Core Java Principles

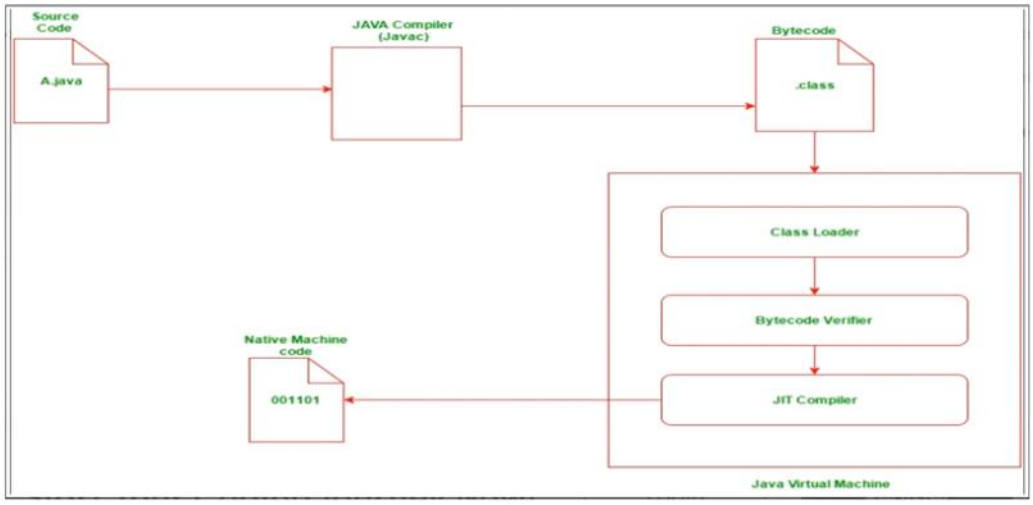
Monday, February 28, 2022 1:16 PM

- According to the project the eclipse version changes.

Core Java and Fundamentals

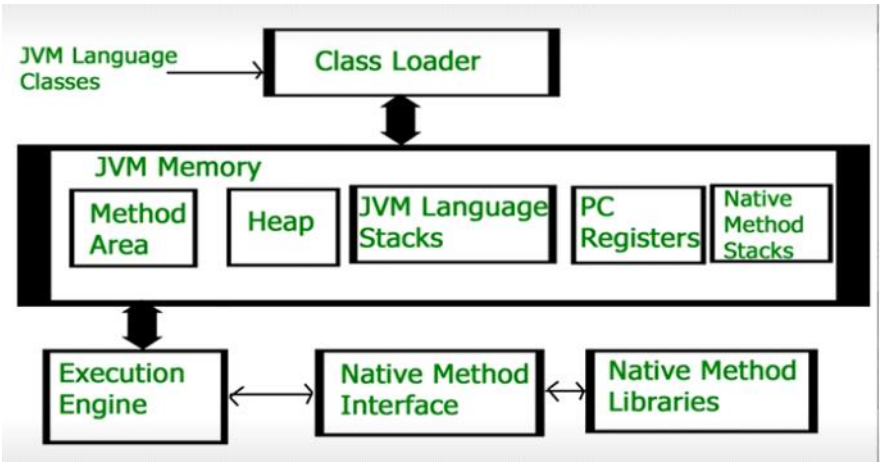
- UI -> Front end (Angular, React , JS)
- API - Application programing interface layer which is used by front end to access the database at the backend (Java, python)
- Backend -> SQL, NoSQL , Mango DB, Oracle
- In General OOP's make complex problem easier to solve and provide modularity and java also offers encapsulation of the data and behavior so that we can reuse it. And java also provide Junit testing feature and helps in trouble shooting.
- Java has write once run anywhere feature.
- In Java we have to stages to execute , Compilation and Interpretation.
- In Compilation phase we first convert the original class code to the bytecode or class code and then that bytecode can be interpreted in any operating system specific JVM. According to the operating system we will have different type of JVM's and hence that JVM convert the bytecode into the machine readable code and execute the code to produce an output. Hence we tell java is "Write once run anywhere" language. Java is platform independent language and the JVM is platform dependent.

Java execution



- Class loader file will load the classes. Three types of class loaders: Bootstrap class loader (Load the libraries) , application class loader (Load the main class) , extension class loader.
- Byte code verifier verifies the bytecode such that no compilation error occurs.
- Just in time compiler compiles the byte code into native machine code. Here machine code is system specific.

JVM Architecture

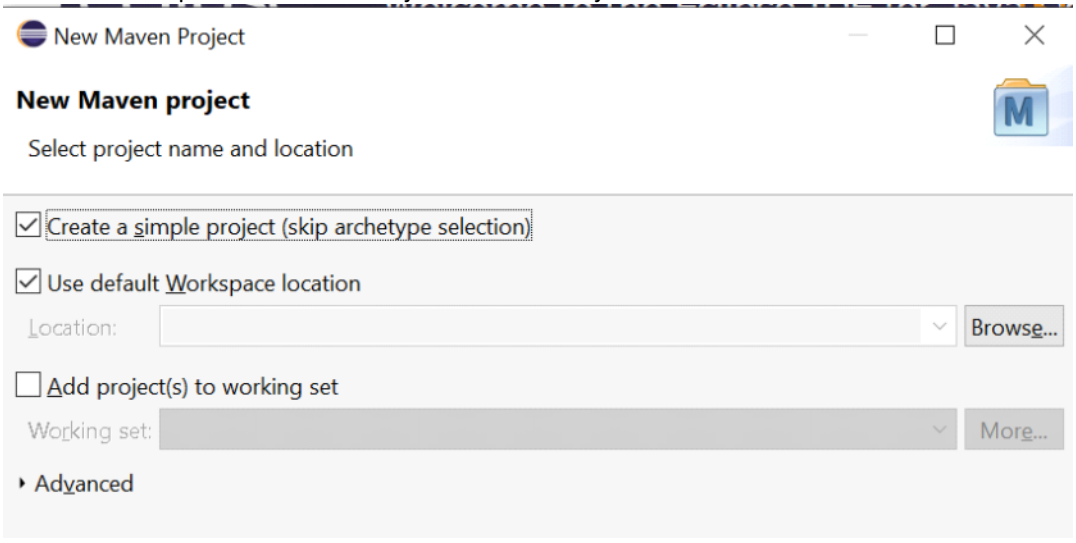


- JVM Class names , static variables , static blocks are all stored in method area of the JVM

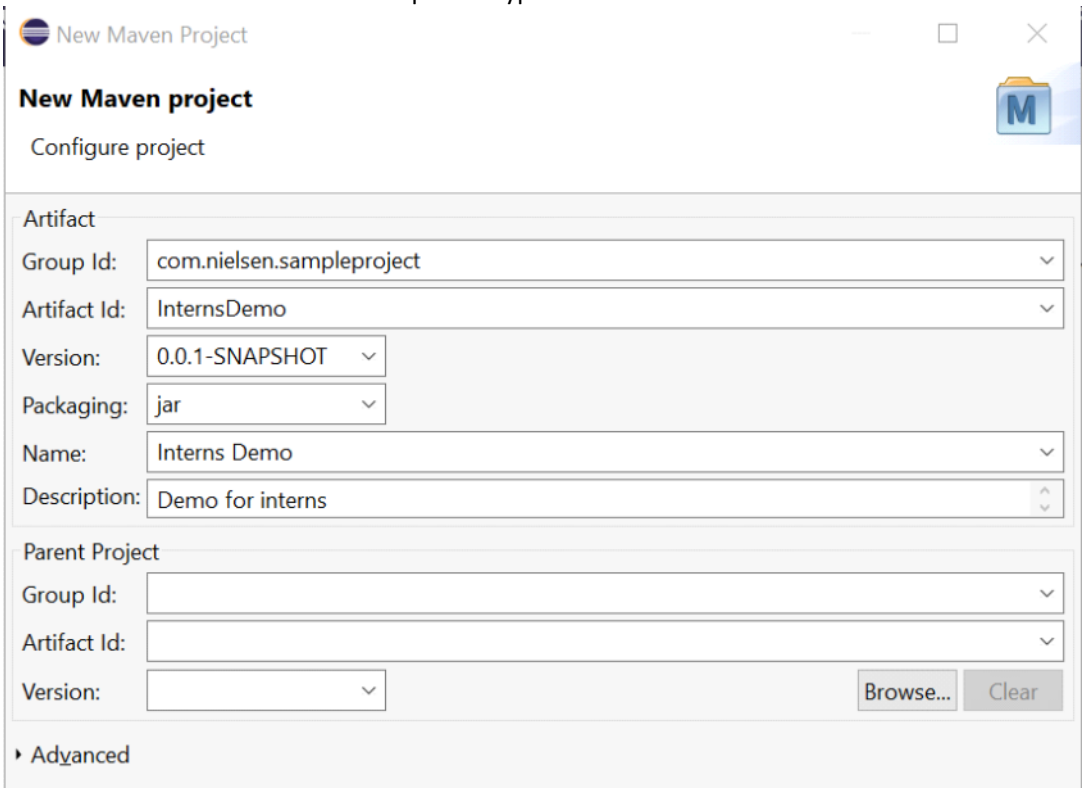
Memory.

- Heap is a shared resource, heap contains all the objects that are created.
- JVM Language stacks store the local variable such as variables that are created in threads . Here each data of each thread is independent of the data of other threads.
- PC Registers store the current instructions in the heap.
- Execution engine consists class loader , JIT compiler etc..
- Native method libraries contain some files like jre
- Eclipse is development environment for Java
- In enterprise applications we use spring framework.
- To run a spring application we write code in java. Once the code is done we build it into the jar file and send it on the server to deploy.
- Normally when we try to project in spring we must require an external server to deploy . But when we try to do a project in spring boot we no need to have a separate server to install because it is in built. For spring project's we are using tomcat server.

- Now we are creating a sample project
- Go to eclipse File > New > Project > Maven Project



- Check the first checkbox to skip archetype selection

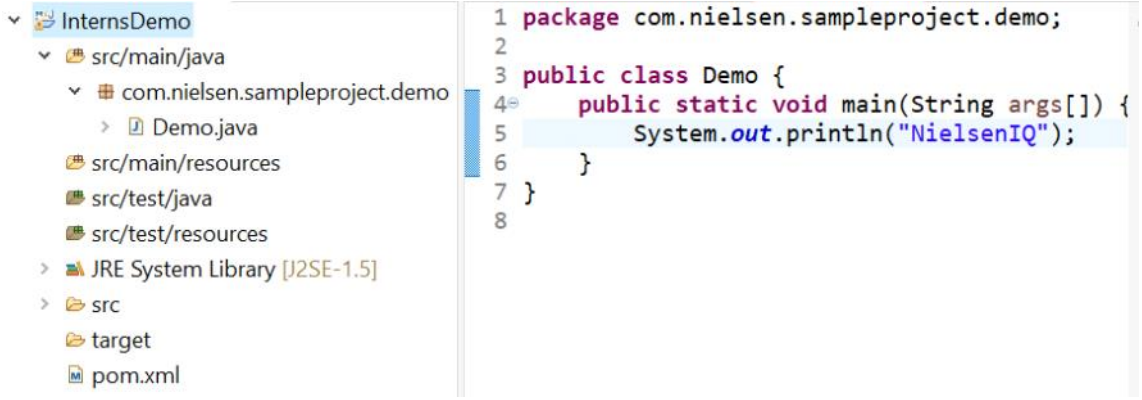


- Create a maven project in such a way
- Group id mean the path that where we want to create the project
- Artifact id is the unique id for the project

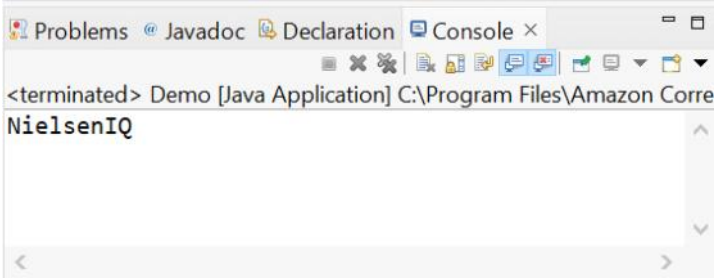
- In Packaging we can find either jar or war
- Name is the name of the project
- Description is description of the project
- We add dependencies and building the jar file in the pom.xml
- In src/main/java we are creating a package. Here we are having our main class. And the package name is (base).demo => com.nielsen.sampleproject.demo and the class name is named as Demo.java. Demo.java is the main class of this project.
- src/main/resources is for application properties
- In src/test/java we can run Junit testing.

```
  <dependencies>
  <dependency>
            <groupId>org.mapstruct</groupId>
            <artifactId>mapstruct</artifactId>
            <version>1.3.1.Final</version>
        </dependency>
  </dependencies>
<build>
  <plugins>
    <plugin>
      <!-- Build an executable JAR -->
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.1.0</version>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
            <classpathPrefix>lib/</classpathPrefix>
            <mainClass>com.nielsen.ogrds.demo.Demo</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- Here as we can see these are the build commands and maven dependencies that added into pom.xml file and if we run the application the java automatically add those dependencies in the project and execute the main code.

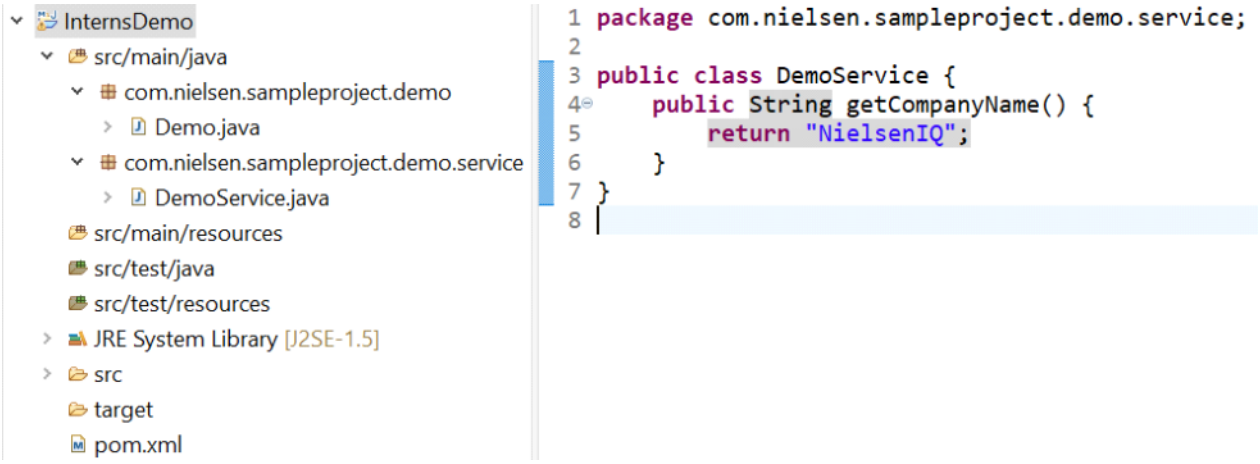


- If we want to run our project then right click on the InternsDemo project folder and click run application , then our project get's build.
- Then we get the following output



- We need to do install maven by clearing it once , so we could get a executable jar file to execute it by using command line in any other system. Hence right click on the project name > Run as > Maven Clean > Maven install and then run the java application , we get a jar file inside target folder of our project in my documents. Open command line at that path and use "java -jar InternDemo-1.0.jar" to execute jar file.
- We use this process because we need sometimes to run the app on the server or the cloud and hence we send this jar file for deployment.

Inter package communication
DemoService.java



Demo.java

- Here we are performing inter package communication. Where we are calling a function getCompany() of other class of other package to the current class. We do it through instantiation of those classes where we want to use those.

Threading

*** Thread states**

- New (Before start it will be here)
- Runnable (When it is going to start , or waiting for the resources to start)
- Blocked (Blocked by some other thread)
- Waiting (In context switching state we get this waiting state)
- Timed_Waiting (Occurs during round robin)
- Terminated (Termination state)

Main is the only thread that Is going to run in our sample program

We use runnable interface only when we want to create a thread and we want to import multi interfaces to a class.

Runnable and threads

Java Basic Codes - OOPS - Modifiers - Functional Interpretations

Monday, February 28, 2022 3:51 PM

- This is the basic hello world program of java

```
// Online Java Compiler
// Use this editor to write, compile and run your Java code online

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

- Java is multi threaded and object oriented programming. We can run two or more operations on java application.
- Variables allows to store the data in a primitive way.

Number Datatypes in Java

```
package com.nielsen.sampleproject.demo;

public class Demo{
    public static void main(String args[]) {
        int a;
        //Anything beyond this number will get
        //an error if we store it in the integer
        System.out.println("Integer Max Value: "+Integer.MAX_VALUE);

        long b;
        //If we store any value in the long it is interpreted as int
        //Hence we couldn't store the Integer.MAX_VALUE in the long unless
        //We appended L at the end of the number
        //We can store values upto Long.MAX_VALUE in the long variable
        //b=24L;
        System.out.println("Long Integer Max value: "+Long.MAX_VALUE);

        short c;
        System.out.println("Short Integer Max value: "+Short.MAX_VALUE);

        byte d;
        System.out.println("Byte Integer Max Value: "+Byte.MAX_VALUE);

        double f=12.0;
        System.out.println("Double Max Value: "+Double.MAX_VALUE);
        System.out.println("Double Max Exponent: "+Double.MAX_EXPONENT);

        float g=12.0f;
        System.out.println("Float Max Value: "+Float.MAX_VALUE);
        System.out.println("Float Max Exponent: "+Float.MAX_EXPONENT);
    }
}
```

Athematic operations

InternsDemo ▸ src/main/java ▸ com.nielsen.sampleproject.JavaBasics ▸ ArthematicOperations ▸ main(String[]) : void

```java
package com.nielsen.sampleproject.JavaBasics;

public class ArthematicOperations {
    public static void main(String[] args) {
        int a=4;
        //Addition operator
        a=a+4;
        System.out.println(a);
        //Subtraction operator
        a=a-4;
        System.out.println(a);
        //Multiplication operator
        a=a*4;
        System.out.println(a);
        //Division operator
        a=a/4;
        System.out.println(a);
        //Modulos operator
        a=a%4;
        System.out.println(a);
        //Equality operator
        System.out.println(a==4);
        //Increment operator
        a++;
        System.out.println(a);
        //Decrement operator
        a--;
        System.out.println(a);
        //If a 17.5 value returned to a integer variable then that value will be 17 in that integer variable.
        //If we want to divide two integers and get the double value as an output then we must do the following
        //The following returns the integer
        System.out.println(5/2);
        //The following returns the double
        System.out.println(5/2D);
        System.out.println(5D/2);
        //Parenthesis have higher order in execution
        //It can change the order of execution
    }
}
```

Boolean Operator

InternsDemo ▸ src/main/java ▸ com.nielsen.sampleproject.JavaBasics ▸ booleanOperators ▸ main(String[]) : void

```java
package com.nielsen.sampleproject.JavaBasics;

public class booleanOperators {
    public static void main(String[] args) {
        boolean a=false;
        boolean b=true;
        System.out.println("a = "+a);
        System.out.println("b = "+b);

        //Negation operator
        a=!a;
        b=!b;
        System.out.println("a = "+a);
        System.out.println("b = "+b);

        boolean c = true && false;
        boolean d = true || false;
        System.out.println(c);
        System.out.println(d);

        //We can use such boolean statements in the if statements
    }
}
```

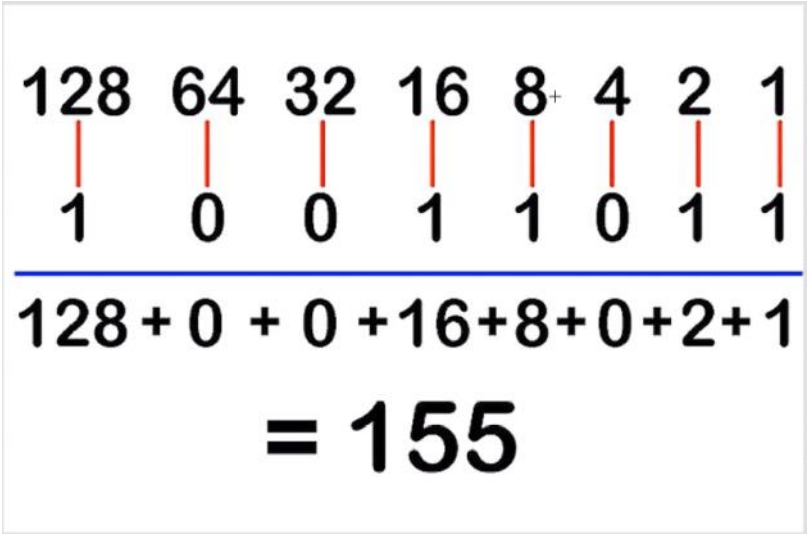
Comparing Numbers Program 1:

```java
package com.nielsen.sampleproject.JavaBasics;

public class ComparingNumbers {
    public static void main(String[] args) {
        int a=1;
        int b=0;
        if(a>b) {
            System.out.println("A is more than B");
        }
    }
}
```

Boolean Operations

```java
package com.nielsen.sampleproject.JavaBasics;

public class ComparingNumbers {
    public static void main(String[] args) {
        //Single line comment
        /* Multi Line comment*/
        /**
         * Multi
         * Line
         * Comment **/
        //Comments are ignored by the program
        boolean a = 1<2;
        //True
        System.out.println(a);
        boolean b = 1==2;
        //False
        System.out.println(b);
        boolean c = 1>2;
        //False
        System.out.println(c);
        boolean d = 1<=2;
        //True
        System.out.println(d);
        boolean e = 1!=2;
        //True
        System.out.println(e);
    }
}

```

Binary Operations



A Byte is 8 bit's

String datatype

```java
package com.nielsen.sampleproject.JavaBasics;

public class StringDatatype {
    public static void main(String[] args) {
        //A single character can be used in the character data type
        //A group of characters can be used in a string data type
        //If we store any number in the character it auto maps to the ASCII character
        //And print the respective character.
        //We could not use group of characters in a character datatype
        char simpleCharacter='a';
        System.out.println(simpleCharacter);
        simpleCharacter=97;
        System.out.println(simpleCharacter);
        //Strings are the first complex datatype
        //8 or 9 primitive datatype
        //String class is build on the character, it is a list of characters
        String sampleString="This is a String";
        //Everything is in double quotes is a string
        System.out.println(sampleString);
        //A string can be concatenated with the number and convert it to string
        int a=1;
        String a1="The number is: "+a;
        System.out.println(a1);

        //charAt function is used to retrieving the character in the string
        String indexedString="0123456789";
        System.out.println("Character at index 2 : "+indexedString.charAt(2));
        //The start index is inclusive and the end index is exclusive
        System.out.println("Substring from index 4 to 8: "+indexedString.substring(4,8));
        //Length of the String
        System.out.println("Length of the String: "+indexedString.length());
        //Equality operator for a string
        //Return true or false
        //Dont use "==" operator while we check equality
        System.out.println(indexedString.equals("0123456789"));
        System.out.println(indexedString.equals("1234567890"));

        System.out.println(indexedString.replaceAll("456", "654"));
        //The above operation doesn't affect the original content
        System.out.println(indexedString);
    }
}
```

If Else Statements

```java
package com.nielsen.sampleproject.JavaBasics;

public class IfElse {

    public static void main(String[] args) {
        int a=2;
        //The first statement we get true will execute first and the rest we leave it away
        //In these statements the operations return boolean
        //It is better to include if else rather than if and if
        if(a<2) {
            System.out.println("a is less than 2");
        }else if(a==2) {
            //We can add as many as else if statements as required
            //We must only have a single if and single else in a loop
            System.out.println("a is equal to 2");
        }else {
            System.out.println("a is more than 2");
        }
    }

}
```

Switch statements

```java
package com.nielsen.sampleproject.JavaBasics;

public class Switch {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String f="2";
        switch(f) {
        case "1":
            System.out.println("a");
            break;
        case "2":
            System.out.println("b");
            break;
        case "3":
            System.out.println("c");
            break;
        case "4":
            System.out.println("d");
            break;
        case "5":
            System.out.println("e");
            break;
        default:
            System.out.println("Enter value between 1 and 5");
        }
    }

}
```

Switch statements is only working for Integer and Enum variables not working for strings or characters

Random Values

```
package com.nielsen.sampleproject.JavaBasics;

public class RandomValues {

    public static void main(String[] args) {
        //We get values which are between 0 to 60 including float point and integer numbers inclusive
        double d=Math.random()*60;
        System.out.println("d= "+d);

        //We get random integers which are in between 0 to 60 inclusively
        int i=(int)(Math.random()*60);
        System.out.println("i= "+i);

        //We get random integers which are in between 30 and 60 inclusively
        int t= 30 + (int)(Math.random()*((60-30)+1));
        System.out.println("t: "+t);
    }

}
```

User Input

```
package com.nielsen.sampleproject.JavaBasics;
//Importing the Scanner class
import java.util.Scanner;

public class UserInput {

    private static Scanner r;

    public static void main(String[] args) {
        r = new Scanner(System.in);
        int a=(int)(Math.random()*40);
        System.out.println("Enter a Number: ");
        int b=r.nextInt();
        System.out.println(a+" + "+b+" is equal to "+(a+b));

//      String t=r.nextLine();
//      boolean y=r.nextBoolean();
//      long u=r.nextLong();
//      double i=r.nextDouble();
    }

}
```

Loops

```
package com.nielsen.sampleproject.JavaBasics;

public class Loops {
    public static void main(String[] args) {

        //While loop
        //It execute the loop body until the condition is true else it exits the loop
        int a=0;
        while(a<10){
            System.out.println(a);
            a++;
        }

        //Do while loop
        //While loop executes at least once only if the first condition is true
        //do while loop executes at least once even if the condition is false
        a=0;
        do {
            System.out.println(a);
            a++;
        }while(a<10);

        //For loop
        for(int i=0;i<10;i++) {
            System.out.println(i);
        }
        //break statement breaks the loop and get out of the loop
        //continue statement skips the iteration and go to the next iteration in the loop

    }
}
```

src/main/java

Arrays

```java
package com.nielsen.sampleproject.JavaBasics;
public class Array {
    public static void main(String[] args) {
        int[] a = new int[4];
        a[0]=12;
        a[1]=4;
        a[2]=5;
        a[3]=5;
        for(int i=0;i<a.length;i++) {
            System.out.println(a[i]);
        }

        boolean[] t=new boolean[4];
        t[0]=true;
        t[1]=false;
        t[2]=true;
        t[3]=false;
        for(int i=0;i<t.length;i++) {
            System.out.println(t[i]);
        }

        String[] s = new String[4];
        s[0]="My ";
        s[1]="Name ";
        s[2]="Is ";
        s[3]="Jeevan";
        for(int i=0;i<s.length;i++) {
            System.out.println(s[i]);
        }

        //We are going to add one element in the arr1
        int[] arr1= {1,2,3,4,5};
        int[] arr2=new int[arr1.length+1];
        for(int i=0;i<arr1.length;i++) {
            arr2[i]=arr1[i];
        }
        arr1=arr2;
        arr1[5]=5;
        for(int i=0;i<arr1.length;i++) {
            System.out.println(arr1[i]);
        }

        //We are going to have a default value in the defarr of integer type
        //For integer datatype it is 0
        //For string datatype it is null (Complex datatype)
        int[] defarr=new int[10];
        for(int i=0;i<defarr.length;i++) {
            System.out.println(defarr[i]);
        }
    }
}
```

Sort the arrays

```java
package com.nielsen.sampleproject.JavaBasics;

public class SortingArrays {

    public static void main(String[] args) {
        int[] a=new int[10];
        for(int i=0;i<10;i++) {
            a[i]=10-i;
        }
        System.out.println("Current array: ");
        for(int i=0;i<10;i++) {
            System.out.print(a[i]+" ");
        }
        System.out.println();
        for(int i=0;i<10;i++) {
            //We do the same operation 10 times
            for(int j=0;j<9;j++) {
                //In this loop we are comparing two elements and swapping them
                //in correct order and move until the end
                if(a[j]>a[j+1]) {
                    int temp=a[j];
                    a[j]=a[j+1];
                    a[j+1]=temp;
                }
            }
        }
        System.out.println("Sorted array: ");
        for(int i=0;i<10;i++) {
            System.out.print(a[i]+" ");
        }
        System.out.println();
    }
}
```

Sum of 2 Dimensional array

```java
package com.nielsen.sampleproject.JavaBasics;

public class MultiDimensionalArrays {

    public static void main(String[] args) {

        int[][] a= {{1,2,3,4,5},{1,2,3,4,5}};
        int arraySum=0;
        for(int i=0;i<a.length;i++) {
            for(int j=0;j<a[i].length;j++) {
                System.out.print(a[i][j]+" , ");
                arraySum+=a[i][j];
            }
            System.out.println();
        }
        System.out.println("Sum of the 2D array: "+arraySum);

    }

}
```

Methods

```java
package com.nielsen.sampleproject.JavaBasics;

public class Methods {
    //If we want to use this method in main method which is static we
    //must make this add1 function also static
    //After return statement no other statement executes
    static int add1(int a) {
        return a+1;
    }
    static void printArray(int[] toPrint) {
        System.out.println("Printing elements of array: ");
        for(int i=0;i<toPrint.length;i++) {
            System.out.print(toPrint[i]+" ");
        }
        System.out.println();
    }
    //public static these two are modifiers
    public static void main(String[] args) {
        System.out.println("4+1="+add1(4));
        int[] y= {1,2,3,4,5};
        printArray(y);
    }

}
```

Object Oriented Programming

- An object can have different type of attributes and functions

```java
package com.nielsen.sampleproject.JavaBasics;

class desk{
    int numberOfLegs;
    String color;
    String material;
}
public class BasicOOP {
    public static void main(String[] args) {
        //creating the object
        desk d=new desk();
        //Initializing the object attributes
        d.numberOfLegs=4;
        d.color="Light-Brown";
        d.material="Ook";
        //To organize the linear data in a meaningful way for humans
        //we use object oriented programming
        System.out.println("Number of legs: "+d.numberOfLegs);
        System.out.println("Color: "+d.color);
        System.out.println("Material: "+d.material);
        //An object oriented program consists of many well-encapsulated objects
        //and interacting with each other by sending messages
        //An object can consists of name, attributes, behaviors
    }
}
```

ModifierTypes.java

```java
package com.nielsen.sampleproject.JavaBasics;

import Utilities.ArrayUtils;

public class ModifierTypes {

    public static void main(String[] args) {
        //Here ArrayUtils is declared as public
        //hence we can access it from other package also by importing it
        ArrayUtils au=new ArrayUtils();
    }

}
```

ArrayUtils.java

```java
package Utilities;

public class ArrayUtils {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

- Here as the ArrayUtils class declared public , even if it is declared in other package. We are able to import it from other package. This modifier is public.

If we modify ArrayUtils.java and declare a method as a default modifier we get an error at the side of ModifierTypes.java

```java
package Utilities;

public class ArrayUtils {
    //default modifier
    String getName() {
        return "Jeevan";
    }
    public static void main(String[] args) {

    }

}
```

ModifierTypes.java

```java
package com.nielsen.sampleproject.JavaBasics;

import Utilities.ArrayUtils;

public class ModifierTypes {

    public static void main(String[] args) {
        //Here ArrayUtils is declared as public
        //hence we can access it from other package also by importing it
        ArrayUtils au=new ArrayUtils();
        au.getName();
    }

}
```

We get an error getName method is not reachable. And if we declare getName() as public then the error in ModifierTypes.java will be resolved.

ArrayUtils.java (We declared the method as public so we wont get any error at the side of ModifierTypes.java)

```java
package Utilities;

public class ArrayUtils {
    //public modifier
    public String getName() {
        return "Jeevan";
    }
    public static void main(String[] args) {

    }

}
```

If we declare ArrayUtils class's method as private then we could not access that method even in different class of the same package. We could only use the method or an attribute within the same class. And similarly default modifier only makes methods and attributes to visible within the same package even in different classes. Default is package private. Protected works across the same package. Public works across all packages.

Static modifier helps us to access the attributes or methods without creating the object , we can use it with the class name. This type of functionality works only within the package, it won't work in

different packages

- Here sample.java and ArrayUtils.java are both under same package called Utilities

ArrayUtils.java

```
package Utilities;

public class ArrayUtils {
    static int getAge() {
        return 21;
    }
    public static void main(String[] args) {

    }
}
```

Sample.java

```
package Utilities;

public class sample {
    public static void main(String[] args) {
        ArrayUtils.getAge();
    }

}
```

Here we are using final modifier , it mean final declares a constant where we couldn't change it any where in further code. Simply it is used to declare constant.

```
package Utilities;

public class ArrayUtils {
    final static String a="Jeevan";
    public static void main(String[] args) {
        a="Jeevan Sai";
    }
}
```

Above we can observe we are getting an error , when we modify it. So we shouldn’t modify it.

Static Methods in the same class

```
package Utilities;

public class ArrayUtils {
    //We have used static because we need to access it without instantiation
    //In general we have to call it with class name
    //But here as both methods are in the same class we no need to call even with class name
    public static int[] addToArray(int[] inputArray,int index,int toAdd) {
        inputArray[index]=toAdd;
        return inputArray;
    }
    public static void main(String[] args) {
        int[] a=addToArray(new int[3],0,45);
        System.out.println(a[0]);
    }
}
```

Encapsulation: wrapping up data under a single unit, It is a mechanism that binds the code and the data it manipulates. It is a protective shield that prevents data from being access by the code outside this shield.

In coding language, Encapsulation is noting but we should not set or retrieve any variable using assignment operator and retrieving operator "." . We need to establish setter and getter functions for everything. And in other words, making content's not able to manipulate directly from outside of the class.

```java
package com.nielsen.sampleproject.JavaBasics;

class TV{
    private ScreenSizeCLS screenSize;
    //Here we have made the screenSize variable as private
    //This variable is not accessible out of this class
    //So it manipulate the variable
    //we have to utilize already existing functions that we have created
    //For example we have to use setter and getter functions
    public void setScreenSize(int PassedInScreenSize) {
        if(PassedInScreenSize<100 && PassedInScreenSize!=50) {
            ScreenSizeCLS t=new ScreenSizeCLS();
            t.setSize(PassedInScreenSize);
            screenSize=t;
        }
    }
    public int getScreenSize() {
        return screenSize.getSize();
    }
}
class ScreenSizeCLS{
    private int size;
    public int getSize() {
        return size;
    }
    public void setSize(int size) {
        this.size = size;
    }
    String dateOfCreation;
}
public class Encapsulation {

    public static void main(String[] args) {
        TV myTV=new TV();
        myTV.setScreenSize(12);
        System.out.println(myTV.getScreenSize());
    }

}
```

Abstraction

```java
package com.nielsen.sampleproject.JavaBasics;

//Abstract classes are the class where we will have un implemented methods even body is not declared
//Abstract classes will have abstract methods
//If we want not to declare the body of a method we can declare abstract classes
//Where abstract classes are used.
//Suppose for example if we have a situation of generating salary for employees who work on monthly basis
//or hourly basis. if we want to create a function for that we need to create two independent classes , which take
//so much of computation memory. Hence instead of that we will create a single abstract class
//use that definition to create individual methods of generating salary
//Abstract class
abstract class computer{
    //abstract method
    public abstract String getSerialNumber();
}
public class Abstraction {
    public static void main(String[] args) {

    }
}
```

Without abstraction

```java
package com.nielsen.sampleproject.JavaBasics;

class MonthlyEmployee{
    private int salary;
    public int getSalary() {
        return this.salary;
    }
    public void setSalary(int Salary) {
        this.salary=Salary;
    }
}
class HourlyEmployee{
    private int salary;
    private int hoursWorked;
    public int getSalary() {
        return this.salary*hoursWorked;
    }
    public void setSalary(int Salary,int hoursWorked) {
        this.salary=Salary;
        this.hoursWorked=hoursWorked;
    }
}
public class Abstraction {
    public static void main(String[] args) {

    }
}
```

With Abstraction (We can prevent repetition of the code and establish user readability)

```java
package com.nielsen.sampleproject.JavaBasics;
abstract class employee{
    protected int salary;
    public abstract void setSalary(int Salary);
    public abstract int getSalary();
}
class MonthlyEmployee extends employee{
    @Override
    public void setSalary(int Salary) { this.salary=Salary; }
    @Override
    public int getSalary() { return salary; }
}
class HourlyEmployee extends employee{
    private int numberOfHoursWorked;
    public int getNumberOfHoursWorked() { return this.numberOfHoursWorked;  }
    public void setNumberOfHoursWorked(int numberOfHoursWorked) { this.numberOfHoursWorked = numberOfHoursWorked; }
    @Override
    public void setSalary(int Salary) { this.salary=Salary; }
    @Override
    public int getSalary() { return this.numberOfHoursWorked*this.salary; }
}
public class Abstraction {
    public static void main(String[] args) {
        //We couldn't do the following instantiation because employee is an abstract class , We can create an object for it's extended classes
        //employee e=new employee();
        HourlyEmployee e=new HourlyEmployee();
        e.setNumberOfHoursWorked(15);
        e.setSalary(15);
        System.out.println("Hourly Employee Salary: "+e.getSalary());
        MonthlyEmployee m=new MonthlyEmployee();
        m.setSalary(1500);
        System.out.println("Monthly Employee Salary: "+m.getSalary());
    }
}
```

For the above example we can even create employee e=new HourlyEmployee(); because it makes an is-a relationship. "HourlyEmployee is-a employee".

Inheritance

```java
package com.nielsen.sampleproject.JavaBasics;
class Size2D{
    int x;
    int y;
}
class Size3D extends Size2D{
    //Size3D inherits Size2D and it's attributes x and y
    int z;
}
public class Inheritance {
    public static void main(String[] args) {
        Size3D t = new Size3D();
        t.x=12;
        t.y=13;
        t.z=14;
        System.out.println("x= "+t.x+" y= "+t.y+" z= "+t.z);
        //Here we are checking instance of Size 3D with Size3D and Size2D
        if(t instanceof Size3D) {//True
            System.out.println("t is instance of Size3D");
        }
        if(t instanceof Size2D) {//True
            System.out.println("t is instance of Size2D");
        }
        //t is an instance which is created on Size3D and that class extends Size2D
        //Hence both the above statements are true
        //Let's create an instance for Size2D
        Size2D u = new Size2D();
        if(u instanceof Size3D) { //False
            System.out.println("u is instance of Size3D");
        }
        if(u instanceof Size2D) { //True
            System.out.println("u is instance of Size2D");
        }
        //From above we can get the following output
        //t is instance of Size3D
        //t is instance of Size2D
        //u is instance of Size2D
        Size2D v = new Size3D();
        //Mapping of bigger object into smaller object is valid
        //The right side thing is what matters in creation of object, the left side is just mapping of right to the left
        if(v instanceof Size2D) { //True
            System.out.println("v is instance of Size2D");
        }
        if(v instanceof Size3D) { //True
            System.out.println("v is instance of Size3D");
        }
    }
}
```

Here if we want to manipulate any variable which is declared in current as well as in the parent class with different datatypes, we need to use super keyword to modify the variable which is in the other class . And if we want to modify the variable in current class we need to use only this keyword.

- Super keyword is used to manipulate variable in the parent class
- This keyword is used to manipulate variable in current class

```java
package com.nielsen.sampleproject.JavaBasics;
class Size1D{
    int x;
}
class Size2D extends Size1D{
    int y;
}
class Size3D extends Size2D{
    String y;

    public String getY() {
        return y;
    }
    public void setY(int y) {
        super.y=y;
    }
    public void setY(String y) {
        this.y = y;
    }
}
public class Inheritance {
    public static void main(String[] args) {
        Size3D t=new Size3D();
    }
}
```

Constructor declaration with inheritance and variable passing

Method 1: (General way)

```java
package com.nielsen.sampleproject.JavaBasics;
class Size1D{
    int x;
}
class Size2D extends Size1D{
    int y;
}
class Size3D extends Size2D{
    int z;
    // constructor doesn't have any return type and need not to mention any modifier (public
    public Size3D(int x,int y,int z) {
        this.x=x;
        this.y=y;
        this.z=z;
    }


}
public class Inheritance {
    public static void main(String[] args) {
        Size3D t=new Size3D(1,2,3);
        System.out.println("x= "+t.x+" y= "+t.y+" z= "+t.z);
    }
}
```

Method 2: (Efficient way)

```java
package com.nielsen.sampleproject.JavaBasics;
class Size1D{
    int x;
    public Size1D(int x) {
        this.x=x;
    }
}
class Size2D extends Size1D{
    int y;
    public Size2D(int x,int y) {
        super(x);
        this.y=y;
    }
}
class Size3D extends Size2D{
    int z;
    // constructor doesn't have any return type and need not to mention any modifier (public
    public Size3D(int x,int y,int z) {
        super(x,y);
        this.z=z;
    }


}
public class Inheritance {
    public static void main(String[] args) {
        Size3D t=new Size3D(1,2,3);
        System.out.println("x= "+t.x+" y= "+t.y+" z= "+t.z);
    }
}
```

Here bill object is showing polymorphism with Object and deer class

```java
package com.nielsen.sampleproject.JavaBasics;
class deer{

}
//Ability of an object to take many shapes
public class Polymorphism {
    public static void main(String[] args) {
        deer bill=new deer();
        Object t=new Object();
        if(bill instanceof deer) {
            System.out.println("Bill is instance of Deer");
        }
        if(bill instanceof Object) {
            System.out.println("Bill is instance of Object");
        }
    }
}
```

Polymorphism

```java
package com.nielsen.sampleproject.JavaBasics;
class vegetarian{
    private int years;
    public int getYears() {
        return years;
    }
    public void setYears(int years) {
        this.years=years;
    }
}
class deer extends vegetarian{
    private int age;
    private String color;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age=age;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public String toString() {//We can use direct variables instead of methods below
        return "Age = "+getAge()+" Color = "+getColor()+" Years = "+getYears();
    }
}
//Ability of an object to take many shapes
public class Polymorphism {
    public static void main(String[] args) {
        deer bill=new deer();
        bill.setAge(13);
        bill.setColor("Brown");
        bill.setYears(13);//In general the following line is used to print to the user, but it is visible to the user hence we have to make it invisible
        System.out.println("Age = "+bill.getAge()+" Color = "+bill.getColor()+" Years = "+bill.getYears());
        //In general toString function returns the address of the object , But we are re modifying it using @Override decoration
        System.out.println(bill.toString());
    }
}
```

Graphical User Interface

FinalProjectCode.java

```
package FinalProject;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class FinalProjectCode {
	static User[] UserArray = new User[100];
	static int currentUser=0;
	public static void main(String[] args) {
		JFrame f = new JFrame();
		f.setSize(200,300);

		final JTextField NameTextField = new JTextField();
		final JTextField UserNameTextField = new JTextField();
		final JTextField PasswordTextField = new JTextField();
		final JButton SignUpButton = new JButton();
		SignUpButton.setText("Sign Up");
		final JButton LogInButton = new JButton();
		LogInButton.setText("Log IN");

		final JLabel NameLabel = new JLabel("Name: ");
		final JLabel UserNameLabel = new JLabel("Username: ");
		final JLabel PasswordLabel = new JLabel("Password: ");


		f.setLayout(new GridLayout(4,2));

		f.add(NameLabel);
		f.add(NameTextField);

		f.add(UserNameLabel);
		f.add(UserNameTextField);

		f.add(PasswordLabel);
		f.add(PasswordTextField);

		f.add(LogInButton);
		f.add(SignUpButton);

		LogInButton.addActionListener(new ActionListener() {

	public void actionPerformed(ActionEvent e) {
		for(int i=0;i<UserArray.length;i++) {
			if(UserArray[i]!=null &&
			UserArray[i].getUsername().equals(UserNameTextField.getText())) {
				if(UserArray[i].getPassword().equals(PasswordTextField.getText())) {
					System.out.println("Logged In");
					NameTextField.setVisible(false);
					UserNameTextField.setVisible(false);
					PasswordTextField.setVisible(false);

					SignUpButton.setVisible(false);
					LogInButton.setVisible(false);

					NameLabel.setText("Welcome, "+UserArray[i].getName());
					UserNameLabel.setVisible(false);
					PasswordLabel.setVisible(false);

					break;
				}
			}
		}
	}

	});

	SignUpButton.addActionListener(new ActionListener() {
```

```
	public void actionPerformed(ActionEvent e) {
		User userToAdd = new
		User(NameTextField.getText(),UserNameTextField.getText(),PasswordTextField.getText
		());
		boolean UserExists = false;
		for(int i=0;i<UserArray.length;i++) {
			if(UserArray[i]!=null && UserArray[i].getName().equals(userToAdd.getName())) {
				UserExists=true;
			}
		}
		if(UserExists==false) {
		UserArray[currentUser]=userToAdd;
		currentUser+=1;
		System.out.println("Added new user with name "+userToAdd.getName());
		}else {
			System.out.println("User Aldready Exists!");
		}
	}

	});

	f.setVisible(true);
	}
}
```

User.java

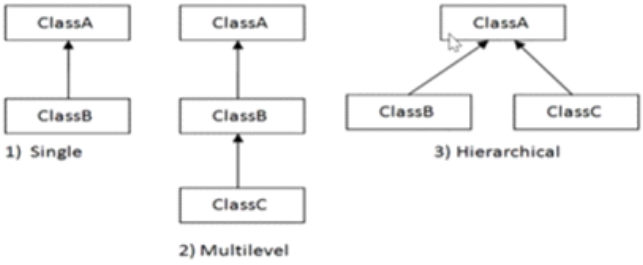
```
package FinalProject;

public class User {
	private String Name;
	private String Username;
	private String Password;
	public String getName() {
		return Name;
	}
	public void setName(String name) {
		Name = name;
	}
	public User(String name, String username, String password) {
		super();
		Name = name;
		Username = username;
		Password = password;
	}
	public String getUsername() {
		return Username;
	}
	public void setUsername(String username) {
		Username = username;
	}
	public String getPassword() {
		return Password;
	}
	public void setPassword(String password) {
		Password = password;
	}
}
```

Output

Inheritance: inheritance represents the IS-A relationship which is also known as a parent-child relationship. Parent and child have some common characteristics which are going to mention in parent class and special child characteristics are going to mention in the child class. The class which extends is derived class.

Java supports only

1) Single inheritance
2) Multilevel Inheritance
3) Hierarchical Inheritance



- **Single and multilevel combinedly forms the hierarchical inheritance.**
- **Aggregation: If a class have an entity reference it is known as aggregation. It represents Has - A relationship.**

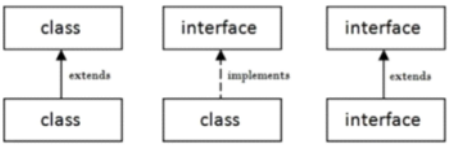
```
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name,Address address) {
    this.id = id;
    this.name = name;
    this.address=address;
}

void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}

public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");
Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);
e.display();
e2.display();
}
}

public class Address {
String city,state,country;
public Address(String city, String state, String country) {
    this.city = city;
    this.state = state;
    this.country = country;
}}
```

Here Employee class have Address class instance in it. It means Employee "has-a" Address relationship.

Interface vs Abstract class

- It is believed that abstraction is implemented 100% in interface but not in Abstract class
- After Java 8 interfaces also can't provide 100% abstraction.

- The main difference is:
    - In interface we must have public static final variables
    - In abstract we can have any variable.



- We must implement the un implemented method if we implements a interface from a class or else we would get an error.

Interface Example 1:

```
interface printable{
void print();
}
class A implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A obj = new A();
obj.print();
 }
}
```

Important points to remember:

- Member variables in interface are strictly public static final.
- Member functions are public by default

New feature of JAVA 8 : We can implement methods inside the interface, but only using "default" or "static" modifier.

Default method in Interface

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

Static method in Interface

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Note: Static methods cannot be overriden like default methods.

Abstract classes

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

- Here we could not instantiate abstract class directly because abstract class will not have body for some methods, so after extending from other class. We declare the body of such type of functions and instantiate the normal class and utilize the abstract methods through this.

Abstract classes

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

Java Inner Class: It is a way of logically grouping classes that are only used in one place. And it increases encapsulation. It increases readable and maintainable code.

Types of inner class

- Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class
- Static nested class

Member Inner Class

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Anonymous Inner Class

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

We use anonymous inner class when we are working with abstract or interface , in general we extends the Person abstract class by the child class and declare a function and create another class and create a instance of the child class and execute the function that we declared. Here we used three classes but in general if we use anonymous inner class then we can do the same work. We can skip the child class and do it in two classes in whole. Anonymous inner class doesn't have instantiation.

Local inner class: These are the inner classes that are defined inside the method under the main class

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Static nested class

A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```

In Java 8 : Functional Interfaces, For each loop and Streams concepts are introduced

- Every functional interface must only have only 1 abstract method.
- Functional Interface is denoted using annotation.

Lambda Expression example

```
Integer[] num = {1,2,3,4,5};
List<Integer> numbers = Arrays.asList(num);

numbers.forEach(x -> System.out.println(x));
```

Functional Interface:

- Interface that contains exactly one abstract method is known as functional interface
- It can have any number of default, static methods but can contain only one abstract method.
- It can also declare methods of object class.
- It is also known as Single Abstract Method interfaces

Method 1:

```
package functionalInterfaces;

@FunctionalInterface
interface sayable{
    void say(String msg);
}
public class funcInterface implements sayable{
    public void say(String msg) {
        System.out.println(msg);
    }
    public static void main(String[] args) {
        funcInterface fie = new funcInterface();
        fie.say("Hello Jeevan");
    }
}
```

When coming to have abstract class methods, we must have only one abstract class in functionalInterface but we can even have any number of Object Class methods similar to abstract but they are not abstract in nature.

```java
package functionalInterfaces;

@FunctionalInterface
interface sayable{
    void say(String msg);//abstract method
    //It can contain any number of Object class methods
    //The following methods are object class methods
    int hashCode();
    String toString();
    boolean equals(Object obj);
}
public class funcInterface implements sayable{
    public void say(String msg) {
        System.out.println(msg);
    }
    public static void main(String[] args) {
        funcInterface fie = new funcInterface();
        fie.say("Hello Jeevan");
    }
}
```

When a functional interface extending a non functional interface

```java
package functionalInterfaces;

interface Doable{
    default void doIt() {
        System.out.println("Do it now");
    }
}
@FunctionalInterface
interface sayable extends Doable{
    void say(String msg);//abstract method
    //It can contain any number of Object class methods
    //The following methods are object class methods
    int hashCode();
    String toString();
    boolean equals(Object obj);
}
public class funcInterface implements sayable{
    public void say(String msg) {
        System.out.println(msg);
    }
    public static void main(String[] args) {
        funcInterface fie = new funcInterface();
        fie.say("Hello Jeevan");
        fie.doIt();
    }
}
```

Functional Interface

- Predicate => returns Boolean result

```java
package functionalInterfaces;

import java.util.function.Predicate;

public class predicate {
    //Functional interface types -- 4 types
    // 1 ) Predicate -- returns boolean result
    public static void main(String[] args) {
        Predicate<String> checkLength = str -> str.length()>5;
        //Check if length of the string more than 5 then return true else false
        System.out.println(checkLength.test("abcd"));
        System.out.println(checkLength.test("abcde"));
        System.out.println(checkLength.test("abcdef"));
    }
}
```

- Consumer => only consumes the data and no result returned

```java
package functionalInterfaces;

import java.util.function.Consumer;

class Person{
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name=name;
    }
}
public class consumer {
    //Functional Interface types -- 4 types
    //Consumer -- modifies data -- no output
    public static void main(String[] args) {
        //We are creating an instance for the person class
        Person p = new Person();
        //Creating a lambda expression and passing it into the consumer function interface
        Consumer<Person> setName = t -> t.setName("Play Java");
        //Consumer functional interface accepts the data but doesnot return anything
        setName.accept(p);
        //So we created our own function and retrieved the data
        System.out.println(p.getName());
    }
}
```

Here we are passing a function as a method reference

```java
// Using method reference for functional interface
Consumer<Integer> show = Practice::show;
numbers.forEach(show);
```

We can even use an anonymous function to create consumer body for the functional interface

```java
// Using anonymous class to implement functional interface
Consumer<Integer> show = new Consumer<Integer>() {

    @Override
    public void accept(Integer t) {
    System.out.println(t*3);

    }
};
numbers.forEach(show);
```

Instead of this boiler plate code we can even use the lambda function

```java
numbers.forEach((t)-> System.out.println(t*3));
```

- forEach method follows the consumer interface
- In String we use filter where we use predicates

- Function => Function takes input and returns output

```java
package functionalInterfaces;

import java.util.function.Function;

public class Functions {
    public static void main(String[] args) {
        //Function functional interface accepts input and generate output
        Function<Integer,String> getInt = t -> t*10 + " is the data we get when "+t+" multiplied by 10";
        System.out.println(getInt.apply(2));
    }
}
```

- Supplier => no input but only supply the data

Java 8 Streams

Friday, March 4, 2022 9:50 PM

- Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, interfaces and Enum to allows functional-style operations on the elements
- Operations performed on a stream does not modify it's source.
- You can use stream to filter, collect, print, and convert from one data structure to other etc.

Filter Stream Method

```
package Streams;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

class Product{
    int id;
    String name;
    float price;
    public Product(int id,String name,float price) {
        this.id=id;
        this.name=name;
        this.price=price;
    }
}
public class Filters {

    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));

        //we are passing the data of product stream into the stream and applying filter using filter function
        //and for that data we are fetching the each price of the productsList and filtering it later using collect
        //function we collect all the data that we filtered and convert it into the list and pass it in productPriceList2
        //first creating filter condition then fetching the data one by one then collecting filtered data and storing it
        List<Float> productPriceList2 = productsList.stream().filter(p->p.price>30000).map(p->p.price).collect(Collectors.toList());

        System.out.println(productPriceList2);
    }

}
```

Output:

```
[90000.0]
```

Iterate Stream method

```
package Streams;

import java.util.stream.Stream;

public class Iterations {

    public static void main(String[] args) {
        Stream.iterate(1, element -> element + 1) //Create a stream where we nee to iterate from 1 to infinite with increment 1
        .filter(element -> element % 5 == 0) //But you need to filter elements which are divided by 5
        .limit(5) //Generate 5 such numbers and stop iteration
        .forEach(System.out::println); //And for each of the filtered element we need to print it
    }

}
```

Filter Iterate Stream Method

```java
package Streams;

import java.util.ArrayList;
import java.util.List;


public class FilterAndIterate {

    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));

        productsList.stream()
        .filter(product -> product.price ==30000)
        .forEach(product -> System.out.println(product.name));
    }

}
```

Console ×

<terminated> FilterAndIterate [Java Application] C:\Program Files\Amazon Corr

```
Dell Laptop
```

Reduce() method

```java
package Streams;

import java.util.ArrayList;
import java.util.List;

public class Reduces {

    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // This is more compact approach for filtering data
        float totalPrice = productsList.stream()
                .map(product->product.price)
                .reduce(0.0f, (sum,price)->sum+price);
        System.out.println(totalPrice);
        //Another approach of accumulating sum
        float totalPrice2 = productsList.stream()
                .map(product->product.price)
                .reduce(0.0f, Float::sum);// accumulating price, by referring method of Float class
        System.out.println(totalPrice2);

    }

}
```

Output:

```
201000.0
201000.0
```

As we are using product class multiple times declare that class as a separate file

```java
package Streams;


class Product{
    int id;
    String name;
    float price;
    public Product(int id,String name,float price) {
        this.id=id;
        this.name=name;
        this.price=price;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public float getPrice() {
        return price;
    }
}
```

Sum using collectors

```java
package Streams;
import java.util.List;
import java.util.stream.Collectors;
import java.util.ArrayList;

public class SumUsingCollectors {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));

        //Using collector's method to sum the prices
        double totalPrice=productsList.stream().collect(Collectors.summingDouble(product->product.price));
        System.out.println(totalPrice);
    }
}
```

Output:

```
201000.0
```

Min Max using streams

```java
package Streams;
import java.util.List;
import java.util.ArrayList;
public class MaxMinStreams {

    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));

        //max() method to get max product based on price
        Product productA = productsList.stream().max((product1,product2)-> product1.price>product2.price ? 1: -1).get();
        System.out.println(productA.id+" , "+productA.name+" , "+productA.price);

        //min() method to get min Product based on price
        Product productB = productsList.stream().min((product1 , product2) -> product1.price>product2.price ? 1:-1).get();
        System.out.println(productB.id+" , "+productB.name+" , "+productB.price);
    }

}
```

Output:

```
90000.0
25000.0
```

Count In Streams

```
package Streams;
import java.util.List;
import java.util.ArrayList;

public class CountInStreams {

    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));

        //count number of products based on filter
        long count = productsList.stream()
                .filter(product->product.price<30000)
                .count();
        System.out.println(count);
    }

}
```

Output:

```
3
```

Convert list into set

ListToSet.java

InternsDemo ▸ src/main/java ▸ Streams ▸ ListToSet ▸ main(String[]) : void

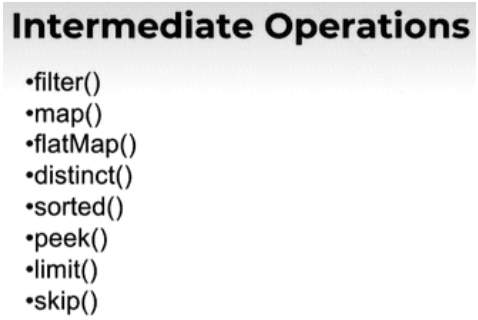
```
1  package Streams;
2  import java.util.List;
3  import java.util.Set;
4  import java.util.stream.Collectors;
5  import java.util.ArrayList;
6  public class ListToSet {
7      public static void main(String[] args) {
8          List<Product> productsList = new ArrayList<Product>();
9
10         //Adding Products
11         productsList.add(new Product(1,"HP Laptop",25000f));
12         productsList.add(new Product(2,"Dell Laptop",30000f));
13         productsList.add(new Product(3,"Lenevo Laptop",28000f));
14         productsList.add(new Product(4,"Sony Laptop",28000f));
15         productsList.add(new Product(5,"Apple Laptop",90000f));
16
17         //Convert productList into Set based on filter
18         Set<Float> productPriceList = productsList.stream()
19                 .filter(product->product.price<30000) //Filter product based on price
20                 .map(product -> product.price) //we are mapping to collect the result
21                 .collect(Collectors.toSet()); //We are collecting the result after converting the result to the set
22         System.out.println(productPriceList);
23     }
24 }
25
```

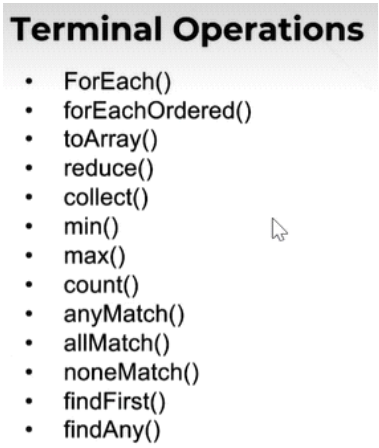
Console

```
<terminated> ListToSet [Java Application] C:\Program Files\Amazon Corretto\jdk16.0.2_7\bin\javaw.exe  (Mar 5, 2022, 12:29:33 PM – 12:2
[25000.0, 28000.0]
```

Method Reference

```java
package Streams;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class MethodReference {

    public static void main(String[] args) {

        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList = productsList.stream()
                .filter(p->p.price>30000)
                .map(Product::getPrice)
                .collect(Collectors.toList());
        System.out.println(productPriceList);
    }

}
```



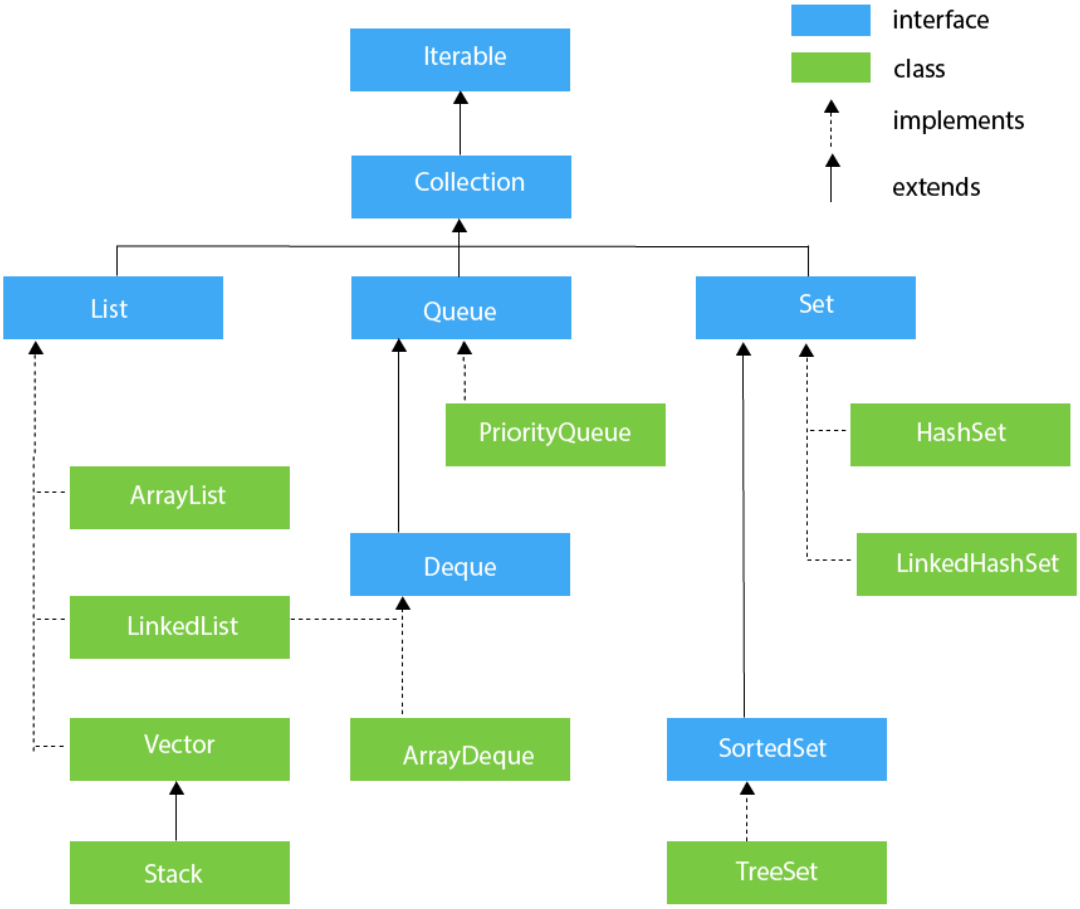


- Intermediate operator doesn't return anything, so for every intermediate operation there must be terminal operation to make a change of the original.
- There are two types stream and parallel stream, stream maintains the order and parallel stream doesn't maintain any order.
- When we directly print the object using System.out.println() we would get a hash code but if we override the toString function in the class we would get the overridden sentence instead of hash code.
- Concurrent Hash Map is thread safe it ensures synchronous operations

Java Collections

Saturday, March 5, 2022 2:35 PM

- **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects. Can perform operations such as searching , sorting , insertion , manipulation and deletion. Provides many interfaces (Set , List ,Queue, Deque) and classes (ArrayList , Vector , LinkedList, PriorityQueue , HashSet , LinkedHashSet, TreeSet)
- Framework mean it provides a readymade architecture , represents a set of classes and interfaces.



List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

ArrayList

- Can store duplicate elements of different data types.
- It can maintains the insertion order and is non synchronized.
- Can be accessed randomly.

```java
package collection;

import java.util.ArrayList;

public class ArrayLists {

    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Ravi");//Adding object in arraylist
        list.add("Vijay");
        list.add("Ravi");
        list.add("Ajay");

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

Console ×

<terminated> ArrayLists [Java Application] C:\Program Files\Amazon C

```
Ravi
Vijay
Ravi
Ajay
```

LinkedList

- Internally implements Doubly linked list.
- Can store duplicate elements.
- Insertion order is maintained and not synchronized.
- Manipulations are fast.

```java
package collection;
import java.util.Iterator;
import java.util.LinkedList;

public class LinkedLists {
    public static void main(String[] args) {

        LinkedList<String> al=new LinkedList<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");

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

Console ×

<terminated> LinkedLists [Java Application] C:\Program Files\Amazo

```
Ravi
Vijay
Ravi
Ajay
```

Vector

- Similar to ArrayList
- Synchronized and contains many methods that are not part of collection framework.

```java
package collection;
import java.util.Iterator;
import java.util.Vector;

public class Vectors {
    public static void main(String[] args) {

        Vector<String> v=new Vector<String>();
        v.add("Ayush");
        v.add("Amit");
        v.add("Ashish");
        v.add("Garima");

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

Console ×

<terminated> Vectors [Java Application] C:\Program Files\

```
Ayush
Amit
Ashish
Garima
```

Stack

- Subclass of Vector
- Implement's last in first our data structure
- Provide methods of vector class

```java
import java.util.Iterator;
import java.util.Stack;
public class Stacks {
    public static void main(String[] args) {

        Stack<String> stack = new Stack<String>();
        stack.push("Ayush");
        stack.push("Garvit");
        stack.push("Amit");
        stack.push("Ashish");
        stack.push("Garima");
        stack.pop();  //Last in first out

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

Console ×

<terminated> Stacks [Java Application] C:\Program Files\Amaz

```
Ayush
Garvit
Amit
Ashish
```

Queue and Deque Interface

- First in First out order
- There are various classes like Priority Queue , Deque and Array Deque which implements the Queue interface.

```java
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

Priority Queue

- Implements the queue interface
- It holds the elements or objects which are to be processed by their priorities
- Doesn't allow null values to store in the queue.

```java
package collection;

import java.util.Iterator;
import java.util.PriorityQueue;

public class PriorityQueues {
    public static void main(String[] args) {

        PriorityQueue<String> queue=new PriorityQueue<String>();
        queue.add("Amit Sharma");
        queue.add("Vijay Raj");
        queue.add("JaiShankar");
        queue.add("Raj");

        System.out.println("head:"+queue.element());  //Displays last element
        System.out.println("head:"+queue.peek()); //Displays last element

        System.out.println("iterating the queue elements:");
        Iterator<String> itr=queue.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }

        System.out.println("Removed: "+queue.remove());  //Removes first element
        System.out.println("Removed: "+queue.poll());  //Removes first element

        System.out.println("after removing two elements:");
        Iterator<String> itr2=queue.iterator();
        while(itr2.hasNext()){
            System.out.println(itr2.next());
        }
    }
}
```

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
Removed: Amit Sharma
Removed: JaiShankar
after removing two elements:
Raj
Vijay Raj
```

ArrayDeque

- Implement deque interface
- Can add or delete elements from both sides
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions

```java
import java.util.*;
public class TestJavaCollection6{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

Output:

```
Gautam
Karan
Ajay
```

Set Interface

- Extends collection interface.
- Represents un ordered set of elements
- Doesn't allow the duplication of items
- Can store at most one null value in Set.
- Implemented by HashSet , Linked Hash Set and Tree Set

```java
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

- Implements Set Interface
- Uses hash table for storage.
- Contain unique elements

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

- Linked List implementation of Set interface.
- Extends HashSet class and implements set interface.
- Maintain insertion order and permits null elements.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ajay
```

Sorted Set Interface

- Alternate of set interface that provide total ordering of elements (elements arranged in increasing order)

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

- Implements set interface.
- Utilize tree for storage.
- Similar to HashSet , it also stores the unique elements.
- As this uses Tree data structure, the retrieval time of Tree set is quite fast.
- Elements of the TreeSet are stored in ascending order.

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ajay
Ravi
Vijay
```

InternsDemo ▸ src/main/java ▸ collection ▸ ArrayDeques ▸

```
1  package collection;
2
3  import java.util.ArrayDeque;
4  import java.util.Deque;
5
6  public class ArrayDeques {
7      public static void main(String[] args) {
8
9          //Creating Deque and adding elements
10         Deque<String> deque = new ArrayDeque<String>();
11         deque.add("Gautam");
12         deque.add("Karan");
13         deque.add("Ajay");
14
15         //Traversing elements
16         for (String str : deque) {
17             System.out.println(str);
18         }
19     }
20 }
```

<terminated>

```
Gautam
Karan
Ajay
```

Comparable and Comparator in Java

05 March 2022 19:07

Comparable interface

- Used to order the objects of user defined class
- Contain a method named compareTo(Object)
- This only provide a single sorting sequence only.
- We can sort based only a single data member. We couldn't use more than one comparators in comparable interface.
- **public int compareTo(Object obj):** It is used to compare the current object with the specified object. It returns
  - positive integer, if the current object is greater than the specified object.
  - negative integer, if the current object is less than the specified object.
  - zero, if the current object is equal to the specified object.
- Example: sorts the list elements based on the age

File: Student.java

```
class Student implements Comparable<Student>{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}

public int compareTo(Student st){
if(age==st.age)
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
}
```

File: TestSort1.java

```
import java.util.*;
public class TestSort1{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

```
105 Jai 21
101 Vijay 23
106 Ajay 27
```

- Example: sorts the list elements based on the age in reverse order (descending)

File: Student.java

```
class Student implements Comparable<Student>{
 int rollno;
 String name;
 int age;
 Student(int rollno,String name,int age){
 this.rollno=rollno;
 this.name=name;
 this.age=age;
 }

 public int compareTo(Student st){
 if(age==st.age)
 return 0;
 else if(age<st.age)
 return 1;
 else
 return -1;
 }
 }
```

File: TestSort2.java

```
import java.util.*;
public class TestSort2{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

```
106 Ajay 27
101 Vijay 23
105 Jai 21
```

Comparator Interface

- Comparator interface is a functional interface that contains only one abstract method
  From <https://www.javatpoint.com/Comparator-interface-in-collection-framework>
- **Java Comparator interface** is used to order the objects of a user-defined class.
- contains 2 methods compare(Object obj1,Object obj2) and equals(Object element)
- It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else
- In current example, let's create a array and sort that array based on age, name , simple sorting by using comparator interface. When coming to comparable we can use only one attribute to sort.

Student.java

This class contains three fields rollno, name and age and a parameterized constructor.

```
class Student{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
}
```

AgeComparator.java

```
import java.util.*;
class AgeComparator implements Comparator{
public int compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;

if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;
}
}
```

NameComparator.java (here we are using inbuilt compareTo method of String class to create the logic)

```
import java.util.*;
class NameComparator implements Comparator{
public int compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;

return s1.name.compareTo(s2.name);
}
}
```

Simple.java (main code)

```
import java.util.*;
import java.io.*;
class Simple{
public static void main(String args[]){

ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

System.out.println("Sorting by Name");

Collections.sort(al,new NameComparator());
for(Student st: al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}

System.out.println("Sorting by age");

Collections.sort(al,new AgeComparator());
for(Student st: al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

Output

```
Sorting by Name
106 Ajay 27
105 Jai 21
101 Vijay 23

Sorting by age
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Java 8 New features in Comparator

Example: Sort based on age and name

Student.java

```
class Student {
  int rollno;
  String name;
 int age;
   Student(int rollno,String name,int age){
   this.rollno=rollno;
   this.name=name;
   this.age=age;
   }

   public int getRollno() {
      return rollno;
   }

   public void setRollno(int rollno) {
      this.rollno = rollno;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }


   }
```

TestSort1.java

File: TestSort1.java

```
import java.util.*;
public class TestSort1{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));
/Sorting elements on the basis of name
Comparator<Student> cm1=Comparator.comparing(Student::getName);
 Collections.sort(al,cm1);
 System.out.println("Sorting by Name");
 for(Student st: al){
  System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
 //Sorting elements on the basis of age
  Comparator<Student> cm2=Comparator.comparing(Student::getAge);
Collections.sort(al,cm2);
 System.out.println("Sorting by Age");
 for(Student st: al){
  System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
}
}
```

```
Sorting by Name
106 Ajay 27
105 Jai 21
101 Vijay 23
Sorting by Age
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Advanced Java - Introduction

Saturday, March 5, 2022 10:06 PM

```
public abstract class Employee {
    public static final String DEFAULT_NAME = "UNKNOWN";
    private static int nextId;

    private Integer id;
    private String name;
    private LocalDate hireDate;

    public Employee() {
        this(DEFAULT_NAME);
    }

    public Employee(String name) {
        id = nextId++;
        this.name = name;
        hireDate = LocalDate.now();
    }
```

- In Java 8 , see the constructor without any parameter . There we are redirecting to the other constructor with a default name as a parameter and execute the second constructor with the parameter name.
- Un implemented methods in abstract class must be declared in child class.

```
public void printEverybody() {
    employees.forEach(System.out::println);
}
```

- Here we are implementing forEach method where we implement System.out.println for every element of the array or list or set.
- In java 8, there is new data type is added that is LocalDate, which stores the date in it's format.

```
import java.text.NumberFormat;
import java.util.Locale;

public class CurrencyPrinter {
    public static void main(String[] args) {
        double amount = 1234567.8901234;
        NumberFormat nf = NumberFormat.getCurrencyInstance();
        System.out.println(nf.format(amount));

        nf = NumberFormat.getCurrencyInstance(Locale.FRANCE);
        System.out.println(nf.format(amount));
    }
}
```

- The following code is used to print the decimal value in currency format.
- Here we can see NumberFormat is an abstract class, so we could not instantiate it. So using getCurrencyInstance we are creating a variable with NumberFormat interface. Then using that we can format the decimal value into currency.
- We can format currency according to the country.
- Stream is an interface
- Suppose if we implements two interfaces into a class then the two interfaces have two methods with some different body then if class have some other different body then this class overrides the implemented interface. And if there is no method in current class to override then we get an error about that different body methods.

```java
public class DefaultMethodsDemo {
    public static void main(String[] args) {
        List<Integer> nums = Stream.of(-3, 1, 4, -5, 2, -6)
                .collect(Collectors.toList());
        System.out.println(nums);

        // removeIf is a default method in Collection
        // returns true if any elements were removed
        boolean removed = nums.removeIf(n -> n <= 0);
        System.out.println("Elements were " + (removed ? "" : "NOT") + " removed");
        System.out.println(nums);

        // Iterator has a default forEach method
        nums.forEach(System.out::println);
    }
}
```

- Simple example of predicates

```java
@Override
public String toString() {
    return String.format("Employee{id=%d, name='%s', hireDate=%s}", id, name, hireDate);
}
```

Here we have overridden the toString function of a class. In general we get class address instead of the detailed explanation. Now instead of that we are replacing the address with the user defined string.

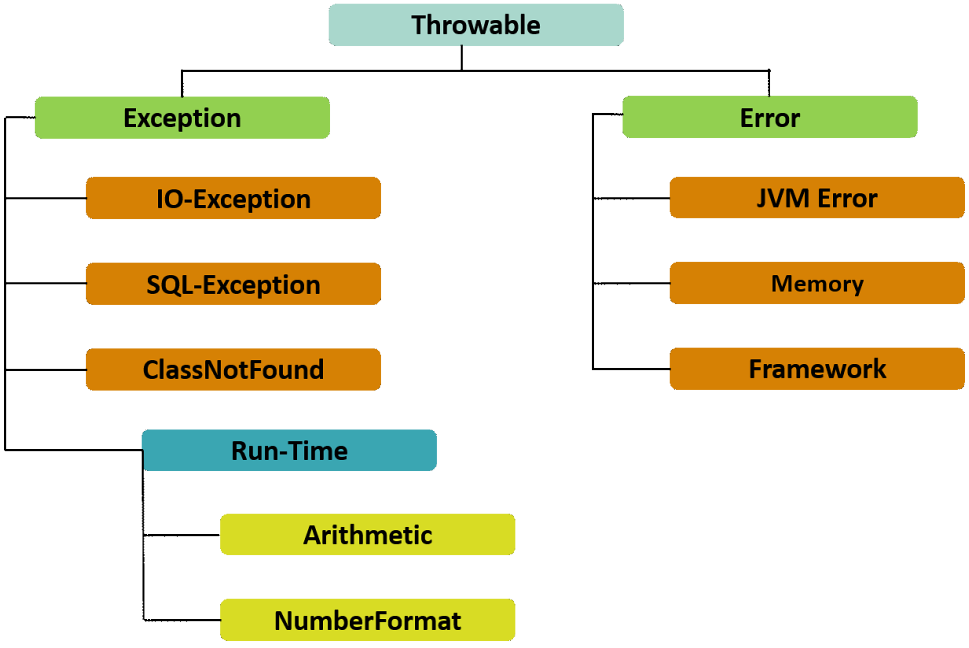
```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Employee)) return false;

    Employee employee = (Employee) o;

    if (id != null ? !id.equals(employee.id) : employee.id != null) return false;
    if (name != null ? !name.equals(employee.name) : employee.name != null) return false;
    return hireDate != null ? hireDate.equals(employee.hireDate) : employee.hireDate == null;

}
```

- We can even edit the equals method by using overridden annotation. And this is basic definition of equals.
- The mechanism of converting an object into integer is called hash code. In general when we have integers and we try to find the availability of that integer we will find the Boolean array. And if we want to do the same process for the string it will be not possible because here we are having strings but not numbers. Hence if we want to convert those strings into numbers and do the same process we use hash codes. The mechanism of converting an object into a number is called hash code. And even for million numbers we couldn't accommodate the Boolean array. Hence we use hash codes for both. Hash codes works in a way that the numbers are distributed in a meaningful way which is optimal to retrieve and add the numbers or strings. Hash code and equals method are auto generated for the class, where hash code will help in collections interface.

Exception Handling

Sunday, March 6, 2022 9:05 AM

- Parent class of exception is throwable.
- Throwable has two direct known subclasses , exception and error.
- Errors are conditions that are never supposed to happen, that are supposed to cause your system to fail. So we couldn't handle those.
- In methods in throwable, there is a series of constructors. There is a default constructor and the one that takes a string and other is used to linking them together.
- Using this we can throw an exception with a user defined message or just cause of an error without message in it.
- If we observe the class we can see some common methods such as getMessage , printStackTrace and toString method as well.
- And the child classes of it , will implement the parent class methods using super call.
- But methods in parent class exception are very limited because in reality all the methods of the Exception class are imported from the parent class throwable.
- All the errors occur at run time though why we call an exception as runtime exception because these are unchecked hence they form an error at runtime. Hence these are also called unchecked exception. These methods does not require a try catch block or throws clause in the method signature.
- And for checked exceptions we require a try catch block or throws clause.
- In checked exception, for example if we faced arithmetic exception divide by zero. We get the location of the error using default stack trace, showing us what happened and where.



- Now as we can see it clearly that arithmetic exception is a subclass of run time exception. We can add it in try catch block and catch the exception.
- Default Stack trace

```
/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/bin/java ...
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at exceptions.Arithmetic.main(Arithmetic.java:7) <5 internal calls>

Process finished with exit code 1
```

- Simple example of try catch

```
public class Arithmetic {
    public static void main(String[] args) {
        int x = 3;
        int y = 0;
        double z = 0;
        try {
            z = x / y;
        } catch (ArithmeticException e) {
            System.err.println(e);
            return;
        }
        System.out.printf("x = %d, y = %d, z = %s%n", x, y, z);

    }
}
```

If the Exceptions are siblings of each other then we can use multi catch exceptions. We can use this only when two exceptions are siblings of each other

```
public static void main(String[] args) {
    int x = 3;
    int y = 0;
    double z = 0;
    try {
        z = x / y;
    } catch (ArithmeticException e) {
        System.err.println(e);
    }
    System.out.printf("x = %d, y = %d, z = %s%n", x, y, z);

    try {
        Arithmetic.class.newInstance();
    } catch (InstantiationException | IllegalAccessException e) {
        e.printStackTrace();
    }


}
```

In general , if we want to read into a file we use the following code without any exception before java 7

```
Path dir = Paths.get("src", "main", "java", "exceptions");
BufferedReader br = null;
try {
    br = Files.newBufferedReader(dir.resolve("Arithmetic.java"));
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (br != null) try {
        br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Later we got a new concept called "try with resources"

```
Path dir = Paths.get("src", "main", "java", "exceptions");
try (BufferedReader br = Files.newBufferedReader(dir.resolve("Arithmetic.java"))) {
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}
```

- Here we no need to use the finally block because , as Bufferedreader is inside the try block after the process is done it will release it's resources automatically after we read the file or complete the try block. Else go to the catch block where we are handling IOException and printing the stackTrace.

If we want to create our own exception class

- Create a class extending the Exception class.
- Here we can see there are two constructors. The un parameterized constructor here , executes the second constructor with the default message when the user doesn't pass any parameter while throwing exception.
- If the user calls an exception with the parameter then directly the second constructor is going

to execute passing the message to the super class and printing that message through MyException.getMessage() this getMessage class is declared in the parent class of this MyException class.

MyException class

```
package exceptions;

public class MyException extends Exception {
    public MyException() {
        this("default message");
    }

    public MyException(String message) {
        super(message);
    }
}
```

The following code must be placed where we want to check an exception

```
try {
    throw new MyException("this is my issue");
} catch (MyException e) {
    System.err.println(e.getMessage());
}
```

Here we are trying to throw the exception with a message then the console will print that message and if we not pass any message into it. Then as per the MyException class code the default message is passed.

- Checked exceptions are those which are checked at compile time. These are the ones which can be handled through exception handling.
- Unchecked exceptions are those which are not checked at compile time

Generic Types

Sunday, March 6, 2022 12:38 PM

List without generic type

```
public void noGenerics() {
    List nums = new ArrayList();
    nums.add(3);
    nums.add(1);
    nums.add(4);
    nums.add("oops");
    System.out.println(nums);

    for (Object n : nums) {
        // Integer val = (Integer) n;  // Ack! ClassCastException
        System.out.println(n);
    }
}
```

Here we created a list without declaring datatype using generics, where we can store any type of data into list. That is a back drop when we want to store list of integers or any particular datatype. And even when we are retrieving we can see that we are mapping the elements to the object type and try to print it. And if we want to print it as an integer we need to map it to integer and if we want string we need to map it to string , so this shows too much ambiguity. Hence generic types are introduced. The following is example.

List with generic type

```
public void genericList() {
    List<Integer> nums = new ArrayList<>();
    nums.add(3); nums.add(1); nums.add(4);
    // nums.add("oops"); // Won't compile
    System.out.println(nums);

    for (Integer n : nums) {
        System.out.println(n);
    }
}
```

Here we see a list where we are declaring that list to store only integer with the help of generics. Hence when we add any string in it. It shows an error . And even when we try to print or retrieve it, we can map it to single datatype because a generic integer list contains of only integer datatypes.

- Due to storing elements of single datatype we can easily store the forEach loop

```
public void genericList() {
    List<Integer> nums = new ArrayList<>();
    nums.add(3); nums.add(1); nums.add(4);
    // nums.add("oops"); // Won't compile
    System.out.println(nums);

    nums.forEach(System.out::println);
}
```

Creating our own generic Type

```
public class Tuple<T,U> {
    private T first;
    private U second;

    public Tuple(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
        return second;
    }

    @Override
    public String toString() {
        return String.format("Tuple{first=%s, second=%s}", first, second);
    }
}
```

- Here we can see , we have created a generic type call Tuple<T,U>
- T and U are type parameters , which are not pre declared , those are declared at the main file

dynamically. For temporary purpose and current type declaration we use T and U as temporary datatypes.

- As the declaration is done. Use it in the user side.

```
public class TupleDemo {
    public static void main(String[] args) {
        Tuple<Integer, String> tuple = new Tuple<>(1, "hello");
        System.out.println(tuple);

        Tuple<LocalDate, List<String>> tuple1 =
                new Tuple<>(LocalDate.now(), Arrays.asList("a", "b", "c"));
        System.out.println(tuple1);
    }
}
```

- This is the main file, where we declared types of T and U as Integer and String dynamically and use it in our current TupleDemo class. And when we try to print object the user declared toString method is declared.
- We declare the datatypes in Dimond operator at the TupleDemo class.
- We can use the generics as the return type. Example:

```
public Pair<T> swap() {
    return new Pair<T>(second, first);
}
```

File Operations

Sunday, March 6, 2022 8:32 PM

```java
package IODemo;

import java.io.File;

public class FilesDemo {
    public static void main(String[] args) {
        //Using Paths.get(...) to create a Path
        //We are creating a path ( the address must exist in PC to do furthur operations )
        //C:\Users\saka2003\eclipse-workspace\InternsDemo\src\main\java\IODemo
        Path mainPath = Paths.get("/","Users","saka2003","eclipse-workspace","InternsDemo","src","main","java","IO
        System.out.println(mainPath);

        //We are moving into the previous created path and finding samplecontent and creating a path for it
        //These process is only for generating path in it's required syntax
        //It doesn't mean that there is a file if the path is created
        Path smpltxt = mainPath.resolve("sampleconZtent");
        System.out.println(smpltxt);

        //We can redirect the path to the sibling
        //Here sample content and files demo are in same folder , so they are called as siblings
        Path javaFile = smpltxt.resolveSibling("FilesDemo");
        System.out.println(javaFile);

        //Project directory
        //Here we are setting the current folder place as the project directory that is InternsDemo folder
        //As we create a project initially the address of it will be returned
        Path project = Paths.get(".");
        System.out.println(project); //This returns "."
        System.out.println(project.toAbsolutePath()); //This returns path upto  "." including InternsDemo
        //C:\Users\saka2003\eclipse-workspace\InternsDemo\.
        System.out.println("As a URI: "+project.toUri()); //Same path is generated in URI version
        //As a URI: file:///C:/Users/saka2003/eclipse-workspace/InternsDemo/./

        //Here when we use normalize all the edge dot's will be removed and returns the parent directory.
        Path p = Paths.get("/Users/../InternsDemo/.").normalize();
        System.out.println("Normalized: "+p);

        Path p1 = Paths.get("/Users/.../.").normalize();
        System.out.println("Normalized: "+p1);

        //. denote current directory
        //.. return sub sub directory
```

```
        Path project1 = Paths.get("/Users/kousen/Documents/./IntelliJ/..");
        System.out.println("parent: "+project1.getParent());
        System.out.println("file name: "+project1.getFileName());
        System.out.println("root: "+project1.getRoot());
        for(Path path : project.toAbsolutePath()) {
            System.out.print(path);
        }

        System.out.println();

        Path project2 = Paths.get("/Users/kousen/Documents/./IntelliJ/..");
        System.out.println("parent: "+project2.toAbsolutePath().getParent());
        System.out.println("file name: "+project2.toAbsolutePath().getFileName());
        System.out.println("root: "+project2.toAbsolutePath().getRoot());
        for(Path path : project.toAbsolutePath()) {
            System.out.print(path);
        }

        System.out.println();

        for(Path path : project1.toAbsolutePath()) {
            System.out.print(path);
        }
        System.out.println();

        File localDir = new File(".");
        System.out.println("Local Path: "+localDir);
        System.out.println("Local Absolute Path: "+localDir.toPath().toAbsolutePath());
        System.out.println("Local Absolute Path: "+localDir.toPath().toAbsolutePath().normalize());

        File localDir1 = new File("..");
        System.out.println("Local Path: "+localDir1);
        System.out.println("Local Absolute Path: "+localDir1.toPath().toAbsolutePath());
        System.out.println("Local Absolute Path: "+localDir1.toPath().toAbsolutePath().normalize());
        }
}
```

Output:

```
\\Users\saka2003\eclipse-workspace\InternsDemo\src\main\java\IODemo
\\Users\saka2003\eclipse-workspace\InternsDemo\src\main\java\IODemo\sampleconZten
\\Users\saka2003\eclipse-workspace\InternsDemo\src\main\java\IODemo\FilesDemo
.
C:\Users\saka2003\eclipse-workspace\InternsDemo\.
As a URI: file:///C:/Users/saka2003/eclipse-workspace/InternsDemo/./
Normalized: \InternsDemo
Normalized: \Users\...
parent: \Users\kousen\Documents\.\IntelliJ
file name: ..
root: \
Userssaka2003eclipse-workspaceInternsDemo.
parent: C:\Users\kousen\Documents\.\IntelliJ
file name: ..
root: C:\
Userssaka2003eclipse-workspaceInternsDemo.
UserskousenDocuments.IntelliJ..
Local Path: .
Local Absolute Path: C:\Users\saka2003\eclipse-workspace\InternsDemo\.
Local Absolute Path: C:\Users\saka2003\eclipse-workspace\InternsDemo
Local Path: ..
Local Absolute Path: C:\Users\saka2003\eclipse-workspace\InternsDemo\..
Local Absolute Path: C:\Users\saka2003\eclipse-workspace
```

Files.copy is used to copy content's of one file to other file

Files.move moves file from one destination to other destination

Files.deleteIfExists is used to delete the file if it exists.

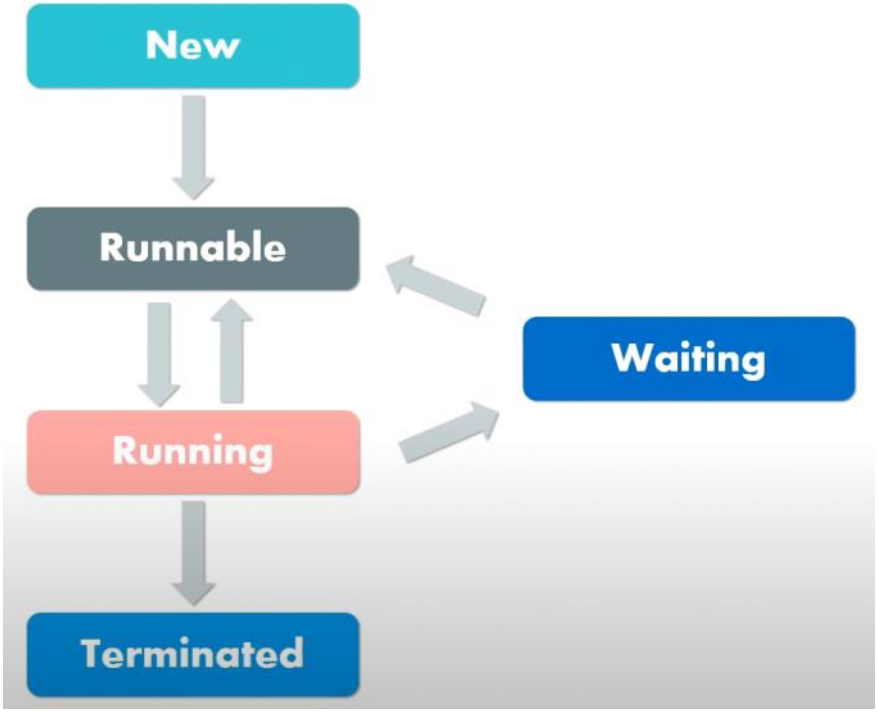
```
// Visit all the files in the source folder
Path javaDir = Paths.get("src", "main", "java");
try (Stream<Path> entries = Files.walk(javaDir)) {
    entries.forEach(System.out::println);
}
```

- The following code is used to visit all the folder's of mentioned directory. This uses depth first search operation where it iterates all the files of the current list until end and go to the next folder.

Introduction to Threads

Monday, March 7, 2022 5:46 PM

- **What is a Java Thread?**
  - It is a light weight sub process. Because it is a part of a huge process.
  - Smallest independent unit of the program. There may be one or more threads in a single program.
  - Contains separate path of execution. If we have multiple threads then each threads run independently.
  - Every Java Program contains at least one thread. That is main thread.
  - A thread is created and controlled by the java.lang.Thread class.
- **Java Thread Life Cycle?**
  - Here Java Thread can be only in one of the shown states at any point of time.



- Whenever a thread is created it will be in new state then when it needs to be run next it will be in runnable state and when the resources are available and the time comes it will be in running state and if we have a more prioritized thread needs to be occurred then the current thread moves to waiting state and when it needs to be executed it moves to runnable to move to running state and when the process is done the thread terminates at terminated state.

New

A new thread begins its life cycle in this state & remains here until the program starts the thread. It is also known as a **born thread**.

Runnable

Once a newly born thread starts, the thread comes under runnable state. A thread stays in this state is until it is executing its task.

Running

In this state a thread starts executing by entering run() method and the yield() method can send them to go back to the Runnable state.

Waiting

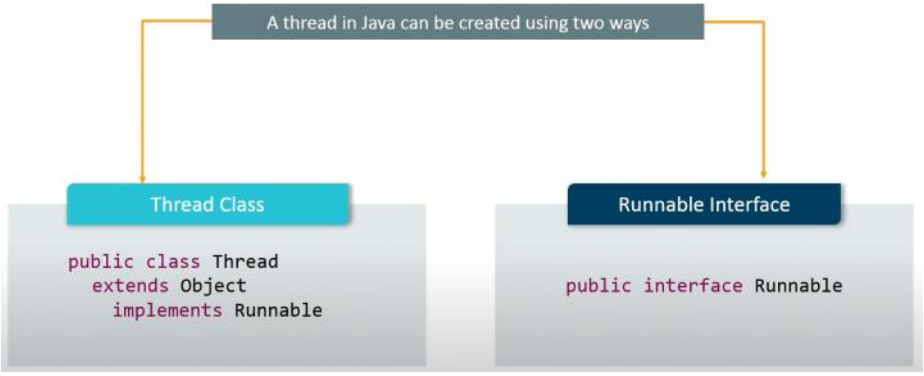
A thread enters this state when it is temporarily in an inactive state i.e it is still alive but is not eligible to run. It is can be in waiting, sleeping or blocked state.

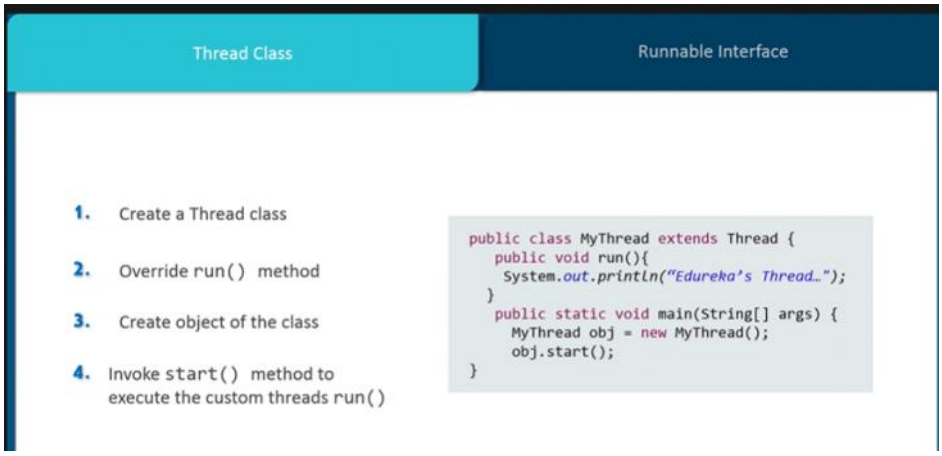
Terminated

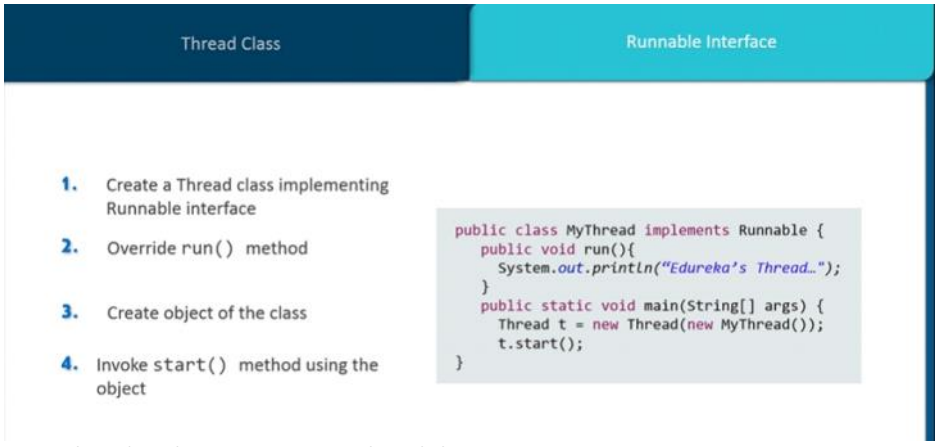
A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Creating a thread

- Suppose think we have a process and that process is taking for a long time to complete then if the surrounding process are tend to execute then the problem comes where we get application not responding problem for the surrounding processes. So there must be parallel execution of the process .







- Thread implementation using Thread class

InternsDemo ▸ src/main/java ▸ Threadss ▸ ThreadExample ▸ main(String[]) : void

```
package Threadss;

class MyTask extends Thread{
    @Override
    public void run() {
        for(int i=1;i<=10;i++) {
            System.out.println("MyTask Thread number: "+i);
        }
    }
}
public class ThreadExample {
    //main method represents main thread
    public static void main(String[] args) {
        //Whatever we write here will be executed by the main thread
        //In general the thread execute the jobs in a sequence
        //Check below we can see in individual threads the process is occuring in a sequence way

        //Job 1
        System.out.println("==Application Started==");
        //Job 2
        MyTask task = new MyTask();
        task.start();

        //If we use plain code instead of threads below written code will be waiting untill the Job 2 finish
        //If Job2 is a long running operation (for example several operations need to be done)
        //In such a case Job 3 waits infinite time and OS shows a warning that app is not responding
        //Some times apps seems to be hanged
        //So to prevent that we are using threads in our code
        //After using the above Job 2 code main and MyTask threads are working parallely

        //Job 3
        for(int i=1;i<=10;i++) {
            System.out.println("Main Thread number: "+i);
        }

        //Job4
        System.out.println("==Application ended==");


    }
}
```

<terminated> ThreadExample

```
==Application Started==
Main Thread number: 1
Main Thread number: 2
MyTask Thread number: 1
MyTask Thread number: 2
MyTask Thread number: 3
MyTask Thread number: 4
MyTask Thread number: 5
MyTask Thread number: 6
MyTask Thread number: 7
Main Thread number: 3
Main Thread number: 4
MyTask Thread number: 8
Main Thread number: 5
MyTask Thread number: 9
MyTask Thread number: 10
Main Thread number: 6
Main Thread number: 7
Main Thread number: 8
Main Thread number: 9
Main Thread number: 10
==Application ended==
```

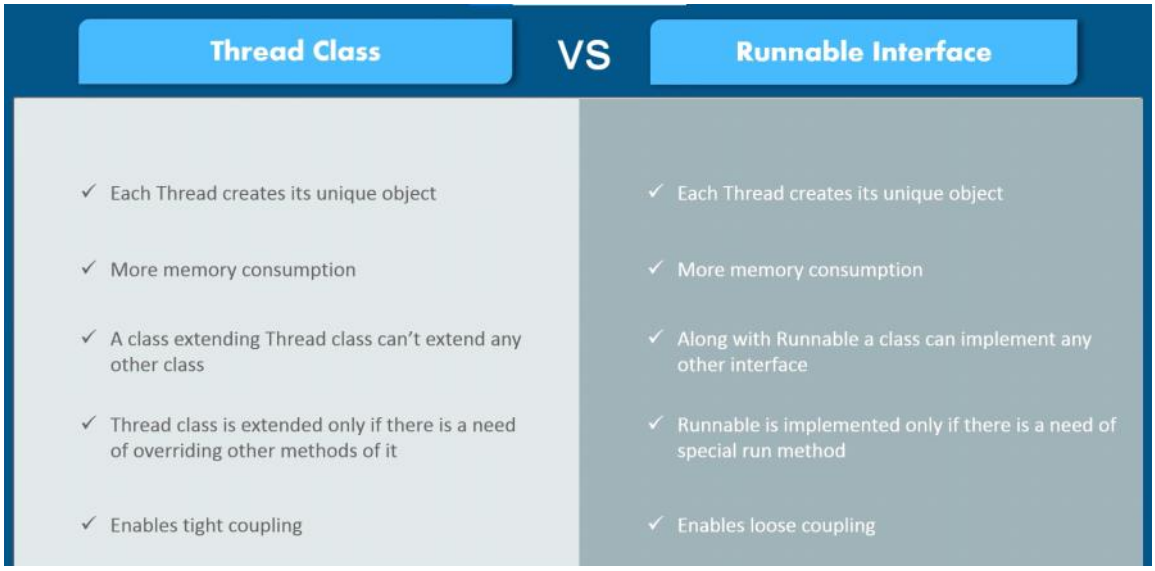
- Thread implementation using runnable interface

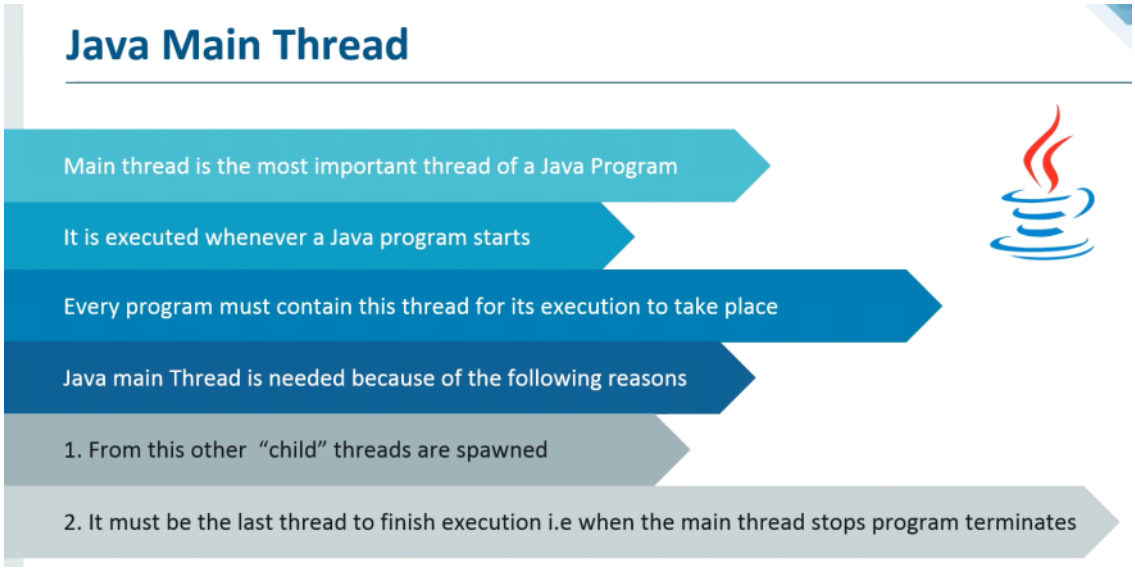
InternsDemo ▸ src/main/java ▸ Threadss ▸ sampleClass

```
package Threadss;
class sampleClass{

}
class MyTask extends sampleClass implements Runnable{
    @Override
    public void run() {
        for(int i=1;i<=10;i++) {
            System.out.println("MyTask Thread number: "+i);
        }
    }
}
public class ThreadExample {
    //main method represents main thread
    public static void main(String[] args) {
        //Whatever we write here will be executed by the main thread
        //In general the thread execute the jobs in a sequence
        //Check below we can see in individual threads the process is occuring in a sequence way

        //Job 1
        System.out.println("==Application Started==");
        //Job 2
        Runnable r = new MyTask();
        Thread task = new Thread(r);
        task.start();

        //If we use plain code instead of threads below written code will be waiting untill the Job 2 finish
        //If Job2 is a long running operation (for example several operations need to be done)
        //In such a case Job 3 waits infinite time and OS shows a warning that app is not responding
        //Some times apps seems to be hanged
        //So to prevent that we are using threads in our code
        //After using the above Job 2 code main and MyTask threads are working parallely

        //Job 3
        for(int i=1;i<=10;i++) {
            System.out.println("Main Thread number: "+i);
        }

        //Job4
        System.out.println("==Application ended==");
    }
}
```

<terminated> ThreadExample

```
==Application Started==
Main Thread number: 1
Main Thread number: 2
Main Thread number: 3
Main Thread number: 4
Main Thread number: 5
MyTask Thread number: 1
MyTask Thread number: 2
MyTask Thread number: 3
MyTask Thread number: 4
MyTask Thread number: 5
Main Thread number: 6
MyTask Thread number: 6
Main Thread number: 7
MyTask Thread number: 7
MyTask Thread number: 8
Main Thread number: 8
MyTask Thread number: 9
MyTask Thread number: 10
Main Thread number: 9
Main Thread number: 10
==Application ended==
```

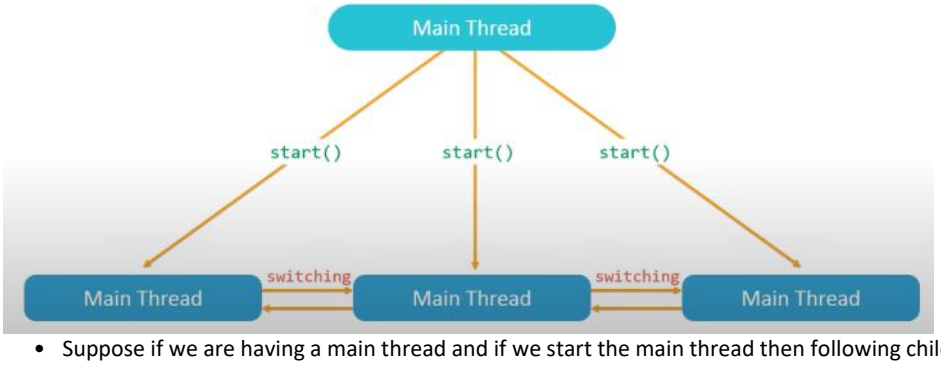




- JVM will start the main threads and other JVM threads. And here main threads in turn start their respective child threads. Main thread is the threads which starts first and ends at last.

Multi Threading

- Multi threading is the ability of a program to run two or more threads concurrently, where each thread can handle a different task at the same time making optional use of the available resouces



- Suppose if we are having a main thread and if we start the main thread then following child threads will also be executed at some time. So those child threads execute parallelly based on the resources available.

Asynchronous Execution of threads

- Threads execute their output parallelly and asynchronously

InternsDemo ▸ src/main/java ▸ Threadss ▸ MySyncThread ▸ MySyncThread(Printer)

```
1  package Threadss;
2  class Printer{
3      void printDocuments(int numOfCopies, String docName) {
4          for(int i=1;i<=10;i++) {
5              System.out.println(">> Printing "+docName+" "+i);
6          }
7      }
8  }
9  class MySyncThread extends Thread{
10     Printer pRef;
11     MySyncThread(Printer p){
12         pRef=p;
13     }
14     @Override
15     public void run() {
16         pRef.printDocuments(10, "JohnsProfile.pdf");
17     }
18 }
19 class YourSyncThread extends Thread{
20     Printer pRef;
21     YourSyncThread(Printer p){
22         pRef=p;
23     }
24     @Override
25     public void run() {
26         pRef.printDocuments(10, "JeevanProfile.pdf");
27     }
28 }
29 public class AsyncApp {
30
31     public static void main(String[] args) {
32         System.out.println("==Application Started==");
33         Printer printer = new Printer();
34         //printer.printDocuments(10, "IshanProfile.pdf"); //Here multiple threads working on same printer object parallel
35         //Printer is shared between two threads now //Some times it is shared for MySyncThread and sometimes YourSyncThread
36         MySyncThread mRef = new MySyncThread(printer); //So we need to take input asynchronously but we need to give output to the user synchronously
37         YourSyncThread yRef = new YourSyncThread(printer);
38         yRef.start();
39         mRef.start();
40         System.out.println("==Application Ended==");
41     }
42 }
```

<terminated> AsyncApp [Java Applicati

```
==Application Started==
==Application Ended==
>> Printing JeevanProfile.pdf 1
>> Printing JeevanProfile.pdf 2
>> Printing JeevanProfile.pdf 3
>> Printing JeevanProfile.pdf 4
>> Printing JohnsProfile.pdf 1
>> Printing JeevanProfile.pdf 5
>> Printing JeevanProfile.pdf 6
>> Printing JohnsProfile.pdf 2
>> Printing JeevanProfile.pdf 7
>> Printing JohnsProfile.pdf 3
>> Printing JeevanProfile.pdf 8
>> Printing JohnsProfile.pdf 4
>> Printing JeevanProfile.pdf 9
>> Printing JohnsProfile.pdf 5
>> Printing JeevanProfile.pdf 10
>> Printing JohnsProfile.pdf 6
>> Printing JohnsProfile.pdf 7
>> Printing JohnsProfile.pdf 8
>> Printing JohnsProfile.pdf 9
>> Printing JohnsProfile.pdf 10
```

Here if we want to establish synchronization between two threads then use the following join method

```
mRef.start();
try {
    mRef.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

yRef.start();
```

Or we can use synchronized keyword where if multiple threads execute same method repeatedly then this synchronized keyword executes a lock on the method it a thread then after completing the entire thread the control goes to next thread.

- Below we are declaring a method as a synchronized method. Only one thread can execute this method at a time . We have to declare it on the method that we want to synchronize

InternsDemo ▸ src/main/java ▸ Threadss ▸ YourSyncThread ▸ YourSyncThread(Printer)

```
1  package Threadss;
2  class Printer{
3      synchronized void printDocuments(int numOfCopies, String docName) {
4          for(int i=1;i<=10;i++) {
5              System.out.println(">> Printing "+docName+" "+i);
6          }
7      }
8  }
9  class MySyncThread extends Thread{
10     Printer pRef;
11     MySyncThread(Printer p){
12         pRef=p;
13     }
14     @Override
15     public void run() {
16         pRef.printDocuments(10, "JohnsProfile.pdf");
17     }
18 }
19 class YourSyncThread extends Thread{
20     Printer pRef;
21     YourSyncThread(Printer p){
22         pRef=p;
23     }
24     @Override
25     public void run() {
26         pRef.printDocuments(10, "JeevanProfile.pdf");
27     }
28 }
29 public class SyncApp {
30
31     public static void main(String[] args) {
32         System.out.println("==Application Started==");
33         Printer printer = new Printer();
34         //printer.printDocuments(10, "IshanProfile.pdf"); //Here multiple threads working on same printer object parallel
35         //Printer is shared between two threads now //Some times it is shared for MySyncThread and sometimes YourSyncThread
36         MySyncThread mRef = new MySyncThread(printer); //So we need to take input asynchronously but we need to give output
37         YourSyncThread yRef = new YourSyncThread(printer);
38         yRef.start();
39         mRef.start();
40         System.out.println("==Application Ended==");
41     }
42 }
```

<terminated> AsyncApp [Java Applica

```
==Application Started==
==Application Ended==
>> Printing JeevanProfile.pdf 1
>> Printing JeevanProfile.pdf 2
>> Printing JeevanProfile.pdf 3
>> Printing JeevanProfile.pdf 4
>> Printing JeevanProfile.pdf 5
>> Printing JeevanProfile.pdf 6
>> Printing JeevanProfile.pdf 7
>> Printing JeevanProfile.pdf 8
>> Printing JeevanProfile.pdf 9
>> Printing JeevanProfile.pdf 10
>> Printing JohnsProfile.pdf 1
>> Printing JohnsProfile.pdf 2
>> Printing JohnsProfile.pdf 3
>> Printing JohnsProfile.pdf 4
>> Printing JohnsProfile.pdf 5
>> Printing JohnsProfile.pdf 6
>> Printing JohnsProfile.pdf 7
>> Printing JohnsProfile.pdf 8
>> Printing JohnsProfile.pdf 9
>> Printing JohnsProfile.pdf 10
```

- Otherwise we can use synchronized block where only one thread we can execute once and later other threads will get control.

InternsDemo ▸ src/main/java ▸ Threadss ▸ MySyncThread ▸ run() : void

```
1  package Threadss;
2  class Printer{
3      synchronized void printDocuments(int numOfCopies, String docName) {
4          for(int i=1;i<=10;i++) {
```

<terminated> SyncApp [Java Applicat

```
==Application Started==
==Application Ended==
>> Printing JeevanProfile.pdf 1
>> Printing JeevanProfile.pdf 2
```

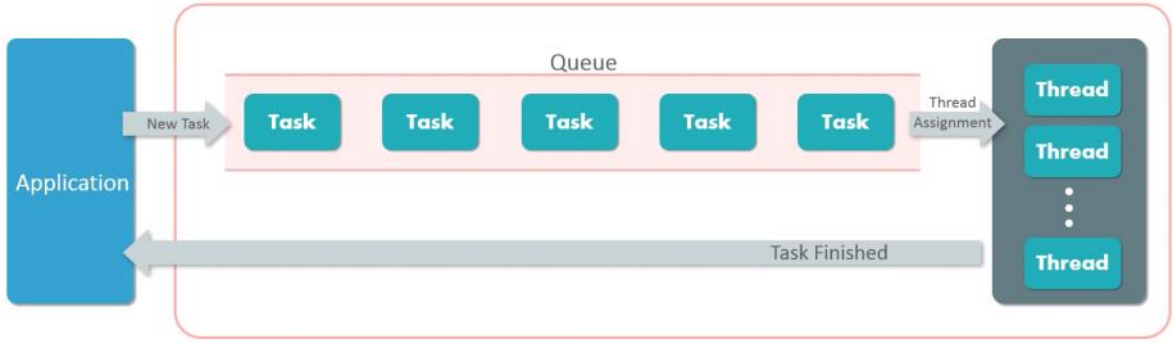
InternsDemo ▸ src/main/java ▸ Threadss ▸ MySyncThread ▸ run() : void

```java
package Threadss;
class Printer{
    synchronized void printDocuments(int numOfCopies, String docName) {
        for(int i=1;i<=10;i++) {
            System.out.println(">> Printing "+docName+" "+i);
        }
    }
}
class MySyncThread extends Thread{
    Printer pRef;
    MySyncThread(Printer p){
        pRef=p;
    }
    @Override
    public void run() {
        synchronized(pRef) {
            pRef.printDocuments(10, "JohnsProfile.pdf");
        }
    }
}
class YourSyncThread extends Thread{
    Printer pRef;
    YourSyncThread(Printer p){
        pRef=p;
    }
    @Override
    public void run() {
        pRef.printDocuments(10, "JeevanProfile.pdf");
    }
}
public class SyncApp {

    public static void main(String[] args) {
        System.out.println("==Application Started==");
        Printer printer = new Printer();
        //printer.printDocuments(10, "IshanProfile.pdf"); //Here multiple threads working on same printer object parallel
        //Printer is shared between two threads now //Some times it is shared for MySyncThread and sometimes YourSyncThread
        MySyncThread mRef = new MySyncThread(printer); //So we need to take input asynchronously but we need to give output
        YourSyncThread yRef = new YourSyncThread(printer);
        yRef.start();
        mRef.start();
        System.out.println("==Application Ended==");
    }
```

<terminated> SyncApp [Java Applicat

```
==Application Started==
==Application Ended==
>> Printing JeevanProfile.pdf 1
>> Printing JeevanProfile.pdf 2
>> Printing JeevanProfile.pdf 3
>> Printing JeevanProfile.pdf 4
>> Printing JeevanProfile.pdf 5
>> Printing JeevanProfile.pdf 6
>> Printing JeevanProfile.pdf 7
>> Printing JeevanProfile.pdf 8
>> Printing JeevanProfile.pdf 9
>> Printing JeevanProfile.pdf 10
>> Printing JohnsProfile.pdf 1
>> Printing JohnsProfile.pdf 2
>> Printing JohnsProfile.pdf 3
>> Printing JohnsProfile.pdf 4
>> Printing JohnsProfile.pdf 5
>> Printing JohnsProfile.pdf 6
>> Printing JohnsProfile.pdf 7
>> Printing JohnsProfile.pdf 8
>> Printing JohnsProfile.pdf 9
>> Printing JohnsProfile.pdf 10
```

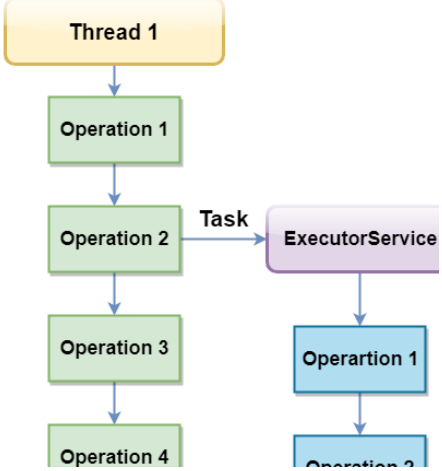
Java thread pool manages the pool of worker threads and contains a queue that keep the tasks waiting to get executed

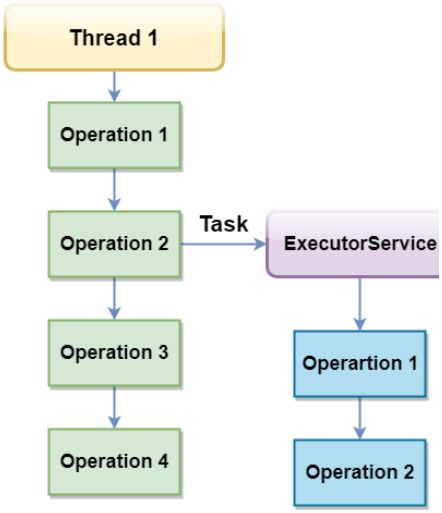


- Daemon thread in java is a **service provider thread that provides services to the user thread**. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

Java Executer service

- Java Executor Service The Java Executor Service is the interface which **allows us to execute tasks on threads asynchronously**.
- Java Executer service interface is present in java.util.concurrent package.
- It helps in maintaining a pool of threads and assign them tasks.
- It also helps us in maintaining a pool of threads and assigns them tasks according to availability of free threads according to the threads available.





Sample Executor Service

InternsDemo ▸ src/main/java ▸ executorService ▸ sampleExecutorService ▸

<terminated> sam

ExecutorService

```
1  package executorService;
2  //Here we are creating an Executor Service with ten threads
3  //And assigning it to an anonymous runnable implementation which print's task to print
4  //ExecutorService and after this task is over we are shutting down the executor service
5  import java.util.concurrent.ExecutorService;
6  import java.util.concurrent.Executors;
7
8  public class sampleExecutorService {
9      public static void main(String[] args) {
10         ExecutorService executorService = Executors.newFixedThreadPool(10);
11         executorService.execute(new Runnable() {
12             @Override
13             public void run() {
14                 System.out.println("ExecutorService");
15             }
16         });
17         executorService.shutdown();
18     }
19
20 }
```

Methods to create threads using executor service

ExecutorService executorService1 = Executors.newSingleThreadExecutor(); //Creates //a ExecutorService object having a single thread.

ExecutorService executorService2 = Executors.newFixedThreadPool(10); // Creates a //ExecutorService object having a pool of 10 threads.

ExecutorService executorService3 = Executors.newScheduledThreadPool(10); //Creates a scheduled thread pool executor with 10 threads. In scheduled thread //pool, we can schedule tasks of the thread

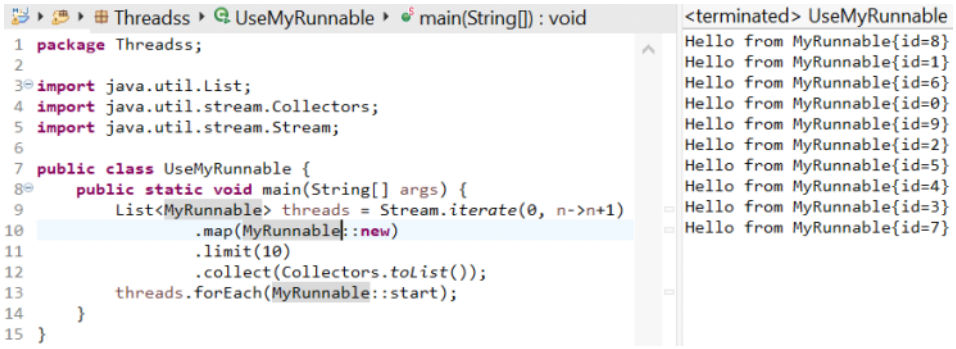
Execute() method takes in a runnable object and performs its task asynchronously. After the call to execute method we call the shut down method which blocks any other task to queue up in the executor service.

MyRunnable.java

```
package Threadss;

public class MyRunnable implements Runnable{
    private int id;
    private Thread thread = new Thread(this);
    public MyRunnable(int id) {
        this.id=id;
    }
    @Override
    public void run() {
        System.out.println("Hello from "+this);
    }
    public void start() {
        thread.start();
    }
    @Override
    public String toString() {
        return String.format("MyRunnable{id=%d}", id);
    }
}
```

```
Hello from MyRunnable{id=8}
Hello from MyRunnable{id=1}
Hello from MyRunnable{id=6}
Hello from MyRunnable{id=0}
Hello from MyRunnable{id=9}
Hello from MyRunnable{id=2}
Hello from MyRunnable{id=5}
Hello from MyRunnable{id=4}
Hello from MyRunnable{id=3}
Hello from MyRunnable{id=7}
```

UseMyRunnable.java




```
1  package Threadss;
2
3  import java.util.List;
4  import java.util.stream.Collectors;
5  import java.util.stream.Stream;
6
7  public class UseMyRunnable {
8      public static void main(String[] args) {
9          List<MyRunnable> threads = Stream.iterate(0, n->n+1)
10                 .map(MyRunnable::new)
11                 .limit(10)
12                 .collect(Collectors.toList());
13         threads.forEach(MyRunnable::start);
14     }
15 }
```