Java

The Essential Programming Concepts In Java For Beginners.

Concepts explained with real world java code examples.

Muhammad Naveed

Table of Contents

About this PDF

Please send feedback and corrections to
muhammadnaveedcis@gmail.com

Installation

Download the JDK and install on your development machine.

There are two parts of Java installation one is Java Development Kit (JDK) and other one is Java Runtime Environment (JRE). JDK is used for development purposes and JRE is used for running java based applications.

Here is the guide to install Java on Debian based Linux distributions.

```
>> sudo apt update
>> sudo apt install default-jdk
>> sudo apt install default-jre
```

After successful installation when you run these commands you will get the similar output.

```
>> java —-version
openjdk 17.0.9 2023-10-17
OpenJDK Runtime Environment (build 17.0.9+9-Debian-1deb12u1)
OpenJDK 64-Bit Server VM (build 17.0.9+9-Debian-1deb12u1, mixed mode, sharing)
```

```
>> javac —-version
javac 17.0.9
```

Java has two installation types, one is OpenJDK, OpenJRE which is the opensource of java. The second is Oracle JDK and JRE. If we are installing opensource version of java we have to install both OpenJDK and OpenJRE. In case of Oracle JDK it comes with JRE packed.

Functions Basics

Functions are the block of statements that perform a specific task. They may be reusable in the program.

Structure:

```
access-specifier return-type functionName(Parameters) {
      // Function Body
}
```

Example:

```
public void functionName() {
      // Function Body
}
```

Main Method

The entry point of the Java programming language is the main method. It is also known as driven function.

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

```java
        System.out.println("Hello, World");
}
```

Classes

In java every thing is in classes. The classes are the wrapper in which all the methods and fields are wrapped.

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

Naming Conventions

There are two naming conventions used in java. The one is 'Pascal Naming Convention' and the other one is 'camel Naming Convention'

Pascal Naming Convention

In Pascal Naming Convention, we capitalized the first letter of the variables/fields or classes/methods name. In Java we only use Pascal naming convention for naming classes, enums, interfaces and records.

```java
// Pascal Naming Convention
public class CalculatorClass {
        public static void main(String[] args) {
                String myName = "Naveed";
                System.out.println(myName);
        }
}
```

camel Naming Convention

In camel Naming Convention, first letter is in lower case and all other words first letter in upper case. We use camel naming convention to name methods, fields, variables, objects etc.

```java
public class CalculatorClass {
        public static void main(String[] args) {
                // camel Naming Convention
                String myName = "Naveed";
                System.out.println(myName);
        }
}
```

In contrast we can say that for naming Classes we use the Pascal Naming Convention and for naming the methods, fields or variables we use the camel Naming Convention.

First Program

As usual we are going to write our first hello world program in java. Create a new file and save it with "HelloWorld.java". Make sure the name of file and public class must be same.

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

For execution use the following commands form your cmd (for windows) or terminal (for mac/linux):

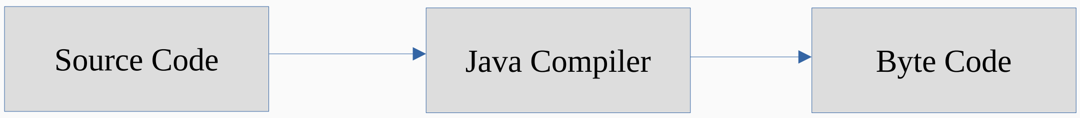
```
javac HelloWorld.java
java HelloWorld
```

make sure you are in the same directory where your source code is located and java is in your system path.

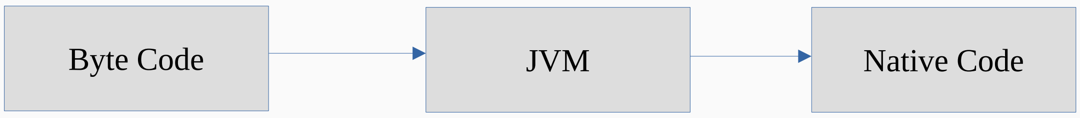
Java Code Execution

Java code execution process involves the two steps:

1. Compilation Step.
In compilation step java compiler converts our source code into byte code.



2. Execution Step.
In execution step java virtual machine converts the byte code to native code, which is platform independent.



Variables

A variable is a data container, it's value can be read, write and modify through out the code. Variables are changeable. Basic declaration of variable's is given below.

```
int iValue = 4;
short sValue = 29;
long lValue = 32L;
byte bValue = 40;
float fValue = 8.9F;
double dValue = 9.3;
```

```java
char cValue = 'C';
boolean isTrue = true;
String myName = “Muhammad Naveed”;
```

By default java compile treat long as int data type so, we use the “L” for long type. Also java compiler tread float as double data type so, we use the “F” for float type.

Constants

A field or variable whose value can’t be changed through out the code is known as constant. In java we use the **final** keyword to make the field, variable or methods constant.

```java
public static void main(String[] args) {
      // Constant double type PI.
      final double PI = 3.14; // usually we name constants in capital letter.
      System.out.println(PI);
}
```

Types in Java

There are two types in java, primitive data type and non-primitive/reference data type.

Primitive type

Primitive data types are independent on others, which means changing the value of one primitive data type instance don’t affect the other.

Types	Bytes	Range
byte	1	-128, 128
short	2	-32K, 32K
int	4	-2B, 2B
long	8	
float	4	
double	8	
char	2	A, B, C...
boolean	1	true/false

Reference type

Reference data types are used for storing complex objects. Reference data types include String, user defined classes, points etc.

```java
import java.awt.Point;
public class Ref {
      public static void main(String[] args) {
            Point p1 = new Point(3, 4);
            Point p2 = p1;
            p1.x = 34;

            System.out.println(p1);
            System.out.println(p2);
      }
}
```

Casting

The conversion between different data types is known as casting. There are two types of casting.

Implicit Conversion

Automatic conversion form small data type to larger data type. This is known as implicit conversion. Therefore, we can say that the conversion is performed in order.

```
byte > short > int > long > float > double

public static void main(String[] args) {
      int num1 = 5;
      double num2 = num1;

      System.out.println(num2);    // 5.0
}
```

Explicit Conversion

Manual conversion form one larger data type to smaller data type. This is known as explicit conversion.

```
public static void main(String[] args) {
      double num1 = 5.3;
      int num2 = (int) num1; // Explicit casting.

      System.out.println(num2);    // 5
}
```

Strings

Strings usually look like the primitive data type but they are not. Strings are the non-primitive data types because, strings are the class and we create the string object.

Defining the strings:

```
public class StringsMethods {
      public static void main(String[] args) {
            String myName = new String("Muhammad Naveed");
```

```java
        System.out.println(myName);
    }
}
```

as we are using string most of the time so string can be defined by

```java
public class StringsMethods {
    public static void main(String[] args) {
        String myName = "Muhammad Naveed";
        System.out.println(myName);
    }
}
```

useful methods in String class:

```java
public class StringsMethods {
    public static void main(String[] args) {
        // Declaring the String
        String myName = "Muhammad Naveed";

        // printing the length of the string.
        System.out.println(myName.length());

        // concatenating two strings.
        System.out.println(myName.concat(". I am a programmer."));

        // formatting the string
        System.out.println(String.format("My name is %s", myName));

        // finding the character at index 3.
        System.out.println(myName.charAt(3));

        // comparing the two strings.
        String yourName = "Muhammad NAveed";
        System.out.println(myName.equals(yourName));

        // comparing by ignore case.
        System.out.println(myName.equalsIgnoreCase(yourName));

        // printing the index of first occurrence of char 'h'
        System.out.println(myName.indexOf('h'));
        // replacing 'M' with '*'
        System.out.println(myName.replace('M', '*'));

        // split the string and print these tokens
        for (String tokens : myName.split("")) {
              System.out.println(tokens);
        }

        // printing the sub-string from the string.
        System.out.println(myName.substring(3, 8));

        // Check whether myName starts with the char 'M'
        boolean isTrue = myName.startsWith("M");
        System.out.println(isTrue);

        // Check whether myName ends with the char 'd'
        isTrue = myName.endsWith("d");
        System.out.println(isTrue);

        // Transform myName to lowercase letters
        System.out.println(myName.toLowerCase());

        // Transform myName to uppercase letters
        System.out.println(myName.toUpperCase());

        // Remove the unnecessary spaces from the string.
        System.out.println(myName.trim());
    }
}
```

One more thing to remember here is, Strings can't be changed if we search and replace from them, a new string object will be returned after the replacement is performed. Therefore, strings are immutable.

Arrays

Arrays can be useful in storing data in sequence in computers memory.

Defining an array

```
public class ArrayClass {
      public static void main(String[] args) {
        // Creating an array
        int[] myArray = new int[3];
        // Storing the values
        myArray[0] = 33;
        myArray[1] = 23;
        myArray[2] = 21;
        // Printing the vales.
        for (int i = 0; i < myArray.length; i++) {
            System.out.println(myArray[i]);
        }
    }
}
```

The given above method is not recommended for array declaration. The useful methods of arrays are given below

```
import java.util.Arrays;

public class ArrayClass2 {
    public static void main(String[] args) {
        // Creating an array
        int[] myArray = {23, 32, 29, 72, 55};
        // Printing the myArray using Arrays class
        System.out.println(Arrays.toString(myArray));
        // Sorting the array
        Arrays.sort(myArray);
        // Printing the myArray using Arrays class
        System.out.println(Arrays.toString(myArray));
    }
}
```

Two-dimensional array

The two dimensional array store the data in the form of matrix.

```
import java.util.Arrays;

public class TwoDimensionalArray {
    public static void main(String[] args) {
        // Creating 2-dimensional array.
        int[][] myArray = { {2, 3, 4}, {5, 6, 7} };
        // printing array
        System.out.println(Arrays.deepToString(myArray));
    }
}
```

Arithmetic Operators

We usually use arithmetic operations in our programs for various purposes. The arithmetic operators are given below:

Operators	Description
+	Addition.
-	Subtraction.
*	Multiplication.
/	Division.
%	Remainder (Modulus).

```
float num1 = 23.0F;
float num2 = 32.0F;

float sum, dif, mul, div, mod;

sum = (num1 + num2);
dif = (num1 - num2);
mul = (num1 * num2);
div = (num1 / num2);
mod = (num1 % num2);

System.out.println(sum); // 55.0
System.out.println(dif); // -9.0
System.out.println(mul); // 736.0
System.out.println(div); // 0.71875
System.out.println(mod); // 23.0
```

Math Class

Math class allow us to perform multiple mathematical operations such as ceiling, flooring etc.

```
public class MathClass {
      public static void main(String[] args) {
            double someValue = 34.23;

            // round off the number => 34
            System.out.println(Math.round(someValue));
            // Ceil the number => 35.0
            System.out.println(Math.ceil(someValue));
            // floor the number => 34.0
            System.out.println(Math.floor(someValue));
            // find maximum number
```

```
        System.out.println(Math.max(42, 66));  // 66
        // find minimum number
        System.out.println(Math.min(42, 66));  // 42
        // generate random number between (0 - 1)
        System.out.println(Math.random());
    }
}
```

Number Format

Number Format is a special class for formatting numbers in format of currency, percentage etc. The useful methods of number format class are.

```
import java.text.NumberFormat;

public class NumberFormating {
    public static void main(String[] args) {
        // format 1024 in dollar format $1,024.00
        String currency = NumberFormat.getCurrencyInstance().format(1024);
        System.out.println(currency);
        // format 0.4 in percentage format 40%
        String percentage = NumberFormat.getPercentInstance().format(0.4);
        System.out.println(percentage);
    }
}
```

Comparison Operators

Comparison operations are used to compare two operands. The most common comparison operators we use are equality operator and inequality operator.

```
public class Equal {
    public static void main(String[] args) {
        int firstUserIncome = 23_000;
        int secondUserIncome = 2_000_000;

        // Equality Operator (==)
        if (firstUserIncome == secondUserIncome){
            System.out.println("Both have same income");
        }

        // Inequality Operator (!=)
        if (firstUserIncome != secondUserIncome){
            System.out.println("Both don't have same income");
        }
    }
}
```

Logical Operators

There are three types of logically operators.

Operator	Description
||	Returns true if one of the statements is true
&&	Returns true if both statements are true
!	Reverse the result, returns false if the result is true

```
public static void main(String[] args) {
      // && operator
      System.out.println((5 > 3) && (8 > 5));    // true
      System.out.println((5 > 3) && (8 < 5));     // false
      // || operator
      System.out.println((5 < 3) || (8 > 5));     // true
      System.out.println((5 > 3) || (8 < 5));     // true
      System.out.println((5 < 3) || (8 < 5));     // false
      // ! operator
      System.out.println(!(5 == 3));             // true
      System.out.println(!(5 > 3));              // false
}
```

If Else Statements

We specify the condition in if block, if the condition is true then, if block get executed else, else block will be executed.

Structure of IF Else Statement

```
if (condition) {
      // if-code-block;
} else if (condition) {
      // else-if-code-block;
} else {
      // else-code-block;
}
```

Example:

```
public class IF_Else {
      public static void main(String[] args) {
            int temp = 29;
            if (temp > 30) {
                  System.out.println("It's a hot day,");
            } else if (temp > 20) {
                  System.out.println("It's a nice day.");
            } else {
                  System.out.println("It's a cold day.");
            }
      }
}
```

Ternary operator/Elvis operator

Ternary or Elvis operator is also used for comparisons but it is very simple and take less lines of code. This operator checks the condition if condition is true the first statement gets executed else second statement gets executed.

```
(Condition) ? first_block : second_block;
```

Example:

```
public class Elvis {
      public static int checkMax(int first, int second) {
            return ((first > second) ? first : second);
      }
      public static void main(String[] args) {
            System.out.println(checkMax(23, 43));
      }
}
```

Switch Statement

When a value requires different actions for a fixed set of values, the if might get more complex, the more the set of values increases. In this case the more suitable statement is the switch statement.

```
import java.util.Scanner;

public class Switch {
      public static void main(String[] args) {
            Scanner s = new Scanner(System.in);
            System.out.print("Enter day number : ");
            int dayNumber = s.nextInt();
            s.close();

            switch(dayNumber) {
                  case 1:
                        System.out.println("Monday");
```

```java
                    break;
                case 2:
                    System.out.println("Tuesday");
                    break;
                case 3:
                    System.out.println("Wednesday");
                    break;
                case 4:
                    System.out.println("Thursday");
                    break;
                case 5:
                    System.out.println("friday");
                    break;
                case 6:
                    System.out.println("Saturday");
                    break;
                case 7:
                    System.out.println("Sunday");
                    break;
                default:
                    System.out.println("Invalid day number! ");
            }
        }
}
```

We are not adding the break statement in default section because after executing default section the control goes out of the switch block automatically.

We can also improve our code by using the enhance switch condition, here is example.

```java
import java.util.Scanner;

public class EnhanceSwitch {
        public static void main(String[] args) {
            Scanner s = new Scanner(System.in);
            System.out.print("Enter day number : ");
            int dayNumber = s.nextInt();
            s.close();

            switch (dayNumber) {
                case 1 -> System.out.println("Monday");
                case 2 -> System.out.println("Tuesday");
                case 3 -> System.out.println("Wednesday");
                case 4 -> System.out.println("Thursday");
                case 5 -> System.out.println("friday");
                case 6 -> System.out.println("Saturday");
                case 7 -> System.out.println("Sunday");
                default -> System.out.println("Invalid day number! ");
            }
        }
}
```

Do While loop

```
public class DoWhileLoop {
        public static void main(String[] args) {
                int i = 10;

                do {
                        System.out.println(i);
                        i--;
                }
                while (i == 10);
        }
}
```

The main different between while loop and do while loop is that do while loop always executed once, if the condition is fulfill because, the block is above the condition checker statement.

For loop

Used for iterations.

```
public class ForLoop {
        public static void main(String[] args) {
                for (int i = 0; i < 10; i++) {
                        System.out.println(i);
                }
        }
}
```

For Each or Enhanced for loop

Usually used for iterating over the arrays or lists.

```
public class ForEach {
        public static void main(String[] args) {
                int[] array = {34, 53, 62, 94, 12, 65, 34, 3, 64, 26};

                for (int items : array)      {
                        System.out.println(items);
                }
        }
}
```

Break – Continue Statement

Break keyword break the loop and transfer the control out of the loop body. Whereas the continue keyword simply skip the loop and continue iterations of loop.

Continue Example:

```
public class SkipNumber {
      public static void main(String[] args) {
            for (int i = 0; i <= 10; i++) {
                  if (i == 6)
                        continue;

                  System.out.print(i + " ");
            }
            System.out.println("\nSkiped 6 by using continue keyword.");
      }
}
```

Break Example:

```
public class Break {
      public static void main(String[] args) {
            int userValue = 11;
            // Iteration for 100 times.
            for (int index = 0; index < 100; index++) {
                  // if index == userValue (11) then, break the loop.
                  if (index == userValue) {
                        break;
                  }
                  // printing the index.
                  System.out.println(index);
            }
      }
}
```

ArrayList

The main feature of arraylist is that it can dynamically shrink 50% of its original size. For-example, if we create a arraylist with initial size of 4 and we add the 5th item to our list then, the size of our arraylist grows 50% of original and becomes the 6 (4 + 2). The useful methods of arraylist are given below:

```
import java.util.ArrayList;

public class ArrayListExample {
      public static void main(String[] args) {
            ArrayList<Integer> myIntList = new ArrayList<Integer>(4);
```

```
        // Adding the elements of array list
        myIntList.add(3);
        myIntList.add(2);
        myIntList.add(89);
        myIntList.add(19);
        myIntList.add(92);

        // Check whether 32 is in array list
        System.out.println(myIntList.contains(32));

        // Return value at given index
        System.out.println(myIntList.get(1));
        // For each loop on array list
        myIntList.forEach(number -> {
            System.out.println(number);
        });

        // Return boolean whether the list is empty or not.
        System.out.println(myIntList.isEmpty());

        // Return the size of array list
        System.out.println(myIntList.size());

        // Convert to string Object.
        System.out.println(myIntList.toString());

        // Clear the array list
        myIntList.clear();
    }
}
```

Hash map

Hash map is similar to the dictionaries in other programming languages such as *python*. Hash map contains the key name and associate value. The useful methods of hashmap are given below:

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        // Creating new hash map of type string as key and also string as value.
        Map<String, String> hm = new HashMap<>(2);
        // Inserting the values associated with keys.
        hm.put("Name", "Naveed");
        hm.put("Age", "21");
        hm.put("Profession", "Software Engineer");

        // Printing the key, Values.
        System.out.println("NAME : " + hm.get("Name"));
        System.out.println("AGE : " + hm.get("Age"));
        System.out.println("PROFESSION : " + hm.get("Profession"));

        // Returning whether the hash map contains the 'Age' key.
        System.out.println(hm.containsKey("Age"));

        // Returning whether the hash map contains the 'Naveed' value.
        System.out.println(hm.containsValue("Naveed"));

        // Clear the entire hash map.
        hm.clear();
    }
}
```

this – Keyword

This – keyword refers to the current object in a method or constructor. this keyword also used to eliminate the confusion between the fields name and parameters name. The common example of this keyword is given below.

```
public class Human {
    private String name;
    private int age;
    private float height;

    // Parameterized constructor.
    Human(String name, int age) {
        // here this.name clarify that name is field name (left hand side)
        this.name = name;
        this.age = age;
    }

    // Overloaded parameterized constructor.
    Human(String name, int age, float height) {
        // By using the 'this' keyword we can call the overloaded constructor.
        this(name, age);
        this.height = height;
    }
```

```java
	// setter method.
	public void setHeight(float height) {
		this.height = height;
	}

	// Details showing method
	public void showDetails() {
		System.out.println("Name : " + this.name);
		System.out.println("Age : " + this.age);
		System.out.println("Height : " + this.height + "m");
	}

	// main driven function.
	public static void main(String[] args) {
		var M = new Human("Naveed", 92);
		var N = new Human("Naveed", 92, 1.91f);

		M.setHeight(1.71f);

		M.showDetails();
		N.showDetails();
	}
}
```

Constructors

Constructor is a special method whose name is same as the class name, without the return-type. Constructors are automatically called when the object reference is created. There are default and parameterized constructors. Every class has default constructor even if we don't write it, the java compiler automatically generate the constructor for us.

```java
public class Library {
	private String currentBookTitle;
	private String currentBookID;
	private String bookAuthor;
	// Default constructor.
	Library() {
		currentBookTitle = "The Complete Sherlock Holmes";
		bookAuthor = "Arthur Conan Doyle";
		currentBookID = "Res-2020-367";
	}

	// Method just for showing default values set by constructor.
	public void showCurrentBookInfo() {
		System.out.println("Book Title : " + this.currentBookTitle);
		System.out.println("Book Author : " + this.bookAuthor);
```

```
            System.out.println("Book ID : " + this.currentBookID);
      }

      public static void main(String[] args) {
            var l1 = new Library();
            l1.showCurrentBookInfo();
      }
}
```

here is the same example with parameterized constructor

```
public class Library {
      private String currentBookTitle;
      private String currentBookID;
      private String bookAuthor;

      // Parameterized constructor.
      Library(String currentBookTitle, String bookAuthor, String currentBookID) {
            this.currentBookTitle = currentBookTitle;
            this.bookAuthor = bookAuthor;
            this.currentBookID = currentBookID;
      }

      // Method just for showing default values set by constructor.
      public void showCurrentBookInfo() {
            System.out.println("Book Title : " + this.currentBookTitle);
            System.out.println("Book Author : " + this.bookAuthor);
            System.out.println("Book ID : " + this.currentBookID);
      }

      public static void main(String[] args) {
            var l1 = new Library("The Complete Sherlock Holmes", "Arthur Conan
            Doyle", "Res-2020-367");
            l1.showCurrentBookInfo();
      }
}
```

Access Specifiers

Access specifiers control the visibility of content. Public, Private and Protected are the access specifiers in java.

Private
Private visibility allows a fields to only be accessed by its class. They are often used in conjunction with public getters and setters. Which is an example of encapsulation. We will talk about encapsulation latter.

Public
Visible to the class, package, and subclass in program.

No Modifier
With no modifier, the default is package visibility. Which means the class is public.

Protected
Protected visibility allows member to visible to its package, along with any of its sub classes.

Object Oriented Programming – Encapsulation

Encapsulation is the process by which the fields and the methods are integrated as a single unit. By encapsulating a class, other classes can't access the private fields of the class. To access the private fields of the class we use the accessors (getters) and mutators (setters) methods. By using the getter and setter methods we can actually access the private fields of the class.

```java
public class Human {
	private String name;

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

	public String getName() {
		return this.name;
	}

	public static void main(String[] args) {
		var h = new Human();
		h.setName("Naveed");

		System.out.println(h.getName());
	}
}
```

Object Oriented Programming – Abstraction

Abstraction is the process of hiding the certain details and showing only essential information to the user. Data abstraction can be achieved with access-specifiers, abstract classes and interfaces. Here is an example.

Abstraction using access-specifiers

```java
class FactorialFinder {
	// private fields.
	private long factorial;
	private final int number;
	// parameterized constructor.
	public FactorialFinder(int number) {
		factorial = 1;
		this.number = number;
	}

	// factorial method.
	public void factorial() {
		// here the user is not concerned with isValid() method.
		if (isValid(number)) {
			for (int i = number; i > 0; i--) {
				factorial *= i;
			}
			System.out.println("The factorial of " + number + " is " +
			factorial);
		} else {
			System.out.println("Can't determine the factorial of negative
			number.");
		}
	}

	// validation checker method.
	private boolean isValid(int number) {
		return (number > 0);
	}
}

public class Main {
	// main divan method.
	public static void main(String[] args) {
		var f = new FactorialFinder(5);
		f.factorial();
	}
}
```

here, in `FactorialFinder` class, we are hiding the `isValid` method from our users by making them private.

Abstract classes and methods

The abstract keyword is non-access modifier, used for classes and methods. Abstract class is restricted class which means no object can be created. To access the abstract class we must have to inherit the class form it.

Abstract methods can only be used in abstract class having no body, the inherited class define the abstract methods.

Abstract class have both the abstract method and normal methods.

```
public abstract class Animal {
      public abstract void sound();
      public void sleep() {
            System.out.println("Zzz...");
      }
}

public class Cat extends Animal {
      public void sound() {
            System.out.println("Meow Meow!");
      }
}

public class Main {
      public static void main(String[] args) {
            var katie = new cat();
            katie.sound();
            katie.sleep();
      }
}
```

Interface

Interfaces are the same as abstract class without having the abstract keyword, also all the methods of interface are public by default. We also achieve abstraction in java with interfaces.
One thing to remember here is in interfaces we inherit the class using the '**implements**' keyword not the '**extends**' keyword.

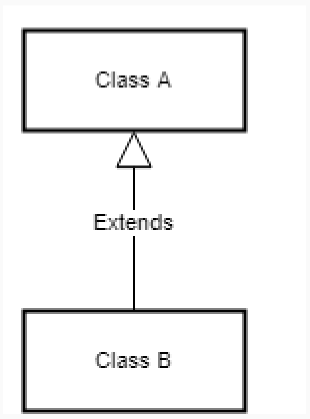
```
public interface Animal {
      public void sound();
      public void sleep();
}

public class Cat implements Animal {
      public void sound() {
            System.out.println("Meow Meow!");
      }

      public void sleep() {
            System.out.println("Zzz....");
      }
}

public class Main {
      public static void main(String[] args) {
            var katie = new cat();
            katie.sound();
            katie.sleep();
        }
}
```

Object Oriented Programming – Inheritance

Inheritance is the technique by which we create a hierarchy between classes by inheriting form other class (parent class).

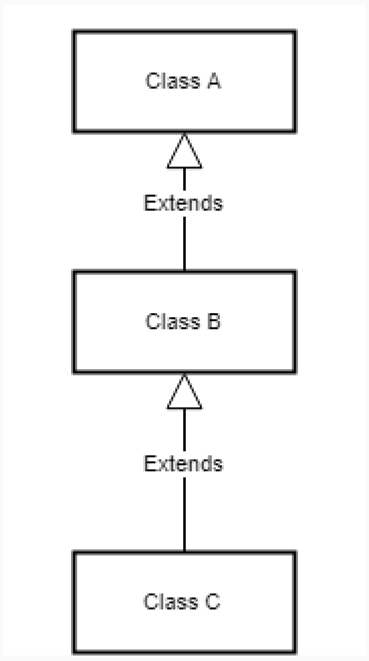
Single Inheritance.

In single inheritance one class let say class B inherits form class A.



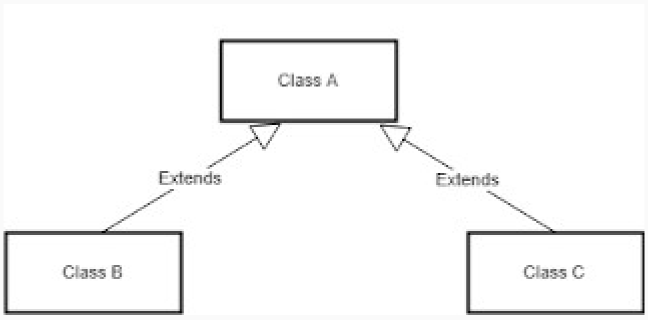
Multilevel Inheritance.

In multilevel inheritance class C inherits form class B and class B inherits from class A.



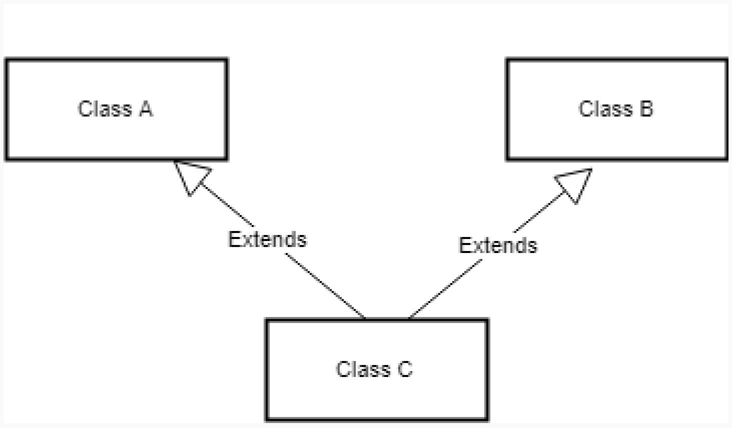
Hierarchical Inheritance.

In hierarchical inheritance class C and class B both inherits form class A.



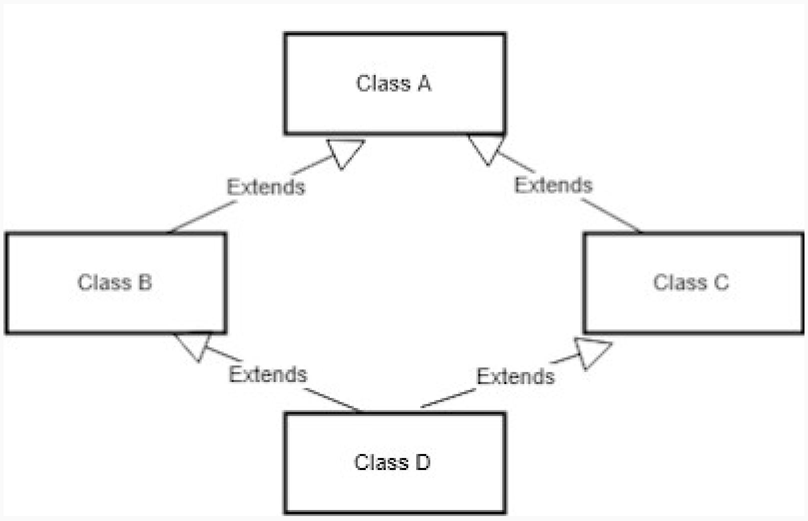
Multiple Inheritance.

In multiple inheritance class C inherits from class B and class A. Java does not supports multiple inheritance with classes but it can be implemented by using **interfaces**.



Hybrid Inheritance.

It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritances with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through **Interfaces**.



```
public class Animal {
      private int legs;
      private String color;
      private boolean vegetarian;

      Animal() {
            legs = 0;
            color = "";
            vegetarian = false;
      }

      Animal(int legs, String color, boolean vegetarian) {
            this.legs = legs;
            this.color = color;
            this.vegetarian  = vegetarian;
      }

      public int getLegs() {
            return legs;
      }

      public void setLegs(int legs) {
            this.legs = legs;
      }

      public String getColor() {
            return color;
      }

      public void setColor(String color) {
            this.color = color;
      }

      public boolean isVegetarian() {
            return vegetarian;
      }

      public void setVegitarian(boolean vegitarian) {
            this.vegitarian = vegitarian;
      }
}

public class Cat extends Animal {
      Cat(int legs, String color, boolean vegetarian) {
            // calling the parent class constructor.
            super(legs, color, vegetarian);
      }
}


public class Main {
      public static void main(String[] args) {
            var kattie = new Cat(4, "White", false);

            System.out.println("Cat has " + kattie.getLegs() + " legs");
            System.out.println("Cat color is " + kattie.getColor());
            System.out.println("Is cat a vegetarian? " + kattie.isVegetarian());
      }
}
```

Upcasting
We can create an instance of subclass and then assign it to super class reference, this is called upcasting.

```
Cat c = new Cat(); //subclass instance
Animal a = c; //upcasting, it's fine since Cat is also an Animal
```

Downcasting
When an instance of Super class is assigned to a Subclass reference, then it's called downcasting.

```
Cat c = new Cat(4, "White", false);
Animal a = c;
Cat c1 = (Cat) a;
```

Note
We can use *instanceof* instruction to check the inheritance between objects.

```
Cat c = new Cat(4, "Black", false);
Dog d = new Dog(4, "White", false);

Animal a = c;

c instanceof Cat; // true
c instanceof Animal; // true
a instanceof Cat; // true
a instanceof Dog; // false
```

Object Oriented Programming – Polymorphism

Polymorphism word was derived from the Greek word "Poly" and "morphs". Poly means "many" and morphs means "forms".

Therefore, the word polymorphism means "many forms". There are two types to polymorphism. Static polymorphism and Dynamic polymorphism.

Method Overloading is known as static polymorphism and Method Overriding is known as dynamic polymorphism.

Method Overloading

Method Overloading is known as static polymorphism. Overloading is the ability of a class to have multiple methods or constructors with same name but different in either number or type of arguments.

*Java does not have the concept of operator overloading that is, the capability of defining the behavior of the built-in operators by defining methods in your own classes.

```java
public class MethodOverloading {
      // method adding two int numbers.
      public int add(int first, int second) {
            return (first + second);
      }

      // Overloaded method adding two double type numbers.
      public double add(double first, double second) {
            return (first + second);
      }

      // Overloaded method concatenating two Strings.
      public String add(String first, String second) {
            return (first + second);
      }

      // main driven function.
      public static void main(String[] args) {
            var obj = new MethodOverloading();

            System.out.println(obj.add(32, 32));                 // 64
            System.out.println(obj.add(3.2, 3.78));              // 6.98
            System.out.println(obj.add("Muhammad ", "Naveed"));  // Muhammad Naveed
      }
}
```

We also overload the **constructors**. The constructor overloading example is given below.

```java
public class Player {
      private String userName;
      private float userHeight;
      private int userLife;
      private boolean hasWin;
```

```java
// Default Constructor.
Player() {
      userLife = 0;
      userHeight = 0.0F;
      userName = "";
      hasWin = true;
}

// Parameterized Constructor.
Player(String userName, float userHeight)     {
      this.userName = userName;
      this.userHeight = userHeight;
}

// Overloaded Parameterized Constructor.
Player(String userName, float userHeight, int userLife) {
      // Calling the above constructor using this keyword.
      this(userName, userHeight);
      this.userLife = userLife;
}

// Accessor or getter method.
public int getLife() {
      return this.userLife;
}

// Accessor or getter method.
public boolean userStatus() {
      return this.hasWin;
}

// Gain Health.
public void addHealth() {
      if (userLife < 3)
            userLife++;
}

// Loss Health.
public void loseHealth() {
      if (userLife < 3)
            userLife--;
      else
            hasWin = false;
}
```

```
      // Showing Game Results.
      public void showSummary() {
            System.out.println("Username : " + this.userName);
            System.out.println("Height : " + this.userHeight + "feets");
            System.out.println("Life : " + this.userLife);
            System.out.println("Win : " + this.userStatus());
      }

      // Main driven function.
      public static void main(String[] args) {
            var p = new Player("Naveed", 1.71F, 3);

            p.showSummary();
      }
}
```

The above are the examples of the static polymorphism.

Method Overriding

Method Overriding is known as dynamic polymorphism. In overriding we overwrite the method body provided by the parent class. In overriding we use the concepts of abstract classes and inheritance.

```
// Shape abstract class
public abstract class Shape {
      // abstract method. Remember: abstract method has no body.
      public abstract double area();
}

// Circle class inherited from shape class
public class Circle extends Shape {
      private double radius;

      public Circle(double radius) {
            this.radius = radius;
      }

      // See this annotation @Override, it is telling that this method is from
      // parent class Shape and is overridden here.
      @Override
      public double area() {
            return 3.14 * radius * radius;
      }
}
```

```java
// Rectangle class inherited from shape class
public class Rectangle extends Shape {
      private double length;
      private double width;

      Rectangle(double length, double width) {
            this.length = length;
            this.width = width;
      }

      // See this annotation @Override, it is telling that this method is from
     // parent class Shape and is overridden here.
      @Override
      public double area() {
            return length * width;
      }
}

public class Main {
      public static void main(String[] args) {
            //This will create an object of circle class
            Shape circle = new Circle(5.0);
            //This will create an object of Rectangle class
            Shape rectangle = new Rectangle(5.4, 5.2);
            System.out.println("Shape of circle : " + circle.area());
            System.out.println("Shape of rectangle: " + rectangle.area());
      }
}
```

or we can write our main method as

```java
public class AreaFinder {
      public static void main(String[] args) {
            printArea(new Circle(5.0));
            printArea(new Rectangle(5.4, 5.2));
      }

      public static void printArea(Shape shape) {
            System.out.println("The area is " + shape.area());
      }
}
```

Is not it great? Two objects of same type calling same methods and returning different values. That is the power of dynamic polymorphism.

Virtual Methods

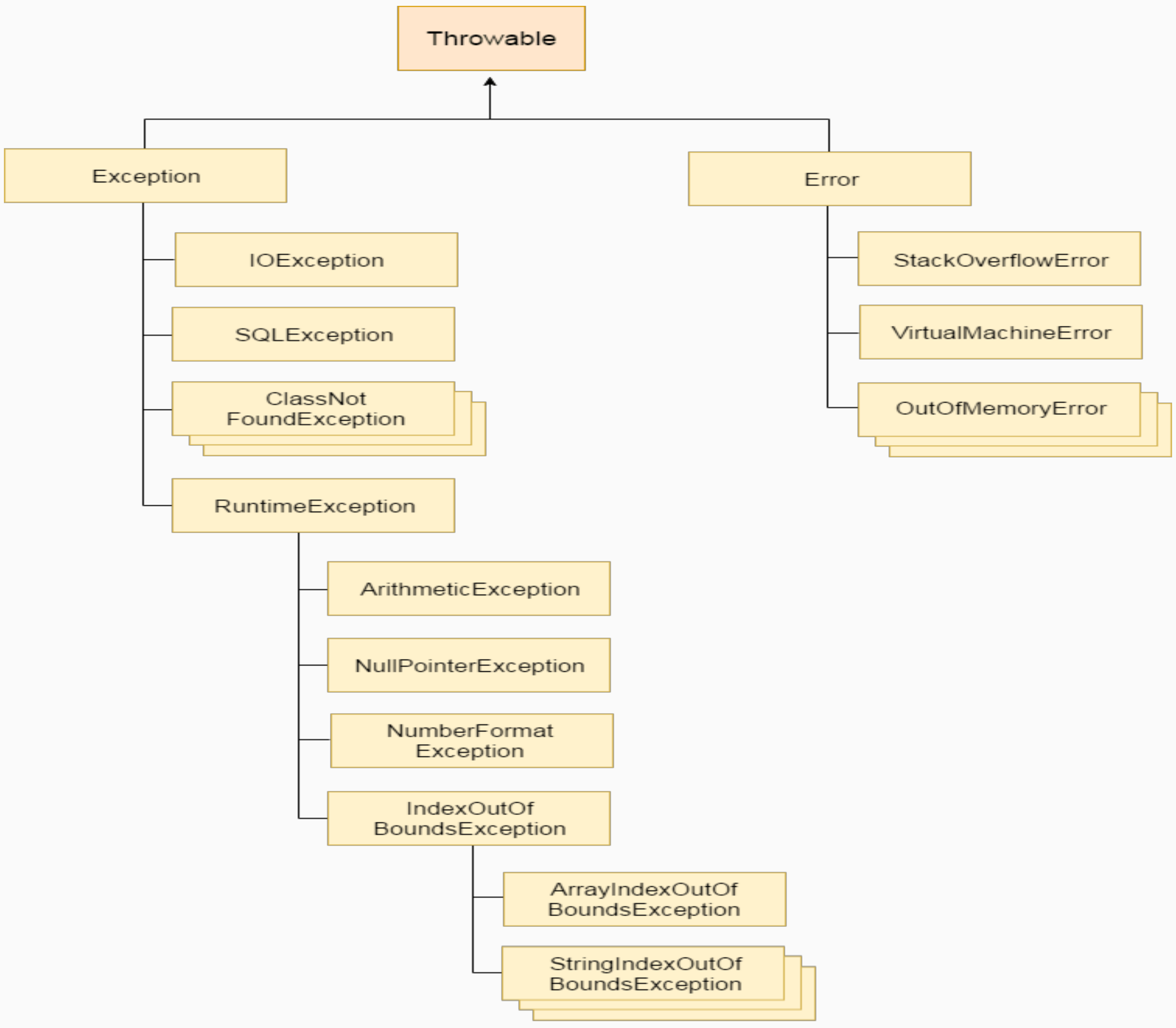
In java, all non-static methods are by default "virtual functions". Only methods marked with *final* keyword, which cannot be overridden, along with private methods, which are not inherited, are non-virtual.

In short we can say that all methods in out java classes are virtual by default. You have to go out of your way to write non-virtual functions by adding the final keyword.

Exceptions and Exception Handling

Objects of type throwable and its sub-types can be sent up to stack with the throw keyword and caught with try catch statements.

The Hierarchy of Java Exception classes is given below:



Catching an exception with try-catch

An exception can be caught and handled using the try...catch statement. Different forms of try catch block are given below.

Try-Catch with one catch block
In try-catch with one catch block has only one catch block for catching the exceptions. Here is the syntax of try catch block.

```
try {
      // some code to try
} catch (SomeException e) {
      // handling the exception here
}
// next statement
```

The statements in the try block are executed. If there is no exception thrown by the statement in the try block, then control passes to out side of the try-catch block. If an exception is thrown within the try block, the exception object is tested to see if it is an instance of Some Exception or sub type. If yes, then the catch block will catch the exception. Else the original exception continues to propagate.

Try-catch with multiple catches
A try-catch may have multiple catch blocks. The syntax for try-catch with multiple catches is given below.

```
try {
      // some code to try
} catch (SomeException e) {
      // if exception is of type SomeException then, handled here
} catch (SomeOtherException e) {
      // if exception is of type SomeOtherException then, handled here
}
// next statement
```

Multi-exception catch blocks
Java SE 7 introduces, a single catch block that can handle a list of unrelated exceptions. The exception type are listed, separated with a "|" symbol. The syntax is given below.

```
try {
      // some code to try
} catch (SomeException | SomeOtherException e) {
      // handle the exception of type SomeException and SomeOtherException here
}
```

One thing to remember here is always write first child exception catch block then, the parent Exception catch block. Let's see a quick example exception handling hierarchy.

```
import java.util.Scanner;

public class TryCatchExample {
      public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);
            String str = "the last of us";

            try {
                  System.out.print("Enter the index : ");
                  System.out.println(str.charAt(scanner.nextInt()));
            }
```

```
        // Child exception catch block.
        catch(StringIndexOutOfBoundsException e) {
            // Print the detail information of exception.
            e.printStackTrace();
        }
        // Parent exception catch block.
        // Mother of all exceptions.
        catch (Exception e) {
            // Print the detail information of exception.
            e.printStackTrace();
        }
        // finally close the open resources
        finally {
            scanner.close();
        }
    }
}
```

The finally block executes even if there is exception occur or not.

The try-with-resources statement

As the try-catch-final statement example illustrates, resource cleanup using a finally clause requires a significant amount of "boiler-plate" code to implement the edge-cases correctly. Java 7 provides a much simpler way to deal with this problem in the form of the try-with-resources statement.

A wide range of standard Java classes and interfaces implement AutoCloseable. These include,
a. InputStream, OutputStream and their subclasses.
b. Reader, Writer and their subclasses.
c. Socket and ServerSocket and their subclasses.
d. Channel and its subclasses.
e. JDBC interfaces Connection, Statement and ResultSet and their subclasses.

```
import java.util.Scanner;

public class TryWithResources {
    public static void main(String[] args) {
        try (Scanner s = new Scanner(System.in)) {
            System.out.print("Enter denominator : ");
            int denominator = s.nextInt();
            System.out.println(5 / denominator);
        } catch(ArithmeticException e) {
            System.out.println("Can't divide by zero");
        } catch (Exception e) {
            System.out.println("Unknown error occur! ");
            }
        }
}
```

in the above example you notice that we don’t close the Scanner object because, try with resource will automatically close the Scanner for us.

Enumerations

An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) or the days of the week.

Because they are constants, the names of an enum type's fields are in uppercase letters. In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days of the week enum type as:

```
public enum Day {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

We can also add the values to our enum types. Here is code example.

```
public enum Days {
        MON("Monday"),
        TUE("Tuesday"),
        WED("Wednesday"),
        THU("Thurday"),
        FRI("Friday"),
        SAT("Saturday"),
        SUN("Sunday");

        private final String value;

        Days(String value) {
                this.value = value;
        }

        public String getValue() {
                return this.value;
        }
}


public class Main {
        public static void main(String[] args) {
                Days day = Days.WED;
```

```
            // Here we are only printing the enum field name and value.
            // name → WED
            // value → Wednesday
            System.out.println(day.name());
            System.out.println(day.getValue());
      }
}
```

We can also iterate over the enum to get all our enum name and vales. Here is example.

```
// iterating over the enum.
// here one thing to notice is `values()` method is static. so, we are
// calling `values()` method as Days.values()
// not like creating the reference first such as:
// Days days = Days.WED
// for (Days day : days.values()) → here we get warning.
for (Days days : Days.values()) {
      System.out.println(days.name() + "\t" + days.getValue());
}
```

Java Annotations

An annotation is a marker which associates information with a program construct, but has no effect at run time.

In the Java programming language, an annotation is a form of syntactic metadata that can be added to Java source code. Classes, methods, variables, parameters and Java packages may be annotated. Java annotations can also be embedded in and read from Java class files generated by the Java compiler. This allows annotations to be retained by the Java virtual machine at run-time and read via reflection. It is possible to create meta-annotations out of the existing ones in Java program.

Built-in annotations

Java defines a set of annotations that are built into the language. Of the seven standard annotations, three are part of **java.lang** and the remaining four are imported from **java.lang.annotation**.

Annotations applied to Java

@Override - Checks that the method is an override. Causes a compilation error if the method is not found in one of the parent classes or implemented interfaces.

Example

```
public abstract class Animal {
      abstract void voice();
}

public class cat extends Animal {
      @Override
```

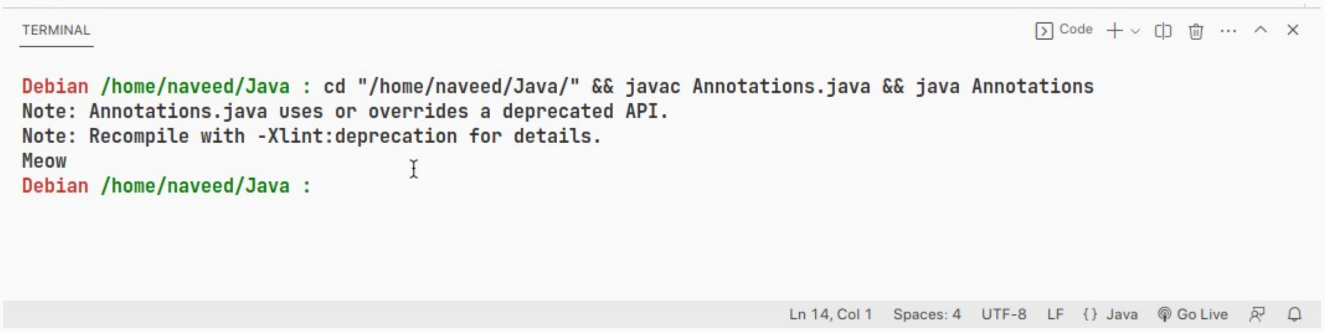
```
        void voice() {
                System.out.println("Meow");
        }
}
```

@Deprecated - Marks the method as obsolete. Causes a compile warning if the method is used.

Example

```
public abstract class Animal {
        abstract void voice();
}

public class cat extends Animal {
        @Override
        void voice() {
                System.out.println("Meow");
        }

        // here, getCatName method cause the compile warning.
        // shown in the output image.
        @Deprecated
        public String getCatName() {
                return "Kattie";
        }
}
```

Output



```
Debian /home/naveed/Java : cd "/home/naveed/Java/" && javac Annotations.java && java Annotations
Note: Annotations.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Meow
Debian /home/naveed/Java :
```

@SuppressWarnings - Instructs the compiler to suppress the compile time warnings specified in the annotation parameters.

Example

```
public class Annotations {
        public static void main(String[] args) {
                int a;
                int b;
        }
}
```

Problem

The value of the local variable a is not used.

The value of the local variable b is not used.

Solution

```
@SuppressWarnings("all")
public class Annotations {
      public static void main(String[] args) {
            int a;
            int b;
      }
}
```

As, **SuppressWarnings** Annotation is not recommended but, in some use-case we use this annotation.

Annotations applied to other annotations

These annotations are also know as "Meta Annotations".

@Retention - Specifies how the marked annotation is stored, whether in code only, compiled into the class, or available at runtime through reflection.

@Documented - Marks another annotation for inclusion in the documentation.

@Target - Marks another annotation to restrict what kind of Java elements the annotation may be applied to.

@Inherited - Marks another annotation to be inherited to subclasses of annotated class (by default annotations are not inherited by subclasses).

Annotations introduced in Java 8

@SafeVarargs - Suppress warnings for all callers of a method or constructor with a generics varargs parameter, since Java 7.

@FunctionalInterface - Specifies that the type declaration is intended to be a functional interface, since Java 8.

@Repeatable - Specifies that the annotation can be applied more than once to the same declaration, since Java 8.

There are several other annotations that we can use in our programs, we can also create our own annotation types also.

Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After serialization take place, we can convert these bytes sequence into object in memory by using process of deserialization.

we use the **java.io.Serializable** which is a marker interface (has no body). It is just used to "mark" Java classes as serializable. The serialization runtime associates with each serializable class a version number, called serialVersionUID, which is used during de-serialization to verify that the sender and receiver of a serialized object have loaded the classes. If both sender and receiver serialVersionUID is different then, the InvalidClassException should be raised.

serialVersionUID is declared as static, final and of type long.

Serialization Process

```java
import java.io.Serializable;
import java.util.Calendar;
import java.util.Date;

public class SerialClass implements Serializable {
      // serialVersionUID.
      @Serial
      private static final long serialVersionUID = 1L;
      private final Date currentTime;

      // default constructor.
      SerialClass() {
            currentTime = Calendar.getInstance().getTime();
      }

      // getter method.
      public Date getCurrentTime() {
            return currentTime;
      }
}
```

Main Class

```java
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class Main {
         public static void main(String[] args) {
             String fileName = "time.ser";
             SerialClass time = new SerialClass();

             try {
                 ObjectOutputStream output = new ObjectOutputStream(new
                 FileOutputStream(fileName));
                 output.writeObject(time);
                 output.close();
             } catch(IOException exception) {
                 exception.printStackTrace();
             }
         }
}
```

De-Serialization Process

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializationClassExample {
    public static void main(String[] args) {
        String fileName = "time.ser";
        SerialClass time = null;

        try {
            ObjectInputStream input = new ObjectInputStream(new
            FileInputStream(fileName));
            time = (SerialClass) input.readObject();
            input.close();
        } catch (IOException | ClassNotFoundException exception) {
            exception.printStackTrace();
        }
        System.out.println("Restored time : " + time.getCurrentTime());
    }
}
```

What if we want any field, not to serialized then, we use the keyword **transient** to avoid the field being serialized.

Multi threading

If you have many tasks to execute, and all these tasks are not dependent of the result of the precedent ones, you can use **Multithreading** for your program to do all these tasks at the same time. This can make your program execution faster if you have some big independent tasks.

A process is an execution environment.

Threads are lightweight processes also, process may have one or more threads. Threads does not execute in sequence, they run when the CPU is ideal. The threads have five states:

New State, when thread is created.

Runnable State, when thread is ready to move to runnable state.

Blocked/Waiting State, when thread is temporarily inactive.

Timed State, when thread wait for notification to receive.

Terminated State, when thread is terminated.

Here are some code examples of multi threading.

```
// class implementing the Runnable interface.
public class Thread1 implements Runnable {
      // Overriding the run method
      @Override
      public void run() {
            System.out.println("Thread 1 class run() is running");
      }
}

// class extending the Thread class, which also implements the Runnable interface.
public class Thread2 extends Thread {
      // Overriding the run method.
      @Override
      public void run() {
            System.out.println("Thread 2 class run() is running");
      }
}

// Main example class
public class Main {
      // main Method
      public static void main(String[] args) {
            // Creating object of Runnable implementation class.
            Thread th1 = new Thread(new Thread1());
            th1.start();
```

```
                // Creating object of Thread extending class.
                Thread2 th2 = new Thread2();
                th2.start();
        }
}
```

This is just a toy example to demonstrate the usage of Threads in your applications. There are different ways to create the threads some different ways of declaration are given below:

```
// Creating object of Runnable implementation class.
new Thread(new Thread1()).start();
// Creating object of Thread extending class
new Thread2().start();

// Creating the runnable.
Runnable r = new Runnable() {
        @Override
        public void run() {
                System.out.println("Running...");
        }
};

// Starting the thread targeting the Runnable r
new Thread(r).start();
```

On different use cases we use different ways to perform multithreading.

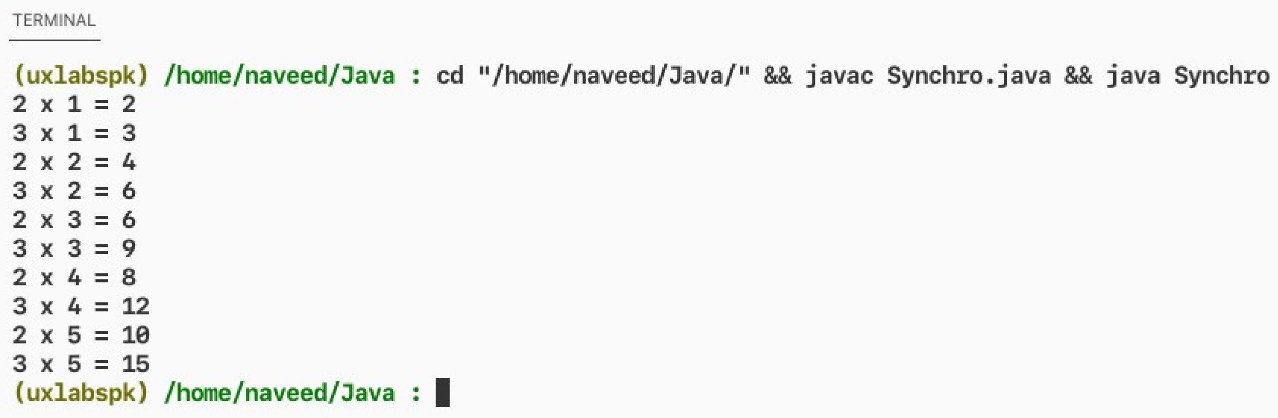
Synchronization

If two threads working on same object, they may not work well. Because, let consider one thread is updating the record and other thread is reading the data. So, the synchronization comes into action. By using synchronization we can synchronize the threads. Here is the code example without synchronization.

```
import java.text.MessageFormat;

// multiply class
public class Multiple {
        // public method to print the table.
        public void Multiply(int number) {
                for (int i = 1; i <= 5; i++) {
```

```
            System.out.println(MessageFormat.format("{0} x {1} = {2}", number, i,
            (number * i)));
            }
      }
}

// Thread class
public class Th1 extends Thread {
      Multiple m = null;
      // Parameterized constructor
      public Th1(Multiple m) {
            this.m = m;
      }

      // overridden run method
      @Override
      public void run() {
            m.Multiply(2);
      }
}

// Another thread class
public class Th2 extends Thread {
      Multiple m = null;
      // parameterized constructor
      public Th2(Multiple m) {
            this.m = m;
      }

      // Overridden run method
      @Override
      public void run() {
            m.Multiply(3);
      }
}


// main example class
public class Main {
      // main driven method
      public static void main(String[] args) {
        // creating the object of Multiple class.
        Multiple m = new Multiple();

        // Starting the th1 and th2 threads.
        new Th1(m).start();
        new Th2(m).start();
    }
}
```

here is the output of the following code. We start first th1 to print table of 2 and then start th2 to print table of 3. But you can see the output,



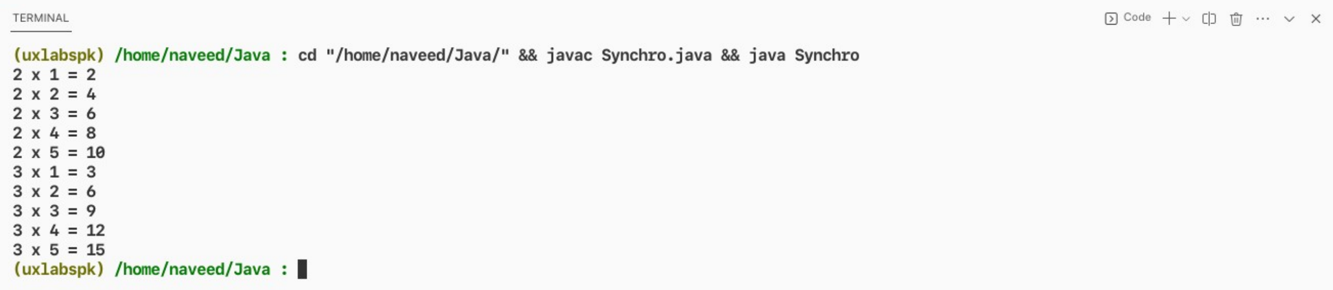
```
TERMINAL

(uxlabspk) /home/naveed/Java : cd "/home/naveed/Java/" && javac Synchro.java && java Synchro
2 x 1 = 2
3 x 1 = 3
2 x 2 = 4
3 x 2 = 6
2 x 3 = 6
3 x 3 = 9
2 x 4 = 8
3 x 4 = 12
2 x 5 = 10
3 x 5 = 15
(uxlabspk) /home/naveed/Java : █
```

To Solve this problem we add only one keyword to our method `Multiply(int number)`.

```
import java.text.MessageFormat;

// multiply class
public class Multiple {
    // public method to print the table.
    synchronized public void Multiply(int number) {
        for (int i = 1; i <= 5; i++) {
        System.out.println(MessageFormat.format("{0} x {1} = {2}", number, i,
        (number * i )));
        }
    }
}

// Thread class
public class Th1 extends Thread {
    Multiple m = null;
```

```java
    // Parameterized constructor
    public Th1(Multiple m) {
        this.m = m;
    }

    // overridden run method
    @Override
    public void run() {
        m.Multiply(2);
    }
}

// Another thread class
public class Th2 extends Thread {
    Multiple m = null;
    // parameterized constructor
    public Th2(Multiple m) {
        this.m = m;
    }

    // Overridden run method
    @Override
    public void run() {
        m.Multiply(3);
    }
}


// main example class
public class Main {
    // main driven method
    public static void main(String[] args) {
        // creating the object of Multiple class.
        Multiple m = new Multiple();

        // Starting the th1 and th2 threads.
        new Th1(m).start();
        new Th2(m).start();
    }
}
```

and the output becomes,



```
(uxlabspk) /home/naveed/Java : cd "/home/naveed/Java/" && javac Synchro.java && java Synchro
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
(uxlabspk) /home/naveed/Java :
```

Input/Output Streams

A Stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements. I/O Streams is used to perform I/O operations on files. Files may includes plain text documents. Most streams must be closed when you are done with them, otherwise you could introduce a memory leak or leave a file open. It is important that streams are closed even if an exception is thrown.

There are two main types of I/O streams;

1. Byte Stream. It is used for any type of input. It uses 8bit to load stream. It read one byte at a time.
2. Character Stream. It is used for text based input. It uses 16 bit to load stream. It read one character at a time.

Here are some code examples of I/O Streams.

```
// Byte Stream Example.
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreamExample {
      // main driven method
      public static void main(String[] args) throws IOException {
            // Input stream object.
            FileInputStream inStream = null;
            // Output stream object.
            FileOutputStream outStream = null;

            // try-catch block
            try {
                  // loading file path.
                  inStream = new FileInputStream("./samp1.txt");
                  outStream = new FileOutputStream("./samp2.txt");

                  // loop to read all data from the file.
```

```
                int content = 0;
                while ((content = inStream.read()) != -1) {
                    // writing all data to another file
                    outStream.write((byte) content);
                }
            } catch (Exception e) {
                // exception details
                e.printStackTrace();
            } finally {
                // if streams are open then close them.
                if (inStream != null && outStream != null) {
                    inStream.close();
                    outStream.close();
                }
            }
        }
    }
```

The Character Stream Example is also given below:

```
// Character Stream Example.
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CharacterStreamExample {
    // main driven method
    public static void main(String[] args) throws IOException {
        // creating FileReader, FileWriter object.
        FileReader freader = null;
        FileWriter fwriter = null;

        // try-catch block
        try {
            // loading file path.
            freader = new FileReader("./samp1.txt");
            fwriter = new FileWriter("./samp2.txt");

            // loop to read all data from the file.
            int content = 0;
            while ((content = freader.read()) != -1) {
                // writing all data to another file
                fwriter.write((char) content);
            }

        } catch (Exception e) {
            // exception details
            e.printStackTrace();
        } finally {
            // if streams are open then close them.
            if (freader != null && fwriter != null) {
                freader.close();
                fwriter.close();
            }
        }
    }
}
```

Generics

Generics are a facility of generic programming that extend Java's type system to allow a type or method to operate on objects of various types while providing compile-time type safety. In particular, the Java collections framework supports generics to specify the type of objects stored in a collection instance.

Generics enable classes, interfaces, and methods to take other classes and interfaces as type parameters.

Here is simple example of generics in java.

```
// example class.
public class Example<T> {
    // obj with T return type, T is assigned in main method.
    T obj;

    // default constructor.
    public Example() {
        obj = null;
    }

    // setter method.
    public void setObj(T obj) {
        this.obj = obj;
    }

    // getter method.
    public T getObj() {
        return this.obj;
    }
}


public class Main {
    public static void main(String[] args) {
        Example<Integer> iValue = new Example<>();
        Example<Float> fValue = new Example<>();
        Example<Double> dValue = new Example<>();
        Example<String> sValue = new Example<>();
        Example<Character> cValue = new Example<>();

        iValue.setObj(1);
        fValue.setObj(1.2F);
        dValue.setObj(3.43);
        sValue.setObj("uxlabspk");
        cValue.setObj('A');

        System.out.println(iValue.getObj());
        System.out.println(fValue.getObj());
        System.out.println(dValue.getObj());
        System.out.println(sValue.getObj());
        System.out.println(cValue.getObj());
    }
}
```

The given above example is just a toy example to demonstrate how to create the generic classes in your programs.

Extending a generic class

We can extend normal classes from the generics. Here is an example

```
// abstract generic class
public abstract class AbstractClass<T> {
    private T value;

    // getter method
    public T getValue() {
        return this.value;
    }
```

```
        // setter method
        public void setValue(T value) {
                this.value = value;
        }
}

// email class extending generic class
public class Email extends AbstractClass<String> { /* Class Body */ }
// age class extending generic class
public class Age extends AbstractClass<Integer> { /* Class Body */ }
// height class extending generic class
public class Height<T> extends AbstractClass<T> { /* Class Body */ }

// main class
public class Main {
        // main driven function
        public static void main(String[] args) {
                // creating email class reference
                Email userEmail = new Email();
                userEmail.setValue("naveed@example.com");
                System.out.println(userEmail.getValue());

                // creating age class reference
                Age userAge = new Age();
                userAge.setValue(Integer.valueOf(21));
                System.out.println(userAge.getValue());

                // creating height class reference
                Height<Integer> userHeight_cm = new Height<>();
                userHeight_cm.setValue(Integer.valueOf(172)); // 172cm
                System.out.println(userHeight_cm.getValue());

                // again creating height class object using float type
                Height<Float> userHeight_feet = new Height<>();
                userHeight_feet.setValue(Float.valueOf(5.7F)); // 5.7Feet
                System.out.println(userHeight_feet.getValue());
        }
}
```

It is also possible to instantiate with anonymous inner class with an empty curly braces ({})

```
AbstractClass<Integer> i = new AbstractClass<>() { /* Empty Class Body */ };
```

```
i.setValue(Integer.valueOf(21));
System.out.println(i.getValue());
```

Multiple type parameters

There may have more than one types a parameters in generic classes. Here is how we can use multiple type parameters.

```
// class with multiple parameters
public class MultiTypeClass<T, S> {
      private T firstParameter;
      private S secondParameter;

      // parameterized constructor
      public MultiTypeClass(T firstParameter, S secondParameter) {
            this.firstParameter = firstParameter;
            this.secondParameter = secondParameter;
      }

      // getter method
      public T getFirstParameter() {
            return firstParameter;
      }

      // setter method
      public void setFirstParameter(T firstParameter) {
            this.firstParameter = firstParameter;
      }

      // getter method
      public S getSecondParameter() {
            return secondParameter;
      }

      // setter method
      public void setSecondParameter(S secondParameter) {
            this.secondParameter = secondParameter;
      }
}
```

In main method the object creation look like this

```
MultiTypeClass<String, String> t = new MultiTypeClass<>("First", "Second");
System.out.println(t.getFirstParameter());
```

```java
System.out.println(t.getSecondParameter());
```

The Diamond

Java 7 introduced the Diamond to remove some boiler-plate around generic class instantiation. With Java 7+ you can write:

```java
List<String> skills = new LinkedList<>();
```

In old versions we create objects,

```java
List<String> skills = new LinkedList<String>();
```

Declaring generic methods

We can also declare the generic methods. Here is the example of generic methods,

```java
public class GenericMethodExample {
      // here <T> is type.
      // void is return type.
      // show is name
      // arg is parameter of type T (generic)
      public <T> void show(T arg) {
            System.out.println(arg);
      }

      public static void main(String[] args) {
            GenericMethodExample g = new GenericMethodExample();
            g.show(232);
            g.show(23.2F);
            g.show("string");
            g.show('c');
      }
}
```

Networking

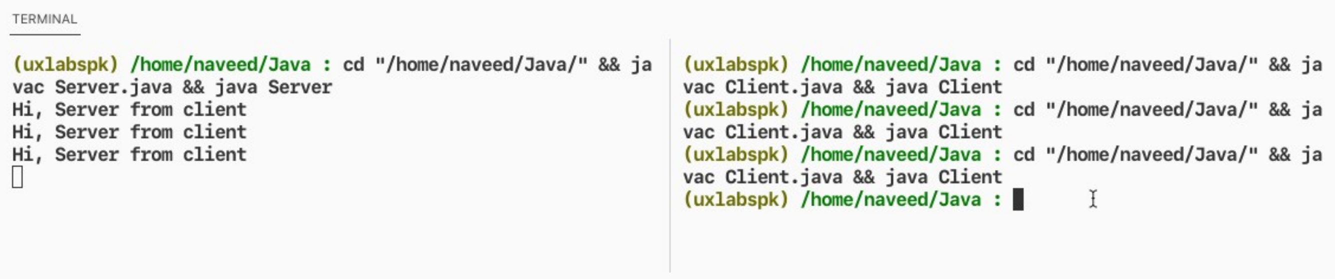
Java also support networking. As, this is a programming guide so, we are not going deeper in networking concepts but, here I am going to show you how to work with Basic Client and Server Communication using a Socket only.

here is the example of basic client and server communication using sockets.

```java
// Complete Server Example.
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Server {
	// main driven method
	public static void main(String[] args) {
		// "try-with-resources" will close the socket once we leave its scope.
		try (ServerSocket server = new ServerSocket(12345)) {
			while(true) {
				// waiting for clients connection.
				Socket clientSocket = server.accept();

				// creating new thread for handling the new clients
				new Thread(() -> {
					try {
						// reading bytes from the socket.
						InputStream in = clientSocket.getInputStream();
						// wrap the Input Stream in a reader to read a
						// String instead of bytes.
						BufferedReader reader = new BufferedReader(new
						InputStreamReader(in, StandardCharsets.UTF_8));

						// reading the socket and print on the console.
						String line = null;
						while((line = reader.readLine()) != null) {
							// printing to the console.
							System.out.println(line);
						}

					} catch (Exception e) {
						// Exception => the mother of all exceptions.
						e.printStackTrace();
					}
				}).start();
			}
		} catch (Exception e) {
			// Exception => the mother of all exceptions.
			e.printStackTrace();
		}
	}
}
```

```java
// Complete Client Example.
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Client {
      // main driven method
      public static void main(String[] args) {
            // "try-with-resources" will close the socket once we leave its scope.
            try(Socket socket = new Socket("127.0.0.1", 12345)) {
                  // Writing bytes to the socket.
                  OutputStream out = socket.getOutputStream();
                  // wrap the Output Stream in a writer to write a String instead of
                  // bytes.
                  PrintWriter writer = new PrintWriter(new OutputStreamWriter(out,
                  StandardCharsets.UTF_8));
                  // Writing message to the socket.
                  writer.println("Hi, Server from client");
                  writer.flush();
            } catch (Exception e) {
                  // Exception => the mother of all exceptions.
                  e.printStackTrace();
            }
      }
}
```

Here is the output of our client server communication. Left side is server and right side is client.



TERMINAL

```
(uxlabspk) /home/naveed/Java : cd "/home/naveed/Java/" && ja
vac Server.java && java Server
Hi, Server from client
Hi, Server from client
Hi, Server from client
```

```
(uxlabspk) /home/naveed/Java : cd "/home/naveed/Java/" && ja
vac Client.java && java Client
(uxlabspk) /home/naveed/Java : cd "/home/naveed/Java/" && ja
vac Client.java && java Client
(uxlabspk) /home/naveed/Java : cd "/home/naveed/Java/" && ja
vac Client.java && java Client
(uxlabspk) /home/naveed/Java :
```

Concurrent Programming

Concurrent computing is a form of computing in which several computations are executed concurrently instead of sequentially. Java language is designed to support concurrent programming through the usage of threads.

Executor, ExecutorService and Thread pools

The Executor interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc.

Normally we use the Executor instead of explicitly creating threads. By using the Executors, we don't have to significantly rewrite our code to be able to easily tune our program's task execution policy.

ExecutorService is an interface that extends the Executor interface, which also create the thread pool executor.

A common Executor used is **ThreadPoolExecutor**, which takes care of Thread handling. We can configure the minimal amount of Threads and maximum amount of Threads. When there is more work to do, Pool can grow and Once the workload declines, Pool slowly reduces the Thread count again to its min size.

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
      1,    // keep at least one thread ready, even if no Runnables are executed.
      5,    // at most five Runnables/Threads executed in parallel.
      1,    // idle Threads terminated after one minute, when min Pool size exceeded.
      TimeUnit.MINUTES,
      new ArrayBlockingQueue<Runnable>(10) // outstanding Runnables are kept here.
);

pool.execute(new Runnable() {
      @Override
      public void run() {
            // Code to run
            System.out.println("Performing some task");
      }
});
```

Note If you configure the ThreadPoolExecutor with an unbounded queue, then the thread count will not exceed corePoolSize since new threads are only created if the queue is full.

Callable

Runnable has a limitation in that it can't return a result from the execution. The only way to get some return value from the execution of a Runnable is to assign the result to a variable accessible in a scope outside of the Runnable.

Callable is same as Runnable but, it has a call method instead of run method. The call method has the additional capability to return a result and is also allowed to throw checked exceptions.

Future

Future can be considered a container of sorts that houses the result of the Callable computation. Computation of the callable can carry on in another thread, and any attempt to tap the result of a Future will block and will only return the result once it is available.

In short the Future interface is responsible for holding the results from the Callable interface implementation.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

// implements Callable interface of type String.
public class Calculation implements Callable<String> {
      @Override
      public String call() throws Exception {
            // sleep for 2 seconds.
            Thread.sleep(2000);
            return "Result";
      }
}

public class Main {
      public static void main(String[] args) throws Exception {
            ExecutorService executorService = Executors.newSingleThreadExecutor();

            // storing the result of callable in future.
            Future<String> result = executorService.submit(new Calculation());

            try {
                  System.out.println(result.get());
                  executorService.shutdown();
            } catch (Exception e) {
                  // if an exception occur we cancel the execution.
                  result.cancel(true);
            }
      }
}
```

CountDownLatch

A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

```
import java.util.concurrent.CountDownLatch;
```

```
public class PrintingThread implements Runnable {
      CountDownLatch latch;
      // parameterized constructor
      public PrintingThread(CountDownLatch latch) {
            this.latch = latch;
      }

      @Override
      public void run() {
            try {
                  System.out.println("Printing...");
                  latch.countDown();
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}


public class Main {
      public static void main(String[] args) {
            try {
                  int numOfThreads = 10;
                  CountDownLatch latch = new CountDownLatch(numOfThreads);

                  for (int i = 0; i < numOfThreads; i++) {
                        new Thread(new PrintingThread(latch)).start();
                  }
                  latch.await();
                  System.out.println("After completing " + numOfThreads + " threads 
                  in main.");
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

Here, CountDownLatch is initialized with a counter of 10 in Main thread. Main thread is waiting by using await() method. Ten instances of PrintingThread have been created. Each instance decremented the counter with countDown() method. Once the counter becomes zero, Main thread will resume.

In this section I discuss some of the methods of concurrent programming, which will help you to get started in concurrent programming. Be sure to check the official documentation for further details in concurrent programming.

Regular Expressions

A regular expression is a special sequence of characters that helps in matching or finding other strings or sets of strings, using a specialized syntax held in a pattern. Java has support for regular expression usage through the java.util.regex package. This topic is to introduce and help developers understand more with examples on how Regular Expressions must be used in Java. A simple email validation program is given.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
      public static void main(String[] args) {
            String emailRE = "[a-zA-Z0-9._-]+@[a-z]+\\.+[a-z]+";
            Pattern emailPattern = Pattern.compile(emailRE);
            Matcher matcher = emailPattern.matcher("muhammadnaveedcis@gmail.com");

            // printing whether email is valid or not.
            System.out.println((matcher.find()) ? "Valid email address" : "Invalid
            email address");
      }
}
```

The given above example shows how we can use the Regular Expressions in our java program. This is very simple example, we can learn different Regular Expressions or learn how to create Regular Expressions.

We can compile our Regular Expression by passing different flags such as given below.

```
Pattern emailPattern = Pattern.compile(emailRE, Pattern.CASE_INSENSITIVE);
```

There are different Regular Expressions we can use to validate different patterns.