JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

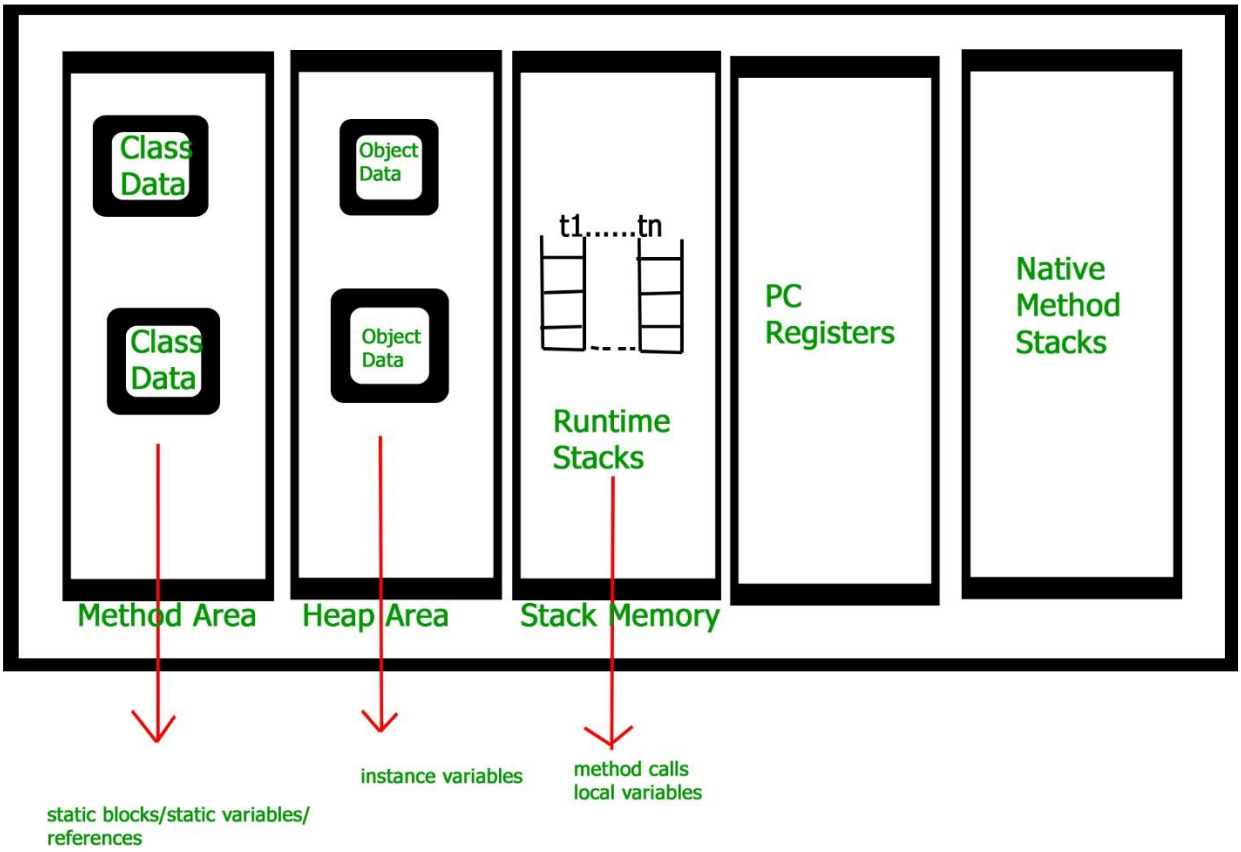
The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM Memory

- **Method area :**In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

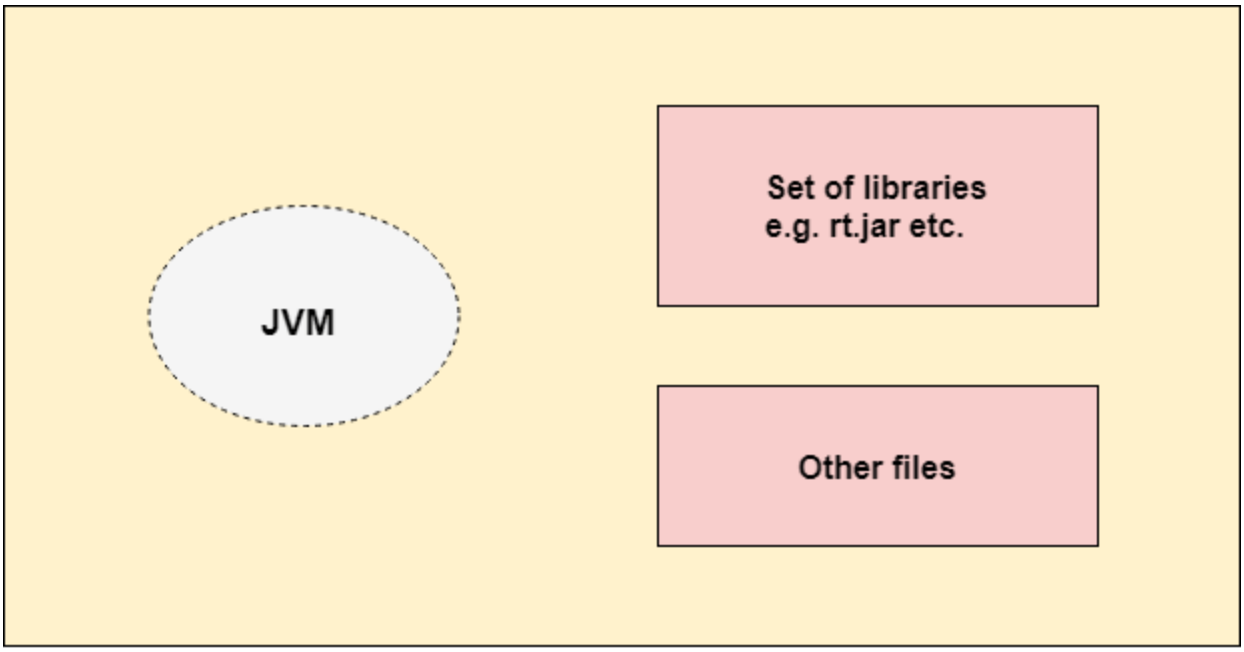
- **Heap area:** Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.
- **Stack area:** For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it’s run-time stack will be destroyed by JVM. It is not a shared resource.
- **PC Register:** Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.
- **Native method stacks:** For every thread, separate native stack is created. It stores native method information.



JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JRE

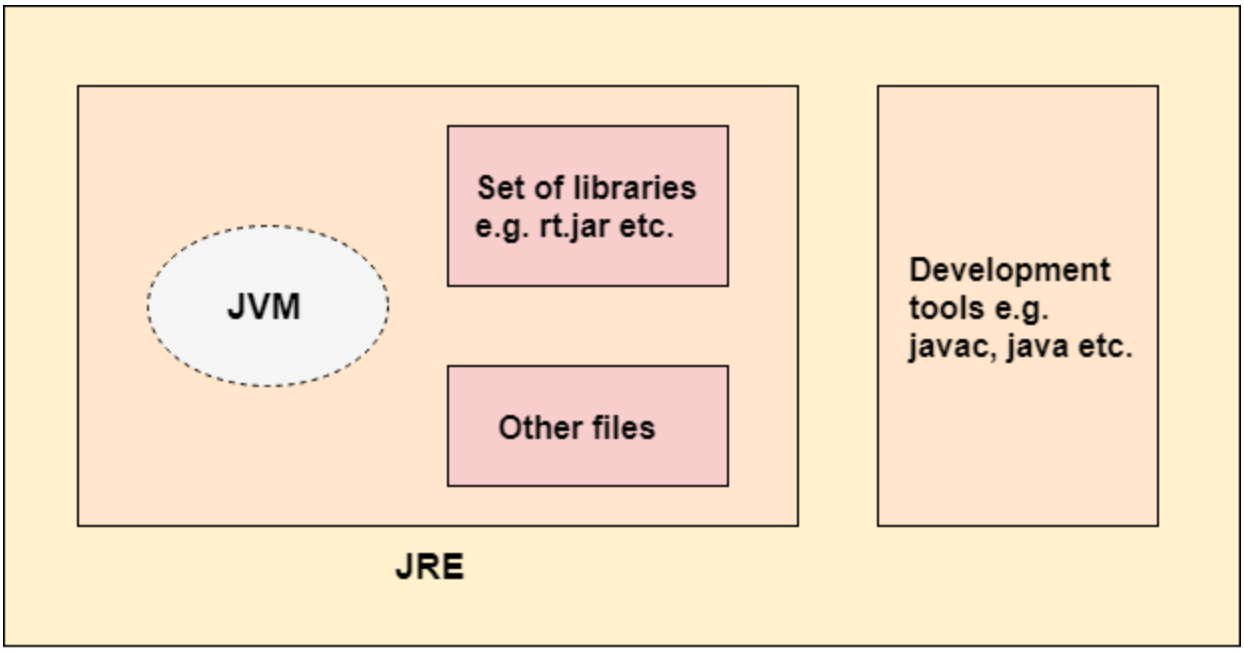
JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



Data types with size and range

Java Primitive Data Types				
Type	**Values**	**Default**	**Size**	**Range**
byte	signed integers	0	8 bits	-128 to 127
short	signed integers	0	16 bits	-32768 to 32767
int	signed integers	0	32 bits	-2147483648 to 2147483647
long	signed integers	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	+/-1.4E-45 to +/-3.4028235E+38, +/-infinity, +/-0, NAN
double	IEEE 754 floating point	0.0	64 bits	+/-4.9E-324 to +/-1.7976931348623157E+308, +/-infinity, +/-0, NaN
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
boolean	`true, false`	`false`	1 bit used in 32 bit integer	NA

- **Types of operator**

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>

Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

static keyword

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

4) Static Class

A class can be made **static** only if it is a nested class.

1. Nested static class doesn't need reference of Outer class
2. A static class cannot access non-static members of the Outer class

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

2) Java final method

If you make any method as final, you cannot override it.

3) Java final class

If you make any class as final, you cannot extend it.

Access Modifiers

	ACCESS LEVELS			
MODIFIER	Class	Package	Subclass	Everywhere
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

java class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers** : A class can be public or has default access .
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

1. Object

Object is the real world entity.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Following are some ways in which you can create objects in Java:

1) Using new Keyword: Using new keyword is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way. By using this method we can call any constructor we want to call (no argument or parameterized constructors).

-

```
// Java program to illustrate creation of Object
// using new keyword
public class NewKeywordExample
{
    String name = "GeeksForGeeks";
    public static void main(String[] args)
    {
        // Here we are creating Object of
        // NewKeywordExample using new keyword
```

```
        NewKeywordExample obj = new NewKeywordExample();
        System.out.println(obj.name);
    }
}
```

- Output:
- GeeksForGeeks

2) Using New Instance : If we know the name of the class & if it has a public default constructor we can create an object –**Class.forName**. We can use it to create the Object of a Class. Class.forName actually loads the Class in Java but doesn't create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.

```
// Java program to illustrate creation of Object

// using new Instance

public class NewInstanceExample

{

    String name = "GeeksForGeeks";

    public static void main(String[] args)

    {

        try

        {

            Class cls = Class.forName("NewInstanceExample");

            NewInstanceExample obj =

                    (NewInstanceExample) cls.newInstance();

            System.out.println(obj.name);

        }

        catch (ClassNotFoundException e)

        {

            e.printStackTrace();

        }

        catch (InstantiationException e)

        {
```

```
            e.printStackTrace();
        }
        catch (IllegalAccessException e)
        {
            e.printStackTrace();
        }
    }
}
```

Output:

```
GeeksForGeeks
```

3) Using clone() method: Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.
To use clone() method on an object we need to implement **Cloneable** and define the clone() method in it.

```
// Java program to illustrate creation of Object
// using clone() method
public class CloneExample implements Cloneable
{
    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
    String name = "GeeksForGeeks";

    public static void main(String[] args)
    {
        CloneExample obj1 = new CloneExample();
```

```
        try
        {
            CloneExample obj2 = (CloneExample) obj1.clone();
            System.out.println(obj2.name);
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
    }
}
```

Output:

```
GeeksForGeeks
```

Note :

- Here we are creating the clone of an existing Object and not any new Object.
- Class need to implement Cloneable Interface otherwise it will throw **CloneNotSupportedException**.

4) Using deserialization : Whenever we serialize and then deserialize an object, JVM creates a separate object. In **deserialization**, JVM doesn't use any constructor to create the object.
To deserialize an object we need to implement the Serializable interface in the class.
Serializing an Object :

```
// Java program to illustrate Serializing
// an Object.
import java.io.*;


class DeserializationExample implements Serializable
{
    private String name;
    DeserializationExample(String name)
```

```
    {
        this.name = name;
    }

    public static void main(String[] args)
    {
        try
        {
            DeserializationExample d =
                    new DeserializationExample("GeeksForGeeks");
            FileOutputStream f = new FileOutputStream("file.txt");
            ObjectOutputStream oos = new ObjectOutputStream(f);
            oos.writeObject(d);
            oos.close();
            f.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Object of DeserializationExample class is serialized using writeObject() method and written to file.txt file.

Deserialization of Object :

```
// Java program to illustrate creation of Object
// using Deserialization.
import java.io.*;
```

```
public class DeserializationExample
{
    public static void main(String[] args)
    {
        try
        {
            DeserializationExample d;
            FileInputStream f = new FileInputStream("file.txt");
            ObjectInputStream oos = new ObjectInputStream(f);
            d = (DeserializationExample)oos.readObject();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        System.out.println(d.name);
    }
}
```

Output:

```
GeeksForGeeks
```

5) Using newInstance() method of Constructor class : This is similar to the newInstance() method of a class. There is one newInstance() method in the **java.lang.reflect.Constructor** class which we can use to create objects. It can also call parameterized constructor, and private constructor by using this newInstance() method.
Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class internally uses newInstance() method of Constructor

```
// Java program to illustrate creation of Object
// using newInstance() method of Constructor class
```

```
import java.lang.reflect.*;

public class ReflectionExample
{
    private String name;
    ReflectionExample()
    {
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public static void main(String[] args)
    {
        try
        {
            Constructor<ReflectionExample> constructor
                = ReflectionExample.class.getDeclaredConstructor();
            ReflectionExample r = constructor.newInstance();
            r.setName("GeeksForGeeks");
            System.out.println(r.name);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Output:

```
GeeksForGeeks
```

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Points to Remember

- 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.

Interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.
- **Key points:** Here are the key points to remember about interfaces:
  1) We can't instantiate an interface in java. That means we cannot create the object of an interface
- 2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.
- 3) implements keyword is used by classes to implement an interface.
- 4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.
- 5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
- 6) Interface cannot be declared as private, protected or transient.
- 7) All the interface methods are by default **abstract and public**.
- 8) Variables declared in interface are **public, static and final** by default.

```
interface Try
{
   int a=10;
   public int a=10;
   public static final int a=10;
   final int a=10;
   static int a=0;
}
```

- All of the above statements are identical.
- 9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

```
interface Try
{
      int x;//Compile-time error
}
```

- Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.
- 10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface "Try" which has a variable x. When we tried to set the

value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

```
class Sample implements Try
{
  public static void main(String args[])
  {
    x=20; //compile time error
  }
}
```

- 11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.
- 12) A **class** can implement any **number of interfaces**.
- 13) If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
   public void aaa();
}
interface B
{
   public void aaa();
}
class Central implements A,B
{
   public void aaa()
   {
       //Any Code here
   }
   public static void main(String args[])
   {
       //Statements
    }
}
```

- 14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A
{
   public void aaa();
}
interface B
{
   public int aaa();
}
```

```
class Central implements A,B
{

   public void aaa() // error
   {
   }
   public int aaa() // error
   {
   }
   public static void main(String args[])
   {

   }
}
```

15) Variable names conflicts can be resolved by interface name.

```
interface A
{
    int x=10;
}
interface B
{
    int x=100;
}
class Hello implements A,B
{
    public static void Main(String args[])
    {
       /* reference to x is ambiguous both variables are x
        * so we are using interface name to resolve the
        * variable
        */
       System.out.println(x);
       System.out.println(A.x);
       System.out.println(B.x);
    }
}
```

Advantages of interface in java:

Advantages of using interfaces are as follows:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

	Abstract Class	**Interface**
1	An abstract class can extend only one class or one abstract class at a time	An interface can extend any number of interfaces at a time
2	An abstract class can extend another concrete (regular) class or abstract class	An interface can only extend another interface
3	An abstract class can have both abstract and concrete methods	An interface can have only abstract methods
4	In abstract class keyword “abstract” is mandatory to declare a method as an abstract	In an interface keyword “abstract” is optional to declare a method as an abstract
5	An abstract class can have protected and public abstract methods	An interface can have only have public abstract methods

6	An abstract class can have static, final or static final variable with any access specifier	interface can only have public static final (constant) variable

Abstraction in Java

Data Abstraction is the property by virtue of which only the essential details are displayed to the user.The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Inheritance

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Child Class:

The class that extends the features of another class is known as child class, sub class or derived class.

- **Parent Class:**
  The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

- Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

Inheritance allows us to reuse of code, it improves reusability in your java application.
Note: The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.

- This means that the data members(instance variables) and methods of the parent class can be used in the child class as.

- If you are finding it difficult to understand what is class and object then refer the guide that I have shared on object oriented programming: OOPs Concepts

- Lets back to the topic:

- Syntax: Inheritance in Java

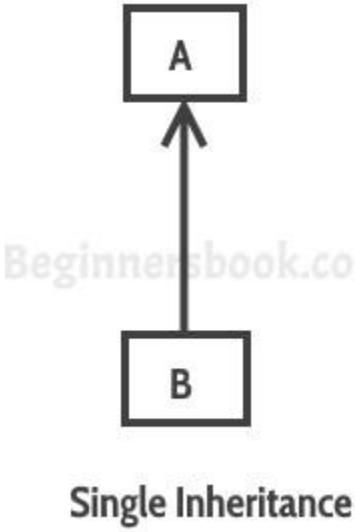
- To inherit a class we use extends keyword. Here class XYZ is child class and class ABC is parent class. The class XYZ is inheriting the properties and methods of ABC class.

```
class XYZ extends ABC
{
}
```

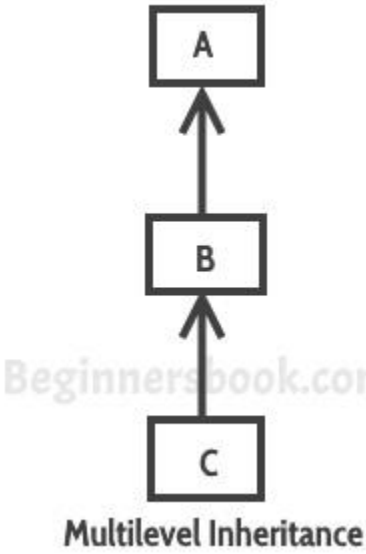
Types of inheritance

To learn types of inheritance in detail, refer: Types of Inheritance in Java.
Single Inheritance: refers to a child and parent class relationship where a class

extends the another class.

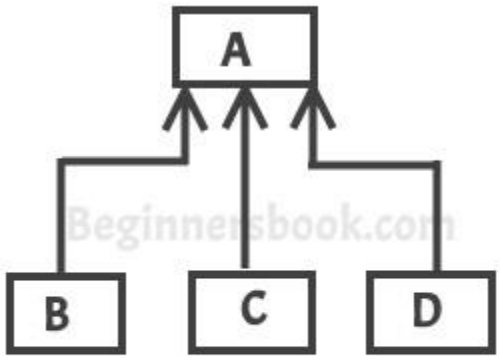


Multilevel inheritance: refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.



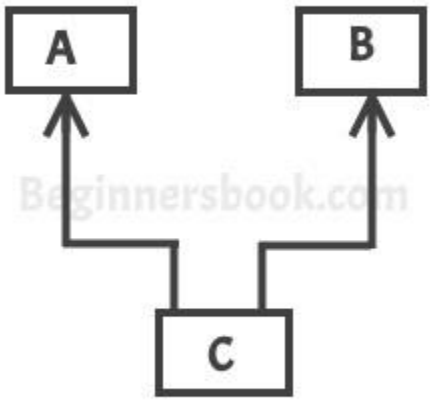
Hierarchical inheritance: refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same

class A.



Hierarchical Inheritance

Multiple Inheritance: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example class C extends both classes A and B. Java doesn't support multiple inheritance, read more about it here.



Multiple Inheritance

Hybrid inheritance: Combination of more than one types of inheritance in a single program. For example class A & B extends class C and another class D extends class A then this is a hybrid inheritance example because it is a combination of single and hierarchical inheritance.

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.
- **Polymorphism**: There are two types of polymorphism in java:
  1) **Static Polymorphism** also known as compile time polymorphism
  2) **Dynamic Polymorphism** also known as runtime polymorphism

- Compile time Polymorphism (or Static polymorphism)

- Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism. **Method Overloading**: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters. We have already discussed Method overloading here: If you didn't read that guide, refer: Method Overloading in Java

- **Example of static Polymorphism**

- Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

```
class SimpleCalculator
{
    int add(int a, int b)
    {
        return a+b;
    }
    int  add(int a, int b, int c)
    {
        return a+b+c;
    }
}
public class Demo
{
```

```
    public static void main(String args[])
    {
            SimpleCalculator obj = new SimpleCalculator();
        System.out.println(obj.add(10, 20));
        System.out.println(obj.add(10, 20, 30));
    }
}
```

Output:

```
30
60
```

Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism. I have already discussed method overriding in detail in a separate tutorial, refer it: Method Overriding in Java.

Example
In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

To understand the concept of overriding, you should have the basic knowledge of inheritance in Java.

```
class ABC{
    public void myMethod(){
        System.out.println("Overridden Method");
    }
}
public class XYZ extends ABC{

    public void myMethod(){
        System.out.println("Overriding Method");
    }
    public static void main(String args[]){
        ABC obj = new XYZ();
        obj.myMethod();
    }
```

- }
- **Output:**

- Overriding Method
- When an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time.
  Since both the classes, child class and parent class have the same method animalSound. Which version of the method(child class or parent class) will be called is determined at runtime by JVM.

Encapsulation

Encapsulation simply means binding object state(fields) and behavior(methods) together. If you are creating class, you are doing encapsulation.

The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.

How to implement encapsulation in java:
1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
2) Have getter and setter methods in the class to set and get the values of the fields.

Advantages of encapsulation

1. It improves maintainability and flexibility and re-usability: for e.g. In the above code the implementation code of void setEmpName(String name) and String getEmpName() can be changed at any point of time. Since the implementation is purely hidden for outside classes they would still be accessing the private field empName using the same methods (setEmpName(String name) and getEmpName()). Hence the code can be maintained at any point of time without breaking the classes that uses the code. This improves the re-usability of the underlying class.
2. The fields can be made read-only (If we don't define setter methods in the class) or write-only (If we don't define the getter methods in the class). For e.g. If we have a field(or variable) that we don't want to be changed so we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.

3. User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call set method and to read a field call get method but what these set and get methods are doing is purely hidden from them.

Encapsulation is also known as "**data Hiding**".

Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. **Abstraction**- Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	4. **Encapsulation**- Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

Java String

In Java, string is basically an object that represents sequence of char values.
An array of characters works same as Java string. For example:

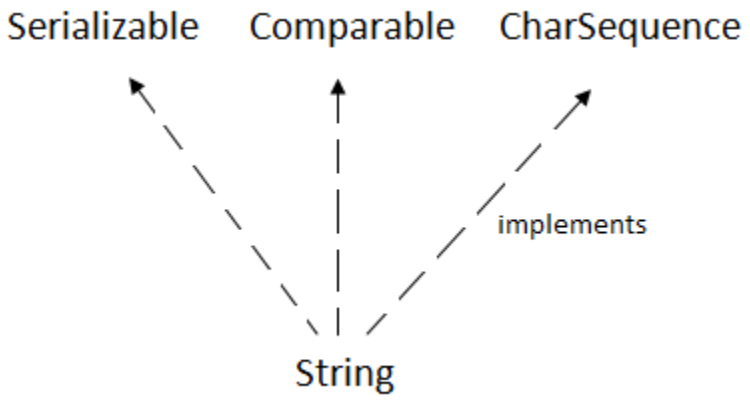
```
1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);
```

is same as:

1. String s="javatpoint";

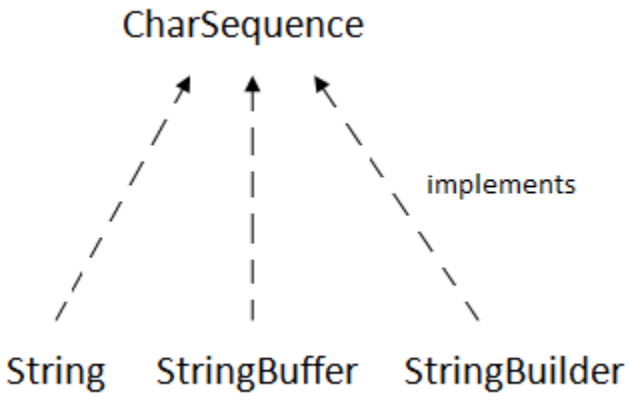
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class
implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what is String in Java and how to create the String object.

What is String in java

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

---

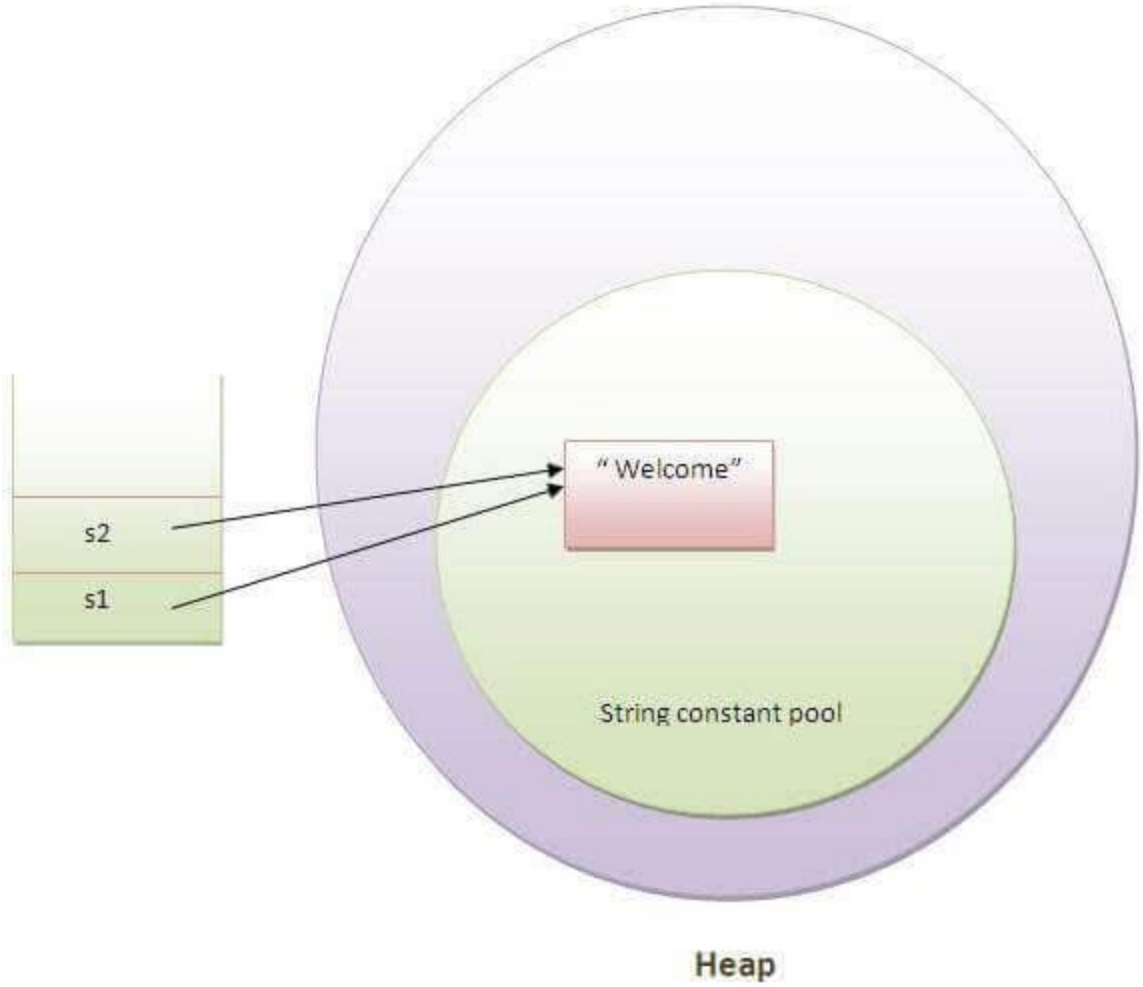
1) String Literal

Java String literal is created by using double quotes. For Example:

```
1. String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance
```



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

---

2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

```
1. public class StringExample{
2. public static void main(String args[]){
3. String s1="java";//creating string by java string literal
4. char ch[]={'s','t','r','i','n','g','s'};
5. String s2=new String(ch);//converting char array to string
6. String s3=new String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}
```

Test it Now

```
java
strings
example
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.

4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.
5	String substring(int beginIndex)	returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string.
10	boolean equals(Object another)	checks the equality of string with the given object.
11	boolean isEmpty()	checks if string is empty.
12	String concat(String str)	concatenates the specified string.

13	String replace(char old, char new)	replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns a split string matching regex.
17	String[] split(String regex, int limit)	returns a split string matching regex and limit.
18	String intern()	returns an interned string.
19	int indexOf(int ch)	returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
21	int indexOf(String substring)	returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
23	String toLowerCase()	returns a string in lowercase.
24	String toLowerCase(Locale l)	returns a string in lowercase using specified locale.

25	String toUpperCase()	returns a string in uppercase.
26	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is an overloaded method.

Parameter	String	StringBuffer	StringBuilder
Storage	String Pool	Heap	Heap
Introduced	Jdk 1.0	Jdk 1.0	Jdk 1.5
Mutability	Immutable	Mutable	Mutable
Thread Safe	No	Yes	No
Performance	Slow	Slower than StringBuilder but faster than String	Faster than StringBuffer
Overriden method	Hashcode(),toString(),equals()	toString()	toString()
Concatenation operator ie. +	allowed	Not allowed	Not allowed
Syntax	Literals: String var =“Edureka”; Obj: String var=new String(“Edureka”); With and without new keyword	StringBuffer var = new StringBuffer("Edureka"); With new keyword only	StringBuilder var = new StringBuilder("Edureka"); With new keyword only

Object class

- The Object class provides many methods. They are as follows:

Method	Description

public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the 8 digit hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Thread

- **Multithreading in Java** is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Creating a thread in Java

There are two ways to create a thread in Java:
1) By extending Thread class.
2) By implementing Runnable interface.

Before we begin with the programs(code) of creating threads, let's have a look at these methods of Thread class. We have used few of these methods in the example below.

- getName(): It is used for Obtaining a thread's name
- getPriority(): Obtain a thread's priority
- isAlive(): Determine if a thread is still running
- join(): Wait for a thread to terminate
- run(): Entry point for the thread
- sleep(): suspend a thread for a period of time
- start(): start a thread by calling its run() method

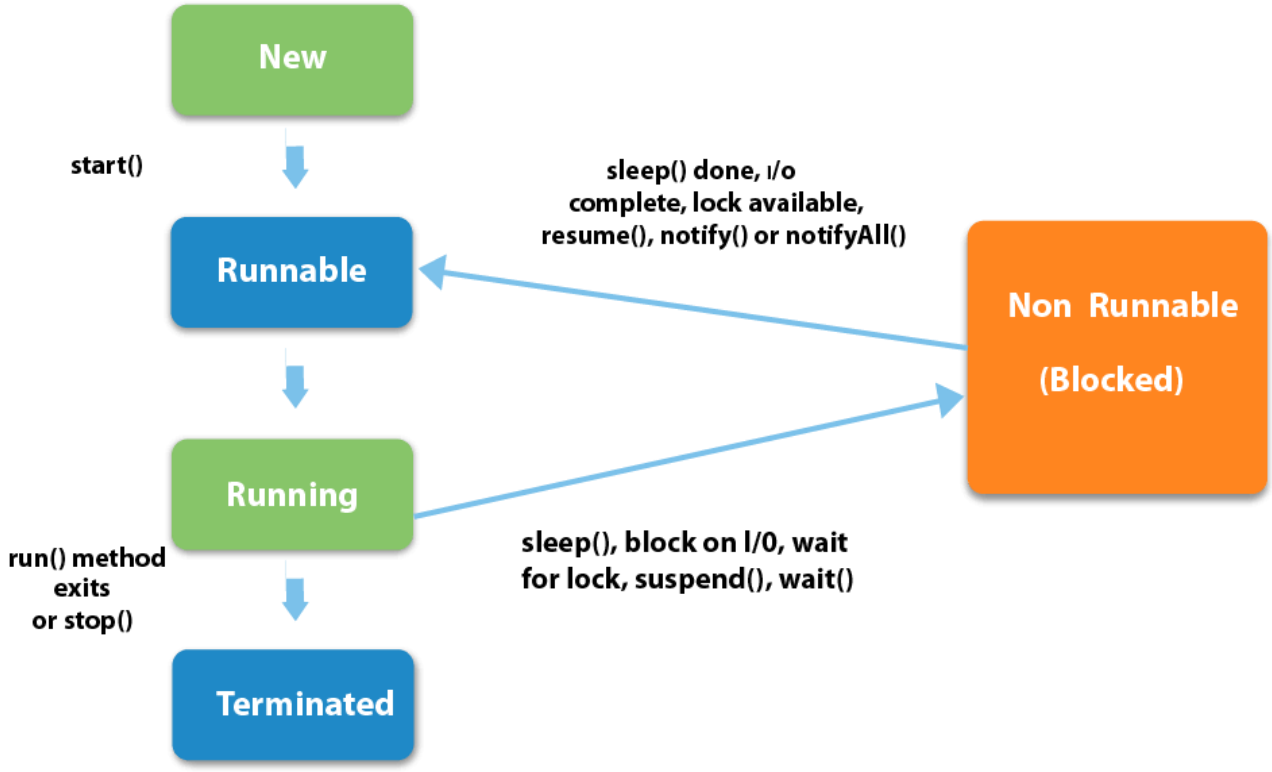
Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycl** non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start()

method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

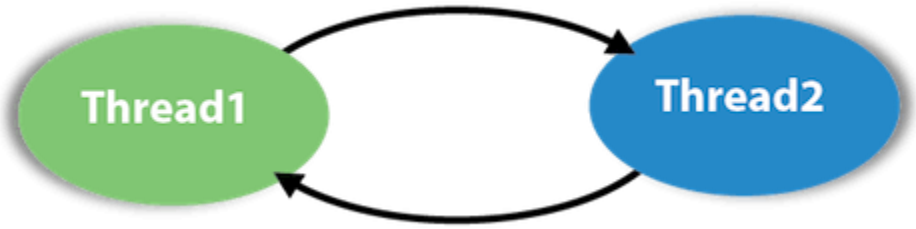
Race condition

Whenever multiple thread try to access the same resource at the same time then **RACE CONDITION** will occur.

To overcome this problem we should make use of synchronized keyword. Synchronized keyword work with object lock mechanism. When multiple thread try to access the same resource then it allow only one thread to execute and JVM will provide the object lock to the thread.

Deadlock in java

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Exception handling

- What is an exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:
An exception generated by the system is given below

```
 Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
 ExceptionDemo : The class name
 main : The method name
 ExceptionDemo.java : The filename
 java:5 : Line number
```

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

Advantage of exception handling

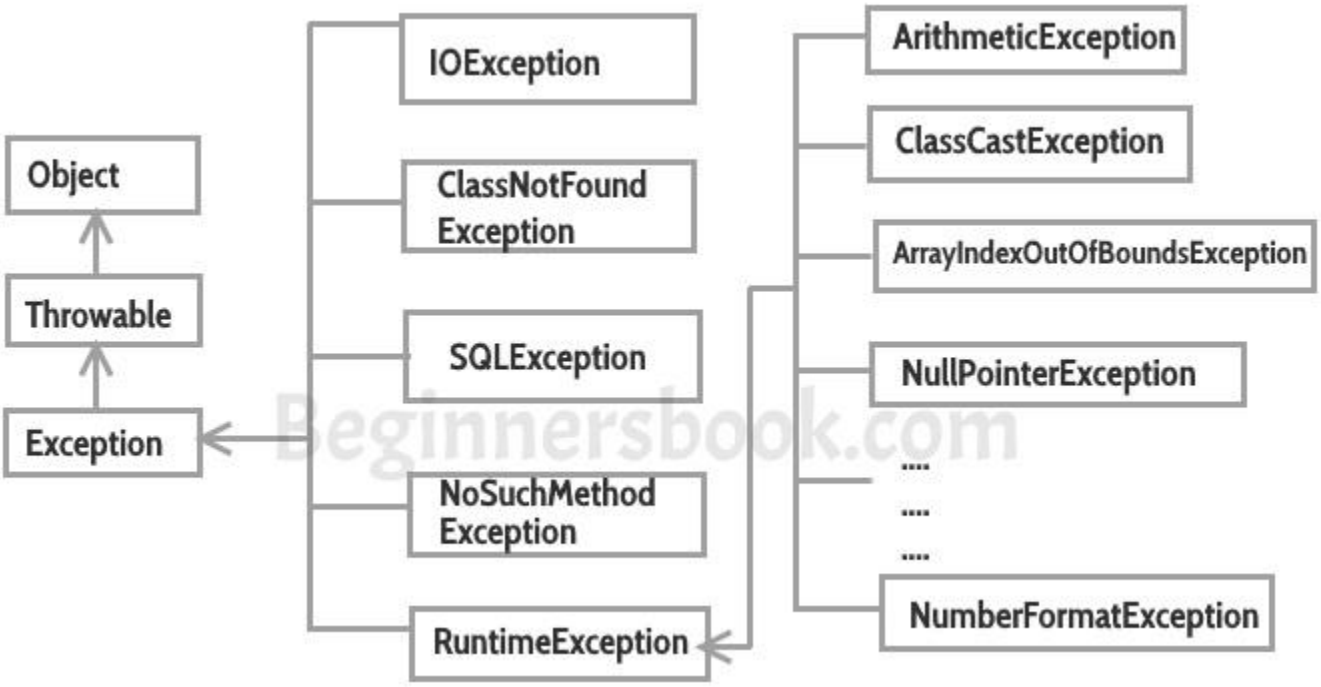
Exception handling ensures that the flow of the program doesn't break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.
By handling we make sure that all the statements execute and the flow of program doesn't break.

Difference between error and exception

Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:
NullPointerException – When you try to use a reference that points to null.
ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.



Types of exceptions

There are two types of exceptions in Java:
1)Checked exceptions
2)Unchecked exceptions

I have covered this in detail in a separate tutorial: Checked and Unchecked exceptions in Java.

Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has

handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

File handling

What is File Handling in Java?

File handling in Java implies reading from and writing data to a file. The File class from the **java.io package**, allows us to work with different formats of files. In order to use the File class, you need to create an object of the class and specify the filename or directory name.

For example:

```
1  // Import the File class
2  import java.io.File
3
4  // Specify the filename
```

5	File obj = new File("filename.txt");

Java uses the concept of a stream to make I/O operations on a file. So let's now understand what is a Stream in Java.

What is a Stream?

In Java, Stream is a sequence of data which can be of two types.

1. Byte Stream

This mainly incorporates with byte data. When an input is provided and executed with byte data, then it is called the file handling process with a byte stream.

2. Character Stream

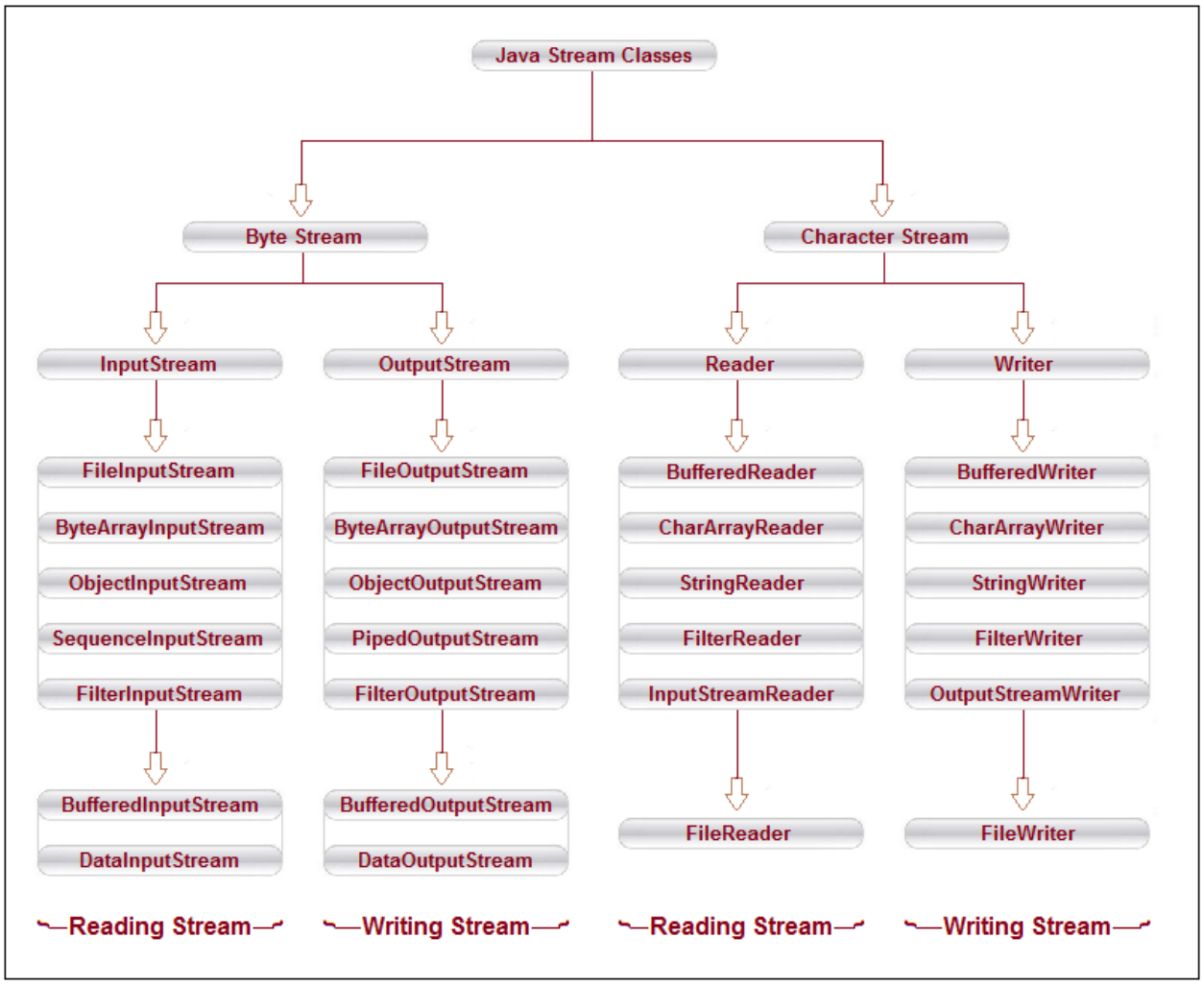
Character Stream is a stream which incorporates with characters. Processing of input data with character is called the file handling process with a character stream.

Now that you know what is a stream, let's dive deeper into this article on File Handling in Java and know the various methods that are useful for operations on the files like creating, reading and writing.

Java File Methods
Below table depicts the various methods that are used for performing operations on Java files.

Method	Type	Description

canRead()	Boolean	It tests whether the file is readable or not
canWrite()	Boolean	It tests whether the file is writable or not
createNewFile()	Boolean	This method creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	It tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory



Byte Stream Classes

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Java provides two kinds of byte stream classes: **input stream classes** and **output stream classes**.

Input Stream Classes

Input stream classes that are used to read bytes include a super class known as **Inputstream** and a number of subclasses for supporting various input-related functions. The super class InputStream is an **abstract** class, and, therefore, we cannot create instances of this class. Rather, we must use the subclasses that inherit from this class.

Output Stream Classes

Output stream classes are derived from the base class **Outputstream** like InputStream, the OutputStream is an **abstract** class and therefore we cannot instantiate it. The

several subclasses of the OutputStream can be used for performing the output operations.

Character Stream Classes

Character streams can be used to read and write 16-bit Unicode characters. Like byte streams, there are two kinds of character stream classes, namely, **reader stream classes** and **writer stream classes**.

Reader Stream Classes

Reader stream classes that are used to read characters include a super class known as **Reader** and a number of subclasses for supporting various input-related functions. Reader stream classes are functionally very similar to the input stream classes, except input streams use bytes as their fundamental unit of information, while reader streams use characters. The Reader class contains methods that are identical to those available in the InputStream class, except Reader is designed to handle characters. Therefore, reader classes can perform all the functions implemented by the input stream classes.

Writer Stream Classes

Like output stream classes, the writer stream classes are designed to perform all output operations on files. Only difference is that while output stream classes are designed to write bytes, the writer stream are designed to write character. The **Writer** class is an **abstract** class which acts as a base class for all the other writer stream classes. This base class provides support for all output operations by defining methods that are identical to those in Outputstream class.

- **Serialization and Deserialization**

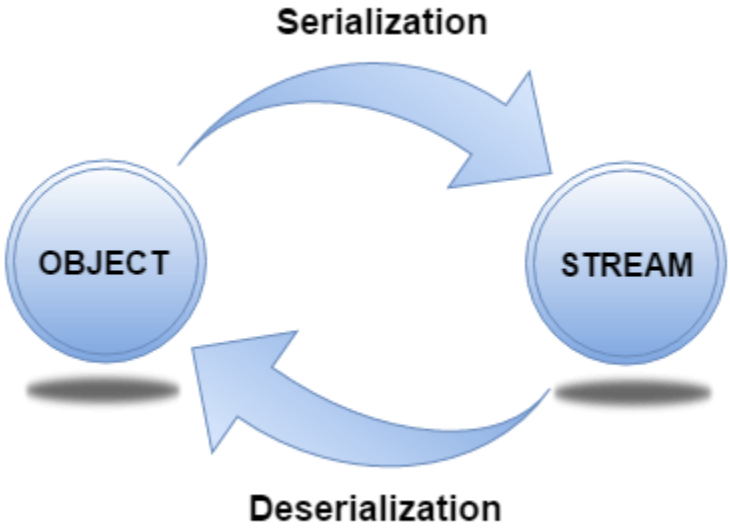
What is Serialization and deserialization?

Serialization is a mechanism to convert an object into stream of bytes so that it can be written into a file, transported through a network or stored into database. De-serialization is just a vice versa. In simple words serialization is converting an object to stream of bytes and de-serialization is rebuilding the object from stream of bytes. Java Serialiation API provides the features to perform seralization & de-serialization. A class must implement java.io.Serializable interface to be eligible for serialization.

For serializing the object, we call the **writeObject()** method *ObjectOutputStream*, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.\

Advantages of Java Serialization

It is mainly used to travel object's state on the network (which is known as marshaling).



ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

1) public ObjectOutputStream(OutputStream out) throws IOException {}	creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	flushes the current output stream.
3) public void close() throws IOException {}	closes the current output stream.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

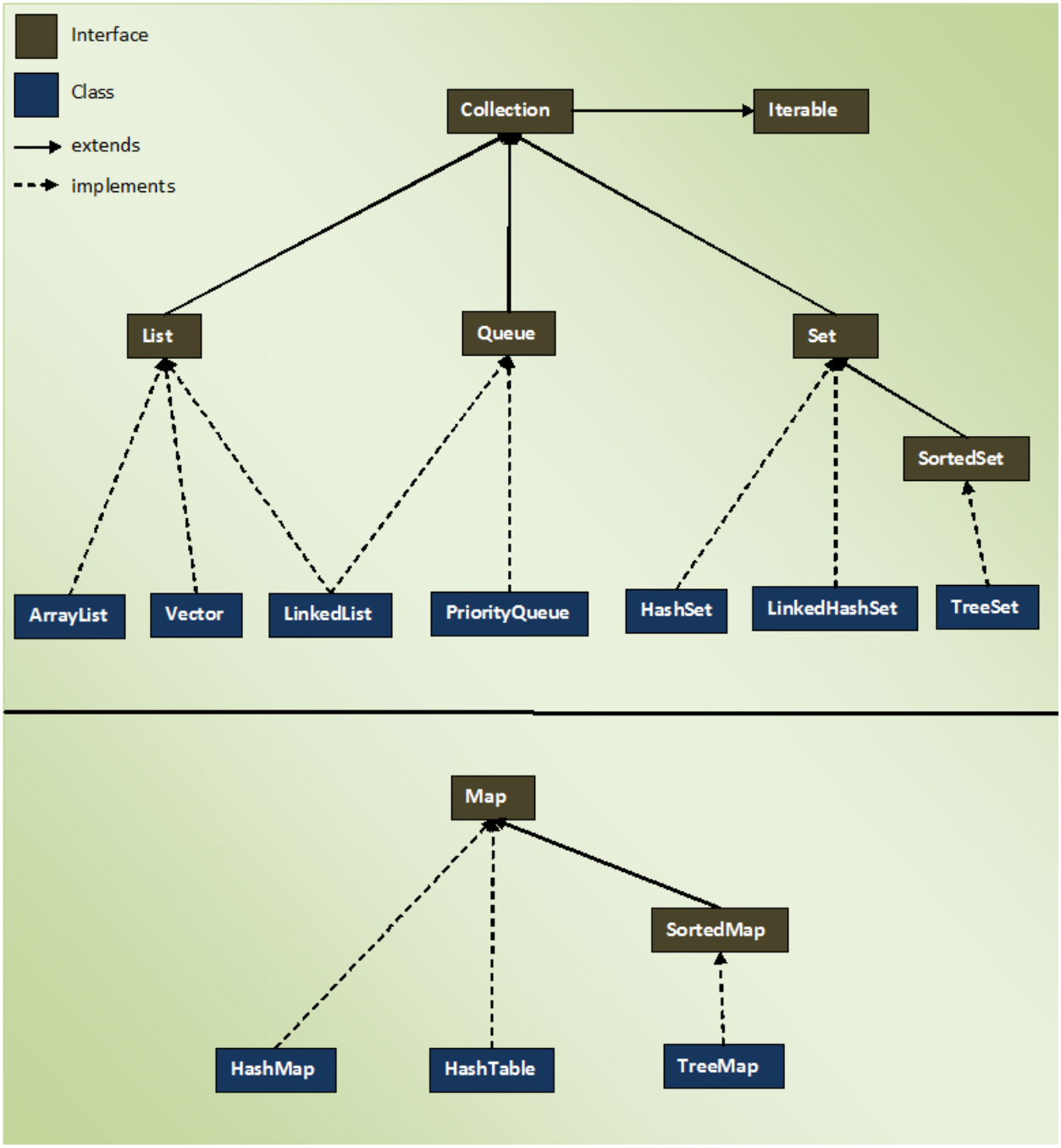
1) public ObjectInputStream(InputStream in) throws IOException {}	creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException{}	reads an object from the input stream.
2) public void close() throws IOException {}	closes ObjectInputStream.

Collection

The Java Collections Framework is a collection of interfaces and classes which helps in storing and processing the data efficiently. This framework has several useful classes which have tons of useful functions which makes a programmer task super easy. I have written several tutorials on Collections in Java.



List

Lists represents an **ordered** collection of elements. Using lists, we can access elements by their integer index (position in the list), and search for elements in the list. index start with 0, just like an array.

Some useful classes which implement List interface are
– **ArrayList**, **CopyOnWriteArrayList**, **LinkedList**, **Stack** and **Vector**.

ArrayList in Java

An **ArrayList in Java** represent a resizable list of objects. We can add, remove, find, sort and replace elements in this list. ArrayList is part of Java's collection framework and implements Java's **List** interface.

Constructors in Java ArrayList:

1. ArrayList(): This constructor is used to build an empty array list
2. ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from collection c
3. ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified

Hierarchy of ArrayList class

Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.

ArrayList Hierarchy

1. ArrayList Features

ArrayList has following features –

1. **Ordered** – Elements in arraylist preserve their ordering which is by default the order in which they were added to the list.
2. **Index based** – Elements can be randomly accessed using index positions. Index start with '0'.
3. **Dynamic resizing** – ArrayList grows dynamically when more elements needs to be added than it's current size.
4. **Non synchronized** – ArrayList is not synchronized, by default. Programmer needs to use synchronized keyword appropiately or simply use **Vector** class.
5. **Duplicates allowed** – We can add duplicate elements in arraylist. It is not possible in sets.

2. Internal Working of ArrayList

ArrayList class is implemented arounda backing array. The elements added or removed from arraylist are actually modified in this backing array. All arraylist methods access this array and get/set elements in the array.

ArrayList basically cab be seen as resizable-array implementation in Java.

ArrayList.java

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess,
                Cloneable, java.io.Serializable
{
    transient Object[] elementData;     //backing array
    private int size;                   //array or list size

    //more code
}
```

Methods of Java ArrayList

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It is used to enhance the capacity of an ArrayList instance.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.

Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.

void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Vector Class in Java

The Vector class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections.

- Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
- They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.
- It extends **AbstractList** and implements **List** interfaces.

Constructor:

- **Vector()**: Creates a default vector of initial capacity is 10.
- **Vector(int size):** Creates a vector whose initial capacity is specified by size.

- **Vector(int size, int incr):** Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time that a vector is resized upward.

- **Vector(Collection c):** Creates a vector that contains the elements of collection c.

Important points regarding Increment of vector capacity:
If increment is specified, Vector will expand according to it in each allocation cycle but if increment is not specified then vector's capacity get doubled in each allocation cycle.
Vector defines three protected data member:

ArrayList	Vector
ArrayList is not synchronized.	Vector is synchronized.
At a time multiple threads are allowed to operate on ArrayList object and hence ArrayList is not thread safe.	At a time only one threads are allowed to operate on vector object and hence vector is thread safe.
Threads are not required to wait to operate on ArrayList, hence relatively performance is high.	Threads are required to wait to operate on vector object, hence relatively performance is low.
ArrayList introduced in 1.2 version and hence it is not legacy class.	Vector introduced in 1.0 version and hence it is legacy class.

Java LinkedList class

Java LinkedList class is doubly-linked list implementation of the List and Deque interfaces. It implements all optional list operations, and permits all elements (including null).

The LinkedList class **extends AbstractSequentialList** class and **implements List and Deque** interfaces. Here 'E' is the type of values linkedlist stores.

1. LinkedList Features

- **Doubly linked list** implementation which implements List and Deque interfaces. Therefore, It can also be used as a Queue, Deque or Stack.
- Permits all elements including duplicates and NULL.
- LinkedList maintains the **insertion order** of the elements.
- It is **not synchronized**. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.
- Use Collections.synchronizedList(new LinkedList()) to get synchronized linkedlist.

- The iterators returned by this class are fail-fast and may throw ConcurrentModificationException.
- It does not implement RandomAccess interface. So we can access elements in sequential order only. It does not support accessing elements randomly.
- We can use ListIterator to iterate LinkedList elements.

2. LinkedList Constructors

1. **LinkedList()** : initializes an empty LinkedList implementation.
2. **LinkedListExample(Collection c)** : initializes a LinkedList containing the elements of the specified collection, in the order they are returned by the collection's iterator.

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Queue

A queue data structure is intended to hold the elements (put by producer threads) prior to processing by consumer thread(s). Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. One such exception is priority queue which order elements according to a supplied Comparator, or the elements' natural ordering.

In general, queues do not support blocking insertion or retrieval operations. Blocking queue implementations classes implement **BlockingQueue** interface.

Some useful classes which implement Map interface are – ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue and SynchronousQueue.

Queue Interface In Java

The Queue interface is available in java.util package and extends the Collection interface. The queue collection is used to hold the elements about to be processed and provides various operations like the insertion, removal etc. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list i.e. it follows the FIFO or the First-In-First-Out principle. Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java.It is to be noted that both the implementations are not thread safe. *PriorityBlockingQueue* is one alternative implementation if thread safe implementation is needed. Few important characteristics of Queue are:

- The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.
- The Java Queue supports all methods of Collection interface including insertion, deletion etc.
- LinkedList, ArrayBlockingQueue and PriorityQueue are the most frequently used implementations.
- If any null operation is performed on BlockingQueues, NullPointerException is thrown.

- BlockingQueues have thread-safe implementations.
- The Queues which are available in java.util package are Unbounded Queues
- The Queues which are available in java.util.concurrent package are the Bounded Queues.
- All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively. The Deques support element insertion and removal at both ends.

Methods in Queue:

1. **add()-** This method is used to add elements at the tail of queue. More specifically, at the last of linked-list if it is used, or according to the priority in case of priority queue implementation.
2. **peek()-** This method is used to view the head of queue without removing it. It returns Null if the queue is empty.
3. **element()-** This method is similar to peek(). It throws *NoSuchElementException* when the queue is empty.

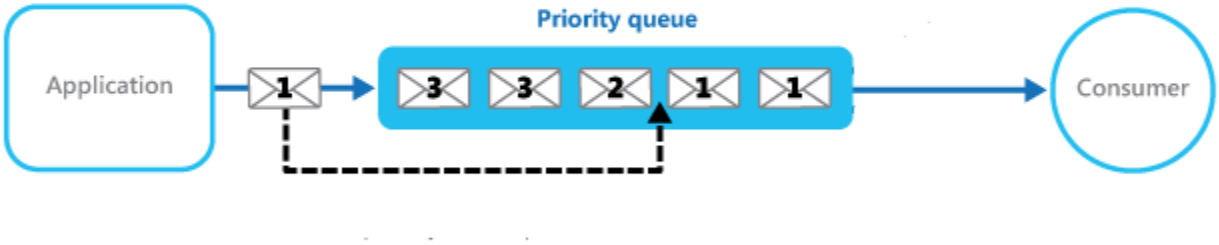
4. **remove()-** This method removes and returns the head of the queue. It throws *NoSuchElementException* when the queue is empty.
5. **poll()-** This method removes and returns the head of the queue. It returns null if the queue is empty.
6. **size()-** This method return the no. of elements in the queue.

OPERATION	THROWS EXCEPTION	RETURN VALLUES
Insert	add(element)	offer(element)
Remove	remove()	poll()
Examine	element()	peek()

Java PriorityQueue class

Java PriorityQueue class is a queue data structure implementation in which objects are processed based on their **priority**. It is different from standard queues where FIFO (First-In-First-Out) algorithm is followed.

In a priority queue, added objects are according to their priority. By default, the priority is determined by objects' natural ordering. Default priority can be overridden by a Comparator provided at queue construction time.



Priority Queue

1. PriorityQueue Features

Let's note down few important points on the PriorityQueue.

- PriorityQueue is an unbounded queue and grows dynamically. The default initial capacity is '11' which can be overridden using **initialCapacity** parameter in appropriate constructor.

- It does not allow NULL objects.
- Objects added to PriorityQueue MUST be comparable.
- The objects of the priority queue are ordered **by default in natural order**.
- A Comparator can be used for custom ordering of objects in the queue.
- The **head** of the priority queue is the **least** element based on the natural ordering or comparator based ordering. When we poll the queue, it returns the head object from the queue.
- If multiple objects are present of same priority the it can poll any one of them randomly.
- PriorityQueue is **not thread safe**. Use `PriorityBlockingQueue` in concurrent environment.
- It provides **O(log(n))** time for add and poll methods.

2. Java PriorityQueue Constructors

PriorityQueue class provides 6 different ways to construct a priority queue in Java.

- **PriorityQueue()** : constructs empty queue with the default initial capacity (11) that orders its elements according to their natural ordering.
- **PriorityQueue(Collection c)** : constructs empty queue containing the elements in the specified collection.
- **PriorityQueue(int initialCapacity)** : constructs empty queue with the specified initial capacity that orders its elements according to their natural ordering.
- **PriorityQueue(int initialCapacity, Comparator comparator)** : constructs empty queue with the specified initial capacity that orders its elements according to the specified comparator.
- **PriorityQueue(PriorityQueue c)** : constructs empty queue containing the elements in the specified priority queue.
- **PriorityQueue(SortedSet c)** : constructs empty queue containing the elements in the specified sorted set.

SET

Sets represents a collection of **sorted** elements. Sets do not allow the duplicate elements. Set interface does not provides no guarantee to return the elements in any predictable order; though some Set implementations store elements in their natural ordering and guarantee this order.

Some useful classes which implement Set interface are
– **ConcurrentSkipListSet**, **CopyOnWriteArraySet**, **EnumSet**, **HashSet**, **LinkedHash Set** and **TreeSet**.

The Java Collections Framework provides three major implementations of the Set interface: HashSet, LinkedHashSet and TreeSet. The Set API is described in the following diagram:

Let's look at the characteristics of each implementation in details:

- **HashSet**: is the best-performing implementation and is a widely-used Set implementation. It represents the core characteristics of sets: no duplication and unordered.
- **LinkedHashSet**: This implementation orders its elements based on insertion order. So consider using a LinkedHashSet when you want to store unique elements in order.
- **TreeSet**: This implementation orders its elements based on their values, either by their natural ordering, or by a Comparator provided at creation time.

Therefore, besides the uniqueness of elements that a Set guarantees, consider using HashSet when ordering does not matter; using LinkedHashSet when you want to order elements by their insertion order; using TreeSet when you want to order elements by their values.

HashSet Features

- It implements Set Interface.
- Duplicate values are not allowed in HashSet.
- One NULL element is allowed in HashSet.
- It is un-ordered collection and makes no guarantees as to the iteration order of the set.
- This class offers constant time performance for the basic operations(add, remove, contains and size).

- HashSet is not synchronized. If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.
- Use **Collections.synchronizedSet(new HashSet())** method to get the synchronized hashset.
- The iterators returned by this class's iterator method are **fail-fast** and may throw `ConcurrentModificationException` if the set is modified at any time after the iterator is created, in any way except through the iterator's own `remove()` method.
- HashSet also implements Searlizable and Cloneable interfaces.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

LinkedHashSet

Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operation and permits null elements.
- Java LinkedHashSet class is non synchronized.
- Java LinkedHashSet class maintains insertion order.

Given below are the list of constructors supported by the LinkedHashSet:

1. **LinkedHashSet():** This constructor is used to create a default HashSet.
2. **LinkedHashSet(Collection C):** Used in initializing the HashSet with the eleements of the collection C
3. **LinkedHashSet(int size):** Used to initialize the size of the LinkedHashSet with the integer mentioned in the parameter.
4. **LinkedHashSet(int capacity, float fillRatio):** Can be used to initialize both the capacity and the fill ratio, also called the load capacity of the LinkedHashSet with the arguments mentioned in the parameter. When the number of elements exceeds the capacity of the hash set is multiplied with the fill ratio thus expanding the capacity of the LinkedHashSet

1. Initial Capacity

The initial capacity means the number of buckets (in backing HashMap) when hashset is created. The number of buckets will be automatically increased if the current size gets full.

Default initial capacity is **16**. We can override this default capacity by passing default capacity in it's constructor **HashSet(int initialCapacity)**.

2. Load Factor

The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. Default load factor is **0.75**.

This is called **threshold** and is equal to (DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY). When HashSet elements count exceed this threshold, HashSet is resized and new capacity is double the previous capacity.

With default HashSet, the internal capacity is 16 and load factor is 0.75. The number of buckets will automatically get increased when the table has 12 elements in it.

3. HashSet Constructors

The HashSet has four types of constructors:

1. **HashSet():** initializes a default HashSet instance with the default initial capacity (16) and load factor (0.75).
2. **HashSet(int capacity):** initializes a HashSet with a specified capacity and load factor (0.75).

3. **HashSet(int capacity, float loadFactor):** initializes HashSet with specified initial capacity and load factor.
4. **HashSet(Collection c):** initializes a HashSet with same elements as the specified collection.

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

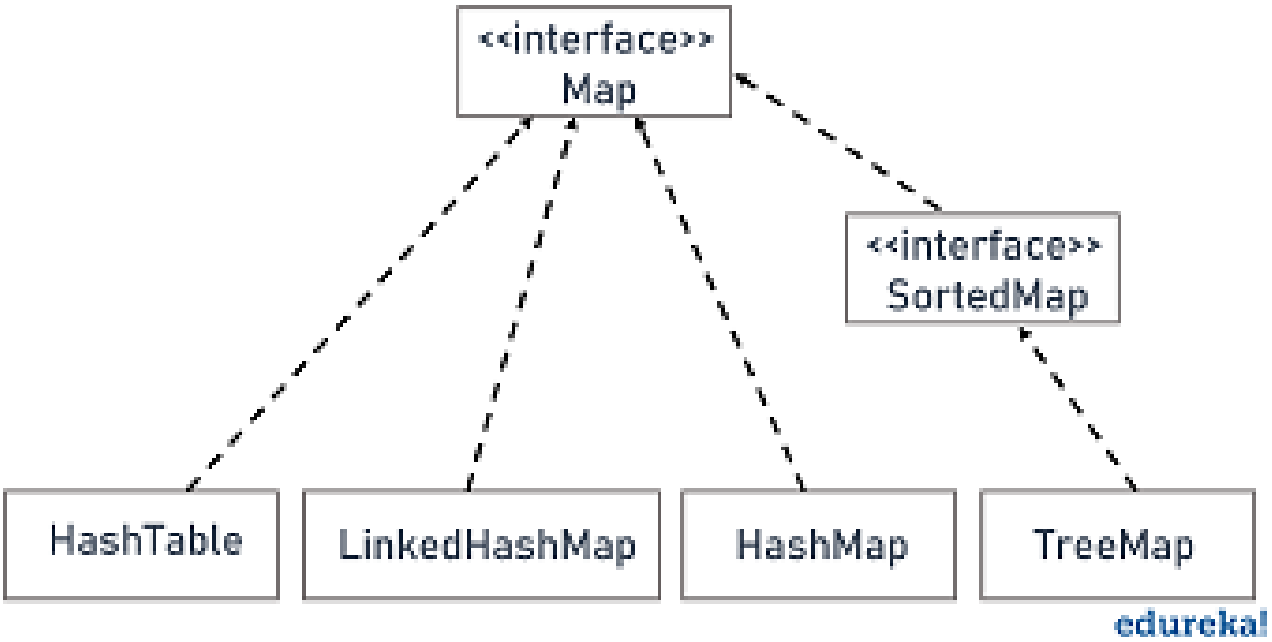
TreeSet Constructors

The TreeSet has four possible constructors:

1. **TreeSet():** creates a new, empty tree set, sorted according to the natural ordering of its elements.
2. **TreeSet(Comparator c):** creates a new, empty tree set, sorted according to the specified comparator.
3. **TreeSet(SortedSet s):** creates a new tree set containing the same elements and using the same ordering as the specified sorted set.
4. **TreeSet(Collection c):** creates a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.

Map Interface - Java Collections

A Map stores data in key and value association. Both key and values are objects. The key must be unique but the values can be duplicate. Although Maps are a part of Collection Framework, they can not actually be called as collections because of some properties that they posses. However we can obtain a **collection-view** of maps.



Useful methods of Map interface

Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.

Set<Map.Entry<K,V>> entrySet()	It returns the Set view containing all the keys and values.
void clear()	It is used to reset the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.

V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

Points to remember

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable Constructors

Hashtable class has four constructors.

- **Hashtable():** It is the default constructor. It constructs a new, empty hashtable with a default initial capacity (11) and load factor (0.75).
- **Hashtable(int size):** It constructs a new, empty hashtable of specified initial size.
- **Hashtable(int size, float fillRatio):** It constructs a new, empty hashtable of specified initial size and fill ratio.
- **Hashtable(Map m):** It constructs a hashtable that is initialized with the key-value pairs in specified map.

Example:

```
//1. Create Hashtable
    Hashtable<Integer, String> hashtable = new Hashtable<>();

    //2. Add mappings to hashtable
    hashtable.put(1,  "A");
    hashtable.put(2,  "B" );
    hashtable.put(3,  "C");

    System.out.println(hashtable);

    //3. Get a mapping by key
    String value = hashtable.get(1);        //A
```

Java LinkedHashMap class

Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

Points to remember

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

LinkedHashMap Constructors

The LinkedHashMap has five types of constructors:

1. **LinkedHashMap():** initializes a default LinkedHashMap implementation with the default initial capacity (16) and load factor (0.75).
2. **LinkedHashMap(int capacity):** initializes a LinkedHashMap with a specified capacity and load factor (0.75).
3. **LinkedHashMap(Map map):** initializes a LinkedHashMap with same mappings as the specified map.
4. **LinkedHashMap(int capacity, float fillRatio):** initializes LinkedHashMap with specified initial capacity and load factor.
5. **LinkedHashMap(int capacity, float fillRatio, boolean Order):** initializes both the capacity and fill ratio for a LinkedHashMap along with whether to maintain the insertion order or access order.
    - `'true'` enable access order.
    - `'false'` enable insertion order. This is default value behavior when using other constructors.

HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

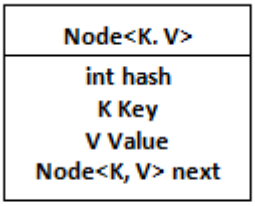


Figure: Representation of a Node

HashMap Features

- HashMap cannot contain duplicate keys.
- HashMap allows multiple `null` values but only one `null` key.
- HashMap is an **unordered collection**. It does not guarantee any specific order of the elements.
- HashMap is **not thread-safe**. You must explicitly synchronize concurrent modifications to the HashMap. Or you can use **Collections.synchronizedMap(hashMap)** to get the synchronized version of HashMap.
- A value can be retrieved only using the associated key.
- HashMap stores only object references. So primitives must be used with their corresponding wrapper classes. Such as `int`
- will be stored as `Integer`.
- HashMap implements **Cloneable** and **Serializable** interfaces.

Constructors in HashMap

HashMap provides 4 constructors and access modifier of each is public:

1. **HashMap():** It is the default constructor which creates an instance of HashMap with initial capacity 16 and load factor 0.75.
2. **HashMap(int initial capacity):** It creates a HashMap instance with specified initial capacity and load factor 0.75.
3. **HashMap(int initial capacity, float loadFactor):** It creates a HashMap instance with specified initial capacity and specified load factor.
4. **HashMap(Map map):** It creates instance of HashMap with same mappings as specified map.

No.	HashMap	Hashtable
1)	HashMap is not synchronized.	Hashtable is synchronized.

2)	HashMap can contain one null key and multiple null values.	Hashtable cannot contain any null key or null value.
3)	HashMap is not ?thread-safe,? so it is useful for non-threaded applications.	Hashtable is thread-safe, and it can be shared between various threads.
4)	4) HashMap inherits the AbstractMap class	Hashtable inherits the Dictionary class.

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

TreeMap Constructors

The TreeMap has four types of constructors:

1. **TreeMap():** creates a new, empty tree map, using the natural ordering of its keys.
2. **TreeMap(Comparator c):** creates a new, empty tree map, ordered according to the given comparator.
3. **TreeMap(Map map):** creates a new tree map containing the same mappings as the given map, ordered according to the natural ordering of its keys.
4. **TreeMap(SortedMap map):** creates a new tree map containing the same mappings and .

Iterable	Iterator
Iterable is an interface	Iterator is an interface
Belongs to java.lang package	Belongs to java.util package
Provides one single abstract method called iterator()	Provides two abstract methods called hasNext() and next()
It is a representation of a series of elements that can be traversed	It represents the object with iteration state

ArrayList	LinkedList
Implements dynamic array internally to store elements	Implements doubly linked list internally to store elements
Manipulation of elements is slower	Manipulation of elements is faster
Can act only as a List	Can act as a List and a Queue
Effective for data storage and access	Effective for data manipulation

Comparable	Comparator
Present in java.lang package	Present in java.util package
Elements are sorted based on natural ordering	Elements are sorted based on user-customized ordering
Provides a single method called compareTo()	Provides to methods equals() and compare()
Modifies the actual class	Doesn’t modifies the actual class

List	Set
An ordered collection of elements	An unordered collection of elements
Preserves the insertion order	Doesn’t preserves the insertion order
Duplicate values are allowed	Duplicate values are not allowed
Any number of null values can be stored	Only one null values can be stored
ListIterator can be used to traverse the List in any direction	ListIterator cannot be used to traverse a Set
Contains a legacy class called vector	Doesn’t contains any legacy class

Set	Map
Belongs to java.util package	Belongs to java.util package
Extends the Collection interface	Doesn’t extend the Collection interface
Duplicate values are not allowed	Duplicate keys are not allowed but duplicate values are
Only one null values can be stored	Only one null key can be stored but multiple null values are allowed
Doesn’t maintain any insertion order	Doesn’t maintain any insertion order

List	Map
Belongs to java.util package	Belongs to java.util package
Extends the Collection interface	Doesn’t extend the Collection interface
Duplicate elements are allowed	Duplicate keys are not allowed but duplicate values are
Multiple null values can be stored	Only one null key can be stored but multiple null values are allowed
Preserves the insertion order	Doesn’t maintain any insertion order
Stores elements based on Array Data Structure	Stores data in key-value pairs using various hashing techniques

PriorityQueue	TreeSet
It is a type of Queue	It is based on a Set data structure
Allows duplicate elements	Doesn’t allows duplicate elements
Stores the elements based on an additional factor called priority	Stores the elements in a sorted order

Singly Linked List(SLL)	Doubly Linked List(DLL)
Contains nodes with a data field and a next node-link field	Contains nodes with a data field, a previous link field, and a next link field
Can be traversed using the next node-link field only	Can be traversed using the previous node-link or the next node-link
Occupies less memory space	Occupies more memory space
Less efficient in providing access to the elements	More efficient in providing access to the elements

44. Differentiate between HashSet and HashMap.

HashSet	HasMap
Based on Set implementation	Based on Map implementation
Doesn’t allow any duplicate elements	Doesn’t allow any duplicate keys but duplicate values are allowed
Allows only a single null value	Allows only one null key but any number of null values
Has slower processing time	Has faster processing time
Uses HashMap as an underlying data structure	Uses various hashing techniques for data manipulation

Iterator	ListIterator
Can only perform remove operations on the Collection elements	Can perform add, remove and replace operations the Collection elements
Can traverse List, Sets and maps	Can traverse only Lists
Can traverse the Collection in forward direction	Can traverse the collection in any direction
Provides no method to retrieve the index of the element	Provides methods to retrieve the index of the elements
iterator() method is available for the entire Collection Framework	listIterator() is only available for the collections implementing the List interface

46. Differentiate between HashSet and TreeSet.

HashSet	TreeSet
Uses HasMap to store elements	Uses Treemap to store elements
It is unordered in nature	By default, it stores elements in their natural ordering
Has faster processing time	Has slower processing time
Uses hasCode() and equals() for comparing	Uses compare() and compareTo() for comparing
Allows only one null element	Doesn’t allow any null element
Takes up less memory space	Takes up more memory space

47. Differentiate between Queue and Deque.

Queue	Deque
Refers to single-ended queue	Refers to double-ended queue
Elements can be added or removed from only one end	Elements can be added and removed from either end
Less versatile	More versatile

HashMap	TreeMap
Doesn’t preserves any ordering	Preserves the natural ordering
Implicitly implements the hashing principle	Implicitly implements the Red-Black Tree Implementation
Can store only one null key	Cannot store any null key
More memory usage	Less memory usage
Not synchronized	Not synchronized

ArrayList	Vector
Non-synchronized in nature	Synchronized in nature
It is not a legacy class	Is a legacy class
Increases size by 1/2 of the ArrayList	Increases size by double of the ArrayList
It is not thread-safe	It is thread-safe

Java 8 features

1. Lambda Expression

Lambda expressions are not unknown to many of us who have worked on other popular programming languages like Scala. In Java programming language, a Lambda expression (or function) is just an *anonymous function*, i.e., a function with no name and without being bounded to an identifier. They are written exactly in the place where it's needed, typically *as a parameter to some other function*.

The basic *syntax of a lambda expression* is:

either

(parameters) -> expression

or

(parameters) -> { statements; }

or

() -> expression

A typical lambda expression example will be like this:

(x, y) -> x + y //This function takes two parameters and return their sum.

Please note that based on type of x and y, method may be used in multiple places. Parameters can match to int, or Integer or simply String also. Based on context, it will either add two integers or concat two strings.

Rules for writing lambda expressions

1. A lambda expression can have zero, one or more parameters.
2. The type of the parameters can be explicitly declared or it can be inferred from the context.
3. Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.
4. When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a*a.
5. The body of the lambda expressions can contain zero, one or more statements.
6. If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of

the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

2. Functional Interface

Functional interfaces are also called *Single Abstract Method interfaces (SAM Interfaces)*. As name suggest, they **permit exactly one abstract method** inside them. Java 8 introduces an annotation i.e. `@FunctionalInterface` which can be used for compiler level errors when the interface you have annotated violates the contracts of Functional Interface.

A typical functional interface example:

```
@FunctionalInterface
public interface MyFirstFunctionalInterface {
    public void firstWork();
}
```

Please note that a functional interface is valid even if the `@FunctionalInterface` annotation would be omitted. It is only for informing the compiler to enforce single abstract method inside interface.

Also, since default methods are not abstract you're *free to add default methods* to your functional interface as many as you like.

Another important point to remember is that if an interface declares an abstract method overriding one of the public methods of `java.lang.Object`, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from `java.lang.Object` or elsewhere. for example, below is perfectly valid functional interface.

```
@FunctionalInterface
public interface MyFirstFunctionalInterface
{
    public void firstWork();

    @Override
    public String toString();              //Overridden from Object class
```

```
    @Override
    public boolean equals(Object obj);        //Overridden from Object class
}
```

3. Default Methods

Java 8 allows you to add non-abstract methods in interfaces. These methods must be declared default methods. Default methods were introduces in java 8 to enable the functionality of lambda expression.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Let’s understand with an example:

```
public interface Moveable {
    default void move(){
        System.out.println("I am moving");
    }
}
```

`Moveable` interface defines a method `move()` and provided a default implementation as well. If any class implements this interface then it need not to implement it’s own version of `move()` method. It can directly call `instance.move()`. e.g.

```
public class Animal implements Moveable{
    public static void main(String[] args){
        Animal tiger = new Animal();
        tiger.move();
    }
}
```

Output: I am moving

If class willingly wants to customize the behavior of `move()` method then it can provide it’s own custom implementation and override the method.

4. Java 8 Streams

Another major change introduced **Java 8 Streams API**, which provides a mechanism for processing a set of data in various ways that can include filtering, transformation, or any other way that may be useful to an application.

Streams API in Java 8 supports a different type of iteration where you simply define the set of items to be processed, the operation(s) to be performed on each item, and where the output of those operations is to be stored.

An example of stream API. In this example, `items` is collection of `String` values and you want to remove the entries that begin with some prefix text.

```
List<String> items;

String prefix;

List<String> filteredList = items.stream().filter(e -> (!e.startsWith(prefix))).collect(Collectors.toList());
```

Here `items.stream()` indicates that we wish to have the data in the `items` collection processed using the Streams API.

5. **Java Method References**

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Four types of method references

1. Method reference to an instance method of an object – object::instanceMethod
2. Method reference to a static method of a class – Class::staticMethod
3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod
4. Method reference to a constructor – Class::new

1. Method reference to an instance method of an object

```
@FunctionalInterface
interface MyInterface{
    void display();
}
public class Example {
    public void myMethod(){
```

```
        System.out.println("Instance Method");
    }
    public static void main(String[] args) {
        Example obj = new Example();
        // Method reference using the object of the class
        MyInterface ref = obj::myMethod;
        // Calling the method of functional interface
        ref.display();
    }
}
```

Output:

```
Instance Method
```

2. Method reference to a static method of a class

```
import java.util.function.BiFunction;
class Multiplication{
    public static int multiply(int a, int b){
        return a*b;
    }
}
public class Example {
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> product = Multiplication::multiply;
        int pr = product.apply(11, 5);
        System.out.println("Product of given number is: "+pr);
    }
}
```

Output:

```
Product of given number is: 55
```

3. Method reference to an instance method of an arbitrary object of a particular type

```
import java.util.Arrays;
public class Example {

    public static void main(String[] args) {
        String[] stringArray = { "Steve", "Rick", "Aditya", "Negan", "Lucy", "Sansa",
"Jon"};
        /* Method reference to an instance method of an arbitrary
         * object of a particular type
         */
        Arrays.sort(stringArray, String::compareToIgnoreCase);
        for(String str: stringArray){
            System.out.println(str);
        }
```

```
    }
}
```
Output:

```
Aditya
Jon
Lucy
Negan
Rick
Sansa
Steve
```
4. Method reference to a constructor

```
@FunctionalInterface
interface MyInterface{
    Hello display(String say);
}
class Hello{
    public Hello(String say){
        System.out.print(say);
    }
}
public class Example {
    public static void main(String[] args) {
            //Method reference to a constructor
        MyInterface ref = Hello::new;
        ref.display("Hello World!");
    }
}
```
Output:

```
Hello World!
```