

BASICS OF OBJECT ORIENTED PROGRAMMING

OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) is a programming language model organized around objects rather than actions and data. An object-oriented program can be characterized as data controlling access to code. Concepts of OOPS

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

OBJECT

Object means a real world entity such as pen, chair, table etc. Any entity that has state and behavior is known as an object. Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing details of each other's data or code, the only necessary thing is that the type of message accepted and type of response returned by the objects.

An object has three characteristics:

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

CLASS

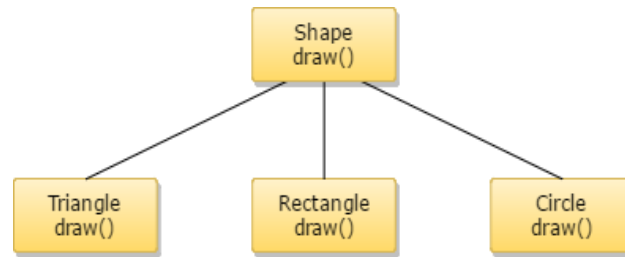
Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. A class consists of Data members and methods. The primary purpose of a class is to hold data/information. The member functions determine the behavior of the class, i.e. it provides a definition for supporting various operations on data held in the form of an object. Class doesn't store any space.

INHERITANCE

Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another, i.e., acquiring the properties and behavior of child class from the parent class. When one object acquires all the properties and behaviors of another object, it is known as inheritance. It provides code reusability and establishes relationships between different classes. A class which inherits the properties is known as Child Class (sub-class or derived class) whereas a class whose properties are inherited is known as Parent class (super-class or base class). Types of inheritance in java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.

POLYMORPHISM

When one task is performed by different ways then it is known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.



Polymorphism is classified into two ways:

Method Overloading (Compile time Polymorphism)

Method Overloading is a feature that allows a class to have two or more methods having the same name but the arguments passed to the methods are different. Compile time polymorphism refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time.

Method Overriding (Run time Polymorphism)

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.

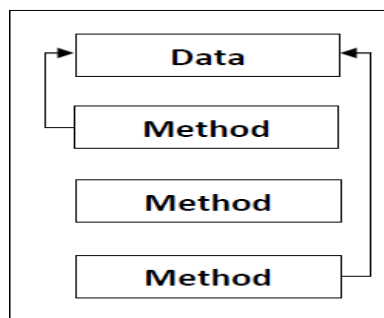
ABSTRACTION

Abstraction is a process of hiding the implementation details and showing only functionality to the user. For example: phone call, we don't know the internal processing. In java, we use abstract class and interface to achieve abstraction.

ENCAPSULATION

Encapsulation in java is a process of wrapping code and data together into a single unit, for example capsule i.e. mixed of several medicines. A java class is the example of encapsulation.

Class



DIFFERENCE BETWEEN PROCEDURE-ORIENTED AND OBJECT-ORIENTED PROGRAMMING

Procedure-Oriented Programming	Object-Oriented Programming
In POP, program is divided into small parts called functions	In OOP, program is divided into parts called objects .
In POP, Importance is not given to data but to functions as well as sequence of actions to be Done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world .
POP follows Top Down approach .	OOP follows Bottom Up approach .
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is less secure .	OOP provides Data Hiding so provides more security .
In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

CONSTRUCTORS

- ❑ Constructors are special member functions whose task is to initialize the objects of its class.
- ❑ It is a special member function; it has the same as the class name.
- ❑ Java constructors are invoked when their objects are created. It is named such because, it constructs the value, that is provides data for the object and are used to initialize objects.
- ❑ Every class has a constructor when we don't explicitly declare a constructor for any java class the compiler creates a default constructor for that class which does not have any return type.
- ❑ The constructor in Java cannot be abstract, static, final or synchronized and these modifiers are not allowed for the constructor.

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Default constructor (no-arg constructor)

A constructor having no parameter is known as default constructor and no-arg constructor.

Parameterized Constructors

A constructor which has a specific number of parameters is called parameterized constructor. Parameterized constructor is used to provide different values to the distinct objects.

DESTRUCTORS:

A destructor is a special method called automatically during the destruction of an object. Actions executed in the destructor include the following:

- Recovering the heap space allocated during the lifetime of an object
- Closing file or database connections
- Releasing network resources
- Releasing resource locks

Destructors are called explicitly in C++. However, in C# and Java this is not the case, as the allocation and release of memory allocated to objects are implicitly handled by the garbage collector. While destructors in Java have to be explicitly invoked by finalizers since their invocation is not guaranteed.

ACCESS CONTROL:

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

The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.

1) private access modifier

The private access modifier is accessible only within 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.

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.

4) public access modifier

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

STRING HANDLING:

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

There are two ways to create String object:

1. By string literal
2. By new keyword

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

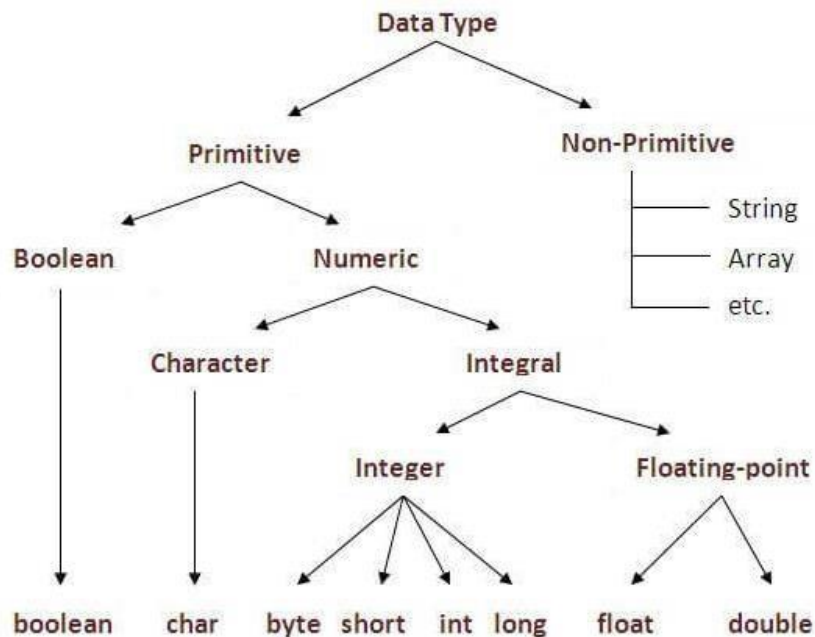
Example:

```
char[] ch={'j','a','v','a'};
String s=new String(ch);
String s="java";
```

DATATYPES IN JAVA

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

- Primitive data types:** The primitive data types include Integer, Character, Boolean, and Floating Point.
- Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

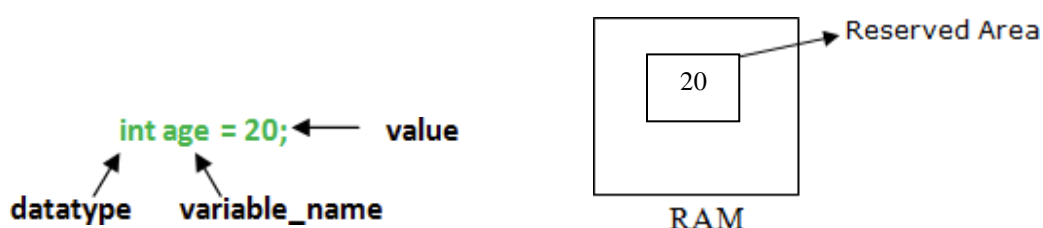


Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. These can be put in four groups:

- **Integers:** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers:** This group includes float and double, which represent numbers with fractional precision.
- **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes boolean, which is a special type for representing true/false values.

VARIABLES

A variable is a container which holds the value and that can be changed during the execution of the program. A variable is assigned with a datatype. Variable is a name of memory location. All the variables must be declared before they can be used. There are three types of variables in java: local variable, instance variable and static variable.



1) Local Variable

A variable defined within a block or method or constructor is called local variable.

- These variable are created when the block is entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.

Example:

```
import java.io.*;
public class StudentDetails
{
    public void StudentAge()
    { //local variable age
        int age = 0;
        age = age + 5;
        System.out.println("Student age is : " + age);
    }
    public static void main(String args[])
    {
        StudentDetails obj = new StudentDetails();
        obj.StudentAge();
    }
}
```

Output:

Student age is : 5

2) Instance Variable

Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.

Example:

```
import java.io.*;
class Marks{
    int m1;
    int m2;
}
class MarksDemo
{
    public static void main(String args[])
    { //first object
        Marks obj1 = new Marks();
        obj1.m1 = 50;
        obj1.m2 = 80;
        //second object
        Marks obj2 = new Marks();
        obj2.m1 = 80;
```

```

obj2.m2 = 60;
//displaying marks for first object
System.out.println("Marks for first object:");
System.out.println(obj1.m1);
System.out.println(obj1.m2);
//displaying marks for second object
System.out.println("Marks for second object:");
System.out.println(obj2.m1);
System.out.println(obj2.m2);
}}

```

Output:

Marks for first object:

50

80

Marks for second object:

80

60

3) Static variable

Static variables are also known as Class variables.

- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at start of program execution and destroyed automatically when execution ends.

Example:

```

import java.io.*;
class Emp {

    // static variable salary
    public static double salary;
    public static String name = "Vijaya";
}
public class EmpDemo
{
    public static void main(String args[]) {

        //accessing static variable without object
        Emp.salary = 10000;
        System.out.println(Emp.name + "'s average salary:" + Emp.salary);
    }
}

```

Output:

Vijaya's average salary:10000.0

Difference between Instance variable and Static variable

INSTANCE VARIABLE	STATIC VARIABLE
Each object will have its own copy of instance variable	We can only have one copy of a static variable per class irrespective of how many objects we create.

Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of instance variable	In case of static changes will be reflected in other objects as static variables are common to all object of a class.
We can access instance variables through object references	Static Variables can be accessed directly using class name.
Class Sample { int a; }	Class Sample { static int a; }

OPERATORS IN JAVA

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

- Arithmetic Operators
- Increment and Decrement
- Bitwise Operators
- Relational Operators
- Boolean Operators
- Assignment Operator
- Ternary Operator

Arithmetic Operators

Arithmetic operators are used to manipulate mathematical expressions

Operator Result

Operator	Result
+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

Example:

// Demonstrate the basic arithmetic operators.

```
class BasicMath
```

```
{
```

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

```
{
```

```
// arithmetic using integers
```

```
System.out.println("Integer Arithmetic");
```

```
int a = 1 + 1;
```

```
int b = a * 3;
```

```
int c = b / 4;
```

```
int d = c - a;
```



```

int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}}

```

Output:

Integer Arithmetic

a = 2

b = 6

c = 1

d = -1

e = 1

Floating Point Arithmetic

da = 2.0

db = 6

dc = 1.5

dd = -0.5

de = 0.5

Modulus Operator

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

Example:

// Demonstrate the % operator.

```

class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}

```

Output:

x mod 10 = 2

y mod 10 = 2.25

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

Syntax:

```
var op= expression;
```

Example:

```
// Demonstrate several assignment operators.
```

```
class OpEquals
{
public static void main(String args[])
{
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
c += a * b;
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

Output:

```
a = 6
b = 8
c = 3
```

Increment and Decrement Operators

The ++ and the -- are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one.

Example:

```
// Demonstrate ++.
```

```
class IncDec
{
public static void main(String args[])
{
int a = 1;
int b = 2;
```

```

int c;
int d;
c = ++b;
d = a++;
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}

```

Output:

```

a = 2
b = 3
c = 4
d = 1

```

Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise Logical Operators

The bitwise logical operators are **&**, **|**, **^**, and **~**. the bitwise operators are applied to each individual bit within each operand.

A	B	A B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Example:

```
// Demonstrate the bitwise logical operators.
```

```

class BitLogic
{
public static void main(String args[])
{
String binary[] = {"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b)|(a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}

```

Output:

```

a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
~a = 1100

```

Left Shift Operator

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

Example:

```

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

```

Output:

```

40
80
80

```

Right Shift Operator

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

Example:

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

Output:

```
2
5
2
```

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The outcome of these operations is a **boolean** value.

Boolean Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
<code>&</code>	Logical AND
<code> </code>	Logical OR
<code>^</code>	Logical XOR (exclusive OR)
<code> </code>	Short-circuit OR
<code>&&</code>	Short-circuit AND
<code>!</code>	Logical unary NOT
<code>&=</code>	AND assignment
<code> =</code>	OR assignment
<code>^=</code>	XOR assignment
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>?:</code>	Ternary if-then-else

Example:

// Demonstrate the boolean logical operators.

```
class BoolLogic
{
public static void main(String args[])
{
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
```

```

boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
}
}

```

Output:

```

a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a=false

```

In the output, the string representation of a Java **boolean** value is one of the literal values **true** or **false**. Java AND Operator Example: Logical && and Bitwise &

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

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

Example:

```

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

```

Output:

```

false
false

```

Assignment Operator

The *assignment operator* is the single equal sign, =.

Syntax:

var = expression;

Here, the type of *var* must be compatible with the type of *expression*.

```
int x, y, z;
```

```
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement.

Ternary Operator

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

Syntax:

expression1 ? expression2 : expression3

Example:

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

Output:

2

CONTROL STATEMENTS

Selection Statements in Java

A programming language uses control statements to control the flow of execution of program based on certain conditions.

Java's Selection statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

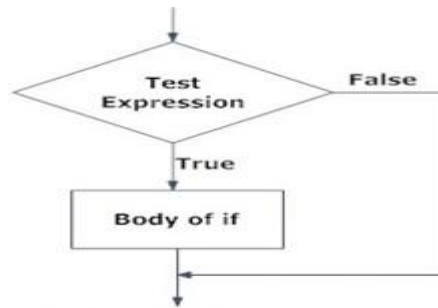
if Statement

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not that is if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
    //statements to execute if
    //condition is true
}
```

Condition after evaluation will be either true or false. If the value is true then it will execute the block of statements under it. If there are no curly braces '{' and '}' after **if(condition)** then by default if statement will consider the immediate one statement to be inside its block.



Example:

```

class IfSample
{
public static void main(String args[])
{
int x, y;
x = 10;
y = 20;
if(x < y)
System.out.println("x is less than y");
x = x * 2;
if(x == y)
System.out.println("x now equal to y");
x = x * 2;
if(x > y)
System.out.println("x now greater than y");
// this won't display anything
if(x == y)
System.out.println("you won't see this");
}
}
  
```

Output:

```

x is less than y
x now equal to y
x now greater than y
  
```

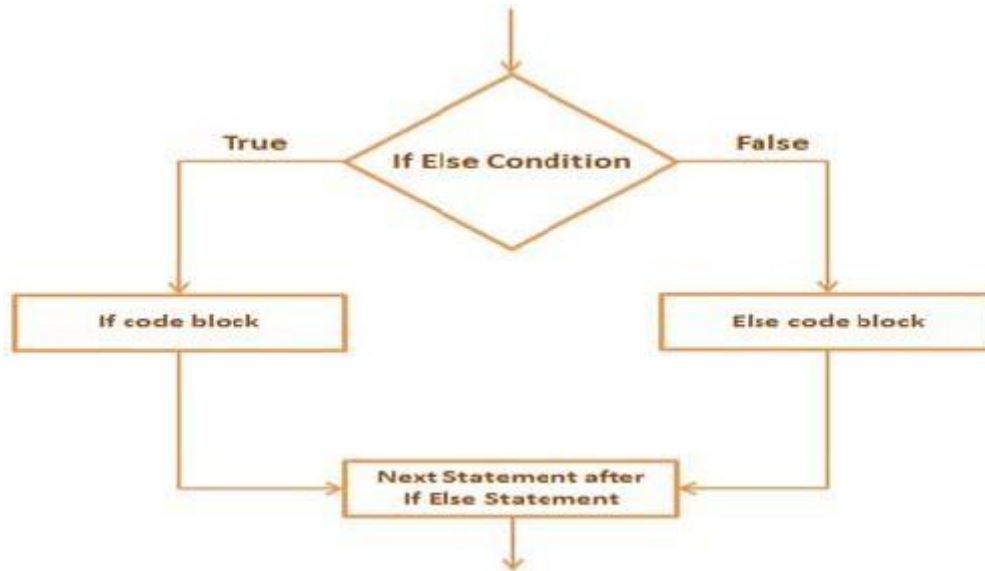
if-else Statement

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

Syntax:

```

If(condition)
{
    //Executes this block if
    //condition is true
}
else
{
    //Executes this block if
    //condition is false
}
  
```

Example:

```

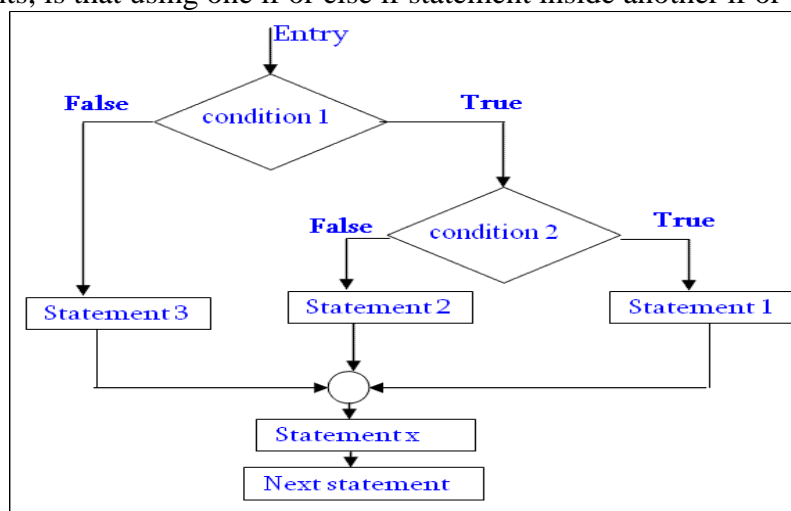
public class IfElseExample
{
    public static void main(String[] args)
    {
        int number=13;
        if(number%2==0){
            System.out.println("even number");
        }else
        {
            System.out.println("odd number");
        }
    }
}
  
```

Output:

odd number

Nested if Statement

Nested if-else statements, is that using one if or else if statement inside another if or else if statement(s).



Example:

// Java program to illustrate nested-if statement

class NestedIfDemo

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

        if (i == 10)
        {
            if (i < 15)
                System.out.println("i is smaller than 15");
            if (i < 12)
                System.out.println("i is smaller than 12 too");
            else
                System.out.println("i is greater than 15");
        }
    }
}
```

Output:

i is smaller than 15

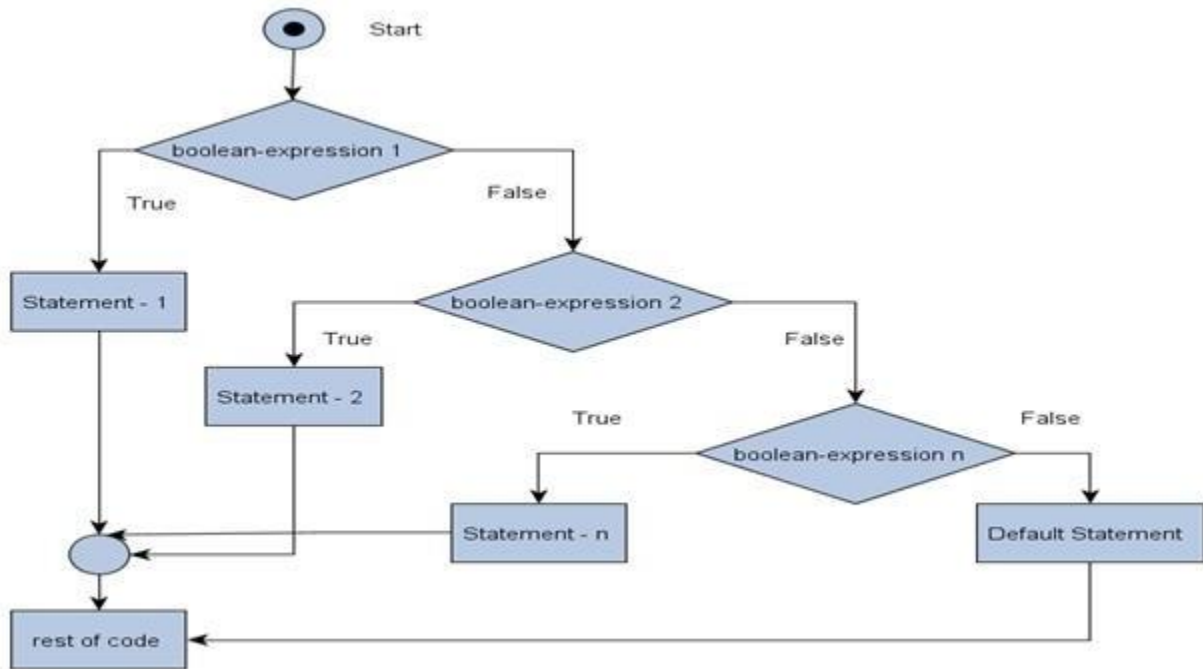
i is smaller than 12 too

if-else-if ladder statement

The *if* statements are executed from the top down. The conditions controlling the *if* is true, the statement associated with that *if* is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final *else* statement will be executed.

Syntax:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
else
statement;
```



Example:

```

public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;
    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}

```

Output:

C grade

Switch Statements

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to

different parts of your code based on the value of an expression.

Syntax:

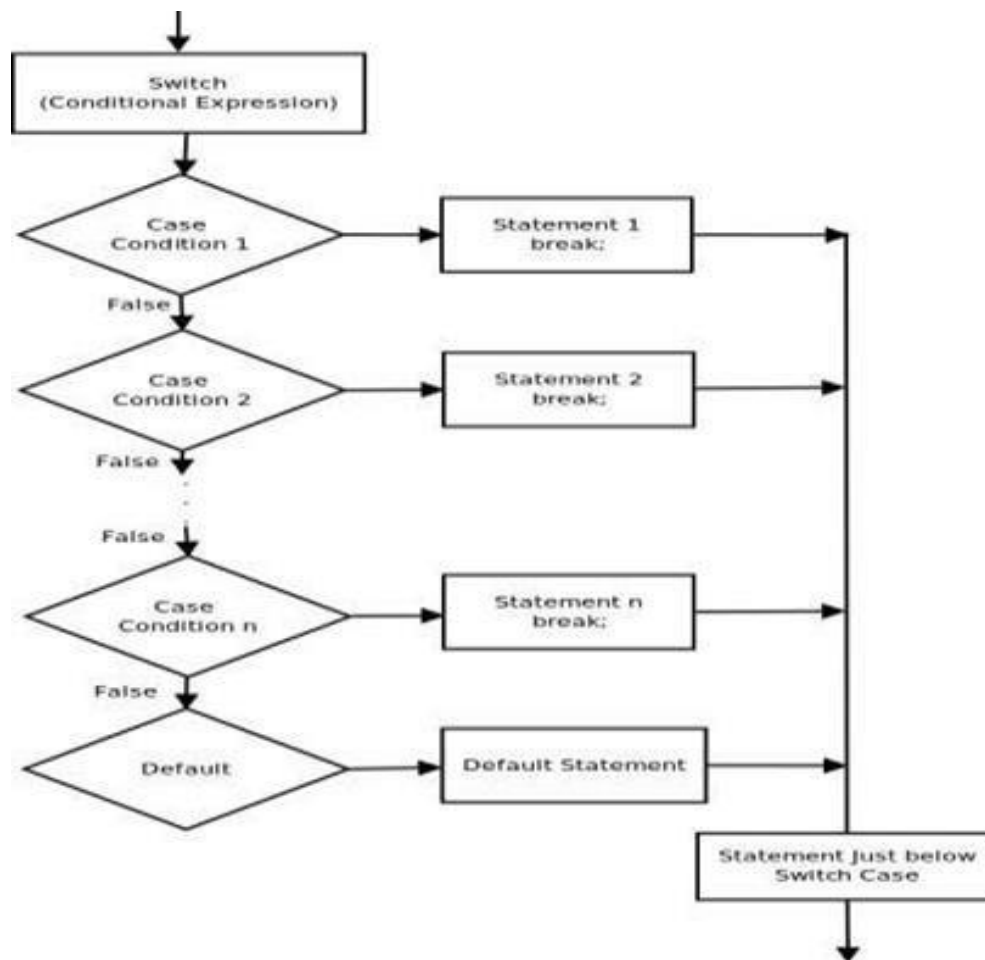
```
switch (expression) {  
  case value1:  
    // statement sequence  
    break;  
  case value2:  
    // statement sequence  
    break;  
  .  
  .  
  case valueN :  
    // statement sequence  
    break;  
  default:  
    // default statement sequence  
}
```

Example:

```
// A simple example of the switch.  
class SampleSwitch {  
  public static void main(String args[]) {  
    for(int i=0; i<6; i++)  
      switch(i) {  
        case 0:  
          System.out.println("i is zero.");  
          break;  
        case 1:  
          System.out.println("i is one.");  
          break;  
        case 2:  
          System.out.println("i is two.");  
          break;  
        case 3:  
          System.out.println("i is three.");  
          break;  
        default:  
          System.out.println("i is greater than 3.");  
      }  
  }  
}
```

Output:

```
i is zero.  
i is one.  
i is two.  
i is three.  
i is greater than 3.  
i is greater than 3.
```



ITERATIVE STATEMENTS

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

- ☐ while loop
- ☐ do-while loop
- ☐ For loop

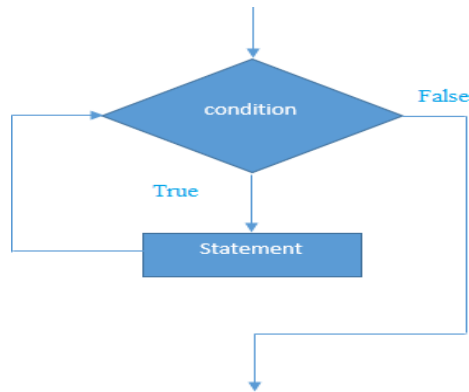
while loop

A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax:

```
while(condition) {
// body of loop
}
```

- ☐ While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. It is called as Entry controlled loop.
- ☐ Normally the statements contain an update value for the variable being processed for the next iteration.
- ☐ When the condition becomes false, the loop terminates which marks the end of its life cycle.



- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. It is called as Entry controlled loop.
- Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

Example:

// Demonstrate the while loop.

```

class While {
public static void main(String args[]) {
int n = 5;
while(n > 0) {
System.out.println("tick " + n);
n--;
}
}
}
  
```

Output:

```

tick 5
tick 4
tick 3
tick 2
tick 1
  
```

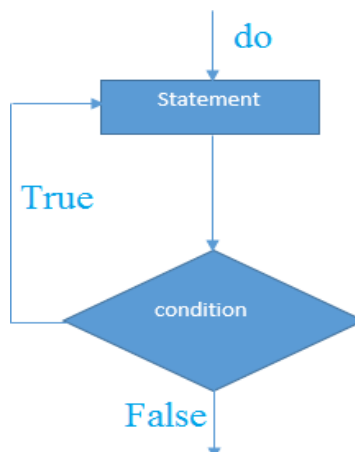
do-while loop:

do while loop checks for condition after executing the statements, and therefore it is called as Exit Controlled Loop.

Syntax:

```

do {
// body of loop } while (condition);
  
```



- ❑ do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
- ❑ After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
- ❑ When the condition becomes false, the loop terminates which marks the end of its life cycle.
- ❑ It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

Example

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
        i++;
    }while(i<=5);
}
}
```

Output:

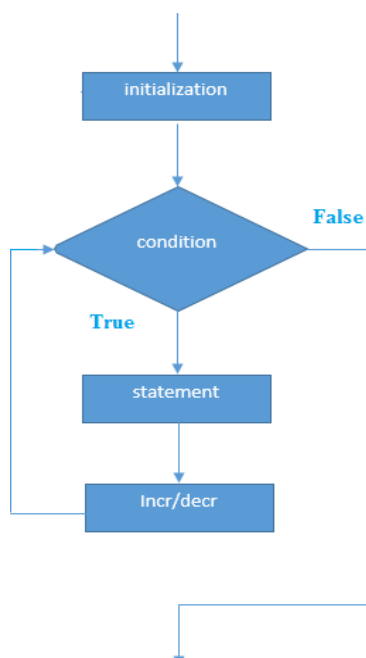
```
1
2
3
4
5
```

for loop

for loop provides a concise way of writing the loop structure. A for statement consumes the initialization, condition and increment/decrement in one line.

Syntax

```
for(initialization; condition; iteration) {
// body
}
```



- ❑ **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
- ❑ **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
- ❑ **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
- ❑ **Increment/ Decrement:** It is used for updating the variable for next iteration.
- ❑ **Loop termination:** When the condition becomes false, the loop terminates marking the end of its life cycle.

Example

```
public class ForExample {
public static void main(String[] args) {
    for(int i=1;i<=5;i++){
        System.out.println(i);
    }
} }
```

Output:

```
1
2
3
4
5
```

for-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

for(type itr-var : collection) statement-block

Example:

```
// Use a for-each style for loop.
class ForEach {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values
for(int x : nums) {
System.out.println("Value is: " + x);
sum += x;
}
System.out.println("Summation: " + sum);
}
}
```

Output:

```
Value is: 1
```


Summation: 55

Nested Loops

Java allows loops to be nested. That is, one loop may be inside another.

Example:

```
// Loops may be nested.
```

}

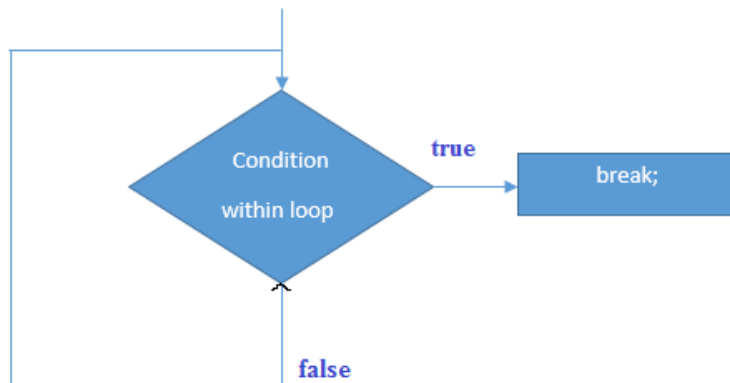
Output:

•

JUMP STATEMENTS

Java Break Statement

- The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.



Example:

// Using break to exit a loop.

```
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

Output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Java Continue Statement

- The continue statement is used in loop control structure when you need to immediately jump to the next iteration of the loop. It can be used with for loop or while loop.
- The Java *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Example:

// Demonstrate continue.

```
class Continue {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
}
}
}
```

This code uses the % operator to check if **i** is even. If it is, the loop continues without printing a newline.

Output:

```
0 1
2 3
4 5
```

6 7

8 9

Return

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

Example:

// Demonstrate return.

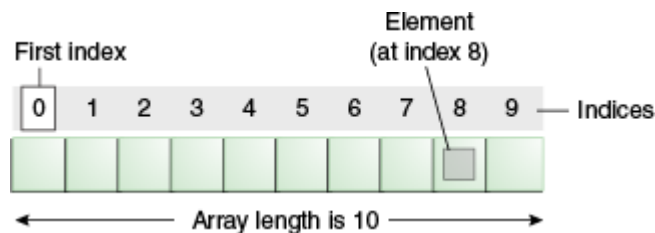
```
class Return {  
    public static void main(String args[]) {  
        boolean t = true;  
        System.out.println("Before the return.");  
        if(t) return; // return to caller  
        System.out.println("This won't execute.");  
    }  
}
```

Output:

Before the return.

ARRAYS

- ❑ Array is a collection of similar type of elements that have contiguous memory location.
- ❑ In Java all arrays are dynamically allocated.
- ❑ Since arrays are objects in Java, we can find their length using member length.
- ❑ A Java array variable can also be declared like other variables with [] after the data type.
- ❑ The variables in the array are ordered and each have an index beginning from 0.
- ❑ Java array can be also be used as a static field, a local variable or a method parameter.
- ❑ The **size** of an array must be specified by an int value and not long or short.
- ❑ The direct superclass of an array type is Object.
- ❑ Every array type implements the interfaces Cloneable and java.io.Serializable.



Advantage of Java Array

- ❑ **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- ❑ **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- ❑ **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

1. One- Dimensional Array
2. Multidimensional Array

One-Dimensional Arrays

An array is a group of like-typed variables that are referred to by a common name. An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. We can also create an array of other primitive data types like char, float, double..etc or user defined data type(objects of a class).Thus, the element type for the array determines what type of data the array will hold.

Syntax:

```
type var-name[ ];
```

Instantiation of an Array in java

```
array-var = new type [size];
```

Example:

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

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

Example:

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

Output:

```
33
3
4
5
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Example:

```

class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
if(min>arr[i])
min=arr[i];
System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};
min(a);//passing array to method
}}

```

Output:

3

Multidimensional Arrays

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array. These are also known as [Jagged Arrays](#). A multidimensional array is created by appending one set of square brackets ([]) per dimension.

Syntax:

type var-name[][]=new type[row-size][col-size];

Example:

// Demonstrate a two-dimensional array.

```

class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}

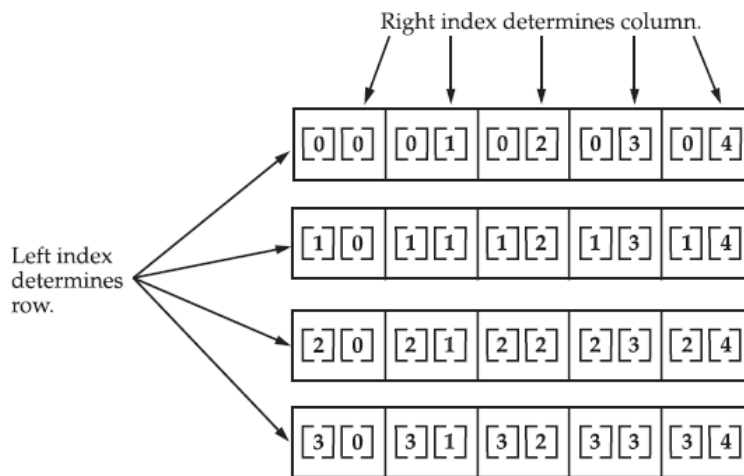
```

Output:

```

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

```



Given: `int twoD [] [] = new int [4] [5];`

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

Syntax:

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

Example:

// Manually allocate differing size second dimensions.

```
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

Output:

0

1 2
3 4 5
6 7 8 9

The array created by this program looks like this:

[0][0]			
[1][0]	[1][1]		
[2][0]	[2][1]	[2][2]	
[3][0]	[3][1]	[3][2]	[3][3]

PACKAGES

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.

Defining a Package

To create a package include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If **package** statement is omitted, the class names are put into the default package, which has no name.

Syntax:

package <fully qualified package name>;

package *pkg*;

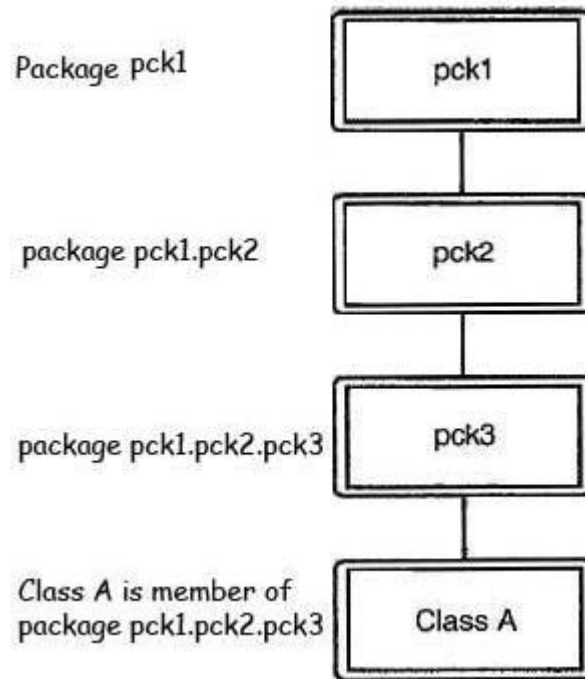
Here, *pkg* is the name of the package. For example, the following statement creates a package called MyPackage.

package MyPackage;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

It is possible to create a hierarchy of packages. The general form of a multileveled package statement is shown here:

package pkg1[.pkg2[.pkg3]];



A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

needs to be stored in **java\awt\image** in a Windows environment. We cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

consider the following package specification:

package MyPack

In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to MyPack*. For example, in a Windows environment, if the path to **MyPack** is

C:\MyPrograms\Java\MyPack

then the class path to **MyPack** is

C:\MyPrograms\Java

Example:

// A simple package

package MyPack;

class Balance {

String name;

double bal;

Balance(String n, double b) {

name = n;


```

bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}

```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

java MyPack.AccountBalance

Remember, you will need to be in the directory above **MyPack** when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.)

As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

java AccountBalance

AccountBalance must be qualified with its package name.

Example:

```

package pck1;
class Student
{
    private int rollno;
    private String name;
    private String address;
    public Student(int rno, String sname, String sadd)
    {
        rollno = rno;
        name = sname;
        address = sadd;
    }
    public void showDetails()
    {
        System.out.println("Roll No :: " + rollno);
        System.out.println("Name :: " + name);
        System.out.println("Address :: " + address);
    }
}

```

```

    public class DemoPackage
    {
        public static void main(String ar[])
        {
            Student st[]=new Student[2];
            st[0] = new Student (1001,"Alice", "New York");
            st[1] = new Student(1002,"BOB","Washington");
            st[0].showDetails();
            st[1].showDetails();
        }
    }

```

There are two ways to create package directory as follows:

1. Create the folder or directory at your choice location with the same name as package name. After compilation of copy *.class* (byte code file) file into this folder.
2. Compile the file with following syntax.
`javac -d <target location of package> sourceFile.java`

The above syntax will create the package given in the *sourceFile* at the *<target location of package>* if it is not yet created. If package already exist then only the *.class* (byte code file) will be stored to the package given in *sourceFile*.

Steps to compile the given example code:

Compile the code with the command on the command prompt.

```
javac -d DemoPackage.java
```

1. The command will create the package at the current location with the name pck1, and contains the file DemoPackage.class and Student.class
2. To run write the command given below
`java pck1.DemoPackage`

Note: The DemoPackate.class is now stored in pck1 package. So that we've to use *fully qualified type name* to run or access it.

Output:

```

Roll No :: 1001
Name :: Alice
Address :: New York
Roll No :: 1002
Name :: Bob
Address :: Washington

```

UNIT II

INHERITANCE AND INTERFACES

Inheritance – Super classes- sub classes –Protected members – constructors in sub classes- the Object class – abstract classes and methods- final methods and classes – Interfaces – defining an interface, implementing interface, differences between classes and interfaces and extending interfaces - Object cloning -inner classes, Array Lists – Strings

INHERITANCE

Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behaviors of one class to another, i.e., acquiring the properties and behavior of child class from the parent class.

When one object acquires all the properties and behaviours of another object, it is known as inheritance. Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

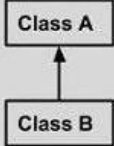
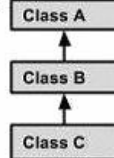
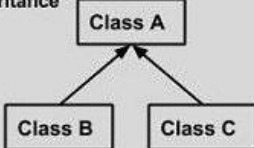
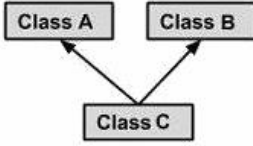
Uses of inheritance in java

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

Types of inheritance in java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.

Syntax:

```
class subClass extends superClass
{
    //methods and fields
}
```

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {.....} </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {.....} </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

Terms used in Inheritance

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

SINGLE INHERITANCE

In Single Inheritance one class extends another class (one class only).

Example:

```
public class ClassA
{
    public void dispA()
    {
        System.out.println("disp() method of ClassA");
    }
}
public class ClassB extends ClassA
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
    public static void main(String args[])
    {
        //Assigning ClassB object to ClassB reference
        ClassB b = new ClassB();
        //call dispA() method of ClassA
        b.dispA();
        //call dispB() method of ClassB
        b.dispB();
    }
}
```

Output :

```
disp() method of ClassA
disp() method of ClassB
```

MULTILEVEL INHERITANCE

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

Example:

```
public class ClassA
{
    public void dispA()
    {
        System.out.println("disp() method of ClassA");
    }
}
public class ClassB extends ClassA
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
}
public class ClassC extends ClassB
{
    public void dispC()
    {
        System.out.println("disp() method of ClassC");
    }
    public static void main(String args[])
    {
        //Assigning ClassC object to ClassC reference
        ClassC c = new ClassC();
        //call dispA() method of ClassA
        c.dispA();
        //call dispB() method of ClassB
        c.dispB();
        //call dispC() method of ClassC
        c.dispC();
    }
}
```

Output :

```
disp() method of ClassA
disp() method of ClassB
disp() method of ClassC
```

HIERARCHICAL INHERITANCE

In Hierarchical Inheritance, one class is inherited by many sub classes.

Example:

```
public class ClassA
{
    public void dispA()
    {
        System.out.println("disp() method of ClassA");
    }
}

public class ClassB extends ClassA
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
}

public class ClassC extends ClassA
{
    public void dispC()
    {
        System.out.println("disp() method of ClassC");
    }
}

public class ClassD extends ClassA
{
    public void dispD()
    {
        System.out.println("disp() method of ClassD");
    }
}

public class HierarchicalInheritanceTest
{
    public static void main(String args[])
    {
        //Assigning ClassB object to ClassB reference
        ClassB b = new ClassB();
        //call dispB() method of ClassB
        b.dispB();
        //call dispA() method of ClassA
        b.dispA();
        //Assigning ClassC object to ClassC reference
        ClassC c = new ClassC();
        //call dispC() method of ClassC
        c.dispC();
    }
}
```

```

//call dispA() method of ClassA
c.dispA();

//Assigning ClassD object to ClassD reference
ClassD d = new ClassD();
//call dispD() method of ClassD
d.dispD();
//call dispA() method of ClassA
d.dispA();
}
}

```

Output :

```

disp() method of ClassB
disp() method of ClassA
disp() method of ClassC
disp() method of ClassA
disp() method of ClassD
disp() method of ClassA

```

Hybrid Inheritance is the combination of both Single and Multiple Inheritance. Again Hybrid inheritance is also not directly supported in Java only through interface we can achieve this. Flow diagram of the Hybrid inheritance will look like below. As you can ClassA will be acting as the Parent class for ClassB & ClassC and ClassB & ClassC will be acting as Parent for ClassD.

Multiple Inheritance is nothing but one class extending more than one class. Multiple Inheritance is basically not supported by many Object Oriented Programming languages such as Java, Small Talk, C# etc.. (C++ Supports Multiple Inheritance). As the Child class has to manage the dependency of more than one Parent class. But you can achieve multiple inheritance in Java using Interfaces.

“super” KEYWORD***Usage of super keyword***

1. super() invokes the constructor of the parent class.
2. super.variable_name refers to the variable in the parent class.
3. super.method_name refers to the method of the parent class.

1. super() invokes the constructor of the parent class

super() will invoke the constructor of the parent class. Even when you don't add **super()** keyword the compiler will add one and will invoke the [Parent Class constructor](#).

Example:

```

class ParentClass
{
    ParentClass()

```

```
        {
            System.out.println("Parent Class default Constructor");
        }
    }
    public class SubClass extends ParentClass
    {
        SubClass()
        {
            System.out.println("Child Class default Constructor");
        }
        public static void main(String args[])
        {
            SubClass s = new SubClass();
        }
    }
```

Output:

Parent Class default Constructor
Child Class default Constructor

Even when we add explicitly also it behaves the same way as it did before.

```
class ParentClass
{
    public ParentClass()
    {
        System.out.println("Parent Class default Constructor");
    }
}
public class SubClass extends ParentClass
{
    SubClass()
    {
        super();
        System.out.println("Child Class default Constructor");
    }
    public static void main(String args[])
    {
        SubClass s = new SubClass();
    }
}
```

Output:

Parent Class default Constructor
Child Class default Constructor

You can also call the parameterized constructor of the Parent Class. For example, **super(10)** will call parameterized constructor of the Parent class.

```
class ParentClass
{
    ParentClass()
    {
        System.out.println("Parent Class default Constructor called");
    }
    ParentClass(int val)
    {
        System.out.println("Parent Class parameterized Constructor, value: "+val);
    }
}
public class SubClass extends ParentClass
{
    SubClass()
    {
        super();//Has to be the first statement in the constructor
        System.out.println("Child Class default Constructor called");
    }
    SubClass(int val)
    {
        super(10);
        System.out.println("Child Class parameterized Constructor, value: "+val);
    }
    public static void main(String args[])
    {
        //Calling default constructor
        SubClass s = new SubClass();
        //Calling parameterized constructor
        SubClass s1 = new SubClass(10);
    }
}
```

Output

```
Parent Class default Constructor called
Child Class default Constructor called
Parent Class parameterized Constructor, value: 10
Child Class parameterized Constructor, value: 10
```

2. super.variable name refers to the variable in the parent class

When we have the same variable in both parent and subclass

```
class ParentClass
{
```

```
        int val=999;
    }
    public class SubClass extends ParentClass
    {
        int val=123;

        void disp()
        {
            System.out.println("Value is : "+val);
        }

        public static void main(String args[])
        {
            SubClass s = new SubClass();
            s.disp();
        }
    }
```

Output

Value is : 123

This will call only the **val** of the sub class only. Without **super** keyword, you cannot call the **val** which is present in the Parent Class.

class ParentClass

```
{
    int val=999;
}
public class SubClass extends ParentClass
{
    int val=123;

    void disp()
    {
        System.out.println("Value is : "+super.val);
    }

    public static void main(String args[])
    {
        SubClass s = new SubClass();
        s.disp();
    }
}
```

Output

Value is : 999

3. *super.method* nae refers to the method of the parent class

When you override the Parent Class method in the Child Class without super keywords support you will not be able to call the Parent Class method. Let's look into the below example

```
class ParentClass
{
    void disp()
    {
        System.out.println("Parent Class method");
    }
}
public class SubClass extends ParentClass
{
    void disp()
    {
        System.out.println("Child Class method");
    }
    void show()
    {
        disp();
    }
    public static void main(String args[])
    {
        SubClass s = new SubClass();
        s.show();
    }
}
```

Output:

Child Class method

Here we have overridden the **Parent Class disp()** method in the SubClass and hence **SubClass disp()** method is called. If we want to call the **Parent Class disp()** method also means then we have to use the super keyword for it.

```
class ParentClass
{
    void disp()
    {
        System.out.println("Parent Class method");
    }
}
public class SubClass extends ParentClass
{

```

```
void disp()
{
    System.out.println("Child Class method");
}

void show()
{
    //Calling SubClass disp() method
    disp();
    //Calling ParentClass disp()
    method super.disp();
}
public static void main(String args[])
{
    SubClass s = new SubClass();
    s.show();
}
}
```

Output

Child Class method
Parent Class method

When there is no method overriding then by default **Parent Class disp()** method will be called.

```
class ParentClass
{
    public void disp()
    {
        System.out.println("Parent Class method");
    }
}
public class SubClass extends ParentClass
{
    public void show()
    {
        disp();
    }
    public static void main(String args[])
    {
        SubClass s = new SubClass();
        s.show(); }}
}
```

Output:

Parent Class method

The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array. **Object** defines the following methods, which means that they are available in every object.

Method	Purpose
<code>Object clone()</code>	Creates a new object that is the same as the object being cloned.
<code>boolean equals(Object <i>object</i>)</code>	Determines whether one object is equal to another.
<code>void finalize()</code>	Called before an unused object is recycled.
<code>Class<?> getClass()</code>	Obtains the class of an object at run time.
<code>int hashCode()</code>	Returns the hash code associated with the invoking object.
<code>void notify()</code>	Resumes execution of a thread waiting on the invoking object.
<code>void notifyAll()</code>	Resumes execution of all threads waiting on the invoking object.
<code>String toString()</code>	Returns a string that describes the object.
<code>void wait()</code> <code>void wait(long <i>milliseconds</i>)</code> <code>void wait(long <i>milliseconds</i>, int <i>nanoseconds</i>)</code>	Waits on another thread of execution.

The methods `getClass()`, `notify()`, `notifyAll()`, and `wait()` are declared as **final**. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: `equals()` and `toString()`. The `equals()` method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The `toString()` method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using `println()`. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

ABSTRACT CLASS

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). Abstraction is a process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it. It needs to be extended and its method implemented. It cannot be instantiated.

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.

abstract void printStatus();//no body and abstract

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes.

If you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

Example1:

File: TestAbstraction1.java

```
abstract class Shape{
    abstract void draw();
}
```

//In real scenario, implementation is provided by others i.e. unknown by end

```
user class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
```

```
class Circle1 extends Shape{
    void draw(){System.out.println("drawing circle");}
}
```

//In real scenario, method is called by programmer or user

```
class TestAbstraction1{
    public static void main(String args[]){
        Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method
        s.draw();
    }
}
```

Output:

drawing circle

Abstract class having constructor, data member, methods

An abstract class can have data member, abstract method, method body, constructor and even main() method.

Example2:

File: TestAbstraction2.java

//example of abstract class that have method body

```
abstract class Bike{
    Bike(){System.out.println("bike is created");}
    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}
class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}
```

```
class TestAbstraction2{
public static void main(String args[]){
    Bike obj = new Honda();
    obj.run();
    obj.changeGear();
}
}
```

Output:

```
bike is created
    running safely..
    gear changed
```

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Example3:

```
interface A{
void a();
void b();
void c();
void d();
}
abstract class B implements A{
public void c(){System.out.println("I am c");}
}
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Output:

```
I am a
I am b
I am c
I am d
```

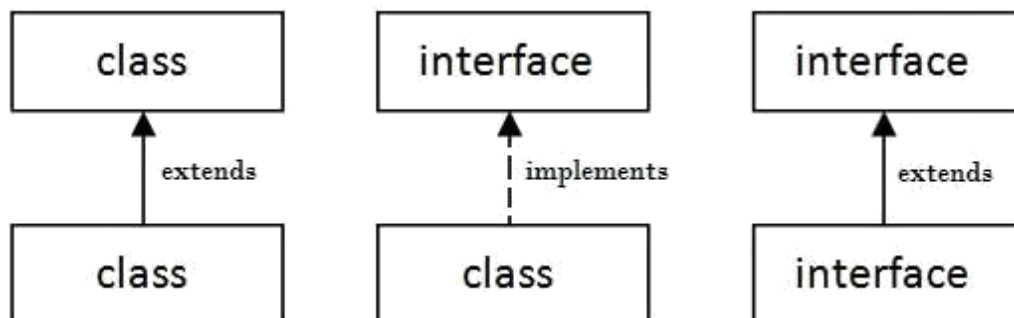
INTERFACE IN JAVA

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 and multiple inheritance.

Interface is declared by using interface keyword. It provides total abstraction; means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface.

Syntax:

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

Relationship between classes and interfaces**Example:** interface

```
printable{ void
print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

Output:

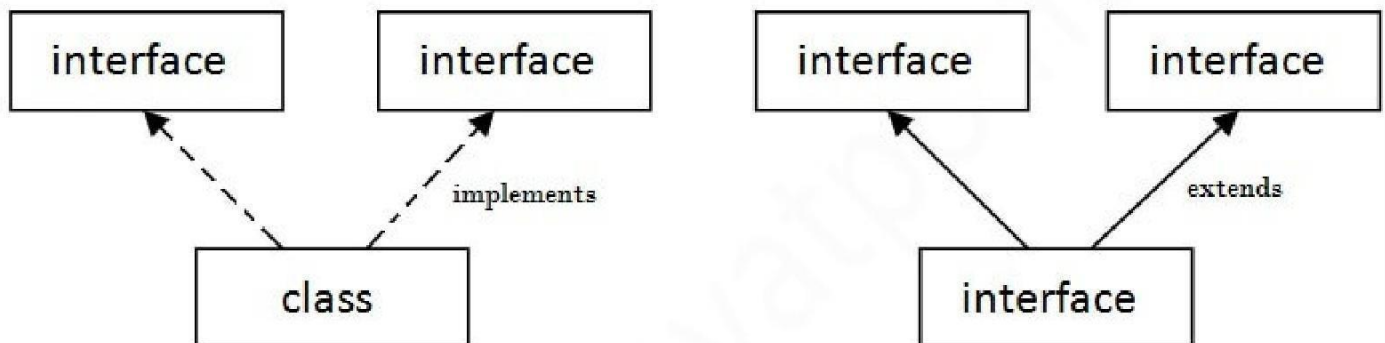
Hello


```
Example: interface
Drawable
{
void draw();
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}
}
```

Output:
drawing circle

Multiple inheritance in Java by interface

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



```
Example: interface
Printable{ void
print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
```

```
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}
```

Output:

Hello
Welcome
Interface inheritance

A class implements interface but one interface extends another interface .

Example:

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){ TestInterface4
obj = new TestInterface4(); obj.print();

obj.show();
}}
```

Output:

Hello
Welcome

Nested Interface in Java

An interface can have another interface i.e. known as nested interface.

```
interface printable{
    void print();
    interface MessagePrintable{
        void msg();
    }
}
```

Key points to remember about interfaces:

- 1) We can't instantiate an interface in java. That means we cannot create the object of an interface
- 2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete (methods with body) methods both.
- 3) "implements" keyword is used by classes to implement an interface.
- 4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.
- 5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
- 6) Interface cannot be declared as private, protected or transient.
- 7) All the interface methods are by default abstract and public.
- 8) Variables declared in interface are public, static and final by default.

interface Try

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

All of the above statements are identical.

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

interface Try

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

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

- 10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface "Try" which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

class Sample implements Try

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

- 11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A class can implement any number of interfaces.

13) If there are two or more same methods in two interfaces and a class implements both interfaces, implementation of the method once is enough.

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

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

```
interface A
{
    public void aaa();
}
interface B
{
    public int aaa();
}
class Central implements A,B
{
    public void aaa() // error
    {
    }
    public int aaa() // error
    {
    }
    public static void main(String args[])
    {
    }
}
```

```

    }
}
15) Variable names conflicts can be resolved by interface
name. interface A
{
    int x=10;
}
interface B
{
    int x=100;
}
class Hello implements A,B
{
    public static void Main(String args[])
    {
        System.out.println(x);
        System.out.println(A.x);
        System.out.println(B.x);
    }
}

```

Advantages of interface in java:

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

DIFFERENCE BETWEEN ABSTRACT CLASS AND INTERFACE

ABSTRACT CLASS	INTERFACE
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface .	Interface can't provide the implementation of abstract class .
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.

6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword extends.	An interface class can be implemented using keyword implements
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

FINAL KEYWORD

Final keyword can be used along with **variables**, **methods** and **classes**.

- 1) **final variable**
- 2) **final method**
- 3) **final class**

1. Java final variable

A **final variable** is a variable whose value **cannot** be changed at anytime once assigned, it remains as a constant forever.

Example:

```
public class Travel
{
    final int SPEED=60;
    void increaseSpeed(){
        SPEED=70;
    }
    public static void main(String args[])
    {
        Travel t=new Travel();
        t.increaseSpeed();
    }
}
```

Output :

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The final field Travel.SPEED cannot be assigned

The above code will give you Compile time error, as we are trying to change the value of a final variable 'SPEED'.

2. Java final method

When you declare a method as **final**, then it is called as **final method**. A **final method** cannot be **overridden**.

```
package com.javainterviewpoint;
```

```
class Parent
{
    public final void disp()
    {
        System.out.println("disp() method of parent class");
    }
}
public class Child extends Parent
{
    public void disp()
    {
        System.out.println("disp() method of child class");
    }
    public static void main(String args[])
    {
        Child c = new Child();
        c.disp();
    }
}
```

Output : We will get the below error as we are overriding the **disp()** method of the **Parent** class. Exception in thread "main" java.lang.VerifyError: class com.javainterviewpoint.Child overrides final method disp()

```
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(Unknown Source)
    at java.security.SecureClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.access$100(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
```

3. Java final class

A final class cannot be extended(cannot be subclassed), lets take a look into the below example package com.javainterviewpoint;

```
final class Parent
{
}
public class Child extends Parent
{
}
```

```
public static void main(String args[])
{
    Child c = new Child();
}
}
```

Output :

We will get the compile time error like ***“The type Child cannot subclass the final class Parent”***
Exception in thread "main" java.lang.Error: Unresolved compilation problem

OBJECT CLONING

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class.

Syntax of the clone() method:

protected Object clone() ***throws*** CloneNotSupportedException

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Advantage of Object cloning

- You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
- It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
- Clone() is the fastest way to copy array.

Disadvantage of Object cloning

- To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
- We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
- Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
- If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.

- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

Example of clone() method (Object cloning)

```
class Student implements Cloneable{
int rollno;
String name;
Student(int rollno,String
name){ this.rollno=rollno;
this.name=name;
}
public Object clone()throws CloneNotSupportedException{
return super.clone();
}
public static void main(String args[]){
try{
Student s1=new Student(101,"amit");
Student s2=(Student)s1.clone();
System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);
}
catch(CloneNotSupportedException c){}
}
}
```

Output:

```
101 amit
101 amit
```

INNER CLASSES

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

- 1) Nested Inner class
- 2) Method Local inner classes
- 3) Anonymous inner classes
- 4) Static nested classes

Nested Inner class

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier. Like class, interface can also be nested and can have access specifiers.

Example:

```
class Outer {
// Simple nested inner class
class Inner {
```

```
        public void show() {  
            System.out.println("In a nested class method");  
        }  
    }  
}  
class Main {  
    public static void main(String[] args) {  
        Outer.Inner in = new Outer().new Inner();  
        in.show();  
    }  
}
```

Output:

In a nested class method

Method Local inner classes

Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().

Example:

```
class Outer {  
    void outerMethod() {  
        System.out.println("inside outerMethod");  
        // Inner class is local to outerMethod()  
        class Inner {  
            void innerMethod() {  
                System.out.println("inside innerMethod");  
            }  
        }  
        Inner y = new Inner();  
        y.innerMethod();  
    }  
}  
class MethodDemo {  
    public static void main(String[] args) {  
        Outer x = new Outer();  
        x.outerMethod();  
    }  
}
```

Output:

Inside outerMethod
Inside innerMethod

Static nested classes

Static nested classes are not technically an inner class. They are like a static member of outer

class.

Example:

```
class Outer {  
    private static void outerMethod() {  
        System.out.println("inside outerMethod");  
    }  
    // A static inner class  
    static class Inner{  
        public static void main(String[] args) {  
            System.out.println("inside inner class Method");  
            outerMethod();  
        }  
    }  
}
```

Output:

inside inner class Method
inside outerMethod

Anonymous inner classes

Anonymous inner classes are declared without any name at all. They are created in two ways.

a) As subclass of specified type

```
class Demo {  
    void show() {  
        System.out.println("i am in show method of super class");  
    }  
}  
  
class Flavor1Demo {  
    // An anonymous class with Demo as base class  
    static Demo d = new Demo() {  
        void show() {  
            super.show();  
            System.out.println("i am in Flavor1Demo class");  
        }  
    };  
    public static void main(String[] args){  
        d.show();  
    }  
}
```

Output:

i am in show method of super class
i am in Flavor1Demo class

In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous

class show() method is overridden.

b) As implementer of the specified interface

Example:

```
class Flavor2Demo {  
    // An anonymous class that implements Hello interface  
    static Hello h = new Hello() {  
        public void show() {  
            System.out.println("i am in anonymous class");  
        }  
    };  
    public static void main(String[] args) {  
        h.show();  
    }  
}  
  
interface Hello {  
    void show(); }  

```

Output:

i am in anonymous class

In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

STRINGS IN JAVA

In java, string is basically an object that represents sequence of char values. **Java String** provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

```
String s="javatpoint";
```

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

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

```
String s="welcome";
```

2) By new keyword

```
String s=new String("Welcome");
```

String methods:		
1.	char charAt(int index)	returns char value for the particular index
2.	int length()	returns string length
3.	static String format(String format, Object... args)	returns formatted string
4.	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5.	String substring(int beginIndex)	returns substring for given begin index
6.	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
7.	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8.	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
9.	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10.	boolean equals(Object another)	checks the equality of string with object
11.	boolean isEmpty()	checks if string is empty
12.	String concat(String str)	concatinates specified string
13.	String replace(char old, char new)	replaces all occurrences of specified char value
14.	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15.	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16.	String[] split(String regex)	returns splitted string matching regex
17.	String[] split(String regex, int limit)	returns splitted string matching regex and limit
18.	String intern()	returns interned string
19.	int indexOf(int ch)	returns specified char value index
20.	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21.	int indexOf(String substring)	returns specified substring index
22.	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23.	String toLowerCase()	returns string in lowercase.
24.	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25.	String toUpperCase()	returns string in uppercase.
26.	String toUpperCase(Locale l)	returns string in uppercase using specified locale.
27.	String trim()	removes beginning and ending spaces of this string.
28.	static String valueOf(int value)	converts given type into string. It is overloaded.

Example:

```
public class stringmethod
{
    public static void main(String[] args)
    {
        String string1 = new String("hello");
        String string2 = new String("hello");
        if (string1 == string2)
        {
            System.out.println("string1= "+string1+" string2= "+string2+" are equal");
        }
        else
        {
            System.out.println("string1= "+string1+" string2= "+string2+" are Unequal");
        }
        System.out.println("string1 and string2 is=
        "+string1.equals(string2)); String a="information";
        System.out.println("Uppercase of String a is= "+a.toUpperCase());
        String b="technology";
        System.out.println("Concatenation of object a and b is= "+a.concat(b));
        System.out.println("After concatenation Object a is= "+a.toString());
        System.out.println("\"Joseph's\" is the greatest\\ college in chennai");
        System.out.println("Length of Object a is= "+a.length());
        System.out.println("The third character of Object a is= "+a.charAt(2));
        StringBuffer n=new StringBuffer("Technology");
        StringBuffer m=new StringBuffer("Information");
        System.out.println("Reverse of Object n is= "+n.reverse());
        n= new StringBuffer("Technology");
        System.out.println("Concatenation of Object m and n is= "+m.append(n));
        System.out.println("After concatenation of Object m is= "+m);
    }
}
```

Output:

```
string1= hello string2= hello are Unequal
string1 and string2 is= true
Uppercase of String a is= INFORMATION
Concatenation of object a and b is= informationtechnology
After concatenation Object a is= information
"Joseph's" is the greatest\ college in chennai
Length of Object a is= 11
The third character of Object a is= f
Reverse of Object n is= ygonlhceT
Concatenation of Object m and n is= InformationTechnology
```

After concatenation of Object m is= InformationTechnology

Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

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 syntax:

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

Constructors of Java ArrayList

CONSTRUCTOR	DESCRIPTION
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

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.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
Object[] toArray(Object[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
Object clone()	It is used to return a shallow copy of an ArrayList.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

Java ArrayList Example: Book

Example:

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}

public class ArrayListExample {
public static void main(String[] args) {
    //Creating list of Books
    List<Book> list=new ArrayList<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to list
    list.add(b1);
    list.add(b2);
    list.add(b3);
    //Traversing list
    for(Book b:list){
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```


Unit – III

Exception Handling and I/O

Exceptions-exception hierarchy-throwing and catching exceptions-built-in exceptions, creating own exceptions, Stack Trace Elements. Input /Output Basics-Streams-Byte streams and character streams-Reading and Writing Console-Reading and Writing Files Templates

Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

Exceptions are conditions within the code. A developer can handle such conditions and take necessary corrective actions. Few examples

- DivideByZero exception
 - NullPointerException
 - ArithmeticException
 - ArrayIndexOutOfBoundsException
-
- An exception (or exceptional event) is a problem that arises during the execution of a program.
 - When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
 - If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

Ex: Exception in thread "main"

**java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)**

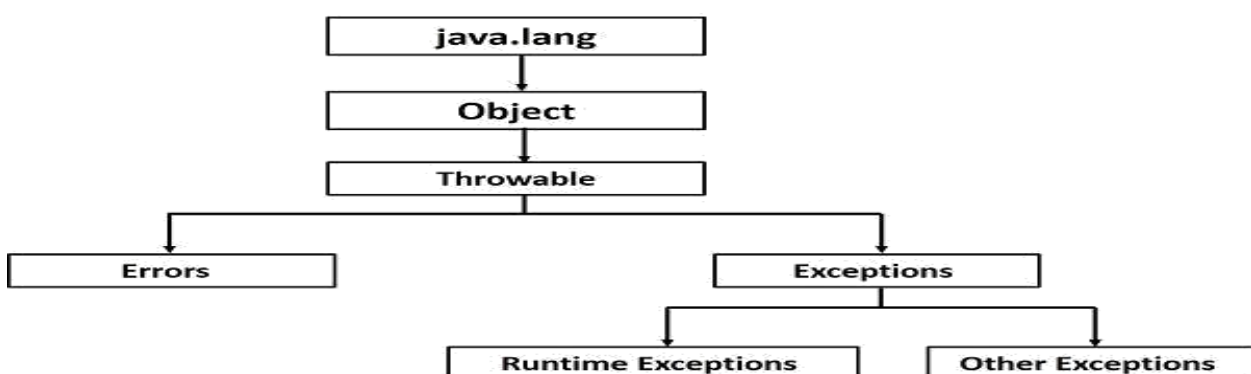
Where, ExceptionDemo : The class name
 main : The method name
 ExceptionDemo.java : The filename
 java:5 : Line number

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an **invalid data**.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class.



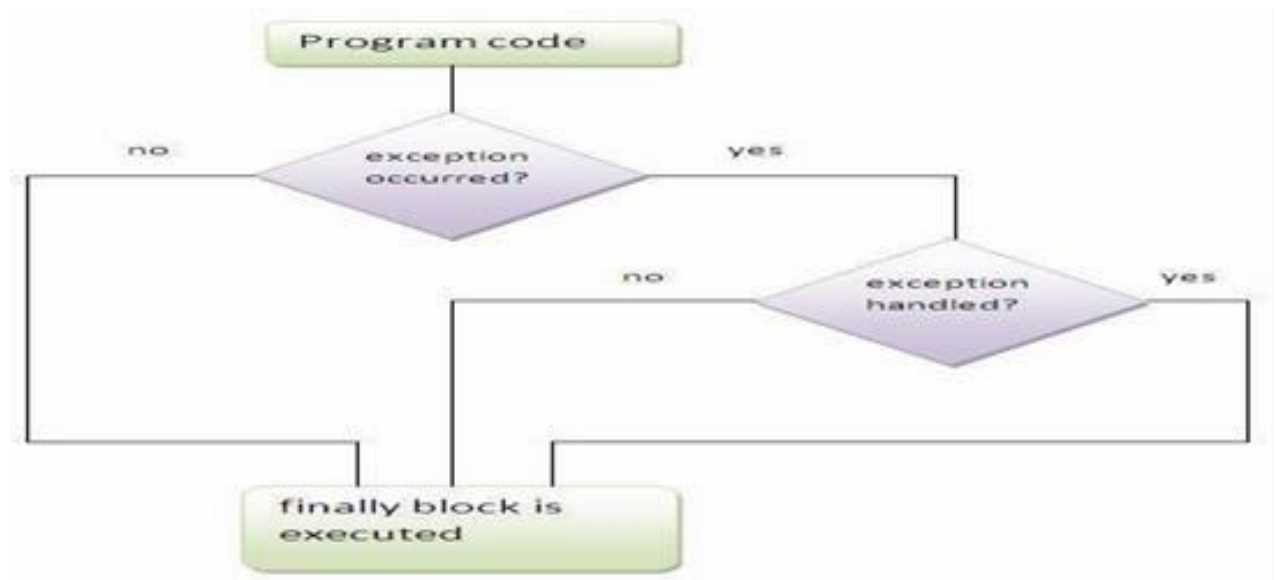
Key words used in Exception handling

There are 5 keywords used in java exception handling.

1. try	A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code.
2. catch	A catch statement involves declaring the type of exception we are trying to catch.
3. finally	A finally block of code always executes, irrespective of occurrence of an Exception.
4. throw	It is used to execute important code such as closing connection, stream etc. throw is used to invoke an exception explicitly.
5. throws	throws is used to postpone the handling of a checked exception.

Syntax : try { //Protected code } catch (ExceptionType1 e1) { //Catch block } catch (ExceptionType2 e2) { //Catch block }	//Example-predefined Exception - for //ArrayIndexOutOfBoundsException public class ExcepTest { public static void main(String args[]) { int a[] = new int[2]; try { System.out.println("Access element three :" + a[3]); } catch (ArrayIndexOutOfBoundsException e) { System.out.println("Exception thrown :" + e); } finally { a[0] = 6; } } }
---	---

<pre> catch(ExceptionType3 e3) { //Catch block } finally { //The finally block always executes. } </pre>	<pre> System.out.println("First element value: " + a[0]); System.out.println("The finally statement is executed"); } } </pre> <p>Output</p> <p><i>Exception thrown</i></p> <p><i>:java.lang.ArrayIndexOutOfBoundsException:3</i></p> <p><i>First element value: 6</i></p> <p><i>The finally statement is executed</i></p> <p>Note : <i>here array size is 2 but we are trying to access 3rd element.</i></p>
--	--



Uncaught Exceptions

This small program includes an expression that intentionally causes a divide-by-zero error: `class Exc0 { public static void main(String args[]) { int d = 0; int a =`

42 / d; } } When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately

Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)
```

Stack Trace:

Stack Trace is a list of method calls from the point when the application was started to the point where the exception was thrown. The most recent method calls are at the top. A stacktrace is a very helpful debugging tool. It is a list of the method calls that the application was in the middle of when an Exception was thrown. This is very useful because it doesn't only show you where the error happened, but also how the program ended up in that place of the code.

Using try and Catch

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. A try and its catch statement form a unit. The the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error:

```

class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}

```

This program generates the following output:

Division by zero.

After catch statement.

The call to `println()` inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
```

```

class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}

```

Here is the output generated by running it both ways:

C:\>java MultiCatch

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

C:\>java MultiCatch TestArg

a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException:42

After try/catch blocks.

Nested try Statements

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```
// An example of nested try statements.
class NestTry {
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
```



```

c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
}

```

C:\>java NestTry

Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One

a = 1

Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two

a = 2

Array index out-of-bounds:

java.lang.ArrayIndexOutOfBoundsException:42

throw

it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Primitive types, such as int or char, as well as non-Throwable classes, such as String and

Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace

// Demonstrate throw.

```
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

Here is the resulting output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

Throws

If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type `Error` or `RuntimeException`, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
```

```
System.out.println("Caught " + e);  
}  
}  
}
```

Here is the output generated by running this example program:

inside throwOne

caught java.lang.IllegalAccessException: demo

finally

The finally keyword is designed to address this contingency. finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional.

```
// Demonstrate finally.  
class FinallyDemo {  
    // Through an exception out of the method.  
    static void procA() {  
        try {  
            System.out.println("inside procA");  
            throw new RuntimeException("demo");  
        } finally {
```

```
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
```

```
}
```

Here is the output generated by the preceding program:

```
inside procA
```

```
procA's finally
```

```
Exception caught
```

```
inside procB
```

```
procB's finally
```

```
inside procC
```

```
procC's finally
```

Categories of Exceptions

Checked exceptions – A checked exception is an exception that occurs at the compiletime, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

Unchecked exceptions – An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Common scenarios where exceptions may occur:

There are given some scenarios where unchecked exceptions can occur.

They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an `NullPointerException`.

```
String s=null;
```

```
System.out.println(s.length());//NullPointerException
```

3) Scenario where `ArrayIndexOutOfBoundsException` occurs

If you are inserting any value in the wrong index, it would result `ArrayIndexOutOfBoundsException` as shown below:

```
int a[]=new int[5];
```

```
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java's Built-in Exceptions

User-defined Exceptions

All exceptions must be a child of **Throwable**.

If we want to write a checked exception that is automatically enforced by the compiler, we need to extend the `Exception` class.

User defined exception needs to inherit (extends) `Exception` class in order to act as an exception.

`throw` keyword is used to throw such exceptions.

```
class MyOwnException extends Exception
```

```
{  
    public
```

```
    MyOwnException(String
```

```
    msg) { super(msg);
```

```
    }
```

```
}
```

```
class EmployeeTest
```

```
{
```

```

static void employeeAge(int age) throws
MyOwnException {
    if(age < 0)
        throw new MyOwnException("Age can't be less
than zero"); else
        System.out.println("Input is valid!!");
    }
public static void main(String[] args)
{
    try { employeeAge(-2);
        }
    catch (MyOwnException e)
    {
        e.printStackTrace();
    }
}
}

```

Advantages of Exception Handling

Exception handling allows us to control the normal flow of the program by using exception handling in program.

It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.

It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

MULTITHREADING

Thread:

A thread is a single sequential (separate) flow of control within program.

Sometimes, it is called an execution context or light weight process.

Multithreading

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Multitasking

Executing several tasks simultaneously is called multi-tasking.

There are 2 types of multi-tasking

1. Process-based multitasking
2. Thread-based multi-tasking

1. Process-based multi-tasking

Executing various jobs together where each job is a separate independent operation is called process-based multi-tasking.

2. Thread-based multi-tasking

Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread-based multitasking and each independent part is called Thread. It is best suitable for the programmatic

level. The main goal of multi-tasking is to make or do a better performance of the system by reducing response time

Multithreading vs Multitasking	
Multithreading is to execute multiple threads in a process concurrently.	Multitasking is to run multiple processes on a computer concurrently.
Execution	
In Multithreading, the CPU switches between multiple threads in the same process.	In Multitasking, the CPU switches between multiple processes to complete the execution.
Resource Sharing	
In Multithreading, resources are shared among multiple threads in a process.	In Multitasking, resources are shared among multiple processes.
Complexity	
Multithreading is light-weight and easy to create.	Multitasking is heavy-weight and harder to create.

Life Cycle of Thread

A thread can be in any of the five following states

1.Newborn State:

When a thread object is created a new thread is born and said to be in Newborn state.

2.Runnable State:

If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion

3. Running State:

It means that the processor has given its time to the thread for execution.

A thread keeps running until the following conditions occurs

(a) Thread give up its control on its own and it can happen in the following situations

i. A thread gets suspended using `suspend()` method which can only be revived with `resume()` method

ii. A thread is made to sleep for a specified period of time using `sleep(time)` method, where time in milliseconds

iii. A thread is made to wait for some event to occur using `wait()` method. In this case a thread can be scheduled to run again using `notify()` method.

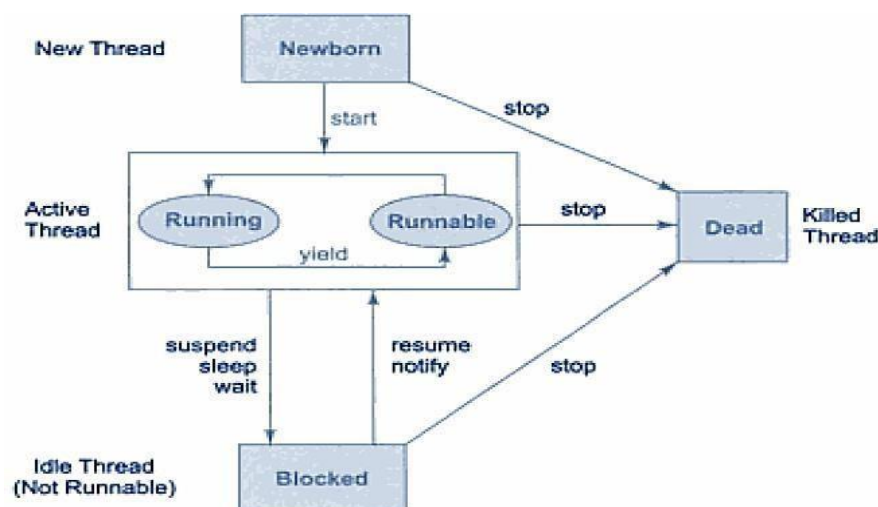
(b) A thread is pre-empted by a higher priority thread

4. Blocked State:

If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.

5. Dead State:

A runnable thread enters the Dead or terminated state when it completes its task.



The Main Thread

When we run any java program, the program begins to execute its code starting from the main method. Therefore, the JVM creates a thread to start

executing the code present in main method. This thread is called as main thread. Although the main thread is automatically created, you can control it by obtaining a reference to it by calling `currentThread()` method.

Two important things to know about main thread are,

- It is the thread from which other threads will be produced.
- main thread must be always the last thread to finish execution.

```
class MainThread
{
    public static void main(String[] args)
    {
        Thread t1=Thread.currentThread();
        t.setName("MainThread");
        System.out.println("Name of thread is "+t1);
    }
}
```

Output:

Name of thread is Thread[MainThread,5,main]

Creation Of Thread

Thread Can Be Implemented In Two Ways

- 1) **Implementing Runnable Interface**
- 2) **Extending Thread Class**

1.Create Thread by Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called `run()`

Example:

```
public class ThreadSample implements Runnable
{
    public void run()
```

```
{  
    try  
    {  
        for (int i = 5; i > 0; i--)  
        {
```

```

        System.out.println("Child Thread" + i);
        Thread.sleep(1000);
    }
}
catch (InterruptedException e)
{
    System.out.println("Child interrupted");
}
System.out.println("Exiting Child Thread");
}
}

public class MainThread
{
    public static void main(String[] arg)
    {
        ThreadSample d = new ThreadSample();
        Thread s = new Thread(d);
        s.start();
        try
        {
            for (int i = 5; i > 0; i--)
            {
                System.out.println("Main Thread" + i);
                Thread.sleep(5000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main interrupted");
        }
        System.out.println("Exiting Main Thread");
    }
}

```

```
}}
```

2. Extending Thread Class

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run() method, which is the entry point for the new thread. It must also call start() to begin execution of the new thread.

Example:

```
public class ThreadSample extends Thread
{
    public void run()
    {
        try
        {
            for (int i = 5; i > 0; i--)
            {
```

```

        System.out.println("Child Thread" + i);
        Thread.sleep(1000);
    }
}
catch (InterruptedException e)
{
    System.out.println("Child interrupted");
}
System.out.println("Exiting Child Thread");
}
}

public class MainThread
{
    public static void main(String[] arg)
    {
        ThreadSample d = new ThreadSample();
        d.start();
        try
        {
            for (int i = 5; i > 0; i--)
            {
                System.out.println("Main Thread" + i);
                Thread.sleep(5000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main interrupted");
        }
        System.out.println("Exiting Main Thread");
    }
}

```


Synchronizing Threads

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource

General Syntax :

```
synchronized(object)
{
    //statement to be synchronized
}
```

Why use Synchronization

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

- Process Synchronization
- Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive

- Synchronized method.
- Synchronized block.
- static synchronization.

2.Cooperation (Inter-thread communication in java)

Synchronized method

- If you declare any method as synchronized, it is known as synchronized method.

- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example of synchronized method

```
package Thread; public
class SynThread
{
    public static void main(String args[])
    {
        share s = new share();
        MyThread m1 = new MyThread(s, "Thread1");
        MyThread m2 = new MyThread(s, "Thread2");
        MyThread m3 = new MyThread(s, "Thread3");
    }
}
class MyThread extends Thread
{
    share s;
    MyThread(share s, String str)
    {
        super(str);
        this.s = s;
        start();
    }
    public void run()
    {
        s.doword(Thread.currentThread().getName());
    }
}
class share
{
```

```
class Table{
    void printTable(int n){
        synchronized(this){//synchronized block
            for(int i=1;i<=5;i++){
                System.out.println(n*i);
                try{
                    Thread.sleep(400);
                }catch(Exception e){System.out.println(e);}
            }
        }
    }
}

//end of the method
```

```

    }
    class MyThread1 extends Thread{
    Table t;
    MyThread1(Table t){
    this.t=t;
    }
    public void run(){
    t.printTable(5);
    }
    }
    class MyThread2 extends Thread{
    Table t;
    MyThread2(Table t){
    this.t=t;
    }
    public void run(){
    t.printTable(100);
    }
    }
    public class TestSynchronizedBlock1{
    public static void main(String args[]){
    Table obj = new Table();//only one object
    MyThread1 t1=new MyThread1(obj);
    MyThread2 t2=new MyThread2(obj);
    t1.start();
    t2.start();
    }
    }

```

Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{
    synchronized static void printTable(int n){
        for(int i=1;i<=10;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){}
        }
    }
}

class MyThread1 extends Thread{
    public void run(){
        Table.printTable(1);
    }
}

class MyThread2 extends Thread{
    public void run(){
        Table.printTable(10);
    }
}

class MyThread3 extends Thread{
    public void run(){
        Table.printTable(100);
    }
}

class MyThread4 extends Thread{
    public void run(){
        Table.printTable(1000);
    }
}
```

```
}  
}  
public class TestSynchronization4{  
public static void main(String t[]){  
MyThread1 t1=new MyThread1();  
MyThread2 t2=new MyThread2();  
MyThread3 t3=new MyThread3();  
MyThread4 t4=new MyThread4();  
t1.start();  
t2.start();  
t3.start();  
t4.start();  
}  
}
```

Daemon Thread in Java

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

