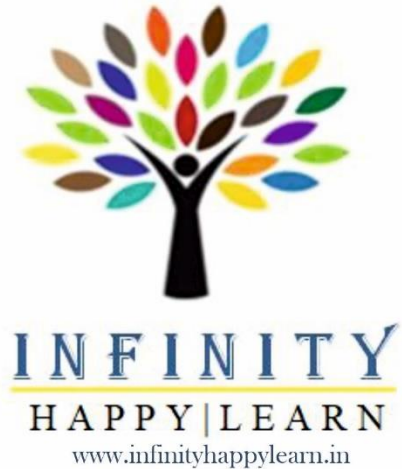


ISO 9001 : 2015 Certified



JAVA

Introduction

What is Java?

Java is a programming language and a platform. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak (tree).

Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java. Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

History of Java:

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc.

However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were “Simple, Robust (strong and healthy), Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral (run anywhere), Object-Oriented, Interpreted, and Dynamic” was developed by James Gosling, who is known as the father of Java, in 1995.

James Gosling and his team members started the project in the early ‘90s

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.

Java Version History

1. JDK Alpha and Beta (1995)
 2. JDK 1.0 (23rd Jan 1996)
 3. JDK 1.1 (19th Feb 1997)
 4. J2SE 1.2 (8th Dec 1998)
 5. J2SE 1.3 (8th May 2000)
 6. J2SE 1.4 (6th Feb 2002)
 7. J2SE 5.0 (30th Sep 2004)
 8. Java SE 6 (11th Dec 2006)
 9. Java SE 7 (28th July 2011)
 10. Java SE 8 (18th Mar 2014)
 11. Java SE 9 (21st Sep 2017)
 12. Java SE 10 (20th Mar 2018)
 13. Java SE 11 (September 2018)
 14. Java SE 12 (March 2019)
 15. Java SE 13 (September 2019)
 16. Java SE 14 (Mar 2020)
 17. Java SE 15 (September 2020)
 18. Java SE 16 (Mar 2021)
 19. Java SE 17 (September 2021)
 20. Java SE 18 (to be released by March 2022)
- 

Application

1. According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:
2. Desktop Applications such as acrobat reader, media player, antivirus, etc.
3. Web Applications such as irctc.co.in, Infinity.com, etc.
4. Enterprise Applications such as banking applications.
5. Mobile
6. Embedded System(microprocessor- or microcontroller-based system of hardware and software designed to perform dedicated functions within a larger mechanical or electrical system)
7. Smart Card(like ATM CARD)
8. Robotics

9. Games, etc

Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

- An application that runs on the server side and creates a dynamic page is called a web application. Currently,
- Spring. With its concept of Dependency Injection and aspect-oriented programming features, Spring took the development world by storm.
- Struts. Apache Struts is another robust open-source framework for web applications. ...
- Hibernate. ...
- Apache Wicket. ...
- JSF (Java Server Faces) ...
- Dropwizard. ...
- Grails. ...
- ATG
- Etc.,

technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, string,

Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition)

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, etc.

3) Java ME (Java Micro Edition)

It is a micro platform that is dedicated to mobile applications.

4) JavaFX

It is used to develop rich internet applications. It uses a lightweight user interface API.

Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java buzzwords.

A list of most important features of Java language is given below.

- Simple
- Object-Oriented
- Portable
- Platform independent
- Secured
- Robust
- Architecture neutral
- Interpreted
- High Performance
- Multithreaded
- Distributed
- Dynamic

Simple:

Java is very easy to learn, and its syntax is simple, clean and easy to understand.

According to Sun, Java language is a simple programming language because: Java syntax is based on C++ (so easier for programmers to learn it after C++). Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules. Basic concepts of OOPs are:

1. Object & Class
2. Encapsulation
3. Inheritance
4. Polymorphism
5. Abstraction

Platform Independent

Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.

A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

Runtime Environment 2.

API(Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode.

This bytecode is a platform-independent code because it can be run on multiple

platforms, i.e., Write Once and Run Anywhere (WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems.

Java is secured because:

1. No explicit pointer Java Programs run inside a virtual machine sandbox

Classloader: Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically.

It adds security by separating the package for the classes of the local file system from those that are imported from network sources.

2. **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
3. **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Robust

Robust simply means strong. Java is robust because:

It uses strong memory management.

There is a lack of pointers that avoids security problems.

There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.

There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code.

It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications.

This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads.

The main advantage of multi- threading is that it doesn't occupy memory for each thread. It shares a common memory area.

Threads are important for multi-media, Web applications, etc.

Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

C++ Vs Java

- Java is derived from C++.
- Java is compiled to bytecode (platform-independent). C++ is compiled to machine code.
- Java is object-oriented. C++ is both object-oriented and procedural.
- Java has automatic garbage collection. C++ does not.
- Java doesn't support operator overloading. C++ does.
- Java doesn't support structures and unions. C++ does.
- Java is slower than C++ on execution.
- Java is easier to use and simpler than C++.
- Java has a powerful cross-platform library. C++ libraries are robust but

simple.

Java Technology (JDK, JRE, JVM, JIT):

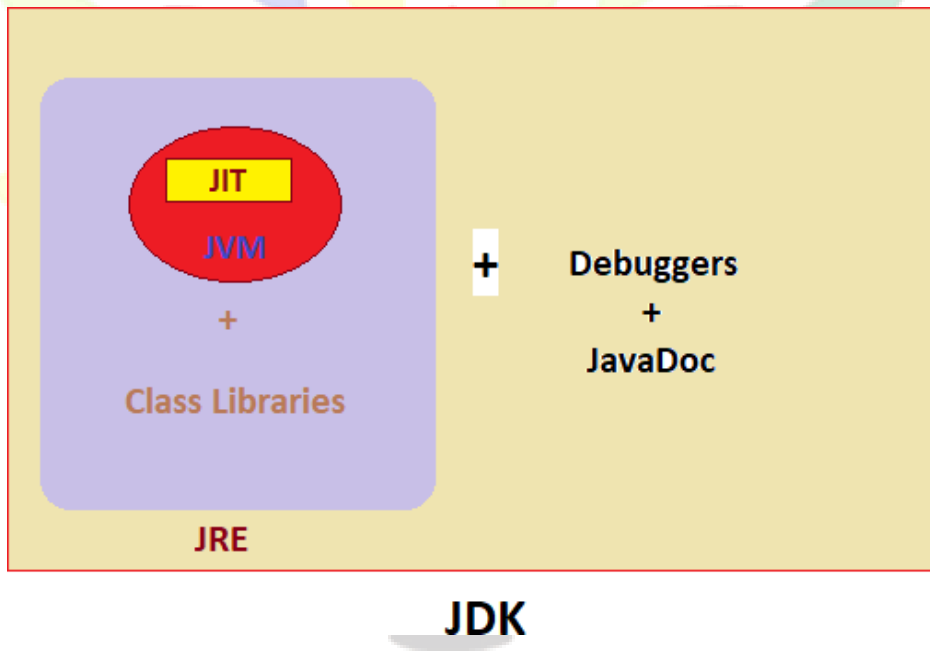
- Java Virtual Machine (JVM) is an abstract computing machine.
- Java Runtime Environment (JRE) is an implementation of the JVM.
- Java Development Kit (JDK) contains JRE along with various development tools like Java libraries, Java source compilers, Java debuggers, bundling and deployment tools.
- Just In Time compiler (JIT) is runs after the program has started executing, on the fly.

JIT is part of JVM, whereas JVM is part of JRE, and JRE is part of JDK.

JVM = Interpreter + JIT

JRE = JVM + Library classes

JDK = JRE + Development Tools

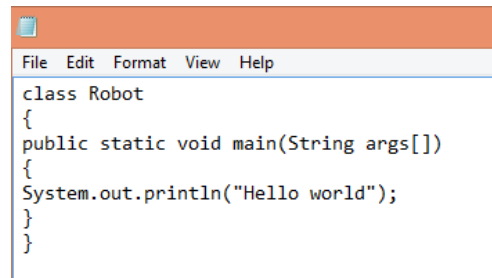


How to install & Set path:

1. Install jdk and then run cmd => java -version.
2. Path set env

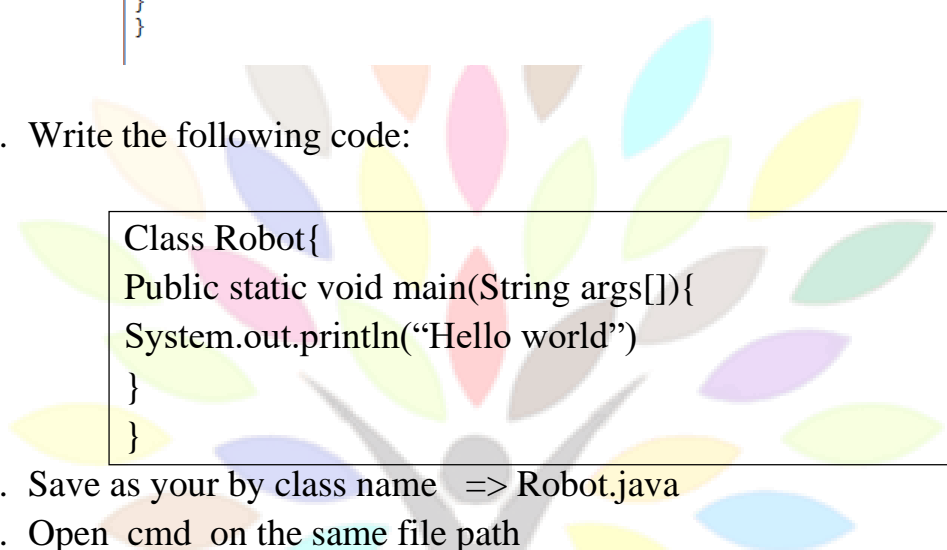
C:\Program Files\Java\jdk-18\bin

3. To start code in java => open notepad



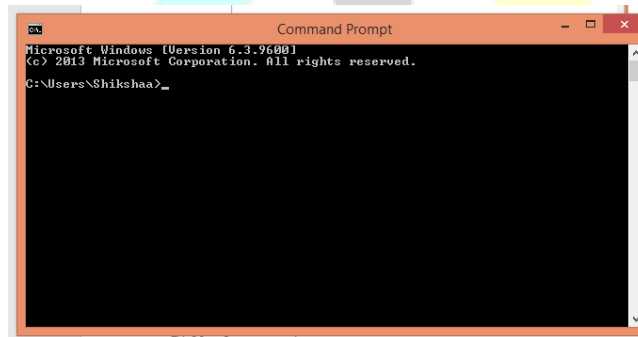
```
File Edit Format View Help
class Robot
{
public static void main(String args[])
{
System.out.println("Hello world");
}
}
```

4. Write the following code:



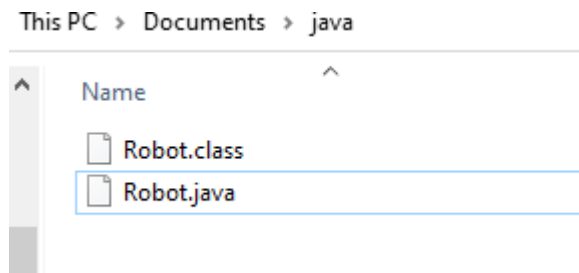
```
Class Robot{
Public static void main(String args[]){
System.out.println("Hello world")
}
}
```

5. Save as your by class name => Robot.java
6. Open cmd on the same file path



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.
C:\Users\Shiksha>
```

7. Run the cmd => javac Robot.java



And

You will get new file that named Robot.class

To get the result of the run cmd => java Robot

You will get the following result

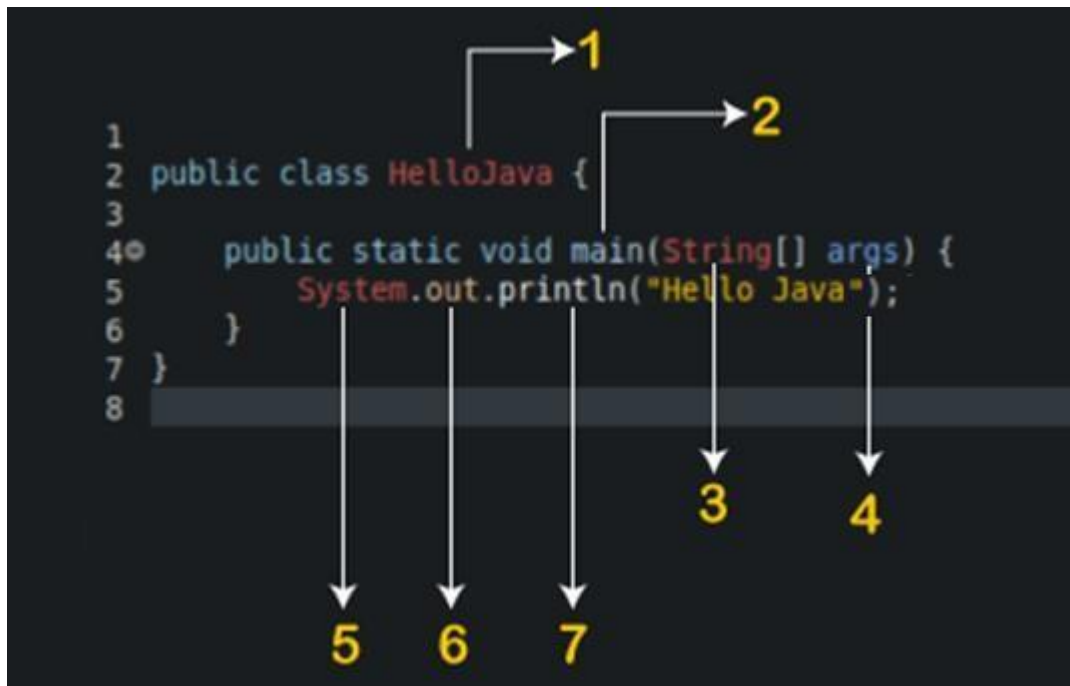
“Hello World“

A simple program:

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

Identifiers:

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



From the above example, we have the following Java identifiers:

- 1) HelloJava (Class name)
- 2) main (main method)
- 3) String (Predefined Class name)
- 4) args (String variables)
- 5) System (Predefined class)
- 6) out (Variable name)
- 7) println (method)

Rules of identifiers:

- A valid identifier must have characters [A-Z] or [a-z] or numbers [0-9], and underscore (_) or a dollar sign (\$). For example, @Infinity is not a valid identifier because it contains a special character which is @.
- There should not be any space in an identifier. For example, java tpoint is an invalid identifier.
- An identifier should not contain a number at the starting. For example, 123Infinity is an invalid identifier.
- An identifier should be of length 4-15 letters only. However, there is no

limit on its length. But, it is good to follow the standard conventions.

- We can't use the Java reserved keywords as an identifier such as int, float, double, char, etc. For example, int double is an invalid identifier in Java.
- An identifier should not be any query language keywords such as SELECT, FROM, COUNT, DELETE, etc.

Keywords:

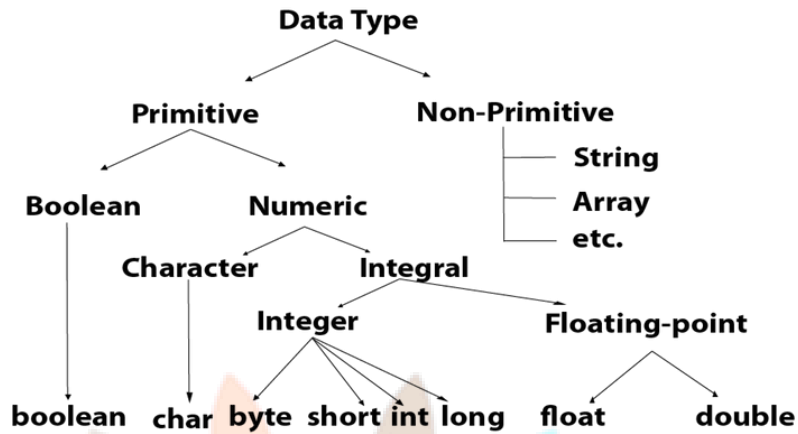
abstract	Continue	for	protected	transient
Assert	Default	Goto	public	Try
Boolean	Do	If	Static	throws
break	Double	implements	strictfp	Package
byte	Else	import	super	Private
case	Enum	Interface	Short	switch
Catch	Extends	instanceof	return	void
Char	Final	Int	synchronized	volatile
class	Finally	long	throw	Date
const	Float	Native	This	while

Data Types:

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

- 1) **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
- 2) **Non-primitive data types:** The non-primitive data types include Classes Interfaces and array

JAVA



DATA TYPE	SIZE	DESCRIPTION
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

JAVA

Operators in Java

Operators is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	expr++ expr--
	prefix	++expr --expr +expr -expr ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	
Logical	logical AND	&&
	logical OR	
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

JAVA

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator

Example: ++ and --

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int x=10;
        System.out.println(x++); //10 (11)
        System.out.println(++x); //12
        System.out.println(x--); //12 (11)
        System.out.println(--x); //10
    }
}
```

Output:

10
12
12
10

Java Unary Operator Example 2: ++ and –

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


JAVA

```
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21
}
}
```

Output:

22

21

Java Unary Operator Example: ~ and !

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int
        a=10;
        int b=-
        10;
        boolean c=true;
        boolean d=false;
        System.out.println(~a
        );
        System.out.println(~b
        );
    }
}
```

Output:

-11

9

false

true

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical

JAVA

operations.

Java Arithmetic Operator Example

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int a=10;
        int b=5;
        System.out.println(a+b);//15
        System.out.println(a-b);//5
        System.out.println(a*b);//50
        System.out.println(a/b);//2
        System.out.println(a%b);//0
    }
}
```

Output:

15
5
50
2
0

Java Arithmetic Operator Example: Expression

```
public class OperatorExample{
    public static void main(String args[]){
        System.out.println(10*10/5+3-1*4/2);
    }
}
```

Output:

21

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

JAVA

```
public class OperatorExample
{
    public static void main(String args[])
    {
        System.out.println(10<<2);//10*2^2=10*4=40
        System.out.println(10<<3);//10*2^3=10*8=80
        System.out.println(20<<2);//20*2^2=20*4=80
        System.out.println(15<<4);//15*2^4=15*16=240
    }
}
```

Output:

40
80
80
240

Java Right Shift Operator

The Java right shift operator `>>` is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
public OperatorExample{
    public static void main(String args[]){
        System.out.println(10>>2);//10/2^2=10/4=2
        System.out.println(20>>2);//20/2^2=20/4=5
        System.out.println(20>>3);//20/2^3=20/8=2
    }}
}
```

Output:

2
5
2

Java Shift Operator Example: `>>` vs `>>>`

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

JAVA

```
{
    //For positive number, >> and >>> works same
    System.out.println(20>>2);
    System.out.println(20>>>2);
    //For negative number, >>> changes parity bit (MSB) to 0
    System.out.println(-20>>2);
    System.out.println(-20>>>2);
}
}
```

Output:

5
5
-5
1073741819

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int a=10;
        int b=5;
        int c=20;
        System.out.println(a<b&&a<c);    //false && true = false
        System.out.println(a<b&a<c);    //false & true = false
    }
}
```

Output:

false
false

JAVA

Java AND Operator

Example: Logical && vs Bitwise &

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int a=10;
        int b=5;
        int c=20;
        System.out.println(a<b&&a++<c);//false && true = false
        System.out.println(a);//10 because second condition is not checked
        System.out.println(a<b&a++<c);//false && true = false
        System.out.println(a);//11 because second condition is checked
    }
}
```

Output:

false
10
false
11

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false. The bitwise | operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int a=10;
        int b=5;
        int c=20;
```

JAVA

```
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}
}
```

Output:

```
true
true
true
10
true
11
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int a=2;
        int b=5;
        int min=(a<b)?a:b;
        System.out.println(min);
    }
}
```

Output:

```
2
```

JAVA

Another Example:

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int a=10;
        int b=5;
        int min=(a<b)?a:b;
        System.out.println(min);
    }
}
```

Output:

5

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

```
public class OperatorExample
{
    public static void main(String args[])
    {
        int a=10;
        int b=20;
        a+=4;//a=a+4 (a=10+4)
        b-=4;//b=b-4 (b=20-4)
        System.out.println(a);
        System.out.println(b);
    }
}
```

Output:

14

16

JAVA

Java Assignment Operator Example

```
public class OperatorExample
{
    public static void main(String[] args)
    {
        int a=10;
        a+=3;//10+3
        System.out.println(a);
        a-=4;//13-4
        System.out.println(a);
        a*=2;//9*2
        System.out.println(a);
        a/=2;//18/2
        System.out.println(a);
    }
}
```

Output:

13
9
18
9

Java Assignment Operator Example: Adding short

```
public class OperatorExample
{
    public static void main(String args[])
    {
        short a=10;
        short b=10;
        //a+=b;//a=a+b internally so fine
        a=a+b;//Compile time error because 10+10=20 now int
        System.out.println(a);
    }
}
```

JAVA

```
}
```

Output:

Compile time error

After type cast:

```
public class OperatorExample
{
public static void main(String args[])
{
short a=10;
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}
}
```

Output:

20

Command-line argument:

The java command-line argument is an argument i.e. passed at the time of running the java program. The arguments passed from the console can be received in the java program and it can be used as an input.

Simple example of command-line argument in java:

```
class CommandLineExample
{
public static void main(String args[])
{
System.out.println("Your first argument is: "+args[0]);
}
}
```

Output:

compile by > javac CommandLineExample.java

run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo

JAVA

Control Structure

If statement is used to test the condition. It checks Boolean condition: true or false. There are various types of if statement in Java.

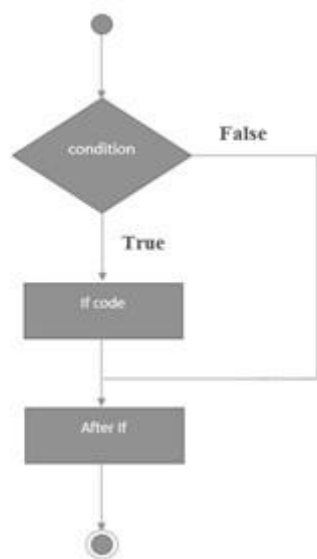
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

Java if Statement:

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```
if(condition){  
  //code to be executed  
}
```



Example:

```
//Java Program to demonstate the use of if statement.
```

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

JAVA

```
//defining an 'age' variable
int age=20;
//checking the age
if(age>18){
System.out.print("Age is greater than 18"); 8.    }
}
}
```

Output:

Age is greater than 18

Java if-else Statement

The Java if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

Syntax:

```
if(condition){
    //code if condition is true
}else{
    //code if condition is false}
```

Example:

```
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample
{
public static void main(String[] args)
{
//defining a variable
int number=13;
//Check if the number is divisible by 2 or not
if(number%2==0)
{
System.out.println("even number");
}
else
```

JAVA

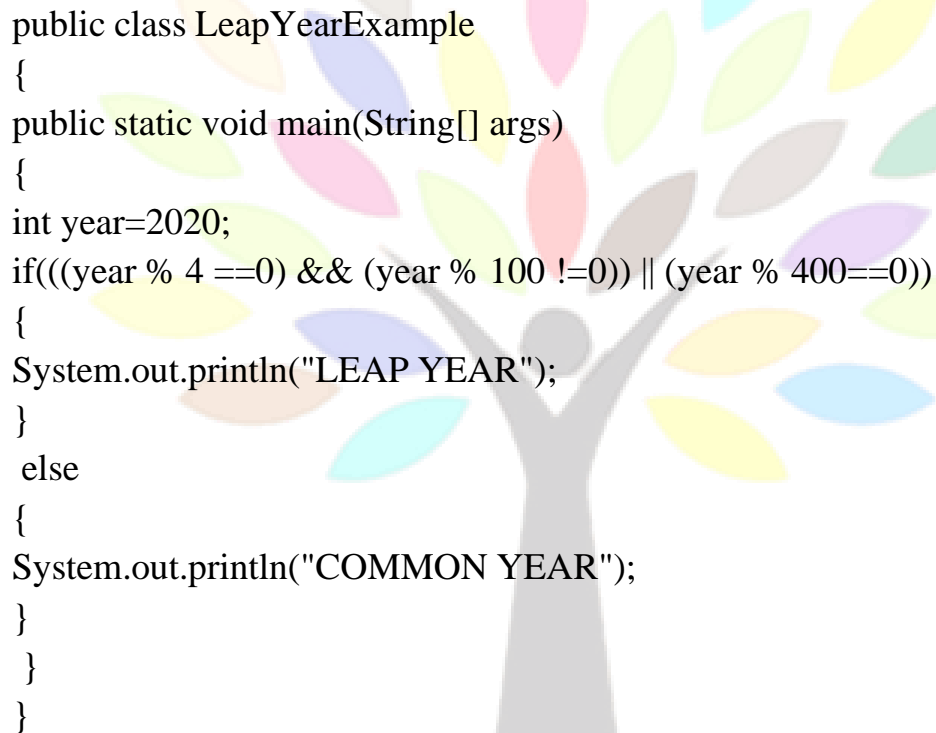
```
{  
System.out.println("odd number");  
}  
}  
}
```

Output:

odd number

Leap Year Example:

A year is leap, if it is divisible by 4 and 400. But, not by 100.



```
public class LeapYearExample  
{  
public static void main(String[] args)  
{  
int year=2020;  
if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0))  
{  
System.out.println("LEAP YEAR");  
}  
else  
{  
System.out.println("COMMON YEAR");  
}  
}  
}
```

Output:

LEAP YEAR

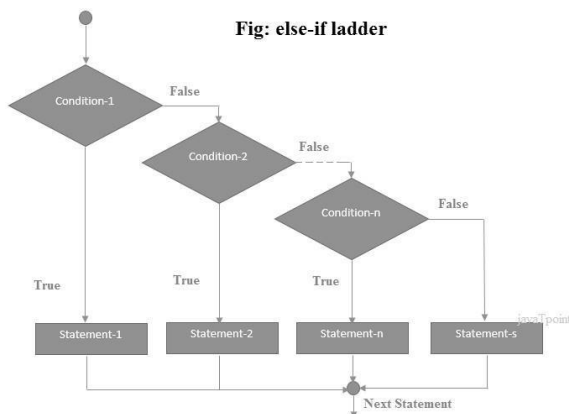
Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

JAVA

```
if(condition1){  
    //code to be executed if condition1 is true  
}else if(condition2){  
    //code to be executed if condition2 is true  
}  
else if(condition3){  
    //code to be executed if condition3 is true  
}  
...  
else{  
    //code to be executed if all the conditions are false  
}
```

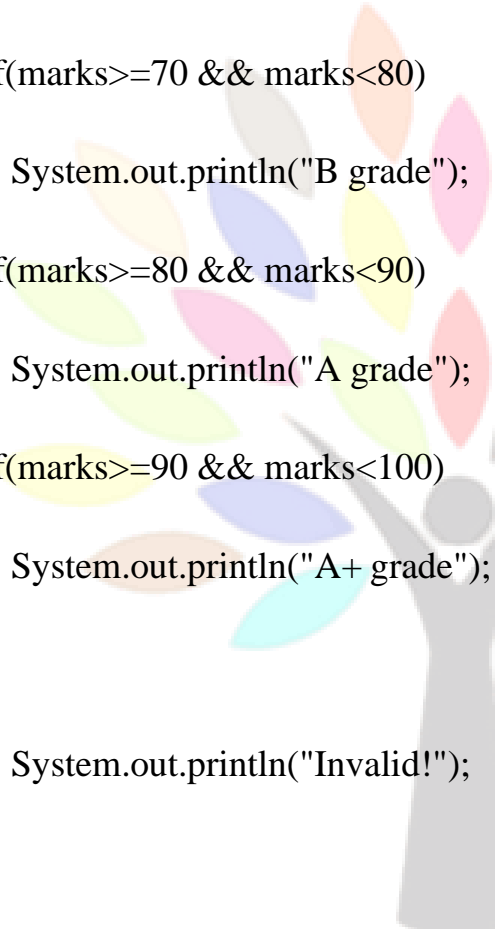


Example:

```
//Java Program to demonstrate the use of If else-if ladder.  
//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.  
public class IfElseIfExample  
{  
    public static void main(String[] args)  
    {  
        int marks=65; 5.  
        if(marks<50)  
        {
```

JAVA

```
        System.out.println("fail"); 8.
    }
    else if(marks>=50 && marks<60)
    {
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70)
    {
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80)
    {
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90)
    {
        System.out.println("A grade");
    }
    else if(marks>=90 && marks<100)
    {
        System.out.println("A+ grade");
    }
    else
    {
        System.out.println("Invalid!");
    }
}
```



Output: C grade

Java Nested if statement:

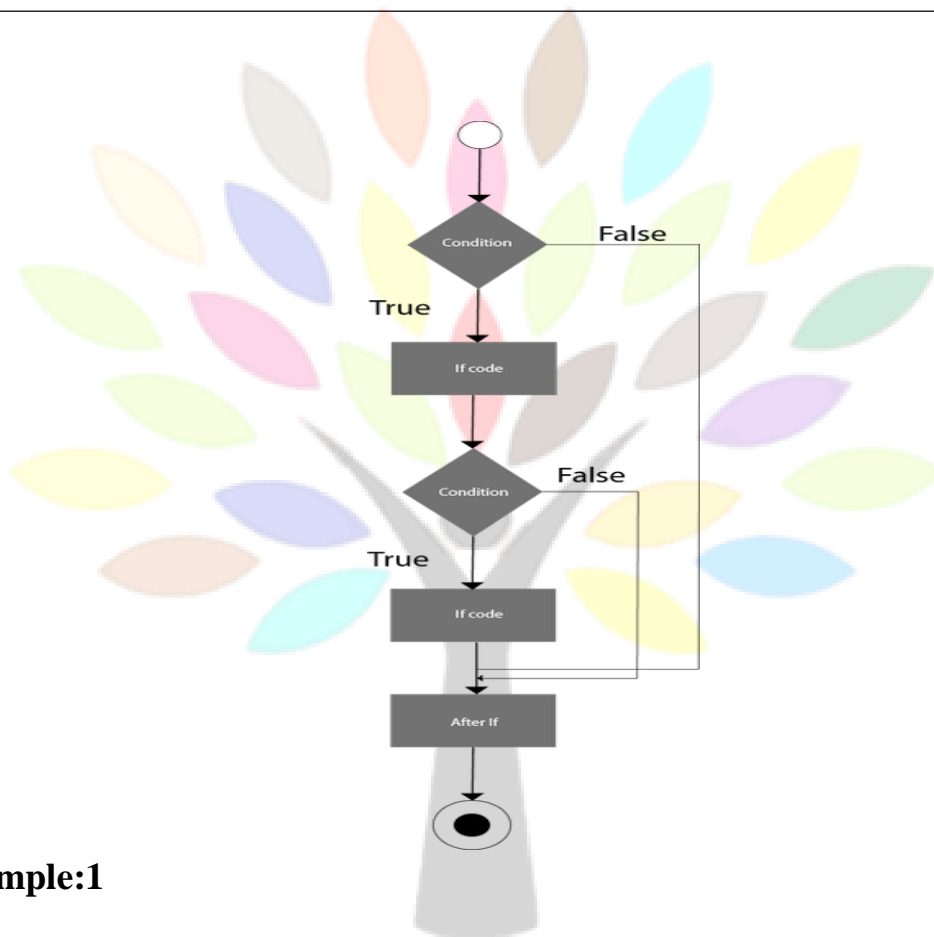
The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

JAVA

Syntax:

if(condition)

```
{  
    //code to be executed  
    if(condition)  
    {  
        //code to be executed  
    }  
}
```



Example:1

```
//Java Program to demonstrate the use of Nested If Statement. public class  
JavaNestedIfExample  
{  
    public static void main(String[] args)  
    {  
        //Creating two variables for age and weight int age=20;
```

JAVA

```
int weight=80
//applying condition on age and weight
if(age>=18)
{
if(weight>50)
{
System.out.println("You are eligible to donate blood");
}
}
}
}
```

Output:

You are eligible to donate blood

Example 2:

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample2
{
public static void main(String[] args)
{
//Creating two variables for age and weight
int age=25;
int weight=48;
//applying condition on age and weight 8.    if(age>=18){
if(weight>50)
{
    System.out.println("You are eligible to donate blood");
} else{
    System.out.println("You are not eligible to donate blood");
}
} else{
    System.out.println("Age must be greater than 18");
}
}
```

JAVA

```
}
```

Output:

You are not eligible to donate blood

Java Switch Statement

The Java switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement. There can be one or N number of case values for a switch expression. The case value must be of switch expression type only. The case value must be literal or constant. It doesn't allow variables.

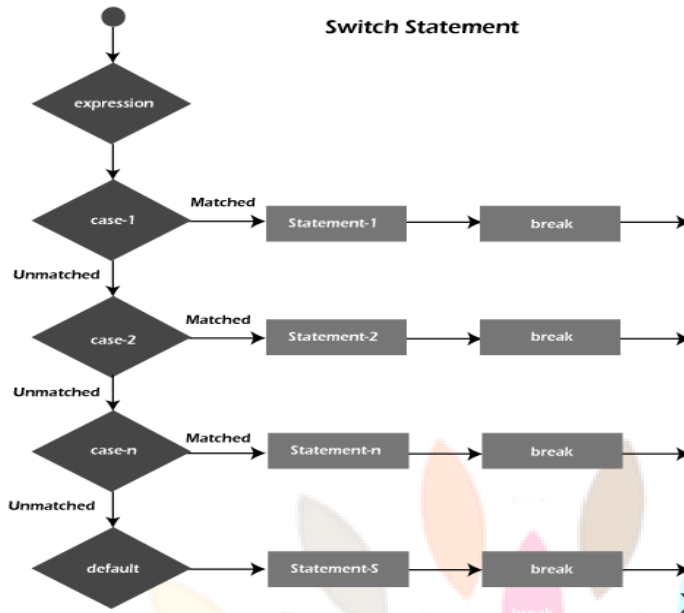
The case values must be unique. In case of duplicate value, it renders compile-time error. The Java switch expression must be of byte, short, int, long (with its Wrapper type), enums and string.

Each case statement can have a break statement which is optional. When control reaches to the statement it jumps the control after the switch expression. If a break statement is not found, it executes the next case. The case value can have a default label which is optional.

Syntax:

```
switch(expression)
{
case value1:
    //code to be executed;
break; //optional
case value2:
    //code to be executed;
break; //optional.....
default:
code to be executed if all cases are not matched;
}
```

JAVA



Example:

SwitchExample.java

```
public class SwitchExample
{
    public static void main(String[] args)
    {
        //Declaring a variable for switch expression
        int number=20;
        //Switch expression
        switch(number){
            //Case statements
            case 10: System.out.println("10");
            break;
            case 20: System.out.println("20");
            break;
            case 30: System.out.println("30");
            break;
            //Default case statement
            default:
                System.out.println("Not in 10, 20 or 30");
        }
    }
}
```

JAVA

```
    }  
}  
}
```

Output:

20

Finding Month Example:

SwitchMonthExample.java

```
//Java Program to demonstrate the example of Switch statement  
  
//where we are printing month name for the given number  
public class SwitchMonthExample  
{  
    public static void main(String[] args)  
    {  
        //Specifying month number  
        int month=7;  
        String monthString="";  
        //Switch statement  
        switch(month){  
            //case statements within the switch block  
            case 1: monthString="1 - January";  
                break;  
            case 2: monthString="2 - February";  
                break;  
            case 3: monthString="3 - March";  
                break;  
            case 4: monthString="4 - April";  
                break;  
            case 5: monthString="5 - May";  
                break;  
            case 6: monthString="6 - June";  
                break;
```

JAVA

```
case 7: monthString="7 - July";  
break;  
case 8: monthString="8 - August";  
break;  
case 9: monthString="9 - September";  
break;  
case 10: monthString="10 - October";  
break;  
case 11: monthString="11 - November";  
break;  
case 12: monthString="12 - December";  
break;  
default:  
System.out.println("Invalid Month!");  
}  
//Printing month of the given number  
System.out.println(monthString);  
}  
}
```

Output:

7 – July

Program to check Vowel or Consonant:

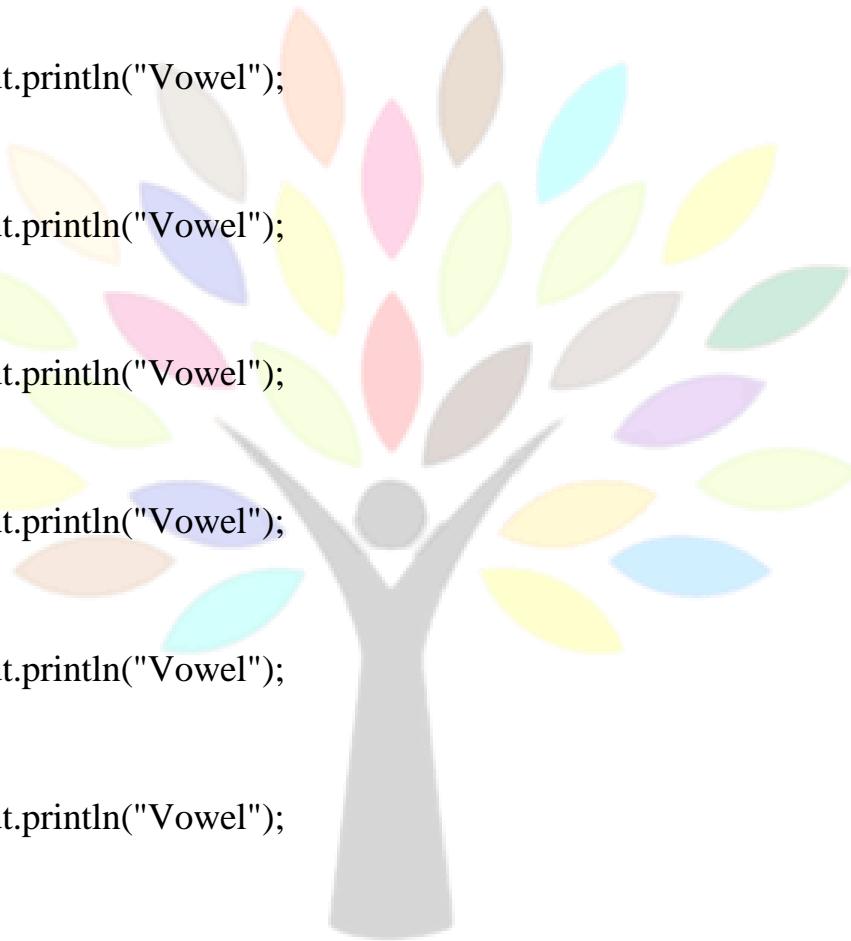
If the character is A, E, I, O, or U, it is vowel otherwise consonant. It is not case-sensitive.

SwitchVowelExample.java

```
public class SwitchVowelExample  
{  
    public static void main(String[] args)  
    {  
        char ch='O';  
        switch(ch){
```

JAVA

```
case 'a':
System.out.println("Vowel");
break;
case 'e':
System.out.println("Vowel");
break;
case 'i':
System.out.println("Vowel");
break;
case 'o':
System.out.println("Vowel");
break;
case 'u':
System.out.println("Vowel");
break;
case 'A':
System.out.println("Vowel");
break;
case 'E':
System.out.println("Vowel");
break;
case 'I':
System.out.println("Vowel");
break;
case 'O':
System.out.println("Vowel");
break;
case 'U':
System.out.println("Vowel");
break;
default:
System.out.println("Consonant");
}
}
}
```



JAVA

Output:

Vowel

Java Switch Statement is fall-through

The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

Example:

SwitchExample2.java

```
//Java Switch Example where we are omitting the
//break statement
public class SwitchExample2 {
public static void main(String[] args) {
int number=20;
//switch expression with int value
switch(number){
//switch cases without break statements
case 10: System.out.println("10");
case 20: System.out.println("20");
case 30: System.out.println("30");
default: System.out.println("Not in 10, 20 or 30");
}
}
}
```

Output:

20

30

Not in 10, 20 or 30

Java Switch Statement with String

Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

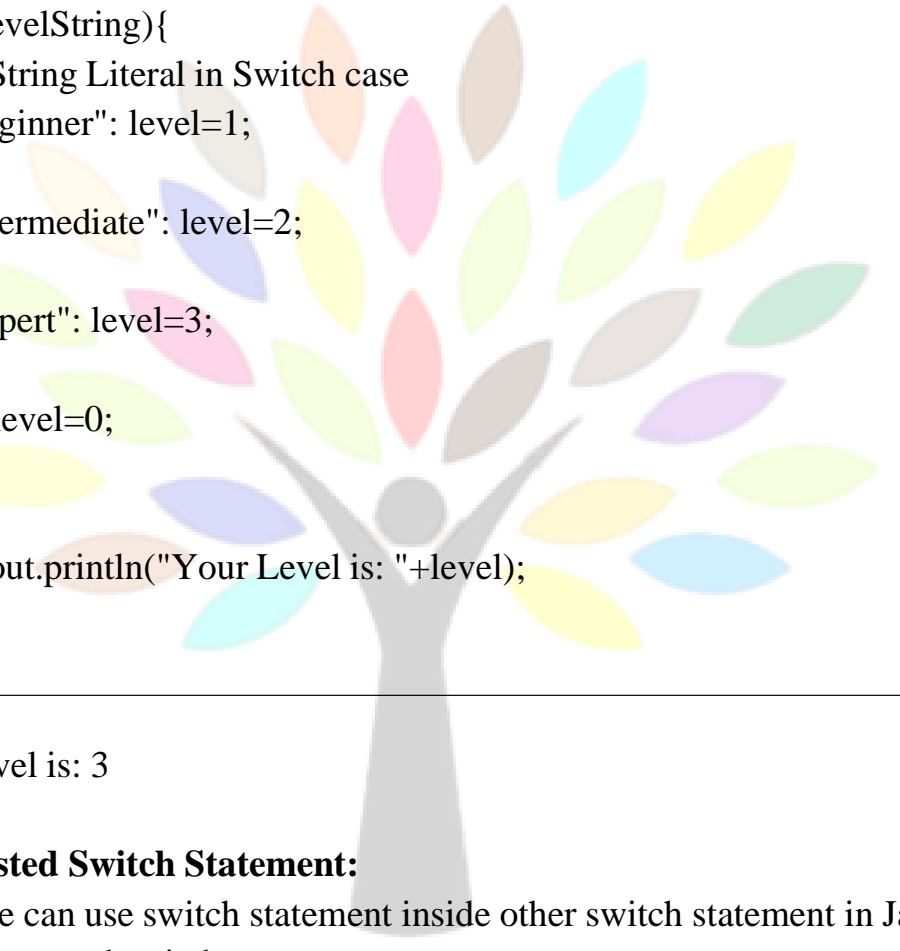
Example:

SwitchStringExample.java

```
//Java Program to demonstrate the use of Java Switch
```

JAVA

```
//statement with String
public class SwitchStringExample
{
    public static void main(String[] args)
    {
        //Declaring String variable
        String levelString="Expert";
        int level=0;
        //Using String in Switch expression
        switch(levelString){
            //Using String Literal in Switch case
            case "Beginner": level=1;
            break;
            case "Intermediate": level=2;
            break;
            case "Expert": level=3;
            break;
            default: level=0;
            break;
        }
        System.out.println("Your Level is: "+level);
    }
}
```



Output:

Your Level is: 3

Java Nested Switch Statement:

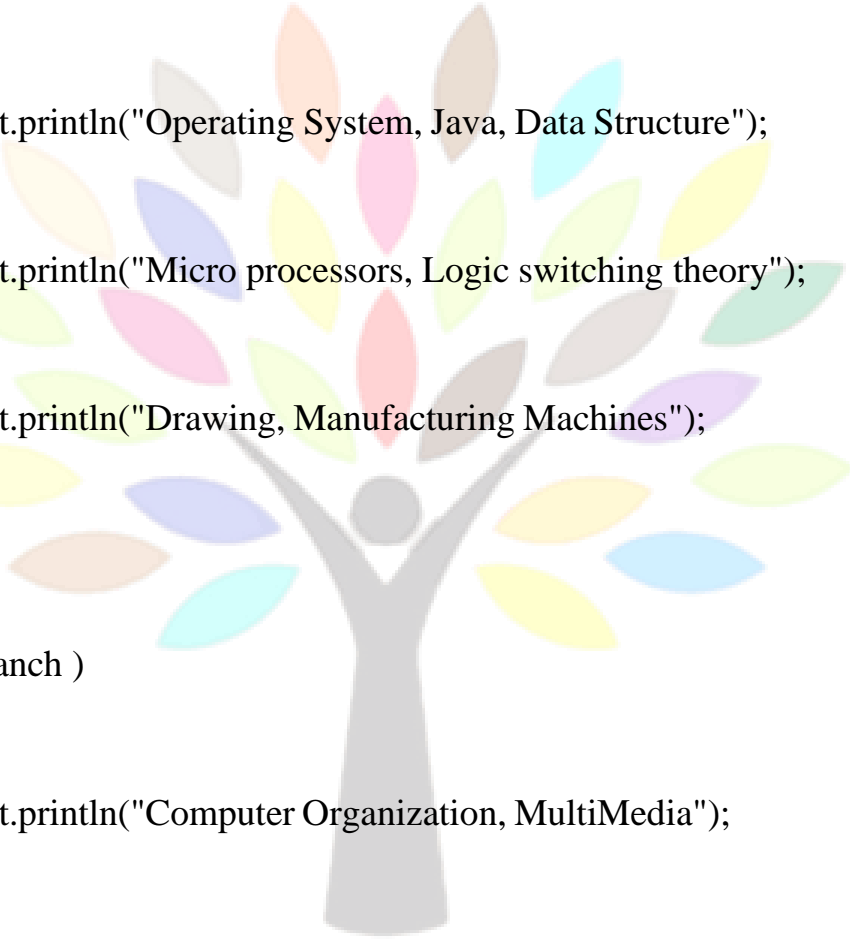
We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

NestedSwitchExample.java

```
//Java Program to demonstrate the use of Java Nested Switch public class
NestedSwitchExample {
    public static void main(String args[])
    {
```

JAVA

```
//C - CSE, E - ECE, M - Mechanical char branch = 'C';
int collegeYear = 4;
switch( collegeYear )
{
case 1:
System.out.println("English, Maths, Science");
break;
case 2:
switch( branch )
{
case 'C':
System.out.println("Operating System, Java, Data Structure");
break;
case 'E':
System.out.println("Micro processors, Logic switching theory");
break;
case 'M':
System.out.println("Drawing, Manufacturing Machines");
break;
}
break;
case 3:
switch( branch )
{
case 'C':
System.out.println("Computer Organization, MultiMedia");
break;
case 'E':
System.out.println("Fundamentals of Logic Design, Microelectronics");
break;
case 'M':
System.out.println("Internal Combustion Engines, Mechanical Vibration");
break;
}
break;
```



JAVA

```
case 4:
switch( branch )
{
case 'C':
System.out.println("Data Communication and Networks, MultiMedia");
break;
case 'E':
System.out.println("Embedded System, Image Processing");
break;
case 'M':
System.out.println("Production Technology, Thermal Engineering");
break;
}
break;
}
}
```

Output:

Data Communication and Networks, MultiMedia.

Looping Statement in java

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true.

There are three types of loops in java.

- for loop
- while loop
- do-while loop

For loop:

The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

Syntax:

```
for(initialization; condition; increment/decrement)
```

JAVA

```
{  
    //statement or code to be executed  
}
```

- **Initialization:** It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
- **Condition:** It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
- **Increment/Decrement:** It increments or decrements the variable value. It is an optional condition. Statement: The statement of the loop is executed each time until the second condition is false.

ForExample.java

Java Program to demonstrate the example of for loop. Which prints table of 1

```
public class ForExample {
```

Example:

ForExample.java

```
public class ForExample  
{  
    public static void main(String[] args)  
    {  
        for(int i=1;i<=10;i++)  
        {  
            System.out.println(i);  
        }  
    }  
}
```

Output:

1
2
3
4
5
6

JAVA

7
8
9
10

Java Nested for Loop:

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

NestedForExample.java

```
public class NestedForExample
{
    public static void main(String[] args)
    {
        //loop of i
        for(int i=1;i<=3;i++){
            //loop of j
            for(int j=1;j<=3;j++){ System.out.println(i+" "+j);
            }//end of i
        }//end of j
    }
}
```

Output:

1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3

Java for-each Loop:

The for-each loop is used to traverse array or collection in Java. It is easier

JAVA

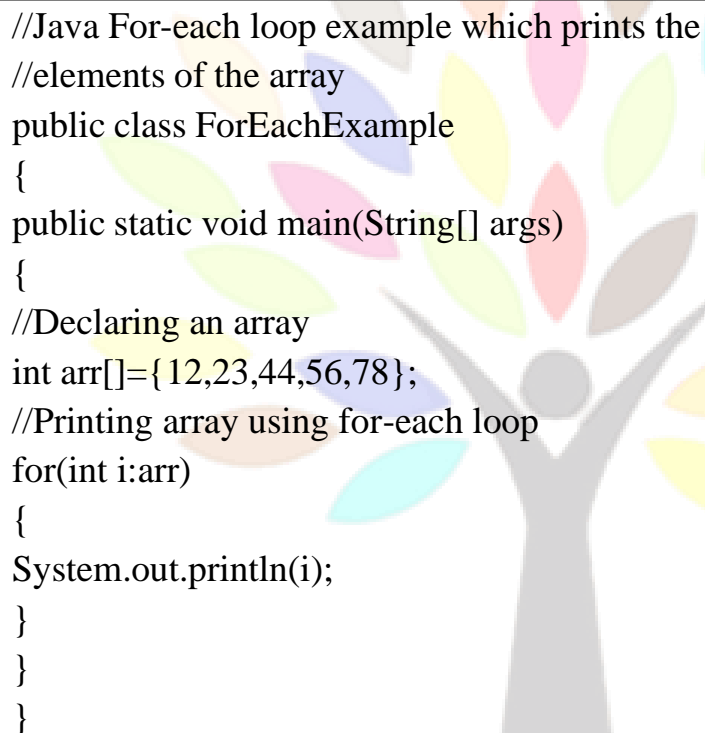
to use than simple for loop because we don't need to increment value and use subscript notation. It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Syntax:

```
for(data_type variable : array_name)
{
//code to be executed
}
```

Example: ForEachExample.java

Output:



```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample
{
public static void main(String[] args)
{
//Declaring an array
int arr[]={ 12,23,44,56,78};
//Printing array using for-each loop
for(int i:arr)
{
System.out.println(i);
}
}
}
```

12

23

44

56

78

Java Infinitive for Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

JAVA

Syntax:

```
for(;;){  
    //code to be executed  
}
```

ForExample.java

```
//Java program to demonstrate the use of infinite for loop  
//which prints an statement  
public class ForExample  
{  
    public static void main(String[] args)  
    {  
        for(;;)  
        {  
            System.out.println("infinite loop");  
        }  
    }  
}
```

Output:

```
infinite loop  
infinite loop  
infinite loop  
ctrl+c
```

Java While Loop:

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

```
while (condition)  
{  
    //code to be executed & increment / decrement statement  
}
```

WhileExample.java

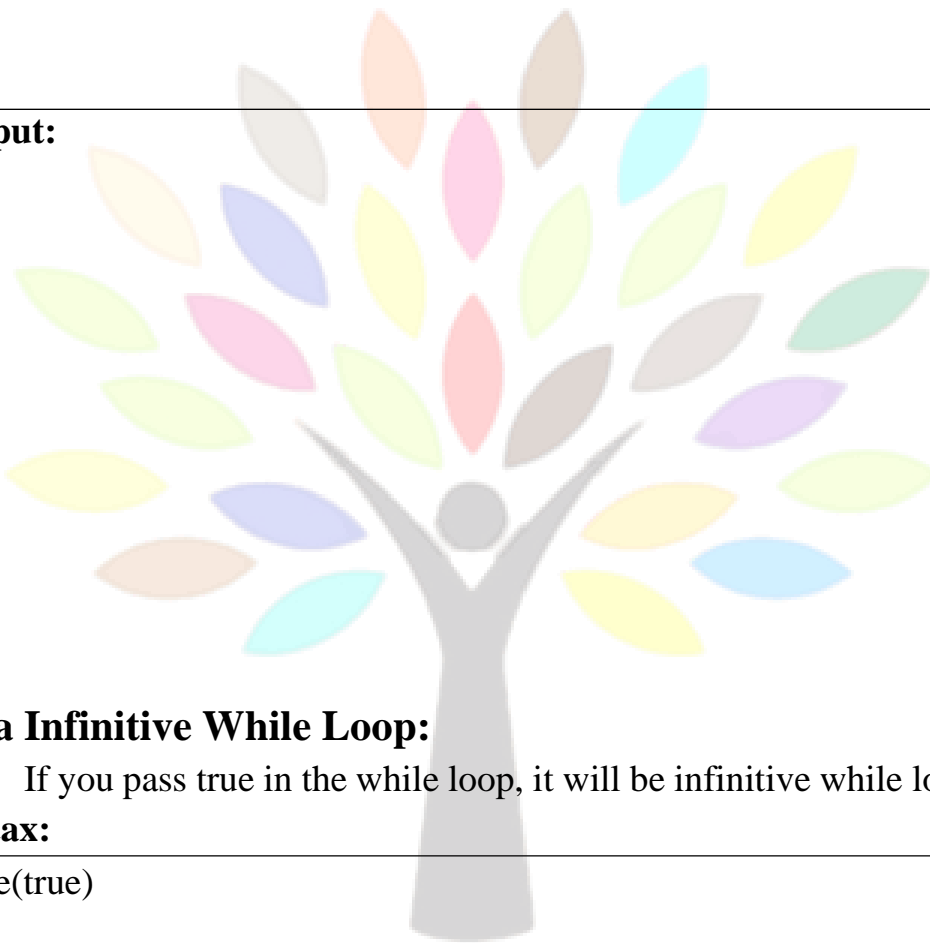
```
public class WhileExample
```


JAVA

```
{  
public static void main(String[] args)  
{  
    int i=1;  
        while(i<=10)  
        {  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

Output:

1
2
3
4
5
6
7
8
9
10



Java Infinitive While Loop:

If you pass true in the while loop, it will be infinitive while loop.

Syntax:

```
while(true)  
{  
    //code to be executed  
}
```

WhileExample2.java

```
public class WhileExample2  
{  
    public static void main(String[] args)
```

JAVA

```
{  
//setting the infinite while loop by passing true to the condition  
while(true)  
{  
System.out.println("infinitive while loop");  
}  
}  
}
```

Output:

infinitive while loop
infinitive while loop
ctrl+c

Java do-while Loop:

The Java do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java do-while loop is executed at least once because condition is checked after loop body.

Syntax:

```
do{  
//code to be executed / loop body  
//update statement  
}while (condition);
```

DoWhileExample.java

```
public class DoWhileExample  
{  
public static void main(String[] args)  
{  
int i=1;  
do  
{  
System.out.println(i);  
i++;  
}}
```

JAVA

```
} while(i<=10);  
}  
}
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Java Infinitive do-while Loop:

If you pass true in the do-while loop, it will be infinitive do-while loop.

Syntax:

```
do{  
  //code to be executed  
}while(true);
```

Example:

```
public class DoWhileExample2  
{  
  public static void main(String[] args)  
  {  
    do{  
      System.out.println("infinitive do while loop");  
    }while(true);  
  }  
}
```

Output:

```
infinitive do while loop  
infinitive do while loop
```

JAVA

infinite do while loop

ctrl+c

In the above code, we need to enter Ctrl + C command to terminate the infinite loop.

Break statement:

The break statement in Java programming language has the following two usages – when the break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. It can be used to terminate a case in the switch statement

Syntax:

```
jump-statement;  
break;
```

Example:

BreakExample.java

```
//Java Program to demonstrate the use of break statement  
//inside the for loop.  
public class BreakExample  
{  
    public static void main(String[] args)  
    {  
        //using for loop  
        for(int i=1;i<=10;i++)  
        {  
            if(i==5){  
                //breaking the loop  
                break;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

Output:

1

JAVA

2
3
4

Java Continue Statement:

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

The Java continue statement is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
jump-statement;  
continue;
```

ContinueExample.java

```
//Java Program to demonstrate the use of continue statement  
//inside the for loop.  
public class ContinueExample  
{  
    public static void main(String[] args)  
    {  
        //for loop  
        for(int i=1;i<=10;i++)  
        {  
            if(i==5)  
            {  
                //using continue statement  
                continue;//it will skip the rest statement  
            }  
            System.out.println(i);  
        }  
    }  
}
```

JAVA

```
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

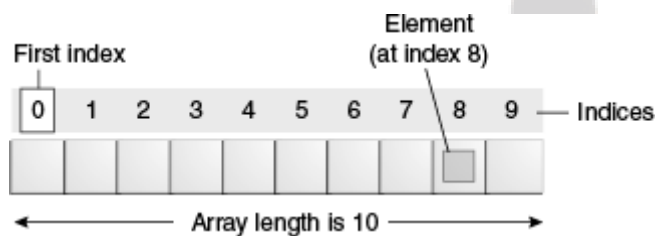
As you can see in the above output, 5 is not printed on the console. It is because the loop is continued when it reaches to 5.

Java Array

Normally, an array is a collection of similar type of elements that have a contiguous memory location.

Java array is an object which contains elements of a similar data type. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array. To declare an array, define the variable type with square brackets:

Array in java is index-based, the first element of the array is stored at the 0 index.



Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

JAVA

Single Dimensional Array in Java

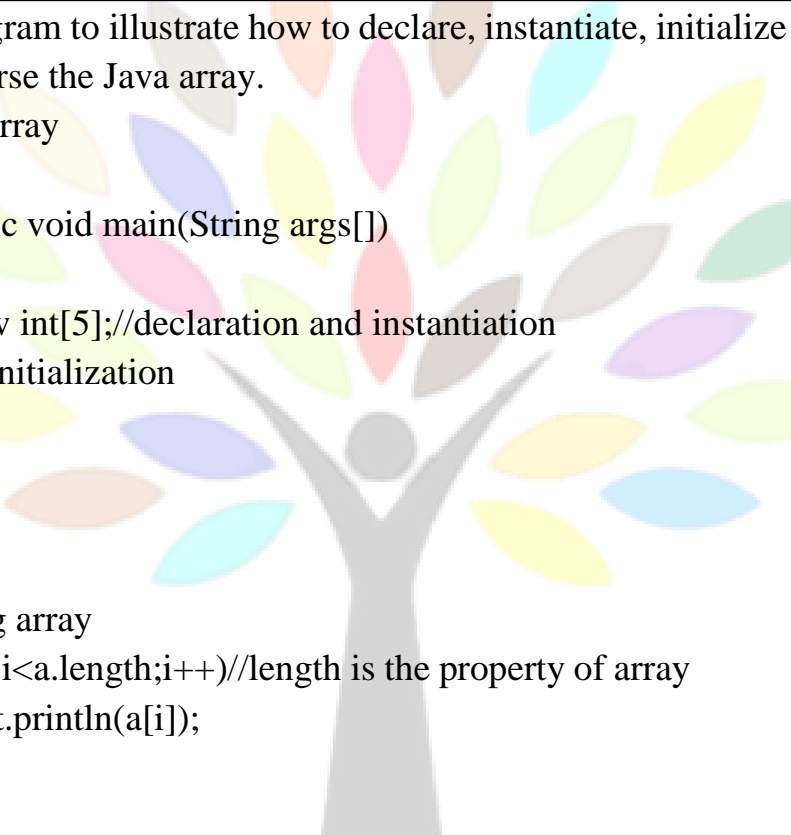
Syntax:

```
dataType[] arr; (or)  
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example:



```
//Java Program to illustrate how to declare, instantiate, initialize  
//and traverse the Java array.  
class Testarray  
{  
public static void main(String args[])  
{  
int a[]=new int[5];//declaration and instantiation  
a[0]=10;//initialization  
a[1]=20;  
a[2]=70;  
a[3]=40;  
a[4]=50;  
//traversing array  
for(int i=0;i<a.length;i++)//length is the property of array  
System.out.println(a[i]);  
}  
}
```

Output:

```
10  
20  
70  
40  
50
```

JAVA

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

int a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1
{
public static void main(String args[])
{
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}
}
```

Output:

33
3
4
5

For-each Loop for Java Array:

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

Syntax:

```
for(data_type variable:array)
{
//body of the loop
}
```

Let us see the example of print the elements of Java array using the for-each loop.

Testarray1.java

```
//Java Program to print the array elements using for-each loop
class Testarray1
```


JAVA

```
{  
public static void main(String args[])  
{  
    int arr[]={33,3,4,5};  
    //printing array using for-each loop  
    for(int i:arr)  
        System.out.println(i);  
}  
}
```

Output:

33
3
4
5

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an `ArrayIndexOutOfBoundsException` if length of the array is negative, equal to the array size or greater than the array size while traversing the array.

```
//ArrayIndexOutOfBoundsException in a Java Array.  
public class TestArrayException  
{  
    public static void main(String args[])  
    {  
        int arr[]={50,60,70,80};  
        for(int i=0;i<=arr.length;i++)  
        {  
            System.out.println(arr[i]);  
        }  
    }  
}
```

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4 at
TestArrayException.main(TestArrayException.java:5)

JAVA

50

60

70

80

Multidimensional Arrays:

A multidimensional array is an array of arrays. To create a two-dimensional array, add each array within its own set of curly braces:

Example:

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

myNumbers is now an array with two arrays as its elements.

Pgm:

```
public class Main
{
    public static void main(String[] args)
    {
        int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
        int x = myNumbers[1][2];
        System.out.println(x);
    }
}
```

Output:

7

Java Program to multiply two matrices

We can multiply two matrices in java using binary * operator and executing another loop. A matrix is also known as array of arrays. We can add, subtract and multiply matrices.

In case of matrix multiplication, one row element of first matrix is multiplied by all columns of second matrix.

Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

```
public class MatrixMultiplicationExample{
    public static void main(String args[]){
        //creating two matrices
        int a[][]={{1,1,1},{2,2,2},{3,3,3}};
        int b[][]={{1,1,1},{2,2,2},{3,3,3}};
        //creating another matrix to store the multiplication of two matrices
        int c[][]=new int[3][3]; //3 rows and 3 columns
```

JAVA

```
//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
} //end of k loop
System.out.print(c[i][j]+" "); //printing matrix element
} //end of j loop
System.out.println();//new line
}
}}
```

Output

6 6 6

12 12 12

18 18 18

Java String

In java string is basically an object that represents sequence of char values. An Array of characters works same as Java string.

For example:

1) char[] ch={'j','a','v','a','t','p','o','i','n','t'};

2) String s=new String(ch);is

same as:

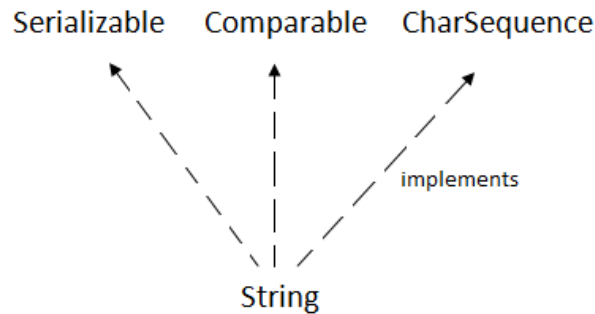
3) String s="Infinity";

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

JAVA

The **java.lang.String** class implements **Serializable**, **Comparable** and **CharSequence** interfaces. The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created.

For mutable strings, you can use **StringBuffer** and **StringBuilder** classes.



Serializable:

To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. A Java object is serializable if its class or any of its super classes **Serializable** interface or its sub interface, **java**.

Comparable:

Java provides **Comparable** interface which should be implemented by any custom class if we want to use **Arrays** or **Collections** sorting methods. The **Comparable** interface has **compareTo()** method which is used by sorting methods, you can check any **String**.

We will discuss mutable string and what is **String** in Java and how to create the **String** object.

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

JAVA

What is Mutable strings?

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

How to create a string object?

There are two ways to create String object:

- 1) By string literal
- 2) By new keyword

1. String literal:

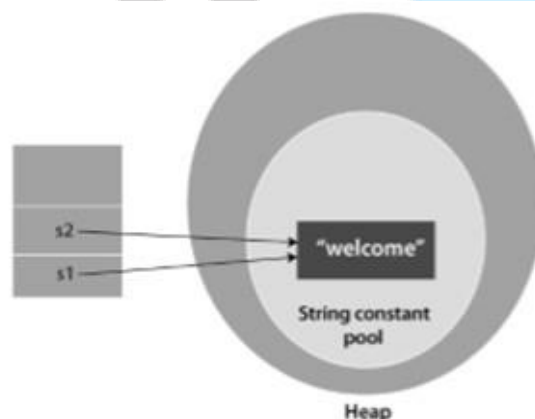
Java String literal is created by using double quotes. For example,

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance In the above example, only one object will be created.

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

Why Java uses the concept of String literal?

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



2. By new keyword

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

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

JAVA

Java String Example:

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

Output:

```
java
strings
example
```

Java String class methods:

Let's see the important methods of String class. Java String toUpperCase() and toLowerCase() method. The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

```
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
```

Output:

```
SACHIN
sachin
Sachin
```

Java String trim() method:

JAVA

The string trim() method eliminates white spaces before and after string.

```
String s=" Sachin ";  
System.out.println(s);// Sachin  
System.out.println(s.trim());//Sachin
```

Output:

Sachin
Sachin

Java String startsWith() and endsWith() method:

```
String s="Sachin";  
System.out.println(s.startsWith("Sa"));//true  
System.out.println(s.endsWith("n"));//true
```

Output:

true
true

Java String charAt() method:

The string charAt() method returns a character at specified index.

```
String s="Sachin";  
System.out.println(s.charAt(0));//S  
System.out.println(s.charAt(3));//h
```

Output:

S
H

Java String length() method:

The string length() method returns length of the string.

```
String s="Sachin";  
System.out.println(s.length());//6
```

Output:

6

Java String replace() method:

JAVA

The string replace() method replaces all occurrence of first sequence of character with second sequence of character.

- 1) String s1="Java is a programming language. Java is a platform. Java is an Island.";
- 2) String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
- 3) System.out.println(replaceString);

Output:

Kava is a programming language. Kava is a platform. Kava is an Island.

String Methods:

Concat():

The **Java String class concat()** method *combines specified string at the end of this string*. It returns a combined string. It is like appending another string.

Pg:1

```
import java.lang.String;
public class Concat
{
    public static void main(String[] args)
    {
        String str1 = "Hello";
        String str2 = "Java";
        String str3 = "Language";
        String str4 = str1.concat(str2);
        System.out.println(str4);
        String str5 = str1.concat(str2).concat(str3);
        System.out.println(str5);
    }
}
```

Output:

HelloJava

HelloJavaLanguage.

JAVA

Pg:2

```
import java.lang.String;
public class Concat1
{
    public static void main(String args[])
    {
        String str="beautiful";
        String s="Rose is ".concat(str);
        System.out.println(s);
    }
}
```

Output:

Rose is beautiful

Equals():

The **Java String class equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true. **true** if characters of both strings are equal otherwise **false**.

Pg:1

```
import java.lang.String;
public class equals
{
    public static void main(String[] args)
    {
        String s1="java";
        String s2="java";
        String s3="Python";
        System.out.println(s1.equals(s2));
        System.out.println(s1.equals(s3));
        System.out.println(s2.equals(s3));
    }
}
```

JAVA

Output:

True

False

False

Pg:2

```
import java.lang.String;
public class equals1
{
    public static void main(String[] args)
    {
        String s1="java";
        String s2="java";
        String s3="Python";
        System.out.println(s1.equals(s2));
        if(s2.equals(s3))
        {
            System.out.println("Condition executed");
        }
        else{
            System.out.println("Condition unsuccessful");
        }
    }
}
```

Output:

True

Condition unsuccessful.

Split():

The java string split() method splits this string against given regular expression and returns a char array.

Pg:1

```
import java.lang.String;
public class Split
```

JAVA

```
{
public static void main(String args[])
{
String s1="java string split method";
String[] words=s1.split("\\s");//splits the string based on whitespace
//using java foreach loop to print elements of string array
for(String w:words)
{
System.out.println(w);
}
}
}
```

Output:

java
string
split
method

Length():

The **Java String class length()** method finds the length of a string. Length of characters. In other words, the total number of characters present in the string.

Pg:1

```
import java.lang.String;
public class Length{
public static void main(String args[]){
String s1="java Language";
String s2="python";
System.out.println("string length is: "+s1.length());
System.out.println("string length is: "+s2.length());
}
}
```

Output:

JAVA

string length is:13

string length is:6

Replace():

The **Java String class replace()** method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Pg:1

```
import java.lang.String;
public class Replace
{
    public static void main(String args[])
    {
        String s1="java is a Object Oriented Programming Language";
        String replaceString=s1.replace('a','e');
        System.out.println(replaceString);
    }
}
```

Output:

jeve is e Object Oriented Progremming Lenguage

Pg:2

```
import java.lang.String;
public class Replace1 {
    public static void main(String[] args) {
        String str = "hhhhh-aaaaa-iiii";
        String sp = str.replace("a","e");
        System.out.println(sp);
        sp = sp.replace("e","a");
        System.out.println(sp);
    }
}
```

Output:

hhhhh-eeee -iiii

JAVA

hhhhh-aaaaa-iiii

CompareTo():

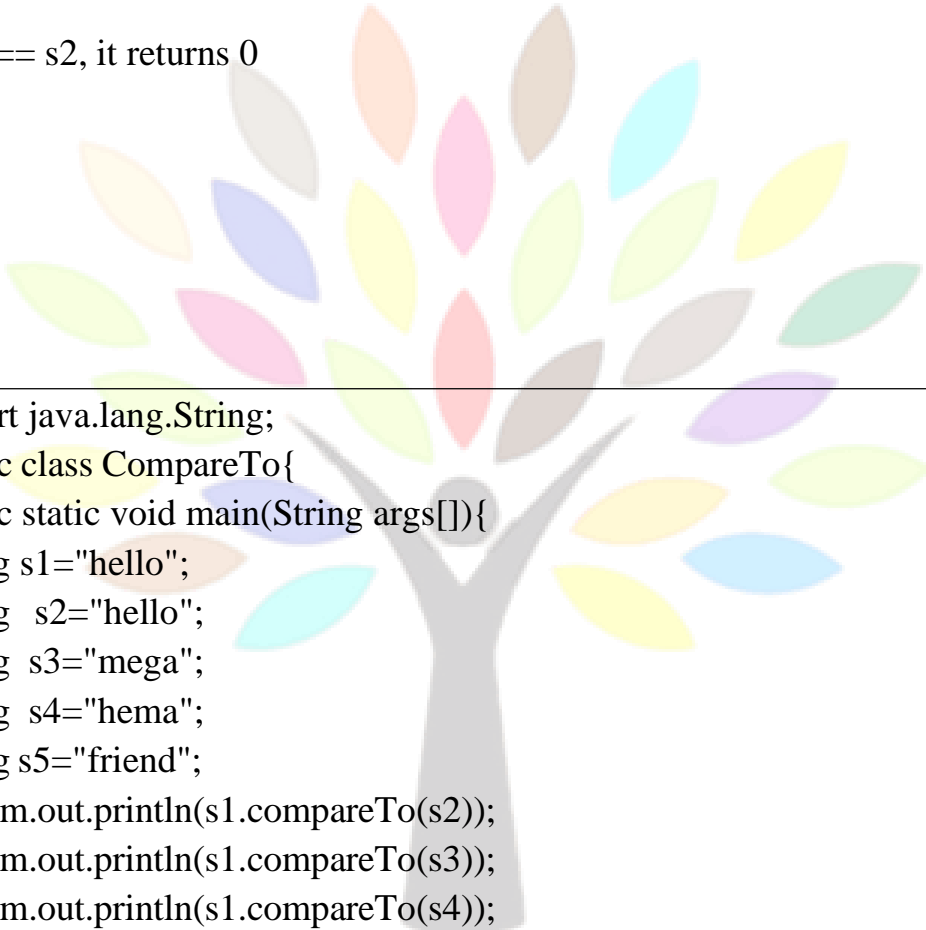
The **Java String class compareTo()** method compares the given string with the current string lexicographically. It returns a positive number, negative number, or 0.

if $s1 > s2$, it returns positive number

if $s1 < s2$, it returns negative number

if $s1 == s2$, it returns 0

Pg:1



```
import java.lang.String;
public class CompareTo{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="mega";
String s4="hema";
String s5="friend";
System.out.println(s1.compareTo(s2));
System.out.println(s1.compareTo(s3));
System.out.println(s1.compareTo(s4));
System.out.println(s1.compareTo(s5));
}
}
```

Output:

0
-5
-1
2

Substring():

JAVA

The **Java String** class **substring()** method returns a part of the string.



JAVA

Pg : 1

```
import java.lang.String;
public class Substring{
public static void main(String args[]){
String s1="javaprogram";
System.out.println(s1.substring(2,4));
System.out.println(s1.substring(2));
}
}
```

Output:

va
vaprogram

Compare():

The compare() method in Java compares two class specific objects (x, y) given as parameters. It returns the value:

0 : if (x==y)

-1: if (x < y)

1 : if (x > y)

Ex:

```
import java.lang.String;
class Compare{
public static void main(String args[])
{
int a = 10;
int b = 20;
System.out.println(Integer.compare(a, b));
int x = 30;
int y = 30;
System.out.println(Integer.compare(x, y));
int w = 15;
int z = 8;
System.out.println(Integer.compare(w, z));
}
```

JAVA

```
}  
}
```

Output:

```
-1  
0  
1
```

Intern():

The **Java String class intern()** method returns the interned string. It can be used to return string from memory if it is created by a new keyword. It creates an exact copy of the heap string object in the String Constant Pool.

Signature

The signature of the intern() method is given below:

1. **public** String intern()

Returns

interned string

The need and working of the String.intern() Method

When a string is created in Java, it occupies memory in the heap. Also, we know that the String class is immutable. Therefore, whenever we create a string using the new keyword, new memory is allocated in the heap for corresponding string, which is irrespective of the content of the array. Consider the following code snippet.

1. String str = **new** String("Welcome to JavaTpoint.");
2. String str1 = **new** String("Welcome to JavaTpoint");
3. System.out.println(str1 == str); // prints false

The println statement prints false because separate memory is allocated for each string literal. Thus, two new string objects are created in the memory i.e. str and str1. that holds different references.

We know that creating an object is a costly operation in Java. Therefore, to save time, Java developers came up with the concept of String Constant Pool (SCP). The SCP is an area inside the heap memory. It contains the unique strings. In order to put the strings in the string pool, one needs to call the **intern()** method. Before creating an object in the string pool, the JVM checks whether the string is already present in the pool or not. If the string is present, its reference is returned.

JAVA

1. String str = **new** String("Welcome to JavaTpoint").intern(); // statement - 1
2. String str1 = **new** String("Welcome to JavaTpoint").intern(); // statement - 2
3. System.out.println(str1 == str); // prints true

In the above code snippet, the intern() method is invoked on the String objects. Therefore, the memory is allocated in the SCP. For the second statement, no new string object is created as the content of str and str1 are the same. Therefore, the reference of the object created in the first statement is returned for str1. Thus, str and str1 both point to the same memory. Hence, the print statement prints true.

Ex:1

```
import java.lang.String;
public class Intern
{
    public static void main(String args[])
    {
        String s1=new String("hello");
        String s2="hello";
        String s3=s1.intern();
        String s4=s2.intern();
        System.out.println(s1==s2);
        System.out.println(s2==s3);
        System.out.println(s2==s4);
    }
}
```

Output:

False
True
True

Ex:2

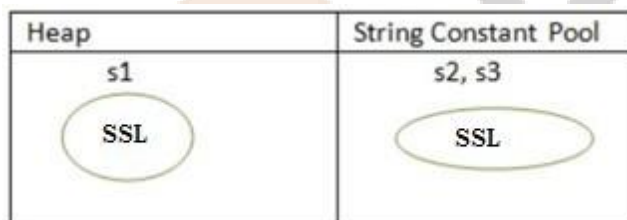
```
class SP {
```

JAVA

```
public static void main(String[] args)
{
// S1 refers to Object in the Heap Area
String s1 = new String("java");
// S2 refers to Object in SCP Area
String s2 = s1.intern();
// Comparing memory locations s1 is in Heap,s2 is in SCP
System.out.println(s1 == s2);
// Comparing only values
System.out.println(s1.equals(s2));
// S3 refers to Object in the SCP Area
String s3 = "java";
System.out.println(s2 == s3);
}
}
```

Output:

False
True
True

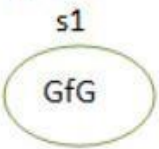



Ex:3

```
class SSL
{
public static void main(String[] args)
{
// S1 refers to Object in the Heap Area
String s1 = new String("SSL");
// S2 now refers to Object in SCP Area
String s2 = s1.concat("SSL");
}
```

JAVA

```
// S3 refers to Object in SCP Area
String s3 = s2.intern();
System.out.println(s2 == s3);
// S4 refers to Object in the SCP Area
String s4 = "SSLSSL";
System.out.println(s3 == s4);
}
}
```

Heap	String Constant Pool
s1 	s2, s3, s4 

Output:

True

True

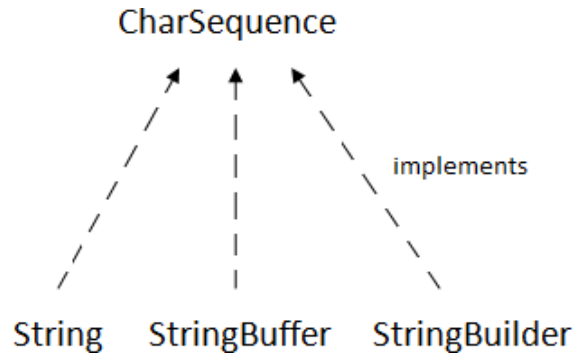
CharSequence interfaces:

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It three classes are,

- 1) String
- 2) String Buffer
- 3) String Builder

The Java String is immutable which means it cannot be changed.

JAVA



String Buffer vs String Builder:

String Buffer	String Builder
1.Thread Safe	Not Thread Safe
2.Synchronized	Not Synchronized
3.Slower	Faster

String Buffer:

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

Ex:1(Append)

```
public class Main{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Hello ");
        sb.append("Java");
        System.out.println(sb);
    }
}
```

Output:

HelloJava

Ex:2(insert)

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

JAVA

```
StringBuffer sb=new StringBuffer("Hello ");  
Sb.insert(1, "Language");  
System.out.println(sb);  
}  
}
```

Output:

HLanguageello

Ex:3(Replace)

```
public class Main  
{  
    public static void main(String args[])  
    {  
        StringBuffer sb=new StringBuffer("Hello ");  
        Sb.insert(1,3 "Java");  
        System.out.println(sb);  
    }  
}
```

Output:

HJavalo

Ex:4(Reverse)

```
public class Main  
{  
    public static void main(String args[])  
    {  
        StringBuffer sb=new StringBuffer("Hello ");  
        sb.reverse();  
        System.out.println(sb);  
    }  
}
```

Output:

JAVA

olleH

String Builder:

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.

Ex:1(Append)

```
public class Main
{
    public static void main(String args[])
    {
        StringBuilder sb=new StringBuilder("Hello ");
        sb.append("Java language");
        System.out.println(sb);
    }
}
```

Output:

Hello Java language

Ex:2(Insert)

```
public class Main
{
    public static void main(String args[])
    {
        StringBuilder sb=new StringBuilder("Hello ");
        sb.insert (1,"Java ");
        System.out.println(sb);
    }
}
```

Output:

HJavaello

Ex:3(Delete)

JAVA

```
public class Main
{
    public static void main(String args[])
    {
        StringBuilder sb=new StringBuilder("Hello ");
        sb.delete(1,3);
        System.out.println(sb);
    }
}
```

Output:

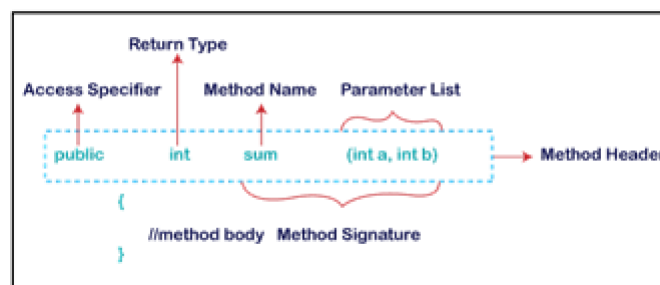
Hlo

Method in java

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code.

We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code.

Method Declaration



Types of methods:

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined method:

JAVA

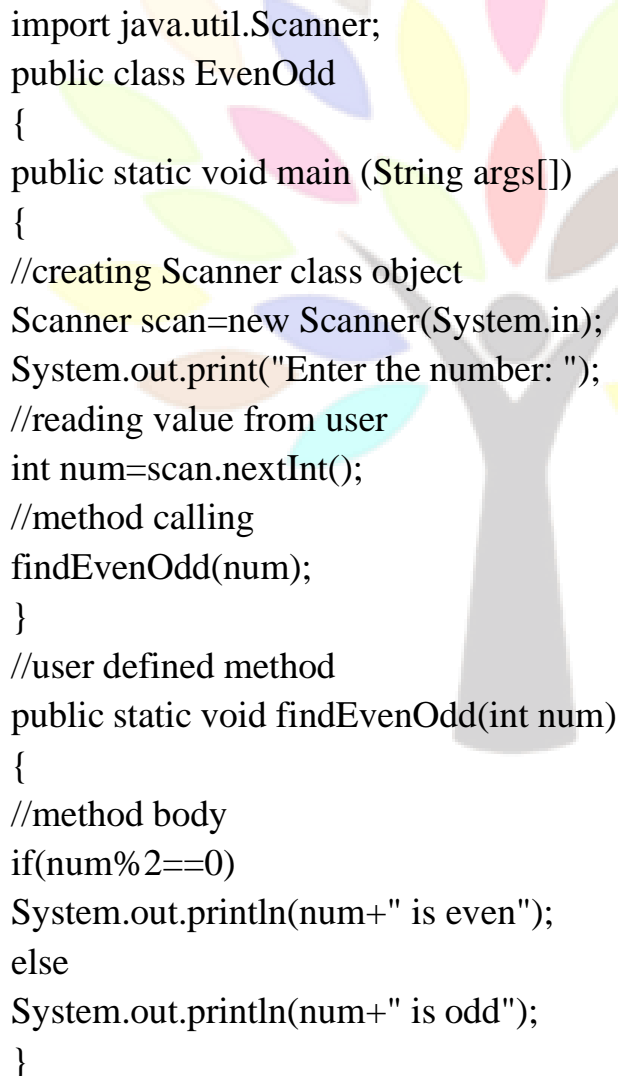
Predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**.

We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length()**, **equals()**, **compareTo()**, **sqrt()**, etc..

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.

Example:



```
import java.util.Scanner;
public class EvenOdd
{
    public static void main (String args[])
    {
        //creating Scanner class object
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from user
        int num=scan.nextInt();
        //method calling
        findEvenOdd(num);
    }
    //user defined method
    public static void findEvenOdd(int num)
    {
        //method body
        if(num%2==0)
            System.out.println(num+" is even");
        else
            System.out.println(num+" is odd");
    }
}
```


JAVA

```
}
```

Output 1:

Enter the number: 12
12 is even

Output 2:

Enter the number: 99
99 is odd

Call by value:

Call by Value means calling a method with a parameter as value. Through this, the argument value is passed to the parameter.

Example:

```
public class Tester
{
    public static void main(String[] args)
    {
        int a = 30;
        int b = 45;
        System.out.println("Before swapping, a = " + a + " and b = " + b);
        // Invoke the swap method
        swapFunction(a, b);
        System.out.println("\n**Now, Before and After swapping values will be same here**");
        System.out.println("After swapping, a = " + a + " and b is " + b);
    }
    public static void swapFunction(int a, int b)
    {
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b);
        // Swap n1 with n2
        int c = a;
        a = b;
```

JAVA

```
        b = c;  
        System.out.println("After swapping(Inside), a = " + a + " b = " + b);  
    }  
}
```

Output:

Before swapping, a = 30 and b = 45

Before swapping(Inside), a = 30 b = 45

After swapping(Inside), a = 45 b = 30

****Now, Before and After swapping values will be same here**:**

After swapping, a = 30 and b is 45

Call by reference:

Java uses only call by value while passing reference variables as well. It creates a copy of references and passes them as valuable to the methods. As reference points to same address of object, creating a copy of reference is of no harm. But if new object is assigned to reference it will not be reflected.

Example:

```
public class JavaTester {  
    public static void main(String[] args) {  
        IntWrapper a = new IntWrapper(30);  
        IntWrapper b = new IntWrapper(45);  
        System.out.println("Before swapping, a = " + a.a + " and b = " + b.a);  
        // Invoke the swap method  
        swapFunction(a, b);  
        System.out.println("\n**Now, Before and After swapping values will be  
different here**:");  
        System.out.println("After swapping, a = " + a.a + " and b is " + b.a);  
    }  
    public static void swapFunction(IntWrapper a, IntWrapper b) {  
        System.out.println("Before swapping(Inside), a = " + a.a + " b = " + b.a);  
        // Swap n1 with n2  
        IntWrapper c = new IntWrapper(a.a);  
        a.a = b.a;
```

JAVA

```
b.a = c.a;  
System.out.println("After swapping(Inside), a = " + a.a + " b = " + b.a);  
}  
}  
class IntWrapper {  
    public int a;  
    public IntWrapper(int a){ this.a = a;}  
}
```

Output:

Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30
Now, Before and After swapping values will be different here:
After swapping, a = 45 and b is 30

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.

OOPs concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object:

➤ Any entity that has state and behavior is known as an object.

JAVA

- For example, a chair, pen, table, keyboard, bike, etc...
- An Object can be defined as an “**Instance of a class**”.
- Example:
- A pen is an object because it has states like color, name, etc. as well as behaviors like writing, drawing, 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.
- So, a class is a template for objects, and an object is an instance of a class.

Class VS Objects:

Objects	Class
Object is an instance of a class	Class is a blueprint or template from which objects are created.
Object is a real world entity .Such as pen, laptop, mobile, mouse, bag, etc.,	Class is a group of similar objects .
Object is a Physical entity.	Class is a logical entity.
Object is created through new keyword . Eg: Student s1=new Student();	Class is declared using class keyword . Eg: Class Student{ }
Object is created many times as per requirement.	Class is declared once .

Class: Mobile phone **Object:** iPhone, Samsung,vivo,etc.,

Class: Fruit **Object:** Apple, Banana, Mango,etc.,

Ex:1

JAVA

```
class Student{
public static void main(String args[]){
int id;
String name;
Student s1=new Student();
s1.id=101;
s1.name="john";
System.out.println(s1.id+" "+s1.name);
}
}
```

Output:

101 john

Ex:2

```
//Stud.java
class Stud{
int id;
String name;
}
//Stud2.java
class Stud2{
public static void main(String args[]){
Student s1=new Student();
s1.id=101;
s1.name="Sana";
System.out.println(s1.id+" "+s1.name);
}
}
```

Output:

C:\Documents\Javac Stud.java

C:\Documents\Javac Stud2.java

JAVA

C:\Documents\Java Stud2
101 Sana

Ex:3

```
public class Main1
{
String fname = "John";
String lname = "das";
int age = 24;
public static void main(String[] args)
{
Main1 my= new Main1();
System.out.println("Name:" +my.fname + " " +my.lname);
System.out.println("Age: " + my.age);
}
}
```

Output:

Name:John das
Age:24

Ex:4

```
public class Main2{
int age=16;
String name="priya";
public static void main(String[] args) {
Main2 my= new Main2();
Main2 my1= new Main2();
System.out.println(my.age);
System.out.println(my1.name);
}
```

JAVA

```
}  
}
```

Output:

16

Priya

Constructors in Java:

A 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.

It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Types of Constructors:

- 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.

Ex:

```
import java.io.*;  
class Hello  
{  
    int num;  
    String name;  
    Hello(){  
        System.out.println("Constructor called");  
    }  
}
```

JAVA

```
}  
class Welcome {  
public static void main(String[] args)  
{  
// this would invoke default constructor.  
Hello Hai = new Hello();  
// Default constructor provides the default values to the object like 0, null  
System.out.println(Hai.name);  
System.out.println(Hai.num);  
}  
}
```

Output:

Constructor called
Null
0

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.

Ex:

```
import java.io.*;  
class Hello{  
String name;  
int id;  
Hello(String name, int id)  
{  
this.name = name;  
this.id = id;  
}  
}
```


JAVA

```
class Welcome{
public static void main(String[] args)
{
Hello a= new Hello("U1",16);
System.out.println("HelloName :"+ a.name +" and HelloId :"+a.id);
}
}
```

Output:

HelloName :U1
HelloId :16

Java Inner class:

Java inner class or nested class is a class that is declared inside the class or interface. We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable. Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
//code
class Java_Inner_class{
//code
}
}
class OuterClass {
int x = 10;

class InnerClass {
int y = 5;
}
}
public class Main {
```

JAVA

```
public static void main(String[] args) {  
    OuterClass myOuter = new OuterClass();  
    OuterClass.InnerClass myInner = myOuter.new InnerClass();  
    System.out.println(myInner.y + myOuter.x);  
}  
}
```

Encapsulation

Binding code and data together into a single unit are known as encapsulation. This also helps to achieve “Data hiding”. 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.

Example:

```
class Test{  
    private int a;  
    int getvalue()  
    {  
        return a;  
    }  
    void setvalue(int x)  
    {  
        a=x;  
    }  
}  
  
public class Encaps{  
    public static void main(String[]args){
```

JAVA

```
Test obj=new Test();  
obj.setvalue(10);  
int b=obj.getvalue();  
System.out.println(b);  
}  
}
```

Output:

10

Inheritance

Inheritance is an important pillar of OOP (Object-Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features (fields and methods) of another class.

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

- **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.

Syntax:

```
class derived-class extends base-class  
{  
//methods and fields  
}
```

The extends keyword indicates that you are making a new class that derives from

JAVA

an existing class. The meaning of "extends" is to increase the functionality.

Example:

```
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);  
    }  
}
```

Output:

Programmer salary is: 40000.0
Bonus of programmer is: 10000

Types of Inheritance in Java:

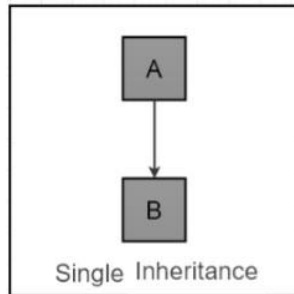
Java supports the following four types of inheritance:

- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

1. Single Inheritance:

In single inheritance, subclasses inherit the features of one super class. In the image below, class A serves as a base class for the derived class B.

JAVA



Example:

```
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:

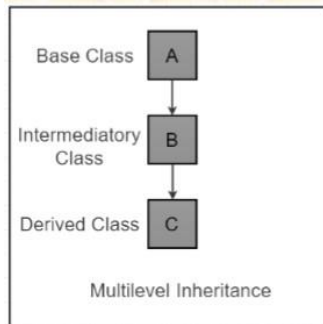
barking...
eating...

2. Multilevel Inheritance:

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the below image, class A serves as a base class for the derived class B, which in turn serves

JAVA

as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



Example:

```
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();
}
}
```

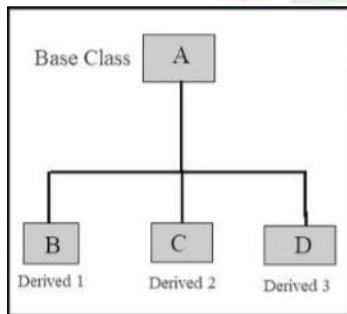
Output:

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

3 Hierarchical Inheritance:

JAVA

If a number of classes are derived from a single base class, it is called hierarchical inheritance.



Example:

```
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
}
}
```

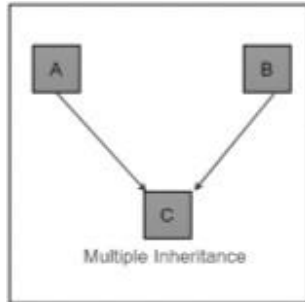
Output:

meowing...
eating...

4. Multiple Inheritance (not supported in java):

Java does not support multiple inheritances due to ambiguity.

JAVA



Example:

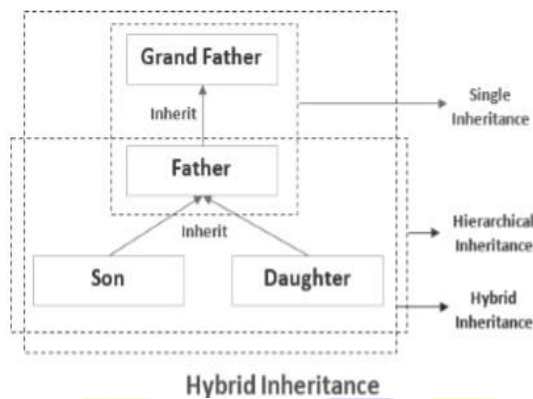
```
class Wishes
{
void message()
{
System.out.println("Best of Luck!!");
}
}
class Birthday
{
void message()
{
System.out.println("Happy Birthday!!");
}
}
public class Demo extends Wishes, Birthday //considering a scenario
{
public static void main(String args[])
{
Demo obj=new Demo();
//can't decide which classes' message() method will be invoked
obj.message();
}
}
```

The above code gives error because the compiler cannot decide which message () method is to be invoked. Due to this reason, Java does not support multiple inheritances at the class level but can be achieved through an interface.

JAVA

5. Hybrid Inheritance (Through Interfaces):

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



Daughter.java

```
//parent class
class GrandFather
{
    public void show()
    {
        System.out.println("I am grandfather.");
    }
}

//inherits GrandFather properties
class Father extends GrandFather
{
    public void show()
    {
        System.out.println("I am father.");
    }
}
```

```
//inherits Father properties
```

```
class Son extends Father
```

JAVA

```
{  
public void show()  
{  
System.out.println("I am son.");  
}  
}  
//inherits Father properties  
public class Daughter extends Father  
{  
public void show()  
{  
System.out.println("I am a daughter.");  
}  
public static void main(String args[])  
{  
Daughter obj = new Daughter();  
obj.show();  
}  
}
```

Output:

I am daughter.

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.
- Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

JAVA

- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

Method overloading:

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

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.

```
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

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.

Rules for Java Method Overriding:

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

Example:

JAVA

```
//Creating a parent class
class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
public static void main(String args[]){
//creating an instance of child class
Bike obj = new Bike();
//calling the method with child class instance
obj.run();
}
}
```

Output:

Vehicle is running

Example 2:

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class
class Vehicle{
//defining a method
void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
//defining the same method as in the parent class
void run(){System.out.println("Bike is running safely");}
public static void main(String args[]){
Bike2 obj = new Bike2();//creating object
obj.run();//calling method
}
}
```

JAVA

```
}
```

Output:

Bike is running safely

Abstraction

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

The “abstract” keyword is a non-access modifier, used for classes and methods:

- **Abstract class:**

Abstract class is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- **Abstract method:**

Abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Example:

```
abstract class Bike
{
    abstract void run();
}
class Honda extends Bike
{
    void run()
    {
        System.out.println("running safely");
    }
    public static void main(String args[])
    {
```

JAVA

```
Bike obj = new Honda();  
obj.run();  
}  
}
```

Output:

running safely

Interface

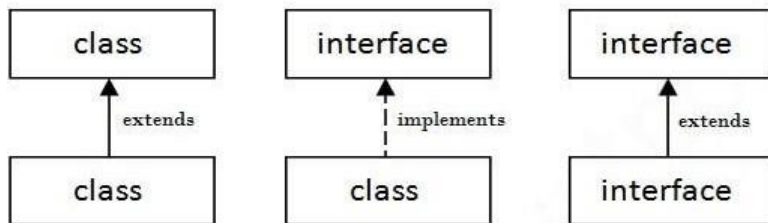
An interface in java is a blueprint of a class. It has static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction.

There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

Syntax:

```
interface <interface_name>{  
    // declare constant fields declare methods that abstract by default.  
}
```

The relationship between classes and interfaces:



Example:

```
interface printable  
{
```

JAVA

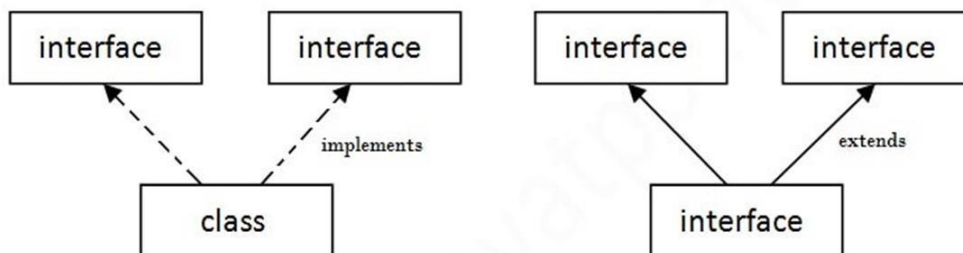
```
void print();
}
class A4 implements printable
{
public void print()
{
System.out.println("Hello World");
}
public static void main(String[] args)
{
A4 obj=new A4();
obj.print();
}
}
```

Output:

Hello World.

Multiple Inheritance in java using Interface:

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example:

```
interface Printable
{
```


JAVA

```
void print();
}
interface Showable
{
void show();
}
class A4 implements Printable,Showable
{
public void print()
{
System.out.println("Hello");
}
public void show()
{
System.out.println("Welcome");
}

public static void main(String args[]){
A4 obj = new A4();
obj.print();
obj.show();
}
}
```

Output:

Hello

Welcome

Exception Handling

- The Exception Handling in Java is one of the powerful *mechanism to handle the runtime errors*.so that the normal flow of the application can be maintained.
- Exception is an abnormal condition.
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

JAVA

Types of Exceptions:

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions:

1) Checked Exception:

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions.

For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) UnChecked Exception:

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc...

Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error:

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc...

Java Exception Keywords:

Java provides five keywords that are used to handle the exception. The following table describes each.

JAVA

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java try and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

Syntax:

```
try{  
    //Block of code to try  
}  
catch(Exception e){  
    //Block of code to handle errors  
}
```

Prgm:

JAVA

```
public class Test1 {  
    public static void main(String[ ] args) {  
        int[] myNumbers = { 1, 2, 3};  
        System.out.println(myNumbers[4]); // error!  
    }  
}
```

Output:

Exception in thread “main” java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 3 at Test1.main(Test1.java:24)

Using try and catch ex:

```
public class Test1 {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = { 1, 2, 3};  
            System.out.println(myNumbers[10]);  
        } catch (Exception e) {  
            System.out.println("Something went wrong.");  
        }  
    }  
}
```

Output:

Something went wrong.

Finally:

The finally keyword is used to execute code (used with exceptions - try.. catch statements) no matter if there is an exception or not.

Using try,catch and finally Example:

```
public class Test1 {  
    public static void main(String[] args){
```

JAVA

```
try{
int data=100/0;
}
catch(Exception e)
{
System.out.println(e);
}
finally{
int a,b,c;
a=34;
b=67;
c=a+b;
System.out.println(c);
}
System.out.println("coding...");
}
}
```

Output:

Java.lang.ArithmeticException:/ by zero

101

Coding...

Throw:

The throw statement allows you to create a custom error.

Ex:

```
public class Test1 {
static void checkAge(int age) {
if (age < 18) {
throw new ArithmeticException("You must be at least 18 years old.");
}
else {
System.out.println("You are old enough!");
}
}
}
```

JAVA

```
    }  
    }  
    public static void main(String[] args) {  
        checkAge(12); // Set age to 15 (which is below 18...)  
        //checkAge(22);(You are old enough)  
    }  
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: You must be at least 18 years old.

at Test1.checkAge(Test1.java:42)
at Test1.main(Test1.java:50)

Throws:

The throws keyword is used in the signature of the method.

Ex:

```
public class Test1 {  
    //defining a method  
    public static int divideNum(int m, int n) throws ArithmeticException {  
        int div = m / n;  
        return div;  
    }  
    //main method  
    public static void main(String[] args) {  
        Test1 obj = new Test1();  
        try {  
            System.out.println(obj.divideNum(45, 0));  
        }  
        catch (ArithmeticException e){  
            System.out.println("\nNumber cannot be divided by 0");  
        }  
        System.out.println("Code..");  
    }  
}
```

JAVA

Output:

Number cannot be divided by 0

Code...

Package

- Package is a Collection of class. It is used for reuse the classes.
- 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, swing, net, io, util, sql etc...

Two Types:

- Pre-defined (build-in)
- User defined

Pre-defined:

Java - Package (Super Package, Base Package)

Util, io, lang, awt – Subpackage

Using a package in java program is very simple.

Just include the Command package at the beginning of the program.

1) import java.io.*;

import - Keyword

java - Packagename

io - subpackage

* - class (all) automatic

2) import java.util.Scanner;

JAVA

import - Keyword

java - Packagename

util - subpackage

Scanner – class

1.Java.lang :

It Contains language support classes(primitive datatypes, math operations).This package is automatically imported.

2.java.io:

It contains classed for supporting input and output operations.

3.java.util:

It contains utility classes which implement data structure like Linkedlist,Dictionary and support for date/time operations.

4.java.applet:

It contains classes for creating Applets.

5.java.awt:

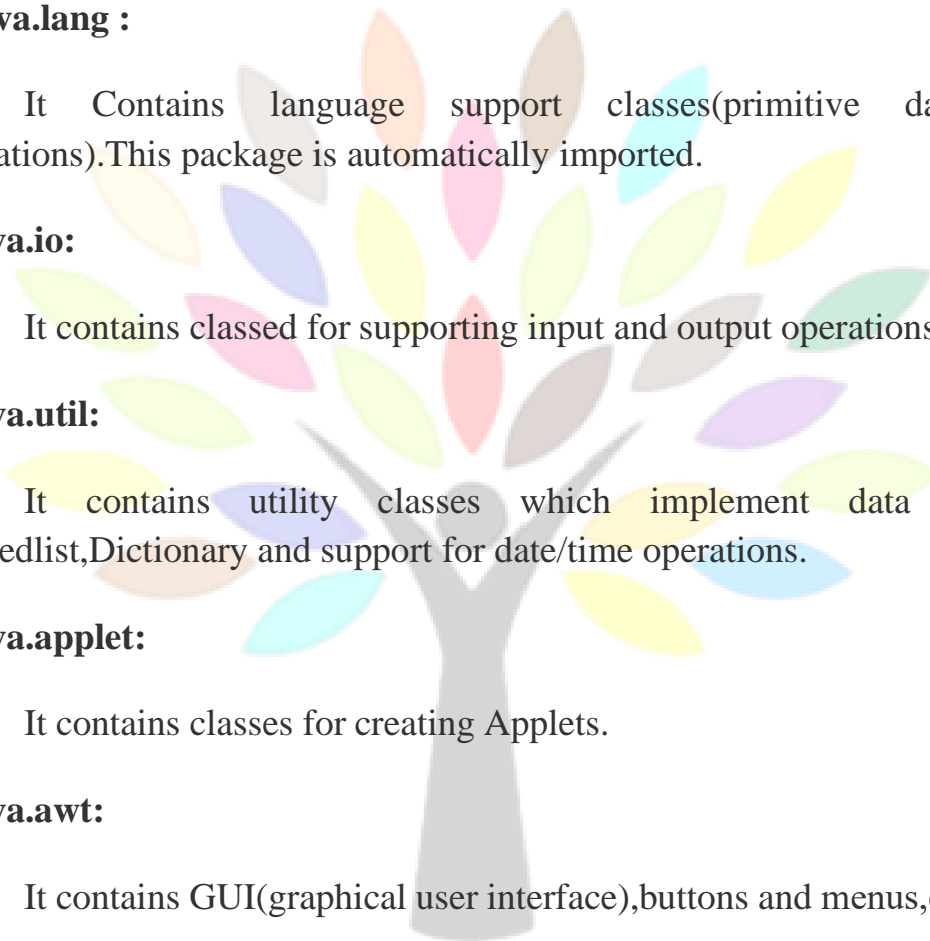
It contains GUI(graphical user interface),buttons and menus,etc....

6.java.net:

It support networking operations.

User Defined:

Syntax:



JAVA

```
Package Packagename;
```

//save as Simple.java

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

To Compile:

```
javac -d . Simple.java
```

To Run:

```
java mypack.Simple
```

Output:

Welcome to package

Subpackage in java

Package inside the package is called the “subpackage”.It should be created to categorize the package further.

Ex:

```
package com.test.add;  
class Simple{  
    public static void main(String args[]){  
        System.out.println("Hello subpackage");  
    }  
}
```

To Compile:javac -d . Simple.java

JAVA

To Run: java com.test.add.Simple

Output:

Hello subpackage

Classpath

CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files.

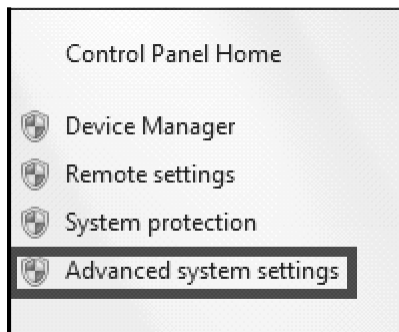
The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH. The CLASSPATH depends on what you are setting the CLASSPATH. The CLASSPATH has a directory name or file name at the end.

Steps to Set CLASSPATH:-

Step 1: Click on the Windows button and choose Control Panel. Select System.

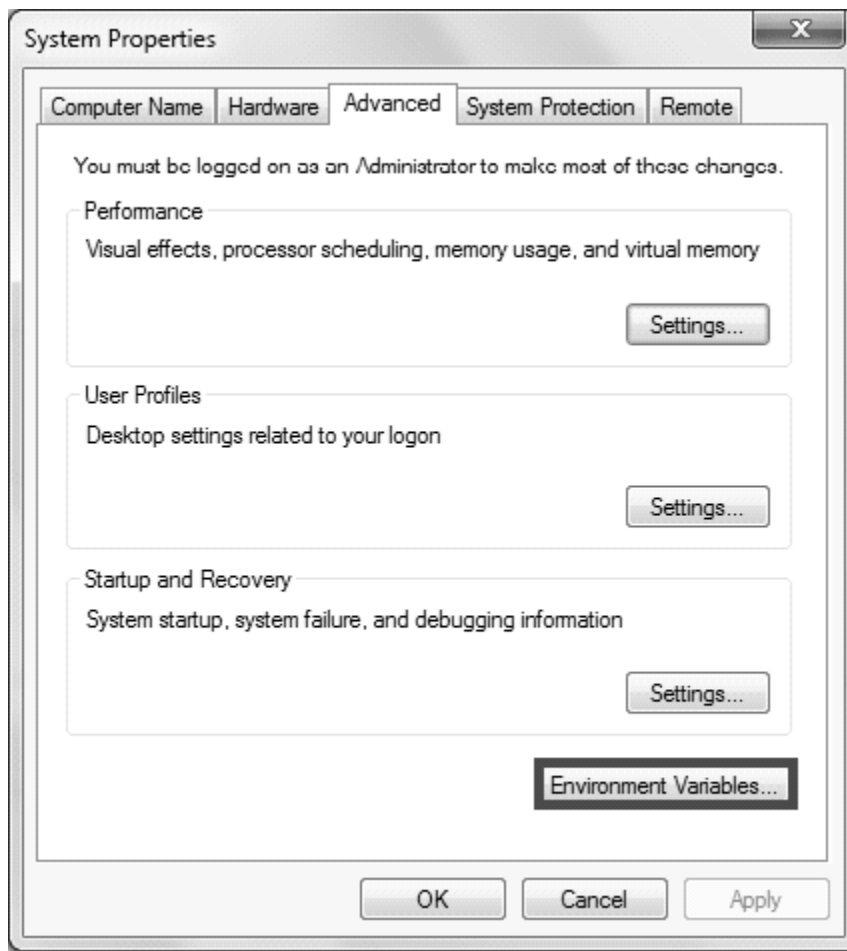


Step 2: Click on Advanced System Settings.



Step 3: A dialog box will open. Click on Environment Variables.

JAVA

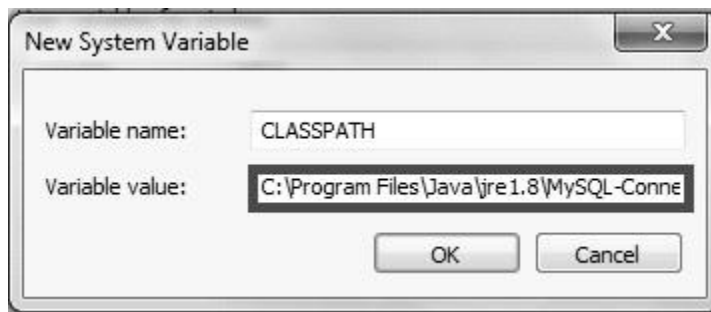


Step 4: If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

If the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as C:\Program Files\Java\jre1.8\MySQL-Connector Java.jar;.

Remember: Put ;. at the end of the CLASSPATH.

JAVA



File Handling

File handling is an important part of any application. Java has several methods for creating, reading, updating, and deleting files.

The File class from the java.io package, allows us to work with files. To use the File class, create an object of the class, and specify the filename or directory name

Stream:

In Java, streams are the sequence of data that are read from the source and written to the destination.

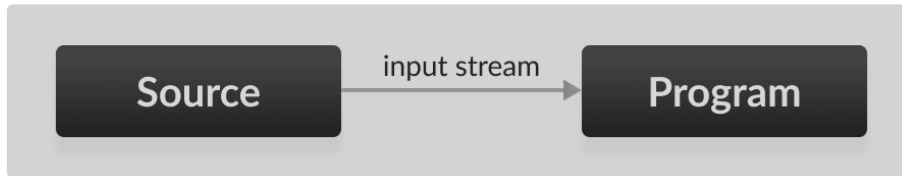
An input stream is used to read data from the source. And, an output stream is used to write data to the destination.

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

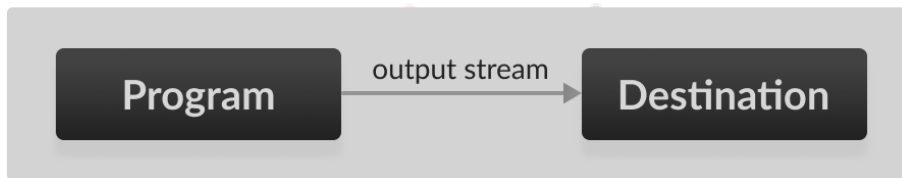
For example, in our first Hello World example, we have used System.out to print a string. Here, the System.out is a type of output stream. Similarly, there are input streams to take input.

JAVA

Reading data from source



Writing data to destination



Types of Streams:

Depending upon the data a stream holds, it can be classified into:

- Byte Stream
- Character Stream

Byte Stream:

- Byte stream is used to read and write a single byte (8 bits) of data.
- All byte stream classes are derived from base abstract classes called `InputStream` and `OutputStream`.
 - a. `InputStream` Class
 - b. `OutputStream` Class

Character Stream:

- Character stream is used to read and write a single character of data.
- All the character stream classes are derived from base abstract classes `Reader` and `Writer`.
 - a. `Reader` Class
 - b. `Writer` Class

JAVA

Example:

```
import java.io.File; // Import the File class
File myObj = new File("filename.txt"); // Specify the filename
```

The File class has many useful methods for creating and getting information about files. For example:

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

JAVA

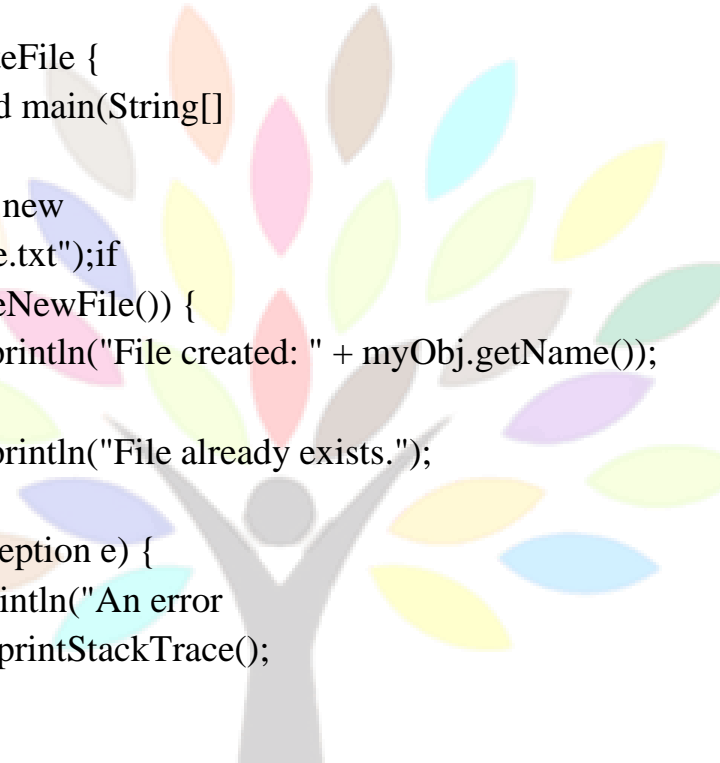
Create a File:

To create a file in Java, you can use the `createNewFile()` method. This method returns a boolean value: `true` if the file was successfully created, and `false` if the file already exists.

Example

```
import java.io.File; // Import the File class
import java.io.IOException; // Import the IOException class to handle errors

public class CreateFile {
    public static void main(String[]
args) { try {
        File myObj = new
        File("filename.txt");if
        (myObj.createNewFile()) {
            System.out.println("File created: " + myObj.getName());
        } else {
            System.out.println("File already exists.");
        }
    } catch (IOException e) {
        System.out.println("An error
        occurred.");e.printStackTrace();
    }
}
```



Write To a File:

In the following example, we use the `FileWriter` class together with its `write()` method to write some text to the file we created in the example above.

Example:

```
import java.io.FileWriter; // Import the FileWriter class
import java.io.IOException; // Import the IOException class to handle errors

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

JAVA

```
try {
    FileWriter myWriter = new FileWriter("filename.txt");
    myWriter.write("Files in Java might be tricky, but it is fun enough!");
    myWriter.close();
    System.out.println("Successfully wrote to the file.");
} catch (IOException e) {
    System.out.println("An error occurred.");
    e.printStackTrace();
}
}
```

Read a File:

In the previous chapter, you learned how to create and write to a file. In the following example, we use the Scanner class to read the contents of the text file we created in the previous chapter:

Example

```
import java.io.File; // Import the File class
import java.io.FileNotFoundException; // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files

public class ReadFile {
    public static void main(String[] args) {
        try {
            File myObj = new File("filename.txt");
            Scanner myReader = new Scanner(myObj);
            while (myReader.hasNextLine()) {
                String data = myReader.nextLine();
                System.out.println(data);
            }
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```


JAVA

```
}  
}  
}
```

Get File Information

To get more information about a file, use any of the File methods:

Example

```
import java.io.File; // Import the File class  
  
public class GetFileInfo {  
    public static void main(String[] args) {  
        File myObj = new  
        File("filename.txt");if  
        (myObj.exists()) {  
            System.out.println("File name: " + myObj.getName());  
            System.out.println("Absolute path: " +  
            myObj.getAbsolutePath());System.out.println("Writeable: " +  
            myObj.canWrite()); System.out.println("Readable " +  
            myObj.canRead()); System.out.println("File size in bytes " +  
            myObj.length());  
        } else {  
            System.out.println("The file does not exist.");  
        }  
    }  
}
```

Delete a File:

To delete a file in Java, use the delete() method:

Example

```
import java.io.File; // Import the File class  
  
public class DeleteFile {  
    public static void main(String[] args) {  
        File myObj = new File("filename.txt");
```

JAVA

```
if (myObj.delete()) {  
    System.out.println("Deleted the file: " + myObj.getName());  
} else {  
    System.out.println("Failed to delete the file.");  
}  
}  
}
```

Delete a Folder:

You can also delete a folder. However, it must be empty:

Example

```
import java.io.File;  
  
public class DeleteFolder {  
    public static void main(String[] args) {  
        File myObj = new File("C:\\Users\\MyName\\Test");  
        if (myObj.delete()) {  
            System.out.println("Deleted the folder: " + myObj.getName());  
        } else {  
            System.out.println("Failed to delete the folder.");  
        }  
    }  
}
```

Garbage Collection:

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using `free()` function in C language and `delete()` in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.

JAVA

- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

By nulling a reference:

```
Employee e=new Employee();  
e=null;
```

By assigning a reference to another:

```
Employee e1=new Employee();  
Employee e2=new Employee();  
e1=e2;//now the first object referred by e1 is available for garbage collection
```

By anonymous object:

```
new Employee();
```

finalize() method:

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){ }
```

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method:

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){ }
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java:

```
public class TestGarbage1{
```

JAVA

```
public void finalize(){System.out.println("object is garbage collected");}
public static void main(String args[]){
    TestGarbage1 s1=new TestGarbage1();
    TestGarbage1 s2=new TestGarbage1();
    s1=null;
    s2=null;
    System.gc();
}
}
```

Output:

object is garbage collected
object is garbage collected

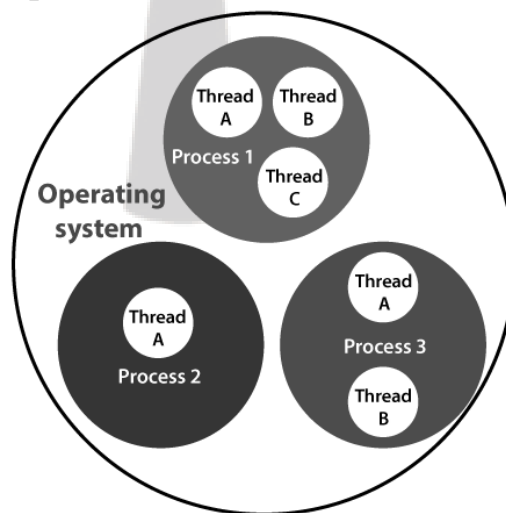
Thread

Thread Concept in Java:

A Thread is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the Thread concept in Java.

All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.



JAVA

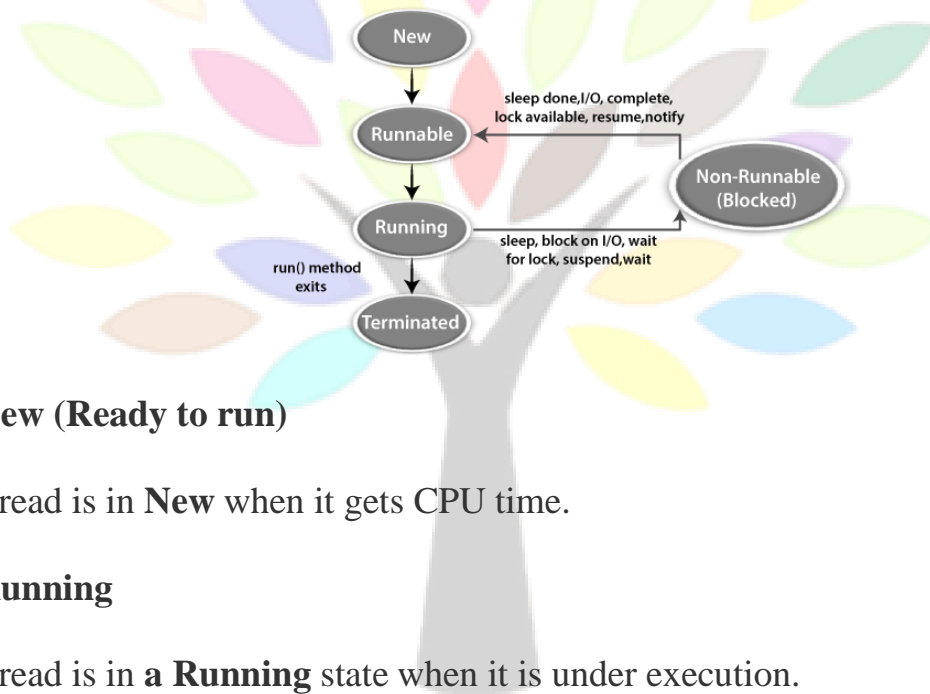
Another benefit of using thread is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as Multithreading.

In a simple way, a Thread is a:

- Feature through which we can perform multiple activities within a single process.
- Lightweight process.
- Series of executed statements.
- Nested sequence of method calls.

Thread Model:

Just like a process, a thread exists in several states. These states are as follows:



1) New (Ready to run)

A thread is in **New** when it gets CPU time.

2) Running

A thread is in a **Running** state when it is under execution.

3) Suspended

A thread is in the **Suspended** state when it is temporarily inactive or under execution.

4) Blocked

JAVA

A thread is in the **Blocked** state when it is waiting for resources.

5) Terminated

A thread comes in this state when at any given time, it halts its execution immediately.

Creating Thread:

A thread is created either by "creating or implementing" the Runnable Interface or by extending the Thread class. These are the only two ways through which we can create a thread.

Let's dive into details of both these way of creating a thread:

Thread Class:

A Thread class has several methods and constructors which allow us to perform various operations on a thread. The Thread class extends the Object class. The Object class implements the **Runnable** interface. The thread class has the following constructors that are used to perform various operations.

- Thread()
- Thread(Runnable, String name)
- Thread(Runnable target)
- Thread(ThreadGroup group, Runnable target, String name)
- Thread(ThreadGroup group, Runnable target)
- Thread(ThreadGroup group, String name)
- Thread(ThreadGroup group, Runnable target, String name, long stackSize)

Thread Class vs Runnable Interface:

If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface

JAVA

Using runnable will give you an object that can be shared amongst multiple threads.

Runnable Interface:

(run() method)

The Runnable interface is required to be implemented by that class whose instances are intended to be executed by a thread. The runnable interface gives us the run() method to perform an action for the thread.

start() method

The method is used for starting a thread that we have newly created. It starts a new thread with a new callstack. After executing the start() method, the thread changes the state from New to Runnable. It executes the run() method when the thread gets the correct time to execute it.

ThreadExample1.java

```
// Implementing runnable interface by extending Thread class
public class ThreadExample1 extends Thread {
    // run() method to perform action for thread.
    public void run()
    {
        int a= 10;
        int b=12;
        int result = a+b;
        System.out.println("Thread started running..");
        System.out.println("Sum of two numbers is: "+ result);
    }
    public static void main( String args[] )
    {
        // Creating instance of the class extend Thread class
        ThreadExample1 t1 = new ThreadExample1();
        //calling start method to execute the run() method of the Thread class
        t1.start();
    }
}
```

Output:

JAVA

```
C:\Windows\System32\cmd.exe

C:\Users\ajet\OneDrive\Desktop\programs>javac ThreadExample1.java

C:\Users\ajet\OneDrive\Desktop\programs>java ThreadExample1
Thread started running..
Sum of two numbers is: 22

C:\Users\ajet\OneDrive\Desktop\programs>
```

Creating thread by implementing the runnable interface:

In Java, we can also create a thread by implementing the runnable interface. The runnable interface provides us both the run() method and the start() method.

Let's take an example to understand how we can create, start and run the thread using the runnable interface.

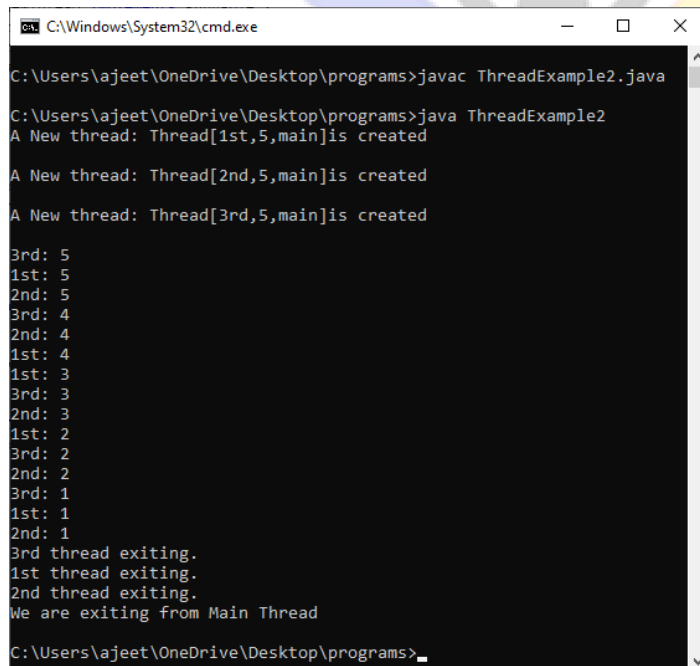
ThreadExample2.java

```
class NewThread implements Runnable {
    String name;
    Thread thread;
    NewThread (String name){
        this.name = name;
        thread = new Thread(this, name);
        System.out.println( "A New thread: " + thread+ "is created\n" );
        thread.start();
    }
    public void run() {
        try {
            for(int j = 5; j > 0; j--) {
                System.out.println(name + ": " + j);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " thread Interrupted");
        }
    }
}
```


JAVA

```
}  
    System.out.println(name + " thread exiting.");  
}  
}  
class ThreadExample2 {  
    public static void main(String args[]) {  
        new NewThread("1st");  
        new NewThread("2nd");  
        new NewThread("3rd");  
        try {  
            Thread.sleep(8000);  
        } catch (InterruptedException excetion) {  
            System.out.println("Inturrption occurs in Main Thread");  
        }  
        System.out.println("We are exiting from Main Thread");  
    }  
}
```

Output:



```
C:\Windows\System32\cmd.exe  
  
C:\Users\ajeet\OneDrive\Desktop\programs>javac ThreadExample2.java  
  
C:\Users\ajeet\OneDrive\Desktop\programs>java ThreadExample2  
A New thread: Thread[1st,5,main]is created  
  
A New thread: Thread[2nd,5,main]is created  
  
A New thread: Thread[3rd,5,main]is created  
  
3rd: 5  
1st: 5  
2nd: 5  
3rd: 4  
2nd: 4  
1st: 4  
1st: 3  
3rd: 3  
2nd: 3  
1st: 2  
3rd: 2  
2nd: 2  
3rd: 1  
1st: 1  
2nd: 1  
3rd thread exiting.  
1st thread exiting.  
2nd thread exiting.  
We are exiting from Main Thread  
  
C:\Users\ajeet\OneDrive\Desktop\programs>_
```

Multithreading:

JAVA

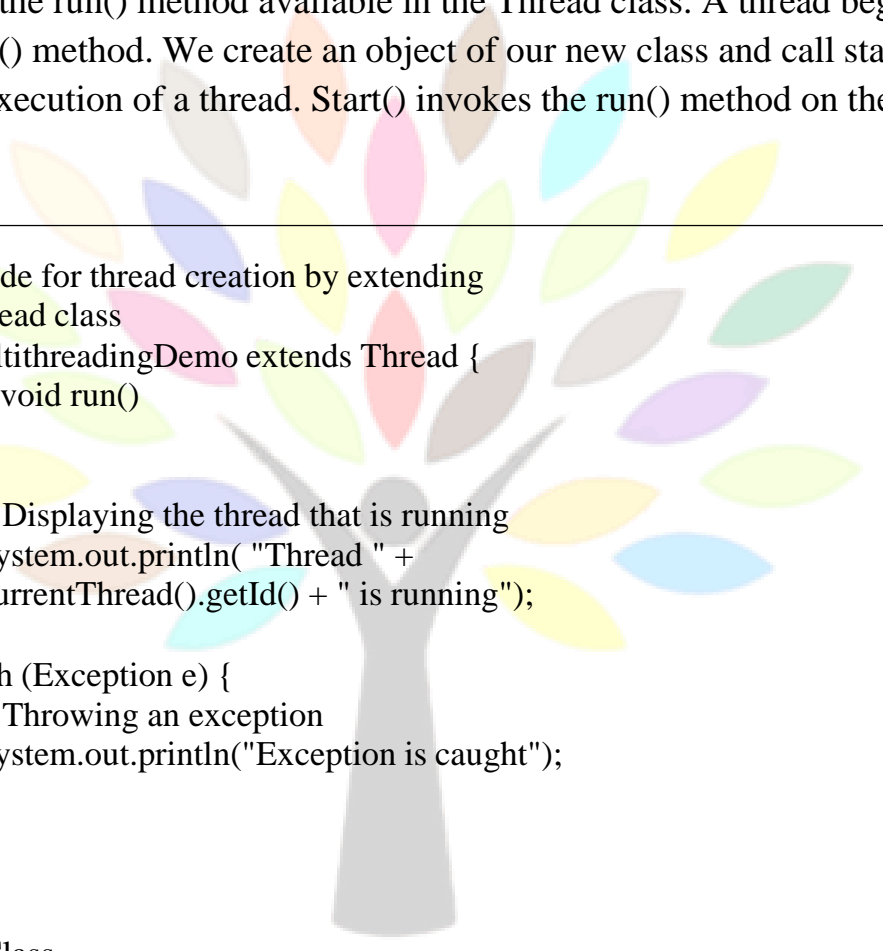
Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms:

- Extending the Thread class
- Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the `java.lang.Thread` class. This class overrides the `run()` method available in the Thread class. A thread begins its life inside `run()` method. We create an object of our new class and call `start()` method to start the execution of a thread. `Start()` invokes the `run()` method on the Thread object.



```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println( "Thread " +
Thread.currentThread().getId() + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
```

JAVA

```
        object.start();
    }
}
```

Output

Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running

Thread creation by implementing the Runnable Interface

We create a new class which implements `java.lang.Runnable` interface and override `run()` method. Then we instantiate a `Thread` object and call `start()` method on this object.

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}
```

JAVA

```
    }  
    }  
}  
  
// Main Class  
class Multithread {  
    public static void main(String[] args)  
    {  
        int n = 8; // Number of threads  
        for (int i = 0; i < n; i++) {  
            Thread object  
                = new Thread(new MultithreadingDemo());  
            object.start();  
        }  
    }  
}
```

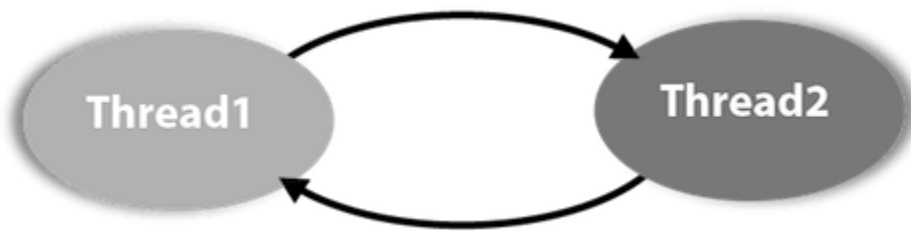
Output

Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running

Deadlock in Java:

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

JAVA



Example of Deadlock in Java

TestDeadlockExample1.java

```
public class TestDeadlockExample1 {
    public static void main(String[] args) {
        final String resource1 = "ratan jaiswal";
        final String resource2 = "vimal jaiswal";
        // t1 tries to lock resource1 then resource2
        Thread t1 = new Thread() {
            public void run() {
                synchronized (resource1) {
                    System.out.println("Thread 1: locked resource 1");

                    try { Thread.sleep(100);} catch (Exception e) {}

                    synchronized (resource2) {
                        System.out.println("Thread 1: locked resource 2");
                    }
                }
            }
        };

        // t2 tries to lock resource2 then resource1
        Thread t2 = new Thread() {
            public void run() {
                synchronized (resource2) {
                    System.out.println("Thread 2: locked resource 2");

                    try { Thread.sleep(100);} catch (Exception e) {}
```

JAVA

```
synchronized (resource1) {  
    System.out.println("Thread 2: locked resource 1");  
}  
}  
};  
t1.start();  
t2.start();  
}  
}
```

Output:

Thread 1: locked resource 1
Thread 2: locked resource 2

Collections in Java

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion. Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (**Arraylist**, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection in Java?

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java?

- It provides readymade architecture.
- It represents a set of classes and interfaces.

JAVA

- It is optional.

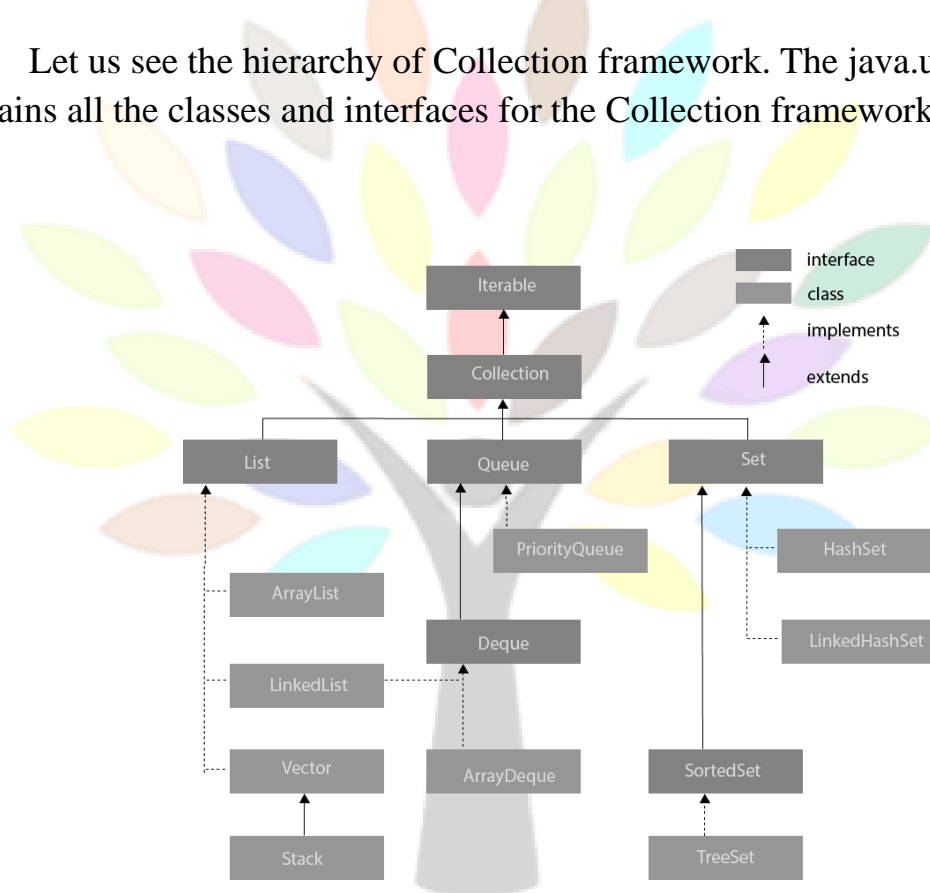
What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

- Interfaces and its implementations, i.e., classes
- Algorithm

Hierarchy of Collection Framework:

Let us see the hierarchy of Collection framework. The java.util package contains all the classes and interfaces for the Collection framework.



Iterator interface:

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

JAVA

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface:

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

`Iterator<T> iterator()`

It returns the iterator over the elements of type T.

Collection Interface:

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface:

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values. List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

JAVA

To instantiate the List interface, we must use :

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

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

ArrayList:

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types.

The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;
class TestJavaCollection1 {
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

JAVA

Output:

Ravi

Vijay

Ravi

Ajay

LinkedList:

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
    public static void main(String args[]){
        LinkedList<String> al=new LinkedList<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");
        Iterator<String> itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

Output:

Ravi

Vijay

JAVA

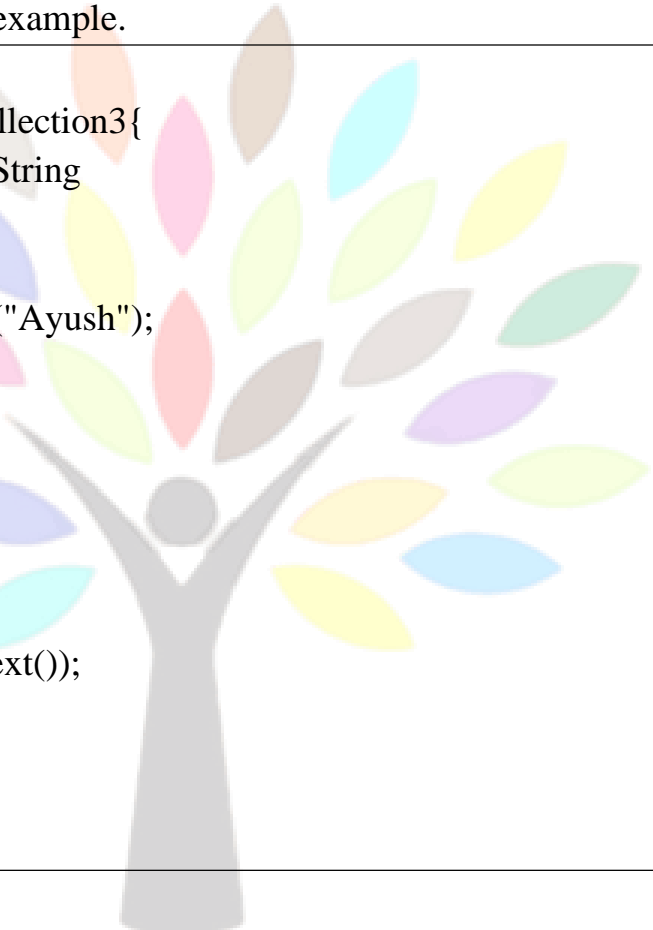
Ravi

Ajay

Vector:

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.



```
import java.util.*;
public class TestJavaCollection3{
public static void main(String
args[]){
Vector<String> v=new
Vector<String>();v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String>
itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

Ayush

Amit

Ashish

Garima

JAVA

Stack:

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
    public static void main(String args[]){
        Stack<String> stack = new Stack<String>();
        stack.push("Ayush");
        stack.push("Garvit");
        stack.push("Amit");
        stack.push("Ashish");
        stack.push("Garima");
        stack.pop();
        Iterator<String> itr=stack.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

Output:

Ayush

Garvit

Amit

Ashish

Queue Interface:

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed.

JAVA

There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

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

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue:

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
    public static void main(String args[]){
        PriorityQueue<String> queue=new PriorityQueue<String>();
        queue.add("Amit Sharma");
        queue.add("Vijay Raj");
        queue.add("JaiShankar");
        queue.add("Raj");
        System.out.println("head:"+queue.element());
        System.out.println("head:"+queue.peek());
        System.out.println("iterating the queue elements:");
        Iterator itr=queue.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
        queue.remove();
        queue.poll();
        System.out.println("after removing two elements:");
    }
}
```

JAVA

```
Iterator<String> itr2=queue.iterator();  
while(itr2.hasNext()){  
System.out.println(itr2.next());  
}  
}  
}
```

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

Deque Interface:

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```
Deque d = new ArrayDeque();
```

JAVA

ArrayDeque:

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

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

Output:

Gautam

Karan

Ajay

Set Interface:

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to

JAVA

store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

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

HashSet:

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

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

Output:

Vijay

JAVA

Ravi

Ajay

LinkedHashSet:

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

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

Output:

Ravi

Vijay

Ajay

JAVA

SortedSet Interface:

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order.

The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

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

TreeSet:

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

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

JAVA

```
}
```

Output:

Ajay

Ravi

Vijay

Generics in Java:

The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics:

There are mainly 3 advantages of generics. They are as follows:

- **Type-safety** : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
- **Type casting is not required**: There is no need to typecast the object. Before Generics, we need to type cast.

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0); //typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0);
```

Compile-Time Checking:

It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time

JAVA

than runtime.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
list.add(32);//Compile Time Error
```

Syntax to use generic collection:

```
ClassOrInterface<Type>
```

Example of Generics in Java:

```
import java.util.*;  
class TestGenerics1 {  
    public static void main(String args[]){  
        ArrayList<String> list=new ArrayList<String>();  
        list.add("rahul");  
        list.add("jai");  
        //list.add(32);//compile time error  
        String s=list.get(1);//type casting is not required  
        System.out.println("element is: "+s);  
        Iterator<String> itr=list.iterator();  
        while(itr.hasNext()){  
            System.out.println(itr.next());  
        }  
    }  
}
```

Output:

element is: jai Rahul jai

What is Java IDE?

Java IDE (Integrated Development Environment) is a software application that enables users to write and debug Java programs more easily.

Most IDEs have features such as syntax highlighting and code completion that helps users to code more easily.

Usually, Java IDEs include a code editor, a compiler, a debugger, and an interpreter that the developer may access via a single graphical user interface.

JAVA

Java IDEs also provide language-specific elements such as Maven, Ant building tools, Junit, and TestNG for testing.

The java IDE or Integrated Development Environment provides considerable support for the application development process. Through using them, we can save time and effort and set up a standard development process for the team or company.

Eclipse, NetBeans, IntelliJ IDEA, and many other IDE's are most popular in the Java IDE's that can be used according to our requirements. In this topic, we will discuss the best Java IDE's that are used by the users.

Best Java IDEs:

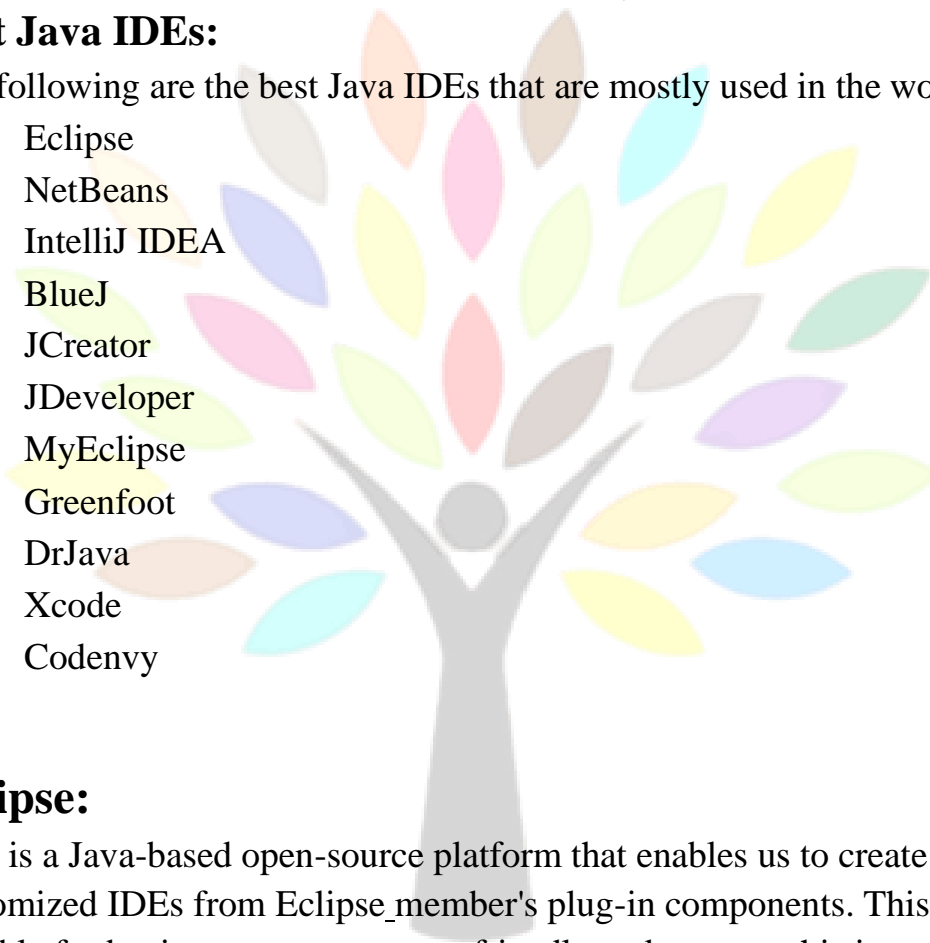
The following are the best Java IDEs that are mostly used in the world:

- Eclipse
- NetBeans
- IntelliJ IDEA
- BlueJ
- JCreator
- JDeveloper
- MyEclipse
- Greenfoot
- DrJava
- Xcode
- Codenvy

Eclipse:

It is a Java-based open-source platform that enables us to create highly customized IDEs from Eclipse_member's plug-in components. This platform is also suitable for beginners to create user-friendly and more sophisticated applications. It contains many plugins that enable developers to develop and test code written in different languages. Some of Eclipse's features are as follows:

- Eclipse provides powerful tools for different software development processes, such as charting, reporting, checking, etc. so that Java developers can build the application as quickly as possible.

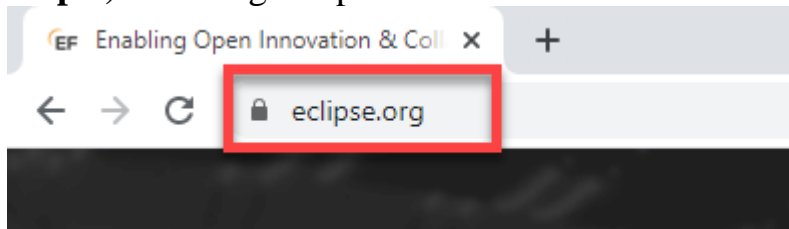


JAVA

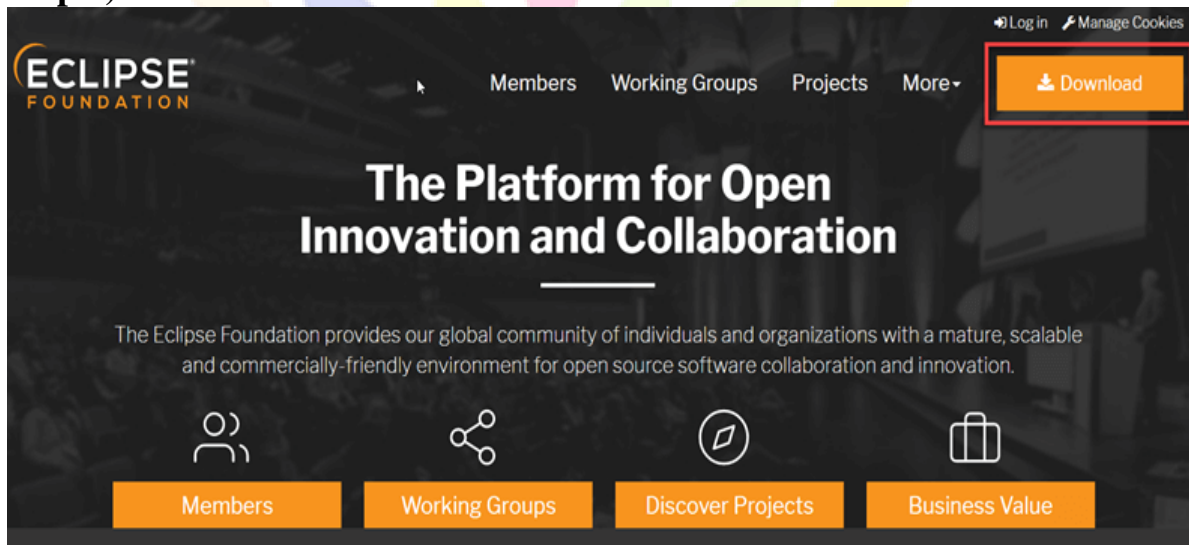
- Eclipse can be used on platforms such as MacOS, Linux, Windows, and Solaris.
- Eclipse could also make several mathematical documents with LaTeX using both the TeXlipse plug-in and Mathematica software packages.

Following is a step by step guide to download and install Eclipse IDE:

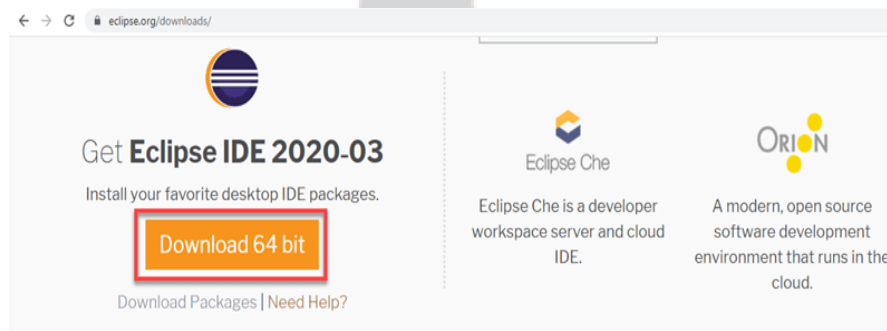
Step 1) Installing Eclipse



Step 2) Click on “Download” button.

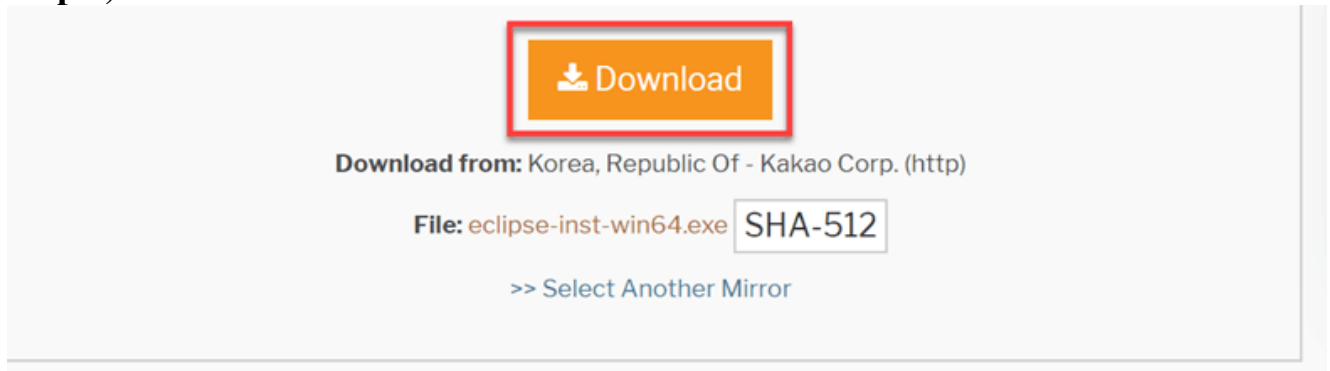


Step 3) Click on “Download 64 bit” button



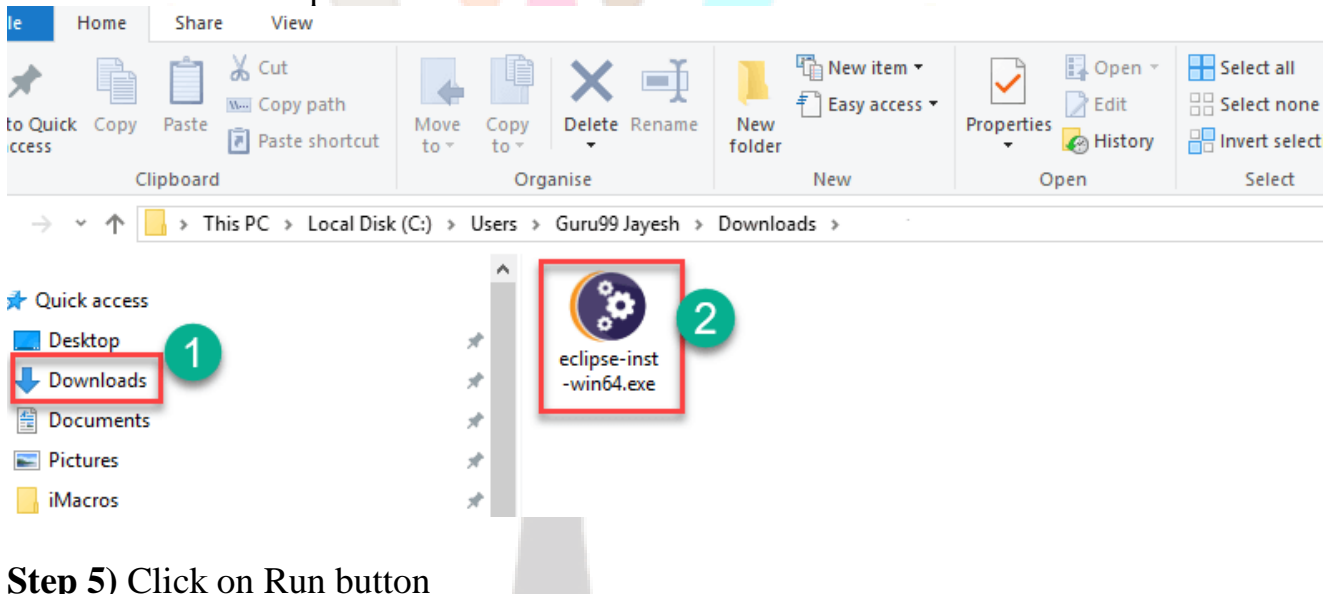
JAVA

Step 4) Click on “Download” button

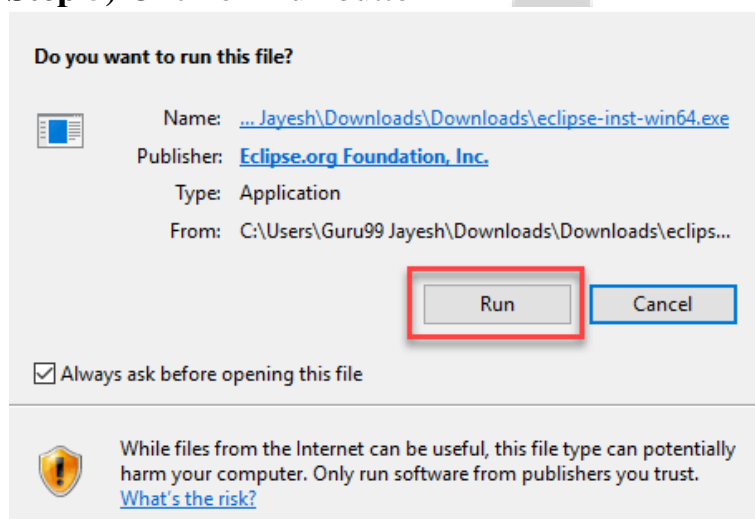


Step 4) Install Eclipse.

1. Click on “downloads” in Windows file explorer.
2. Click on “eclipse-inst-win64.exe” file.

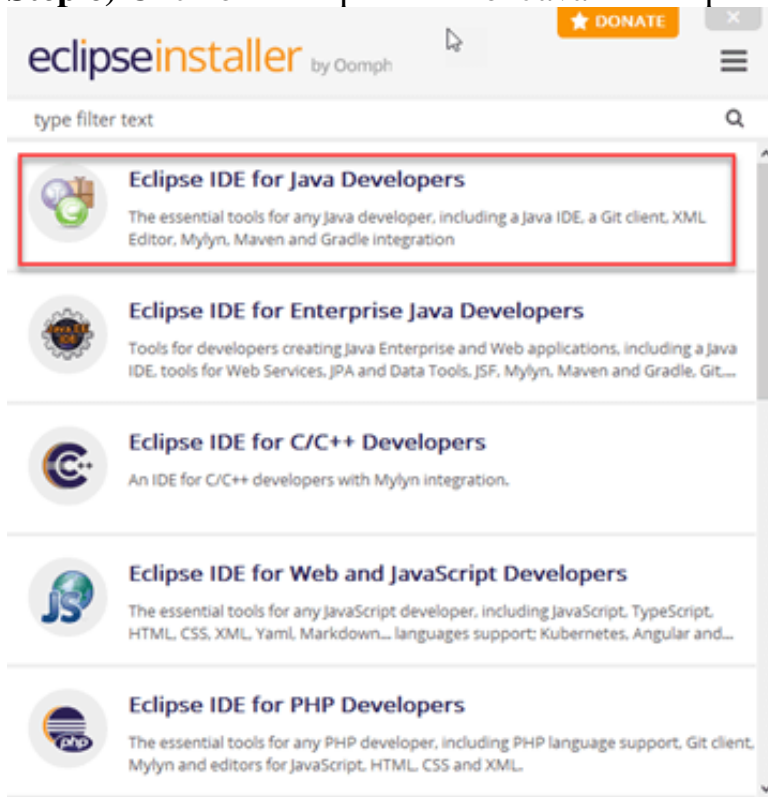


Step 5) Click on Run button

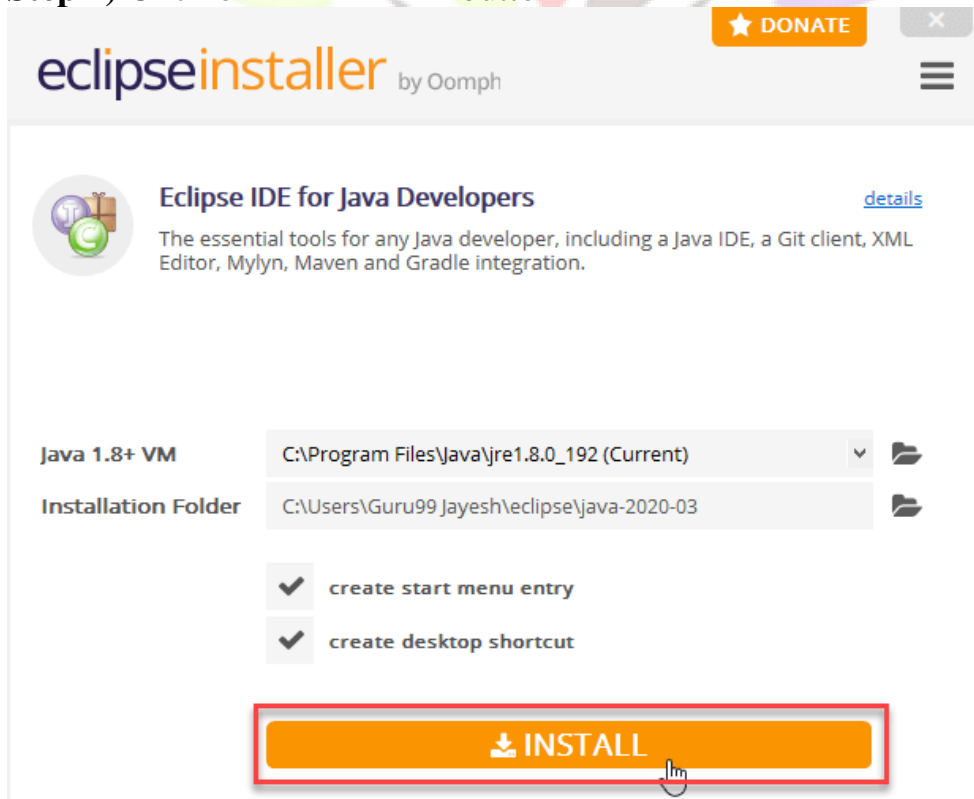


JAVA

Step 6) Click on “Eclipse IDE for Java Developers”



Step 7) Click on “INSTALL” button

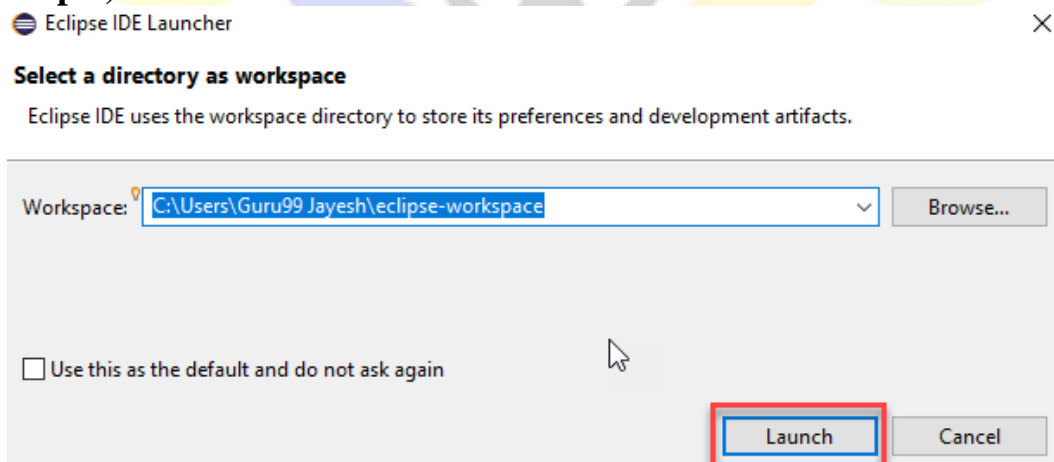


JAVA

Step 8) Click on “LAUNCH” button.

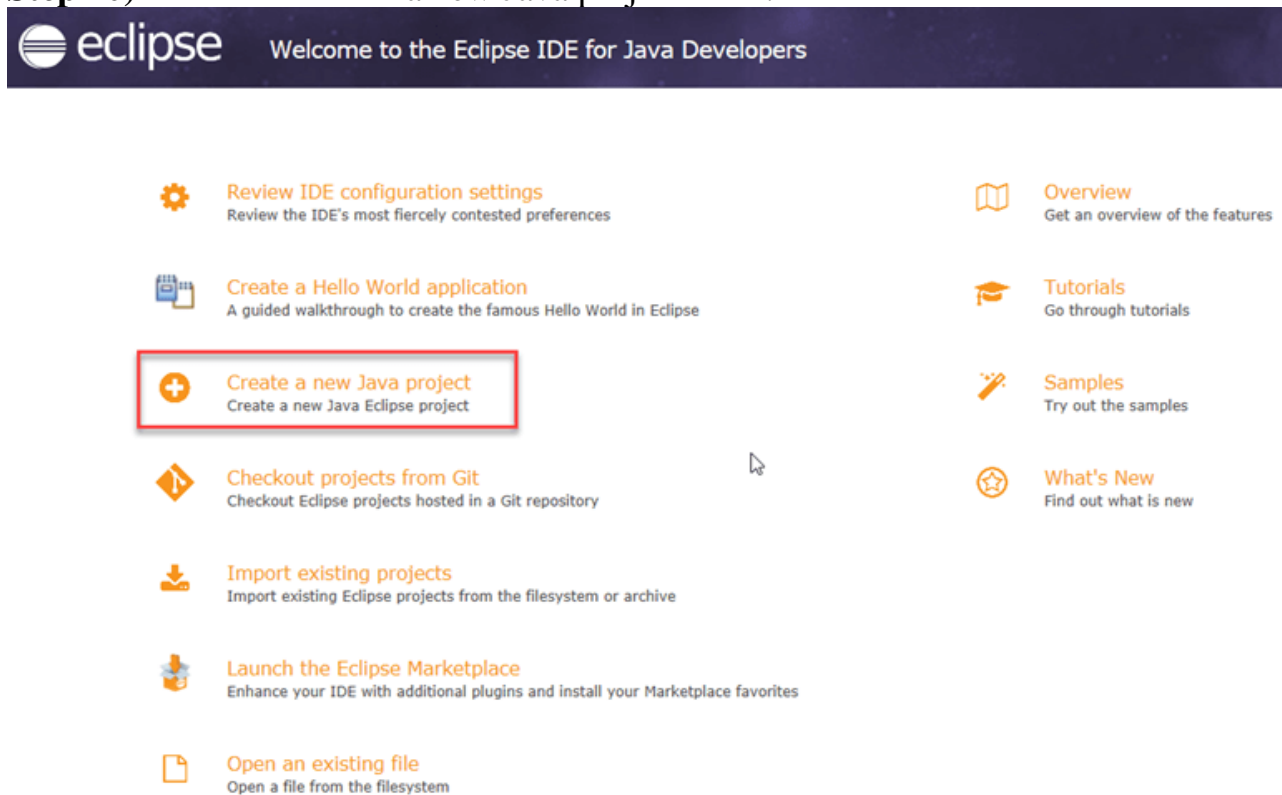


Step 9) Click on “Launch” button.



JAVA

Step 10) Click on “Create a new Java project” link.



Step 11) Create a new Java Project

1. Write project name.
2. Click on “Finish button”.



JAVA

Create a Java Project

Create a Java project in the workspace or in an external location.



1

Project name: HelloWorld

☒ Use default location

Location: C:\Users\Guru99 Jayesh\eclipse-workspace\HelloWorld [Browse...](#)

JRE

☐ Use an execution environment JRE: JavaSE-1.8

☐ Use a project specific JRE: jre1.8.0_192

☐ Use default JRE 'jre1.8.0_192' and workspace compiler preferences [Configure JREs...](#)

Project layout

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files [Configure default...](#)

Working sets

☐ Add project to working sets [New...](#)

Working sets: [Select...](#)

The wizard will automatically configure the JRE and the project layout based on the existing source.

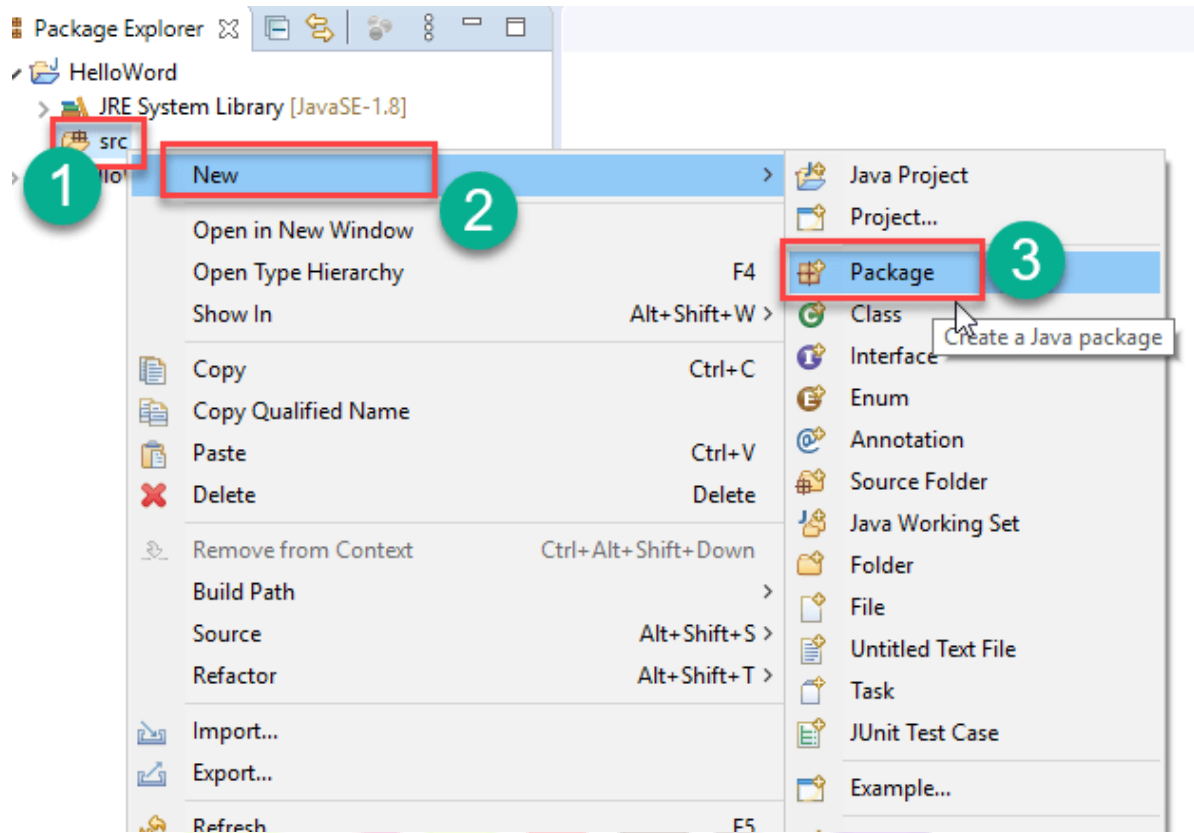
2

[? < Back](#) [Next >](#) [Finish](#) [Cancel](#)

Step 12) Create Java Package.

1. Goto "src".
2. Click on "New".
3. Click on "Package".

JAVA



Step 13) Writing package name.

1. Write name of the package
2. Click on Finish button.

JAVA

Java Package

Create a new Java package.



Creates folders corresponding to packages.

Source folder:

Name: **1**

☒ Create package-info.java

☐ Generate comments (configure templates and default value [here](#))

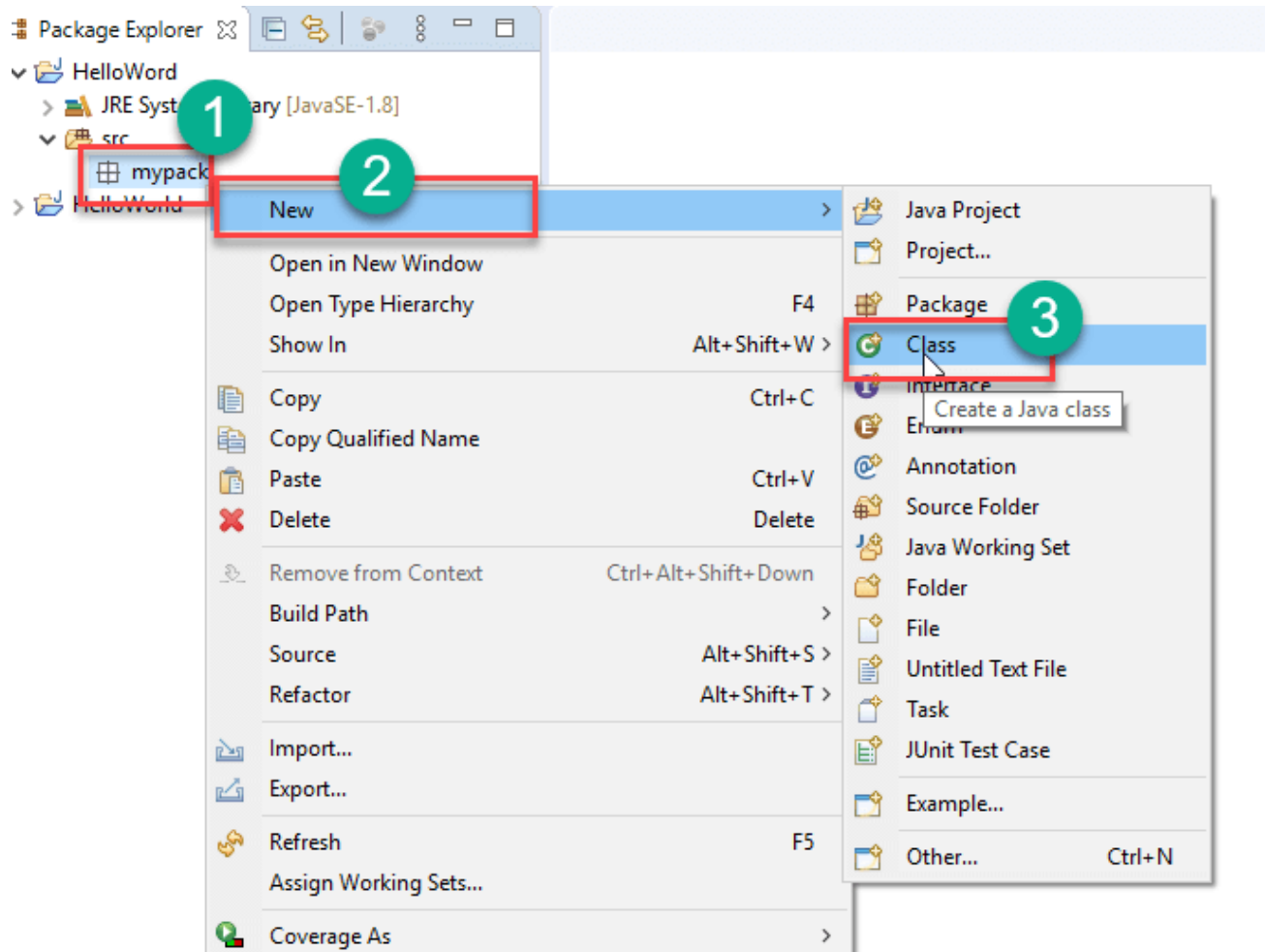


2

Step 14) Creating Java Class

1. Click on package you have created.
2. Click on “New”.
3. Click on “Class”.

JAVA



Step 15) Defining Java Class.

1. Write class name
2. Click on “public static void main (String[] args)” checkbox.
3. Click on “Finish” button.

JAVA

Java Class

Create a new Java class.



Source folder: HelloWorld/src Browse...

Package: mypack Browse...

☐ Enclosing type: Browse...

Name: HelloWorld 1

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

Which method stubs would you like to create? 2

☒ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

3

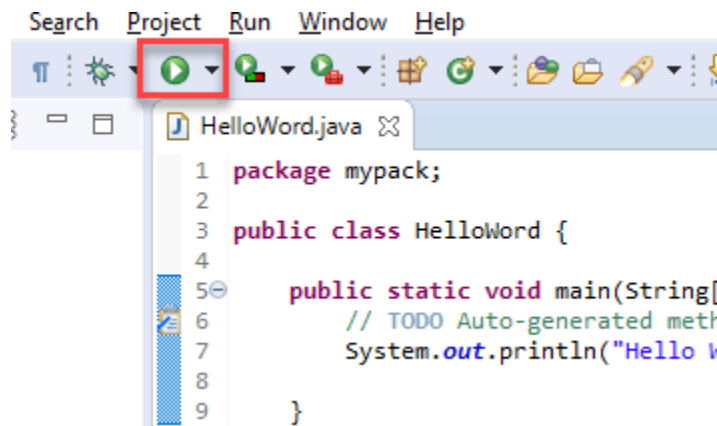
Finish Cancel

Helloword.java file will be created as shown below:

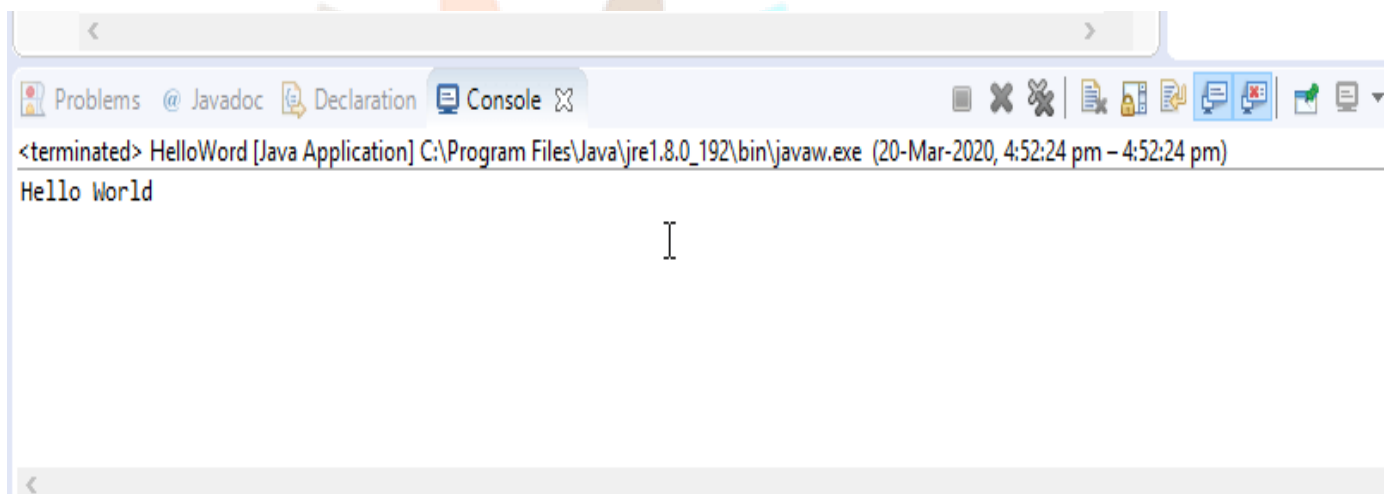
```
1 package mypack;
2
3 public class HelloWorld {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         System.out.println("Hello World");
8     }
9 }
10
11 }
12
```

Step 16) Click on “Run” button.

JAVA



Output will be displayed as shown below.



JAVA

JDBC

- JDBC or Java Database Connectivity is a Java API to connect and execute the query with the database.
- It is a specification from Sun microsystems that provides a standard abstraction (API or Protocol) for java applications to communicate with various databases.
- It provides the language with java database connectivity standards. It is used to write programs required to access databases.
- JDBC, along with the database driver, can access databases and spreadsheets. The enterprise data stored in a relational database (RDB) can be accessed with the help of JDBC APIs.

Definition of JDBC (Java Database Connectivity):

JDBC is an API (Application programming interface) used in java programming to interact with databases. The classes and interfaces of JDBC allow the application to send requests made by users to the specified database.

Purpose of JDBC:

Enterprise applications created using the JAVA EE technology need to interact with databases to store application-specific information. So, interacting with a database requires efficient database connectivity, which can be achieved by using the ODBC(Open database connectivity) driver. This driver is used with JDBC to interact or communicate with various kinds of databases such as Oracle, MS Access, Mysql, and SQL server database.

Components of JDBC

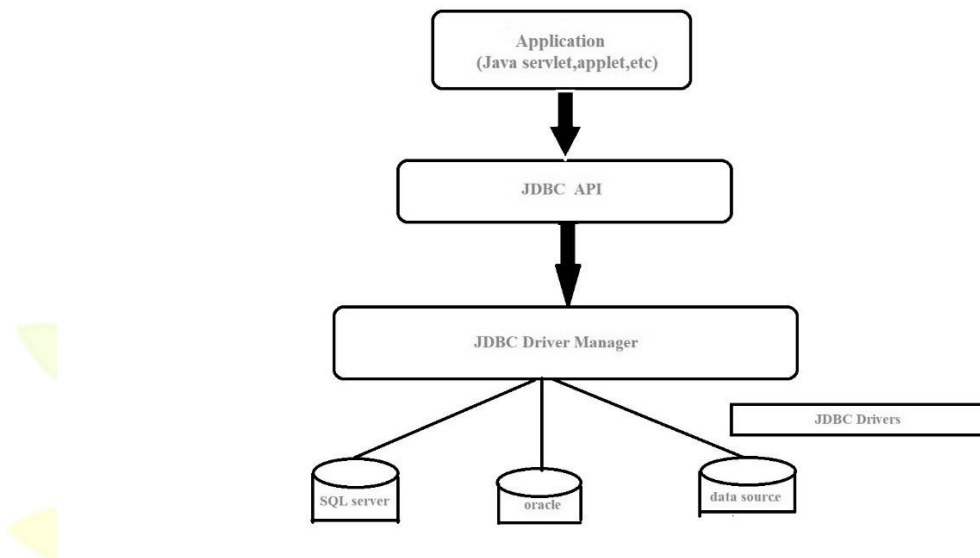
There are generally four main components of JDBC through which it can interact with a database. They are as mentioned below:

1. **JDBC API:** It provides various methods and interfaces for easy communication with the database. It provides two packages as follows, which contain the java SE and Java EE platforms to exhibit WORA(write once run anywhere) capabilities.
`java.sql.*;`
2. It also provides a standard to connect a database to a client application.
3. **JDBC Driver manager:** It loads a database-specific driver in an application to establish a connection with a database. It is used to make a database-specific call to the database to process the user request.

JAVA

4. **JDBC Test suite:** It is used to test the operation (such as insertion, deletion, updation) being performed by JDBC Drivers.
5. **JDBC-ODBC Bridge Drivers:** It connects database drivers to the database. This bridge translates the JDBC method call to the ODBC function call. It makes use of the sun.jdbc.odbc package which includes a native library to access ODBC characteristics.

Architecture of JDBC:



Description:

Application: It is a java applet or a servlet that communicates with a data source.

The JDBC API: The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows:

DriverManager: It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

JDBC drivers: To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

JDBC Drivers:

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

JAVA

- Type-1 driver or JDBC-ODBC bridge driver
- Type-2 driver or Native-API driver
- Type-3 driver or Network Protocol driver
- Type-4 driver or Thin driver

Types of JDBC Architecture (2-tier and 3-tier):

The JDBC architecture consists of two-tier and three-tier processing models to access a database. They are as described below:

Two-tier model: A java application communicates directly to the data source. The JDBC driver enables the communication between the application and the data source. When a user sends a query to the data source, the answers for those queries are sent back to the user in the form of results.

The data source can be located on a different machine on a network to which a user is connected. This is known as a client/server configuration, where the user's machine acts as a client, and the machine has the data source running acts as the server.

Three-tier model: In this, the user's queries are sent to middle-tier services, from which the commands are again sent to the data source. The results are sent back to the middle tier, and from there to the user.

This type of model is found very useful by management information system directors.

Interfaces of JDBC API:

A list of popular interfaces of JDBC API is given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

Classes of JDBC API:

A list of popular classes of JDBC API is given below:

- DriverManager class
- Blob class

JAVA

- Clob class
- Types class

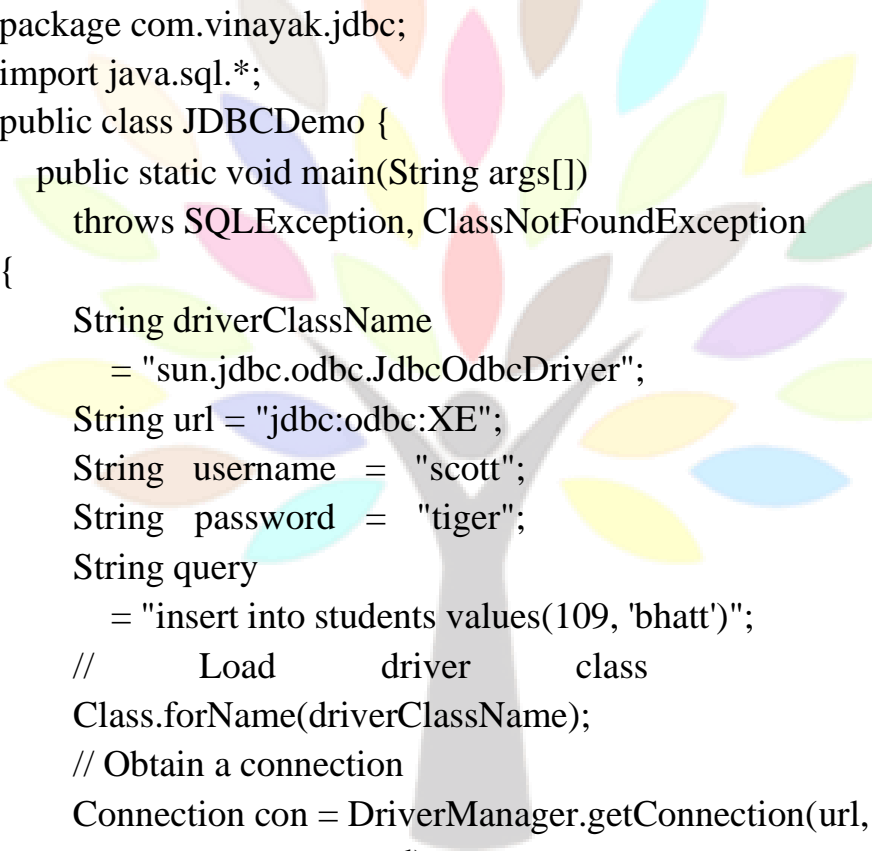
Working of JDBC:

Java application that needs to communicate with the database has to be programmed using JDBC API.

JDBC Driver supporting data sources such as Oracle and SQL server has to be added in java application for JDBC support which can be done dynamically at run time.

This JDBC driver intelligently communicates the respective data source.

Creating a simple JDBC application:



```
package com.vinayak.jdbc;
import java.sql.*;
public class JDBCdemo {
    public static void main(String args[])
        throws SQLException, ClassNotFoundException
    {
        String driverClassName
            = "sun.jdbc.odbc.JdbcOdbcDriver";
        String url = "jdbc:odbc:XE";
        String username = "scott";
        String password = "tiger";
        String query
            = "insert into students values(109, 'bhatt')";
        // Load driver class
        Class.forName(driverClassName);
        // Obtain a connection
        Connection con = DriverManager.getConnection(url,
            username, password);
        // Obtain a statement
        Statement st = con.createStatement();
        // Execute the query
        int count = st.executeUpdate(query);
        System.out.println(
            "number of rows affected by this query= "
            + count);
    }
}
```

JAVA

```
// Closing the connection as per the
// requirement with connection is completed con.close();
}
} // class
```

The above example demonstrates the basic steps to access a database using JDBC. The application uses the JDBC-ODBC bridge driver to connect to the database. You must import java.sql package to provide basic SQL functionality and use the classes of the package.

Frame:

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){
First f=new First();
}
}
```

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

JAVA

Output:

