**Core Java**

**Introduction of Java**

Author - **James Gosling**

Vendor - **Sun Microsystems (Oracle Corporation)**

Initial Name - Oak

After Name – Green Project

Present name – Java

Beta version - 1995

Initial version - 1.0 in 1996

Present version - 20

Stable version - Java 8 and Java 11 (Companies are using)

Type of the software - open source (e.g. java, android, selenium etc.)

extension - .java, .class, .jar

Operating System - Multiple operating System

Implementation Language - c, c++

Symbol: coffee cup with saucer (Hot coffee cup with raising stream)

SUN: Stanford Universally Network

Slogan/Motto: WORA (write once run anywhere)

(Write on any operating system and run on any operating system.)

Compilation: java compiler (Javac – Check the syntax)

Execution: JVM (java virtual machine)

Java website: **https://www.oracle.com/java/**

**Importance of core java**:

According to the SUN 3 billion devices run on the java language only.

✓ Java is used to develop Desktop Applications such as Media Player, Antivirus etc.

✓ Java is Used to Develop Web Applications such as durgasoft.com, irctc.co.in etc.

✓ Java is Used to Develop Enterprise Application such as Banking applications.

✓ Java is Used to Develop Mobile Applications.

✓ Java is Used to Develop Embedded System.

✓ Java is Used to Develop Smartcards.

✓ Java is Used to Develop Robotics.

✓ Java is used to Develop Games…. etc.

**Parts of java**

Core java, advance java are not official terms,

As per the sun micro systems standard the java divided into 3 parts,

a. J2SE/JSE - Java 2 standard edition : standalone applications

b. J2EE/JEE - Java 2 enterprise edition : web application, enterprise application

c. J2ME/JME - Java 2 Micro edition : Mobile application

**Core java belongs to J2SE….**

Java is object-oriented programming language deals with classes and objects.

Java supports oops features like

encapsulations,

inheritance,

polymorphism,

abstraction.

e.g. Java, Python, .Net, Ruby, Kotlin CPP, Dart, Julia etc.

**Note**: All Object-oriented Programming languages the features are same but syntax is different.

**Java is object-oriented programming language…**

**Learning process of java:**

**Level-1**: core java, Adv java

**Level-2**: Spring, Spring Boot, Security......

Web-Services (SOAP---- Xml, REST ----- Json)

**Installation:**

1. **Install Java on Windows**

Website: [**https://www.oracle.com/java/**](https://www.oracle.com/java/)

[**https://www.guru99.com/install-java.html**](https://www.guru99.com/install-java.html)

1. **Install IntelliJ on windows**:

Website:[**https://www.jetbrains.com/idea/download/#section=windows**](https://www.jetbrains.com/idea/download/#section=windows)

**Features of Java**

1. **Open Source and Simple**

Open source means it’s free. We don’t have to pay for license. Java is a Simple language. Easy to understand. All difficult concepts in C++ like Pointers, Operator Overloading etc. are removed from java or made it in simpler ways.

1. **Object Oriented**

After 1.7 java has become purely object-oriented language. After introducing Wrapper classes, it has become purely oops. In Java everything is in the form of an object. Java can be easily extended as it is based on object model. Following are some OOP s concepts.

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

6. Polymorphism

1. **Robust and Secure**

Software which we are creating by using Java are robust means It has high possibility of less errors. Java makes an effort to eliminate errors by compile time error checking and runtime checking. Java Provides Exception Handling concepts which handles runtime errors by which your programs will not collapse or stop. In C or C++ there are pointers which directly defines the address. If u get address of anything u can hack the data. If u get IP address, then u can easily hack the data. In java we don’t have such kind of concepts like pointers. That’s reason it is more secure.

1. **Multi-threading**

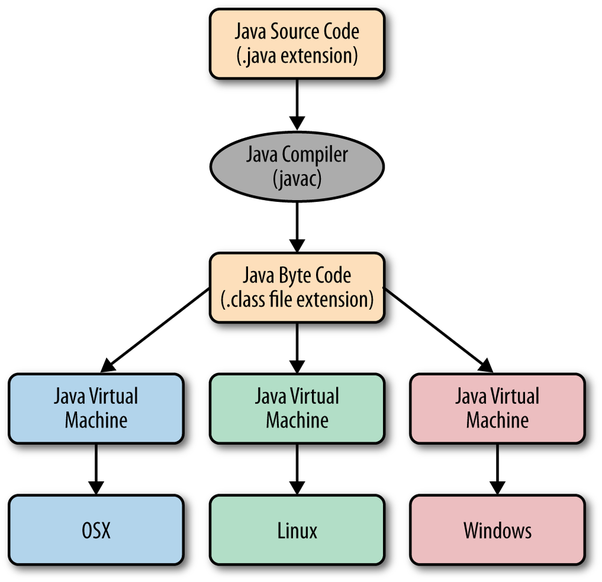
In Operating System we use concept of Multitasking. In Programming Language, we use concept of multithreading. In Programming language main problem is memory management. In multithreading we can share or utilize same memory or common resource and access same resource by multiple users.

1. **High Performance**

We are writing the code once and converting that into .class file. And same .class file we are using everywhere.

1. **Platform Independent & Portable.**

We can run any java program on any machine or any operating system. We write java source code (High level Language), then java compiler translates it into bytecode (intermediate language) then the JVM which is platform dependent translates it into machine code (Low level Language)



**JDK**:

The JDK also called Java Development Kit is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.

**JVM**:

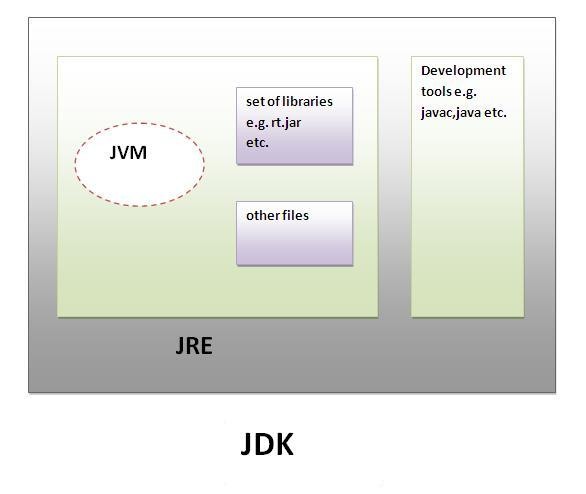
Java virtual Machine control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap. The JVM doesn't understand Java type, that's why you compile your \*.java files to obtain \*.class files that contain the bytecodes understandable by the JVM.

**JRE**:

The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language.

**Internal Structure of JDK.**

**Java Development Kit**



**Java Compiler**

**Java Program**

**Java Interpreter**

**Machine Code**

**Virtual Machine**

**Bytecode**

**Source Code**

**Bytecode**

**Virtual Machine**

**Real Machine**

**Process of Compilation**

**Process of converting source code to machine code**

**Data Types**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Data Types in Java** |  |  | | --- | | **Primitive** |  |  | | --- | | **Non-Primitive (Derived)** |  |  | | --- | | **Numeric** |  |  | | --- | | **Non-Numeric** |  |  | | --- | | **Classes** |  |  | | --- | | **Interfaces** |  |  | | --- | | **Integer** |  |  | | --- | | **Floating Point** |  |  | | --- | | **Character** |  |  | | --- | | **Boolean** |  |  | | --- | | **Arrays** | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Primitive Type** | **Size** | **Minimum Value** | **Maximum Value** | **Wrapper Type** |
| Char | 16-bit | Unicode 0 | Unicode 216-1 | Character |
| Byte | 8-bit | -128 | +127 | Byte |
| Short | 16-bit | -215 (-32,768) | +215-1 (32,767) | Short |
| Int | 32-bit | -231 (-2,147,483,648) | +231-1 (2,147,483,647) | Integer |
| Long | 64-bit | -263 (9,223,372,036,854,775,88) | +263-1 (9,223,372,036,854,775,807) | Long |
| Float | 32-bit | Approx range 1.4e-045 to 3.4e+038 | | Float |
| Double | 64-bit | Approx range 4.9e-324 to 1.8e+308 | | Double |
| Boolean | 1-bit | true or false | | Boolean |

**class** PrimitiveDemo

{

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

byte b =100;

short s =123;

int v = 123543;

int calc = -9876345;

long amountVal = 1234567891L;

float intrestRate = 12.25f;

double sineVal = 12345.234;

boolean flag = true;

boolean val = false;

char ch1 = 88; // code for X

char ch2 = 'Y';

System.out.println("byte Value = "+ b);

System.out.println("short Value = "+ s);

System.out.println("int Value = "+ v);

System.out.println("int second Value = "+ calc);

System.out.println("long Value = "+ amount Val);

System.out.println("float Value = "+ interstate);

System.out.println("double Value = "+ sineVal);

System.out.println("boolean Value = "+ flag);

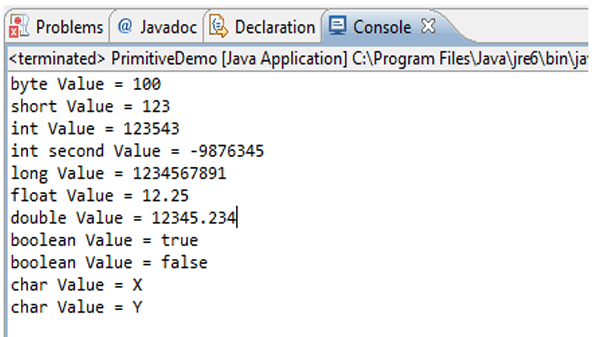
System.out.println("boolean Value = "+ val);

System.out.println("char Value = "+ ch1);

System.out.println("char Value = "+ ch2);

}

}

****

**First Java Application**

**The HelloWorld application.**

**class** HelloWorld

{

**public static void main** (String args[])

{

System.out.println(“Hello World!!”);

// public – access modifier

// static – Keyword

// Void --- Return type

// main -- Method Name

// (String[] args) --- parameters

}

}

**Class Declaration**

The first line public class HelloWorld declares a class, which is an Object-Oriented construct. As stated, earlier Java is true Object-Oriented language and therefore, everything must be placed inside a class. Class is a keyword and declares that a new class definition follows.

Opening Brace

Every class definition in Java begins with an opening brace “{“and ends with a matching closing brace “}”, appearing in the last line in the example.

The main () method

Every java application program must include the main () method. This is starting point for the interpreter to begin the execution of the program. A Java application can have any number of classes but only one of them must include a main method to initiate the execution.

Public: The Keyword public is an access specifier that declares the main method as unprotected and therefore making it to accessible to all other classes.

Static: The keyword static which declares this method as one that belongs to the entire Class and not a part of any Objects of the class.

**Escape Sequence Characters**

|  |  |
| --- | --- |
| ***Sequence*** | ***Purpose*** |
| \b | Backspace |
| \t | Horizontal tab |
| \n | Line feed |

**Example**:

System.*out*.println("Hello");  
System.*out*.println("first \"java\" Program");  
System.*out*.println("first \'java\' Program");  
System.*out*.println("first \\java\\ Program");  
System.*out*.println("first\tjava\tProgram");  
System.*out*.println("first \bjava \bProgram");  
System.*out*.println("first \njava\n Program");

**Comments in Java**

Java supports three types of comment delimiters-the traditional /\*

and \*/ of C, the // of C++, and a new variant that starts with /\*\* and

ends with \*/.

The /\* and \*/ delimiters are used to enclose text that is to be treated as a

comment by the compiler. These delimiters are useful when you want to

designate a lengthy piece of code as a comment, as shown in the

following:

/\* This is a comment that will span multiple source code lines. \*/

The // comment delimiter is borrowed from C++ and is used to indicate that

the rest of the line is to be treated as a comment by the Java compiler.

This type of comment delimiter is particularly useful for adding comments

adjacent to lines of code, as shown in the following:

Date today = new Date();      // create an object with today's date

System.out.println(today);   // display the date

Finally, the /\*\* and \*/ delimiters are new to Java and are used to indicate that the enclosed text

is to be treated as a comment by the compiler, but that the text is also part of the automatic

class documentation that can be generated using Javadoc

**Java comment delimiters.**

|  |  |  |
| --- | --- | --- |
| *Start* | *End* | *Purpose* |
| /\* | \*/ | The enclosed text is treated as a comment. |
| // | (none) | The rest of the line is treated as a comment. |
| /\*\* | \*/ | The enclosed text is treated as a comment by the compiler but is used by Javadoc to automatically generate documentation. |

**Operators and Expressions**

1. **Arithmetic Operators:**

+🡪 addition

-🡪subtraction

\*🡪multiplication

/🡪 division

%🡪 modulus

**Example:**

int a=20;  
int b=3;  
System.*out*.println("Addition of a and b: " + (a+b));  
System.*out*.println("Subtraction of a and b: " + (a-b));  
System.*out*.println("Multiplication of a and b: " + (a\*b));  
System.*out*.println("division of a and b: " + (a/b));  
System.*out*.println("mod of a and b: " + (a%b));

1. **Relational Operators (Always Returns Boolean Value)**

==🡪 Comparison

<🡪 Less than

>🡪 Greater than

<=🡪 Less than Equal to

>=🡪 Greater than equal to

! =🡪 not equal to

**Example:**

boolean x = true;  
boolean y= false;  
System.*out*.println(x && y); //false  
System.*out*.println(x || y); // true  
System.*out*.println(!x); // false  
System.*out*.println(!y); // true

1. **The Conditional-And and Conditional-Or Operators**

There are two primary Boolean operators:

* Logical-AND: &&

Logical-OR: ||

**Example:**

|  |  |  |  |
| --- | --- | --- | --- |
| When A is | And when B is | (A && B) | (A || B) |
| false | false | false | false |
| false | true | false | true |
| true | false | false | true |
| true | true | true | true |

1. **Increment & Decrement Operator**

The increment and decrement operators are used with one variable (they are known as unary operators):

++ increment operator

-- decrement operator

For instance, the increment operator (++) adds one to the operand, as shown in the next line of code:

x++;

is the same as

x+=1;

The increment and decrement operators behave slightly differently based on the side of the operand they are placed on. If the operand is placed before the operator (for example, ++x), the increment occurs before the value is taken for the expression. So, in the following code fragment, the result of y is 6:

int x=5;int y=++x; // y=6 x=6

If the operator appears after the operand, the addition occurs after the value is taken. So y is 5 as shown in the next code fragment. Notice that in both examples, x is 6 at the end of the fragment.

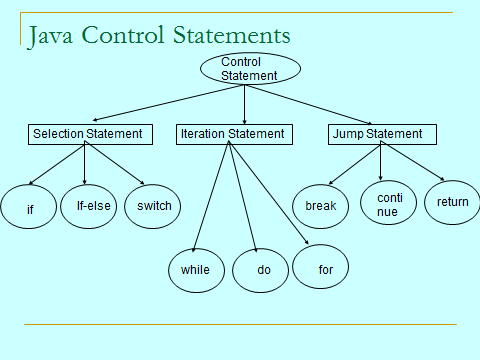
int x=5;int y = x++; //y=5 x=6

Similarly, the decrement operator (--) subtracts one from the operand, and the timing of this is in relation to the evaluation of the expression that it occurs in.

**Example:**

int i = 10;  
i++; //i=i+1 i+=1;  
System.*out*.println(i);  
  
int j = 100;  
j--; //j=j-1 j-=1;  
System.*out*.println(j);  
  
int x=5;int y=++x; // y=6 x=6  
System.*out*.println(x);  
System.*out*.println(y);  
  
int a=20;int b = a++; //a=21 b=20  
System.*out*.println(a);  
System.*out*.println(b);

**Java Control Statement**



**Types of Selection Statement**

* If Statement
* If – Else statement
* If- Else-if statement
* Switch Statement

**Selection Statement:** These select one of several control flows. There are three types of selection statement in Java: if, if-else, and switch.

**If statement:**

The if statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is a two-way decision statement and is used in conjunction with an expression. The general form is:

**If (test expression)**

**{**

**Statement-block;**

**}**

**Statement-x;**

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is ‘true’ or ‘false’. It transfers the control to a particular statement.

If the statement is true then the Statement block will be executed; otherwise, the statement-block will be skipped and the execution will jump to the statement-x. It should be remembered that when the condition is true both the statement-block and statement-x are executed in sequence

**If-else statement:**

**if (test expression)**

**{**

**True-Block Statement(s);**

**}**

**Else**

**{**

**False-Block statement(s);**

**}**

**Statement-x;**

If the test expression is true, then the true-block statement(s) executed immediately following to the if statement, are executed; otherwise, the false statement(s) will be executed, not both. In both the cases, the control is transferred subsequently to the statement-x.

**Nesting of If—else Statement:**

**If (test condition1)**

**{**

**if (test condition2)**

**{**

**Statement-1;**

**}**

**else**

**{**

**Statement-2;**

**}**

**}**

**else**

**{**

**Statement-3;**

**}**

**Statement-x;**

If the condition-1 is false, the statement-3 will be executed; otherwise, it continues to perform the second test. If the condition-2 is true, the statement-1 will be evaluated; otherwise, statement-2 will be evaluated and then control is transferred to the statement-x.

e.g :

**class** Demo

{

**public static void main**(String [] args)

{

System.out.println("Hello");

int a=7;

if(a>9)

{

if(a>0 && a<11)

{

System.out.println("No range between 1 to 10");

}

else

{

System.out.println("No range not between 1 to 10");

}

}

Else

{

System.out.println("No a is not > 9");

}

System.out.println("Hello!!");

}

}

**Switch Statement:**

The Java switch statement is ideal for testing a single expression against a series of possible values and executing the code associated with the matching case statement.

Switch(expression)

{

Case value-1:

block-1;

break;

Case value-2:

block-2;

break;

……

……

default:

default-block;

break;

}

Statement-x;

Switch Case statement is mostly used with [break statement](https://beginnersbook.com/2017/08/java-break-statement/) even though it is optional. We will first see an example without break statement and then we will discuss switch case with break.

## A Simple Switch Case Example

public class SwitchCaseExample1

{

public static void main(String args[])

{

int num=2;

switch(num+2)

{

case 1:

System.out.println("Case1: Value is: "+num);

case 2:

System.out.println("Case2: Value is: "+num);

case 3:

System.out.println("Case3: Value is: "+num);

default:

System.out.println("Default: Value is: "+num);

}

}

}

**Output:**

Default: Value is: 2

**Explanation:**

In switch I gave an expression, you can give variable also. I gave num+2, where num value is 2 and after addition the expression resulted 4. Since there is no case defined with value 4 the default case got executed. This is why we should use default in switch case, so that if there is no catch that matches the condition, the default block gets executed.

## Break statement in Switch Case

Break statement is optional in switch case but you would use it almost every time you deal with switch case. Before we discuss about break statement, Let’s have a look at the example below where I am not using the break statement:

public class SwitchCaseExample2 {

public static void main(String args[]){

int i=2;

switch(i)

{

case 1:

System.out.println("Case1 ");

case 2:

System.out.println("Case2 ");

case 3:

System.out.println("Case3 ");

case 4:

System.out.println("Case4 ");

default:

System.out.println("Default ");

}

}

}

**Output:**

Case2

Case3

Case4

Default

In the above program, we have passed integer value 2 to the switch, so the control switched to the case 2, however we don’t have break statement after the case 2 that caused the flow to pass to the subsequent cases till the end. The solution to this problem is break statement

Break statements are used when you want your program-flow to come out of the switch body. Whenever a break statement is encountered in the switch body, the execution flow would directly come out of the switch, ignoring rest of the cases

## Example with break statement

public class SwitchCaseExample2 {

public static void main(String args[]){

int i=2;

switch(i)

{

case 1:

System.out.println("Case1 ");

break;

case 2:

System.out.println("Case2 ");

break;

case 3:

System.out.println("Case3 ");

break;

case 4:

System.out.println("Case4 ");

break;

default:

System.out.println("Default ");

}

}

}

**Output:**

Case2

Now you can see that only case 2 had been executed, rest of the cases were ignored.

**Why didn’t we use break statement after default?**  
 The control would itself come out of the switch after default so we didn’t use it, however if you still want to use the break after default then you can use it, there is no harm in doing that.

## Few points about Switch Case

1) Case doesn’t always need to have order 1, 2, 3 and so on. It can have any integer value after case keyword. Also, case doesn’t need to be in an ascending order always, you can specify them in any order based on the requirement.

2) You can also use characters in switch case. for example –

public class SwitchCaseExample2 {

public static void main(String args[]){

char ch='b';

switch(ch)

{

case 'd':

System.out.println("Case1 ");

break;

case 'b':

System.out.println("Case2 ");

break;

case 'x':

System.out.println("Case3 ");

break;

case 'y':

System.out.println("Case4 ");

break;

default:

System.out.println("Default ");

}

}

}

# Java Program to check Vowel or Consonant using Switch Case

import java.util.Scanner;

class JavaExample

{

public static void main(String[ ] arg)

{

boolean isVowel=false;;

Scanner scanner=new Scanner(System.in);

System.out.println("Enter a character : ");

char ch=scanner.next().charAt(0);

scanner.close();

switch(ch)

{

case 'a' :

case 'e' :

case 'i' :

case 'o' :

case 'u' :

case 'A' :

case 'E' :

case 'I' :

case 'O' :

case 'U' : isVowel = true;

}

if(isVowel == true) {

System.out.println(ch+" is a Vowel");

}

else {

if((ch>='a'&&ch<='z’) || (ch>='A'&&ch<='Z'))

System.out.println(ch+" is a Consonant");

else

System.out.println("Input is not an alphabet");

}

}

}

In this program we are not using [break statement](https://beginnersbook.com/2017/08/java-break-statement/) with cases intentionally, so that if user enters any vowel, the program continues to execute all the subsequent cases until Case 'U' is reached and that’s where we are setting up the value of a boolean variable to true. This way we can identify that the alphabet entered by user is vowel or not.

**Iteration Statements (Loops)**

**1. For Loop:**

For(for) loop is used when you want to iterate over a range of value.

1. **for** (initialization; termination/condition; increment/decrement) {
2. //Do Something Here
3. }

class ForLoopExample {

public static void main(String args[]){

for(int i=10; i>1; i--){

System.out.println("The value of i is: "+i);

}

}

}

In the above program:  
int i=1 is initialization expression  
i>1 is condition (Boolean expression)  
i– Decrement operation

**infinite For loop**:

// infinite loop

for ( ; ; ) {

// statement(s)

}

## For loop example to iterate an array:

Here we are iterating and displaying array elements using the for loop.

class ForLoopExample3

{

public static void main(String args[]){

int arr[]={2,11,45,9};

//i starts with 0 as array index starts with 0 too

for(int i=0; i<arr.length; i++){

System.out.println(arr[i]);

}

}

}

Output:

2

11

45

9

## 2. While loop

WHILE statement is useful when you want to execute something till a particular condition is true.

1. **while** (expression) {
2. //Do Something
3. }

e.g

class WhileLoopExample

{

public static void main(String args[]){

int i=10;

while(i>1){

System.out.println(i);

i--;

}

}

}

**Output:**

10

9

8

7

6

5

4

3

2

**infinite while loop:**

while (true)

{

statement(s);

}

## Iterating an array using while loop

Here we are iterating and displaying array elements using while loop.

class WhileLoopExample3

{

public static void main(String args[]){

int arr[]={2,11,45,9};

//i starts with 0 as array index starts with 0 too

int i=0;

while(i<4){

System.out.println(arr[i]);

i++;

}

}

}

**Output:**

2

11

45

9

## 3. Do While

Do – While loop is same as While loop except in this case loop is confirmed to execute once(Means in any case statements in Do-While block will execute atleast once).

1. do {
2. // Do Something Here
3. } **while** (expression);

## do-while loop example

class DoWhileLoopExample

{

public static void main(String args[]){

int i=10;

do{

System.out.println(i);

i--;

}while(i>1);

}

}

**Output:**

10

9

8

7

6

5

4

3

2

## Iterating array using do-while loop

class DoWhileLoopExample2

{

public static void main(String args[]){

int arr[]={2,11,45,9};

//i starts with 0 as array index starts with 0

int i=0;

do{

System.out.println(arr[i]);

i++;

}while(i<4);

}

}

**Output:**

2

11

45

9

|  |  |  |
| --- | --- | --- |
| while(Condition) { ————–//statements if(condition) { break; } } | while(Condition) { —————–//statements if(condition) { continue ; } } | modifier returnType MethodName(Parameters) { —————- —————- //statements return resultType ; } |

**Variables in Java**

Variable is used to hold the data. (container)

int num = 100;

In java there are three types of variables.

1. local variables

2. instance variables

3. static variables.

**1. Local variable** –

The variable which are declared inside the method, constructor, block is called as a local variable.

Declared: Inside the method, constructors, blocks.

Scope(permission): inside the method, constructors, blocks.

Memory: stored in stack memory.

**Example:**

**public** class Test

{

public static void main (String[] args)

{

int num1;

int num2;

int res;

res = num1 + num2;

system.out.println(res);

}

**2. Instance variables** -

The variable which are declared inside the class is called as a instance variable.

declared - inside the class.

scope - inside the class.

stored - in a Heap memory.

**Example:**

public class Test

{

// Instance variables (can accessed by creating objects)

int num1;

int num2;

public static void main (String[] args)

{

int res;

res = num1 + num2;

system.out.println(res);

}

}

**Note**: Always access the instance data by using object name.

Instance to instance direct access is possible but within the same class.

**3. Static variables -**

The variable which are declared inside the class with static keyword is called as a local variable

declare: inside the class with static modifier.

scope: inside the class.

stored: inside the method area.

class Test

{

// static variables;

static int num1;

static int num2;

public static void main (String [] args)

{

system.out.println(Test.num1 + Test.num2);

}

}

**Note:** Static data are accessed by three ways.

1. by using class name (Recommended)

2 directly

3. by creating the object name.

**Example for all variables:**

class Test

{

int a = 100;

static boolean b=true;

public static void main(String[] args)

{

double d = 20.5;

system.out.println(d);

Test t = new Test();

system.out.println(t.a);

system.out.println(Test.b);

}

}

**Note**: JVM is assigning the default values to the instance and static variables.

Local variables default values are not assigned by the JVM.

In java, Local data must in initialization.

Always access the local data directly.

Always access the instance data by using object name.

Always access the Static data by using class name.

**Final keyword with Variable**:

Final variable is fixed constants.

Modifications are not allowed.

For local variables only one keyword is applicable that is **final.**

**Examples:**

class Test

{

public static void main(String[] args)

{

final int a = 100;

a += 10;

system.out.println(a);

}

}

**Method in Java**

In java, a method is like function i.e. it is a block of code that defines a behaviour or an action that object can perform.

1. Inside the class business logics is not allowed. so inside the class declare the method to write the logics.

2. Methods are used to write business logics of the application.

3. Method name start with lower case and inner word starts with upper case.

**Advantage of Method**

* Code Reusability
* Code Optimization

There are two types of methods in java.

1. Instance method

2. Static method.

**Creating Method**

**Syntax**

modifier returnType nameOfMethod (Parameter List)

{

// method body

}

The syntax shown above includes −

* **modifier** − It defines the access type of the method and it is optional to use. Public, protected, Private or Default.
* **returnType** − Method may return a value. void or int, char, boolean
* **nameOfMethod** − This is the method name. The method signature consists of the method name and the parameter list.
* **Parameter List** − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
* **method body** − The method body defines what the method does with the statements.

Considering the following example to explain the syntax of a method −

**Syntax**

public static int methodName(int a, int b) //Method Declaration

{

// body or implimentation

}

methodName() //Method Calling

Here,

* **public static** − modifier
* **int** − return type
* **methodName** − name of the method
* **a, b** − formal parameters
* **int a, int b** − list of parameters

**Note**: Instance and static methods are used to write the business logics of the application.

Access the instance data by using object name.

Access the static data by using class Name.

**Example:**

**main within same class:**

*File: Student.java*

**class** Student {

**int** id; //field or data member or instance variable

  String name;

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

  Student s1=**new** Student(); //creating an object of Student

System.out.println(s1.id);//accessing member through reference variable

System.out.println(s1.name);

  }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student)

Output:

0

null

**main outside class**

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main()method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

*File: TestStudent1.java*

**class** Student{

**int** id;

String name;

}

**class** TestStudent1{

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

   Student s1=**new** Student();

System.out.println(s1.id);

System.out.println(s1.name);

}

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStudent1)

Output:

0

null

**Different ways to passing the arguments to Method**

**class** Test {

static void add(int a, int b)

{

**int** result;

result = a+b;

System.out.println(result);

}

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

//Directly passing the constants

Test.add(10,20);

// passing the variable

int x = 100 , y= 200;

Test.add(x,y);

// Taking input from the end user passing the variable values.

Scanner s = new Scanner(System.in);

 System.out.println("Enter first Number");

int num1 = s.nextInt();

 System.out.println("Enter second Number");

int num2 = s.nextInt();

Test.add(num1, num2);

}

}

**Important Point to Remember**

* Java Method return type is mandatory.
* Inside the method return statement should be last.
* Inside the method only one return statement is allowed.
* Method can return only one value or expression.
* If Method is expecting 2 Arguments while calling the method, must pass 2 values,

the order of argument is also important.

* Method arguments are local variables, so we can use them within the method only.
* All user defined methods are called inside the main method only.
* Holding the method return value is optional, but it is recommended to hold the return value.

**Constructor in Java**

**Constructor in java** is a special type of method that is used to initialize the object.

Java constructor is invoked at the time of object creation. It constructs the values i.e.

provides data for the object that is why it is known as constructor.

**Rules for creating java constructor**

There are basically two rules defined for the constructor.

* Constructor name must be same as its class name
* Constructor must have no explicit return type

**Types of java constructors**

There are two types of constructors:

* Default constructor (no-arg constructor)
* Parameterized constructor(User defined constructor)

**1. Java Default Constructor**

|  |
| --- |
| A constructor that has no parameter & without implementation body is known as default constructor. |

*Rule: If there is no constructor in a class, compiler automatically creates a default constructor.*

**Syntax of default constructor:**

* <class\_name> ()
* {}

**Q) What is the purpose of default constructor?**

Default constructor provides the default values to the object like 0, null etc. depending on the type.

**Example of default constructor that displays the default values**

**class** Student3

{

**int** id;

String name;

**void** display()

{

System.out.println(id+" "+name);

}

**public** **static** **void** main(String args[])

{

Student3 s1=**new** Student3();

Student3 s2=**new** Student3();

s1.display();

s2.display();

}

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student3)

Output:

0 null

0 null

**Explanation:**

In the above class, you are not creating any constructor so compiler provides you a default constructor.

Here 0 and null values are provided by default constructor.

**2. Java parameterized constructor**

|  |
| --- |
| A constructor that have parameters is known as parameterized constructor. |

**Why use parameterized constructor?**

|  |
| --- |
| Parameterized constructor is used to provide different values to the distinct objects. |

**Example of parameterized constructor**

|  |
| --- |
| In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor. |

**class** Student4

{

**int** id;

     String name;

     Student4(**int** i, String n)

{

     id = i;

     name = n;

     }

**void** display()

{

System.out.println(id+" "+name);

}

**public** **static** **void** main(String args[])

{

     Student4 s1 = **new** Student4(111,"Karan");

     Student4 s2 = **new** Student4(222,"Aryan");

     s1.display();

     s2.display();

    }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student4)

Output:

111 Karan

**Access Modifiers in java**

There are 4 types of java access modifiers:

* private
* default
* protected
* public

**1) private access modifier**

|  |
| --- |
| The private access modifier is accessible only within class. |

**Simple example of private access modifier**

|  |
| --- |
| In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error. |

**class** A{

**private** **int** data=40;

**private** **void** msg()

{

System.out.println("Hello java");}

}

**public** **class** Simple{

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

A obj=**new** A();

    System.out.println(obj.data); //Compile Time Error

obj.msg(); //Compile Time Error

   }

}

**Role of Private Constructor**

|  |
| --- |
| If you make any class constructor private, you cannot create the instance of that class from outside the class. For example: |

**class** A

{

**private** A()

{

} //private constructor

**void** msg()

{

System.out.println("Hello java");}

}

**public** **class** Simple

{

**public** **static** **void** main(String args[])

{

    A obj=**new** A();//Compile Time Error

  }

}

*Note: A class cannot be private or protected except nested class.*

**2) default access modifier**

|  |
| --- |
| If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. |

**Example of default access modifier**

|  |
| --- |
| In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package. |

//save by A.java

**package** pack;

**class** A

{

**void** msg()

{

System.out.println("Hello");

}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B

{

**public** **static** **void** main(String args[])

{

    A obj = **new** A();//Compile Time Error

    obj.msg();//Compile Time Error

   }

}

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

**3) protected access modifier**

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

**package** pack;

**public** **class** A

{

**protected** **void** msg()

{

System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B **extends** A

{

**public** **static** **void** main(String args[])

{

    B obj = **new** B();

    obj.msg();

   }

}

Output: Hello

**4) public access modifier**

|  |
| --- |
| The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers. |

**Example of public access modifier**

//save by A.java

**package** pack;

**public** **class** A

{

**public** **void** msg()

{

System.out.println("Hello");}

}

/save by B.java

**package** mypack;

**import** pack.\*;

**class** B

{

**public** **static** **void** main(String args[])

{

    A obj = **new** A();

    obj.msg();

   }

}

Output: Hello

**Understanding all java access modifiers**

Let's understand the access modifiers by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

**Java static keyword**

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

The static can be:

* variable (also known as class variable)
* method (also known as class method)
* nested classes
* block

**1) Java static variable**

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

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

**Advantage of static variable**

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

***Understanding problem without static variable***

**class** Student{

**int** rollno;

     String name;

     String college="ITS";

}

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created. All student has its unique rollno and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

*Java static property is shared to all objects.*

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student8)

Output: 111 Karan ITS

222 Aryan ITS

**2) Java static method**

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

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

**Example of static method**

//Program of changing the common property of all objects(static field).

**class** Student9

{

**int** rollno;

     String name;

**static** String college = "ITS";

**static** **void** change()

{

      college = "BBDIT";

     }

     Student9(**int** r, String n)

{

      rollno = r;

      name = n;

     }

**void** display ()

{

System.out.println(rollno+" "+name+" "+college);

}

**public** **static** **void** main(String args[])

{

     Student9.change();

       Student9 s1 = **new** Student9 (111,"Karan");

   Student9 s2 = **new** Student9 (222,"Aryan");

     Student9 s3 = **new** Student9 (333,"Sonoo");

     s1.display();

     s2.display();

     s3.display();

    }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student9)

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

**Restrictions for static method**

|  |
| --- |
| There are two main restrictions for the static method. They are: |
| * The static method cannot use non static data member or call non-static method directly. * this and super cannot be used in static context. | |

**class** A

{

**int** a=40;//non static

**public** **static** **void** main(String args[])

{

   System.out.println(a);

  }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A)

Output: Compile Time Error

**Q) why java main method is static?**

|  |
| --- |
| Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation. |

**3) Java static block**

* Is used to initialize the static data member.
* It is executed before main method at the time of class loading.

**Example of static block**

**class** A2

{

**Static**

{

System.out.println("static block is invoked");}

**public** **static** **void** main(String args[])

{

    System.out.println("Hello main");

   }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A2)

Output: static block is invoked

Hello main

**3) Nested Classes**

Nested classes can make your code more readable and maintainable by keeping related classes close together. Nested classes can access private members of their outer class.

This allows for more precise control over access rights and improves the overall security of the code.

**Example:**

public class OuterClass {  
  
 private int a =20 , b=30;  
 void m1()  
 {  
 System.*out*.println("outer class method");  
 }  
 public class Inner  
 {  
 int x = 200, y =300;  
 void m2()  
 {  
 System.*out*.println("Inner class method");  
 System.*out*.println("Outer Class Variable in inner Class :" + a);  
 m1();  
 }  
 }  
  
 public static void main(String[] args) {  
 OuterClass o = new OuterClass();  
 o.m1();  
 OuterClass.Inner i = o.new Inner();  
 i.m2();  
  
 }  
}

**OOPS (Object-Oriented Programming System)**

**Class vs. Object:**

Class is a logical entity it contains logics whereas object is physical entity it is representing memory.

Class is blue print it decides object creation without class we are unable to create object.

Based on single class it is possible to create multiple objects but every object occupies memory.

We are declaring the class by using class keyword but we are creating object by using new keyword.

**Encapsulation**

The process of binding the data(variables) and code(methods) as a single unit is called encapsulation.

Group mechanism is called encapsulation. The process of hiding the implementation details to user is called encapsulation. And we are achieving this concept by declaring variables as a private modifier because it is possible to access private members with in the class only.

**Data hiding**: The main objective is data hiding is security and it is possible to hide the data by using private modifier. If a variable declared as a private it is possible to access those variables only inside the class is called data hiding.

If the variables are declared as a private it is possible to access those variables only with in the class but it possible to set(update) the data by using setter methods and it is possible to get(read) the data by using getter methods. The data of the private field can be accessed only by using public setter & getter method

In this way we are hiding implementation to other classes. The setter and getter methods are user defined methods. Syntax: - setXXX() where xxx=property name getXXX() where xxx=property name The setter method return type is always void & getter method return type is always property return type.

**Example**:

Student.java

Public class Student

{

private int sid;

private int sname;

public void setSid(int sid)

{

this.sid=sid;

}

public void setSname(String sname)

{

this.sname=sname;

}

public int getSid()

{

return sid;

}

public String getSname()

{

return sname;

}

}

Accessing encapsulated use fallowing code:

Test.java class Test

{

public static void main(String[] args)

{

Student s=new Student();

e.setSid(100);

e.setSname("Rohan");

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

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

}

}

1. **Abstraction:**

### Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user. For example, when you login to your bank account online, you enter your user id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from you.

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

**Abstract class in Java**

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body). Cannot create an object of abstract class.

**Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

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

**Example abstract class**

abstract class A

{

}

**abstract method**

A method that is declared as abstract and does not have implementation is known as abstract method.

**Example abstract method**

abstract void printStatus();

;//no body and abstract

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

abstract class Bike

{

abstract void run();

}

class Honda4 extends Bike

{

void run()

{

System.out.println("running safely..");

}

public static void main(String args[])

{

Bike obj = new Honda4();

obj.run();

}

}

**2. Using 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.

It cannot be instantiated just like abstract class.

**Why use Java interface?**

It is used to achieve abstraction.

By interface, we can support the functionality of multiple inheritance.

Java 8 Interface Improvement

Since Java 8, interface can have default and static methods

**Internal addition by compiler**

The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.

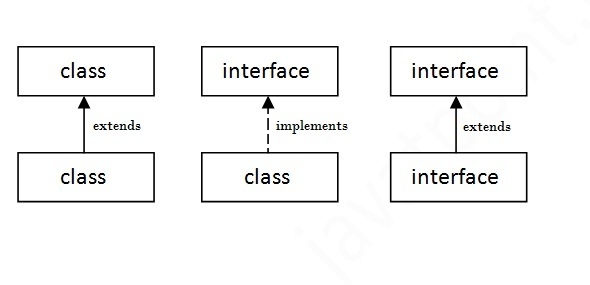
In other words, Interface fields are public, static and final by default, and methods are public and abstract.

|  |  |  |
| --- | --- | --- |
| Interface Test {  int num = 5;  void display();  } | **Compiler** | Interface Test {  Public static final int num = 5;  Public abstract void display();  } |

.java .class

**Understanding relationship between classes and interfaces**

As shown in the figure given below, a class extends another class, an interface extends another interface but a class implements an interface.

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

**3. Inheritance**

Inheritance in java is a mechanism in which one object acquires all the properties and behaviours of parent object.

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

**Why use inheritance in java**

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

For Code Reusability.

Syntax

class Subclass-name extends Superclass-name

{

//methods and fields

}

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality. In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child class.

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);

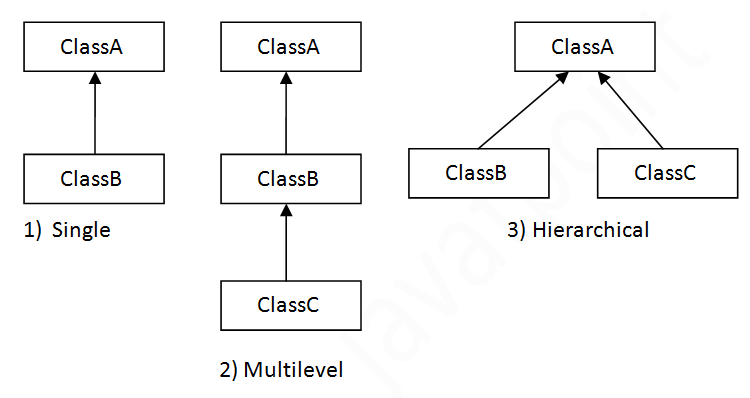
}

}

**Types of inheritance in java**

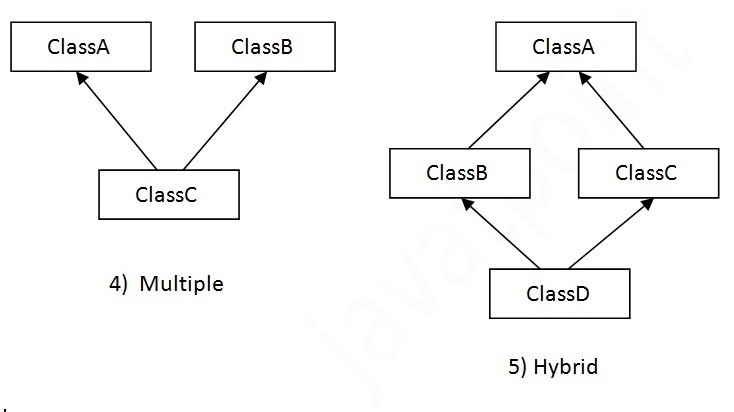
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.



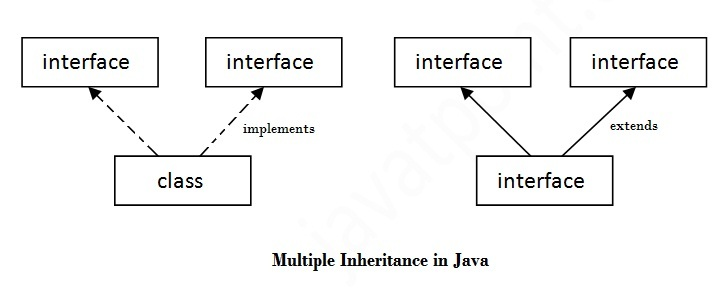
Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance. For Example:



Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



**4. Polymorphism**

Polymorphism is one of the OOPs feature that allows us to perform a single action in different ways. For example, lets say we have a class Animal that has a method sound(). Since this is a generic class so we can’t give it a implementation like: Roar, Meow, Oink etc. We had to give a generic message.

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 subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

**Usage of Java Method Overriding**

Method overriding is used to provide specific implementation of a method that is already provided by its super class.

Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

method must have same name as in the parent class

method must have same parameter as in the parent class.

must be IS-A relationship (inheritance).

Let's understand the problem that we may face in the program if we don't use method overriding.

class Vehicle

{

void run()

{

System.out.println("Vehicle is running");}

}

class Bike extends Vehicle

{

public static void main(String args[])

{

Bike obj = new Bike();

obj.run();

}

}

Test it Now

Output: Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding[e.g dad's bike]

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

class Vehicle

{

void run()

{

System.out.println("Vehicle is running");}

}

class Bike2 extends Vehicle

{

void run()

{

System.out.println("Bike is running safely");

}

public static void main (String args[])

{

Bike2 obj = new Bike2();

obj.run();

}

String in Java

**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 string objects.

**How to create a String object?**

There are two ways to create string object

1. String Literal
2. By New Keyword

1) String Literal

Java String literal is created by using double quotes.

e.g:

String s="java";

2) new keyword

Java String is created by using New keyword.

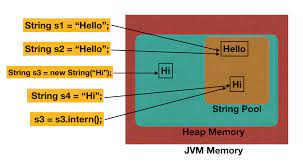
e.g:

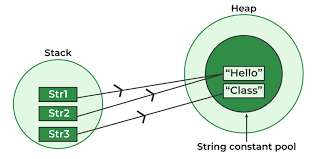
String s=**new** String("java");

**String Constant Pool :**

String constant pool located inside a section of Heap memory. This is a part of memory that is used for run-time operations, working with classes and objects. String constant pool does not allow users to create duplicate strings. If the user tries to create a duplicate string using "(double quotes), then the string pool returns the address of the existing string.

So at any given time, the Constant String pool will contain only unique values.





**Compare Strings**:

We cannot compare strings using '==' relational operator, as '==' operator tries to compare the address of the object. But most of the time, strings may not be stored in the same object so the address will be different for each string. We should use equals() present in the String object to compare two strings.

**Example:**

Public Class StringCompare

{

Public static void main(String[] args)

{

String data = "java";   
 String data1 = "java”;   
  
 if (data==data1) // compare address  
 {  
 System.*out*.println("Strings are equal by ==");  
 } else

{  
 System.*out*.println("Strings are different by ==");  
 }  
  
 if (data.equals(data1)) // compare value  
 {  
 System.*out*.println("Strings are equal by equals method");  
 } else  
 {  
 System.*out*.println("Strings are different by equals method");  
 }  
  
 String temp = new String(" java");   
 String temp1 = new String("Java");   
  
 if (temp==temp1) // compare address  
 {  
 System.*out*.println("Strings are equal by ==");  
 } else  
 {  
 System.*out*.println("Strings are different by ==");

}  
  
 if (temp.equals(temp1)) // compare value  
 {  
 System.*out*.println("Strings are equal by equals method");  
 } else  
 {  
 System.*out*.println("Strings are different by equals method");  
 }

}

}

### Java String class methods

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

|  |  |
| --- | --- |
| **Method** | **Description** |
| [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| [boolean contains(Char Sequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value |
| [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with object |
| [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty |
| [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatenates specified string |
| [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified char value |
| [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. |
| [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns spitted string-matching regex |
| [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index |
| [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |

**Java: String is Immutable**

class StringImMutable {

public static void main(String[] args)

{

String s1 = "java";

s1.concat(" Class");

System.out.println("s1 refers to " + s1);

}

}

**Output:**

s1 refers to java

**What’s happening:**

1. The first line is pretty straightforward: create a new String "java" and refer s1 to it.
2. Next, the VM creates another new String "java rules", but nothing refers to it. So, the second String is instantly lost. We can’t reach it.

Strings are immutable, but not all the String are immutable. Strings created using String class are immutable, but Strings created using StringBuffer, StringBuilder are mutable.

When we create strings using String class in java, JVM creates immutable strings. That means once you create a string object, you can’t modify the contents of that object. If you try to modify the contents of the String object, a new string object is created with modified content.

String Buffer and String Builder classes

1. StringBuffer class is used to create mutable strings, which means we can alter the value of the string after we create it. It represents a grow-able and append-able character sequence. Strings are formed using the StringBuffer class when we have to make a lot of modifications to our string.

**Example:**

class StringMutable {

public static void main(String[] args)

{

StringBuffer s1 = new StringBuffer ("java");

s1.append(" Class");

System.out.println("s1 refers to " + s1);

}

}

**Output:**

s1 refers to java class.

2. StringBuffer class methods are synchronized, so multiple threads cannot access the string value. StringBuilder class is similar to the StringBuffer class; the only difference is StringBuilder class methods are not synchronized so that multiple threads can access the String value.

**Example:**

class StringMutable {

public static void main(String[] args)

{

StringBuider s1 = new StringBuider ("java");

s1.append(" Class");

System.out.println("s1 refers to " + s1);

}

}

**Output:**

s1 refers to java class.

Exception Handling in Java

Exception handling in Java is a mechanism that allows you to gracefully handle runtime errors, also known as exceptions

### What is exception

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

### What is exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.



Java provides a robust exception handling framework to catch and manage these exceptions, preventing your program from crashing and enabling you to provide meaningful error messages or take appropriate actions when exceptions occur.

Types of Exception

There are mainly two types of exceptions: Checked and Unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception/compile time exception
2. Unchecked Exception/run time exception

Checked vs Unchecked Exceptions:

1. Checked Exceptions: These are exceptions that the compiler forces you to handle or declare. They are subclasses of `java.lang.Exception` but not subclasses of `RuntimeException`. Examples include `IOException` and `SQLException`.

2. Unchecked Exceptions (Runtime Exceptions): These are exceptions that the compiler does not force you to catch or declare. They are subclasses of `RuntimeException`. Examples include `NullPointerException` and `ArrayIndexOutOfBoundsException`.

Exception handling is an essential part of writing reliable and maintainable code. It helps you manage unexpected situations and prevents crashes in your programs. By understanding and properly implementing exception handling, you can create more robust and user-friendly applications.

## Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws
6. **Try-Catch Blocks**

A try-catch block is used to catch exceptions that might occur within a specific portion of code. The syntax is as follows:

**Syntax**

try {

// Code that might throw an exception

} catch (ExceptionType1 e1)

{

// Code to handle ExceptionType1

} catch (ExceptionType2 e2)

{

// Code to handle ExceptionType2

} finally

{

// Optional block that executes regardless of whether an exception occurred

}

# Java Nested try block

The try block within a try block is known as nested try block in java.

### Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

### Syntax:

1. ....
2. **try**
3. {
4. statement 1;
5. statement 2;
6. **try**
7. {
8. statement 1;
9. statement 2;
10. }
11. **catch**(Exception e)
12. {
13. }
14. }
15. **catch**(Exception e)
16. {
17. }
18. ....

Java finally block

The `finally` block is used to execute code that must run regardless of whether an exception occurred. It's often used to clean up resources like closing files or releasing connections.

**throws:**

## What is the need of having throws keyword when you can handle exception using try-catch?

we can [handle exceptions](https://beginnersbook.com/2013/04/java-exception-handling/) using [try-catch block](https://beginnersbook.com/2013/04/try-catch-in-java/).  
The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch. For example:  
Lets say we have a method myMethod() that has statements that can throw either ArithmeticException or NullPointerException, in this case you can use try-catch as shown below:

public void myMethod()

{

try {

// Statements that might throw an exception

}

catch (ArithmeticException e) {

// Exception handling statements

}

catch (NullPointerException e) {

// Exception handling statements

}

}

But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary long and will be less-readable.

One way to overcome this problem is by using throws like this: declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.  
Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.

public void myMethod() throws ArithmeticException, NullPointerException

{

// Statements that might throw an exception

}

public static void main(String args[]) {

try {

myMethod();

}

catch (ArithmeticException e) {

// Exception handling statements

}

catch (NullPointerException e) {

// Exception handling statements

}

}

## Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The syntax of java throw keyword is given below.

1. **throw** exception;

## java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

1. **public** **class** TestThrow1{
2. **static** **void** validate(**int** age){
3. **if**(age<18)
4. **throw** **new** ArithmeticException("not valid");
5. **else**
6. System.out.println("welcome to vote");
7. }
8. **public** **static** **void** main(String args[]){
9. validate(13);
10. System.out.println("rest of the code...");
11. }
12. }

Array In Java

Collection of similar type of elements is known as Array. Array in Java is an Object that holds fixed number of values of a similar data type which means an array of int will contain only integers, an array of string will contain only strings etc.

**Syntax**

|  |  |
| --- | --- |
| 1  2 | dataType[] arrayName;  arrayName = new dataType[array Size]; |

or

|  |  |
| --- | --- |
| 1 | arrayName[index] = arrayElement; |

**Examples:**

int[] numbers = new int[5]; // Declare an array of integers with a size of 5

// Initialize elements

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

numbers[4] = 50;

// Accessing elements

int thirdNumber = numbers[2]; // Access the element at index 2 (30)

// Array length

int arrayLength = numbers.length; // Get the length of the array (5)

// Iterating over elements

for (int i = 0; i < numbers.length; i++) {

System.out.println(numbers[i]);

}

// Enhanced for loop (for-each loop)

for (int num : numbers) {

System.out.println(num);

}

Java also supports arrays for other types, such as strings, objects, and custom classes:

// Array of strings

String[] names = new String[3];

names[0] = "Alice";

names[1] = "Bob";

names[2] = "Charlie";

// Array of objects (using wrapper classes for primitive types)

Integer[] integerArray = new Integer[] { 1, 2, 3, 4, 5 };

// Array of custom objects

class Person {

String name;

int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

}

Person[] people = new Person[2];

people[0] = new Person("Alice", 25);

people[1] = new Person("Bob", 30);

**2D Array:**

2D Arrays are same like 1D Array but it consists of rows and columns.

**2D Array Declaration:**

**Syntax:**

|  |  |
| --- | --- |
| 1 | dataType[] arrayName = new dataType[row][column]; |

**Example:**

|  |  |
| --- | --- |
| 1 | int[][] arr = new int[2][3]; |

Here, **arr** is a 2-D array of type int which consists of **2 rows** and **3 columns**.

**2D Array Initialization:**

Declaring 2D array and then assigning values to its elements

int[][] a = new int[2][3];

a[0][0]=11;

a[0][1]=22;

a[0][2]=33;

a[1][0]=44;

a[1][1]=55;

a[1][2]=66;

package classFiveArrays;

public class Array2DClass {

public static void main (String [] args){

int[][] arr = new int[2][3];

arr[0][0]=11;

arr[0][1]=22;

arr[0][2]=33;

arr[1][0]=44;

arr[1][1]=55;

arr[1][2]=66;

System.out.println(arr[0][0]);

System.out.println(arr[0][1]);

System.out.println(arr[0][2]);

System.out.println(arr[1][0]);

System.out.println(arr[1][1]);

System.out.println(arr[1][2]);

}

}

Remember that arrays have a fixed size, meaning you cannot change the number of elements after the array is created. If you need dynamic sizing, you might consider using other data structures like **ArrayList** from the Java Collections framework.

Arrays are a core concept in Java and play a crucial role in various programming tasks, including algorithms, data manipulation, and more.

**Limitations of Array:**

1. Arrays won’t allow multiple data type.

2. In Array, we can store only fixed size of elements.

Collections in Java

**Collections in java** is a framework that provide a way to store manipulate and manage groups of objects. They are part of the java collections Framework and offer various interfaces and classes to handle different types of data structures efficiently. All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, HashSet, LinkedHashSet, TreeSet etc).

### **Hierarchy of Collection Framework**

The **java.util** package contains all the classes and interfaces for Collection framework.



### **Iterator interface**

The `Iterator` interface in Java is a fundamental component of the Java Collections Framework that provides a way to iterate (loop) over the elements of a collection without exposing the underlying structure of the collection. It's a part of the `java.util` package and is widely used to traverse elements in collections such as lists, sets, and maps.

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

#### **Methods of Iterator interface**

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

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if iterator has more elements. |
| 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 rarely used. |

**Example:**

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class IteratorExample {

public static void main(String[] args) {

List<String> fruits = new ArrayList<>();

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

Iterator<String> iterator = fruits.iterator();

while (iterator.hasNext()) {

String fruit = iterator.next();

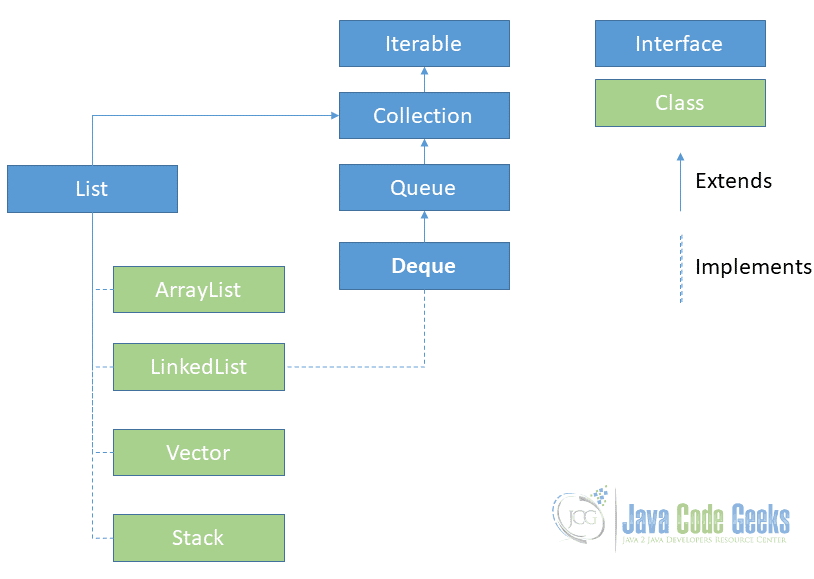
System.out.println(fruit);

}

}

}

**List: i/f**



**Java ArrayList class**

ArrayList is one of the most commonly used classes in the java collections framework. It’s a dynamic array implementation that provides a resizable array with some additional methods to manipulate and work with the elements.

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list

### **ArrayList class declaration**

Let's see the declaration for java.util.ArrayList class.

**public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

### **Methods of Java ArrayList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| void clear() | It is used to remove all of the elements from this list. |

### **Java ArrayList Example**

**import** java.util.\*;

**class** TestCollection1{

**public** **static** **void** main(String args[])

{

  ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist

  list.add("Java");//Adding object in arraylist

  list.add("Ruby");

  list.add("java");

  list.add("C++");

  //Traversing list through Iterator

  Iterator itr=list.iterator();

**while**(itr.hasNext()){

    System.out.println(itr.next());

   }

  }

}

**There are two ways to traverse collection elements:**

1. By Iterator interface.
2. By for-each loop.

### Iterating Collection through for-each loop

import java.util.\*;

class TestCollection2{

 public static void main(String args[]){

  ArrayList<String> al=new ArrayList<String>();

  al.add("Java");

  al.add("Ruby");

  al.add("Java");

  al.add("C++");

  for (String obj:al)

    System.out.println(obj);

 }

}

### **User-defined class objects in Java ArrayList**

**class** Student

{

**int** stdId;

**String** stdName;

**int** age;

   Student(**int** stdId, **String** stdName, **int** age)

{

**this**.stdId = stdId;

**this**.stdName = name;

**this**.age = age;

   }

}

**class** ArrayListUserDefined

{

**import** java.util.\*;

**public** **class** TestCollection3{

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

  //Creating user-defined class objects

  Student s1=**new** Student(101,"Suraj",24);

  Student s2=**new** Student(102,"Rani",21);

  Student s2=**new** Student(103,"Raman",25);

//creating arraylist

  ArrayList<Student> al=**new** ArrayList<Student>();

  al.add(s1); //adding Student class object

  al.add(s2);

  al.add(s3);

  //Getting Iterator

  Iterator itr=al.iterator();

  //traversing elements of ArrayList object

**while**(itr.hasNext()){

    Student st=(Student)itr.next();

    System.out.println(st.rollno+" "+st.name+" "+st.age);

  }

 }

}

**Java LinkedList class**

Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.

### How To Implement Queue Using Doubly Linked List In C | PrepBytes Blog

### **LinkedList class declaration**

**public** **class** LinkedList<E> **implements** List<E>

### Methods of Java LinkedList

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| void addFirst(Object o) | It is used to insert the given element at the beginning of a list. |
| void addLast(Object o) | It is used to append the given element to the end of a list. |
| int size() | It is used to return the number of elements in a list |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean contains(Object o) | It is used to return true if the list contains a specified element. |
| boolean remove(Object o) | It is used to remove the first occurence of the specified element in a list. |
| Object getFirst() | It is used to return the first element in a list. |
| Object getLast() | It is used to return the last element in a list. |
| int indexOf(Object o) | It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element. |
| int lastIndexOf(Object o) | It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element. |

### **Java LinkedList Example**

### **import** java.util.\*;

**public** **class** TestCollection7

{

**public** **static** **void** main(String args[])

{

   LinkedList<String> al=**new** LinkedList<String>();

   al.add("Java");

   al.add("Python");

   al.add("Java");

   al.add("Ruby");

   Iterator<String> itr=al.iterator();

**while**(itr.hasNext())

{

    System.out.println(itr.next());

   }

  }

}

### Java LinkedList Example: Book

**import** java.util.\*;

**class** Fruits

{

**int** id;

String name

**int** quantity;

**public** Book(**int** id, String name **int** quantity)

{

**this**.id = id;

**this**.name = name;

**this**.quantity = quantity;

}

}

**public** **class** LinkedListUserDefined {

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

     //Creating list of Fruits

     List<Fruits> list=**new** LinkedList< Fruits>();

     //Creating Fruits

     Fruits f1=**new** Fruits (101,"Orange",8);

     Fruits f2=**new** Fruits (102,"Apple",12);

Fruits f2=**new** Fruits (103,"Mango",20);

//Adding Fruits to list

list.add(f1);

     list.add(f2);

     list.add(f3);

    //Traversing list

**for**(Book f:list)

{

    System.out.println(f.id + " " + f.name + " " + f.quantity);

    }

}

}

# Difference between ArrayList and LinkedList

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| 1) ArrayList internally uses dynamic array to store the elements. | 1) LinkedList internally uses **doubly linked list** to store the elements |
| 2) Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory | 2) Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList is better for storing and accessing data. | 3) LinkedList is **better for manipulating** data. |

**Java List Interface**

The `ListIterator` interface in Java is part of the Java Collections Framework and extends the capabilities of the `Iterator` interface. It provides bidirectional traversal (both forward and backward) of elements in a collection, along with methods for adding, modifying, and removing elements while iterating. It's particularly useful for navigating and modifying elements in lists.

Here's are the methods of the `ListIterator` interface in Java

|  |  |
| --- | --- |
| Method | Description |
| hasNext() | Returns `true` if there's a next element, otherwise `false` |
| next() | Returns the next element and moves the iterator forward |
| hasPrevious() | Returns ‘true’ if there's a previous element, otherwise `false` |
| previous() | Returns the previous element and moves the iterator backword |
| nextIndex() | Returns the index of the next element that would be returned by a subsequent call to next() |
| previousIndex() | Returns the index of the previous element that would be returned by a subsequent call to `previous() |

**Example:**

import java.util.ArrayList;

import java.util.List;

import java.util.ListIterator;

public class ListIteratorExample {

public static void main(String[] args) {

List<String> colors = new ArrayList<>();

colors.add("Red");

colors.add("Green");

colors.add("Blue");

ListIterator<String> iterator = colors.listIterator();

while (iterator.hasNext()) {

String color = iterator.next();

System.out.println("Next color: " + color);

}

while (iterator.hasPrevious()) {

String color = iterator.previous();

System.out.println("Previous color: " + color);

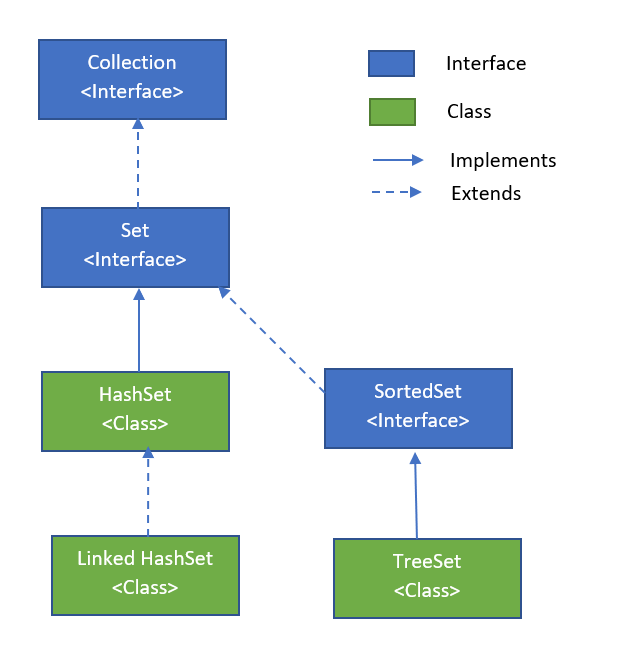
}

}

}

**Set:**

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



1. Java HashSet class

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.

### Methods of Java HashSet class:

|  |  |
| --- | --- |
| Method | Description |
| void clear() | It is used to remove all of the elements from this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean add(Object o) | It is used to adds the specified element to this set if it is not already present. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| Iterator iterator() | It is used to return an iterator over the elements in this set. |
| int size() | It is used to return the number of elements in this set. | |

### Java HashSet Example

import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

// Create a HashSet to store strings

HashSet<String> colors = new HashSet<>();

// Add elements to the HashSet

colors.add("Red");

colors.add("Green");

colors.add("Blue");

colors.add("Red"); // Duplicates are not allowed in HashSet

// Display the HashSet

System.out.println("HashSet: " + colors);

// Check if an element exists in the HashSet

boolean containsBlue = colors.contains("Blue");

System.out.println("Contains Blue: " + containsBlue);

// Remove an element from the HashSet

boolean removedGreen = colors.remove("Green");

System.out.println("Removed Green: " + removedGreen);

// Display the updated HashSet

System.out.println("Updated HashSet: " + colors);

// Get the size of the HashSet

int size = colors.size();

System.out.println("Size of HashSet: " + size);

// Clear all elements from the HashSet

colors.clear();

System.out.println("Cleared HashSet: " + colors);

}

}

**Java HashSet Example: Student**

import java.util.HashSet;

class Student {

int id;

String name;

public Student(int id, String name)

{

this.id = id;

this.name = name;

}

}

public class HashSetUserDefinedExample {

public static void main(String[] args) {

HashSet<Student> studentSet = new HashSet<>();

Student student1 = new Student(1, "Alice");

Student student2 = new Student(2, "Bob");

Student student3 = new Student(3, "Charlie");

studentSet.add(student1);

studentSet.add(student2);

studentSet.add(student3);

System.out.println("Student Set: " + studentSet);

Student student4 = new Student(2, "Bob"); // Duplicate ID, different name

boolean added = studentSet.add(student4);

System.out.println("Added Student: " + added);

System.out.println("Updated Student Set: " + studentSet);

    //Traversing hash table

**for**(Student b: studentSet)

{

     System.out.println(b.id + " " + b.name);

}

}

}

# Java LinkedHashSet class

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

The important points about Java LinkedHashSet class are:

* Contains unique elements only like HashSet.
* Provides all optional set operations, and permits null elements.
* Maintains insertion order.

**Example of LinkedHashSet class:**

import java.util.LinkedHashSet;

public class LinkedHashSetExample {

public static void main(String[] args) {

// Create a LinkedHashSet to store strings

LinkedHashSet<String> colors = new LinkedHashSet<>();

// Add elements to the LinkedHashSet

colors.add("Red");

colors.add("Green");

colors.add("Blue");

colors.add("Red"); // Duplicates are not allowed in LinkedHashSet

// Display the LinkedHashSet

System.out.println("LinkedHashSet: " + colors);

// Check if an element exists in the LinkedHashSet

boolean containsBlue = colors.contains("Blue");

System.out.println("Contains Blue: " + containsBlue);

// Remove an element from the LinkedHashSet

boolean removedGreen = colors.remove("Green");

System.out.println("Removed Green: " + removedGreen);

// Display the updated LinkedHashSet

System.out.println("Updated LinkedHashSet: " + colors);

### // Get the size of the LinkedHashSet

### int size = colors.size();

### System.out.println("Size of LinkedHashSet: " + size);

### // Clear all elements from the LinkedHashSet

### colors.clear();

### System.out.println("Cleared LinkedHashSet: " + colors);

### }

### }

### **Java LinkedHashSet Example**: **Person**

import java.util.LinkedHashSet;

class Person {

int id;

String name;

public Person(int id, String name) {

this.id = id;

this.name = name;

}

}

public class LinkedHashSetUserDefinedExample {

public static void main(String[] args) {

LinkedHashSet<Person> personSet = new LinkedHashSet<>();

Person person1 = new Person(1, "Alice");

Person person2 = new Person(2, "Bob");

Person person3 = new Person(3, "Charlie");

personSet.add(person1);

personSet.add(person2);

personSet.add(person3);

System.out.println("Person Set: " + personSet);

Person person4 = new Person(2, "Bob"); // Duplicate ID, different name

boolean added = personSet.add(person4);

System.out.println("Added Person: " + added);

System.out.println("Updated Person Set: " + personSet);

     //Traversing hash table

**for**(Student p: personSet)

{

     System.out.println(p.id + " " + p.name);

}

}

}

## 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:

1. SortedSet<data-type> set = **new** TreeSet();

## Java TreeSet Class

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.

The important points about Java TreeSet class are:

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order.

### Methods of Java TreeSet class

|  |  |
| --- | --- |
| Method | Description |
| boolean addAll(Collection c) | It is used to add all of the elements in the specified collection to this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void add(Object o) | It is used to add the specified element to this set if it is not already present. |
| void clear() | It is used to remove all of the elements from this set. |
| int size() | It is used to return the number of elements in this set. |

### **Java TreeSet 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()){

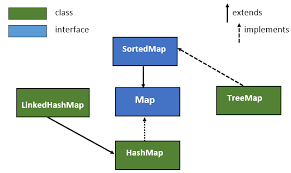
System.out.println(itr.next());

}

}  }

**MAP:**

The **Map** interface in java is part of the java collections Framework and represents a collection of key-value pairs, where each key is associated with a corresponding value. It is used to store and manipulate data in the form of mapping between keys and values. The **Map** interface provides methods to add, remove, retrieve and manipulate these key value pairs. It is implemented by various classes in java to provide different types of map implementations.



# Java HashMap class

A `HashMap` is a class in Java that implements the `Map` interface, which is a part of the Java Collections Framework. It is used to store key-value pairs, where each key is unique and is used to retrieve a corresponding value. `HashMap` provides fast and constant-time performance for basic operations like adding, removing, and retrieving elements, making it a commonly used data structure for lookups and mappings.

The important points about Java HashMap class are:

* A HashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains no order.

### **HashMap class Parameters**

Let's see the Parameters for java.util.HashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Methods of Java HashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object put(Object key, Object value) | It is used to associate the specified value with the specified key in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |

### **Java HashMap Example**:

import java.util.\*;

public class HashMapExample {

public static void main(String[] args) {

// Create a HashMap

HashMap<Integer, String> ages = new HashMap<>();

// Add key-value pairs

ages.put(30, "Alice");

ages.put(25, "Bob");

ages.put(40, "Carol");

// Retrieve values

int aliceAge = ages.get("Alice");

System.out.println("Alice's age: " + aliceAge);

// Update value

ages.put(31, "Alice");

// Print all key-value pairs

for (Map.Entry<Integer, String> entry : ages.entrySet()) {

System.out.println(entry.getValue() + " is " + entry.getKey() + " years old.");

}

}

}

### **Java HashMap Example**: **Person**

import java.util.\*;

class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public class CustomHashMapExample {

public static void main(String[] args) {

// Create a HashMap with Person objects as keys

HashMap<Integer, Person> personMap = new HashMap<>();

Person alice =**new** Person ("alice",30);

Person bob =**new** Person("bob",24);

// Add key-value pairs

personMap.put(1, alice);

personMap.put(2, bob);

    //Traversing map

**for**(Map.Entry<Integer, person> entry:map.entrySet())

{ **int** key=entry.getKey(); personMap b=entry.getValue();

         System.out.println(key+" Details:");

         System.out.println(b.id+" "+b.name+" "+b.age);

   }

}

}

**Java LinkedHashMap class**

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

The important points about Java LinkedHashMap class are:

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead maintains insertion order.

### LinkedHashMap class Parameters

Let's see the Parameters for java.util.LinkedHashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Methods of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object get(Object key) | It is used to return the value to which this map maps the specified key. |
| void clear() | It is used to remove all mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map maps one or more keys to the specified value. |

### **Java LinkedHashMap Example**

import java.util.\*;

public class LinkedHashMapExample {

public static void main(String[] args) {

LinkedHashMap<Integer, String> linkedHashMap = new LinkedHashMap<>();

// Add key-value pairs

linkedHashMap.put(3, "C");

linkedHashMap.put(1, "A");

linkedHashMap.put(2, "B");

// Iterate over the LinkedHashMap and observe the insertion order

System.out.println("Insertion Order:");

for (Map.Entry<Integer, String> entry : linkedHashMap.entrySet()) {

System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());

}

// Accessing a value to change the order (it doesn't affect order)

String value = linkedHashMap.get(1);

// Insert a new key-value pair

linkedHashMap.put(4, "D");

// Print the LinkedHashMap after adding a new entry

System.out.println("LinkedHashMap after adding a new entry:");

for (Map.Entry<Integer, String> entry : linkedHashMap.entrySet()) {

System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());

}

}

}

### **Java LinkedHashMap :Student**

import java.util.\*;

class Student {

private int studentId;

private String name;

public Student(int studentId, String name) {

this.studentId = studentId;

this.name = name;

}

public class CustomLinkedHashMapExample {

public static void main(String[] args) {

// Create a LinkedHashMap with Student objects as keys

LinkedHashMap<Student, Integer> studentMarks = new LinkedHashMap<>();

// Create Student instances and use them as keys

Student alice = new Student(101, "Alice");

Student bob = new Student(102, "Bob");

// Add key-value pairs

studentMarks.put(alice, 90);

studentMarks.put(bob, 85);

// Print the student marks using Student keys

for (Map.Entry<Student, Integer> entry : studentMarks.entrySet()) {

System.out.println(entry.getKey() + " - Marks: " + entry.getValue());

}

}

}

**Java Tree Map class**

Java TreeMap class implements the Map interface by using a tree. It provides an efficient means of storing key/value pairs in sorted order.

The important points about Java TreeMap class are:

* A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead maintains ascending order.

### TreeMap class Parameters

Let's see the Parameters for java.util.TreeMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Java TreeMap Example**:

**import** java.util.\*;

**class** TestCollection15{

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

  TreeMap<Integer,String> hm=**new** TreeMap<Integer,String>();

  hm.put(100,"Amit");

  hm.put(102,"Ravi");

  hm.put(101,"Vijay");

  hm.put(103,"Rahul");

**for**(Map.Entry m:hm.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }

}

### **Java TreeMap Example: Book**

import java.util.\*;

class Book implements Comparable<Book> {

private String title;

private String author;

public Book(String title, String author) {

this.title = title;

this.author = author;

}

@Override

public int compareTo(Book otherBook) {

return this.title.compareTo(otherBook.title);

}

public class TreeMapExample {

public static void main(String[] args) {

// Create a TreeMap with Book objects as keys

TreeMap<Book, Integer> bookRatings = new TreeMap<>();

// Create Book instances and use them as keys

Book book1 = new Book("The Great Gatsby", "F. Scott Fitzgerald");

Book book2 = new Book("To Kill a Mockingbird", "Harper Lee");

Book book3 = new Book("1984", "George Orwell");

// Add key-value pairs

bookRatings.put(book1, 4);

bookRatings.put(book2, 5);

bookRatings.put(book3, 4);

// Print book ratings using Book keys

for (Map.Entry<Book, Integer> entry : bookRatings.entrySet()) {

System.out.println(entry.getKey() + " - Rating: " + entry.getValue());

}

} }

**Java Hashtable class**

In Java, a `Hashtable` is a legacy class that's part of the Java Collections Framework. It's a data structure that stores key-value pairs and is similar to the more modern `HashMap` class. A `Hashtable` provides fast and efficient access to elements based on their keys and is synchronized, making it thread-safe for concurrent access.

import java.util.Hashtable;

import java.util.Map;

public class HashtableExample {

public static void main(String[] args) {

// Create a Hashtable with String keys and Integer values

Hashtable<String, Integer> ageTable = new Hashtable<>();

// Add key-value pairs

ageTable.put("Alice", 30);

ageTable.put("Bob", 25);

ageTable.put("Carol", 40);

// Retrieve values using keys

int aliceAge = ageTable.get("Alice");

System.out.println("Alice's age: " + aliceAge);

// Print all key-value pairs

for (Map.Entry<String, Integer> entry : ageTable.entrySet())

{

System.out.println(entry.getKey() + " is " + entry.getValue() + " years }

}

}

# Java Collections class

The Java Collections Framework provides a wide range of classes and interfaces to work with collections objects.

**Example:**

import java.util.\*;

public class CollectionsExample {

public static void main(String[] args) {

// ArrayList example

List<String> arrayList = new ArrayList<>();

arrayList.add("Apple");

arrayList.add("Banana");

arrayList.add("Orange");

System.out.println("ArrayList:");

for (String fruit : arrayList) {

System.out.println(fruit);

}

}

}

**Java Collections Example: Max() & Min()**

import java.util.\*;

public class CollectionsMaxMinExample {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(15, 7, 42, 23, 8, 35);

// Finding the maximum element using Collections.max()

Integer maxNumber = Collections.max(numbers);

System.out.println("Maximum number: " + maxNumber);

// Finding the minimum element using Collections.min()

Integer minNumber = Collections.min(numbers);

System.out.println("Minimum number: " + minNumber);

}

}

# Sorting in Collection

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

### **Method of Collections class for sorting List elements**

**Public Void sort(List list):** it is used to sort the elements of List elements must be of comparable type.

### **Example of Sorting the elements of List that contains string objects**

**import** java.util.\*;

**class** TestSort1{

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

ArrayList<String> al=**new** ArrayList<String>();

al.add("Viru");

al.add("Saurav");

al.add("Mukesh");

al.add("Tahir");

Collections.sort(al);

Iterator itr=al.iterator();

**while**(itr.hasNext())

{

System.out.println(itr.next());

 }

}

}

# Java Comparable interface

Java Comparable interface is used to order the objects of user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provide single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be rollno, name, age or anything else.

## Java Comparable Example

**class** Student **implements** Comparable<Student>{

**int** rollno;

String name;

**int** age;

Student(**int** rollno,String name,**int** age){

**this**.rollno=rollno;

**this**.name=name;

**this**.age=age;

}

**public** **int** compareTo(Student st){

**if**(age==st.age)

**return** 0;

**else** **if**(age>st.age)

**return** 1;

**else**

**return** -1;

}

}

**import** java.util.\*;

**import** java.io.\*;

**public** **class** TestSort3{

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

ArrayList<Student> al=**new** ArrayList<Student>();

al.add(**new** Student(101,"Vijay",23));

al.add(**new** Student(106,"Ajay",27));

al.add(**new** Student(105,"Jai",21));

Collections.sort(al);

**for**(Student st:al){

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

}

}

# Java Comparator interface

**Java Comparator interface** is used to order the objects of user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

#### **compare() method**

public int compare(Object obj1,Object obj2): compares the first object with second object.

#### **Method of Collections class for sorting List elements**

**public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

## Java Comparator Example (Generic)

**Student.java**

**class** Student{

**int** rollno;

String name;

**int** age;

Student(**int** rollno,String name,**int** age){

**this**.rollno=rollno;

**this**.name=name;

**this**.age=age;

}

}

**AgeComparator.java**

**import** java.util.\*;

**class** AgeComparator **implements** Comparator<Student>{

**public** **int** compare(Student s1,Student s2){

**if**(s1.age==s2.age)

**return** 0;

**else** **if**(s1.age>s2.age)

**return** 1;

**else**

**return** -1;

}

}

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

**import** java.util.\*;

**class** NameComparator **implements** Comparator<Student>{

**public** **int** compare(Student s1,Student s2){

**return** s1.name.compareTo(s2.name);

}

}

**Simple.java**

In this class, we are printing the objects values by sorting on the basis of name and age.

**import** java.util.\*;

**import** java.io.\*;

**class** Simple{

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

ArrayList<Student> al=**new** ArrayList<Student>();

al.add(**new** Student(101,"Vijay",23));

al.add(**new** Student(106,"Ajay",27));

al.add(**new** Student(105,"Jai",21));

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

Collections.sort(al,**new** NameComparator());

**for**(Student st: al){

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

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

Collections.sort(al,**new** AgeComparator());

**for**(Student st: al){

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

}

}

**Difference between Comparable and Comparator**

Comparable and Comparator both are interfaces and can be used to sort collection elements.

But there are many differences between Comparable and Comparator interfaces that are given below.

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc. |
| 2) Comparable **affects the original class** i.e. actual class is modified. | Comparator **doesn't affect the original class** i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method. |

**Garbage Collection**

**Garbage –** An Object without reference.

**Garbage Collector-**Collector who collects the garbage.

In C language programmer are allocating memory by using malloc() function and programmer are destroying memory by using free() , here the developer is responsible for both operations .

In CPP language programmer are allocating memory by using constructors and programmer are destroying memory by using destructors, here the developer is responsible for both operations.

In java programmer is responsible to allocate the memory by creating of object and memory will be destroyed by Garbage collector it is a part of the JVM.

Automatic Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

1. An in-use object or a reference object, means that some part of the program still maintains the pointer to the object.
2. An unused object or an unreferenced object, is no longer referenced by any part of your program. So, the memory used by an unreferenced object

Can be reclaimed.

In Programming languages like C, CPP, allocating and deallocating memory is manual process. In java, Process of deallocating memory is handled automatically by the garbage collector.

In Application level the good practice is,

* 1. Create the object.
  2. Use the object.
  3. Once the usage completed give the object to the garbage collector.

Then Garbage collector will destroy the object.

Note: Giving the object to the garbage collector is nothing but making the object without reference.

When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation.

**Advantages:**

1. It makes the java memory efficient because garbage collector removes useless objects.

2. It is automatically called by JVM no need to write extra code.

3. Garbage collector eliminating memory leaks and other memory-related problems.

There are 4 ways to give the object to the garbage collector

1. Assign null value to the object.

Test t = new Test();

t=null;

2. Name less object.

new Test();

3. Reassign Reference Variables

str1 =”java”;

str2 =”Ruby”;

str1= str2; //str1 is pointing to str2

---- java

str1 str2 ---- Ruby

Example:

String s1= new String("Vijay");  
String s2=new String("Anu");  
System.*out*.println(s1 + " " + s2);  
System.*out*.println(s1.hashCode() + "\t" + s2.hashCode());  
s1=s2;  
System.*out*.println(s1 + " " + s2);  
System.*out*.println(s1.hashCode() + "\t" + s2.hashCode());

Output:

Vijay Anu

78733042 65992

Anu Anu

65992 65992

1. Once we create the object inside the memory, Once the method is completed object is eligible to garbage collector.

Void m1()

{

Test t = new Test();

stack heap

}

The reference variable is stored in a stack memory, Object is stored in heap memory.

Once the method is completed then the stack memory variable is destroyed, then the object becomes without reference(garbage).

**Garbage Collector**

**Example:**

Class Test{

Public static void main (String[] arg)

{

Test t1 = new Test();

Test t2 = new Test();

System.out.println(t1);

System.out.println(t2);

t1 = null;

t2 = null;

System.gc();

}

1. When the JVM is running then only the garbage collector will run automatically.

The JVM is main thread which contains multiple threads,

Garbage collector

Profiler

Execution Engine … etc.

Once main thread is completing its execution all threads are automatically stopped.

Once JVM stops the garbage will stop automatically.

2. To call the garbage collector use gc() method. It is a static method of system class.

3. The finalize() method called by the garbage collector just before destroying every object. finalize() present in object class just override to write the logics.

4. Garbage collector uses “Mark & Sweep” algorithm to destroy the useless objects.

Note: In finalized method if any exceptions are raised those exceptions are ignored and objects are destroyed.

Public void finalize()

{

System.out.println(“Calling finalize method”);

System.out.println(10/0);

}

**Calling Garbage Collector**

We can call garbage collector in two ways,

System class : gc() : static Method.

Runtime class : gc() : instance method.

Note: Only Runtime class gc() method can call the garbage collector directly.

The System.gc() internally calls Runtime class gc() Method.

Class System

{

Public static void gc()

{

Runtime.getRuntime.gc();

}

}

Public static void gc()

Runs the garbage collector.

Calling the gc method suggests that the JVM expand effort towards recycling unused objects in order to make the memory, they currently occupy available for quick reuse.

Class Test{

Public void finalize()

{

System.out.println(“Finalize Method called”);

}

Public static void main (String[] arg)

{

Test t1 = new Test();

Test t2 = new Test();

System.out.println(t1);

System.out.println(t2);

t1 = null;

t2 = null;

// System.gc();

Runtime r = RunTime.getRuntime();

r.gc();

}

**Factory Method:**

The Method capable to return the object is called as Factory Method.

The Factory method internally creates & return the object using new keyword.

Example

Runtime r = RunTime.getRuntime();

Here getrunTime() is a Factory Method it returns Runtime Object.

Integer I = Integer.valueof(100);

Here valueof() is a Factory Method it returns Integer Object.

**File Handling**

To Perform IO Operations, we need classes & interfaces these are present in java.io.

Using java.io package classes & interfaces it is possible to work with only files.

Java application can read the data from text file & write the data to text file.

Java------text file.

**Channel:** It is a communication medium to transfer the data.

Every Channel can perform 2 operations

1. Read Operations

2. Write

There are two types of channels,

**1 Byte oriented channel**

The data is transferred in the form of Bytes. – 8-bit image data

FileInputStream: To Read the data.

FileoutputStream: To write the data.

**2 Character oriented channel**

The data is transferred in the form of characters. – 16-bit text data

FileReader: To Read the data.

FileWriter: To write the data.

Steps to design the application

1. Create the channel.

2. read and write operations.

To read the data : use read() Method.

To write the data : use write() method.

3. close the resources.

Close the channels.

e.g. abc.txt to xyz.txt

Note: To read the data file is mandatory but to write the data file is optional.

To write the data, If the file is not available then it will create and write the data.

If the file is already existing it will override the data.

public class Normalstrreams {  
 public static void main(String[] args) throws IOException {  
  
 //Create the channel  
 FileReader reader = new FileReader("abc.txt");  
 FileWriter writer = new FileWriter("xyz.txt");  
  
 //read & write the operations  
 int c;  
 while ((c = reader.read())!=-1)  
 {  
 System.*out*.println(c);  
 writer.write(c);  
 }  
 // close the resources.  
 reader.close();  
 writer.close();  
 System.*out*.println("Resources are released...");  
 }  
}

If the application terminated normally the resources are released.

In Above example read & write operations failed. the application terminated abnormally, so the rest of the application is not executed. The resources will not release.

To overcome above problem, to release the resources use finally block.

**Example:**

With try catch & finally block.

try

{

Create the channel, Read and write the operations...

}

Catch(Exception e)

{

Handle the exceptions;

}

Finally

{

Release the resources

}

package FileHandling;  
  
import java.io.\*;  
  
public class BufferedStream {  
  
 public static void main(String[] args) {  
 BufferedReader reader =null;  
 BufferedWriter writer =null;  
  
 try {  
 reader = new BufferedReader(new FileReader("abc.txt"));  
 writer = new BufferedWriter(new FileWriter("zzz.txt"));  
 String line;  
 while( (line = reader.readLine()) !=null)  
 {  
 System.*out*.println(line);  
 writer.write(line);  
 writer.newLine();  
 }  
 }catch(IOException e)  
 {  
 System.*out*.println(e.getMessage());  
 }finally {  
 try {  
 reader.close();  
 writer.close();

System.*out*.println("Resources are released...");   
 }catch(IOException e)  
 {  
 System.*out*.println(e.getMessage());  
 }  
 }  
 }  
}

Above two examples are of normal streams,

1. Reading the data char by char.

2. for every character it is interact with Hard Disk.

To overcome above problems use Buffered streams

1. Reading the data line by line.

2. one time reading the data from HD stored into buffer memory later it is reading the data from buffer.

**1. Byte oriented channel(image)**

BufferedInputStream

BufferedOutputStream

2 **character oriented channel**.(**Text**)

BufferedReader

BufferedWriter

Buffered Streams are developed based on the normal streams.

Buffered Streams can read the data only from buffer memory.

**Normal Streams vs BufferedStreams**

**Normal Stream:**

FileReader : new FileReader(“abc.txt”)

FileWriter : new FileWriter(“xyz.txt”)

FileInputStream : new FileInputStream(“test.jpg”)

FileOutputStream : new FileOutputStream(“test1.jpg”)

1. Read the data char by char, if file contains more characters, it required more read and write operations, it will affect on performance.
2. Normal Streams interact with Hard Disk to read the data.

**Buffer Stream:**

BufferedReader : new BufferedReader (new FileReader(“abc.txt”))

BufferedWriter : new Bufferedwriter (new FileWriter(“xyz.txt”))

BufferedInputStream : new Bufferedinputstream

(new FileInputStream(“test.jpg”))

BufferedOutputStream : new BufferedOutputStream (new FileOutputStream(“test1.jpg”))

1. Read the data line by line format, it improves the performance.
2. First time read the data from hard disk, it will store into buffer memory

Later if we need the data it will read from buffered memory.

**Note:** NormalStreams can read the data from only Hard disk. Not possible to interact with buffered memory.

Buffer Streams can read the data from only buffered memory. Not possible to interact with Hard Disk.

**Examples on BufferedStream**

**Ex.1** With try catch & finally block

package FileHandling;  
  
import java.io.\*;  
  
public class BufferedStream {  
  
 public static void main(String[] args) {  
 BufferedReader reader =null;  
 BufferedWriter writer =null;  
  
 try {  
 reader = new BufferedReader(new FileReader("abc.txt"));  
 writer = new BufferedWriter(new FileWriter("zzz.txt"));  
 String line;  
 while( (line = reader.readLine()) != null)  
 {  
 System.*out*.println(line);  
 writer.write(line);  
 writer.newLine();  
 }  
 }catch(IOException e)  
 {  
 System.*out*.println(e.getMessage());  
 }finally {  
 try {  
 reader.close();  
 writer.close();  
 }catch(IOException e)  
 {  
 System.*out*.println(e.getMessage());  
 }  
 }  
 }  
}

**Ex.2**

**Try with Resources concept**

Declare the resources using try block, once the try block is completed the resources are automatically released.

It releases automatically means it internally uses AutoCloseable interface, which contains close method.

The close method is automatically called or invoked on objects managed by the try with resource statement.

package FileHandling;  
  
import java.io.\*;  
  
public class BufferedImagestream {  
 public static void main(String[] args) {  
  
 BufferedInputStream reader = null;  
 BufferedOutputStream writer =null;  
 try (BufferedInputStream inputStream = new BufferedInputStream(new FileInputStream("test.jpg"));  
 BufferedOutputStream outputStream = new BufferedOutputStream(new FileOutputStream("New.jpg"))  
 )  
 {  
 int c;  
 while((c = inputStream.read()) != -1)  
 {  
 outputStream.write(c);  
 }  
  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 System.*out*.println("Operations are completed...");  
 }  
}

**Multithreading**

A thread is a thread of execution in a program. The java virtual machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Thread with higher priority is executed in preference to threads with lower priority.

**Thread vs Process**

1. Threads are light weight tasks are created by JVM.

Processes are heavy weight. Created by Operating system.

1. One process contains multiple threads every thread is independent thread.

e.g. Browser is process and every tab is called thread.

**There are two types of multi-tasking:**

1. Process based multi-tasking (Process based Request processing)

When client sends the request to the server, every request one process is created. If the number of requests is increases then number of processes increases it affects on the performance because processor load increases.

1. Thread based multi-tasking (Thread based Request processing)

When client sends the request to the server, every request one thread is created. If the number of requests is increases then number of threads increases then performance is better because processor load decreases.

**Multitasking vs Multithreading**

1. **Multitasking:**

Executing more than one task simultaneously is called multitasking.

Every task is independent.

e.g. operating system level we have multitasking.

Writing the code in IntelliJ idea

Downloading file.

Listening music etc.

1. **Multithreading:**

Executing more than one thread simultaneously is called multitasking.

Every thread is independent.

e.g. In programming level we have multithreading.

JVM running, Garbage collection etc.

**Programming models**

There are two types of programming models

1. **Single threaded model/Uni programming**

Whenever the execution starts from main method one thread is started

automatically is called main thread.

Class Test

{

public static void main (String[] args) //main Thread Starts

{

System.out.println(“Good Morning….”); //main Thread executing

System.out.println(Thread.currentThrerad().getName());

} // main thread ended.

}

1. **Multithreaded model**

The application contains more than one thread.

We can create the thread in two ways.

1. **extends Threads**
2. **implements Runnable**

Thread class & Runnable interface present in java.lang package

1. **First approach to create thread extending Thread class:**

Step 1: Our normal java class will become Thread class when we extend predefined Thread class.

class My Thread extends Thread

{

}

Step 2: Override the run() method to write the business logic of the Thread. The run method presents in Thread class with empty implementation.

class MyThread extends Thread

{

}

public void run()

{ //logics here

}

Step 3: Create object of user defined Thread class.

MyThread t = new MyThread();

Step 4: Start the Thread by using start() method of Thread class. t.start();

ex-1: class MyThread extends Thread

{

@override

//logics of user thread public void run()

{

for (int i=0; i<10; i++)

{

System.out.println("user defined Thread");

}

}

}

Public class ThreadDemo

{

public static void main(String[] args)throws InterruptedException

{

MyThread t = new MyThread(); //creating Runtime class Object

t.start();

t.start();

for(int i = 1; i<=10 ; i++)

{

System.out.println("main Thread logics… ");

}

}

}

Public void run()

Used to write the thread logics.

Subclasses of Thread should override this method to write the logics.

Public void start()

Causes this thread to begin execution;

The JVM calls the run method of this thread.

The result is that two threads are running concurrently. The current thread (which returns from the call to the start method.) and the other thread (which executes its run method)

Current thread: main thread

Other thread: user thread

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

**Importance of main Thread**

1. Execution always starts with main Thread.
2. Starts the user defined threads inside the main method.

**Thread Scheduler**

If the application contains more than one thread then thread execution decided by the Thread scheduler.

Thread scheduler is unexpected behaviour so multi-threaded example we will unexpectedly output.

Application areas of multithreading

Server side implemented multithreaded environment, because the server needs to handle the multiple requests at a time. Every Request is a thread.

The main objective of multithreading is to develop the gamming application where the multiple objects are moving at a time.

**Thread life cycle of process**

New/born ---🡪 ready/Runnable ---🡪 Running ---🡪 dead.

New/born : MyThread t = new MyThread();

t.start() : Ready to start, once if thread scheduler allocates the CPU

Running : run() method executing.

Dead : once the run method completed thread goes to dead state.

**Observations**

1. Case 1: It is always recommended to override the run() method to write the thread logics.

If we are not overriding the run() method then parent (thread) run(), method executed having empty implementation.

Class MyThread extends Thread{

run(){

}

}

Case 2: Overriding start() method is not recommended. When we override start() method then our class start() method gets executed.Thread will not start. Thread will be started only when thread class start executed.

Class Mythread extends Thread

{

@Override

Public void start()

{

for(int i= 1; i<=4; i++){

System.out.println(“User defined Thread”);

}  
 }

}

Case 3: Can we overload the run method? - yes

Start() method begins thread execution, java virtual machine calls the 0-argument run method of this thread.

When we overload run() method the JVM will calls always 0-arguments run() method.

Class Mythread extends Thread

{

Public void run()

{

System.out.println(“0-argument Run Method”);

}

Public void run(int a)

{

System.out.println(“1-argument Run Method”);

}

}

Class ThreadDemo

{

Public static void main(String [] args)

{

MyThread t = new MyThread();

t,start();

}

}

Case 4: No need to take separate class for main thread.

Case 5: It is not legal to start a thread more than once. In particular, a thread may not be restarted once it has completed the execution.

Throws IllegalThreadStateException – if the thread is already started.

Case 6: No need to write all the logic in One run() Method, write your own methods and call all those methods in run() method.

Casse 7: Thread Names

Every thread has a name for identification purpose. More than one thread may have the same name. If a name is not specified when thread is created a new name is generated for it.

Every thread is having a names,

The default name of main thread is main.

The default names for user defined threads starts from Thread-0, Thread-1, Thread-2….;

It is possible to set the user defined names to the threads using setName() method.

Public final void setName(String Name)

It is possible to get the names to the threads using getName() method.

Public final String getName()

Ex:

Class MyThread extends Thread

{

}

Class ThreadDemo1

{

Public static void main(String [] args)

{

MyThread t1 = new MyThread();

MyThread t2 = new MyThread();

t1.start();

t2.start();

System.out.println(t1.getName());

System.out.println(t2.getName());

System.out.println(Thread.currentThread().getName());

t1.setName(“Geeta”);

t2.setName(“Rohini”);

Thread.currentThread().setName(“Sonali”);

System.out.println(t1.getName());

System.out.println(t2.getName());

System.out.println(Thread.currentThread().getName());

}}

Case 8: To stop the Thread few seconds we use sleep() Method.

Class WishThread extends Thread

{

Public void run()

{

For (int i= 1; i<=4; i++)

{

System.out.println(“Good morning…”);

try {

Thread.sleep(2000);

} catch (InterruptedException e)

{

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

}

}  
 }

}

Class ThreadDemo

{

Public static void main(String [] args)

{

MyThread t = new MyThread();

t,start();

}

}

**Synchronization:**

Synchronized modifier - It is only for methods.

By default, all threads are non-synchronized.

Synchronized-only one thread can access: thread safe: performance is decreased.

Non-Synchronized-multiple threads can access: non thread safe: performance is increased.

class Greetings

{

public static Synchronized void wish(String name)

for (int i= 1; i<=4; i++)

{

System.out.println(“Good morning…” + name);

try {

Thread.sleep(2000);

} catch (InterruptedException e)

{

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

}

}  
 }

Class MyThread1 extends Thread

{

@Override

Public void run()

{

Greetings.wish(“Ratna”);

}

}

Class MyThread2 extends Thread

{

@Override

Public void run()

{

Greetings.wish(“Mansi”);

}

}

Class MyThread3 extends Thread

{

@Override

Public void run()

{

Greetings.wish(“Vijay”);

}

}

Class ThreadDemo

{

Public static void main(String [] args)

{

MyThread1 t1 = new MyThread1();

t1,start();

MyThread2 t2 = new MyThread2();

t2,start();

MyThread3 t3 = new MyThread();

t3,start();

}

}

**Join() Method:**

Public final void **join()** throws InnterptedException

* Waits for this Thread for Termination.
* Other threads wait until current Thread terminates.

Public final void **join(**2 milliseconds**)** throws InnterptedException

* Waits at most 2 milliseconds, for this Thread for Termination.
* Other threads wait up to 2 milliseconds for current Thread to terminates.

After 2 milliseconds other threads starts execution.

1. **Second approach to create thread by implementing Thread class:**

Step- 1To create the thread our class is implements Runnable interface.

class MyRunnable implements Runnable()

{

}

Step -2 Override the run method to write the logics.

class MyRunnable implements Runnable()

{

@Override

Public void run()

{

//logics

}

}

Step -3 Instantiate the Runnable Class

MyRunnable m = new MyRunnable();

Step -4 Start the Thread using start() method.

Thread t = new Thread(m);

t.start();

**Example**:

class MyRunnable implements Runnable()

{

@Override

Public void run()

{

System.out.println(“User Defined class”);

}

}

Class ThreadDemo

{

Public static void main(String [] args)

{

MyRunnable m1 = new MyRunnable (); Thread t = new Thread(m1);

t.start();

}

}

**Aggregation & Composition**

**Aggregation –** One class has reference of another class is called as Aggregation.

Aggregation is concept of reusing without extending, just by declaring the properties. Aggregation is also known as “has +a” relation.

Aggregation is week relation.

Example:

class Address(){

}

class Student(){

Address addr;

}

class Employee(){

Address addr;

}

Here once student object destroyed address class can be used by another class.

**Composition -**

Strong aggregation is called as **Composition**.

Class college1

{

}

Class college2

{

}

class University()

{

College col1;

College col2;

}

Here university object destroyed then college object become useless.