their running

// This optimisation, besides using "Thread.sleep()", can also be done by manually setting the priority of the threads where 10 is the maximum and 1 is the minimum

### **Implementing “Runnable” Interface**

This is better than extending “Thread” class because the former restricts the child class from extending another class. A class can implement an interface and extend a class at the same time.

// The parent class being inherited by "B"

class A {

    public static void show() {

        System.out.println("Hi");

    }

}

// The Thread class that is also extending the class "A" and implementing the "Runnable" interface

class B extends A implements Runnable {

    public void run() {

        for(int i = 0; i<5; i++) {

            show();

            try {

                Thread.sleep(10);

            } catch (InterruptedException e) {

                // TODO Auto-generated catch block

                e.printStackTrace();

            }

        }

    }

}

// The thread class that is implementing the "Runnable" class but not extending A

class C implements Runnable {

    public void run() {

        for(int i = 0; i<5; i++) {

            System.out.println("Hello");

            try {

                Thread.sleep(10);

            } catch (InterruptedException e) {

                // TODO Auto-generated catch block

                e.printStackTrace();

            }

        }

    }

}

public class Threads {

    public static void main(String[] args) {

        // Dynamic Method dispatching, i.e. Creating a "Runnable" interface reference but the objects are of different types.

        Runnable obj1 = new B();

        Runnable obj2 = new C();

        // Creating the threads but passing the objects into the constructor because the "Runnable" interface does not have a "run()" method implementation.

        Thread t1 = new Thread(obj1);

        Thread t2 = new Thread(obj2);

        // Starts the threads

        t1.start();

        // Suggests that the scheduler waits before continuing

        try {

            Thread.sleep(10);

        } catch (InterruptedException e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

        t2.start();

    }

}

### **Using lambda expressions (Anonymous Inner Classes)**

”Runnable” is a Functional Interface, i.e., a method with one method declaration

public class Threads {

    public static void main(String[] args) {

        // Using anonymous inner classes

        // Runnable obj1 = new Runnable() {

        //     public void run() {

        //         for(int i = 0; i<5; i++) {

        //             System.out.println("Hi");

        //             try {

        //                 Thread.sleep(10);

        //             } catch (InterruptedException e) {

        //                 // TODO Auto-generated catch block

        //                 e.printStackTrace();

        //             }

        //         }

        //     }

        // };

        // Runnable obj2 = new Runnable() {

        //     public void run() {

        //         for(int i = 0; i<5; i++) {

        //             System.out.println("Hello");

        //             try {

        //                 Thread.sleep(10);

        //             } catch (InterruptedException e) {

        //                 // TODO Auto-generated catch block

        //                 e.printStackTrace();

        //             }

        //         }

        //     }

        // };

        // Using lambda expressions to simply the above

        Runnable obj1 = () -> {

            for(int i = 0; i<5; i++) {

                System.out.println("Hi");

                try {

                    Thread.sleep(10);

                } catch (InterruptedException e) {

                    // TODO Auto-generated catch block

                    e.printStackTrace();

                }

            }

        };

        Runnable obj2 = () -> {

            for(int i = 0; i<5; i++) {

                System.out.println("Hello");

                try {

                    Thread.sleep(10);

                } catch (InterruptedException e) {

                    // TODO Auto-generated catch block

                    e.printStackTrace();

                }

            }

        };

        // Creating the threads but passing the objects into the constructor because the "Runnable" interface does not have a "run()" method implementation.

        Thread t1 = new Thread(obj1);

        Thread t2 = new Thread(obj2);

        // Starts the threads

        t1.start();

        // // Suggests that the scheduler waits before continuing

        // try {

        //     Thread.sleep(10);

        // } catch (InterruptedException e) {

        //     // TODO Auto-generated catch block

        //     e.printStackTrace();

        // }

        t2.start();

    }

}

## **Race Conditions**

A race condition refers to a situation where more than one thread access the same resource, say a variable, at the same time and modify the same leading to inconsistencies. For example, having a bank account have money withdrawn from it at the exact same time by different ATMS.

This can be solved by:

* Declaring the methods being called by the threads as “synchronized” so that only one thread can call them at a time.
* Calling the join() method on the individual threads from the parent method starting the threads.

class Counter {

    int counter;

    // This makes the method thread-safe

    public synchronized void increment() {

        counter++;

    }

}

public class RaceConditions {

    public static void main(String args[]) {

        // The shared resource

        Counter c = new Counter();

        // The different threads that are calling the shared resource

        Runnable obj1 = () -> {

            for(int i = 0; i<5000; i++) {

                c.increment();

            }

        };

        Runnable obj2 = () -> {

            for(int i = 0; i<5000; i++) {

                c.increment();

            }

        };

        Thread t1 = new Thread(obj1);

        Thread t2 = new Thread(obj2);

        t1.start();

        t2.start();

        try {

            // Instruct the thread-starting method (in this case, the "main" method), to wait for the completion of the tasks by the threads before continuing.

            t1.join();

            t2.join();

        } catch (InterruptedException e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

        System.out.println(c.counter);

    }

}

## **Thread States**

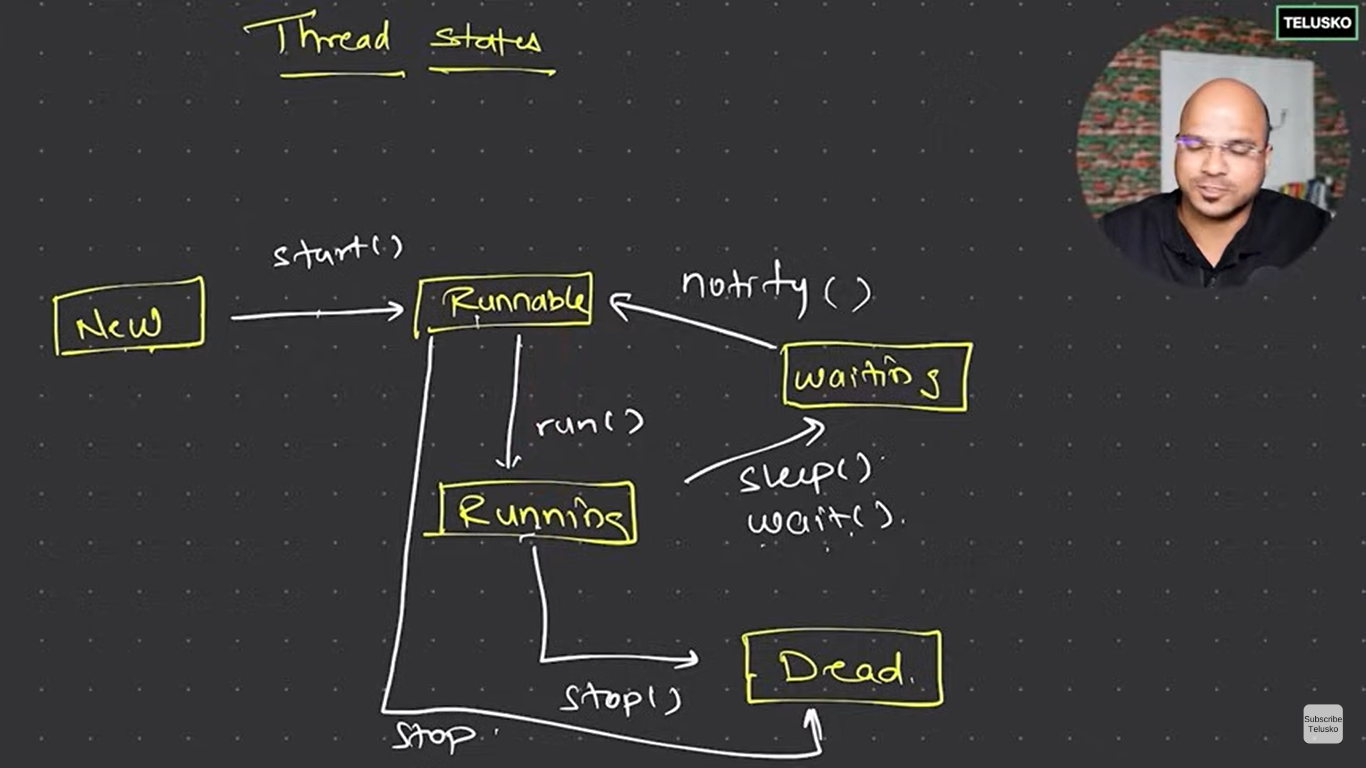


Figure 2: Thread States

## **collection API/ Collection/ Collections**

collection API/ collection framework – concept

Collection – Interface

Collections – Class

Arrays are a great tool in programming but they have some limitations:

* They are fixed in size
* They require a lot of work to turn into data structures such as stacks, queues etc.

As such, java provides some in built ways work with these more complex data structures, an API of sorts.

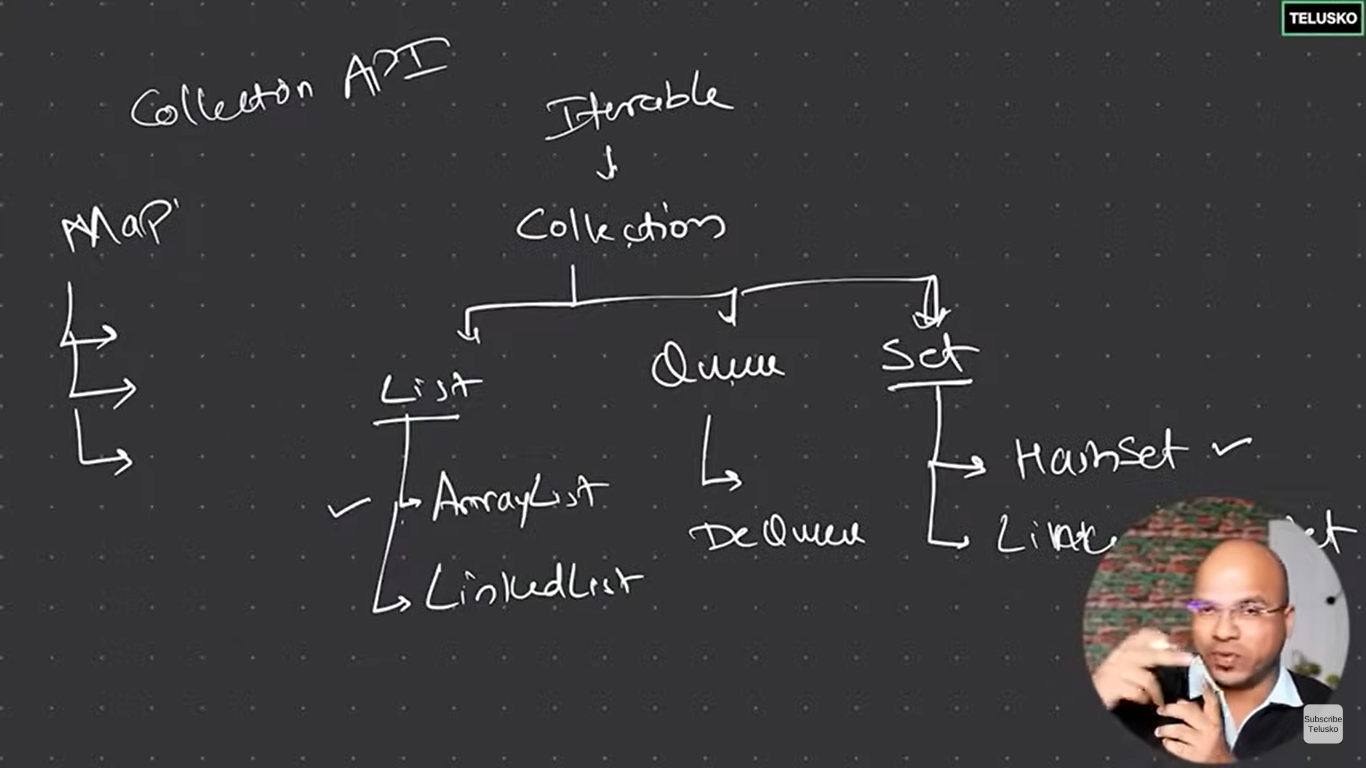


Figure 3: Collections Summary

**Collection Interface**

Belongs to the “java.util” package.

It has other children interfaces that extend the same, e.g. List, Queue and Set. Each of these has its own classes that implement the same, and these classes are what can be instantiated to create the complex data structures. These data structures work with objects and not primitives, so wrapper classes such as Integer are used.

Alongside this Collection Interface in the collection API concept, there is also a Map interface that has its own class implementations.

Collection’ class implementation’ instances can also be printed directly, without having to use a loop. However, because Collection interface is a child of another interface called “Iterable”, we can use this Iterable to loop through the collections.

These collection API interfaces provide a feature known as **generics** that is used to make the class instances type safe, i.e., allowing you to restrict the type of values a collection can take. Otherwise, collections can accept values of different types.

Collection interface does not support indexing but its child interface, List, does. When in need of this indexing functionality, we should therefore declare the type of the reference variables to the collection as “List”.

### **Collection examples:**

import java.util.ArrayList;

import java.util.Collection;

import java.util.HashSet;

import java.util.Iterator;

import java.util.List;

import java.util.Set;

import java.util.TreeSet;

public class Collections {

    public static void main(String[] args) {

// Recall: Collection is an interface, and NOT a functional interface.

        // Collection nums = new Collection<E>() {};       The use of anonymous inner class is unwise because the "Collection" interface has too many method declarations that would, otherwise, need to be implemented, failure to which it would require the class implementation be declared abstract, which in turn, cannot be instantiated.

        Collection<Integer> numsCollectionArrayList = new ArrayList<Integer>();    // The <Integer> are called generics and are used to make the collections type-safe.

        numsCollectionArrayList.add(6);

        numsCollectionArrayList.add(5);

        numsCollectionArrayList.add(4);

        System.out.println("Collection");

        System.out.println(numsCollectionArrayList);

        // List

            // supports indexing.

            // Allow for duplicate values.

        List<Integer> numsListArrayList = new ArrayList<Integer>();

        numsListArrayList.add(6);

        numsListArrayList.add(5);

        numsListArrayList.add(4);

        numsListArrayList.add(4);

        numsListArrayList.add(3);

        numsListArrayList.add(6);

        numsListArrayList.set(3, 1);

        System.out.println("ArrayList");

        System.out.println(numsListArrayList.get(0));

        System.out.println(numsListArrayList.indexOf(4));

        System.out.println(numsListArrayList);

        // Set

            // Does not support indexing.

            // Does not allow duplicate values.

            // Does not sort the elements in any particular order.

        Set<Integer> numsSetHashSet = new HashSet<Integer>();

        numsSetHashSet.add(86);

        numsSetHashSet.add(55);

        numsSetHashSet.add(32);

        numsSetHashSet.add(86);     // Omitted from the collection because it is a duplicate

        System.out.println("HashSet");

        System.out.println(numsSetHashSet);

        for(int n : numsSetHashSet) {

            System.out.println(n);

        }

        // TreeSet - similar to HashSet but this one sorted the elements in ascending order

        Set<Integer> numsSetTreeSet = new TreeSet<Integer>();

        numsSetTreeSet.add(86);

        numsSetTreeSet.add(55);

        numsSetTreeSet.add(32);

        numsSetTreeSet.add(44);

        System.out.println("TreeSet");

        System.out.println(numsSetTreeSet);

        for(int n : numsSetTreeSet) {

            System.out.println(n);

        }

        // Iterator - Type of Interface that is returned when the "iterator()" method is called by a child of the "Interable" interface.

        // "Interable" is the top most interface above Collection for iteration purposes.

        Set<Integer> numsIteratorTreeSet = new TreeSet<Integer>();

        numsIteratorTreeSet.add(86);

        numsIteratorTreeSet.add(55);

        numsIteratorTreeSet.add(32);

        numsIteratorTreeSet.add(44);

        System.out.println("Iterator");

        Iterator<Integer> values = numsIteratorTreeSet.iterator();

        while(values.hasNext()) {       // hasNext() returns a boolean if there is still another element in the collection thatis yet to be looped over.

            System.out.println(values.next());

        }

    }

}

### **Maps examples:**

import java.util.HashMap;

import java.util.Hashtable;

import java.util.Map;

public class Maps {

    public static void main(String[] args) {

        // Maps

            // Take key-value pairs, where the key is an actual label, rather than an index

            // The keys are unique

            // The put() method overwrites the value for the existing element with the key specified, otherwise it creates a new element.

            // The values do not need to be unique, they can be repeated

            // In summary, the keys are a set, and the values are a List

        // HashMap

            // Does not sort the elements

            // Not synchronized by default, as opposed to HashTables, when working with threads, so this has to be done externally, by declaring the methods making use of the hashMap as "synchronized"

        Map<String, Double> students = new HashMap<>();

        students.put("Daniel", 64.23);

        students.put("Blair", 71.23);

        students.put("Charles", 65.23);

        students.put("Agnes", 64.23);

        students.put("Daniel", 64.57);      // Overwriting the value for "Daniel" from "64.23" to "64.57"

        System.out.println("HashMap");

        System.out.println(students);

        System.out.println(students.get("Daniel"));     // Return the value at the key specified

        System.out.println(students.keySet());          // Returns the set of keys in the map

        System.out.println(students.values());          // Returns the list of values in the map

        // To loop through a map, we have to you the set of keys because there are no indices

        for(String key: students.keySet()) {

            System.out.println(key + ": " + students.get(key));

        }

        // HashTable

            //  Sorts the elements in ascending order based on the key

            //  Synchronized by default when working with threads. Otherwise it works the same way as the HashMap

        Map<String, Integer> players = new Hashtable<>();

        players.put("Lin Dan", 5);

        players.put("Lee Cong Wei", 0);

        players.put("Chen Long", 2);

        players.put("Victor Axelsen", 3);

        players.put("Kento Momota", 2);      // Overwriting the value for "Daniel" from "64.23" to "64.57"

        System.out.println("HashTable");

        System.out.println(players);

        System.out.println(players.get("Kento Momota"));     // Return the value at the key specified

        System.out.println(players.keySet());          // Returns the set of keys in the map

        System.out.println(players.values());          // Returns the list of values in the map

        // To loop through a map, we have to you the set of keys because there are no indices

        for(String key: players.keySet()) {

            System.out.println(key + ": " + players.get(key));

        }

    }

}

## **Sorting Collections Using Custom Logic**

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class CustomSorting {

    public static void main(String[] args) {

        List<Integer> nums = new ArrayList<>();

        nums.add(23);

        nums.add(64);

        nums.add(55);

        nums.add(16);

        nums.add(37);

        //  Sorting functionality provided by the "Collections" class

        Collections.sort(nums);     // Sort the list in ascending order

        System.out.println(nums);

        // Creating a custom sorting algorthm using the "Comparator" interface and extending it to define this algorithm in the "compare" method

        Comparator<Integer> com = new Comparator<Integer>() {

            public int compare(Integer i, Integer j) {

                if(i%10 > j%10) {

                    return 1;           // Returning 1 means swap the elements

                } else {

                    return -1;          // Returning -1 means do not swap

                }

            }

        };

        // Passing the collection and the new sorting algorithm to the "sort()" method of the "Collections" class

        Collections.sort(nums, com);     // Sort the list in ascending order based on the last digit of the elements

        System.out.println(nums);

        List<String> strs = new ArrayList<>();

        strs.add("Joseph");

        strs.add("Aquillas");

        strs.add("Peter");

        strs.add("Bartholomew");

        strs.add("Mary");

        Collections.sort(strs);     // Sort the list in ascending order

        System.out.println(strs);

        // Creating a custom sorting algorthm using the "Comparator" interface and extending it to define this algorithm in the "compare" method

        Comparator<String> comStr = new Comparator<String>() {

            public int compare(String i, String j) {

                if(i.length() > j.length()) {

                    return 1;           // Returning 1 means swap the elements

                } else {

                    return -1;          // Returning -1 means do not swap

                }

            }

        };

        // Passing the collection and the new sorting algorithm to the "sort()" method of the "Collections" class

        Collections.sort(strs, comStr);     // Sort the list in ascending order based on the lengths of the strings in the collection

        System.out.println(strs);

    }

}

If you want to compare objects of a custom class held within a collection, you can use a comparator, as above, or:

* Make the class implement the “Comparable” interface that declares a method “compareTo()”
* Override this “compareTo()” method within your class so that when calling the “sort()” method on the collection holding the class objects, you do need to pass a comparator class implementation.
* Use lambda expressions to simplify the regular comparator because the “Comparable” interface is a functional interface (having one method declaration (compareTo()))

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

class Student implements Comparable<Student>{

    int age;

    String name;

    public Student(int age, String name) {

        this.age = age;

        this.name = name;

    }

    public String toString() {

        return "Student [age=" + age + ", name=" + name + "]";

    }

    public int compareTo(Student that) {

        if(this.age > that.age) {

            return 1;           // Returning 1 means swap the elements

        } else {

            return -1;          // Returning -1 means do not swap

        }

    }

}

public class CustomSorting {

    public static void main(String[] args) {

        // Sorting Objects

        List<Student> students = new ArrayList<>();

        students.add(new Student(32, "Michael"));

        students.add(new Student(23, "Karongo"));

        students.add(new Student(28, "Maina"));

        students.add(new Student(37, "Njeri"));

        students.add(new Student(19, "Charles"));

        // 1. Sorting Using a regular comparator

        Comparator<Student> comStudents = new Comparator<Student>() {

            public int compare(Student i, Student j) {

                if(i.age > j.age) {

                    return 1;           // Returning 1 means swap the elements

                } else {

                    return -1;          // Returning -1 means do not swap

                }

            }

        };

        Collections.sort(students, comStudents);

        // 2. Sorting using lambda expressions

        Comparator<Student> comStudentsLambda = (i,j) -> i.age > j.age ? 1 : -1;

        Collections.sort(students, comStudentsLambda);

        // 3. Using the overridden "compareTo()" method from the "Comparable" interface

        Collections.sort(students);

        // Displaying the collection using the enhanced for loop

        for(Student s: students) {

            System.out.println(s);

        }

    }

}

## **ForEach and Consumer**

Besides the regular and enhanced for loops, java, since version 1.8, provided a new functionality called “forEach” that loops through a list giving you one element at a time.

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.function.Consumer;

public class ForEachAndConsumer {

    public static void main(String[] args) {

        List<Integer> nums = new ArrayList<>();

        nums.add(35);

        nums.add(13);

        nums.add(24);

        nums.add(35);

        nums.add(91);

        List<Integer> nums1 = Arrays.asList(35, 13, 24, 35, 91);    // Equivalent to the above

        System.out.println(nums);

        // System.out.println(nums1);

        // To loop through either of the lists the following functionalities exist:

        // 1. Regular for loop

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

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

        }

        // 2. Enhanced for loop

        for(Integer i: nums) {

            System.out.println(i);

        }

        // 3. forEach loop

            // Consumer

            Consumer<Integer> consumer = new Consumer<Integer>() {

                public void accept(Integer e) {

                    System.out.println(e);

                }

            };

            // forEach accepts a consumer class implementation as a parameter

            nums.forEach(consumer);

            // Consumer lambda expression - Consumer is a functional interface

            Consumer<Integer> consumerLambda = e -> System.out.println(e);

            nums.forEach(consumerLambda);

            nums.forEach(e -> System.out.println(e));       // Replacing the Consumer parameter with the body of the class implementation "e -> System.out.println(e)"

    }

}

## **Stream API**

Consider a list. This list can be turned into a Stream class implementation object (Stream is an interface) using the stream() method.

* Changes can then made to this stream without affecting the original list.
* Once used, a Stream cannot be reused.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class Streams {

    public static void main(String[] args) {

        List<Integer> nums = Arrays.asList(8,4,5,7,1);

        // Stream<Integer> s1 = nums.stream();         // Creating a stream from the list

        // s1.forEach(e -> System.out.println(e));     // Allows for the same operations as "List" itself

        // s1.forEach(e -> System.out.println(e));     // This will not work as it will return an exception "stream has already been operated upon or closed"

        // methods executed on streams return a new stream destroying the old one.

        // Stream<Integer> s2 = s1.filter(n -> n%2 == 0);      // Returns the even numbers from the stream

        // Stream<Integer> s3 = s2.map(n -> n\*2);      // Doubles the numbers in the stream

        // int result = s3.reduce(0, (c,e) -> c+e);      // Add all the elements in the stream and returns a single value

        // System.out.println(result);

        // The above can be simplified into one statement as:

        int result = nums.stream()

                        .filter(e -> e%2 == 0)

                        .map(e -> e\*2)

                        .reduce(0, (c,e) -> c+e);

        System.out.println(result);

        Stream<Integer> sortedStream = nums.stream()

                                           .map(e -> e\*2)

                                           .sorted();

        sortedStream.forEach(n -> System.out.println(n));

    }

}

### **Filter Method**

This is used to separate data that meets a specified condition from a stream, returning a new stream.

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

import java.util.stream.Stream;

public class FilterMethod {

    public static void main(String[] args) {

        List<Integer> nums = Arrays.asList(1,2,3,4,5,6);

        Stream<Integer> s1 = nums.stream();

        // filter method requires a parameter of type "Predicate", which is a functional interface with one method declaration, "test()"

        Predicate<Integer> p = new Predicate<Integer>() {

            public boolean test(Integer n) {

                return n%2==0;

            }

        };

        // This can be simplified into a lambda expression

        Predicate<Integer> p1 = n -> n%2==0;

        // Stream<Integer> s2 = s1.filter(p);

        // Stream<Integer> s2 = s1.filter(p1);          // Equivalent of the above

        Stream<Integer> s2 = s1.filter(n -> n%2==0);    // Equivalent of the above

        s2.forEach(n -> System.out.println(n));

    }

}

### **Map Method**

This one performs an arbitrary function on the elements in a stream, returning a new stream of the modified values.

import java.util.Arrays;

import java.util.List;

import java.util.function.Function;

import java.util.function.Predicate;

import java.util.stream.Stream;

public class MapMethod {

    public static void main(String[] args) {

        List<Integer> nums = Arrays.asList(1,2,3,4,5,6);

        Stream<Integer> s1 = nums.stream();

        // map method requires a parameter of type "Function", which is a functional interface with one method declaration, "apply()"

        Function<Integer, Integer> f = new Function<Integer, Integer>() {

            public Integer apply(Integer n) {

                return n\*2;

            }

        };

        // This can be simplified into a lambda expression

        Function<Integer, Integer> f1 = n -> n\*2;

        // Stream<Integer> s2 = s1.map(f);

        // Stream<Integer> s2 = s1.map(f1);          // Equivalent of the above

        Stream<Integer> s2 = s1.map(n -> n\*2);    // Equivalent of the above

        s2.forEach(n -> System.out.println(n));

    }

}

### **Reduce Method**

This one performs an arbitrary function on the elements in a stream, returning an aggregate value of the function carried out on all the elements of the stream.

public class ReduceMethod {

    public static void main(String[] args) {

        List<Integer> nums = Arrays.asList(1,2,3,4,5,6);

        Stream<Integer> s = nums.stream();

        // reduce method takes two parameters, an initial value and a function

        int result = s.reduce(0, (c,e) -> c+e);

        System.out.println(result);

    }

}

## **Sealed Classes**

Recall that declaring a class as final makes the class uninheritable. We may want to restrict a class such that it is uninheritable but having some classes that can inherit it. For example, you may have a class Computer that can only be restricted by the classes Laptop, Desktop and Mobile but no other class. To do this, we use the “sealed” keyword.

// Note

    // Once a class is sealed, it has to specify the classes that it permits

    // Classes included in the "permits" section of a sealed must extend the sealed parent class

    // Classes included in the "permits" section of a sealed must be declared as either sealed, non-sealed or final

    // The grand-parent sealed class can extend other classes

sealed class A extends Thread permits B,C,D {          // Class E is not permitted

}

non-sealed class B extends A {          // class B can be extended by any class

}

final class C extends A {               // Class C cannot be extended by any other class

}

sealed class D extends A permits E {

}

non-sealed class E extends D {

}

public class SealedClasses extends E {

}

## **Sealed Interfaces**

Interfaces can also be declared as sealed. They also have to declare the classes and interfaces that can implement/extend the same.

sealed interface A permits B, C {

}

non-sealed interface B  extends A {

}

non-sealed class C implements A {

}

// class D implements A, B {        // Class D is not permitted to implement interface A

// }

class D implements B {        // Class D is can implement interface B because it non-sealed

}

public class SealedInterfaces {

}

## **Record Classes**

This were introduced to provide a shorter, less verbose method for creating Data Carrier classes, i.e., classes with no functional methods. Classes that only have properties. Used during data storage/transfer between clients and servers.

// The class below is a data carrier class

class Alien {

    // The properties are encapsulated and therefore require getters and setters(constructor)

    private final int age;

    private final String name;

    public Alien(int age, String name) {

        this.age = age;

        this.name = name;

    }

    public int getAge() {

        return age;

    }

    public String getName() {

        return name;

    }

    // This is meant to return a meaningful output when the object is rendered out by: "System.out.println(a1);"

    @Override

    public String toString() {

        return "Alien [age=" + age + ", name=" + name + "]";

    }

    @Override

    public int hashCode() {

        final int prime = 31;

        int result = 1;

        result = prime \* result + age;

        result = prime \* result + ((name == null) ? 0 : name.hashCode());

        return result;

    }

    // Meant to change the equality comparison of the class instances to only look at the values rather than the "hashcode" as well.

    @Override

    public boolean equals(Object obj) {

        if (this == obj)

            return true;

        if (obj == null)

            return false;

        if (getClass() != obj.getClass())

            return false;

        Alien other = (Alien) obj;

        if (age != other.age)

            return false;

        if (name == null) {

            if (other.name != null)

                return false;

        } else if (!name.equals(other.name))

            return false;

        return true;

    }

}

// The whole of the above can be simplified into the following, a RECORD CLASS

record AlienSimplied (int age, String name) implements Cloneable {}

// This creates everything as above including the constructor. This type of constructor is called a canonical constructor.

// Note: "record" cannot extend any other class because it already extends a class called "Record". It can however implement interfaces.

public class RecordClasses {

    public static void main(String[] args) {

        Alien a1 = new Alien(23, "Karongo");

        Alien a2 = new Alien(23, "Karongo");

        System.out.println(a1.equals(a2));

        System.out.println(a1.getAge());

        System.out.println(a1.getName());

        System.out.println(a1);

        AlienSimplied aS1 = new AlienSimplied(23, "Karongo");

        AlienSimplied aS2 = new AlienSimplied(23, "Karongo");

        System.out.println(aS1.equals(aS2));

        System.out.println(aS1.age());          // The record class's getters are as these, they do not contain "get". There are no setters, besides the constructor, because the variables declared are final.

        System.out.println(aS1.name());

        System.out.println(aS1);

    }

}

# **JUNIT**

This is a unit-testing framework for java. To demonstrate this:

* Create a maven project. Maven is a project building tool, alongside “Gradle”, that is used to configure the project making it easier to manage the same from start (design) to finish (building). Add the JUNIT dependency to pom.xml and save the file. This will add the dependencies to the project in a folder called “Maven Dependencies”.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.karongo.learning</groupId>

  <artifactId>JUnitIntroduction-01</artifactId>

  <version>0.0.1-SNAPSHOT</version>

  <dependencies>

    <!-- https://mvnrepository.com/artifact/junit/junit -->

    <dependency>

        <groupId>junit</groupId>

        <artifactId>junit</artifactId>

        <version>4.13.2</version>

        <scope>test</scope> <!-- This dependency will not be included in the build when deploying the project -->

    </dependency>

  </dependencies>

</project>

* Within “src/main” directory, create a class with some function.

package com.introduction.JUnit;

public class StringFunctions {

    public static String reverseString(String inputString) {

        char[] inputCharArray = inputString.toCharArray();

        char[] reversedCharArray = new char[inputString.length()];

        for(int i=inputString.length()-1, j=0; i>=0; i--, j++) {

            reversedCharArray[j] = inputCharArray[i];

        }

        return new String(reversedCharArray);

    }

}

* Within “src/test” directory, create a class that describes a test case then run it.

package com.introduction.JUnit;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

class StringFunctionsTest {

    @Test

// Necessary to make the respective be run as a test case.

// In JUnit 4, this annotation required that the method be declared "public".

// In JUnit 5, this method can be default, protected or public

    void testReverseString() {  // This method checks for failures, not success, such that it only fails if there is a failure and every thing else is considered a success

//      fail("Not yet implemented");        - Invoked to explictly fail the test

//      The third parameter is the error message in the case of a failed assertion

        assertEquals("leinaD", StringFunctions.reverseString("Daniel"));

        assertEquals("avaJ", StringFunctions.reverseString("Java"));

    }

@Test

void testReverseString\_Multiple() {

//      fail("Not yet implemented");

    assertEquals("eman ym si leinaD", StringFunctions.reverseString("Daniel is my name"));

}

}

## **JUNIT5**

This is also a testing framework that is more recent but that is not the same as JUNIT4 (and its predecessors) + a few features. JUNIT5 is a completely different thing.

It provides some APIs that can be used to interact with the testing engine/platforms. These APIS include:

* Jupiter – main API used by JUNIT 5. It contains all the methods, annotations etc. necessary for making the assertions.
* Vintage – JUNIT 5 does not support backward compatibility to the JUNIT 4 series. As such this API is used to run all the test cases that were written in the previous frameworks.
* Extension – Used as an alternative when we do not want to use Jupiter.

### **Junit5 Features**

1. Supplier Interface – For efficiently handling the error messages in the event of a failed assertion (Lazy Evaluation)

package com.introduction.JUnit;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

class StringFunctionsTest {

    @Test

    // Using the Supplier interface to pass the error message. The lambda expression is what denotes the Supplier interface

    void testReverseString\_with\_Supplier\_Interface() {

//      fail("Not yet implemented");

//      The the parameter is the error message in the case of a failed assertion

        assertEquals("leinaD", StringFunctions.reverseString("Daniel"), ()->"Method implemented wrongly");

        assertEquals("avaJ", StringFunctions.reverseString("Java"), ()->"Method implemented wrongly");

    }

}

1. assertNotEquals() – This works in the opposite of how “assertEquals()” works. It fails the test case when the actual and expected values are the same.

@Test

void testReverseString\_AssertNotEquals() {

    assertNotEquals("avaJ", StringFunctions.reverseString("Java"), ()->"Method implemented correctly");

}

1. assertTrue() –Checks if the assertion returns “true”, in which case it passes.

@Test

void testAssertTrue() {

    String str = "Java";

    assertTrue("Java".equals(str), ()->"Failing because result is false");

}

1. assertFalse() – Passes the test if the assertion returns false.

@Test

void testAssertFalse() {

    assertFalse("JUnit5".length()<5), ()->"Failing because result is true"););

}

1. assertArrayEquals() – Passes the test if the arrays being compared:
   * are of the same type
   * have the same number of elements
   * have the same elements
   * have the same order of elements

@Test

void testArrays() {

    int[] actual = {2,3,4,5};

    int[] expected = {5,3,4,2};

    Arrays.sort(expected);

    assertArrayEquals(actual, expected);

//      assertEquals(actual, expected);     Checks the reference variables and is therefore false

}

1. assertThrows() - This passes the test when there is an exception generated within the test case.

@Test

void testArray() {

//      In this case, the lambda expression is compulsory

    assertThrows(NullPointerException.class, ()-> ArraySorting.sortArray(null));

}

1. assertTimeout – This passes when the operation passed is executed within the duration specified

@Test

//  @Test(timeout=100)          // Equivalent for JUNIT 4

void testSortArrayManyTimes() {

//      In this case, the lambda expression is also compulsory

    int unsorted[] = {4,7,2,9,1};

    assertTimeout(Duration.ofMillis(100), ()-> ArraySorting.sortArrayManyTimes(unsorted));

}

### **Junit5 Annotations/ Test Case Life Cycle**

Every test method in a Junit test case is annotated with “@Test”. We may however manipulate the order with which we execute the methods in a test case using annotations such as:

1. @BeforeEach

This makes the method it is associated with run before each method annotated with “Test” such that if there are five “@Test” methods, this “@BeforeEach” annotated method will run five times as well. Used when there are shared operations between the various “@Test” methods.

1. @AfterEach

This makes the method it is associated with run before each method annotated with “Test” such that if there are five “@Test” methods, this “@BeforeEach” annotated method will run five 5 times as well. Also used when there are shared operations between the various “@Test” methods.

1. @BeforeAll

This makes the method it is associated with run before every other method within the test class. This method runs only once and must be a “static” method. For example, a method that establishes the database connection.

1. @AfterAll

This makes the method it is associated with run after every other method within the test class. This method runs only once and must also be a “static” method. For example, a method that closes the database connection.

Also, it is important to note that, by default, new class instances (objects) are created for every “@test” method. This can be verified by the creation of a custom constructor as described below. To change this behavior so that the class instance is created only once, and all the methods be run by that one class, we have to use the “@TestInstance” annotation as described below. Doing this alleviates the necessity to declare the methods under “@BeforeAll” and “@AfterAll” as static because the object will only be one for the class.

@TestInstance(TestInstance.Lifecycle.PER\_CLASS)

//@TestInstance(TestInstance.Lifecycle.PER\_METHOD)      -- This is the default

class JavaTest {

    JavaTest() {

        System.out.println("Test Class Instance being created");

    }

    @BeforeAll

//  @BeforeClass - JUnit 4 equivalent

//  static void beforeAll() {       -- -- Alleviated by the "TestInstance"

    void beforeAll() {

        System.out.println("Before all tests");

    }

    @AfterAll

//  @AftereClass - JUnit 4 equivalent

//  static void afterAll() {        -- Alleviated by the "TestInstance"

    void afterAll() {

        System.out.println("After all tests");

    }

    Java java;

    @BeforeEach

    void init() {

        java = new Java();

        System.out.println("Before Each Test");

    }

    @Test

    void testPurchaseCourseNonStatic() {

        java.purchaseCourseNonStatic();

    }

    @Test

    void testPurchaseCourseNonStaticB() {

        java.purchaseCourseNonStatic();

    }

    @AfterEach

    void destroy() {

        java = new Java();

        System.out.println("After Each Test");

    }

}

Test Class Instance being created

Before all tests

Before Each Test

Java course purchased

After Each Test

Before Each Test

Java course purchased

After Each Test

After all tests

## **Test-Driven Development**

This is where you build the test case first, invoking a dummy entity that will later be developed. This can be done by the project lead as he/she outlines a list functions that need to be implemented and then requires his team to create the said functions. He/she creates tests that those methods listed will need to pass so that when all is done, he/she/they can test the success of the methods.

# **DATA STRUCTURES AND ALGORITHMS**

**Abstract Data Type (ADT)** - This is a concept that implies having a conventional way of storing certain types of data, alongside the sort of operations you can perform on this data; but whose implementation varies from language to language.

**Time Complexity** – This is the measure of how the running time of an algorithm increases as the size of the input data increases. This is denoted using a standard known as the Big O Notation which specifies the relationship between the input data and the number of steps required to find the solution.

The higher the notation in the figure below, the more efficient the algorithm and therefore the more scalable the application (able to handle multiple users concurrently).

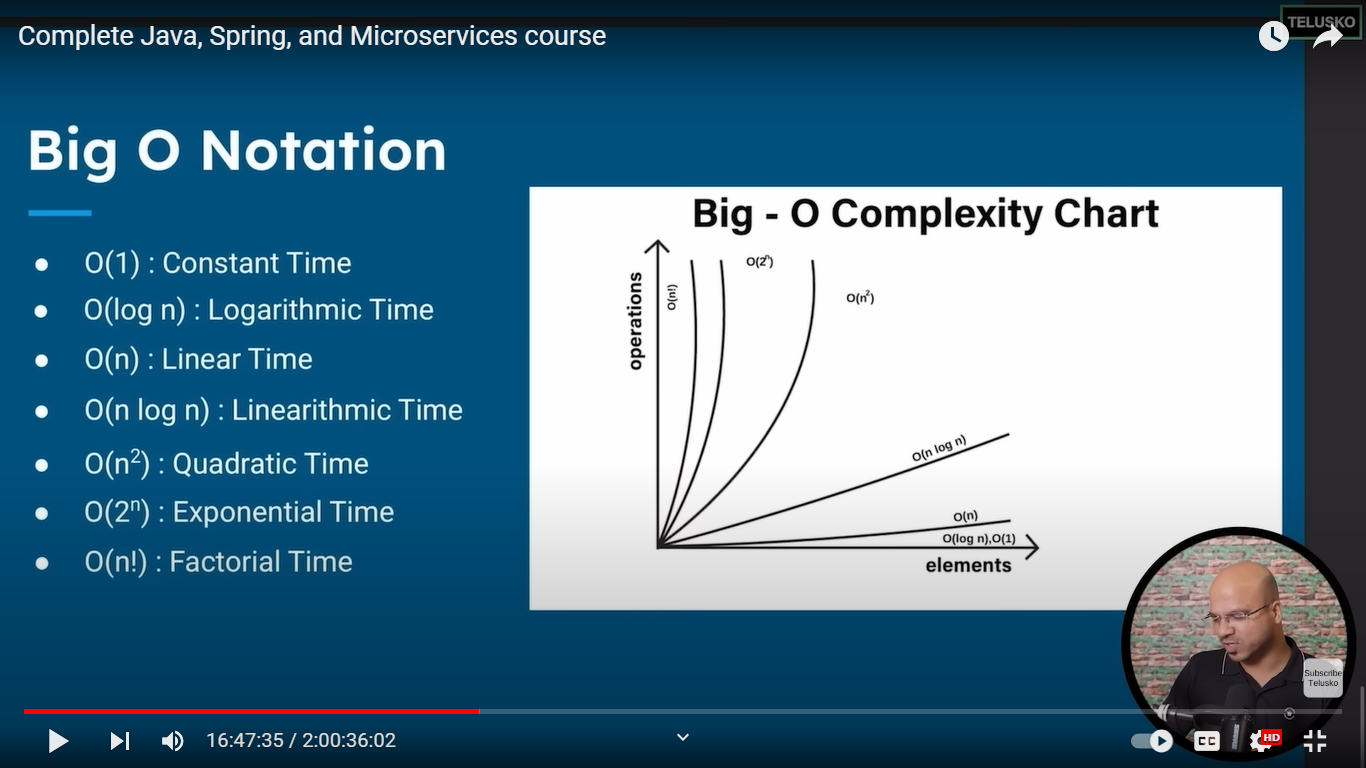


Figure 4: Chart of Big O notations. In increasing order of time complexity

## **Algorithms**

### **Searching Algorithms**

Any Abstract Data Type needs to allow for a read operation. This reading may involve searching for a particular element from within a larger collection of similar-typed values, an array of sorts. To do this searching, there are several search algorithms available:

#### **Linear Search Algorithm**

Elements within the array are looped one by one until the target value is found.

// Returns the index in the array passed of the value specified by looping through the array element by element. To determine the big O notation, we look at the worst case scenario (in this case, if the target value is the last value in the array/ does not exist). The more the numbers in the array, the more the number of steps/iterations necessary to find the index. E.g. If the array has 7 numbers, it will take 7 iterations to find the last value of the array. If it has 1000 numbers, it will take 1000 iterations to do the same.

    public static int linearSearch(int[] arr, int target) {

        int n = arr.length;

        for(int i = 0; i < n; i++) {

            if(arr[i] == target) {                  // O(n) where n is the size of the input array

                return i;

            }

        }

        return -1;

    }

#### **Binary Search Algorithm**

The SORTED array is split into two according to a midpoint that is updated during each iteration until the target value is found. This algorithm has a big O value of O(log n) such that the number of iterations required in the worst case scenario are still significantly less that those it would take to loop through the array one by one.

// Returns the index in the array of the value specified by looping through the array and splitting the array into two during each iteration according to a midpoint, then working with the side that the value being searched for is in, disregarding the rest.

    // To determine the big O notation, we look at the worst case scenario (in this case, if the target value is the last value in the array/ does not exist). The more the numbers in the array, the more the number of steps/iterations necessary to find the index. E.g. If the array has 7 numbers, it will take 3 iterations to find the last value of the array. If it has 1000 numbers, it will take log 1000 (base 2)iterations to do the same.

    public static int binarySearch(int[] arr, int target) {

        int right = arr.length - 1;

        int left = 0;

        int mid;

        while(left <= right) {

            mid = ((left+right)/2)+1;

            if(arr[mid] == target){

                return mid;

            } else if (arr[mid] > target) {                 // O(log n) where n is the size of the input array

                right = mid;

            } else {

                left = mid;

            }

        }

        return -1;

    }

### **Sorting Algorithms**

Some sorting algorithms are very efficient but very difficult to understand. Others are very easy to understand but very inefficient.

The sorting algorithms include bubble sort, selection sort, quick sort, merge sort, heap sort, etc.

#### **Bubble Sort Algorithm**

This is the easiest one to understand but it is not very efficient. It involves looping through an array, one element at a time and swapping the elements, two at a time, if the one before is larger than the one after. Once the looping is done from left to right and the largest value is at the right, the process is repeated but this time up to the second-last value. The procedure is repeated until all the values are in their correct order.

This algorithm has a big O notation value of O(n2) because it involves the use of a nested loop.

private static void bubbleSort(int[] nums) {

    int size = nums.length;

    int temp = 0;

    for(int i=0; i<size; i++) {

        for(int j=0; j<(size-i)-1; j++) {

//  The -1 is because of the last element, i.e, “if(nums[j]>nums[j+1]” will return an “ArrayIndexoutOfBounds” exception if already on the last element.

// The “size-i” is because for iteration of i, i number of elements are in their correct position at the end of the array.

            if(nums[j] > nums[j+1]) {

                temp = nums[j+1];

                nums[j+1] = nums[j];

                nums[j] = temp;

            }

        }

//            To display the array after each iteration

//            System.out.println();

//            for(int num : nums){

//                System.out.print(num + " ");

//            }

    }

}

#### **Selection Sort Algorithm**

This works in a similar manner as bubble sort except for the swapping. Here, swapping only occurs once per iteration, rather than for every pair of elements that are in the wrong order, as happens in the bubble sort algorithm. To do this a tracker variable is used to keep track of the index of the largest element as the array of elements is looped over. At the end of the loop, the value at the end is swapped with the value whose index was held in the tracker variable.

Note the same can be done but swapping according to the smallest element rather than the largest.

Also, selection sort has a time complexity of O(n2), similar to bubble sort because of the use of a nested loop, but it is more efficient because of the significantly fewer number of swaps per iteration.

public class Selection\_Sort {

    public static void main(String[] args) {

        int nums[] = {8, 47, 5, 22, 9, 3, 50, 6};

        System.out.println("Before Sorting");

        for (int num : nums) {

            System.out.print(num + " ");

        }

        System.out.println();

//        selectionSort(nums);

        selectionSortModified(nums);

        System.out.println();

        System.out.println("After Sorting");

        for (int num : nums) {

            System.out.print(num + " ");

        }

    }

    private static void selectionSort(int[] nums) {

        int countOfSorted = 0;

        int maxElement = nums[0];

        int indexOfMaxElement = 0;

        int temp = 0;

        while(countOfSorted != nums.length) {

            for(int i=0; i<((nums.length) - countOfSorted); i++) {

                if(nums[i] > maxElement) {

                    maxElement = nums[i];

                    indexOfMaxElement = i;

                }

                if(i == (((nums.length) - countOfSorted)-1)) {

                    temp = nums[i];

                    nums[i] = nums[indexOfMaxElement];

                    nums[indexOfMaxElement] = temp;

                    countOfSorted++;

                    maxElement=nums[0];

                    indexOfMaxElement=0;

                    i=0;

                    System.out.println();

                    for (int num : nums) {

                        System.out.print(num + " ");

                    }

                }

            }

        }

    }

    private static void selectionSortModified(int[] nums) {

        int maxIndex = 0;

        int temp = 0;

        System.out.println();

        for(int i=0; i<nums.length; i++) {

            for(int j=0; j<(nums.length-i); j++) {

                if(nums[j] > nums[maxIndex]) {

                    maxIndex = j;

                }

            }

//            System.out.print("Elements swapped: " + nums[(nums.length-i)-1] + ", " + nums[maxIndex] + "    [ ");

            temp = nums[(nums.length-i)-1];

            nums[(nums.length-i)-1] = nums[maxIndex];

            nums[maxIndex] = temp;

            for (int num : nums) {

                System.out.print(num + " ");

            }

//            System.out.print("]      Last Element index sorted: " + ((nums.length-i)-1));

            System.out.println();

            maxIndex = 0;

        }

        System.out.println();

    }

}

#### **Insertion Sort**

This algorithm is different from the two previous ones in that it does not involve swapping. Instead it involves shifting elements.

First the array of elements is “divided” into two such that the one part is, initially, considered sorted and the other not sorted. The sorted part is the first element of the array. Each element in the unsorted portion of the array is looped over and compared with each element in the sorted portion of the array. If the adjacent element, in the sorted portion, is larger than the one being looped over in the unsorted portion, then the element is shifted right. Before the element being looped over is placed in the space left, all the other elements in the sorted portion are compared against it and, again, if there is any other element larger than the same, the element is shifted right. Only when there is no larger element to the left of the element being looped over will this element be placed in its correct position.

This algorithm also has a time complexity of O(n2).

 private static void insertionSort(int[] nums) {

    for(int i=1; i<nums.length; i++) {

        int j = 0;

        int currentElement = nums[i];

        j = i-1;

        // Decrementing j means at some point j=-1 which is not a valid index for an array. Also, only shift when the element is larger than the current element

        while(j>=0 && nums[j]>currentElement) {

            nums[j+1] = nums[j];        // Shift the element one place to the right. currentElement holds the element whose place is being taken

            j--;                        // To look at the other elements before the one that has been shifted.

        }

        nums[j+1] = currentElement;

        // For Rendering the array for each iteration.

        for (int num : nums) {

            System.out.print(num + " ");

        }

        System.out.println();

    }

}

#### **Quick Sort**

This algorithm is more efficient than the above ones, having a best case time complexity of O(nlogn). In the worst case, however, it also has a time complexity of O(n2).

It works by dividing the arrays in chunks, sorting these chunks individually, and then combining them at the end. It is very complex. It uses the following concepts: recursion, divide and conquer (the splitting), pivot and tree.

The choosing of the pivot needs to be done on an element that is in its correct final position. Finding this pivot is when most of the work lies.

public class Quick\_Sort {

    public static void main(String[] args) {

        int nums[] = {8, 47, 5, 22, 9, 3, 50, 6, 100, -3, 45, 13};

        System.out.println("Before Sorting");

        for (int num : nums) {

            System.out.print(num + " ");

        }

        System.out.println();

        int low = 0;

        int high = nums.length-1;

        quickSort(nums, low, high);

        System.out.println();

        System.out.println("After Sorting");

        for (int num : nums) {

            System.out.print(num + " ");

        }

    }

    // Takes the array itself, the starting point, and the end point of the section that needs to be sorted. In the beginning, the section will be the whole array but eventually, it will be a smaller and smaller section of the array.

    private static void quickSort(int[] nums, int low, int high) {

        if(low < high) {

            int pivot = partition(nums, low, high);

            // Called twice because the array is split into two, based on the pivot.

            quickSort(nums, low, pivot-1);

            quickSort(nums, pivot+1, high);

        }

    }

    private static int partition(int[] nums, int low, int high) {

        int pivot = nums[high];         // For the first time, the pivot is a always the last element in the array.

        int i = low - 1;

        int temp = 0;

        for(int j=low; j<high; j++) {

            // To determine the correct position of the pivot element

            if(nums[j] < pivot) {

                i++;

                temp = nums[i];

                nums[i] = nums[j];

                nums[j] = temp;

            }

        }

        // To swap the pivot into where, in the array, it should be.

        temp = nums[i+1];

        nums[i+1] = nums[high];

        nums[high] = temp;

        return i+1;

    }

}

#### **Merge Sort**

This algorithm, similar to the quick sort algorithm, is a divide and conquer algorithm. However, it does not involve using a pivot. Instead it involves splitting the array into two parts based on a midpoint (calculated on the indices of the array rather than the elements thems), and dividing these sub-array again and again until they are one-element arrays at the bottom of the tree.

Once they have been split, they are then merged starting from the arrays on the left of the tree, and as they are merged, elements are compared between two arrays at a time, and one element at a time, until the original array is fully merged again.

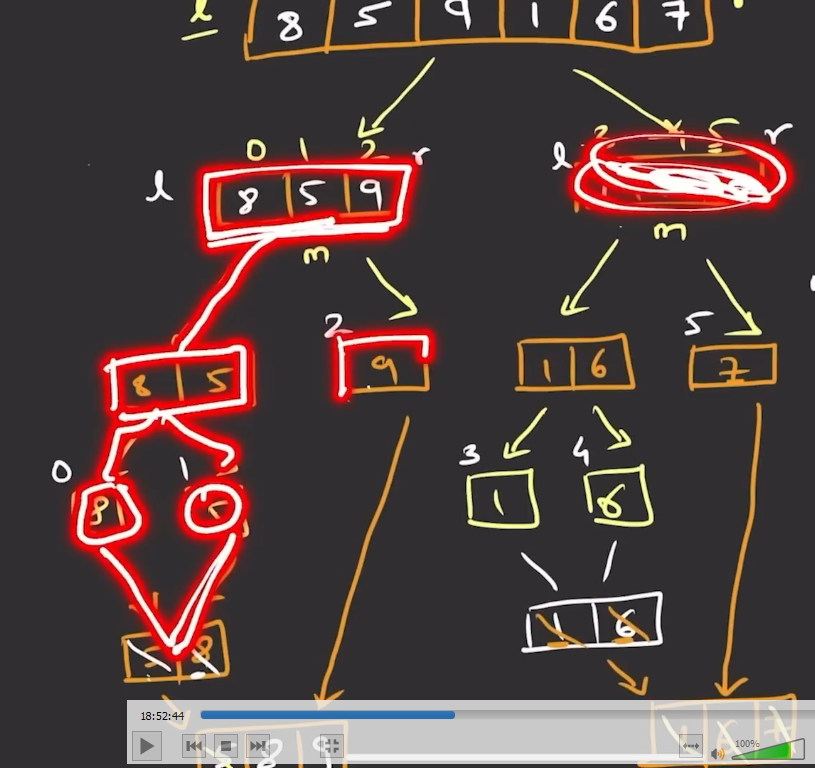


Figure 5: Merge Sort Demonstration

public class Merge\_Sort {

    private static void mergeSort(int[] nums, int left, int right) {

        if(left < right) {

            int mid = (left+right)/2;

            mergeSort(nums, left, mid);

            mergeSort(nums, mid+1, right);

            merge(nums, left, mid, right);

        }

    }

    private static void merge(int[] nums, int left, int mid, int right) {

        int leftArraySize = (mid-left)+1;

        int rightArraySize = right-mid;

        // Create two arrays to hold the elements on either side of the mid-point.

        int[] leftArray = new int[leftArraySize];

        int[] rightArray = new int[rightArraySize];

        // Copy the values from the original array to these newly-created arrays.

        for(int i=0; i<leftArraySize; i++) {

            leftArray[i] = nums[left + i];

        }

        for(int i=0; i<rightArraySize; i++) {

            rightArray[i] = nums[i+mid+1];

        }

        // Start merging the two arrays.

        int i = 0;      // To track the left array

        int j = 0;      // To track the right array

        int k = left;   // To track the original array as it is being modified.

        while(i<leftArraySize && j<rightArraySize) {    // Loop through both the left and right arrays

            if(leftArray[i] <= rightArray[j]) {         // If the left array has the smaller of values being compared, place this smaller number in the original number at the index tracked by k.

                nums[k] = leftArray[i];

                i++;

            } else {            // If the right array has the smaller value, do the same.

                nums[k] = rightArray[j];

                j++;

            }

            k++;    // Move the original array's tracker one place to the right after one element is added from either the left or right array.

        }

        // To handle the elements that remain in either the left or right array. These may been left because one of the sub-arrays was fully looped over having contained more of the smaller elements, forcing the termination of the while loop above.

        while(i<leftArraySize) {

            nums[k] = leftArray[i];

            k++;

            i++;

        }

        while(j<rightArraySize) {

            nums[k] = rightArray[j];

            k++;

            j++;

        }

    }

    public static void main(String[] args) {

        int nums[] = {8, 47, 5, 22, 9, 3, 50, 6, 100, -3, 45, 13};

//        int nums[] = {6,5,4,3,2,1};

        System.out.println("Before Sorting");

        for (int num : nums) {

            System.out.print(num + " ");

        }

        System.out.println();

        mergeSort(nums, 0, nums.length-1);

        System.out.println();

        System.out.println("After Sorting");

        for (int num : nums) {

            System.out.print(num + " ");

        }

    }

}

### **Tips for Determining if your algorithm is good**

* **If the algorithm is not iterating over all/most of the values.** E.g. To print all the even numbers between 1 and 10, the algorithm with a loop that starts at 2 and increments by 2, is more efficient that one that starts at 1, increments by 1, and checks if the number being iterated over is divisible by 2.

## **Data Structures**

### **Linked List**

This is a data structure similar to an array but with some differences. An array:

* Is stored in a continuous block of memory.
* Cannot be extended/ shrunk in size.
* Allows for indexing of its elements and therefore data retrieval from the same is very fast.

A linked list:

* Is a collection of objects known as nodes.
* Allows for the nodes to be stored in different locations on the memory. Each node has two attributes:
  + The data itself that it is supposed to hold.
  + The address of the next node in the list.
* Is extendable/ shrinkable because of the property described above.
* Is slower in terms of data retrieval because it does not support indexing. Rather, to get to a particular node, a loop is necessary to move from the first node in the list, known as the head, to the specific node being searched for.

In java, there is already an implementation of the linked list. This linked list implementation allows for several operations/methods such as:

* insert(value) – To add to the end of the list
* insertAt(index, value) – To a value at a specific “index” of the list.
* insertAtStart(value) – To add at the start/ head of the list.
* delete(value)

However, below is a custom-made linked list. Three classes are necessary to implement the same:

* A “Node” class – To represent the elements (nodes) of the linked list.
* A “LinkedList” class – To represent the linked list itself.
* A “Runner” class – To hold the main method from where the linked list can be created and operated on.

#### **Custom-made Linked List**

package Data\_Structures\_And\_Algorithms.Data\_Structures.Linked\_List;

public class LinkedListRunner {

    public static void main(String[] args) {

        LinkedList list = new LinkedList();

        list.insert(12);

        list.insert(22);

        list.insert(32);

        list.insertAtStart(25);

        list.insertAt(2,67);

        list.insertAt(0,457);

        list.deleteAt(1);

        list.show();

    }

}

package Data\_Structures\_And\_Algorithms.Data\_Structures.Linked\_List;

public class Node {         // To hold the elements of the linked list.

    int data;       // The data itself

    Node next;      // The pointer to the next node in the list.

}

package Data\_Structures\_And\_Algorithms.Data\_Structures.Linked\_List;

public class LinkedList {

    Node head;          // The Linked list must have a head, pointing to the first node. This head is null if there are nodes in the list.

    public void insert(int data) {      // To add a node at the end of the linked list

        Node node = new Node();     // To create a node object

        node.data = data;           // To assign the data being passed to the "data" attribute of the node object.

        if(head == null) {      // True for a list with no nodes yet.

            head = node;        // Assign the newly-created node to the head attribute of the Linked List.

            System.out.println("First Node Being Populated");

        } else {                // When there is at list one other node in the linked list

            Node n = head;      // Take the node in the head attribute of the list.

            while(n.next != null) {     // Loop through all the nodes in the list by moving from one node to the next using the "next" attribute of the nodes. Stop looping when the last node is reached (this last node does not have a "next" attibute. It is null).

                n = n.next;

            }

            n.next = node;      // Assign the newly-created node to the "next" attribute of the current, last node in the list.

            System.out.println("Consequent node being populated");

        }

    }

    public void insertAtStart(int data) {

        Node node = new Node();     // To create a node object

        node.data = data;

        node.next = head;

        head = node;

    }

    public void insertAt(int index, int data) {

        Node node = new Node();     // To create a node object

        node.data = data;

        if(index == 0) {            // If the index is zero, using the loop below will insert the node at index 1.

            insertAtStart(data);

        } else {

            Node n = head;

            for(int i=0; i<index-1; i++) {      // "index-1" because we need to modify the node before the one specified by "index".

                n = n.next;

            }

            node.next = n.next;         // Assign the node that was at the index specified to the newly-created node.

            n.next = node;              // Place the newly-created node at the index specified.

        }

    }

    public void deleteAt(int index) {

        if(index == 0) {

            Node n1 = head;

            head = n1.next;

            n1 = null; // Make the deleted object null so that it is available for garbage collection

        } else {

            Node n = head;

            for(int i=0; i<index-1; i++) {      // "index-1" because we need to modify the node before the one specified by "index".

                n = n.next;

            }

            Node n1 = n.next;

            n.next = n1.next;

            System.out.println(n1.data + " deleted");

            n1 = null; // Make the deleted object null so that it is available for garbage collection

        }

    }

    public void show() {        // To display the values (data) in the list.

        Node node = head;       // Start at the head

        while(node.next != null) {                  // Loop through all the nodes that point to some other node in their "next" attribute.

            System.out.print(node.data + " ");      // Print the "data" attribute of each node.

            node = node.next;                       // Go to the next node.

        }

        System.out.println(node.data);              // Print the "data" of the last node. The one whose "next" attribute is null.

    }

}

### **Stack**

A stack is another abstract data structure, whose implementation varies from language to language, and that works as Last In First Out (LIFO). Operations/methods that can be done/ called on a stack include:

* pop() – returns the lastly entered value and removes it from the top of the stack.
* push(value) – adds a value to the top of the stack.
* peak() - returns the lastly entered value but does not remove it from the top of the stack.
* isEmpty() – returns a boolean that is based on whether the stack has elements or not. True for an empty stack.
* size() – returns the number of elements remaining in the stack, that have not been popped off, that had been pushed into the stack.
* capacity() – returns the number of elements that can be held in a stack. I.e. the size/length of the array.

Note: Pushing data to an already full stack results in an overflow exception. Popping from an empty stack results in an underflow exception.

Again, java has its own implementation of stacks. However, we can created our own custom-stack. To do this we require to modify an array and create a class that will represent the stack entity, so that objects can be created as individual stacks.

Given the nature of arrays, as fixed-sized, we can create two types of stacks: fixed-sized and expandable/shrinkable stacks.

#### **Custom-made Stacks**

##### **Fixed-Sized Stack**

package Data\_Structures\_And\_Algorithms.Data\_Structures.Stack;

public class Stack {

    int stack[] = new int[5];

    int top = 0;

    public int size() {

        return top;

    }

    public int capacity() {

        return stack.length;

    }

    public void push(int data) {

        if(top >= stack.length) {

            System.out.println("Stack already full");

        } else {

            stack[top] = data;

            top++;

        }

    }

    public int pop() {

        int topElement = 0;

        if(top <= 0) {

            System.out.println("No elements remaining to pop");

        } else {

            topElement = stack[top-1];

            stack[top-1] = 0;

            top--;

        }

        return topElement;

    }

    public int peak() {

        int topElement = 0;

        if(top > 0) {

            topElement = stack[top-1];

        }

        return topElement;

    }

    public void show() {

        for(int n : stack) {

            System.out.print(n + " ");

        }

    }

}

##### **Dynamically-Sized Stack**

This involves the creation of two method that expand or shrink the stack based on whether the limit has been reached when pushing or popping respectively.

At the lowest level, it involves creating a new array of a carefully-calculated size based on the size of the existing stack. Once created, elements are copied from the original array to the newly created one and the reference variable of the stack changed to that of the newly-created array.

package Data\_Structures\_And\_Algorithms.Data\_Structures.Stack;

public class StackRunner {

    public static void main(String[] args) {

        Dynamic\_Stack dynamicStack = new Dynamic\_Stack();

        dynamicStack.push(3);

        dynamicStack.push(5);

        dynamicStack.show();

        dynamicStack.push(6);

        dynamicStack.show();

        dynamicStack.push(7);

        dynamicStack.show();

        dynamicStack.push(8);

        dynamicStack.show();

        dynamicStack.push(9);

        dynamicStack.show();

        dynamicStack.push(10);

        dynamicStack.show();

        dynamicStack.push(11);

        dynamicStack.show();

        dynamicStack.push(12);

        dynamicStack.show();

        dynamicStack.pop();

        dynamicStack.show();

        dynamicStack.pop();

        dynamicStack.show();

    }

}

package Data\_Structures\_And\_Algorithms.Data\_Structures.Stack;

public class Dynamic\_Stack {

    int capacity = 2;

    int stack[] = new int[capacity];

    int top = 0;

    public int size() {

        return top;

    }

    public int capacity() {

        return stack.length;

    }

    public void expand() {

        int[] newStack = new int[capacity \* 2];

        System.arraycopy(stack, 0, newStack, 0, stack.length);

        stack = newStack;

        capacity = stack.length;

    }

    public void shrink() {

        int[] newStack = new int[capacity / 2];

        System.arraycopy(stack, 0, newStack, 0, top);

        stack = newStack;

        capacity = stack.length;

    }

    public void push(int data) {

        if(top >= stack.length) {

            expand();

        }

        stack[top] = data;

        top++;

    }

    public int pop() {

        int topElement = 0;

        if(top <= 0) {

            System.out.println("No elements remaining to pop");

        } else {

            if(top <= (capacity / 2)) {

                shrink();

            }

            topElement = stack[top-1];

            stack[top-1] = 0;

            top--;

        }

        return topElement;

    }

    public int peak() {

        int topElement = 0;

        if(top > 0) {

            topElement = stack[top-1];

        }

        return topElement;

    }

    public void show() {

        for(int n : stack) {

            System.out.print(n + " ");

        }

        System.out.println();

    }

}

3 5

3 5 6 0

3 5 6 7

3 5 6 7 8 0 0 0

3 5 6 7 8 9 0 0

3 5 6 7 8 9 10 0

3 5 6 7 8 9 10 11

3 5 6 7 8 9 10 11 12 0 0 0 0 0 0 0

3 5 6 7 8 9 10 11 0 0 0 0 0 0 0 0

3 5 6 7 8 9 10 0

### **Queue**

This is similar to the stack except it uses the “First In First Out” (FIFO) approach rather than the stack’s LIFO. For this reason, it’s implementation involves the use of two variables, one marking the front, and another marking the rear of the queue. Also, the array implementation is known as a circular array. Here, even though the array is linear at the physical level, the “front” and “rear” variables create the illusion of having a circular array. This is done be using the modulus (%) operator when wanting to access the array indices. This is advantageous because the is no necessary swapping/ shifting of values in the queue when an element is dequeued.

The methods involved in queues include:

* enqueue(data) – adds an element to the rear of the queue.
* dequeue() – returns the element at the front of the queue.

#### **Custom-made Queue**

To create a fixed-sized queue:

package Data\_Structures\_And\_Algorithms.Data\_Structures.Queue;

public class Queue {

    int capacity = 5;

    int queue[] = new int[capacity];

    int size;

    int front;

    int rear;

    public void enqueue(int data) {

        if(!isFull()) {

            queue[rear%capacity] = data;

            rear++;

            size++;

        } else {

            throw new ArrayIndexOutOfBoundsException("The queue is full");

        }

    }

    public int dequeue() {

        if(!isEmpty()) {

            int data = queue[front%capacity];

            queue[front%capacity] = 0;

            front++;

            size--;

            return data;

        } else {

            throw new ArrayIndexOutOfBoundsException("The queue is empty");

        }

    }

    public int getSize() {

        return size;

    }

    public boolean isEmpty() {

        return getSize() == 0;

    }

    public boolean isFull() {

        return getSize() == capacity;

    }

    public void show() {

        System.out.print("Queue Elements: ");

        for(int i=0; i<getSize(); i++) {

            System.out.print(queue[(front + i)%capacity] + " ");

        }

        System.out.println();

        System.out.print("Queue Structure: ");

        for(int i=0; i<capacity; i++) {

            System.out.print(queue[i] + " ");

        }

        System.out.println();

    }

}

package Data\_Structures\_And\_Algorithms.Data\_Structures.Queue;

public class Queue\_Runner {

    public static void main(String[] args) {

        Queue queue = new Queue();

        try {

            queue.enqueue(5);

            queue.enqueue(4);

            queue.enqueue(3);

            queue.enqueue(2);

            queue.enqueue(1);

//            queue.enqueue(0);

            queue.dequeue();

            queue.dequeue();

            queue.dequeue();

            queue.enqueue(0);

            queue.enqueue(-1);

        } catch(Exception e) {

            e.printStackTrace();

        }

        queue.show();

    }

}

Queue Elements: 2 1 0 -1

Queue Structure: 0 -1 0 2 1

### **Binary Search Tree**

This ADT is similar to the linked list. However, there are a few differences:

* The elements (nodes) are sorted on insertion.
* The nodes themselves have three attributes rather than the linked list’s two: the data itself, the left node and the right node.

The binary search tree is a subset of the **tree** data structure. Trees have the nodes and edges. Edges are the linked between the nodes, drawn as straight lines. The nodes in turn have children nodes. Consequently, trees can have levels of nodes and children with the level furthest away from the root node having what are known as the leaf nodes.

Nodes also have two attributes: height and depth. Height refers to the number of levels that are directly after the same, similar to the number of descendants but for levels rather than the individual nodes. Depth refers to the number of the levels above the node, from the root node to the one just above the node. For this reason, for example, the root node has a depth of 0 and its height is the same as the height of the tree.

There are several kind of binary trees, beside the binary search tree:

* Strict Tree – where each node has either two or no children.
* Full Binary tree – where all the leaf nodes are on the same level
* Complete tree – where all the leaf nodes are either on the same level or on the level just above the lowest leaf nodes as in:

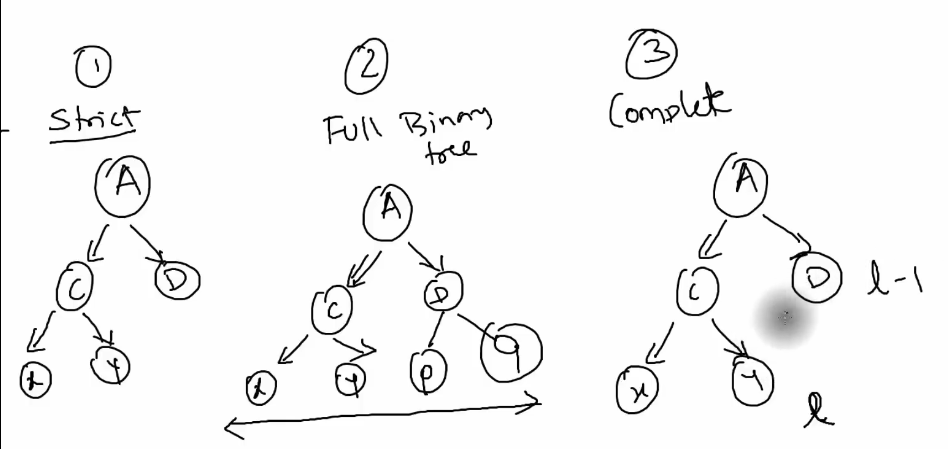


Figure 6: Binary Trees

With the binary search tree, every node has the smaller elements on the left and the bigger elements on the right. There are also several traversal modes across the tree, say during displaying the same:

* In-order – you start with the left, then root, then right nodes, e.g. X, C, Y in the strict binary tree diagram above .
* Pre-order – you start with the root, then the left, then the right nodes, as in C, X, Y.
* Post-order – you start with the left, then the right, then the root nodes, as in X, Y, C.

#### **Custom-made binary search tree:**

To implement a binary search tree:

package Data\_Structures\_And\_Algorithms.Data\_Structures.Binary\_Search\_Tree;

public class Node {

    int data;

    Node left;

    Node right;

    public Node(int data) {

        this.data = data;

    }

}

package Data\_Structures\_And\_Algorithms.Data\_Structures.Binary\_Search\_Tree;

public class BinarySearchTree {

    Node root;

    public void insert(int data) {

        root = insertRec(root, data);

    }

    public Node insertRec(Node root, int data) {

        if(root == null) {

            root = new Node(data);

        } else if (data < root.data) {

            root.left = insertRec(root.left, data);

        } else if (data > root.data) {

            root.right = insertRec(root.right, data);

        }

        return root;

    }

    public void inOrder() {

        System.out.print("In Order: ");

        inOrderRec(root);

        System.out.println();

    }

    public void preOrder() {

        System.out.print("Pre-Order: ");

        preOrderRec(root);

        System.out.println();

    }

//

    public void postOrder() {

        System.out.print("Post-Order: ");

        postOrderRec(root);

        System.out.println();

    }

    public void inOrderRec(Node root) {

        if(root != null) {

            inOrderRec(root.left);

            System.out.print(root.data + " ");

            inOrderRec(root.right);

        }

    }

    public void preOrderRec(Node root) {

        if(root != null) {

            System.out.print(root.data + " ");

            preOrderRec(root.left);

            preOrderRec(root.right);

        }

    }

    public void postOrderRec(Node root) {

        if(root != null) {

            postOrderRec(root.left);

            postOrderRec(root.right);

            System.out.print(root.data + " ");

        }

    }

}

package Data\_Structures\_And\_Algorithms.Data\_Structures.Binary\_Search\_Tree;

public class BinarySearchTreeRunner {

    public static void main(String[] args) {

        BinarySearchTree tree = new BinarySearchTree();

        tree.insert(8);

        tree.insert(7);

        tree.insert(12);

        tree.insert(15);

        tree.insert(2);

        tree.insert(5);

        tree.inOrder();

        tree.preOrder();

        tree.postOrder();

    }

}

In Order: 2 5 7 8 12 15

Pre-Order: 8 7 2 5 12 15

Post-Order: 5 2 7 15 12 8

# **JDBC (JAVA DATABASE CONNECTIVITY)**

This is how you can connect your java application to a database via a database management software (DBMS). There are many kinds of DBMSs available. Some manage relational databases (RDBMS) e.g. MySQL, MariaDB, etc. some NoSQL databases e.g. MongoDB etc. There are 8 steps involved in connecting a java application to a database:

## **Procedure For Connection A Java Application To A Database**

1. Import the SQL package - from “java.sql.\*”.
2. Load the specific DBMS’s driver - e.g. if the DBMS is MySQL, the driver is from “com.mysql.jdbc.Driver”. Download the driver and place it in the libraries, “libs”, folder of your application. Clean the project and build it again if need be.
3. Register the loaded driver - using the “forName” method from the class “Class”, say, “Class.forName(com.mysql.jdbc.Driver)“.
4. Establish the connection between the application and the database – by instantiating the “Connection” interface.
5. Create the SQL statements. There are three kinds of SQL statements:
   1. Statement – normal queries. For DQL (Data Query language) SQL.
   2. PreparedStatement – queries whose values are not provided at compile time. Queries with parameters. For DML (Data Manipulation Language) SQL. E.g. Inserting and updating. Returns the numbers of rows affected.
   3. CallableStatement – queries that emulate the stored procedures in SQL.
6. Execute the queries.
7. Process the results, e.g. if the query was a select queries, the result is a table of records. If the query was an insert query, the response is the number of records affected.
8. Close the database connection

Example code of the above flow of actions:

import java.sql.\*;

public class JDBC {

    public static void main(String[] args) {

        String url = "jdbc:mysql://localhost:3306/students";

        String username = "root";

        String password = "";

        try {

//            Class.forName("com.mysql.jdbc.Driver");           -- Deprecated

            Class.forName("com.mysql.cj.jdbc.Driver");

            Connection conn = DriverManager.getConnection(url, username, password);

            selectQuery(conn);

            insertQuery(conn);

            selectQuery(conn);

            updateQuery(conn);

            selectQuery(conn);

            conn.close();

        } catch (ClassNotFoundException e) {

            throw new RuntimeException(e);

        } catch (SQLException e) {

            throw new RuntimeException(e);

        }

    }

    public static void selectQuery(Connection conn) {

        String selectQuery = "select \* from students";

        try {

            Statement st = conn.createStatement();

            ResultSet rs = st.executeQuery(selectQuery);

            while(rs.next()) {          // rs.next() checks whether there is another record and also, if they is a next record, moves the pointer to the next record.

//                Also, it is important to note that this "rs" pointer is, at first, always before the first record. Therefore, you to "rs.next()" before doing anything if you want to retrieve the record(s) passed.

                System.out.println("Sn: " + rs.getInt("sn") + " Full Name: " + rs.getString("full\_name"));

            }

            System.out.println();

            st.close();

        } catch (SQLException e) {

            throw new RuntimeException(e);

        }

    }

    public static void insertQuery(Connection conn) {

        int sn = 6;

        String fullName = "Ivy Njeri";

        String insertQuery = "insert into students values (?,?)";

        try {

            PreparedStatement st = conn.prepareStatement(insertQuery);

            st.setInt(1, sn);

            st.setString(2, fullName);

            int count = st.executeUpdate();

            System.out.println(count + " rows affected");

            st.close();

        } catch (SQLException e) {

            throw new RuntimeException(e);

        }

    }

    public static void updateQuery(Connection conn) {

        int sn = 5;

        String fullName = "Stella Wambui Kihara";

        String insertQuery = "update students set full\_name = ? where sn = ?";

        try {

            PreparedStatement st = conn.prepareStatement(insertQuery);

            st.setInt(2, sn);

            st.setString(1, fullName);

            int count = st.executeUpdate();

            System.out.println(count + " rows affected");

            st.close();

        } catch (SQLException e) {

            throw new RuntimeException(e);

        }

    }

}

## **Data Access Model With JDBC**

For the sake of simplicity, it is important to hand the database connection outside of the business logic so that this connection logic can be handled in one module of the application rather than having to replicate the same everywhere whenever some database functionality is needed.

To do this, we use what is known as a Data Access Model which, essentially, is a class that has the database connection logic that is then called from other classes and given the parameters needed for the various queries this DAO class supports. It is conventional to store this DAO class in a separate package.

package DataAccessModel;

public class Student {

    String fullName;

    int sn;

    public Student(String name) {

        this.fullName = name;

        this.sn = StudentDAO.getLastSn() + 1;

    }

    public static void addStudent(Student s) {

        StudentDAO.addStudent(s);

    }

    public static void deleteStudent(String studentName) {

        StudentDAO.deleteStudent(studentName);

    }

}

package DataAccessModel;

import java.io.IOException;

import java.util.Scanner;

public class DAORunner {

    public static void main(String[] args) {

        System.out.print("Enter the student's name: ");

        String name = "";

        Scanner sc = new Scanner(System.in);

        name = sc.nextLine();

        Student s = new Student(name);

        Student.addStudent(s);

        System.out.print("Enter the name of the student you want to delete: ");

        name = "";

        name = sc.nextLine();

        sc.close();             // Closed once every input has been entered.

        Student.deleteStudent(name);

    }

}

package DataAccessModel;

import java.sql.\*;

public class StudentDAO {

    static String url = "jdbc:mysql://localhost:3306/students";

    static String username = "root";

    static String password = "";

    public static Connection connect() {

        try {

            Class.forName("com.mysql.cj.jdbc.Driver");

            return DriverManager.getConnection(url, username, password);

        } catch (ClassNotFoundException e) {

            throw new RuntimeException("Driver not found", e);

        } catch (SQLException e) {

            throw new RuntimeException("Connection failed", e);

        }

    }

    public static int getLastSn() {

        String selectQuery = "SELECT MAX(sn) FROM students";

        try (Connection conn = connect();

             Statement st = conn.createStatement();

             ResultSet rs = st.executeQuery(selectQuery)) {

            if (rs.next()) {

                return rs.getInt(1);

            } else {

                return 0;

            }

        } catch (SQLException e) {

            throw new RuntimeException("Error fetching last SN", e);

        }

    }

    public static void addStudent(Student s) {

        String insertQuery = "INSERT INTO students (sn, full\_name) VALUES (?, ?)";

        try (Connection conn = connect();

             PreparedStatement st = conn.prepareStatement(insertQuery)) {

            st.setInt(1, s.sn);

            st.setString(2, s.fullName);

            st.executeUpdate();

            System.out.println(s.fullName + " added as a student");

        } catch (SQLException e) {

            throw new RuntimeException("Error adding student", e);

        }

    }

    public static void deleteStudent(String studentName) {

        String deleteQuery = "DELETE FROM students WHERE full\_name = ?";

        try (Connection conn = connect();

             PreparedStatement st = conn.prepareStatement(deleteQuery)) {

            st.setString(1, studentName);

            int count = st.executeUpdate();

            if (count == 0) {

                System.out.println("No such student in the records");

            } else {

                System.out.println(studentName + " deleted from the record of students");

            }

        } catch (SQLException e) {

            throw new RuntimeException("Error deleting student", e);

        }

    }

}

# **SERVLETS AND JAVA SERVER PAGES (JSP)**

## **Servlets**

Consider a client and server setup. The server machine has a unit known as a web container that handles the dynamic requests from the client, e.g. returning a html file that is non-existent at the moment but whose creation-logic is present in the web container. Within this container are java files known as servlets, which, essentially, are class files that extend a class known as “HttpServlet”. Within this web container, is a unit known as the deployment descriptor which maps the client requests to the necessary servlet, and converts the response from the servlet into a format acceptable for return to the client, e.g. XML, HTML or JSON.

### **Procedure for creating your first Java Servlets project**

1. Ensure you have eclipse installed, *“Eclipse IDE for Enterprise Java and Web Developers - 2024-06”*.
2. Ensure you have tomcat installed. You can download it directly from google or install it as part of XAMPP.
3. Configure the tomcat server on your IDE. Here you will need to pass the directory within which tomcat was installed, e.g., *“C:\xampp\tomcat”*.
4. Create a new project of type “Dynamic Web Project”. Ensure you check the “Generate web.xml deployment descriptor”.
5. Set the newly-configured tomcat as the target runtime for your project. Here, you can right-click on your project > properties > targeted runtimes.
6. Create some html files in your project. Here, right-click on the project and select “new > HTML File”. These will be created within the “src/main/webapp” folder.
7. Create the Servlet classes within the project. Again, right-click on the project and select “new > Servlet”.
8. Link the HTML files with their respective servlet classes from within the “web.xml” deployment descriptor file located within the “src/main/webapp/WEB-INF” folder.

To demonstrate the above:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <form action="add">

        Enter the first number: <input type="number" name="num1"><br>

        Enter the second number: <input type="number" name="num2"><br>

        <input type="submit">

    </form>

</body>

</html>

*The HTML that accept two numbers and sends them to the server to be added.*

package com.demo;

import java.io.IOException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

// This is the class/servlet responsible for the adding.

// Extending the "HttpServlet" makes the class a servlet.

public class AddServlet extends HttpServlet {

    // Servlets implement the services provided by the server hence the method name "service()".

    // The servlet takes the request from the client and returns an response.

    // The "res.getWriter()" may return an IOException and hence the "throws".

    public void service(HttpServletRequest req, HttpServletResponse res) throws IOException {

        int a = Integer.parseInt(req.getParameter("num1"));             // the "getParameter()" returns a string and hence the Integer.parseInt().

        int b = Integer.parseInt(req.getParameter("num2"));

        int sum = a+b;

        res.getWriter().println("The Result is: " + sum);

    }

    //  It is possible to restrict the servlet so that it can only accept either get or post requests. We do this by using the doGet() or doPost() methods instead of service(). E.g. However, the service method in this case and should omitted.

    public void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException {

        int a = Integer.parseInt(req.getParameter("num1"));             // the "getParameter()" returns a string and hence the Integer.parseInt().

        int b = Integer.parseInt(req.getParameter("num2"));

        int sum = a+b;

        res.getWriter().println("The Result is: " + sum);

    }

    public void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException {

        int a = Integer.parseInt(req.getParameter("num1"));             // the "getParameter()" returns a string and hence the Integer.parseInt().

        int b = Integer.parseInt(req.getParameter("num2"));

        int sum = a+b;

        res.getWriter().println("The Result is: " + sum);

    }

}

*The servlet that accepts the numbers sent from the browser, add them and prints out the result.*

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd" id="WebApp\_ID" version="3.1">

  <!-- Each Servlet class requires two XML tags: <servlet> and <servlet-mapping>-->

  <servlet>

    <servlet-name>addingServlet</servlet-name>          <!-- Here the name is arbitrary -->

    <servlet-class>com.demo.AddServlet</servlet-class>          <!-- Here you must provide the fully-qualified class name. I.e the package + class name -->

  </servlet>

  <servlet-mapping>

    <servlet-name>addingServlet</servlet-name>          <!-- Here the name is arbitrary -->

    <url-pattern>/add</url-pattern>         <!-- This is the same url as that called by the "action" attribute of the form -->

  </servlet-mapping>

</web-app>

*The deployment descriptor file that maps the servlet class to the request’s named service.*

### **Calling a servlet from another servlet**

Sometimes it is necessary to call one servlet from another servlet, e.g. to call a servlet that handles the squaring of numbers from one that handles the summing of numbers such that the former returns the square of the sum of the parameters passed.

There are two ways to call servlets from other servlets:

* RequestDispatcher
* sendRedirect

#### **RequestDispatcher**

This is an interface. It can, however, be instantiated using the “request.getRequestDispatcher()” method that is provided in the HttpReceptor package.

Using RequestDispatcher is okay when the servlets belong to the same website (are in the same server) because requestDispatcher does not notify the client/browser that there has been a redirect. In fact the url, on the browser does not change from the one specified on the html form in the “action” attribute.

package com.demo;

import java.io.IOException;

import javax.servlet.RequestDispatcher;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class AddServlet extends HttpServlet {

    public void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException {

        int a = Integer.parseInt(req.getParameter("num1"));

        int b = Integer.parseInt(req.getParameter("num2"));

        int sum = a+b;

        req.setAttribute("sum", sum);       // There is no available "req.setParameter()" method;

        RequestDispatcher rd = req.getRequestDispatcher("square");          // This takes the servlet url in "<url-pattern>/square</url-pattern>" of the deployment descriptor

        rd.forward(req, res); // Works in the same way as require(file name in php, i.e. even though this servlet calls another file the url on the browser only shows this parent url, and none of the other servlet urls .

    }

}

The caller servlet

  <servlet>

    <servlet-name>squaringServlet</servlet-name>

    <servlet-class>com.demo.SquareServlet</servlet-class>           <!-- Here you must provide the fully-qualified class name. I.e the package + class name -->

  </servlet>

  <servlet-mapping>

    <servlet-name>squaringServlet</servlet-name>

    <url-pattern>/square</url-pattern>          <!-- This is the same url as that called by the "action" attribute of the form -->

  </servlet-mapping>

</web-app>

The xml configuration of the newly-added servlet

package com.demo;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class SquareServlet extends HttpServlet {

    // The type of method (service, doGet, doPost), even if called from another servlet still restrict the type of method accepted from the browser.

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException {

        int sum = (int)request.getAttribute("sum");

        int square = sum \* sum;

        PrintWriter out = response.getWriter();

        out.println("The sum is " + sum + " and the square is " + square);

    }

}

The servlet being called

#### **sendRedirect**

Sometimes we may want to notify the client that there is a redirect happening. For example, on Amazon when you click on the checkout button, you will be redirected to a payment gateway, say, paypal.

This works as follows:

* The submit of the form calls a servlet, say s1.
* The servlet triggers the redirect.
* This redirect is a response to the browser, say, r1, telling the same to send another request to the other servlet say s2. This second servlet handles the new request and returns a response, say, r2.

*Note: Using sendRedirect() always creates GET requests rather than POST methods.*

During this triggering of the redirect by the first servlet, there may be a need to pass some data. This can be done in three ways, all of which fall under a concept known as session management:

* URL Rewriting
* Cookies
* Session

##### **sendRedirect with URL Rewriting**

public class AddServlet extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException {

        int a = Integer.parseInt(req.getParameter("num1"));

        int b = Integer.parseInt(req.getParameter("num2"));

        int sum = a+b;

        res.sendRedirect("square?sum=" + sum);          // URL Rewriting - This always creates a GET request.

    }

}

public class SquareServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {

        int sum = Integer.parseInt(request.getParameter("sum"));

        response.getWriter().println("The sum of the values is " + sum + " and the square of this sum is " + sum \* sum);

    }

}

##### **sendRedirect using Session**

Sometimes we may want to pass multiple parameters without wanting to cluter the URL, or we may have sensitive information we want to pass etc. Passing the same over the URL is risky because the client machine can easily be penetrated illegally. For this reason, it is wiser to store the data within the server where it is more secure. Sessions provide a means using which we can do that.

In the same way that the tomcat server provides the instances of the HttpServletRequest and HttpServletResponse, tomcat also provides the instances of the sessions. As such:

public class AddServlet extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException {

        int a = Integer.parseInt(req.getParameter("num1"));

        int b = Integer.parseInt(req.getParameter("num2"));

        int sum = a+b;

        HttpSession session = req.getSession();

        session.setAttribute("sum", sum);

        res.sendRedirect("square");

    }

}

public class SquareServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {

        HttpSession session = request.getSession();     // Refers to the same session as the one referenced by the add servlet.

        int sum = (int)session.getAttribute("sum");

        response.getWriter().println("The sum of the values is " + sum + " and the square of this sum is " + sum \* sum);

        session.removeAttribute("sum");         // Remove the attribute from session storage.

    }

}

##### **sendRedirect using Cookies**

This is slightly similar to session except, cookies are sent back to the client as part of the HttpServletResponse. When retrieving the same from the server, from the next servlet, the HttpServletRequest contains all the cookies from the client and the server therefore has to find the one it created from amongst the list of cookies passed by the request, using the key of the cookie.

public class AddServlet extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException {

        int a = Integer.parseInt(req.getParameter("num1"));

        int b = Integer.parseInt(req.getParameter("num2"));

        int sum = a+b;

        Cookie cookie = new Cookie("sum", sum + "");        // Adding the empty string converts the integer to a string.

        res.addCookie(cookie);

        res.sendRedirect("square");

    }

}

public class SquareServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {

        Cookie cookies[] = request.getCookies();

        int sum = 0;

        for(Cookie c : cookies) {

            if(c.getName().equals("sum")) {

                sum = Integer.parseInt(c.getValue());

            }

        }

        response.getWriter().println("The sum of the values is " + sum + " and the square of this sum is " + sum \* sum);

    }

}

### **ServletContext and ServletConfig**

Sometimes, you may want to specify some parameters outside the individual servlets so that the same can be shared across the servlets, or still do the same but the restrict the parameter’s visibility to a particular servlet. For example, you may want to specify the username, or file location etc. that are used within at least one of the servlets. To do this, we use ServletContext and ServletConfig.

ServletContext provides these parameters to all the servlets. ServletConfig provides the parameters to a particular servlet.

  <servlet>

    <servlet-name>myServlet</servlet-name>

    <servlet-class>com.demo.MyServlet</servlet-class>

    <init-param>                                          <!-- ServletConfig - Restricted to this servlet-->

      <param-name>bodyCount</param-name>

      <param-value>0</param-value>

    </init-param>

  </servlet>

  <servlet-mapping>

    <servlet-name>myServlet</servlet-name>

    <url-pattern>/myServlet</url-pattern>

  </servlet-mapping>

  <context-param>                                        <!-- ServletContext - Available for all the servlets-->

    <param-name>name</param-name>

    <param-value>Daniel</param-value>

  </context-param>

  <context-param>

    <param-name>age</param-name>

    <param-value>23</param-value>

  </context-param>

</web-app>

@WebServlet(name = "MyServlet", urlPatterns = { "/MyServlet" })

public class MyServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        // ServletContext:

//          - All the servlets can access its parameters

//          - Defined using the <context-param> tag

        ServletContext ctx = getServletContext();

        String name = ctx.getInitParameter("name");

        int age = Integer.parseInt(ctx.getInitParameter("age"));

        response.getWriter().println("Hi " + name + ". You are " + age + " years old");

//      ServletConfig:

//          - Restricted to one servlet, the one within which the parameters are defined.

//          - Defined using the <init-param> tag, with the servlet's <servlet> tag

        ServletConfig cfg = getServletConfig();

        int bodyCount = Integer.parseInt(cfg.getInitParameter("bodyCount"));

        response.getWriter().println("You have slept with " + bodyCount + " people");

    }

}

### **Servlet Annotation Configuration**

We do not need to configure everything on the web.xml deployment descriptor. Instead, we can use annotations on the servlets themselves, passing the URLs of the servlets.

@WebServlet("/add")

public class AddServlet extends HttpServlet {

}

@WebServlet("/square")

public class SquareServlet extends HttpServlet {

}

Following this notation, the forms targeting this servlet files should have their “action” attributes being as:

<form action="add" method="post"> <!-- Note the missing "/"-->

  Enter the first number: <input type="number" name="num1"><br>

  Enter the second number: <input type="number" name="num2"><br>

<input type="submit">

</form>

## **Java Server Pages (JSP)**

Servlets are good but they are also very verbose. For instance, to render a HTML page via servelets, the html tags have to be written within “response.getWriter().println()”. This, if the page is big will lead to the you writing a lot of code.

Java server pages are how that problem was handled. These JSPs are essentially HTML files, that allow for java for to be coded within the same. This java code is separated using “<% java code %>”, known as a scriplet.

To link the HTML files from the client machine to the JSP files in the server, we link the same in the “action” attribute of the forms, where we include the “.jsp” extention.

Also, JSP provide implicit objects such as request and response so that you do not have to explicitly, manually write them down.

However, tomcat can only run servlets. It cannot run JSP files. For this reason, all JSP files are translated in servlets with the class names of the servlets being the JSP file name together with the JSP extension combined into a single name. E.g. “demo.JSP” becomes “DemoJSP”.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <form action="add.jsp" method="post">

        Enter the first number: <input type="number" name="num1"><br>

        Enter the second number: <input type="number" name="num2"><br>

        <input type="submit">

    </form>

</body>

</html>

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body bgcolor="cyan">

    <%

        int a = Integer.parseInt(request.getParameter("num1"));     // Implicit "request" object

        int b = Integer.parseInt(request.getParameter("num2"));

        int sum = a+b;

        out.println("The sum is " + sum);                           // Implicit "PrintWriter" object

    %>

</body>

</html>

### **Types of tags in JSP**

There are four kinds of tags in JSP:

#### **Directive**

Where you want to specify the imports. Denoted by *“<%@ page import=”someImport, anotherImport” %>”*. Directive tags have three sub-directives:

##### **page**

Used to define page-dependent attributes like scripting language, error pages, buffer size, etc. It has the following attributes:

* language: Specifies the scripting language used in the page, typically set to "java".
* contentType: Defines the MIME (Multipurpose Internet Mail Extensions) type of the response. Default is "text/html".
* import: Used to import Java classes, similar to the import statement in Java.
* session: Indicates whether the page participates in an HTTP session. Default is true.
* buffer: Specifies the buffer size for the output stream.
* autoFlush: Indicates whether the buffer should automatically flush when it is full. Default is true.
* isThreadSafe: Declares whether the JSP page is thread-safe. Default is true.
* errorPage: Defines a JSP page to be used as an error page.
* isErrorPage: Specifies whether the page is an error page. Default is false.
* extends: Specifies a superclass for the JSP-generated servlet.
* info: Provides information about the JSP page.

An example:

<%@ page

    language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" import="java.util.\*, java.text.\*" session="true" buffer="8kb" autoFlush="true" isThreadSafe="true" errorPage="error.jsp" isErrorPage="false" extends="com.example.CustomServlet" info="This is a JSP page example"

%>

##### **include**

Used to include the content of another file at the time the JSP is translated into a servlet. It’s a static inclusion, meaning that the included content is integrated into the JSP page before the page is processed.

It has one attribute, “file”, that specifies the actual file to be included. For example, To incorporate a header that is common to many pages, simply include the separated-out file that has the header’s design and/or logic:

<%@ include file="header.jsp" %>

##### **taglibs**

Used to declare a tag library containing custom tags that can be used in the JSP page. This directive has two attributes:

* uri: Identifies the URI that references the tag library descriptor (TLD) file.
* prefix: Defines a prefix that will be used in the JSP page to refer to the custom tags.

For example:

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

#### **Declaration**

Where you want to specify the properties of the JSP file’s destination class that are outside the “service()” method, say the global variables. Denoted by *“<%! someClassAttributes %>”*

#### **Scriplet**

Where you specify what you want to go into the service method. Denoted by *“<%%>”*

#### **Expression**

Where you specify what you want to go into the out.println(). Denoted by *“<%= someOutput %>”*. Accepts a single expression only. This expression is not even terminated by a semi-colon.

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body bgcolor="cyan">

    <%@ page

        import="javax.servlet.ServletException"

    %>

    <%!

        int sum = 0;

    %>

    <%

        int a = Integer.parseInt(request.getParameter("num1"));

        int b = Integer.parseInt(request.getParameter("num2"));

        sum = a+b;

    %>

    <%=

        "The sum is " + sum

    %>

</body>

</html>

### **Implicit Objects in JSP**

JSP provides several objects that are already initialized by the server and only need to be referenced within the JSP files. These objects can be used within the scriplet and expression directives of the JSPs.

1. request
2. response
3. out
4. session
5. application – this is the “ServletContext”
6. config – this is the “ServletConfig”
7. pageContext – This one is similar to the “ServletContext” and “ServletConfig” except it is, by default, restricted to the page you are on. However, it takes a third parameter that allows you to change the scope to which it is restricted, say to the session. For example:

pageContext.setAttribute("name", "Daniel", PageContext.SESSION\_SCOPE);

1. exception – this gets the exception triggered in the event of an runtime-error. However, it only works in files that are error pages, i.e. with a *“<%@ page isErrorPage=”true” %>”.* It is important to note that JSP allow for the try … catch syntax but this is discouraged. Instead, it is an industry convention to have a dedicated page that shows the errors, hence the “isErrorPage” page attribute.

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8" errorPage="errorpage.jsp"%>

    <!-- Note the errorPage="errorpage.jsp" above. This is what links this page to the error page -->

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body bgcolor="cyan">

    <%

        <!-- Discouraged -->

        <!-- try {

            sum = sum / 0;      // Manually triggering an error

        } catch (Exception e) {

            e.getMessage();

        } -->

        sum = sum / 0;      // Manually triggering an error

    %>

</body>

</html>

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8" isErrorPage="true"%>

    <!-- Note the "isErrorPage="true"" -->

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body bgcolor = "red">

    <%= "Error: " + exception.getMessage() %>           <!-- The implicit object "exception"-->

</body>

</html>

### **Connecting a database to a JSP file using JDBC**

The procedure for doing this is the same as that used when linking a normal .java file/project to a database. However, it is important to note that when loading the jdbc jar file, you should place it within the “src/main/webapp/WEB-INF/lib” directory.

### **MVC Architecture**

This is an industry standard that works as follows:

* The entities involved are the client machine, The JSP files that are used to only render the user interface/view (V), and the data that is being rendered on the view, known as the model (M). This model data being fed into the view is coming from a controller (C) , that essentially is a servlet.
* The controller servlet should, at the same time, only accept requests from the client and return responses to the same. No business logic should be written within it. For this reason, the controller should have its business logic implemented by another class.

### **Expression Language in JSP (JSP EL)**

According to the MVC model described above, there should be little, to no, java code written within the JSP files, the view. This is probably because the designers of this page are not even java developers. For this reason, the expression languages was developed that simplified the code so that it would be less verbose and allow for the use of short variable-like statements that would hold the model data that was being passed from the controller servlets.

For example, instead of having the JSP file create a variable and initialize this variable with the data gotten from the Session attribute, after first creating a reference to this session, the whole of this logic can be simplified into a single variable, say, ${sessionAttributeName}.

The end result is that EL helps to keep the JSP pages clean by replacing Java scriptlets (<% %>).

@WebServlet("/ProfileController")

public class ProfileController extends HttpServlet {

        protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException{

        request.setAttribute("name", "Karongo");       // There is no available "req.setParameter()" method;

        RequestDispatcher rd = request.getRequestDispatcher("/profile.jsp");

//        This will work because the path provided to "getRequestDispatcher()" is relative to the web application's context root even though this file is in a different package altogether.

        rd.forward(request, response);

        }

}

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <!-- <%

        String name = (String)request.getAttribute("name");

    %>

    <%= name %> -->

    <!-- The above can be simplified as: -->

    ${name}

</body>

</html>

### **JSTL (JavaServer Pages Standard Tag Library)**

The JavaServer Pages Standard Tag Library (JSTL) is a powerful collection of tags designed to simplify the development of dynamic web pages in Java-based web applications. JSTL encapsulates common tasks like iteration, conditional processing, and internationalization into a set of easy-to-use tags, making it easier for developers to manage complex logic within JSP pages without embedding Java code directly in the HTML.

JSTL has a close and complementary relationship with JSP Expression Language (EL). For example, JSTL tags can evaluate expressions written in EL. I.e. JSTL core tags, such as <c:out>, <c:if>, and <c:forEach>, often utilize EL to evaluate expressions within the tags. For example, <c:out value="${user.name}" /> uses EL to output the name property of the user object.

#### **Procedure for getting started with JSTL**

* Download the necessary JAR files from the internet. Depending on the source, there are a combination of files, or one file is enough.
  + jstl-1.2.jar
  + standard.jar (if needed)
* Copy the files and paste them into the “WEB-INF/lib” folder. This will automatically add them into the classpath.
* Define the “taglib” in the JSP file where you want to use them. To do this:
  + Use the <%@ taglib%> directive
  + Set the prefix as any arbitrary string. The convention, however, is ”c”.
  + Set the URI. This is similar to the web.xml deployment descriptor in servlets, i.e, it is a URL that helps to uniquely identify and locate the tag library within the JSP engine. On eclipse, pressing “ctrl+space” when inside the quotes in uri”” will return the list of available URIs and you can then choose “=<http://java.sun.com/jsp/jstl/core>.
* Use the JSTL tags within the file.

#### **JSTL Nuances**

When working with objects, JSTL tags can only work with beans, i.e, objects with defined getters and setters.

public class Student {

    int regNo;

    String Name;

    public Student(int regNo, String name) {

        super();

        this.regNo = regNo;

        Name = name;

    }

    // Not using the getters and setters will result in a nullExceptionPointer

    public int getRegNo() {

        return regNo;

    }

    public void setRegNo(int regNo) {

        this.regNo = regNo;

    }

    public String getName() {

        return Name;

    }

    public void setName(String name) {

        Name = name;

    }

@Override

public String toString() {

    return "Student [regNo=" + regNo + ", Name=" + Name + "]";

}

}

public class ProfileController extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException{

        Student s = new Student(1, "Karongo");

        request.setAttribute("student", s);

        RequestDispatcher rd = request.getRequestDispatcher("/profile.jsp");

        rd.forward(request, response);

    }

}

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <c:out value="${student.name}" />

</body>

</html>

JSTL has many tags as described in the official documentation:

|  |  |
| --- | --- |
| [**catch**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/catch.html) | Catches any Throwable that occurs in its body and optionally exposes it. |
| [**choose**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/choose.html) | Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise> |
| [**if**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/if.html) | Simple conditional tag, which evalutes its body if the supplied condition is true and optionally exposes a Boolean scripting variable representing the evaluation of this condition |
| [**import**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/import.html) | Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'. |
| [**forEach**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/forEach.html) | The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality |
| [**forTokens**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/forTokens.html) | Iterates over tokens, separated by the supplied delimeters |
| [**out**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/out.html) | Like <%= ... >, but for expressions. |
| [**otherwise**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/otherwise.html) | Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false' |
| [**param**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/param.html) | Adds a parameter to a containing 'import' tag's URL. |
| [**redirect**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/redirect.html) | Redirects to a new URL. |
| [**remove**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/remove.html) | Removes a scoped variable (from a particular scope, if specified). |
| [**set**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/set.html) | Sets the result of an expression evaluation in a 'scope' |
| [**url**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/url.html) | Creates a URL with optional query parameters. |
| [**when**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/when.html) | Subtag of <choose> that includes its body if its condition evalutes to 'true' |

public class ProfileController extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException{

        List<Student> students = Arrays.asList(new Student(1, "Karongo"), new Student(2, "Violet"), new Student(3, "Isaac"));

        request.setAttribute("students", students);

        RequestDispatcher rd = request.getRequestDispatcher("/profile.jsp");

        rd.forward(request, response);

    }

}

<body>

    <c:forEach items="${students}" var="s">

        ${s} <br/>

    </c:forEach>

</body>A

#### **JSTL SQL Tags**

JSTL also allows for database connectivity similar to JDBC. However, this functionality should be handled by the Controller implementer in the MVC model.

For example:

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>           <!-- URI for the SQL functions -->

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>            <!-- URI for the core JSTL tags -->

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <sql:setDataSource var="db" driver="com.mysql.cj.jdbc.Driver" url="jdbc:mysql://localhost:3306/students" user="root" password=""/>      <!-- To establish the connectioN. You need:

     the arbitrary name of the variable holding the database (var)

     the jdbc driver (similar procedure as the jdbc except you place the jar file within "WEB-INF/lib") - (driver)

     the url of the db - (url)

     the user name of the database server - (user)

     the password of the database - (password)

-->

<sql:query var="rs" dataSource="${db}">select \* from students</sql:query>

<!-- To specify the sql query itself, you need:

     the arbitrary name of the variable holding the resultset (var)

     the variable holding the database within which the query will be executed. - (dataSource)

     the query itself

-->

    <c:forEach items="${rs.rows}" var="student">

        <c:out value="${student.regNo} : ${student.name}"></c:out>

        </br>

    </c:forEach>

</body>

</html>

#### **JSTL Function tags**

JSTL also provides many functions that can be applied on the content. For example, finding the length of a string:

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <c:set var="str" value="Karongo is Full Stack Developer. He is proficient in Angular(TS) and Springboot(Java)"/>

    ${str} </br>

    Length: ${fn:length(str)}   </br>

    <c:forEach items="${fn:split(str, ' ')}" var="word">

        ${word} </br>

    </c:forEach>

    Index of "is": ${fn:indexOf(str, "is")} </br>

    <c:if test="${fn:contains(str, 'Java')}">

        Karongo is a Java developer </br>

    </c:if>

    <c:choose>

        <c:when test="${fn:contains(str, 'C#')}">

            Karongo is a C# developer</br>

        </c:when>

        <c:when test="${fn:contains(str, 'Python')}">

            Karongo is a Python developer</br>

        </c:when>

        <c:otherwise>

            Karongo is not mentioned as a C# or Python developer</br>

        </c:otherwise>

    </c:choose>

    <c:if test="${fn:endsWith(str, '(Java)')}">

        Karongo is a proficient Java developer  </br>

    </c:if>

    ${fn:toUpperCase(str)}      </br>

</body>

</html>

## **Filters**

These are classes that lie in between the client and the servlets. Their job is to intercept the Http request and do some validation, stopping the request and returning a response to the client in the event of a failure in the validation of the request. Multiple filters can be configured in what is called a filter chain such that the request is passed onto the next filter if it passes the validation of the current filter. The filters are unaware of each other. The servlets are also unaware of the filter. Also, individual filters can be linked to multiple servlets.

Similar to the servletconfig, filters also have a filterconfig, which is also done on the web.xml deployment descriptor file. These filter classes implement an interface known as “Filter”.

To create a filter simply, on eclipse, right-click on the project > new > filter and then configure the necessary parameters.

<body>

    <a href="addTeacher.html">Add Teacher</a>

    <form action="AddStudent" method="post">

        Enter the id of the student: <input type="number" name="id"><br>

        Enter the name of the student: <input type="text" name="name"><br>

        <input type="submit">

    </form>

</body>

<body>

    <a href="index.html">Add Student</a>

    <form action="AddTeacher" method="post">

        Enter the id of the teacher: <input type="number" name="id"><br>

        Enter the name of the teacher: <input type="text" name="name"><br>

        <input type="submit">

    </form>

</body>

Two views pointing to two different servlets

@WebServlet("/AddStudent")

public class AddStudent extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response) throws IOException {

        System.out.println("In Student Servlet");

        int id = Integer.parseInt(request.getParameter("id"));

        String name = request.getParameter("name");

        response.setContentType("text/html");

        response.getWriter().println(id + " : " + name);

    }

}

@WebServlet("/AddTeacher")

public class AddTeacher extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        System.out.println("In Teacher Servlet");

        int id = Integer.parseInt(request.getParameter("id"));

        String name = request.getParameter("name");

        response.setContentType("text/html");

        response.getWriter().println(id + " : " + name);

    }

}

Two servlets for the two different views.

@WebFilter(urlPatterns={"/AddStudent", "/AddTeacher"})      // Linked to multiple servlets

public class IDFilter implements Filter {

    public void destroy() {

    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

        System.out.println("In ID Filter");

        HttpServletRequest req = (HttpServletRequest) request;      // Type-cast the request to be of type "HttpServletRequest"

        int id = Integer.parseInt(req.getParameter("id"));

        if(id >= 1) {

            chain.doFilter(request, response);

        } else {

            response.setContentType("text/html");

            response.getWriter().println("Invalid ID");

        }

    }

    public void init(FilterConfig fConfig) throws ServletException {

    }

}

@WebFilter(urlPatterns={"/AddStudent", "/AddTeacher"})  // Linked to multiple servlets

public class NameFilter implements Filter {

    public void destroy() {

    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

        System.out.println("In Name Filter");

        HttpServletRequest req = (HttpServletRequest) request;      // Type-cast the request to be of type "HttpServletRequest"

        String name = req.getParameter("name");

        if(name.length() >= 3) {

            chain.doFilter(request, response);

        } else {

            response.setContentType("text/html");

            response.getWriter().println("Invalid Name");

        }

    }

    public void init(FilterConfig fConfig) throws ServletException {

    }

}

Two different filters each linked to multiple servlets such that the filter intercepts all requests that are headed to any of the linked servlets.

## **Logging In/Out Management using Sessions**

Many times in enterprise-level applications, there is usually a need for logging in so as to access the enterprise’s system. Consequently, there are some views within that enterprise’s website that will be locked such that only a user who is logged can access the said pages.

Also, once the user is done he/she may want to log out. This should be implemented in such a way that if someone presses the back button on the browser after logging out, the restricted pages should still remain locked.

To implement the above, two things are needed:

* To invalidate/clear the session on the server. This session is browser specific such that a user on one browser will still need to log in if he/rsshe wants to access the same pages in a different browser.
* To configure the restricted pages such that they are not cached in the browser and/or proxy servers.

To demonstrate this:

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <form action="login" method="post">

        Enter the user name: <input type="text" name="uname"><br>

        Enter the password: <input type="password" name="pass"><br>

        <input type="submit" value="Submit">

    </form>

</body>

</html>

The first login page

@WebServlet("/login")

public class Login extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException  {

        String uname = (String) request.getParameter("uname");

        String pass = (String) request.getParameter("pass");

        if(DAO.login(uname, pass)) {

            HttpSession session = request.getSession();

            session.setAttribute("uname", uname);

            response.sendRedirect("welcome.jsp");

        } else {

            response.sendRedirect("login.jsp");

        }

    }

}

The controller called from the login page.

package com.demo.DAO;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

public class DAO {

    static String url = "jdbc:mysql://localhost:3306/students";

    static String username = "root";

    static String password = "";

    public static boolean login(String uname, String pass) {

        try {

            Class.forName("com.mysql.cj.jdbc.Driver");

            // Try-with-resources block for Connection

            try (Connection conn = DriverManager.getConnection(url, username, password)) {

                String query = "select \* from users where username=? and password=?";

                // Try-with-resources block for PreparedStatement

                try (PreparedStatement st = conn.prepareStatement(query)) {

                    st.setString(1, uname);

                    st.setString(2, pass);

                    try (ResultSet rs = st.executeQuery()) {

                        if (rs.next()) {

                            return true;

                        }

                    }

                }

            }

        } catch (Exception e) {

            e.printStackTrace();

            e.getMessage();

            throw new RuntimeException(e);

        }

        return false;

    }

}

The DAO for this login controller

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

    <meta charset="UTF-8">

    <title>Insert title here</title>

</head>

<body>

    <%

        // Prevents caching

        response.setHeader("Cache-Control", "no-cache, no-store, must-revalidate");  // HTTP 1.1

        response.setHeader("Pragma", "no-cache");  // HTTP 1.0

        response.setHeader("Expires", "0");  // Proxies

        // Check if the user is logged in

        if (session.getAttribute("uname") == null) {

            response.sendRedirect("login.jsp");

            return;  // Ensure no further code is executed after redirection

        }

    %>

    <!-- Display welcome message and username -->

    Welcome, ${sessionScope.uname}<br>

    <a href="videos.jsp">Videos</a><br>

    <!-- Logout form -->

    <form action="logout" method="post">

        <input type="submit" value="Logout">

    </form>

</body>

</html>

<%@ page language="java" contentType="text/html; charset=UTF-8"

    pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <%

        // Prevents caching

        response.setHeader("Cache-Control", "no-cache, no-store, must-revalidate");  // HTTP 1.1

        response.setHeader("Pragma", "no-cache");  // HTTP 1.0

        response.setHeader("Expires", "0");  // Proxies

        // Check if the user is logged in

        if (session.getAttribute("uname") == null) {

            response.sendRedirect("login.jsp");

            return;  // Ensure no further code is executed after redirection

        }

    %>

    <iframe width="560" height="315" src="https://www.youtube.com/embed/4XTsAAHW\_Tc?si=4Axsiu0g0-3Zolyr" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" referrerpolicy="strict-origin-when-cross-origin" allowfullscreen></iframe>

    <!-- This tag is obtained from youtube's share > embed panel-->

</body>

</html>

The pages blocked for all non-logged-in users

@WebServlet("/logout")

public class Logout extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException{

        HttpSession session = request.getSession();

        session.removeAttribute("uname");

        session.invalidate();               // To clear the session

        response.sendRedirect("login.jsp");     // To redirect to the login page

    }

}

The controller holding the logout logic

# **MAVEN**

Consider the steps we followed to add the jdbc and jstl libraries to the projects. Imagine having a project with very many, such, dependencies. This procedure would be very cumbersome to repeat for all the dependencies. As such, there is need for an automated equivalent to the above.

Maven is a build tool that provide the automation described above. When using maven, all you have to do is specify the dependency you want to include in your project in a file called ”pom.xml”. On saving this file, maven automatically downloads these dependencies and adds them to your project.

To create a maven project in eclipse IDE:

* Go “File > New > Maven Project”.
* Choose the location of the project.
* Select an archetype for your project. This is like the type of the project, say a web application, etc.
  + Select the catalog as “Internal”
  + Write a filter, say “web” then select the apache archetype that is left, say, “maven-archetype-webapp”
* Configure the other project details
  + Group ID, e.g. “com.housesearch”
  + Articact ID, e.g.”FileUpload”
  + The package in the above case will be “com.housesearch.FileUpload”
* Wait for the maven initialization downloads to finish.
* Configure the pom.xml to include your desired project dependencies.

## **Maven Example Project – File Uploads**

Create a view from which you can select the files you want to upload.

<html>

<body>

    <h2>Hello World!</h2>

    <form action="upload" method="post" enctype="multipart/form-data">

        <input type="file" name="file" multiple/>

        <br>

        <input type="submit">

    </form>

</body>

</html>

Create a servlet to control the actual file upload:

import java.io.File;

import java.io.IOException;

import java.util.List;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;

import org.apache.commons.fileupload.FileItem;

import org.apache.commons.fileupload.disk.DiskFileItemFactory;

import org.apache.commons.fileupload.servlet.ServletFileUpload;

public class FileUpload extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException {

        // Configure the DiskFileItemFactory to use a temporary directory

        DiskFileItemFactory factory = new DiskFileItemFactory();

        File tempDir = (File) request.getServletContext().getAttribute("javax.servlet.context.tempdir");

        factory.setRepository(tempDir);

        HttpSession session = request.getSession();

        // Create a new instance of ServletFileUpload to handle file uploads

        ServletFileUpload sf = new ServletFileUpload(factory);

        try {

            // Get the real path of the "File Uploads" directory relative to the project context

            String uploadDirectory = request.getServletContext().getRealPath("File Uploads");

            // Create the "File Uploads" directory if it doesn't exist

            File uploadDir = new File(uploadDirectory);

            if (!uploadDir.exists()) {

                uploadDir.mkdirs();

            }

            // Parse the request to get all the uploaded files

            List<FileItem> files = sf.parseRequest(request);

            // Iterate over each uploaded file

            for (FileItem file : files) {

                // Print the name of the uploaded file to the console (for debugging purposes)

                System.out.println(file.getName());

                // Handle duplicate file names by checking if the file already exists

                File uploadedFile = new File(uploadDir, file.getName());

                if (uploadedFile.exists()) {

                    // Generate a unique file name if a file with the same name exists

                    String baseName = file.getName().substring(0, file.getName().lastIndexOf('.'));

                    String extension = file.getName().substring(file.getName().lastIndexOf('.'));

                    int count = 1;

                    // Increment the count until a unique name is found

                    while (uploadedFile.exists()) {

                        String newFileName = baseName + "\_" + count + extension;

                        uploadedFile = new File(uploadDir, newFileName);

                        count++;

                    }

                }

                // Write the file to the "File Uploads" directory

                file.write(uploadedFile);

                // Delete the temporary file used during the upload process

                if (!file.isInMemory()) {

                    file.delete();

                }

            }

            session.setAttribute("fileUploadSuccess", "true");

            System.out.println("About to redirect after a success");

            // Redirect to a success page (PRG pattern)

            response.sendRedirect("afterUpload.jsp");

        } catch (Exception e) {

            // Handle any exceptions that occur during file upload

            System.out.println(e.getMessage());

            session.setAttribute("fileUploadSuccess", "false");

            response.sendRedirect("afterUpload.jsp"); // Redirect on error

        }

    }

}

Create a view to redirected to after the upload are done. This is important so that in the event that the user reloads the page after the download is done, the server does not re-upload the same file(s) again.

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<!DOCTYPE html>

<html>

<head>

    <title>Upload Success</title>

</head>

<body>

    <c:if test="${sessionScope.fileUploadSuccess == 'true'}">

        <h1>Files Uploaded Successfully</h1><br>

           <a href="index.jsp">Upload another file</a>

    </c:if>

    <c:if test="${sessionScope.fileUploadSuccess == 'false'}">

        <h1>File Upload Failed.</h1><br>

            <a href="index.jsp">Please try again</a>

    </c:if>

    Hello

</body>

</html>

# **HIBERNATE**

Hibernate is an ORM (Object Relational Mapping) tool that bridges the gap between DBMSs and the java applications. Originally, this connection was done via JDBC but with Hibernate, we don’t even need to write SQL queries. For example to save an object to a record in a table in a database we can simply say “save(object)”.

This mapping works as follows:

The tables in the database are essentially classes in the java applications. The columns of the tables are the class attributes. Objects are the records in the tables.

## **Sample Hibernate Project**

1. Create a simple maven project.
2. Add hibernate and DBMS connector dependencies to the pom.xml file.

<dependencies>

    <dependency>

      <groupId>junit</groupId>

      <artifactId>junit</artifactId>

      <version>3.8.1</version>

      <scope>test</scope>

    </dependency>

    <!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core -->

  <dependency>

      <groupId>org.hibernate.orm</groupId>

      <artifactId>hibernate-core</artifactId>

      <version>6.6.0.Final</version>

  </dependency>

  <dependency>

      <groupId>com.mysql</groupId>

      <artifactId>mysql-connector-j</artifactId>

      <version>9.0.0</version>

  </dependency>

  </dependencies>

1. Install a Hibernate configuration plugin to the IDE, say, the JBoss plugin.
2. Create a Hibernate configuration file (hibernate.cfg.xml) and configure the database connection settings. This file should be under “src/main/resources” and this folder configured as the “resources” folder of the project. Also, you add a property that updates the database automatically, creating where none exists and updating where something exists, e.g. tables.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC

    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/javarefresher</property>

        <property name="hibernate.connection.username">root</property>

        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

        <property name="hbm2ddl.auto">update</property>

    </session-factory>

</hibernate-configuration>

1. Create an entity class and specify its attributes.

package com.housesearchKE.DemoHibernate;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table(name = "developers") // Optional: If you want to specify a custom table name

public class Developer {

  @Id

  @GeneratedValue(strategy = GenerationType.IDENTITY) // AUTO\_INCREMENT equivalent

  @Column(name = "sn") // Optional: If your column name is different in the database

  private int sn;

  @Column(name = "full\_name") // Optional: Maps this field to the full\_name column

  private String full\_name;

  @Column(name = "technology") // Optional: Maps this field to the technology column

  private String technology;

  // Default constructor required by Hibernate

  public Developer() {}

  public Developer(String full\_name, String technology) {

    this.full\_name = full\_name;

    this.technology = technology;

  }

  // Getters and Setters

  public int getSn() {

    return sn;

  }

  public void setSn(int sn) {

    this.sn = sn;

  }

  public String getFull\_name() {

    return full\_name;

  }

  public void setFull\_name(String full\_name) {

    this.full\_name = full\_name;

  }

  public String getTechnology() {

    return technology;

  }

  public void setTechnology(String technology) {

    this.technology = technology;

  }

  @Override

  public String toString() {

    return "Developer [sn=" + sn + ", full\_name=" + full\_name + ", technology=" + technology + "]";

  }

}

1. Create another class to instantiate and save the entity objects.
2. Create an object of the “Configuration” class, link it to the Hibernate configuration file and add the entity class to this object.
3. Create an object of the “ServiceRegistry” class that gathers all necessary configurations.
4. Create an object of the “SessionFactory“ class that manages all database interactions.
5. Open a session that opens a specific communication channel to actually perform database operations.
6. Begin a transaction because Hibernate requires the ACID properties of databases to be kept.
7. Save/ persist the Developer object
8. Commit the transaction
9. Close the session
10. Close the SessionFactory.

package com.housesearchKE.DemoHibernate;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

import org.hibernate.service.ServiceRegistry;

public class App {

    public static void main(String[] args) {

        Developer dev = new Developer("Daniel Karongo", "Java");

        // Create configuration and add annotated class

        Configuration con = new Configuration().configure().addAnnotatedClass(Developer.class);

        // Create ServiceRegistry

        ServiceRegistry reg = new StandardServiceRegistryBuilder()

                                  .applySettings(con.getProperties())

                                  .build();

        // Build SessionFactory

        SessionFactory sf = con.buildSessionFactory(reg);

        // Open a session

        Session session = sf.openSession();

        // Begin transaction

        Transaction tx = session.beginTransaction();

        // Persist the Developer object

        //        session.save(dev);      - Deprecated

        session.persist(dev);

        // Commit the transaction

        tx.commit();

        // Close the session

        session.close();

        // Close the SessionFactory

        sf.close();

    }

}

## **Annotations in Hibernate**

Hibernate provides a wide range of annotations to map Java classes to database tables, and to configure how the properties of those classes relate to columns in the database. These annotations come from the JPA (Java Persistence API) specification as well as from Hibernate-specific extensions.

Core JPA Annotations (Used in Hibernate)

1. @Entity - Marks a class as a persistent entity, meaning it will map to a table in the database.
2. @Table - Specifies the name of the database table that the entity maps to.Example: @Table(name = "developers")
3. @Id - Specifies the primary key of the entity.
4. @GeneratedValue - Specifies how the primary key should be generated (e.g., auto-increment). Example: @GeneratedValue(strategy = GenerationType.IDENTITY)
   1. GenerationType.AUTO - It's the default strategy that lets Hibernate choose the generation strategy based on the underlying database.
   2. GenerationType.IDENTITY - Uses the identity column feature in databases that support it (like MySQL). The database automatically generates a unique identifier for the primary key.
   3. GenerationType.SEQUENCE - Uses a database sequence to generate unique identifiers. It’s often used with databases like PostgreSQL and Oracle. You can specify a custom sequence using the @SequenceGenerator annotation.
   4. GenerationType.TABLE - This strategy uses a special table in the database to generate unique identifiers. It’s a fallback for databases that don’t support sequences or identity columns.
5. @Column - Specifies the name of the database column that a field maps to and its properties like nullable, length, etc. Example: @Column(name = "full\_name", nullable = false, length = 100)
6. @OneToOne - Defines a one-to-one relationship between two entities. Example: @OneToOne(mappedBy = "developer")
7. @OneToMany - Defines a one-to-many relationship, where one entity is related to multiple entities. Example: @OneToMany(mappedBy = "developer")
8. @ManyToOne - Defines a many-to-one relationship, where many entities are related to one entity. Example: @ManyToOne
9. @ManyToMany - Defines a many-to-many relationship between entities. Example: @ManyToMany(mappedBy = "developers")
10. @JoinColumn - Specifies the column that joins two entities in a relationship. Example: @JoinColumn(name = "developer\_id")
11. @JoinTable - Specifies the table used for a many-to-many relationship. Example: @JoinTable(name = "developer\_project", joinColumns = @JoinColumn(name = "developer\_id"), inverseJoinColumns = @JoinColumn(name = "project\_id"))
12. @Embedded - Used to embed a value object or another entity inside an entity.
13. @Embeddable - Marks a class whose instances are stored as an intrinsic part of the owning entity.
14. @Transient - Indicates that a field should not be persisted in the database.
15. @Lob - Marks a field for storing large objects (like BLOB or CLOB).
16. @Temporal - Specifies the date/time precision for a field (e.g., DATE, TIME, TIMESTAMP). Example: @Temporal(TemporalType.DATE)
17. @Enumerated - Specifies that a field should be persisted as an enumerated type. Example: @Enumerated(EnumType.STRING)
18. @Version - Used for optimistic locking by maintaining a version field in the entity.
19. @MappedSuperclass - Indicates that the class is a superclass whose fields are inherited by subclasses but is not an entity itself.
20. @Inheritance - Specifies the inheritance strategy for an entity hierarchy. Example: @Inheritance(strategy = InheritanceType.SINGLE\_TABLE)
21. @DiscriminatorColumn - Used in single-table inheritance to differentiate between entity types. Example: @DiscriminatorColumn(name = "entity\_type")
22. @DiscriminatorValue - Specifies the value to use for a particular entity type in the discriminator column. Example: @DiscriminatorValue("Developer")

Hibernate-Specific Annotations

1. @NaturalId - Marks a field as a natural ID, which is an alternate key that uniquely identifies a record.
2. @Formula - Allows mapping a field to a SQL expression instead of a column.
3. @Generated - Marks a property as generated by the database and not to be inserted during insert/update.
4. @Fetch - Specifies the fetching strategy (FetchType.EAGER, FetchType.LAZY).
5. @Cascade - Defines cascading of operations (e.g., save, delete) to related entities.
6. @Type - Used to specify a custom type for a field. Example: @Type(type = "org.hibernate.type.StringType")
7. @DynamicUpdate - Instructs Hibernate to generate dynamic SQL for updates, only including the changed fields.
8. @DynamicInsert - Instructs Hibernate to generate dynamic SQL for inserts, only including non-null fields.
9. @Any and @AnyMetaDef - Used to define a polymorphic association to any entity that can be stored in a specific table.
10. @Filter - Used to apply a filter condition to an entity or collection.
11. @BatchSize - Specifies the size of batches to be loaded at once for collections.
12. @Where - Adds a SQL fragment to the generated SQL for the annotated entity or collection.

## **Fetching Data using Hibernate**

The procedure for this is similar to the one for saving except for the method called on the session, I.e., “get” or “load”.

public class App {

    public static void main(String[] args) {

        Configuration con = new Configuration().configure().addAnnotatedClass(Developer.class);

        ServiceRegistry reg = new StandardServiceRegistryBuilder()

                                  .applySettings(con.getProperties())

                                  .build();

        SessionFactory sf = con.buildSessionFactory(reg);

        Session session = sf.openSession();

        Transaction tx = session.beginTransaction();

        Developer dev = session.get(Developer.class, 2);    // Takes the class and the primary key value as parameters

        tx.commit();

        session.close();

        sf.close();

        System.out.println(dev);

    }

}

## **Embeddables in Hibernate**

These are classes whose objects are saved as part of the object of another class that has been configured as an entity. For example, we may have a class called “FullName” that has “surname”, “first\_name” and “last\_name” as class attributes. Another class that is also an entity, say “Student”, may have an attribute “student\_name” whose type is the class “FullName”. This “FullName” is what is called an “Embeddable”.

package com.housesearchKE.DemoHibernate;

import jakarta.persistence.Embeddable;

@Embeddable

public class FullName {

  private String surname;

  private String first\_name;

  private String last\_name;

  public String getSurname() {

    return surname;

  }

  public void setSurname(String surname) {

    this.surname = surname;

  }

  public String getFirst\_name() {

    return first\_name;

  }

  public void setFirst\_name(String first\_name) {

    this.first\_name = first\_name;

  }

  public String getLast\_name() {

    return last\_name;

  }

  public void setLast\_name(String last\_name) {

    this.last\_name = last\_name;

  }

  // No-argument constructor (required by Hibernate)

    public FullName() {}

  public FullName(String surname, String first\_name, String last\_name) {

    super();

    this.surname = surname;

    this.first\_name = first\_name;

    this.last\_name = last\_name;

  }

  @Override

  public String toString() {

    return "FullName [surname=" + surname + ", first\_name=" + first\_name + ", last\_name=" + last\_name + "]";

  }

}

package com.housesearchKE.DemoHibernate;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table(name = "developers")

public class Developer {

  @Id

  @GeneratedValue(strategy = GenerationType.IDENTITY)

  @Column(name = "sn")

  private int sn;

  @Column(name = "full\_name")

  private FullName full\_name;

  @Column(name = "technology")

  private String technology;

  // Default constructor required by Hibernate

  public Developer() {}

  public Developer(FullName full\_name, String technology) {

    this.full\_name = full\_name;

    this.technology = technology;

  }

  public int getSn() {

    return sn;

  }

  public void setSn(int sn) {

    this.sn = sn;

  }

  public FullName getFull\_name() {

    return full\_name;

  }

  public void setFull\_name(FullName full\_name) {

    this.full\_name = full\_name;

  }

  public String getTechnology() {

    return technology;

  }

  public void setTechnology(String technology) {

    this.technology = technology;

  }

  @Override

  public String toString() {

    return "Developer [sn=" + sn + ", full\_name=" + full\_name + ", technology=" + technology + "]";

  }

}

public class App {

    public static void main(String[] args) {

      Developer dev = new Developer(new FullName("Njuguna", "Jecinta", "Wangui"), "Typescript");

      Configuration con = new Configuration().configure().addAnnotatedClass(Developer.class);

      ServiceRegistry reg = new StandardServiceRegistryBuilder()

                                .applySettings(con.getProperties())

                                .build();

      SessionFactory sf = con.buildSessionFactory(reg);

      Session session = sf.openSession();

      Transaction tx = session.beginTransaction();

      session.persist(dev);

      tx.commit();

      session.close();

      sf.close();

      System.out.println(dev);

    }

}

## **Mapping Relations In Hibernate**

Consider two entities: Laptop and Student.

Consider the following scenarios:

### **One-To-One Relationships**

One student can only have one laptop

This is a one-to-one relationship. Drawing an entity relationship diagram (ERD) would entail having a foreign key on either of the tables, probably on the Student table. In hibernate. This mapping can be done using the “@OneToOne” annotation.

package com.housesearchKE.DemoHibernate;

import jakarta.persistence.\*;

@Entity

@Table(name="laptops")

public class Laptop {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int sn;

    private String manufacturer;

    private String model;

    public int getSn() { return sn; }

    public void setSn(int sn) { this.sn = sn; }

    public String getManufacturer() { return manufacturer; }

    public void setManufacturer(String manufacturer) { this.manufacturer = manufacturer; }

    public String getModel() { return model; }

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

    public Laptop(String manufacturer, String model) {

      this.manufacturer = manufacturer;

      this.model = model;

    }

    public Laptop() {}

    @Override

    public String toString() {

        return "Laptop{" + "sn=" + sn + ", manufacturer='" + manufacturer + '\'' + ", model='" + model + '\'' + '}';

    }

}

Laptop class/entity

package com.housesearchKE.DemoHibernate;

import jakarta.persistence.\*;

@Entity

@Table(name = "students")

public class Student {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int sn;

    private FullName name;

    @Column(name = "year\_of\_study")

    private int yearOfStudy;

    private double gpa;

    @OneToOne

    private Laptop laptop;

    public Student() {

    }

    public Student(FullName name, int yearOfStudy, double gpa, Laptop laptop) {

        this.name = name;

        this.yearOfStudy = yearOfStudy;

        this.gpa = gpa;

        this.laptop = laptop;

    }

    public int getSn() { return sn; }

    public void setSn(int sn) { this.sn = sn; }

    public FullName getName() { return name; }

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

    public int getYearOfStudy() { return yearOfStudy; }

    public void setYearOfStudy(int yearOfStudy) { this.yearOfStudy = yearOfStudy; }

    public double getGpa() { return gpa; }

    public void setGpa(double gpa) { this.gpa = gpa; }

    public Laptop getLaptop() { return laptop; }

    public void setLaptop(Laptop laptop) { this.laptop = laptop; }

    @Override

    public String toString() {

        return "Student{" + "sn=" + sn + ", name=" + name + ", yearOfStudy=" + yearOfStudy + ", gpa=" + gpa + ", laptop=" + laptop + '}';

    }

Student class/entity

package com.housesearchKE.DemoHibernate;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

import org.hibernate.cfg.Configuration;

import org.hibernate.service.ServiceRegistry;

public class App {

    public static void main(String[] args) {

      Laptop laptop1 = new Laptop("HP", "840 G3");

      Laptop laptop2 = new Laptop("HP", "Pavilion");

      Student student1  = new Student(new FullName("Kungu", "Daniel", "Karongo"), 4, 64.82, laptop1);

      Student student2  = new Student(new FullName("Kungu", "Victor", "Maina"), 4, 64.82, laptop2);

      Configuration con = new Configuration().configure().addAnnotatedClass(Developer.class).addAnnotatedClass(Laptop.class).addAnnotatedClass(Student.class);

      ServiceRegistry reg = new StandardServiceRegistryBuilder()

                                .applySettings(con.getProperties())

                                .build();

      SessionFactory sf = con.buildSessionFactory(reg);

      Session session = sf.openSession();

      Transaction tx = session.beginTransaction();

      session.persist(laptop1);

      session.persist(laptop2);

      session.persist(student1);

      session.persist(student2);

      tx.commit();

      session.close();

      sf.close();

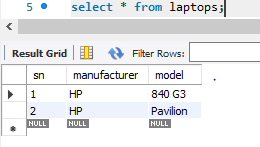
      System.out.println(student1);

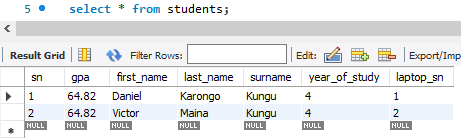
      System.out.println(student2);

    }

}

Main class





### **One-To-Many Relationships**

1. One student can have many laptops

This is a one-to-many relationship. Drawing an ERD would entail having a foreign key on the laptops table so that each laptop record can have a reference to a student record. In hibernate. This mapping can be done using the “@OneToMany” annotation on the parent table (students) alongside a “mappedBy” attribute pointed to the same annotation having the a value of the field of the child entity (Laptop).

@Entity

@Table(name="laptops")

public class Laptop {

    @ManyToOne          // Many laptops can be owned by one student. Consequently, each laptop record will have a foreign key pointing to the student who owns it.

    private Student student;

    public Laptop(String manufacturer, String model) {

        this.manufacturer = manufacturer;

        this.model = model;

    }

}

The Laptop entity

@Entity

@Table(name = "students")

public class Student {

  @OneToMany(mappedBy="student")      // So that "students" table can let the foreign key be assigned to the "laptops" table, rather than have a new composite table created.

  private List<Laptop> laptops;

  public Student(FullName name, int yearOfStudy, double gpa, List<Laptop> laptops) {

      this.name = name;

      this.yearOfStudy = yearOfStudy;

      this.gpa = gpa;

      this.laptops = laptops;

  }

  .

  .

  .

}

The student entity

public class App {

    public static void main(String[] args) {

      Laptop laptop1 = new Laptop("HP", "840 G3");

      Laptop laptop2 = new Laptop("HP", "Pavilion");

      Laptop laptop3 = new Laptop("Dell", "Inspiron 15");

      Laptop laptop4 = new Laptop("Dell", "XPS 14");

//    Passing the list of laptops when creating the Student objects, logically links the Laptop objects to the Student objects.

      Student student1  = new Student(new FullName("Kungu", "Daniel", "Karongo"), 4, 64.82, Arrays.asList(laptop1,laptop2));

      Student student2  = new Student(new FullName("Kungu", "Victor", "Maina"), 4, 64.82, Arrays.asList(laptop3,laptop4));

//    For the database to reflect this relationship properly, you must also set the Student reference in each Laptop object

      laptop1.setStudent(student1);

      laptop2.setStudent(student1);

      laptop3.setStudent(student2);

      laptop4.setStudent(student2);

      Configuration con = new Configuration().configure().addAnnotatedClass(Developer.class).addAnnotatedClass(Laptop.class).addAnnotatedClass(Student.class);

      ServiceRegistry reg = new StandardServiceRegistryBuilder()

                                .applySettings(con.getProperties())

                                .build();

      SessionFactory sf = con.buildSessionFactory(reg);

      Session session = sf.openSession();

      Transaction tx = session.beginTransaction();

      session.persist(laptop1);

      session.persist(laptop2);

      session.persist(laptop3);

      session.persist(laptop4);

      session.persist(student1);

      session.persist(student2);

      tx.commit();

      session.close();

      sf.close();

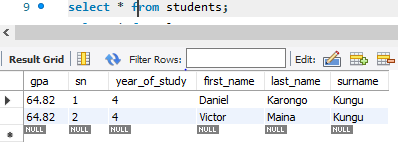
      System.out.println(student1);

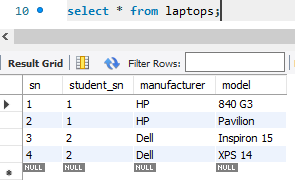
      System.out.println(student2);

    }

}

Main class





1. One laptop can be used by many students

This is the opposite of the above but still using the “@OneToMany” annotation. The “@OneToMany” annotation will be on the parent table (laptops) alongside a “mappedBy” attribute pointed to the same annotation having the a value of the field of the child entity (Student).

@Entity

@Table(name = "students")

public class Student {

    @ManyToOne      // Many students can use one laptop. So that "laptops" table can let the foreign key be assigned to the "students" table, rather than have a new composite table created.

    private Laptop laptop;

    public void setLaptop(Laptop laptop) {

        this.laptop = laptop;

    }

    public Laptop getLaptop() {

        return laptop;

    }

    public Student(FullName name, int yearOfStudy, double gpa, Laptop laptop) {

        this.name = name;

        this.yearOfStudy = yearOfStudy;

        this.gpa = gpa;

        this.laptop = laptop;

    }

}

The child (Student) entity

@Entity

@Table(name="laptops")

public class Laptop {

    @OneToMany          // One laptop can be used by many students. Consequently, each student record will have a foreign key pointing to the laptop he/she uses.

    private List<Student> students;

    public List<Student> getStudents() {

        return students;

    }

    public void setStudents(List<Student> students) {

        this.students = students;

    }

    public Laptop(String manufacturer, String model) {

        this.manufacturer = manufacturer;

        this.model = model;

    }

}

The parent (Laptop) entity

public class App {

    public static void main(String[] args) {

      Laptop laptop1 = new Laptop("HP", "840 G3");

      Laptop laptop2 = new Laptop("HP", "Pavilion");

//    Passing the list of laptops when creating the Student objects, logically links the Laptop objects to the Student objects.

      Student student1  = new Student(new FullName("Kungu", "Daniel", "Karongo"), 4, 64.82, laptop1);

      Student student2  = new Student(new FullName("Kungu", "Victor", "Maina"), 4, 64.82, laptop1);

      Student student3  = new Student(new FullName("Kamau", "Ivy", "Njeri"), 4, 64.82, laptop2);

      Student student4  = new Student(new FullName("Kamau", "Evelyne", "Waithera"), 4, 64.82, laptop2);

//    For the database to reflect this relationship properly, you must also set the Laptop reference in each Student object

      laptop1.setStudents(Arrays.asList(student1,student2));

      laptop2.setStudents(Arrays.asList(student3,student4));

      Configuration con = new Configuration().configure().addAnnotatedClass(Developer.class).addAnnotatedClass(Laptop.class).addAnnotatedClass(Student.class);

      ServiceRegistry reg = new StandardServiceRegistryBuilder()

                                .applySettings(con.getProperties())

                                .build();

      SessionFactory sf = con.buildSessionFactory(reg);

      Session session = sf.openSession();

      Transaction tx = session.beginTransaction();

      session.persist(laptop1);

      session.persist(laptop2);

      session.persist(student1);

      session.persist(student2);

      session.persist(student3);

      session.persist(student4);

      tx.commit();

      session.close();

      sf.close();

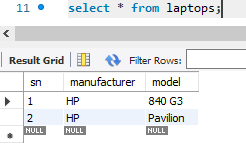
      System.out.println(student1);

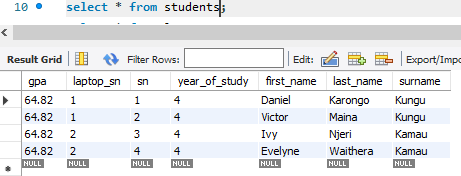
      System.out.println(student2);

    }

}

The Main class





### **Many-To-Many Relationships**

Any laptop can be used by any student/ Any student can use any laptop

This is a many to many relationship and will involve having a new table with a composite key holding all the various combinations of this relationship. However, this new table should be mapped by only one of the tables. This is done using the “@ManyToMany” annotation together with a “mappedBy” attribute. Failure to use this attribute will result in two new composite table, one mapped by each of the entities. With this kind of relationship, you should be careful about circular references leading to Stack Overflow errors, as a result of the toString() method invoked when printing out the objects involved.

@Entity

@Table(name = "students")

public class Student {

    @ManyToMany    // Any student can use any laptop. This will require a composite table to hold the various combinations.

    private List<Laptop> laptops;

    public void setLaptops(List<Laptop> laptops) {

      this.laptops = laptops;

    }

    public List<Laptop> getLaptops() {

      return laptops;

    }

    public Student(FullName name, int yearOfStudy, double gpa, List<Laptop> laptops) {

      this.name = name;

      this.yearOfStudy = yearOfStudy;

      this.gpa = gpa;

      this.laptops = laptops;

    }

    @Override

    public String toString() {

      return "Student{" +

              "sn=" + sn +

              ", name=" + name +

              ", yearOfStudy=" + yearOfStudy +

              ", gpa=" + gpa +

              ", laptop=" + (laptops != null ? laptops.size() + " laptops" : "no laptops") +

              '}';

  }

}

The Student entity

@Entity

@Table(name="laptops")

public class Laptop {

    @ManyToMany(mappedBy = "laptops")            // Any laptop can be used by any student.

    private List<Student> students;

    public List<Student> getStudents() {

      return students;

    }

    public void setStudents(List<Student> students) {

      this.students = students;

    }

    @Override

    public String toString() {

      return "Laptop{" +

              "sn=" + sn +

              ", manufacturer='" + manufacturer + '\'' +

              ", model='" + model + '\'' +

              ", students=" + (students != null ? students.size() + " users" : "no user") +

              '}';

    }

}

The Laptop entity

public class App {

    public static void main(String[] args) {

      Laptop laptop1 = new Laptop("HP", "840 G3");

      Laptop laptop2 = new Laptop("HP", "Pavilion");

      Student student1  = new Student(new FullName("Kungu", "Daniel", "Karongo"), 4, 64.82, Arrays.asList(laptop1, laptop2));

      Student student2  = new Student(new FullName("Kungu", "Victor", "Maina"), 4, 64.82, Arrays.asList(laptop1, laptop2));

      laptop1.setStudents(Arrays.asList(student1,student2));

      laptop2.setStudents(Arrays.asList(student1,student2));

      Configuration con = new Configuration().configure().addAnnotatedClass(Developer.class).addAnnotatedClass(Laptop.class).addAnnotatedClass(Student.class);

      ServiceRegistry reg = new StandardServiceRegistryBuilder()

                                .applySettings(con.getProperties())

                                .build();

      SessionFactory sf = con.buildSessionFactory(reg);

      Session session = sf.openSession();

      Transaction tx = session.beginTransaction();

      session.persist(laptop1);

      session.persist(laptop2);

      session.persist(student1);

      session.persist(student2);

      tx.commit();

      session.close();

      sf.close();

      System.out.println();

      System.out.println(student1);

      System.out.println(student2);

      System.out.println(laptop1);

      System.out.println(laptop2);

    }

}

The Main class

