Object Oriented Methodology

OOPs (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

Object

Class

Inheritance

Polymorphism

Abstraction

Encapsulation

Object

Java Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism in Java

Polymorphism

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

OOP's top benefits:

Modularity for easier troubleshooting. When working with object-oriented programming languages, you know exactly where to look when something goes wrong. ...

Reuse of code through inheritance. ...

Flexibility through polymorphism. ...

Effective problem solving.

Applications of Object-Oriented Programming

Client-Server Systems. ...

Object-Oriented Databases. ...

Object-Oriented Databases. ...

Real-Time System Design. ...

Simulation and Modeling System. ...

Hypertext and Hypermedia. ...

Neural Networking and Parallel Programming. ...

Office Automation Systems.

What is Java

Java is an Object-Oriented Programming (OOP) structure. Java is a class-based programming language. Java technology is used for developing both, applets and applications. Provides an easy to use: Avoids many of the pitfalls of other languages.

concept and syntax of class in java

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake. A Class is like an object constructor, or a "blueprint" for creating objects.

What is the syntax of a class?

A syntax class may have formal parameters, in which case they are bound as variables in the body. Syntax classes support optional arguments and keyword arguments using the same syntax as lambda. The body of the syntax-class definition contains a non-empty sequence of pattern variants.

string literal in java

A string literal in Java is basically a sequence of characters from the source character set used by Java programmers to populate string objects or to display text to a user. These characters could be anything like letters, numbers or symbols which are enclosed within two quotation marks

Arrays in Java

An array in Java is a group of like-typed variables referred to by a common name.

In Java, all arrays are dynamically allocated. (discussed below)

Arrays are stored in contagious memory [consecutive memory locations].

Since arrays are objects in Java, we can find their length using the object property

A Java array variable can also be declared like other variables with [] after the data type.

The variables in the array are ordered, and each has an index beginning from 0.

Java array can also be used as a static field, a local variable, or a method parameter.

The size of an array must be specified by int or short value and not long.

The direct superclass of an array type is Object.

Every array type implements the interfaces Cloneable and java.io.Serializable.

This storage of arrays helps us in randomly accessing the elements of an array [Support Random Access].

The size of the array cannot be altered(once initialized). However, an array reference can be made to point to another array.

Non primitive Datatypes

Non-primitive types are created by the programmer and is not defined by Java (except for String Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot. A primitive type has always a value, while non-primitive types can be null

There are five types of Non-Primitive data types in Java : Class, Object, String, Array, and Interface. They are used to store multiple values of either the same data type or different data types.

Execution model of java

Java's program execution model is divided into two distinct stages – Compilation, Bytecode Execution. These two stages are not directly related to each other. In fact, most of the times the second step occurs on a different machine and usually long after the first step.

JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What is JVM

A specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.

An implementation Its implementation is known as JRE (Java Runtime Environment).

Runtime Instance Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

concept and syntax of Methods in java

A method is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also known as functions.

The syntax to declare a method is: returnType methodName() { // method body } Here, returnType - It specifies what type of value a method returns For example if a method has an int return type then it returns an integer value. If the method does not return a value, its return type is void .

Widening and narrowing conversion in java

Widening conversions preserve the source value but can change its representation. This occurs if you convert from an integral type to Decimal , or from Char to String . A narrowing conversion changes a value to a data type that might not be able to hold some of the possible values.1

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

A first program to java

Java "Hello, World!" Program

// Your First Program

Objects and Classes in Java

we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

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.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
class <class_name>{
    field;
    method;
}
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Test it Now

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
```

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
```

```
    s2.insertRecord(222,"Aryan");
    s1.displayInformation();
    s2.displayInformation();
  }
}
```

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
}
}
```
Test it Now

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
new Calculation();//anonymous object
```

Calling method through a reference:

```java
Calculation c=new Calculation();
c.fact(5);
```

Calling method through an anonymous object

```java
new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```java
class Calculation{
 void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){
 new Calculation().fact(5);//calling method with anonymous object
}
}
```

Output:

```
Factorial is 120
```

Constructors in Java

constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.
In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

How Constructors are Different From Methods in Java?

- onstructors must have the same name as the class within which it is defined while it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Now let us come up with the syntax for the constructor being invoked at the time of object or instance creation.

```
class Geek

{

  .......


  // A Constructor

  new Geek() {

  }


  .......

}



// We can create an object of the above class

// using the below statement. This statement

// calls above constructor.

Geek obj = new Geek();
```

The first line of a constructor is a call to super() or this(), (a call to a constructor of a super-class or an overloaded constructor), if you don’t type in the call to super in your constructor the compiler will provide you with a non-argument call to super at the first line of your code, the super constructor must be called to create an object:

```
port java.io.*;

class Geeks {
    Geeks() { super(); }
    public static void main(String[] args)
    {
        Geeks geek = new Geeks();
    }
}
```

Need of Constructor

constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

When is a Constructor called?
Each time an object is created using a **new()** keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the **data members** of the same class.

The rules for writing constructors are as follows:

- Constructor(s) of a class must have the same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static, or Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of Constructors in Java

Two types of constructors in java:

- No-argument constructor
- Parameterized Constructor

1. **No-argument constructor:** A constructor that has no parameter is known as the default constructor. If we don't define a constructor in a class, then the compiler creates a **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no arguments then the compiler does not create a default constructor.
2. 2. Parameterized Constructor: A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Example:

```java
// Java Program to Illustrate Working of
// Parameterized Constructor

// Importing required inputoutput class
import java.io.*;

// Class 1
class Geek {
    // data members of the class.
    String name;
    int id;

    // Constructor would initialize data members
    // With the values of passed arguments while
    // Object of that class created
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}

// Class 2
class GFG {
    // main driver method
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        Geek geek1 = new Geek("adam", 1);
        System.out.println("GeekName :" + geek1.name
                           + " and GeekId :" + geek1.id);
    }
}
```

Access specifies in java

Access specifiers define the visibility of the class. If no keyword is mentioned then that is default access modifier. Four modifiers in Java include public, private, protected and default.

There are 4 types of access variables in Java:

- Private.

- Public.
- Default.
- Protected.

Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifier in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

Types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Example:

```
class A{
private int data=40;
private void msg (){System.out.println("Hello java");}
}
```

```
public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Access Control

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods. For example, **class Animal {public void method1() {...} private void method2() {...} }**

Private: declarations are visible within the class

Default: declarations are visible only within the

Modifier: Description

Public: declarations are visible everywhere

USING JAVA OBJECTS

String builder and String buffer Definition in Java:

String builder in Java: Constructors, Methods, and Examples

Table of Contents:

I. Class Declaration of String Builder
II. Looking at the Constructors of String Builder in Java
III. Discussing Various Methods of String Builder in Java
IV. Using the Methods of String Builder in Java

String Builder in Java is a **class** used to create a mutable, or in other words, a modifiable succession of characters. Like String Buffer, the String Builder class is an alternative to the **Java Strings Class**, as the Strings class provides an immutable succession of characters. However, there is one significant **difference between String Buffer and String Builder**, and it is that the latter is non-synchronized. It means that

String Builder in Java is a more suited choice while working with a single thread, as it will be quicker than String Buffer.

I. Class Declaration of String Builder

The java.lang.String Builder class is a part of java.lang package and has the following class declaration:

Public final class String Builder

extends Object

implements Serializable, CharSequence

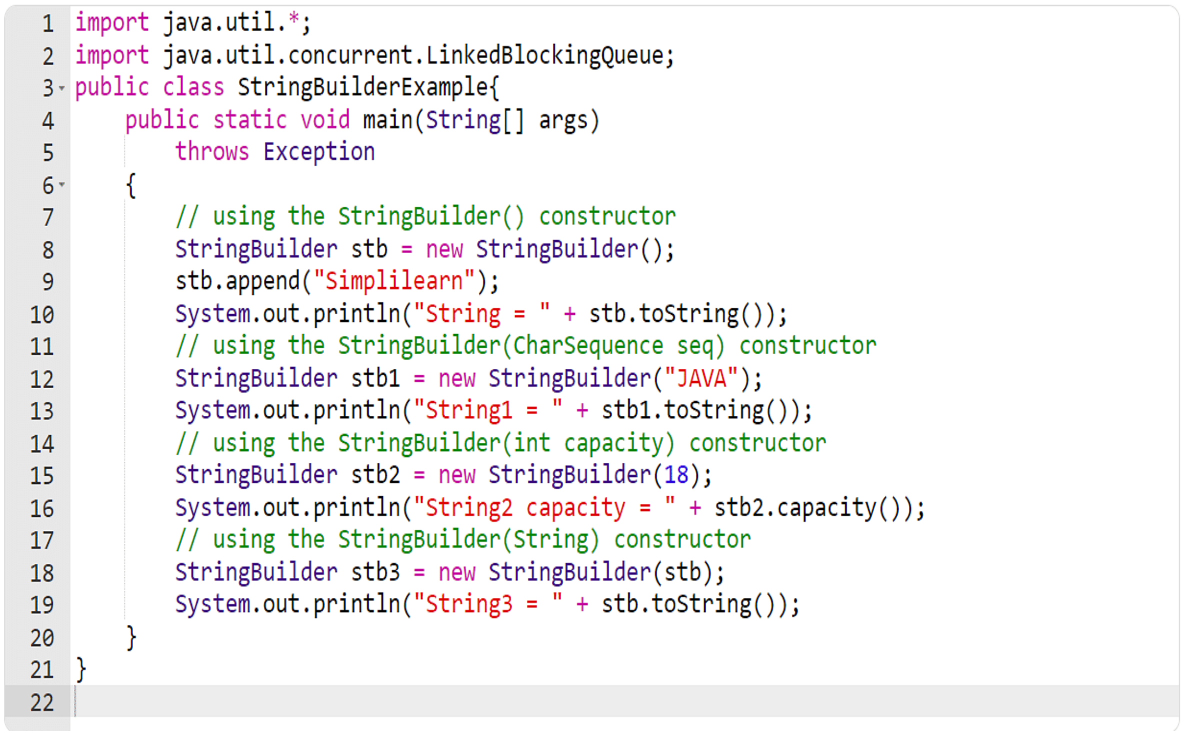
II. Looking at the Constructors of String Builder in Java

Constructor Name	Description
String Builder()	It constructs a blank string builder with a capacity of 16 characters
String Builder(int capacity)	It creates an empty string builder with the specified capacity
String Builder(CharSequence seq)	It creates a string builder with the same characters specified as the argument

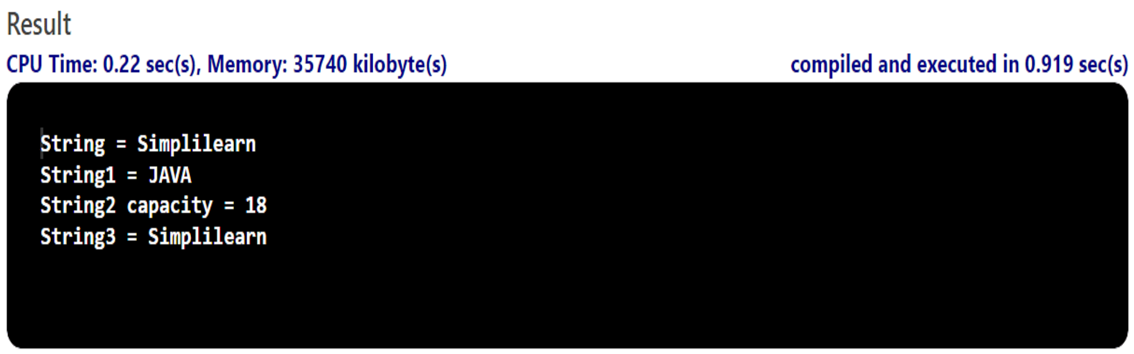
String Builder(String str)	It will construct a string builder with the string specified in the argument

The following table lists and describes the constructors of String Builder in Java

Since you now know about the **constructors** and class declaration of String Builder in Java, it's time to look at an example where you will use some of these constructors to create various sequences of characters.



```
import java.util.*;
import java.util.concurrent.LinkedBlockingQueue;
public class StringBuilderExample{
    public static void main(String[] args)
        throws Exception
    {
        // using the StringBuilder() constructor
        StringBuilder stb = new StringBuilder();
        stb.append("Simplilearn");
        System.out.println("String = " + stb.toString());
        // using the StringBuilder(CharSequence seq) constructor
        StringBuilder stb1 = new StringBuilder("JAVA");
        System.out.println("String1 = " + stb1.toString());
        // using the StringBuilder(int capacity) constructor
        StringBuilder stb2 = new StringBuilder(18);
        System.out.println("String2 capacity = " + stb2.capacity());
        // using the StringBuilder(String) constructor
        StringBuilder stb3 = new StringBuilder(stb);
        System.out.println("String3 = " + stb.toString());
    }
}

```



Result

CPU Time: 0.22 sec(s), Memory: 35740 kilobyte(s)   compiled and executed in 0.919 sec(s)

```
String = Simplilearn
String1 = JAVA
String2 capacity = 18
String3 = Simplilearn
```

III. Discussing Various Methods of String Builder in Java

The String Builder in Java provides numerous methods to perform different operations on the string builder. The table depicted below enlists some primary methods from the String Builder class.

Method	Description
String Builder append (String s)	This method appends the mentioned string with the existing string. You can also with arguments like Boolean, char, int, double, float, etc.
String Builder insert (int offset, String s)	It will insert the mentioned string to the other string from the specified offset position. Like append, you can overload this method with arguments like (int, boolean), (int, int), (int, char), (int, double), (int, float), etc.
String Builder replace(int start, int end, String s)	It will replace the original string with the specified string from the start index till the end index.
String Builder delete(int start, int end)	This method will delete the string from the mentioned start index till the end index.

String Builder reverse()	It will reverse the string.
int capacity()	This will show the current String Builder capacity.
void ensure Capacity(int min)	This method ensures that the String Builder capacity is at least equal to the mentioned minimum.
char charAt(int index)	It will return the character at the specified index.
int length()	This method is used to return the length (total characters) of the string.
String substring(int start)	Starting from the specified index till the end, this method will return the substring.
String substring(int start, int end)	It will return the substring from the start index till the end index.

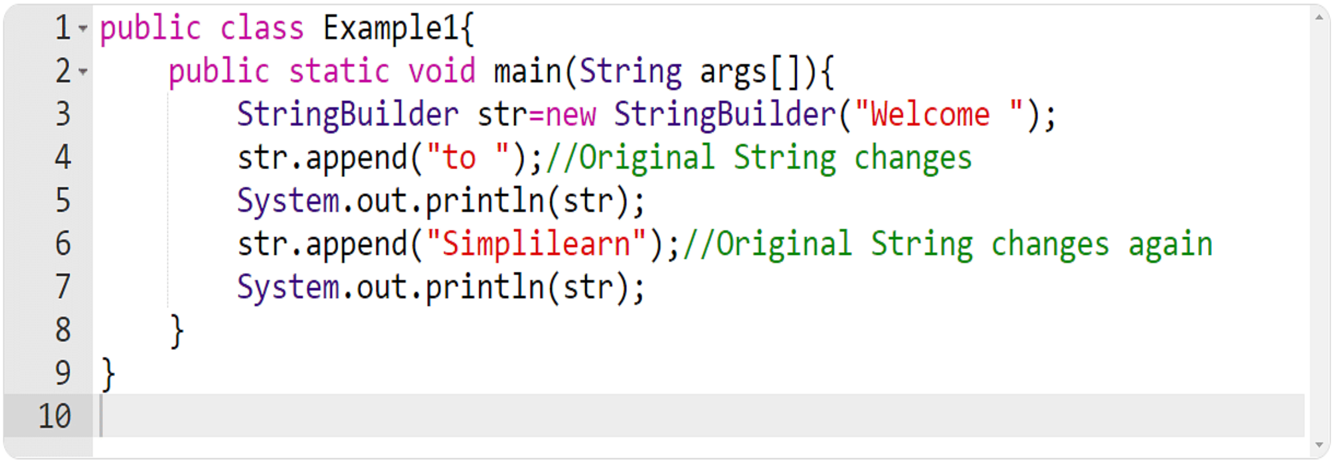
int indexOf(String str)	This method will return the index where the first instance of the specified string occurs.
int lastIndexOf(String str)	It will return the index where the specified string occurs the last.
Void trimToSize()	It will attempt to reduce the size of the String Builder.

IV. Using the Methods of String Builder in Java

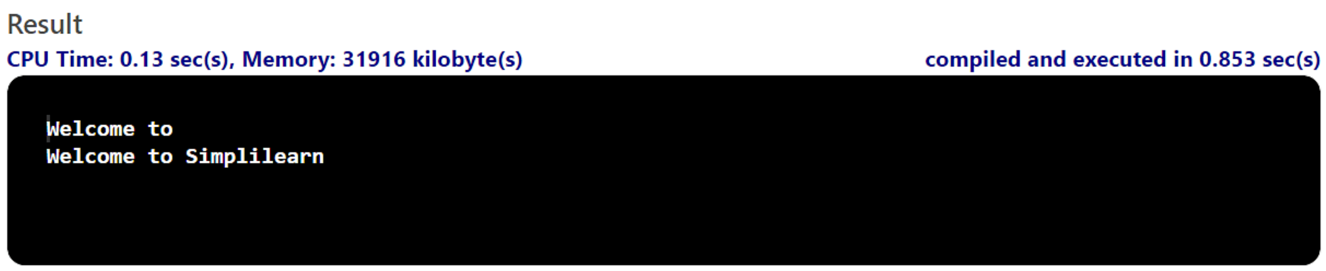
Let’s have a look at examples of some examples of the String Builder methods.

Example 1: Applying the Append () Method of String Builder in Java

Here, you must concatenate three strings using the append () method in the below example.



```
public class Example1{
    public static void main(String args[]){
        StringBuilder str=new StringBuilder("Welcome ");
        str.append("to ");//Original String changes
        System.out.println(str);
        str.append("Simplilearn");//Original String changes again
        System.out.println(str);
    }
}
```


Result

CPU Time: 0.13 sec(s), Memory: 31916 kilobyte(s)   compiled and executed in 0.853 sec(s)

```
Welcome to
Welcome to Simplilearn
```

Java String Buffer Class

Java String Buffer class is used to create mutable (modifiable) String objects. The String Buffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Note: Java String Buffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important Constructors of String Buffer Class

Constructor	Description
String Buffer()	It creates an empty String buffer with the initial capacity of 16.
String Buffer(String str)	It creates a String buffer with the specified string..
String Buffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

Important methods of String Buffer class

Modifier and Type	Method	Description

public synchronized String Buffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized String Buffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and String Builder classes are used for creating mutable strings.

1) String Buffer Class append () Method

The append () method concatenates the given argument with this String.

StringBufferExample.java

```
class String Buffer Example {
public static void main(String args[]){
StringBuffer sb=new StringBuffer ("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

Output:

```
Hello Java
```

METHODS AND MESSAGES IN JAVA

Method in Java:

Method in Java or Java Method is a collection of statements that perform some specific task and return the result to the caller. A Java method can perform some specific task without returning anything. Methods in Java allow us to reuse the code without retyping the code. In Java, every method must be part of some class that is different from languages like C, C++, and Python.

1. A method is like function i.e. used to expose behaviour of an object.
2. it is a set of codes that perform a particular task.

Syntax: Declare a method
<access_modifier> <return_type> <method_name>(list_of_parameters)

{

```
    //body
}
```

Advantage of Method

- Code Reusability
- Code Optimization

Note: Methods are time savers and help us to reuse the code without retyping the code.

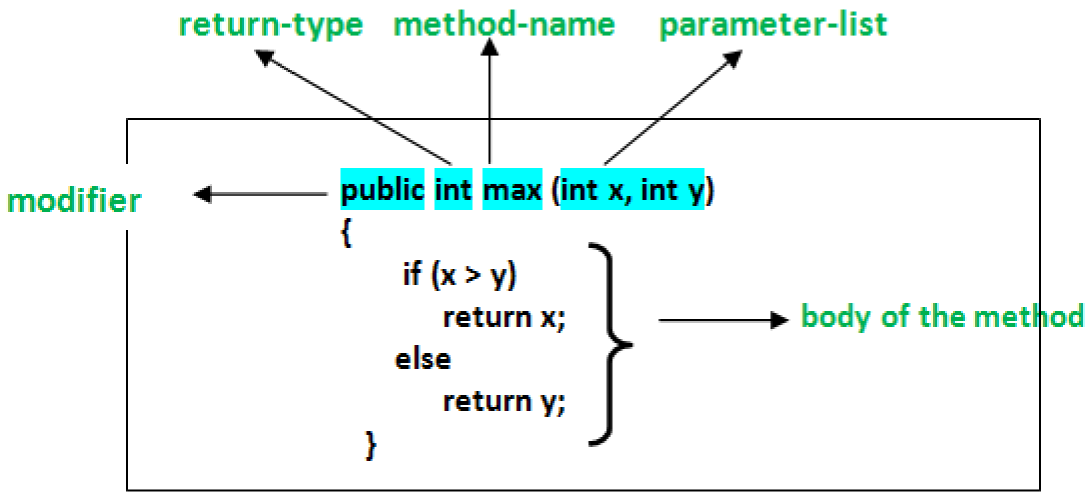
Method Declaration

In general, method declarations have six components:

1. Modifier: It defines the access type of the method i.e. from where it can be accessed in your application. In Java, there 4 types of access specifiers.

- Public: It is accessible in all classes in your application.
- protected: It is accessible within the class in which it is defined and in its subclass/es
- private: It is accessible only within the class in which it is defined.
- default: It is declared/defined without using any modifier. It is accessible within the same class and package within which its class is defined.

2. The return type: The data type of the value returned by the method or void if does not return a value.
3. Method Name: the rules for field names apply to method names as well, but the convention is a little different.
4. Parameter list: Comma-separated list of the input parameters is defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
5. Exception list: The exceptions you expect by the method can throw, you can specify these exception(s).
6. Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations.



Types of Methods in Java

There are two types of methods in Java:

1. Predefined Method: In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point.
2. User-defined Method: The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

Method Signature

It consists of the method name and a parameter list (number of parameters, type of the parameters, and order of the parameters). The return type and exceptions are not considered as part of it.

Method Signature of the above function:

max(int x, int y) Number of parameters is 2, Type of parameter is int.

How to Name a Method?

A method name is typically a single word that should be a verb in lowercase or multi-word, that begins with a verb in lowercase followed by an adjective, noun….. After the first word, the first letter of each word should be capitalized.
Rules to Name a Method

- While defining a method, remember that the method name must be a verb and start with a lowercase letter.
- If the method name has more than two words, the first name must be a verb followed by an adjective or noun.
- In the multi-word method name, the first letter of each word must be in uppercase except the first word. For example, findSum, computeMax, setX and getX.

Generally, a method has a unique name within the class in which it is defined but sometimes a method might have the same name as other method names within the same class as method overloading is allowed in Java.

Method Calling

The method needs to be called for using its functionality. There can be three situations when a method is called:
A method returns to the code that invoked it when:

- It completes all the statements in the method
- It reaches a return statement
- Throws an exception

Example:

- Java

```
// Java Program to Illustrate Methods

// Importing required classes

Import java.io.*;

// Class 1

// Helper class

class Addition {



   // Initially taking sum as 0
```

```
    // as we have not started computation

    int sum = 0;



    // Method

    // To add two numbers

    public int addTwoInt(int a, int b) {

// Adding two integer value

        sum = a + b;

 // Returning summation of two values

        return sum;

    }

}

 // Class 2

// Helper class

class GFG {

 // Main driver method

 public static void main(String[] args)

    {

   // Creating object of class 1 inside main() method

Addition add = new Addition();
```

```
        // Calling method of above class

        // to add two integer

        // using instance created

        int s = add.addTwoInt(1, 2);



        // Printing the sum of two numbers

        System.out.println("Sum of two integer values :"  + s);

    }

}
```

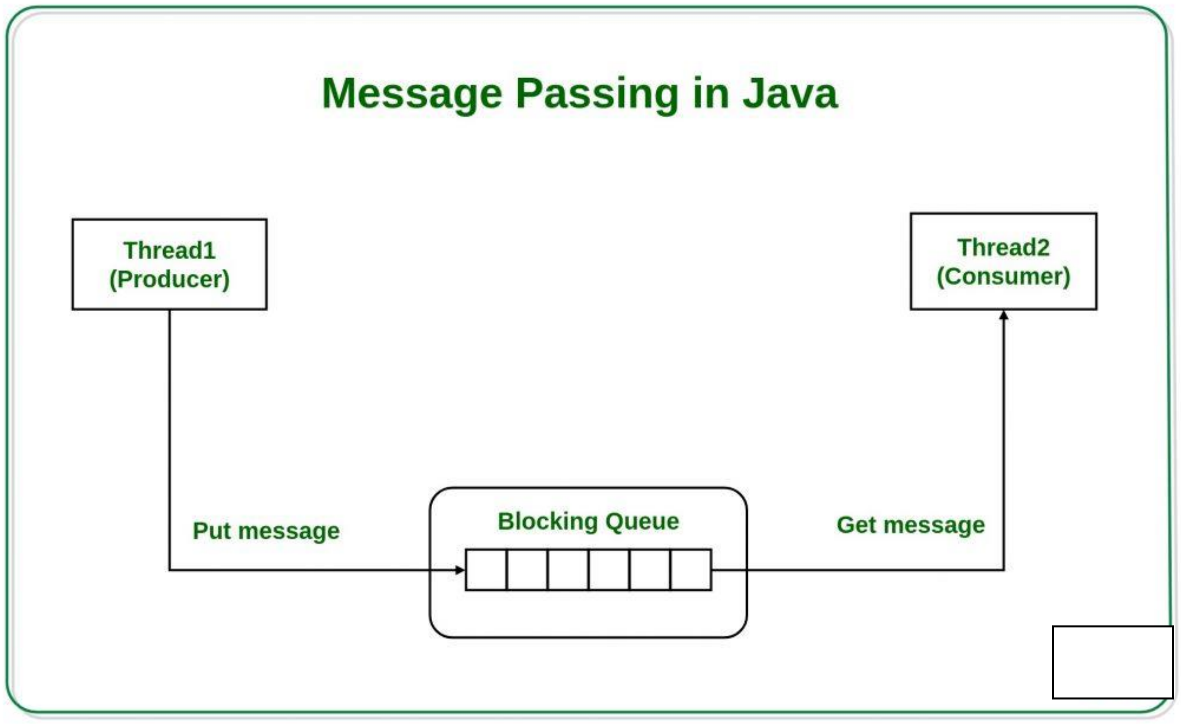
Output

```
Sum of two integer values: 3
```

Message Passing in Java

What is message passing and why it is used?

Message Passing in terms of computers is communication between processes. It is a form of communication used in object-oriented programming as well as parallel programming. Message passing in Java is like sending an object i.e. message from one thread to another thread. It is used when threads do not have shared memory and are unable to share monitors or semaphores or any other shared variables to communicate. Suppose we consider an example of producer and consumer, likewise what producer will produce, the consumer will be able to consume that only. We mostly use **Queue** to implement communication between threads.



In the example explained below, we will be using vector (queue) to store the messages, 7 at a time and after that producer will wait for the consumer until the queue is empty.

In Producer there are two synchronized methods **putMessage()** which will call form **run()** method of Producer and add message in Vector whereas **getMessage()** extracts the message from the queue for the consumer. Using message passing simplifies **the producer-consumer problem** as they don't have to reference each other directly but only communicate via a queue.

Example:

```
import java.util.Vector;

class Producer extends Thread {

  // initialization of queue size

   static final int MAX = 7;

   private Vector messages = new Vector();
```

```
@Override
public void run()
    {
        try {
            while (true) {

                // producing a message to send to the consumer
                putMessage();

                // producer goes to sleep when the queue is full
                sleep(1000);
            }
        }
        catch (InterruptedException e) {
        }
    }

    private synchronized void putMessage()
        throws InterruptedException
    {
```

```
    // checks whether the queue is full or not
    while (messages.size() == MAX)

        // waits for the queue to get empty
        wait();

    // then again adds element or messages
    messages.addElement(new java.util.Date().toString());
    notify();
}

public synchronized String getMessage()
    throws InterruptedException
{
    notify();
    while (messages.size() == 0)
        wait();
    String message = (String)messages.firstElement();
```

```
        // extracts the message from the queue

        messages.removeElement(message);

        return message;

    }

}


class Consumer extends Thread {

    Producer producer;



    Consumer(Producer p)

    {

        producer = p;

    }



    @Override

    public void run()

    {

        try {

            while (true) {

                String message = producer.getMessage();
                // sends a reply to producer got a message

                System.out.println("Got message: " + message);

                sleep(2000);

            }

        }

        catch (InterruptedException e) {

        }

    }



    public static void main(String args[])

    {

        Producer producer = new Producer();

        producer.start();

        new Consumer(producer).start();

    }

}
```

Output:

```
Got message: Thu May 09 06:57:53 UTC 2019

Got message: Thu May 09 06:57:54 UTC 2019

Got message: Thu May 09 06:57:55 UTC 2019

Got message: Thu May 09 06:57:56 UTC 2019
Got message: Thu May 09 06:57:57 UTC 2019

Got message: Thu May 09 06:57:58 UTC 2019

Got message: Thu May 09 06:57:59 UTC 2019

Got message: Thu May 09 06:58:00 UTC 2019
```

Passing and Returning Objects in Java

Although Java is strictly passed **by value**, the precise effect differs between whethe a **primitive type**r or a reference type is passed. When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference.
Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

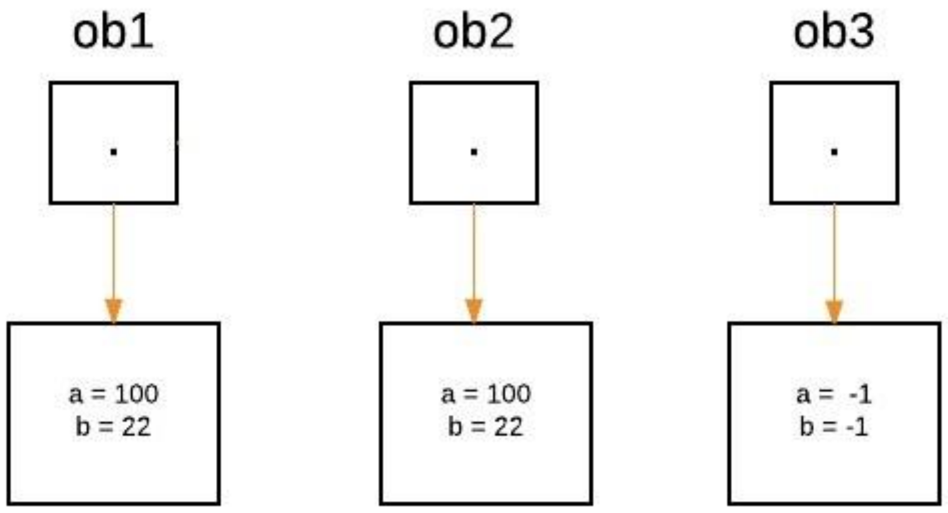
- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect the object used as an argument.

Illustration: Let us suppose three objects 'ob1' , 'ob2' and 'ob3' are

created:
ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);

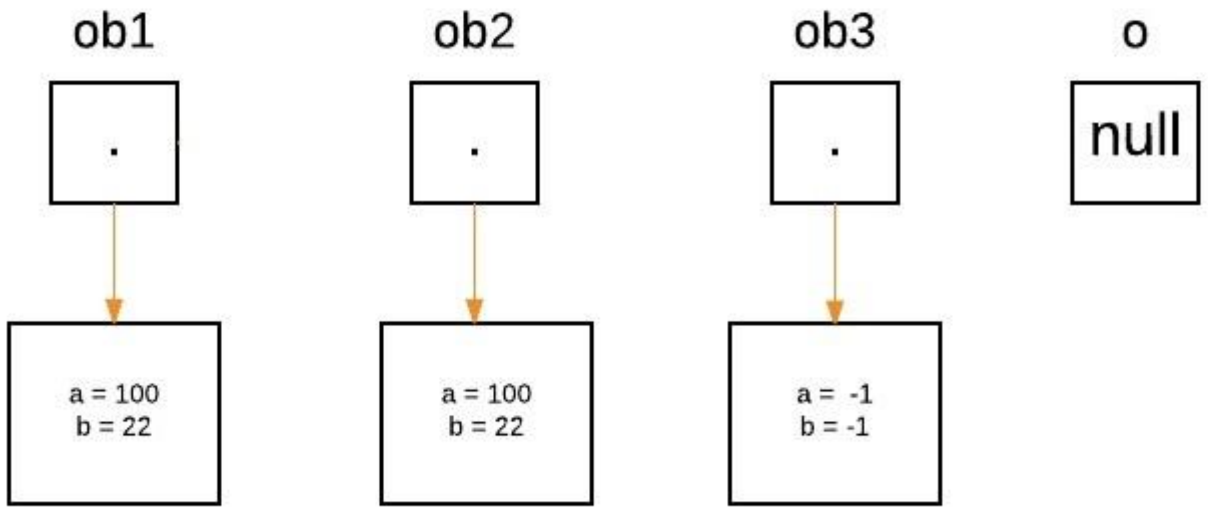
ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);

ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);



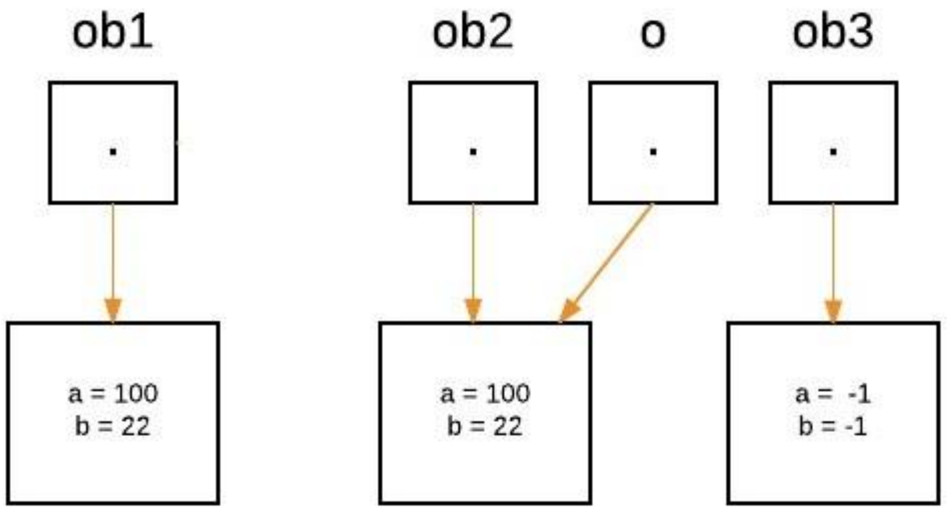
From the method side, a reference of type Foo with a name a is declared and it’s initially assigned to null.

boolean equalTo(ObjectPassDemo o);



As we call the method equalTo, the reference ‘o’ will be assigned to the object which is passed as an argument, i.e. ‘o’ will refer to ‘ob2’ as the following statement execute.

System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));

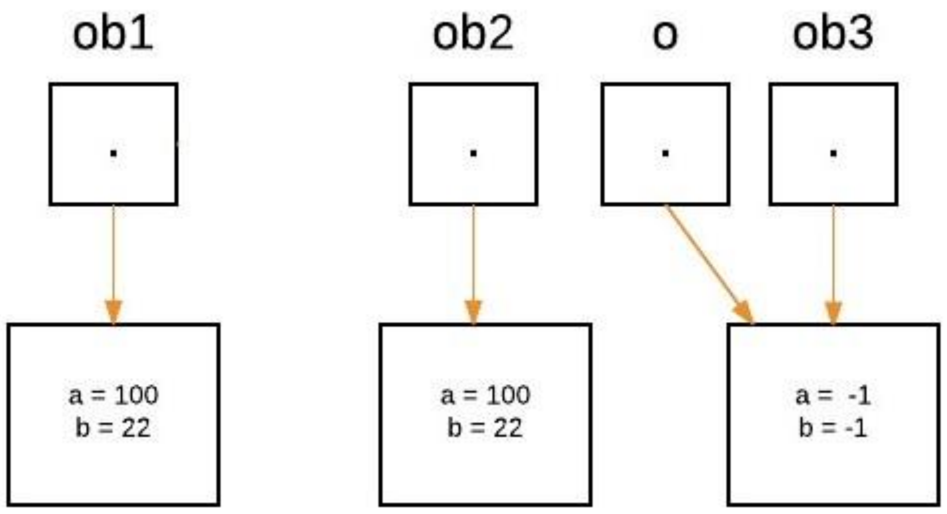


Now as we can see, equalTo method is called on ‘ob1’ , and ‘o’ is referring to ‘ob2’. Since values of ‘a’ and ‘b’ are same for both the references, so if(condition) is true, so boolean true will be return.

if(o.a == a && o.b == b)

Again ‘o’ will reassign to ‘ob3’ as the following statement execute.

System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));



- Now as we can see, the equalTo method is called on ‘ob1’ , and ‘o’ is referring to ‘ob3’. Since values of ‘a’ and ‘b’ are not the same for both the references, so if(condition) is false, so else block will execute, and false will be returned.

In Java we can pass objects to methods as one can perceive from the below program as follows:

Example:

- Java

```
// Java Program to Demonstrate Objects Passing to Methods.

 // Class

// Helper class

class ObjectPassDemo {

    int a, b;

  // Constructor

    ObjectPassDemo(int i, int j)

    {

       a = i;

        b = j;

    }



    // Method

    boolean equalTo(ObjectPassDemo o)

    {

        // Returns true if o is equal to the invoking

        // object notice an object is passed as an

        // argument to method
        return (o.a == a && o.b == b);

    }

}



// Main class

public class GFG {

    // MAin driver method

    public static void main(String args[])

    {

        // Creating object of above class inside main()

        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);

        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);

        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);



        // Checking whether object are equal as custom

        // values

        // above passed and printing corresponding boolean

        // value

        System.out.println("ob1 == ob2: "

                           + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: "
                           + ob1.equalTo(ob3));
    }
}
```

Output

```
ob1 == ob2: true
ob1 == ob3: false
```

Defining a constructor that takes an object of its class as a parameter

One of the most common uses of object parameters involves constructors. Frequently, in practice, there is a need to construct a new object so that it is initially the same as some existing object. To do this, either we can use **Object.clone()** method or define a constructor that takes an object of its class as a parameter.
Example

- Java

```
// Java program to Demonstrate One Object to
// Initialize Another


// Class 1
class Box {
    double width, height, depth;
```

```
// Notice this constructor. It takes an

// object of type Box. This constructor use

// one object to initialize another

Box(Box ob)

{

    width = ob.width;

    height = ob.height;

    depth = ob.depth;

}


// constructor used when all dimensions

// specified

Box(double w, double h, double d)

{

    width = w;

    height = h;

    depth = d;

}


// compute and return volume
```

```
    double volume() { return width * height * depth; }

}



// MAin class

public class GFG {

    // Main driver method

    public static void main(String args[])

    {

        // Creating a box with all dimensions specified

        Box mybox = new Box(10, 20, 15);



        //  Creating a copy of mybox

        Box myclone = new Box(mybox);



        double vol;



        // Get volume of mybox

        vol = mybox.volume();

        System.out.println("Volume of mybox is " + vol);
        // Get volume of myclone

        vol = myclone.volume();

        System.out.println("Volume of myclone is " + vol);

    }

}
```

Output

```
Volume of mybox is 3000.0

Volume of myclone is 3000.0
```

Returning Objects

In java, a method can return any type of data, including objects. For example, in the following program, the incrByTen( ) method returns an object in which the value of an (an integer variable) is ten greater than it is in the invoking object.

Example

- Java

```
// Java Program to Demonstrate Returning of Objects



// Class 1

class ObjectReturnDemo {

    int a;
    // Constructor

    ObjectReturnDemo(int i) { a = i; }



    // Method returns an object

    ObjectReturnDemo incrByTen()

    {

        ObjectReturnDemo temp

            = new ObjectReturnDemo(a + 10);

        return temp;

    }

}



// Class 2

// Main class

public class GFG {



    // Main driver method

    public static void main(String args[])

    {
        // Creating object of class1 inside main() method

        ObjectReturnDemo ob1 = new ObjectReturnDemo(2);

        ObjectReturnDemo ob2;



        ob2 = ob1.incrByTen();



        System.out.println("ob1.a: " + ob1.a);

        System.out.println("ob2.a: " + ob2.a);

    }

}
```

Output

```
ob1.a: 2

ob2.a: 12
```

Note: When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does. That's why we said that java is **strictly pass-by-value.**

This article is contributed by Gaurav Miglani. If you like GeeksforGeeks and would like to contribute, you can also write an article using **write.geeksforgeeks.org** or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

How to Compare Two Objects in Java

Java Object class is the super class of all the Java classes. All Java classes implements the Object class by default. The Java Object class provides the two important methods to compare two **objects in Java**,
i.e. equals() and hashCode() method. In this section, we will learn
how equals() and hashCode() method works. Along with this, we will also
learn how to compare two objects in Java with proper examples.

Java provides the two methods of the Object class to compare the objects are as follows:

- Java equals() Method
- Java hashCode() Method

Java equals () Method

The equals() method of the Object class compare the equality of two objects. The two objects will be equal if they share the same memory address.

Syntax:

```
1. public boolean equals(Object obj)
```

The method parses a reference object as a parameter. It returns true if the objects are equal, else returns false.

It is also possible that an object is equal to another given object, then the equals() method follow the equivalence relation to compare the objects.

- Reflexive: If x is a non-null reference, the calling of x.equals(x) must return true.
- Symmetric: If the two non-null references are x and y, x.equals(y) will return true if and only if y.equals(x) return true.
- Transitive: If the three non-null references are x, y, and z, x.equals(z) will also return true if x.equals(y) and y.equals(z) both returns true.

- Consistent: If the two non-null references are x and y, the multiple calling of x.equals(y) constantly returns either true or false. It does not provide any information used in the comparison.
- For any non-null reference x, x.equals(null) returns false.

In short, for any non-null reference say x and y, it returns true if and only if both references refer to the same object.

Remember: When we override the equals() method, it is necessary to override the hashCode() method. Overriding follow the convention for the hashCode() method that states, the equal object must have equal hash code.

Example of equals() method

In the following example, we have created constructor of the Double and Long class and passes corresponding values, as an argument that stored in their objects, respectively.

After that, in the first println statement, we have invoked equals() method and parse an object y as a parameter that compares the object x and y. It returns false because x holds the double value and y holds the long value that is not equal.

Similarly, in the second println statement, we have invoked equals() method and parse the same value as in the constructor of the Double class. It returns true because the object of double class i.e. x holds the same value as we have passed in the equals() method.

ObjectComparisonExample.java

```
public class ObjectComparisonExample
{
public static void main(String[] args)
{
//creating constructor of the Double class
Double x = new Double(123.45555);
//creating constructor of the Long class
Long y = new Long(9887544);
//invoking the equals() method
System.out.println("Objects are not equal, hence it returns " + x.equals(y));
System.out.println("Objects are equal, hence it returns " + x.equals(123.45555));
}
}
```

Output:

```
Objects are not equal, hence it returns false
Objects are equal, hence it returns true
```

Difference Between == Operator and equals() Method

In Java, the == operator compares that two references are identical or not. Whereas the equals() method compares two objects.

Objects are equal when they have the same state (usually comparing variables). Objects are identical when they share the class identity.

For example, the expression obj1==obj2 tests the identity, not equality. While the expression obj1.equals(obj2) compares equality.

Java hashCode() Method

In Java, hash code is a 32-bit signed integer value. It is a unique id provided by JVM to Java object. Each Java object is associated with the hash code. The hash code is managed by a hash-based data structure, such as HashTable, HashSet, etc.

Remember: When we override the equals() method, it is necessary to override the hashCode() method, also.

Syntax:

```
1. public int hashCode()
```

It returns a randomly generated hash code value of the object that is unique for each instance. The randomly generated value might change during the several executions of the program.

The general contract for hashCode is:

- When it is invoked more than once during the execution of an application, the hashCode() method will consistently return the same hash code (integer value). Note that the object should not be modified.
- If the two objects are equal according to the equals() method, then invoking the hashCode() method on these two objects must produce the same integer value.
- It is not necessary that if the two objects are unequal according to equals() method, then invoking the hashCode() method on these two objects may produce distinct integer value. It means that it can produce the same hash code for both objects.

Example of hashCode() method

In the following example, we have created two classes Employee.java and HashCodeExample.java.

In the Employee class, we have defined two fields regno of type int and name of type String. After that, we have created a constructor of the Employee class and passes these two fields as a parameter.

To perform the comparison of objects, we have created a separate class named HashCodeExample. In this class, we have created two instances of the Employee class i.e. emp1 and emp2. After that, we have invoked the hashCode() method using objects. We have stored the hash code value in the variable a and b, respectively.

Employee.java

```
public class Employee
{
private int regno;
private String name;
//constructor of Employee class
public Employee(int regno, String name)
{
this.name = name;
this.regno = regno;
}
public int getRegno()
{
return regno;
}
public void setRegno(int Regno)
{
this.regno = regno;
}
public String getName()
{
return name;
}
public void setName(String name)
{
this.name = name;
}
}
```

HashCodeExample:

```
public class HashcodeExample
{
public static void main(String[] args)
{
//creating two instances of the Employee class
Employee emp1 = new Employee(918, "Maria");
Employee emp2 = new Employee(918, "Maria");
//invoking hashCode() method
int a=emp1.hashCode();
int b=emp2.hashCode();
System.out.println("hashcode of emp1 = " + a);
System.out.println("hashcode of emp2 = " + b);
System.out.println("Comparing objects emp1 and emp2 = " + emp1.equals(emp2));
}
1.  }
```

Output:

```
hashcode of emp1 = 2398801145
hashcode of emp2 = 1852349007
Comparing objects emp1 and emp2 = false
```

Overriding equals() Method

We can override the equals() method in the following way if we want to provide own implementation.

```
//overriding equals() method
@Override
public boolean equals(Object obj)
{
if (obj == null)
return false;
if (obj == this)
return true;
return this.getRegno() == ((Employee) obj). getRegno();
}
```

The above code snippet shows that two employees will be equal if they are stored in the same memory address or they have the same regno. When we run the program (HashCodeExample.java) with the above code snippet, we get the following output.

Output:

```
hashcode of emp1 = 2032578917
hashcode of emp2 = 1531485190
Comparing objects emp1 and emp2 = true
```

INHERITANCE

Inheritance in Java

1. Inheritance
2. Types of Inheritance
3. Why multiple inheritance is not possible in Java in case of class?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

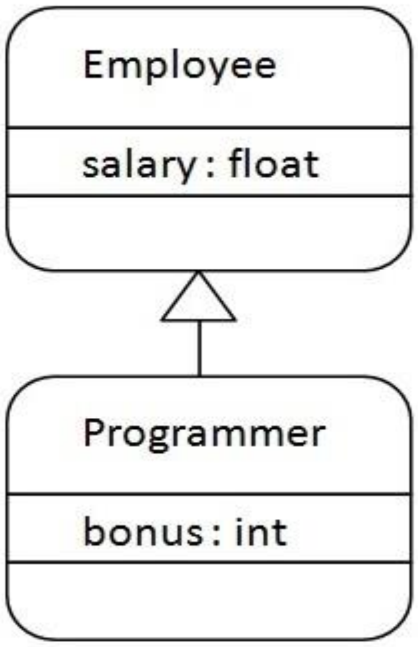
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Test it Now

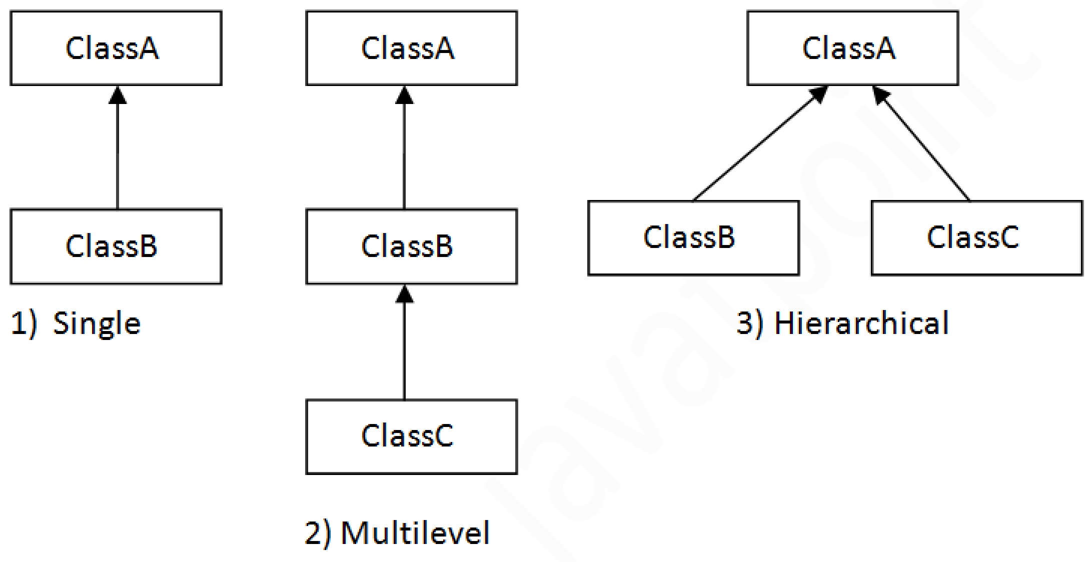
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

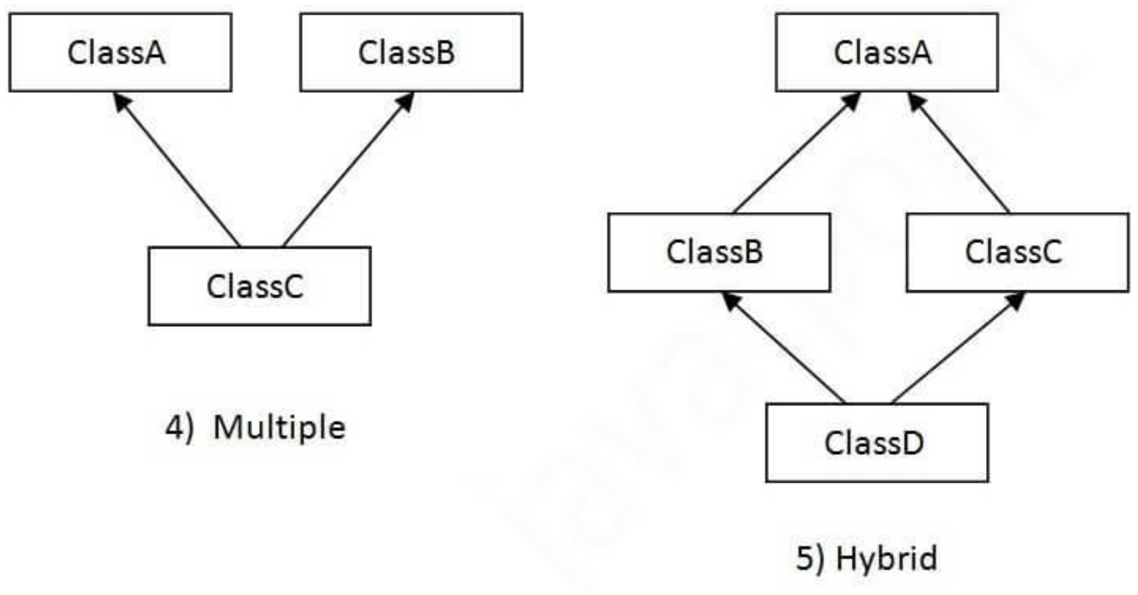
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



.

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}
}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example:

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
```

```
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}
}
```

Output:

```
meowing...
eating...
```

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
```

```
    obj.msg();//Now which msg() method would be invoked?
}
}
```

Polymorphism in Java

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples −

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal −

Example

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

Types of Polymorphism

Polymorphism in Java is divided into two types: compile time polymorphism and run time polymorphism. Static polymorphisms and dynamic polymorphisms are terms used to describe this type of java polymorphism.

1. Static polymorphism (or compile-time polymorphism)

Polymorphism in Java, like most other OOP programming languages, allows for the inclusion of several methods within a single class. Although the methods have the same name, the parameters differ. The static polymorphism is represented by this. This polymorphism is achieved through method overloading and is resolved during the compiler time. There are three conditions by which the parameter sets must differ:

The number of parameters should be varied.

Different parameter types should be used.

The parameters are in a different sequence. If one method accepts a string and a long while the other accepts a long and a string, for example. This form of order, on the other hand, makes it tough for the API to grasp.

Every method has a different signature due to the differences in parameters. The Java compiler knows which method is invoked.

Example of static polymorphism

One of the ways by which Java supports static polymorphism is method overloading. An example showing the case of method overloading in static polymorphism is shown below:

Example:

```
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(25, 25));

System.out.println(obj.add(25, 25, 30));

}

}
```

Output of the program

```
50

80
```

2. Dynamic Polymorphism (or run time polymorphism in Java)

The compiler does not determine the method to be executed in this type of polymorphism in Java. The process is carried out at runtime by the Java Virtual Machine (JVM). When a call to an overridden process is resolved at run time, it is referred to as dynamic polymorphism. The overridden method is called by a superclass's reference variable. While the methods implemented by both the subclass and the superclass have the same name, they provide separate functionality.

Before you can understand the concept of run time polymorphism, you must first understand the process of upcasting. Upcasting is the process of referring to a child class object with a reference variable from the superclass.

Example of Dynamic polymorphism (or run time)

Example1:

The classes Bike and Splendor are created, with the Splendor class extending Bike and overriding its run() method. The parent class's reference variable invokes the run() method. Because the subclass method is overriding the parent class method, it is called at run time.

The program:

```
class Bike{

void run(){System.out.println(“running”);}

}

class Splendor extends Bike{

void run(){System.out.println(“walking safely with 30km”);}
public static void main(String args[]){

Bike b = new Splendor();//upcasting

b.run();

}

}
```

Output: walking safely with 60km

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In Java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods

so that we don't need to create instance for calling methods.

```java
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type

. The first add method receives two integer arguments and second add method receives two double arguments.

```java
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
```

```
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

```
22
24.9
```

Runtime Polymorphism in Java

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output:

Running safely with 60km.

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
class Vehicle{
 void run(){System.out.println("Vehicle is running");}
}
```

```
class Bike2 extends Vehicle{

  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();
  obj.run();
  }
}
```

Output:

```
Bike is running safely
```

Packages: Putting Classes Together

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

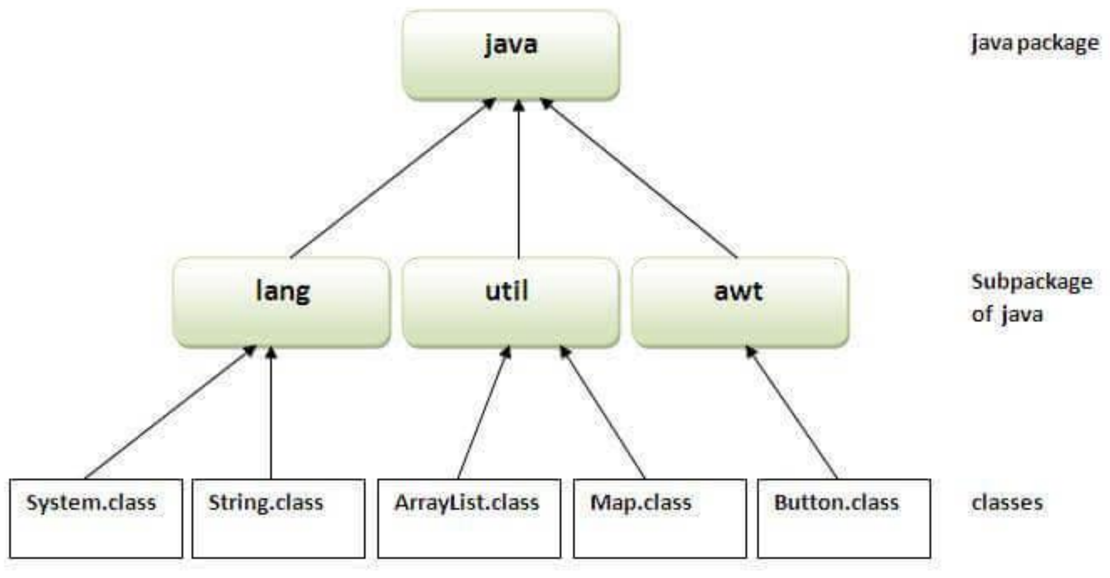
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
package mypack;
public class Simple{
 public static void main(String args[]){
 System.out.println("Welcome to package");
   }
}
```

API in Java

API in Java is delivered via Java Development Kit or JDK. JDK is made up of three entities.

1. **Java compiler:** A pre-quoted program used for breaking the complex user-written codes into simple and computer-understandable codes, known as byte-code.
2. **Java Virtual Machine (JVM):** Allotted to process the byte code and generate an easy-to-understand output.
3. **Java API:** The pre-integrated software components used for establishing a communication between desired software/platforms/components.

Java brings multiple pre-designed components into action for accomplishing the development process. Its API connects to included components and enable programmers to use their functionality so that they can make most of them.

Developers can refer to the classes and packages of available APIs and speed up the process of explaining the classes and packages for the planned program.

Why use the Java APIs?

The Java APIs can be a valuable tool for developers. They let you access a wide variety of third-party services with just a few lines of code, which can be helpful in solving problems or building apps that are difficult to do without them.

Even if you have no plans to build your own apps, knowing how these APIs work likely will help make your time as an app developer easier.

The Importance of Using APIs in Java

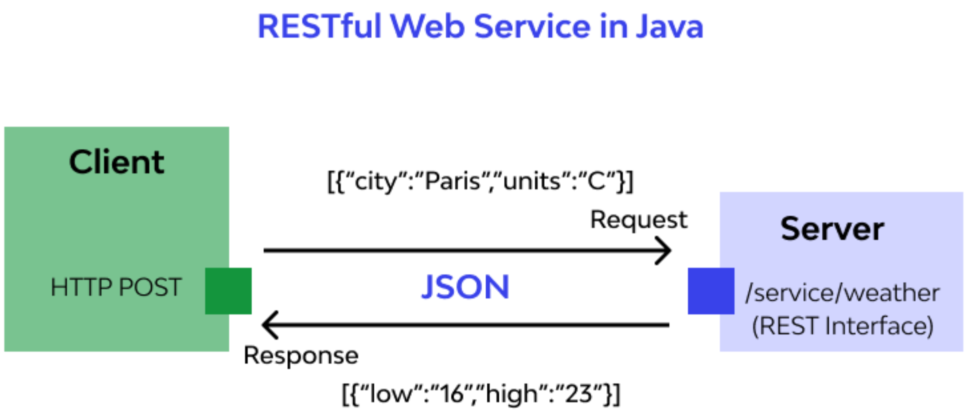
API is more than any other ordinary entity. It’s a way to make development smooth and enhance the development experience. Here is what significance API holds in Java.

- API in Java makes key operational techniques and processes streamlined than ever. The live example of this is The Develop Social Intelligent Inbox. By simply login on to this platform, using Facebook and Twitter, allows one to pay attention to the messages, revert to the tagged posts, and operate on the search items. As the platform is developed using Java API, all these details are presented in a unified view saving time and effort.
- Use of API in Java grants access to a wide range of SQL support.
- Java API brings customization of high quality accessible for developers.

The Java REST API work

REST API is one of the most widely used API in Java, along with Web API, Java Help, Facebook.4J, and Twitter.4J. It makes code available and usable for every related application/program. While one is using REST API in Java, the basic rules to be followed are:

- **Stateless:** REST follows client-server architecture to remain state-independent.
- **Uniform interface:** Applications using REST API in Java and beyond will be requiring the undeviating client and server interface via HTTP and URIs
- **Client-server:** Client and servers, involved in the communication, are independent of each other.
- **Cache:** Cache is an imperative part of REST API in Java as its presence makes recording intermediate responses easier than ever.
- **Layered:** REST API features layered structure and each layer is independent.

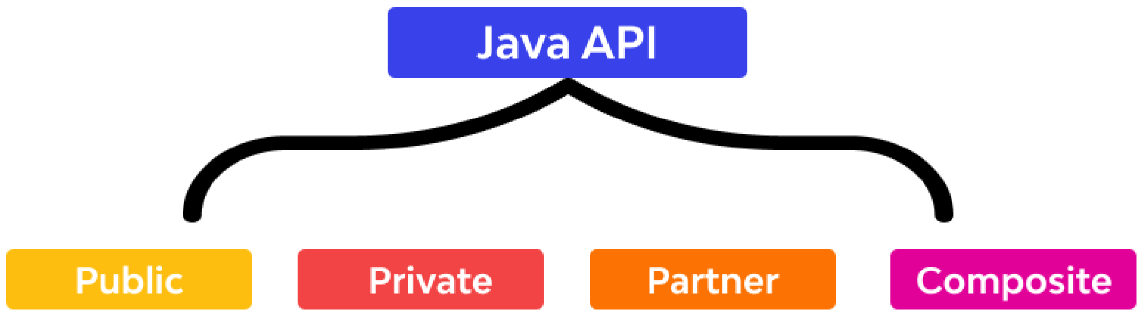

RESTful Web Service in Java
Client
HTTP POST
[{"city":"Paris","units":"C"}]
Request
JSON
Response
[{"low":"16","high":"23"}]
Server
/service/weather
(REST Interface)

-

Types of Java APIs

The five acceptable types of Java API are explained next:

1. Public Java APIs are often referred to as open Java API as they are part of JDK and don't need any extra payment. Also, they are free from the areas and use cases of their implementation.
2. Private or internal Java API is designed by a particular developer/organization and is accessible only to authorized professionals.
3. Partner Java APIs are the third-party APIs offered to businesses for specific operations.
4. Composite Java API is basically microservices developed using clubbing different kinds of APIs.
5. Web Java API is accessed via HTTP protocol and is used to establish a communication bridge for browser-based applications/services like web storage and web notifications.



Java API example

APIs in Java can be custom as well as open-source packages. For the APIs your developer or a service provider has created, you will have to learn relevant functions and get to know how to fetch data or perform functions through it. However, Java's own API have detailed descriptions and guidance available on Oracle's official website and throughout the internet.

Let's take the example of JDBC API. It is the API that lets you access your project's database and fetch data using various legit queries. It has 2 packages, namely - java.sql and javax.sql.

To use the classes in these packages in your code, you must first import these in the beginning of your file. For this, the code will be:

```
import java.sql.*
import javax.sql.*
```

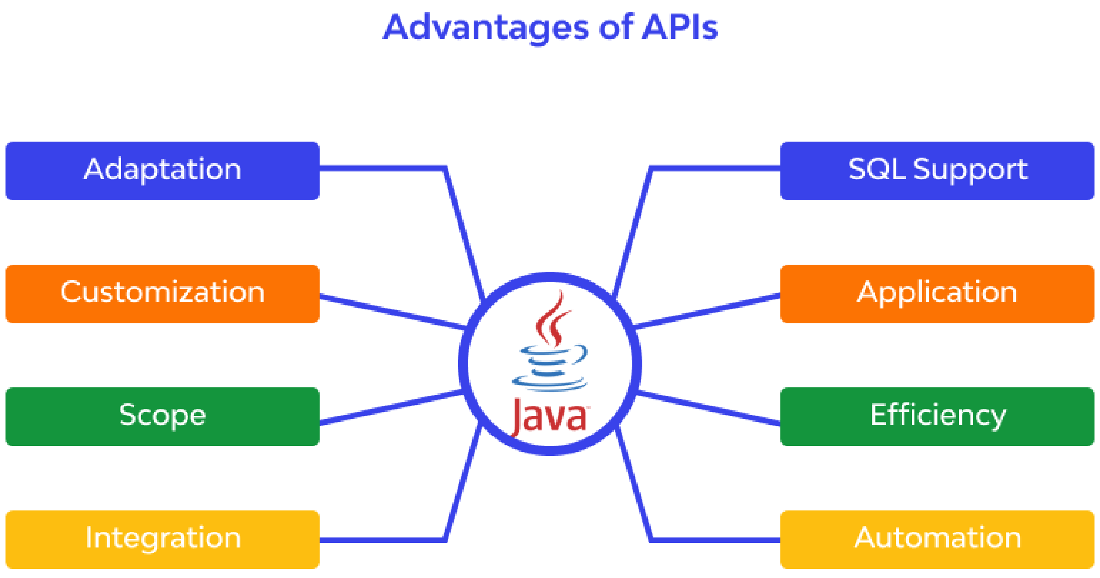
If you want to use a few specific classes from any of these APIs, you can perform the import using following function:

```
import java.sql.Connection;
import java.sql.SQLException;
```

In above example, imported classes are SQLException and Connection. These both belong to java.sql package of JDBC API.

Here is an example code that uses data received using the Java JDBC API:

```
public static void commit() {
  Connection chk_con = this.get();
  if (chk_con != null) {
    try {
      chk_con.commit();
    } catch (SQLException e) {
      e.printStackTrace();
      throw new RuntimeException("Transaction related exception occurred when tried to establish a
connection...");
    }
  }
}
```



Advantages of API in Java

The direct impact of using API in Java is expedient the development as it makes pre-defined classes and packages available for intended programs. However, that’s not the only advantage. There are many more, such as:

- Java APIs keep human involvement as least as possible and empower computers so much that they take care of the entire development job. Java API deploys automaton of highest grade into action and makes crucial workflow quick and error-free.
- Service delivery is much more flexible with Java APIs as these APIs are available for every component and lift all the data access restrictions. Additionally, the content or code generated using Java API is set to be released and work. All the related channels are allowed to access this freshly-generated content. This makes development quick from the ground level.
- Java APIs exhibit unmatched integration abilities as they are ready-to-be-embedded in any software/program/website. The high-end integration makes data fluid enough to be used at any platform wherever enhanced user-experience goal fulfillment demands.

The Java API Standard Library

The Java API Standard Library is an important part of the Java language, and it’s one of the main reasons why Java has become the most popular programming language. This library offers over 100 APIs that are used for everything from building apps to graphics to web services.

These APIs are designed to make your life as a developer easier. They provide access to everything from mathematical functions (the Math class) to X-Ray technology (the X-Ray class). Additionally, they offer ways in which you can integrate your app with other resources on your computer or online, such as Google Maps or Twitter.

If you’re looking for a more advanced way to handle data storage, you can use the ObjectInputStream class—or if you need help making your program work offline, you can use the Service Locator interface. There are many different APIs in the Java API Standard Library that allow developers to accomplish their goals in creative and efficient means.

Securing APIs in Java

APIs, lacking security, bear high risks for the developed applications. This is why developers are allowed to adopt key API security practices without any compromise. There are several frameworks for this job. For instance, if REST API is used in Java then Apache Shiro is a great choice to make.

Using this framework, one can easily execute API token security of your choice. Java EE and Spring offer a robust security framework for Java API. However, one has to make tedious efforts to bring these frameworks into action to reap the benefits. This is why Wallarm is the most preferred choice. It endows developers with tailored-made API security solutions that can protect all sorts of Java API, regardless of the nature of OS.

Java Naming Convention

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of Naming Conventions in Java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

CamelCase in Java naming conventions

Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put
a package statement with that name at the top of *every source*

file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, package graphics;) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If you put the graphics interface and classes listed in the preceding section in a package called graphics, you would need six source files, like this:

```
//in the Draggable.java file
package graphics;
public interface Draggable {
   . . .
}

//in the Graphic.java file
package graphics;
public abstract class Graphic {
   . . .
}

//in the Circle.java file
package graphics;
public class Circle extends Graphic
   implements Draggable {
   . . .
}

//in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic
   implements Draggable {
   . . .
}
```

```
//in the Point.java file
package graphics;
public class Point extends Graphic
   implements Draggable {
   . . .
}

//in the Line.java file
package graphics;
public class Line extends Graphic
   implements Draggable {
   . . .
}
```

If you do not use a package statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

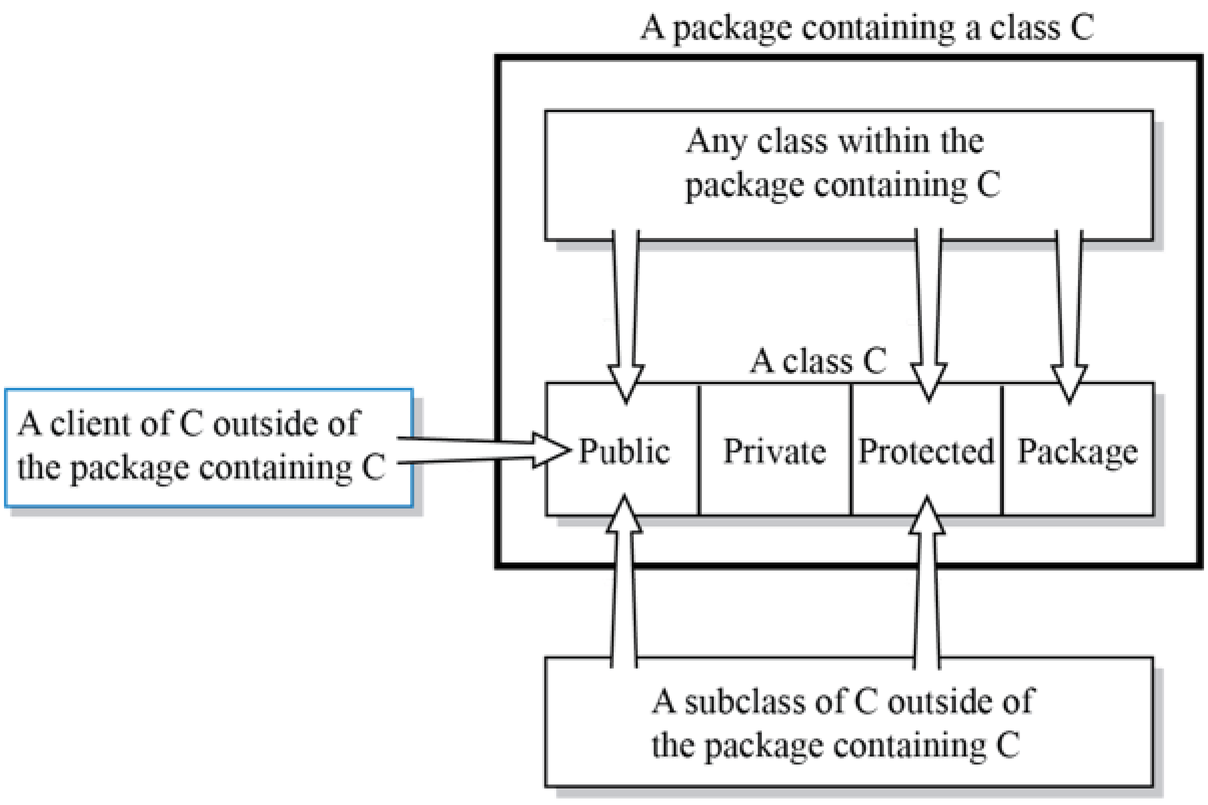
Accessing a Package

A **public class**—whether it is within a package or not—is available to any other class. If you omit the class's access modifier entirely, the class has **default access** and is only available to other classes within its package. This kind of class is said to have **package access**.

A default access modified class can only be accessed within its package. This class cannot be accessed outside the package.

Package access is more restricted than protected access and gives you more control when defining classes. You can use package access in situations where you have a package of cooperating classes that act as a single encapsulated unit. When you control the package directory, you control who is allowed to access the package. This is also referred to as default package access modifier.

Here is an illustration of the various kinds of access:


A package containing a class C
Any class within the package containing C
A class C
Public
Private
Protected
Package
A client of C outside of the package containing C
A subclass of C outside of the package containing C

Adding Classes to Packages

In order to put add Java classes to packages, you must do two things:

1. Put the Java source file inside a directory matching the Java package you want to put the class in.
2. Declare that class as part of the package.

Putting the Java source file inside a directory structure that matches the package structure, is pretty straightforward. Just create a source root directory, and inside that, create directories for each package and subpackage recursively. Put the class files into the directory matching the package you want to add it to.

When you have put your Java source file into the correct directory (matching the package the class should belong to), you have to declare inside that class file, that it belongs to that Java package. Here is how you declare the package inside a Java source file:

```
package com.jenkov.navigation;

public class Page {
    ...
}
```

What Is a Hidden Class?

Hidden classes are classes that cannot be used directly by the bytecode or other classes. Even though it's mentioned as a class, it should be understood to mean either a hidden class or interface. It can also be defined as a member of the access control nest and can be unloaded independently of other classes.

Properties of Hidden Classes

Let's take a look at the properties of these dynamically generated classes:

- Non-discoverable – a hidden class is not discoverable by the JVM during bytecode linkage, nor by programs making explicit use of class loaders. The reflective methods *Class::forName*, *ClassLoader::findLoadedClass*, and *Lookup::findClass* will not find them.
- We can't use the hidden class as a superclass, field type, return type, or parameter type.
- Code in the hidden class can use it directly, without relying on the class object.
- *final* fields declared in hidden classes are not modifiable regardless of their accessible flags.
- It extends the access control nest with non-discoverable classes.
- It may be unloaded even though its notional defining class loader is still reachable.
- Stack traces don't show the methods or names of hidden classes by default, however, tweaking JVM options can show them.

Creating Hidden Classes

The hidden class isn't created by any class loader. It has the same defining class loader, runtime package, and protection domain of the lookup class.

First, let's create a *Lookup* object:

```
MethodHandles.Lookup lookup = MethodHandles.lookup();Copy
```

The *Lookup::defineHiddenClass* method creates the hidden class. This method accepts an array of bytes.

For simplicity, we'll define a simple class with the name *HiddenClass* that has a method to convert a given string to uppercase:

```
public class HiddenClass {
    public String convertToUpperCase(String s) {
        return s.toUpperCase();
    }
}Copy
```

Let's get the path of the class and load it into the input stream. After that, we'll convert this class into bytes using *IOUtils.toByteArray()*:

```
Class<?> clazz = HiddenClass.class;
String className = clazz.getName();
String classAsPath = className.replace('.', '/') + ".class";
InputStream stream = clazz.getClassLoader()
    .getResourceAsStream(classAsPath);
byte[] bytes = IOUtils.toByteArray();Copy
```

Lastly, we pass these constructed bytes into *Lookup::defineHiddenClass*:

```
Class<?> hiddenClass = lookup.defineHiddenClass(IOUtils.toByteArray(stream),
  true, ClassOption.NESTMATE).lookupClass();Copy
```

The second *boolean* argument *true* initializes the class. The third argument *ClassOption.NESTMATE* specifies that the created hidden class will be added as a nestmate to the lookup class so that it has access to the *private* members of all classes and interfaces in the same nest.

Suppose we want to bind the hidden class strongly with its class loader, *ClassOption.STRONG*. This means that the hidden class can only be unloaded if its defining loader is not reachable.

Using Hidden Classes

Hidden classes are used by frameworks that generate classes at runtime and use them indirectly via reflection.

In the previous section, we looked at creating a hidden class. In this section, we'll see how to use it and create an instance.

Since casting the classes obtained from *Lookup.defineHiddenClass* is not possible with any other class object, we use *Object* to store the hidden class instance. If we wish to cast the hidden class, we can define an interface and create a hidden class that implements the interface:

```
Object hiddenClassObject = hiddenClass.getConstructor().newInstance();Copy
```

Now, let's get the method from the hidden class. After getting the method, we'll invoke it as any other standard method:

```
Method method = hiddenClassObject.getClass()
    .getDeclaredMethod("convertToUpperCase", String.class);
Assertions.assertEquals("HELLO", method.invoke(hiddenClassObject, "Hello"));CopyCopy
```

Now, we can verify a few properties of a hidden class by invoking some of its methods:

The method *isHidden()* will return *true* for this class:

```
Assertions.assertEquals(true, hiddenClass.isHidden());Copy
```

Also, since there's no actual name for a hidden class, its canonical name will be *null*:

```
Assertions.assertEquals(null, hiddenClass.getCanonicalName());Copy
```

The hidden class will have the same defining loader as the class that does the lookup. Since the lookup happens in the same class, the following assertion will be successful:

```
Assertions.assertEquals(this.getClass()
  .getClassLoader(), hiddenClass.getClassLoader());Copy
```

If we try to access the hidden class through any methods, they'll throw *ClassNotFoundException*. This is obvious, as the hidden class name is sufficiently unusual and unqualified to be visible to other classes. Let's check a couple of assertions to prove that the hidden class is not discoverable:

```
Assertions.assertThrows(ClassNotFoundException.class, () ->
Class.forName(hiddenClass.getName()));
Assertions.assertThrows(ClassNotFoundException.class, () ->
lookup.findClass(hiddenClass.getName()));
```

Java Static Import

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage of static import:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

Simple Example of static import

```
import static java.lang.System.*;
class StaticImportExample{
  public static void main(String args[]){
    out.println("Hello");//Now no need of System.out
    out.println("Java");
  }
}
```

JAVA FILES AND I/O

What is a stream?

- What is a Stream and what are the types of Streams and classes in Java? Java provides I/O Streams to read and write data where, a Stream represents an input source or an output destination which could be a file, I/o devise, other program etc. In general, a Stream will be an input stream or, an output stream.

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
The features of Java stream are –

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-
Intermediate Operations:

1. **map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.
   ```
   List number = Arrays.asList(2,3,4,5);
   List square = number.stream().map(x-
   >x*x).collect(Collectors.toList());
   ```
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.
   ```
   List names = Arrays.asList("Reflection","Collection","Stream");
   List result = names.stream().filter(s-
   >s.startsWith("S")).collect(Collectors.toList());
   ```
3. **sorted:** The sorted method is used to sort the stream.
   ```
   List names = Arrays.asList("Reflection","Collection","Stream");
   List result =
   names.stream().sorted().collect(Collectors.toList());
   ```

Terminal Operations:

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.
   ```
   List number = Arrays.asList(2,3,4,5,3);
   ```

```
Set square = number.stream().map(x-
>x*x).collect(Collectors.toSet());
```

2. **forEach:** The forEach method is used to iterate through every element of the stream.
   ```
   List number = Arrays.asList(2,3,4,5);
   number.stream().map(x->x*x).forEach(y->System.out.println(y));
   ```
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.
   The reduce method takes a BinaryOperator as a parameter.
   ```
   List number = Arrays.asList(2,3,4,5);
   int even = number.stream().filter(x-
   >x%2==0).reduce(0,(ans,i)-> ans+i);
   ```
   Here ans variable is assigned 0 as the initial value and i is added to it .

Program to demonstrate the use of Stream

```
//a simple program to demonstrate the use of stream in java

import java.util.*;

import java.util.stream.*;


class Demo

{

  public static void main(String args[])

  {


    // create a list of integers

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


    // demonstration of map method

    List<Integer> square = number.stream().map(x -> x*x).

                           collect(Collectors.toList());

    System.out.println(square);


    // create a list of String

    List<String> names =

                Arrays.asList("Reflection","Collection","Stream");
```

```
    // demonstration of filter method
    List<String> result = names.stream().filter(s->s.startsWith("S")).
                          collect(Collectors.toList());
    System.out.println(result);

    // demonstration of sorted method
    List<String> show =
            names.stream().sorted().collect(Collectors.toList());
    System.out.println(show);

    // create a list of integers
    List<Integer> numbers = Arrays.asList(2,3,4,5,2);

    // collect method returns a set
    Set<Integer> squareSet =
         numbers.stream().map(x->x*x).collect(Collectors.toSet());
    System.out.println(squareSet);

    // demonstration of forEach method
    number.stream().map(x->x*x).forEach(y->System.out.println(y));

    // demonstration of reduce method
    int even =
       number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

    System.out.println(even);
  }
}
```

Output:

```
[4, 9, 16, 25]
```

```
[Stream]
[Collection, Reflection, Stream]
[16, 4, 9, 25]
4
9
16
25
6
```

Important Points/Observations:

1. A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
2. Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

Reading and writing to files (only txt files)

When programming, whether you're creating a mobile app, a web application, or just writing scripts, you often have the need to read or write data to a file. This data could be cache data, data you retrieved for a dataset, an image, or just about anything else you can think of.

In this tutorial, we are going to show the most common ways you can read and write to files in Java.

Java provides several API (also known as Java I/O) to read and write files since its initial releases. With subsequent releases, Java I/O has been improved, simplified and enhanced to support new features.

Before we get in to some actual examples, it would help to understand the classes available to you that will handle the reading and writing of data to files. In the following sections we'll provide a brief overview of the Java I/O classes and explain what they do, then we'll take a look at Java NIO Streams, and finally we'll show some examples of reading and writing data to files.

I/O Streams

There are two types of Streams you can use to interact with files:

1. Character Streams

2. Byte Streams

For each of the above stream types, there are several supporting classes shipped with Java, which we'll take a quick look at below.

Character Streams

Character Streams are used to read or write the characters data type. Let's look at the most commonly used classes. All of these classes are defined under `java.io` package.

Here are some classes you should know that can be used to **read** character data:

- Reader: An abstract class to read a character stream.
- InputStreamReader: Class used to read the byte stream and converts to character stream.
- FileReader: A class to read the characters from a file.
- BufferedReader: This is a wrapper over the `Reader` class that supports buffering capabilities. In many cases this is most preferable class to read data because more data can been read from the file in one `read()` call, reducing the number of actual I/O operations with file system.

And here are some classes you can use to **write** character data to a file:

- Writer: This is an abstract class to write the character streams.
- OutputStreamWriter: This class is used to write character streams and also convert them to byte streams.
- FileWriter: A class to actually write characters to the file.
- BufferedWriter: This is a wrapper over the `Writer` class, which also supports buffering capabilities. This is most preferable class to write data to a file since more data can be written to the file in one `write()` call. And like the `BufferedReader`, this reduces the number of total I/O operations with file system.

Byte Streams

Byte Streams are used to read or write byte data with files. This is different from before in the way they treat the data. Here you work with raw bytes, which could be characters, image data, unicode data (which takes 2 bytes to represent a character), etc.

In this section we'll take a look at the most commonly used classes. All of these classes are defined under `java.io` package.

Here are the classes used to **read** the byte data:

- InputStream: An abstract class to read the byte streams.
- FileInputStream: A class to simply read bytes from a file.
- BufferedInputStream: This is a wrapper over `InputStream` that supports buffering capabilities. As we saw in the character streams, this is a more efficient method than `FileInputStream`.

And here are the classes used to **write** the byte data:

- OutputStream: An abstract class to write byte streams.
- FileOutputStream: A class to write raw bytes to the file.
- ByteOutputStream: This class is a wrapper over `OutputStream` to support buffering capabilities. And again, as we saw in the character streams, this is a more efficient method than `FileOutputStream` thanks to the buffering.

Java NIO Streams

Java NIO is a non-blocking I/O API which was introduced back in Java 4 and can be found in the `java.nio` package. In terms of performance, this is a big improvement in the API for I/O operations.

Buffers, Selectors, and Channels are the three primary components of Java NIO, although in this article we'll focus strictly on using the NIO classes for interacting with files, and not necessarily the concepts behind the API.

As this tutorial is about reading and writing files, we will discuss only the related classes in this short section:

- Path: This is a hierarchical structure of an actual file location and is typically used to locate the file you want to interact with.
- Paths: This is a class that provides several utility methods to create a `Path` from a given string URI.
- Files: This is another utility class which has several methods to read and write files without blocking the execution on threads.

Input and Output systems

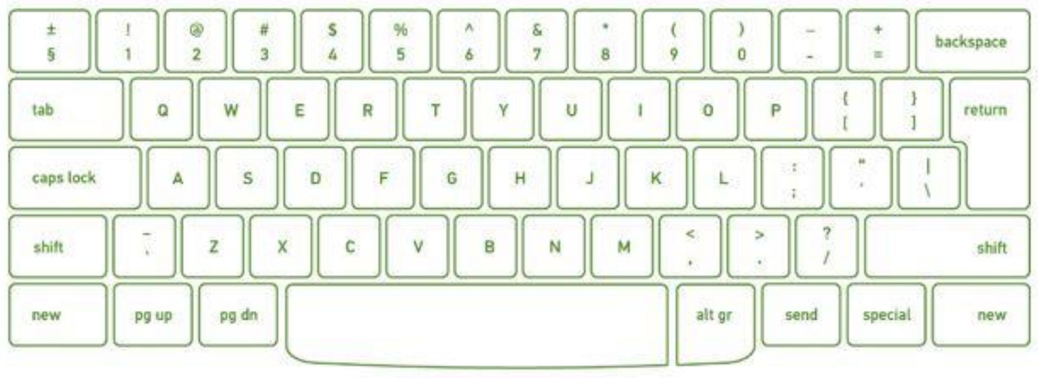
An input/output device, often known as an IO device, is any hardware that allows a human operator or other systems to interface with a computer. Input/output devices, as the name implies, are capable of delivering data (output) to and receiving data from a computer (input). An input/output (I/O) device is a piece of hardware that can take, output, or process data. It receives data as input and provides it to a computer, as well as sends computer data to storage media as a storage output.

There are many IO Devices available, some of them are:

Input Devices

Keyboard

The keyboard is the most frequent and widely used input device for entering data into a computer. Although there are some additional keys for performing other operations, the keyboard layout is similar to that of a typical typewriter.
Generally, keyboards come in two sizes: 84 keys or 101/102 keys, but currently keyboards with 104 keys or 108 keys are also available for Windows and the Internet.



Types of Keys

- **Numeric Keys:** It is used to enter numeric data or move the cursor. It usually consists of a set of 17 keys.
- **Typing Keys:** The letter keys (A-Z) and number keys (09) are among these keys.
- **Control Keys:** These keys control the pointer and the screen. There are four directional arrow keys on it. Home, End, Insert, Alternate(Alt), Delete, Control(Ctrl), etc., and Escape are all control keys (Esc).

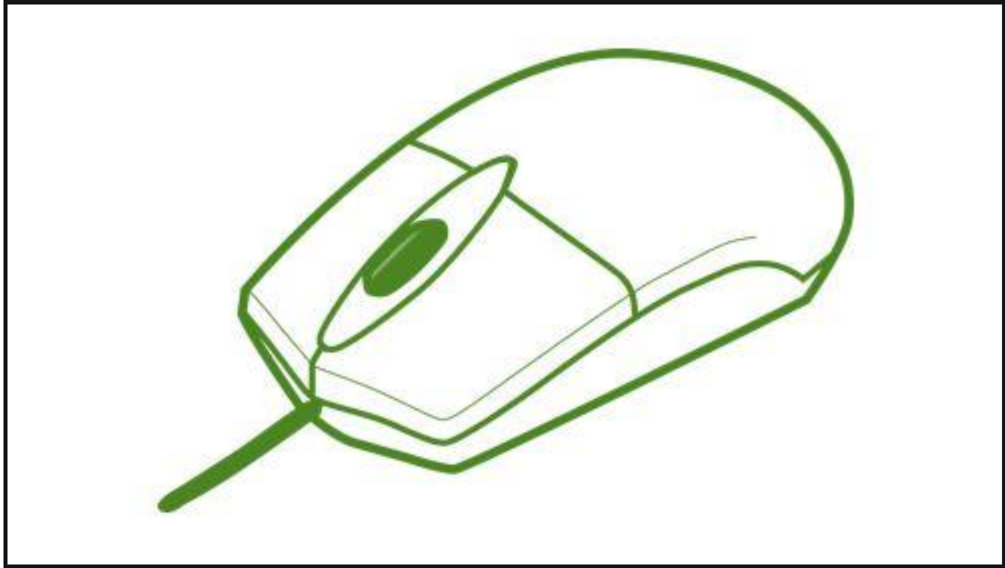
- **Special Keys:** Enter, Shift, Caps Lock, NumLk, Tab, etc., and Print Screen are among the special function keys on the keyboard.
- **Function Keys:** The 12 keys from F1 to F12 on the topmost row of the keyboard.

Mouse

The most common pointing device is the mouse. The mouse is used to move a little cursor across the screen while clicking and dragging. The cursor will stop if you let go of the mouse. The computer is dependent on you to move the mouse; it won't move by itself. As a result, it's an input device.

A mouse is an input device that lets you move the mouse on a flat surface to control the coordinates and movement of the on-screen cursor/pointer.

The left mouse button can be used to select or move items, while the right mouse button when clicked displays extra menus.



Joystick

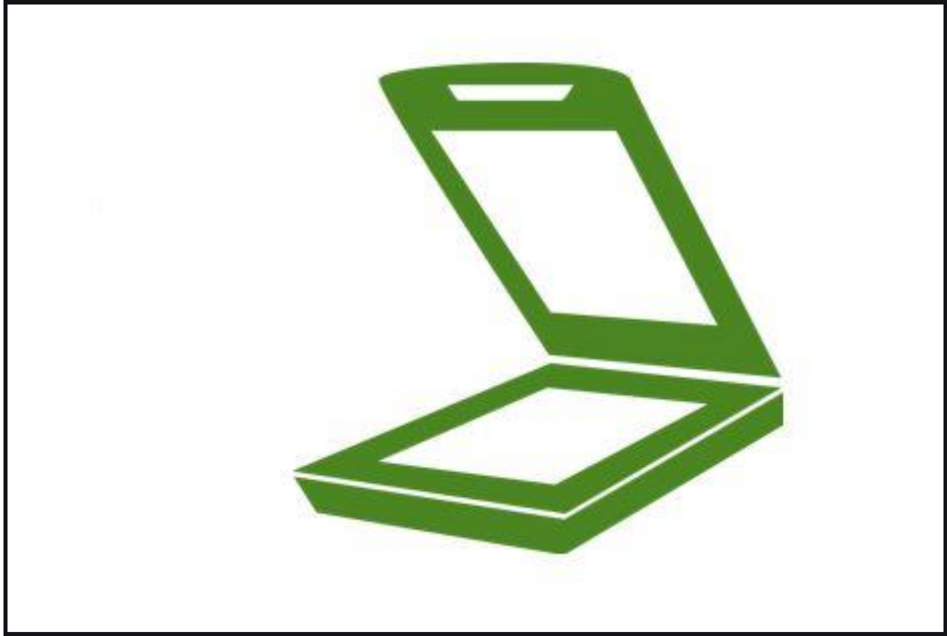
A joystick is a pointing device that is used to move the cursor on a computer screen. A spherical ball is attached to both the bottom and top ends of the stick. In a socket, the lower spherical ball slides. You can move the joystick in all four directions.

The joystick’s function is comparable to that of a mouse. It is primarily used in CAD (Computer-Aided Design) and playing video games on the computer.

Light Pen
A light pen is a type of pointing device that looks like a pen. It can be used to select a menu item or to draw on the monitor screen. A photocell and an optical system are enclosed in a tiny tube.
When the tip of a light pen is moved across a monitor screen while the pen button is pushed, the photocell sensor element identifies the screen location and provides a signal to the CPU.

Scanner
A scanner is an input device that functions similarly to a photocopier. It’s employed when there’s information on paper that needs to be transferred to the computer’s hard disc for subsequent manipulation. Scanner collects images from the source and converts them to a digital

format that may be saved on a disc. Before they are printed, these images can be modified.



OCR
OCR stands for optical character recognition, and it is a device that reads printed text. OCR optically scans the text, character by character, turns it into a machine-readable code, and saves it to the system memory.

Bar Code Reader
A bar code reader is a device that reads data that is bar-coded (data that is represented by light and dark lines).
Bar-coded data is commonly used to mark things, number books, and so on. It could be a handheld scanner or part of a stationary scanner.
A bar code reader scans a bar code image, converts it to an alphanumeric value, and then sends it to the computer to which it is connected.

Web Camera

Because it records a video image of the scene in front of it, a webcam is an input device. It is either built inside the computer (for example, a laptop) or attached through a USB connection.
A webcam is a computer-connected tiny digital video camera. It’s also known as a web camera because it can take images and record video. These cameras come with software that must be installed on the computer in order to broadcast video in real-time over the Internet. It can shoot images and HD videos, however, the video quality isn’t as good as other cameras (In Mobiles or other devices or normal cameras).

Output Devices

Monitor

Monitors, also known as Visual Display Units (VDUs), are a computer's primary output device. It creates images by arranging small dots, known as pixels, in a rectangular pattern. The amount of pixels determines the image's sharpness.

The two kinds of viewing screen used for monitors are:

(1) Cathode-Ray Tube (CRT): Pixels are minuscule visual elements that make up a CRT display. The higher the image quality or resolution, the smaller the pixels.

(2) Flat-Panel Display Cathode-Ray Tube Monitor: In comparison to the CRT, a flat-panel display is a type of video display with less volume, weight, and power consumption. They can be hung on the wall or worn on the wrist.

Flat-panel displays are currently used in calculators, video games, monitors, laptop computers, and graphical displays.

Printer

Printers are output devices that allow you to print information on paper.

There are two types of printers:

(a) Impact Printer:
Characters are printed on the ribbon, which is subsequently crushed against the paper, in impact printers. The following are the characteristics of impact printers:

- Exceptionally low consumable cost.
- Quite noisy
- Because of its low cost, it is ideal for large-scale printing.
- To create an image, there is physical contact with the paper.

(b) Non-Impact Printers:
Characters are printed without the need for a ribbon in non-impact printers. Because these printers print a full page at a time, they're also known as Page Printers. The following are the characteristics of non-impact printers:

- Faster
- They don't make a lot of noise.
- Excellent quality
- Supports a variety of typefaces and character sizes

Manipulating Input data

Data manipulation is the method of organizing data to make it easier to read or more designed or structured. For instance, a collection of any kind of data could be organized in alphabetical order so that it can be understood easily. On the other hand, it can be difficult to find information about any particular employee in an organization if all the employees' information is not organized. Therefore, all the employee's information could be organized in alphabetical order that makes it easier to find information easily of any individual employee. Data manipulation helps website owners to monitor their sources of traffic and their most popular pages. Hence, it is frequently used on web server logs.

Data manipulation is also used by accounting users or similar fields to organized data in order to figure out product costs, future tax obligations, pricing patterns, etc. It also helps the stock market predictors to forecast developments and predicts how stocks might perform in the adjacent future. Furthermore, data manipulation may also use by computers to display information to users in a more realistic way on the basis of web pages, the code in a software program, or data formatting.

The DML is used to manipulate data, which is a programming language. It short for Data Manipulation Language that helps to modify data like adding, removing, and altering databases. It means that changing the information in a way that can be read easily.

Objective of Data Manipulation

Data manipulation is a key feature for business operations and optimization. You need to deal with data in a proper manner and manipulate it into meaningful information like doing trend analysis, financial data, and consumer behaviour. Data manipulation offers an organization multiple advantages; some are discussed below:

- **Consistent data:** Data manipulation provides a way to organize your data inconsistent format that makes it structured, which can be read easily and better understood. When you are collecting data from different-different sources, you may not have a unified view; but data manipulation provides you surety that the data is well-organized, structured, and stored consistently.
- **Project data:** Especially when it comes to finances, data manipulation is more useful as it helps to provide more in-depth analysis by using historical data to project the future.
- **Delete or neglect redundant data:** Data manipulation helps to maintain your data and delete unusable data that is always present.
- Overall, with the data, you can do many operations such as edit, delete, update, convert, and incorporate data into a database. It helps to create more value from the data. If you do not know how to use data in an effective manner, it becomes pointless. Therefore, it will be beneficial to make better business decisions when you are able to organize your data accordingly.

Steps involved in Data Manipulation

Below there are some important steps given that may help you out to get started with data manipulation.

1. First of all, data manipulation is possible only if you have data. Therefore, you are required to create a database that is generated from data sources.
2. This knowledge needs restructuring and reorganization, which could be done with data manipulation that helps you to cleanse your information.
3. Then, you need to import a database and create it to get start work with data.
4. With the help of data manipulation, you can edit, delete, merge, or combine your information.
5. Finally, data analysis becomes easier at the time of manipulating data.

Opening and Closing Stream

Opening data streams is similar to opening files. You can open a stream by using the **AVIFileGetStream** function. This function creates a stream interface, places a handle of the stream in the interface, and returns a pointer to the interface. **AVIFileGetStream** also maintains a reference count of the applications that have opened a stream, but not of those that have closed it.

If you want to access a single stream in a file, you can open the file and the stream by using the **AVIStreamOpenFromFile** function. This function combines the operations and function arguments of the **AVIFileOpen** and **AVIFileGetStream** functions.

To access more than one stream in a file, use **AVIFileOpen** once followed by multiple calls to **AVIFileGetStream**.

You can increment the reference count of a stream by using the **AVIStreamAddRef** function to keep a stream open when using a function that would normally close the stream.

You can close a stream by using the **AVIStreamRelease** function. This function decrements the reference count of the stream and closes it when the reference count reaches zero. Your applications should balance the reference count by including a call to **AVIStreamRelease** for each use of the **AVIFileGetStream**, **AVIFileCreateStream**, **AVIStreamAddRef**, or **AVIStreamOpenFromFile** function. When you release a stream that has been opened by using **AVIStreamOpenFromFile**, AVIFile closes the file containing the stream. If your application releases a file that has open data streams, AVIFile will not close the streams until all of the streams are released.

Predefined streams

System class from *java.lang* package contains three predefined stream variables:

1. in
2. out
3. err

These fields are declared as *public*, *static*, and *final* within **System**.

System.out refers to the standard output stream. By default, this is the console.

System.in refers to standard input, which is the keyboard by default.

System.err refers to the standard error stream, which also is the console by default.

However, these streams may be redirected to any compatible I/O device.

System.in is an object of type **InputStream**

System.out and **System.err** are objects of type **PrintStream**.

These are byte streams, even though they are typically used to read and write characters from and to the console.

You can wrap these within character-based streams.

The preceding chapters have been using **System.out** in their examples.

You can use **System.eerr** in much the same way.

As you know, all Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. Among other things, it contains three predefined stream variables, called **in**, **out**, and **err**. These fields are declared as **public**, **final**, and **static** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.

System. out refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is by default the keyboard. **System. err** refers to the standard error stream, which is also the console by default. However, these streams can be

File handling Classes and Methods

In Java, with the help of File Class, we can work with files. This File Class is inside the java.io package. The File class can be used by creating an object of the class and then specifying the name of the file.

Why File Handling is Required?

- File Handling is an integral part of any programming language as file handling enables us to store the output of any particular program in a file and allows us to perform certain operations on it.
- In simple words, file handling means reading and writing data to a file.

- Java

```java
// Importing File Class
import java.io.File;

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

        // File name specified
        File obj = new File("myfile.txt");
          System.out.println("File Created!");
    }
}
```

Output

In Java, the concept Stream is used in order to perform I/O operations on a file. So at first, let us get acquainted with a concept known as Stream in Java.

Streams in Java

- In Java, a sequence of data is known as a stream.
- This concept is used to perform I/O operations on a file.
- There are two types of streams :

1. Input Stream:

The Java InputStream class is the superclass of all input streams. The input stream is used to read data from numerous input devices like the keyboard, network, etc. InputStream is an abstract class, and because of this, it is not useful by itself. However, its subclasses are used to read data.

There are several subclasses of the InputStream class, which are as follows:

1. AudioInputStream
2. ByteArrayInputStream
3. FileInputStream
4. FilterInputStream
5. StringBufferInputStream
6. ObjectInputStream

Creating an InputStream

2. Output Stream:

The output stream is used to write data to numerous output devices like the monitor, file, etc. OutputStream is an abstract superclass that represents an output stream. OutputStream is an abstract class and because of this, it is not useful by itself. However, its subclasses are used to write data.

There are several subclasses of the OutputStream class which are as follows:

1. ByteArrayOutputStream
2. FileOutputStream
3. StringBufferOutputStream
4. ObjectOutputStream
5. DataOutputStream
6. PrintStream

Creating an OutputStream

Based on the data type, there are two types of streams :

1. Byte Stream:

This stream is used to read or write byte data. The byte stream is again subdivided into two types which are as follows:

- **Byte Input Stream:** Used to read byte data from different devices.
- **Byte Output Stream:** Used to write byte data to different devices.

2. Character Stream:

This stream is used to read or write character data. Character stream is again subdivided into 2 types which are as follows:

- **Character Input Stream:** Used to read character data from different devices.
- **Character Output Stream:** Used to write character data to different devices.

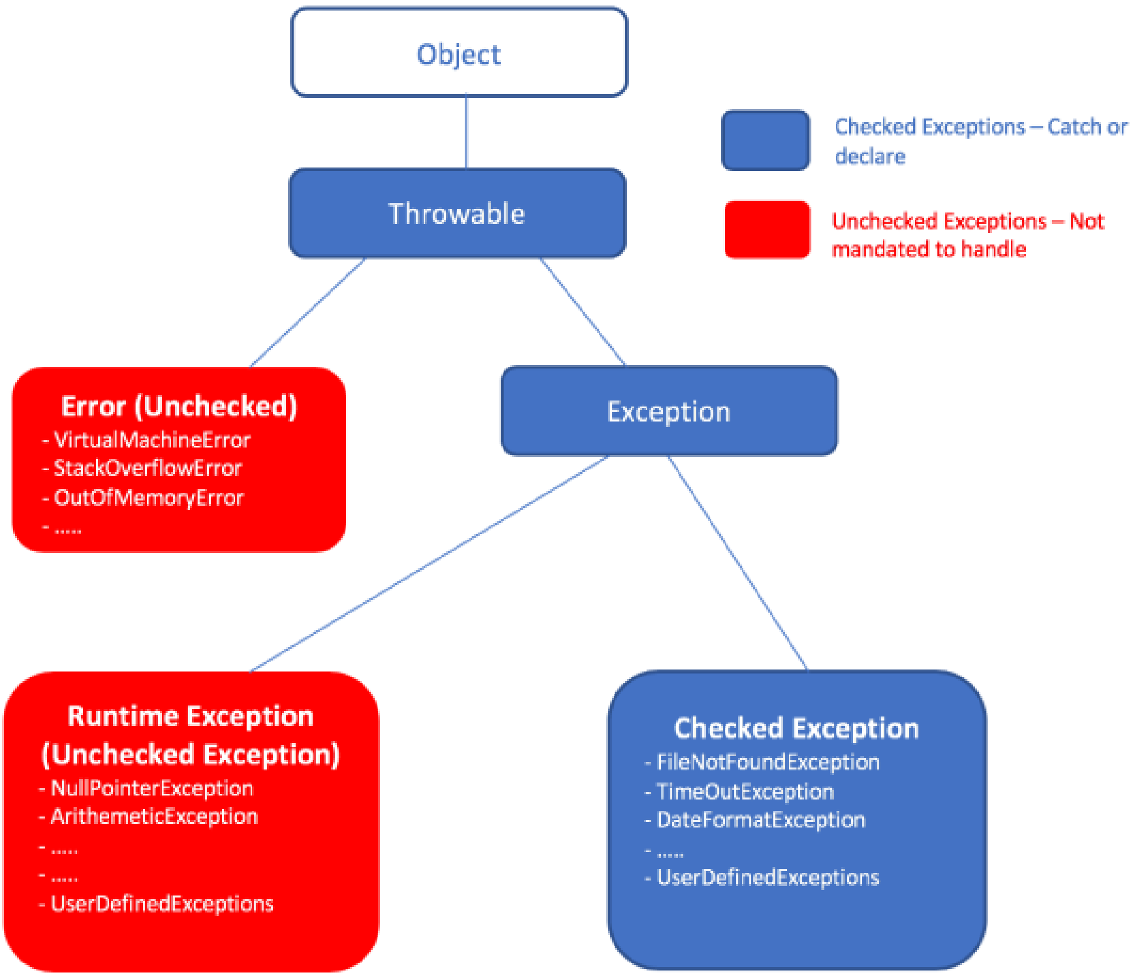
Owing to the fact that you know what a stream is, let’s polish up File Handling in Java by further understanding the various methods that are useful for performing operations on the files like creating, reading, and writing files.

Exception handling

Exception Overview

An **exception is** an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. Instead of executing the next instruction in the sequence, the control is transferred to the Java Virtual Machine (JVM) which tries to find an appropriate exception handler in the program and transfer control to it (hence disrupting the normal program flow).

This diagram illustrates the class hierarchy of the Throwable class and its most significant subclasses.



Exceptions can be

- Checked Exceptions (direct subclasses of Exception or Custom exceptions that extend Exception class)
- Unchecked Exceptions or RuntimeException(direct subclasses of RuntimeException or Custom exceptions that extend RuntimeException class)

Checked Exceptions

- Checked exceptions represent an exceptional condition that is usually recoverable
- Direct subclasses of Exception or Custom exceptions that extend Exception class

In these scenarios, your Java application is trying to connect or use outside program/ software/resource. The complier makes sure that once your java application is done using the outside resource, the resource is released gracefully with out any errors. So, it is mandatory to **handle** these errors. The compiler always complains if you don't. It says "OK, if you do not handle, then I won't create class files for your application". **Handling checked exceptions are mandatory.**
Examples:

- **Either** use direct sub classes of Exception class. Below are few listed examples
  - IOException signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.
  - FileNotFoundException indicates when a referenced file is not found on the file system
  - SQLException provides information on a database access error or other errors
- **OR** define a custom class that extends Exception class
  - Ex: public class MovieNotFoundException extends Exception

As you can see in above examples, IOException, FileNotFoundException, SQLException etc are exceptions thrown when your Java application is trying to connect with outside resource like accessing input output device or accessing file system or accessing database.

Exception Keywords

Java provides try, catch, finally, throw, throws keywords to dealing with exceptions. try-catch, finally are used for handle exceptions. 'throw' is used to throw our own exception. 'throws' is used to declare an exception.

Keyword	**Description**
try	We have to place risky code inside "**try**" block and the corresponding handling code inside catch block. (The code which may cause an exception is called risky code)
catch	The "**catch**" block is used to handle the exception which is raised in try block. It should be preceded by try block which means we can't use catch block alone without try block. It can be followed by finally block later.
finally	A **finally** block contains all the crucial statements that must be executed whether exception occurs or not. Crucial statements could be closing a connection, stream, file etc. ('cleanup' or 'resource releasing' code).
throw	The "**throw**" keyword is used to throw an exception by developer when business rules of an application is violated.
throws	The "**throws**" keyword is used to declare exceptions as part of method signature. It doesn't really throw an exception. It just specifies that, method body code may raise an execution in its execution.

Catching Exceptions

This section describes how to use the three exception handler components — the try, catch, and finally blocks — to write an exception handler. Then, the try-with-resources statement, introduced in Java SE 7, is explained. The try-with-resources statement is particularly suited to situations that use Closeable resources, such as streams.

The last part of this section walks through an example and analyzes what occurs during various scenarios.

The following example defines and implements a class named ListOfNumbers. When constructed, ListOfNumbers creates an ArrayList that contains 10 Integer elements with sequential values 0 through 9. The ListOfNumbers class also defines a method named writeList(), which writes the list of numbers into a text file called OutFile.txt. This example uses output classes defined in java.io, which are covered in the Basic I/O section.

```
// Note: This class will not compile yet.
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(i);
        }
    }

    public void writeList() {
    // The FileWriter constructor throws IOException, which must be caught.
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {
            // The get(int) method throws IndexOutOfBoundsException, which must be
caught.
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }
}
```
Copy

The first line in boldface is a call to a constructor. The constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an IOException. The second boldface line is a call to the ArrayList class's get method, which throws an IndexOutOfBoundsException if the value of its argument is too small (less than 0) or too large (more than the number of elements currently contained by the ArrayList.

If you try to compile the ListOfNumbers class, the compiler prints an error message about the exception thrown by the FileWriter constructor. However, it does not display an error message about the exception thrown by get(). The reason is that the exception thrown by the constructor, IOException, is a checked exception, and the one thrown by the get() method, IndexOutOfBoundsException, is an unchecked exception.

Now that you're familiar with the ListOfNumbers class and where the exceptions can be thrown within it, you're ready to write exception handlers to catch and handle those exceptions.

Exception Methods

An unlikely event which disrupts the normal flow of the program is known as an Exception. Java Exception Handling is an object-oriented way to handle exceptions. When an error occurs during the execution of the program, an exception object is created which contains the information about the hierarchy of the Exception and other information which is essential for debugging.

Types of Exceptions:

- Checked Exceptions
- Unchecked Exceptions

Handling the Exceptions:

Example 1:

We are asked to distribute chocolates to every class in the school based on the average performance of the class. We are given two arrays of equal length. One array contains the number of chocolates present in each box and the second array contains the number of students in each class. If the average performance of a class is below par, they are not eligible to get any chocolates and the number of chocolates present in the box will be equally distributed among all other boxes. Each class is given one chocolate box respectively.

So to divide the number of chocolates among the students in a class, we have to divide the number of chocolates by the number of students in the class. Suppose the average performance of a class is below par they wouldn't get any chocolates and the number of students in that section would be zero. One can face a divide by zero exception while solving the above problem. In order to overcome it, we can use a try-catch block and ask the user to update the information given.

Below is the implementation of the above approach :

- Java

```java
// Java program to demonstrate Arithmetic Exception

class GFG {
    public static void main(String[] args)
    {
        // Number of chocolates in each box
        int chocolates[] = { 106, 145, 123, 127, 125 };

        // Number of students in class
        int students[] = { 35, 40, 0, 34, 60 };

        // Number of chocolates given to each student of a
        // particular class
        int numChoc[] = new int[5];
        try {
            for (int i = 0; i < 5; i++) {
                // Calculating the chocolates
                // to be distributed
                numChoc[i] = chocolates[i] / students[i];
            }
        }
        // Catching Divide by Zero Exception
        catch (ArithmeticException error) {
            System.out.println("Arithmetic Exception");
            System.out.println(error.getMessage()
                               + " error.");
        }
    }
}
```

Output

```
Arithmetic Exception

/ by zero error.
```

Example 2:

Java has a robust Error Handling Mechanism that lets us handle multiple Exceptions in one try block using different catch blocks. Catch blocks in java are like if-else statements that will become active when an exception occurs. When an exception occurs, the program compares the exception object generated to the exception specified in the catch blocks. The program checks the first catch block then moves on to other and so-on until the generated exception is matched. If no catch block is matched, the program halts, and an exception is thrown at the console.

After an Exception is generated in the try block, the control immediately shifts to the catch block, and try block will no longer execute. Tinker with the below code by changing the sizes of the array or changing a particular element in the array2 to zero or initializing the answer array, to get a better understanding of Java Exception Handling.

Below code illustrates how various types of errors can be handled in a single try block.

- Java

```java
// Java Program to Handle Various Exceptions

class GFG {
    public static void main(String[] args)
    {
        // Array1 Elements
        int[] array1 = { 2, 4, 6, 7, 8 };

        // Array2 Elements
        int[] array2 = { 1, 2, 3, 4, 5 };
        // Initialized to null value
        int[] ans = null;
        try {
            for (int i = 0; i < 5; i++) {
                ans[i] = array1[i] / array2[i];
                // Generates Number Format Exception
                Integer.parseInt("Geeks for Geeks");
            }
        }
        catch (ArithmeticException error) {
            System.out.println(
                "The catch block with Arithmetic Exception is executed");
        }
        catch (NullPointerException error) {
            System.out.println(
                "The catch block with Null Pointer Exception is executed");
        }
        catch (ArrayIndexOutOfBoundsException error) {
            System.out.println(
                "The catch block with Array Index Out Of Bounds Exception is executed");
        }
        catch (NumberFormatException error) {
            System.out.println(
                "The catch block with Number Format Exception is executed");
        }
        // Executes when an exception which
        // is not specified above occurs
        catch (Exception error) {
            System.out.println(
                "An unknown exception is found "
                + error.getMessage());
        }

        // Executes after the catch block
        System.out.println("End of program");
    }
}
```

Output

```
The catch block with Null Pointer Exception is executed

End of program
```

Declaring Exceptions

Exception Handling in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IO Exception, SQL Exception, Remote Exception, etc.

Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

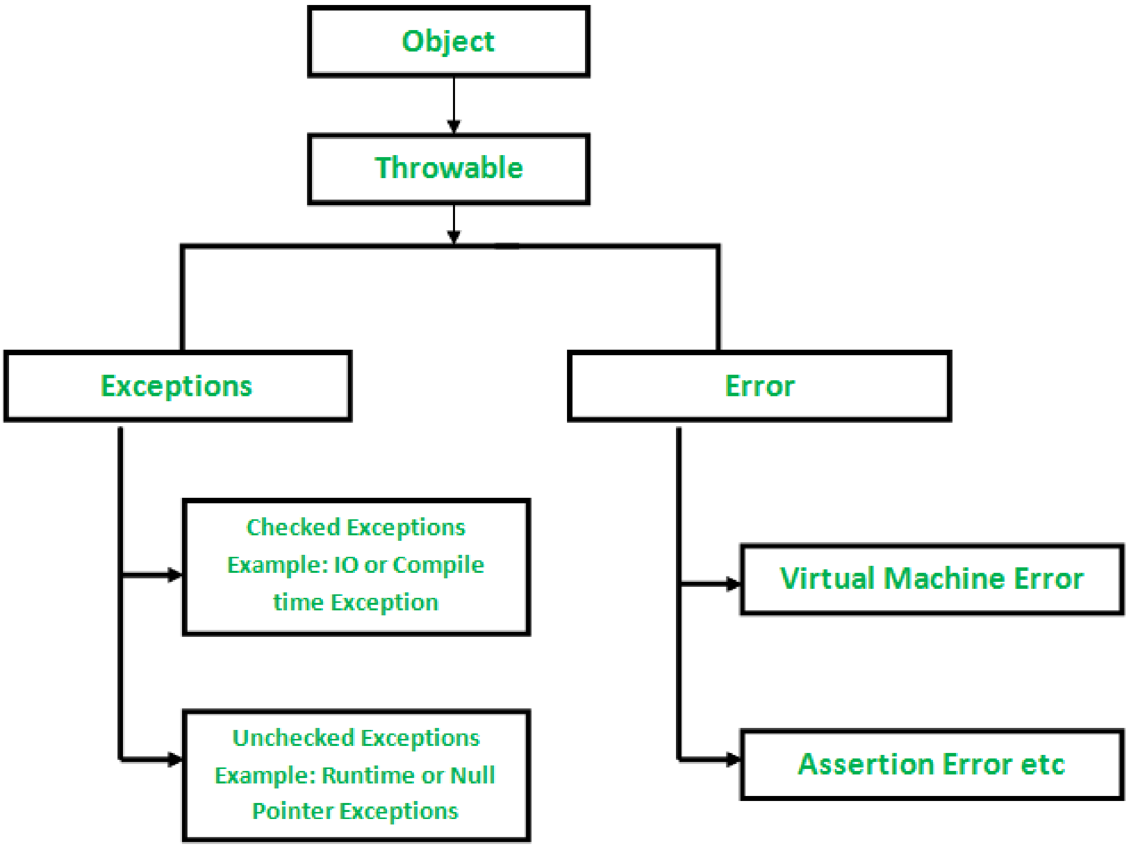
Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

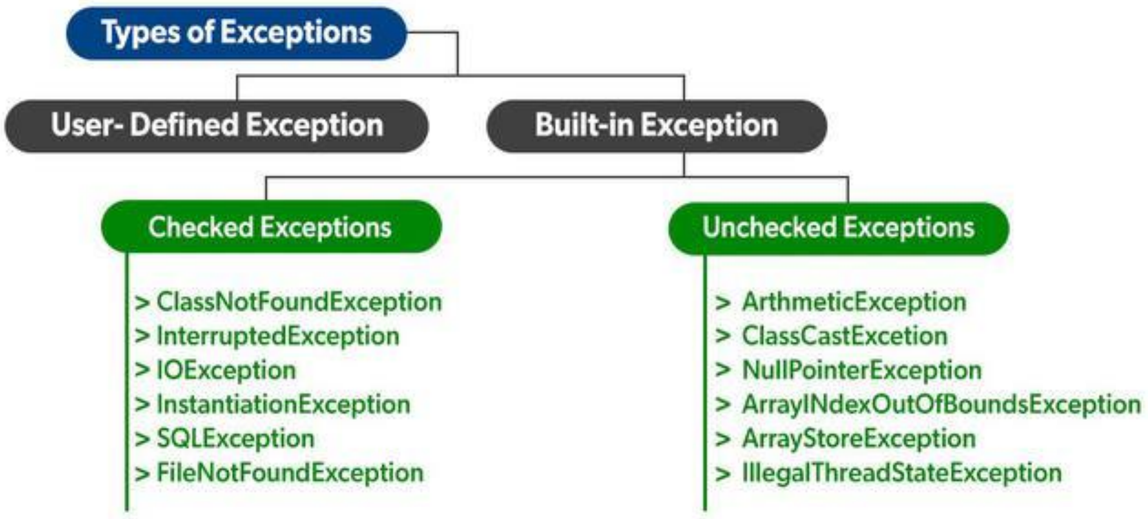
Exception Hierarchy

All exception and error types are subclasses of class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. Null Pointer Exception is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). Stack Over flow Error is an example of such an error.


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. **Built-in Exceptions**

- Checked Exception
- Unchecked Exception

2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

A. Built-in Exceptions:
Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Checked vs Unchecked Exceptions

B. User-Defined Exceptions:
Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

The ***advantages of Exception Handling in Java*** are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

Methods to print the Exception information:
1.printStackTrace()– This method prints exception information in the format of Name of the exception: description of the exception, stack trace.

- Java

```
//program to print the exception information using printStackTrace() method


import java.io.*;
```

```java
class GFG {
   public static void main (String[] args) {
     int a=5;
     int b=0;
       try{
         System.out.println(a/b);
       }
     catch(ArithmeticException e){
       e.printStackTrace();
     }
   }
}
```

Output:

2.toString() – This method prints exception information in the format of Name of the exception: description of the exception.

- Java

```java
//program to print the exception information using toString() method

import java.io.*;

class GFG1 {
   public static void main (String[] args) {
     int a=5;
     int b=0;
       try{
         System.out.println(a/b);
       }
```

```
    catch(ArithmeticException e){

      System.out.println(e.toString());

    }

  }

}
```

Output:

3.getMessage() -This method prints only the description of the exception.

- Java

```
//program to print the exception information using getMessage() method


import java.io.*;


class GFG1 {

  public static void main (String[] args) {

    int a=5;

    int b=0;

      try{

        System.out.println(a/b);

      }

    catch(ArithmeticException e){

      System.out.println(e.getMessage());

    }

  }

}
```

Output:

Defining and throwing exceptions

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. Exception Handling in Java is one of the powerful mechanisms to handle runtime errors so that the normal flow of the application can be maintained. We can use different ways to handle java exceptions. One of the ways is to use the java throw exception handling method. In this tutorial, we will learn about the **java throw exception** and the **throws** keyword to handle java exceptions.

First, we will discuss different types of java exceptions including checked and unchecked exceptions. We will also look at some of the built-in exceptions that are common and can occur frequently in the java program. At the same time, we will take various different types of java exceptions and will handle them using the `throws` keyword and java throw exception method.

To summarize, this tutorial will contain all the details and necessary examples that you need to know in order to start working and handling java exceptions using the java throw exception method.

Getting started with Java throw exception

The exception refers to some unexpected or contradictory situation or an error that is unexpected. There may be some situations that occur during program development. These are the situations where the code-fragment does not work right. There are different ways to handle the exceptions. For example, using try and catch block or using java throw exception method.

You can read more about the try and catch method from the article on try catch java. In this tutorial, we will learn about java throw exceptions, but first, let us have a look at some of the built-in java exceptions and different types of exceptions. Generally, java exceptions are divided into two different types including checked, and unchecked exceptions. Let us first differentiate between these two different types.

Checked exceptions in Java

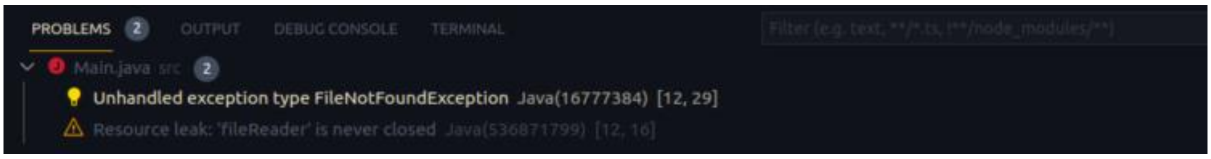
It is actually a compile-time exception that occurs when the java compiler checks or notifies during the compilation time. That is why they occur during the compile time. The compiler checks the checked exceptions during compilation to check whether the programmer has written the code to handle them or not.

We cannot simply ignore these exceptions and should handle them properly in order to run our program without any exceptions. If we will not write the code to handle them then there will be a compilation error that is why a method that throws a checked exception needs to either specify or handle it. Some of the common checked exceptions are `SQLException`, `ClassNotFoundException`, `FileNotFoundException`, `IOException`, etc.

For example, See the java program below which raise an exception because the file that we want to open does not exist.

```
// importing file

import java.io.File;

// importing filereader to read file

import java.io.FileReader;

// java main class

public class Main{

   // java main method

  public static void main(String args[]){

   //   creating new file typed object and opening the file

   File file = new File("file.txt");

   // reading file

   FileReader fileReader = new FileReader(file);

  }

}
```

Output:



```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL        Filter (e.g. text, **/*.ts, !**/node_modules/**)
v Main.java src 2
    Unhandled exception type FileNotFoundException  Java(16777384)  [12, 29]
    Resource leak: 'fileReader' is never closed  Java(536871799)  [12, 16]
```

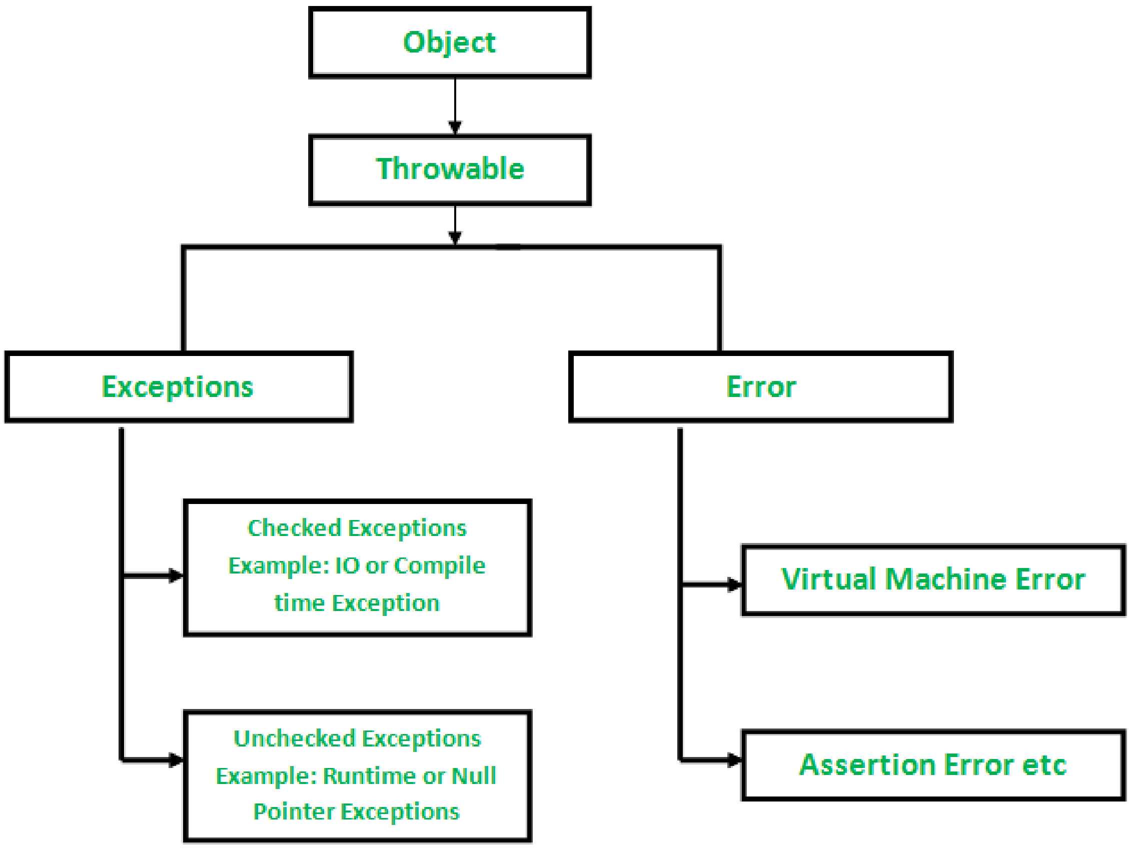
Notice that we get the exception that says File Not Found Exception, which is an example of a java checked exception.

Errors and runtime Exception

In java, both Errors and Exceptions are the subclasses of java. lang. Throwable class. Error refers to an illegal operation performed by the user which results in the abnormal working of the program. Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing. It is of three types:

- Compile-time
- Run-time
- Logical

Whereas exceptions in java refer to an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Now let us discuss various types of errors in order to get a better understanding over arrays. As discussed in the header an error indicates serious problems that a reasonable application should not try to catch. Errors are conditions that cannot get recovered by any handling techniques. It surely causes termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are Out of memory errors or System crash errors.

Example 1 Run-time Error

- Java

```
// Java Program to Illustrate Error

// Stack overflow error via infinite recursion


// Class 1

class StackOverflow {


    // method of this class

    public static void test(int i)

    {

        // No correct as base condition leads to

        // non-stop recursion.

        if (i == 0)

            return;

        else {

            test(i++);

        }

    }

}


// Class 2

// Main class

public class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        // Testing for error by passing
        // custom integer as an argument
        StackOverflow.test(5);
    }
}
```