UNIT-II

JAVA

Introduction to object-oriented programming-Features of Java – Data types, variables and arrays – Operators – Control statements – Classes and Methods – Inheritance. Packages and Interfaces – Exception Handling – Multithreaded Programming – Input/Output – Files – Utility Classes – String Handling.

OverView of java

In 1990, SunMicrosystems has conceived a project to develop a software for consumer electronic devices that could be controlled by remote. This project was called as Green project .

In January 1991, Bill Joy, James Goslings, Patrick Naughton, Mike sheradin met at Aspen to discuss this project. Mike sheradin was to focus on business Development. Ptarick Naughton was to begin work on graphics system and James Gosling was to identify the proper programming language for the project. Gosling though C and C++ could be used to develop the project. But the problem is faced with them is that they were system dependent languages which could not used an various processors which the electronic devices might use. So he started developing a new language which was completely system independent. This language was initially called *Oak*. Since this name was registered by some other company later it was changed to java.

Why the name java? James Gosling and his team members were consuming a lot of coffee while developing this language. They felt that they were able to develop a better language because of good quality coffee they had consumed. So, the coffee also had its own role in developing this language and hence, they fixed the name for the language as java. Thus the symbol for java is *coffee cup and saucer.*

Features of Object Oriented Programming System

Some of the features of OOPS are

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

Object

Entire OOP methodology has been derived from a single root concept called object. *An object is anything that really exist in the world and can be distinguished from others*. This definition specifies

everything is an object for eg: a table, a ball, a car a dog, a person etc. everything will come under objects. Every object has properties and a can perform certain actions.

For example: let us take a example of dog. It got properties like name, height, color, age etc. these properties are represented by variable. Now the object dog have some actions like running, sleeping, eating, barking etc. these are represented by various methods(functions) in our programming. So, we can conclude that objects contains variable and methods.

Class

A group of objects exhibiting same properties and actions will come under the same group called a class. A class represents a group name given to several objects.

For example: lets us take an example "*flower is a class"*. But if we take Rose, Lily, Jasmine, there are all objects of flower class. The class flower does not exist physically but its object like Rose, Lily, Jasmine exist physically.

Hence we could define a *class as a model or blueprint for creating the objects.*

Abstraction: There may be a lot of data, a class contains and the user does not need the entire data. The user requires only some part of the available data. In this case we can hide the unnecessary data from the user and expose only that data that is of interest to the user. This is called abstraction.

For example A good example for abstraction is a car, any car will have some parts like engine, radiator, mechanical and electrical equipment etc. the use of the car (driver) should know how to drive the car and does not require any knowledge of these parts. For eg: driver is never bothered about how the engine is designed and the internal parts of the engine.

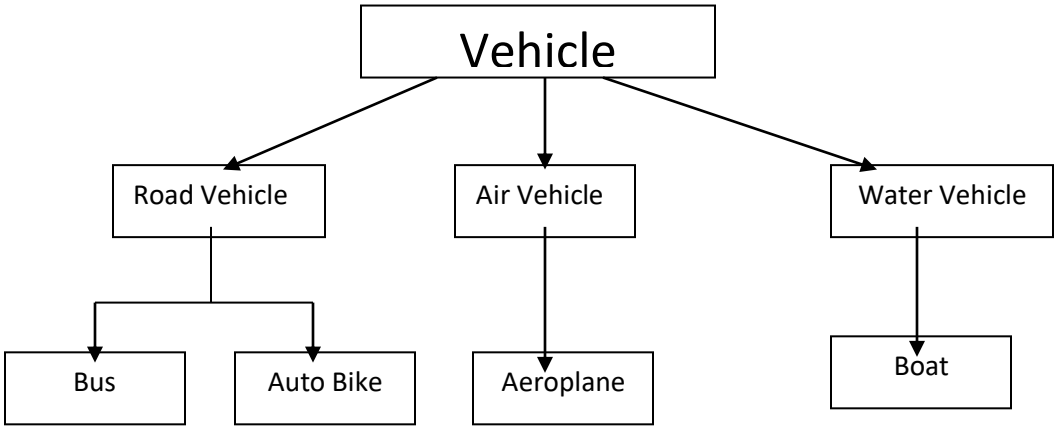
The advantage of abstraction is that every user will get his own view of the data according to his requirements and will not get confused with unnecessary data.

Encapsulation: Encapsulation is a mechanism where the data (variables) and code (methods) that act on the data will bind together. For eg: if we take a class, we write the variables and methods inside the class. Thus, class is binding them together so, class is an example of encapsulation.

Inheritance : The process of creating a new class from existing class is called inheritance. There exist a parent-child relationship among the classes. When a class inherits another class, it has all the properties of parent class and it adds some new properties of its own. Inheritance aids to **reusability**

For eg:

1. A good example for inheritance in nature is parents producing the children and children inheriting the qualities of the parents. This concept is important because it supports the concept of hierarchical classification.
2. We categorize vehicles into different subclass as shown below



Polymorphism: the word polymorphism came from two greek words ***"poly"*** means many and **"morphos"** meaning forms. Thus, polymorphism represents the ability to assume several different forms. Polymorphism provides flexibility in writing programs in such a way that the programmer uses same method call to perform different operations depending on the requirement.

Java Buzz words

Sun Microsystems described java language as java is a simple, Object-oriented, interpreted, Robust, Secure, architecture Neutral, Portable, high –performance, Multithreaded, Dynamic, Distributed language. These are known as Buzz words

1. **Java is Simple :** Although mush of the syntax of java is based on the earlier object-oriented language C++, the java is considerably simpler than C++.
   Java made simple by eliminating some of the complicated concepts from C/C++. Some of them are
   i. Many of the keywords have been eliminated.
   ii. There is no preprocessor.
   iii. Operator overloading, global variables, the goto statement, structures, pointer are eliminated.

2. **Java is Object-Oriented**: Java is a true object oriented language. Almost everything in java is object oriented. The only unit of programming is the class description. Unlike other languages, java has no functions, and no variables that can exist outside of class boundaries. Thus, all java programs must be built out of objects.
3. **Java is Robust**: Robust means strong. Java programs are strong and they don't crash easily like a C or C++ program. There are two reasons for this. Firstly, java has got excellent inbuilt Exception Handling features. An Exception is an error that occurs at run-time. If an exception occurs, the program terminates abruptly giving rise to problems like loss of data. Overcoming such problems is called exception handling. This, means that even though an exception occurs in a java program no harm will happen.
   Another reason, why java is robust lies in its memory management features. Most of the C and C++ programs crash in the middle because of not allocating sufficient memory or

forgetting the memory to be freed in a program. Such problems will not occur in java because the user need not allocate or deallocate the memory. In java everything will be taken care by JVM only. Suppose a variable or an object is created in memory and is not referenced for long time. Then after sometime it is automatically removed by Garbage Collector of JVM by using some algorithm.

4. **Java is Portable:** Java is portable means we can easily move java programs from one system to another system at anytime and from anywhere. Changes and upgrades in OS, Processors and system resources will not force any changes in java programs.
   Java ensures portability in **two ways:**
   First java compiler generates byte code instructions that can be implemented on any machine. Secondly the size of the primitive data types is machine independent.

5. **Java is Architecture Neural**: one of the main problems facing programmers that no guarantee exists that if you write a program today, it will run tomorrow. Even on the same machine. Operating system upgrades and changes in core system resources can all combine to make program malfunction.
   The java designers made several hard decisions in the java language and the JVM in an attempt to alter this situation. Their goal was **"Write once, run anywhere, anytime, forever**". To a great extent, this goal was accomplished.

6. **Java is secure**: java is secure in the following ways.
   i. By eliminating pointers, the java language removes the most common source of programming errors, such as overwriting memory locations that are being addressed by pointers with improperly set values.
   ii. The java language also insists that array index values are checked for validity before they are referenced and that all variables must be assigned a value before being used
   iii. Security problems like eavesdropping, tampering, impersonation and virus threats eliminated or minimized by using java on internet.

7. **Java is multithreaded**: java supports multithreaded programming which allows us to write a program that do many thing simultaneously means we need not wait for the application to complete one task before beginning another.
   for eg: we can check our mails and also play audio at the back. This feature gives interactive perform of graphical application.
   The java runtime comes with the tools the supports multiprocess synchronization and construct smoothly running interactive system.

8. **Java is interpreted**: Java programs are compiled to generate the Byte code. This code(bye code) can be executed on any system that implements the JVM(Java Virtual machine) or java interpreter. If we take any other language, only a interpreter or a compiler is used to execute the programs. But in java, we use both compiler and interpreter for the execution.
9. **High Performance**: the problem with interpreter inside the JVM is that it is slow. Because of this, java a programs used to run slow. To overcome this problem, along with the interpreter,

javasoft people have introduced JIT(Just In Time) complier, which enhance the speed of execution, so now in JVM both interpreter and JIT compiler work together to run the program.

10. **Java is Distributed**: Java is designed for the distributed environment of internet because it handles TCP,IP Protocols. Java also supports RMI(Remote Method Invocation) this feature enables a program to invoke methods across network.
11. **Java is Dynamic**: Before the development of java only static text used to be displayed in the browser. But when James Gosling demonstrated an animated atomic molecule where the rays are moving and stretching, the viewer were dumbstruck. This animation was done using an applet program, which are the dynamically interacting programs on internet.

Data Types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple* types. 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.

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers

Name	Width	Range
long	64	–9,223,372,036,854,775,808 **to** 9,223,372,036,854,775,807
int	32	–2,147,483,648 **to** 2,147,483,647
short	16	–32,768 **to** 32,767
byte	8	–128 **to** 127

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127.Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. Byte variables are declared by use of the **byte** keyword.

For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Here are some examples of **short** variable declarations:

short s; short t;

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays.

long

long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days.

Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.

double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values.

Characters

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits.

Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**.

boolean is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

Variable

Variable:- The name given to computer memory location is called variable. The purpose of variable is to store data.

Syntax:-

Type identifier[=value][,identifier [=value]…];

Rules to be followed in naming a variable

1. Variable name may consists of alphabets (A-Z , a-z) ,digits and special symbols $, _ are allowed.
2. The first character must not be digit.
3. Variables names are case sensitive.
4. Keywords should not be used as variable names.
5. Variable name can be of any length but, it is better to take length to be reasonable.

There are different types of variables. Some of them are

1. Instance variable.
2. Static variable.
3. Local variable.

Instance variable

1. If the value of the variable is varied from one object to another object is called instance variable.
2. For every object a separate copy of instance will be created.
3. Instance variables are created at object creation and destroyed at the time of object destruction.
4. Instance variables should be created within class.
5. JVM supplies default values for instances variables.

```
class sample
{
        int a ,b;  //  instance variables;
void display()
{
}
-----
-----
}
```

Static variables

1. if the value of variable is not varied from object to object is called static variable.
2. For static variables a single copy will created and shared by all objects of that class.
3. Static variables are declared by means of static modifier.
4. We can access the static members either by class-name (or) object reference. Usage of class-name is recommended.
5. Within the same class it is not required to use class-name , we can access directly.
6. JVM will provide default value for static variable.

Local variables

1. The variables which are declared inside a method such type of variables are called local variables.
2. Local variables are also called as stack variables or temporary variables or automatic variables.
3. JVM won't provide any default value.

```
class sample
{
void display()
{
int a ,b;  //  local variables;

}
-----
-----
}
```

Arrays

Definition:- An array is an indexed collection of fixed number of homogeneous data elements.

(or)

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

Arrays offers a convenient means of grouping related information.

Advantages

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

Disadvantages

- **Size Limit:** We can store only the 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 which grows automatically.

Types of arrays

Arrays are generally categorized into two parts

1. One-Dimensional arrays(or 1D arrays)
2. multi dimensional arrays(or 2D, 3D.. arrays)

In java obtaining an array is a two step process.

1. We must declare a variable of desired array type.(**Array Declaration**)
2. We must allocate the memory that will hold the array using new, and assign it to the array variable(**Array Construction**).

Array Declaration

Syntax: **type var-name[];**

Here type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus the base type for the array determines what type of data the array will hold.

For example

```
int arr[ ];
```

→ this declaration tells that **arr** is an array variable, no array actually exists. In fact value of **arr** is set to **null**(not existing/absence of value), which represents an array with no values.

Array Construction

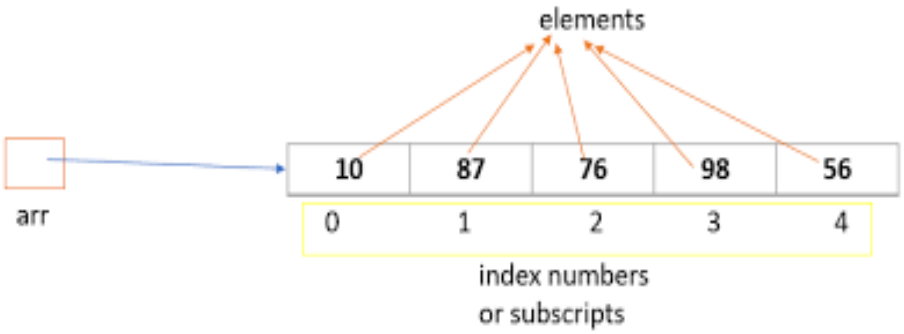
Every array is an object hence we can construct by using **new** operator. For every array type internally the corresponding classes are variable but these classes are not available to the programmer level.

General form of array construction using new operator

array-var=new type[size];

Here type specifies the type of data being allocated. Size specifies the number of elements in the array and ***array-var*** is the array variable that is linked to the array. The elements in the array allocated by new will automatically be initialized to **zero.**

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



We combine array declaration and array construction in single statement like

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

we know that in array the first element is stored at 0^{th} index and last element is stored at $(n-1)^{th}$ index. Where 'n' is the size of the array.

Some important points.

1. It is legal to have array with zero size i.e int a[]=new int[0];
2. The allowed data types for array size are byte, short, char, int.

```
int [] a=new int['a'];     //valid becoz ASCII code of 'a' is equal 97
int [] a=new int[10.5];   //invalid
int [] a=new int[10L];    //invalid
```

3. The java runtime system will check to be sure that all array index are in the correct range. If we try to access elements outside the range of the array (negative numbers or numbers greater than length of the array).it will cause run-time error.
4. As arrays are internally objects , it is having a property **length** which tells the size of the array.

For example int [] a=new int[6];

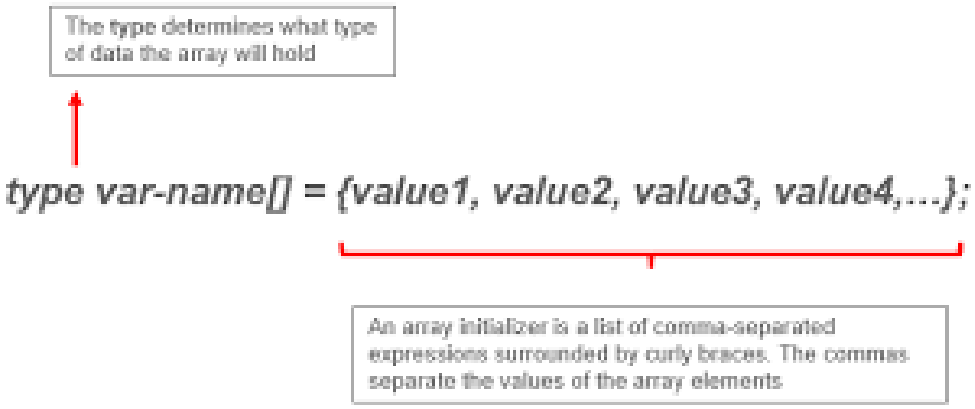
System.out.println(a.length) // output→ 6

5. Sometimes we can declare array without name. Such type of arrays are called **anonymous arrays**. The main purpose of anonymous arrays is just for instant use. We can create anonymous arrays as follows

new int[] {1,2,3,4,5};

Array Declaration, Construction and initialization in a single line

Arrays can be initialized when they are declared. An array intializer is a list of comma-separated expressions surrounded by curly braces. The comma separates the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array intializer.



For example → **int [ ] a={1,2,3,4,5};**

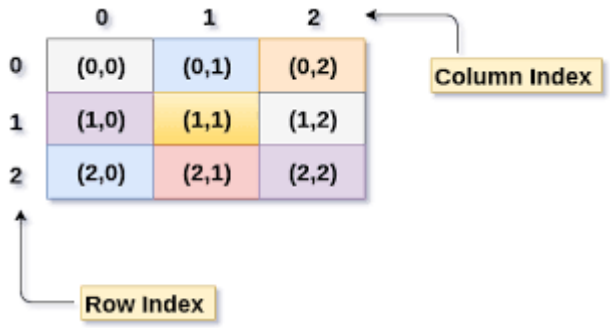
Multidimensional Arrays

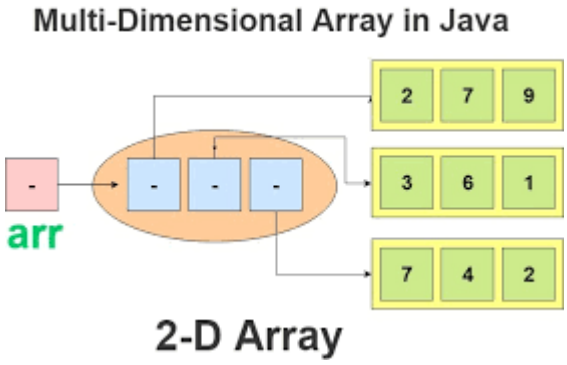
In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

For example, the following declares a two dimensional array variable called arr.

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

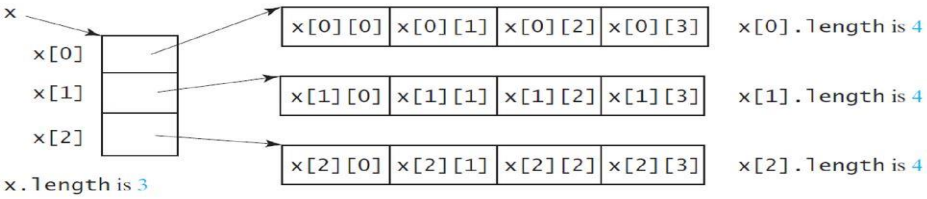
This allocates a 3 by 3 array and assigns it to **arr** Internally this matrix is implemented as an *array* of *arrays* of **int**.





Lengths of Two-dimensional Arrays

```
int[][] x = new int[3][4];
```



Initializing a Two-Dimensional Array

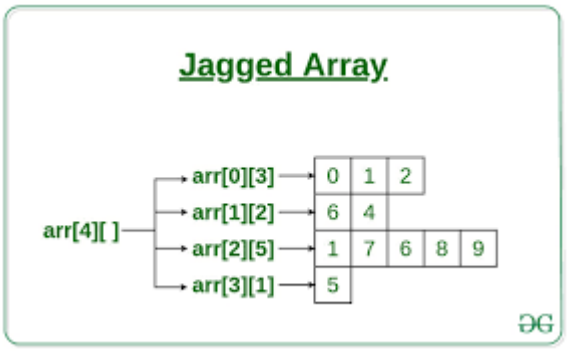
- Initializing a two-dimensional array requires enclosing each row's initialization list in its own set of braces.

```
int[][] numbers = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

- Java automatically creates the array and fills its elements with the initialization values.
  - row 0 {1, 2, 3}
  - row 1 {4, 5, 6}
  - row 2 {7, 8, 9}
- Declares an array with three rows and three columns.

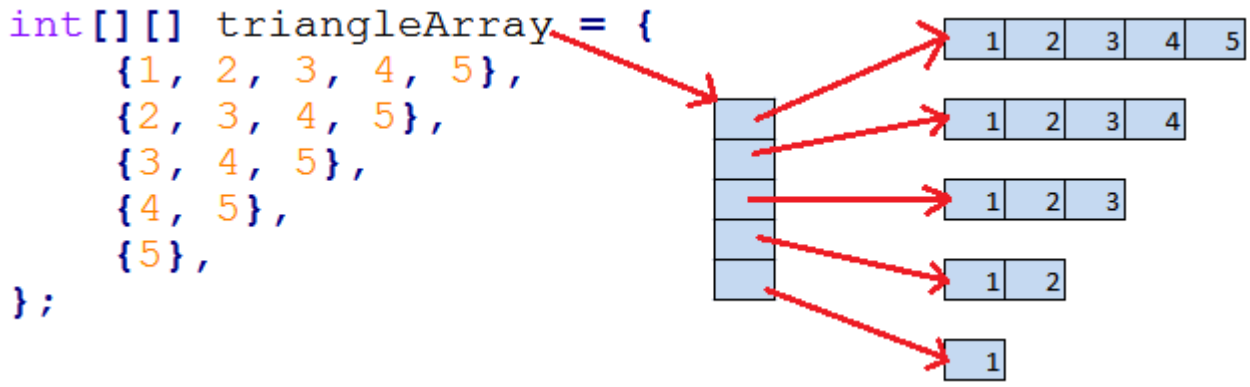
Ragged arrays

Ragged array is **an array of arrays such that member arrays can be of different sizes**, i.e., we can create a 2-D array but with a variable number of columns in each row. These types of arrays are also known as Jagged arrays.




```
int[][] arr = new int[4][];  // this will initialize the array elements by 0
arr[0] = new int[3];
arr[1] = new int[2];
arr[2] = new int[5];
arr[3] = new int[1];
```

Initialization the 2D array



Programs on Arrays

1. WAP to demonstrate 1D array.
2. WAP to demonstrate 2D arrays
3. WAP to display the following output(Ragged array)

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

4. WAP to display the following output

```
1   0   0

0   1   0

0   0   1
```

5. WAP to display the following output

```
1   2   2

3   1   2

3   3   1
```

1) Solution

```
import java.util.*;
class OneArray
{
public static void main(String args[])
{
int []a=new int[5];
Scanner s=new Scanner(System.in);
System.out.println("Enter 5 array elelments");
for(int i=0;i<a.length;i++)
a[i]=s.nextInt();

System.out.println("Entered Array elements are");
for(int i=0;i<a.length;i++)
System.out.print(a[i]+" ");
}
}
```

2) Solution

```
import java.util.*;
```

```
class TwoArray
{
public static void main(String args[])
{
int [][]a=new int[3][3];

Scanner s=new Scanner(System.in);
System.out.println("Enter 9 array elelments");
for(int i=0;i<a.length;i++)
{
for(int j=0;j<a[i].length;j++)
a[i][j]=s.nextInt();
}
System.out.println("Entered Array elements are");
for(int i=0;i<a.length;i++)
{
for(int j=0;j<a[i].length;j++)
System.out.print(a[i][j]+" ");
System.out.println();
}
}
}
```

3) Solution

```
import java.util.*;
class RaggedArray
{
public static void main(String args[])
{
int [][]a=new int[4][];

a[0]=new int[1];
a[1]=new int[2];
a[2]=new int[3];
a[3]=new int[4];
Scanner s=new Scanner(System.in);
System.out.println("Enter 9 array elelments");
for(int i=0,k=0;i<a.length;i++)
{
for(int j=0;j<a[i].length;j++,k++)
a[i][j]=k;
}
System.out.println("Entered Array elements are");
for(int i=0;i<a.length;i++)
{
for(int j=0;j<a[i].length;j++)
System.out.print(a[i][j]+" ");
```

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

4) Solution

```
import  java.util.*;
class  Test
{
public  static  void  main(String  args[])
{
int [][]a=new  int[3][3];

for(int  i=0,k=0;i<a.length;i++)
{
for(int  j=0;j<a[i].length;j++,k++)
{
if(i==j)
a[i][j]=1;
else
a[i][j]=0;
}
}
for(int  i=0;i<a.length;i++)
{
for(int  j=0;j<a[i].length;j++)
System.out.print(a[i][j]+" ");
System.out.println();
}
}
}
```

5) Solution

```
import  java.util.*;
class  Test1
{
public  static  void  main(String  args[])
{
int [][]a=new  int[3][3];


for(int  i=0,k=0;i<a.length;i++)
{
for(int  j=0;j<a[i].length;j++,k++)
{
if(i==j)
```

```
a[i][j]=1;
else  if(i<j)
a[i][j]=2;
else
a[i][j]=3;
}}
for(int  i=0;i<a.length;i++)
{
for(int  j=0;j<a[i].length;j++)
System.out.print(a[i][j]+" ");
System.out.println();
}}}
```

Operators

Operator:- Operator is a symbol that tells the compiler to perform mathematical operation and returns the resultant value.

Java provides a rich set of operator. Most of its operators are divided into the following four groups:

1. Arithmetic Operators.
2. Bitwise Operators.
3. Relational Operators, and
4. Logical Operators.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Meaning	Example
+	Addition	6+5=11
–	Subtraction (also unary minus)	6-5=1
*	Multiplication	6*5=30
/	Division	6/5=1
%	Modulus	6%5=1
++	Increment	
+=	Addition assignment	a+-=1 →a=a+1
–=	Subtraction assignment	a-=1 →a=a-1
*=	Multiplication assignment	a*=1 →a=a*1
/=	Division assignment	a/=1 →a=a/1
%=	Modulus assignment	a%=1 →a=a%1
– –	Decrement	

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

The 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. They are summarized in the following table:

Operator	Meaning	Example
~	Bitwise unary NOT or Bitwise one's complement operator	~(4) → 5
&	Bitwise AND	42 & 15 →10
|	Bitwise OR	42 | 15 → 47
^	Bitwise exclusive OR	42 ^ 15 → 37
>>	Shift right	42 >> 2 →10
>>>	Shift right zero fill	-1 >>> 24 → 255
<<	Shift left	42<<2 →168
&=	Bitwise AND assignment	
|=	Bitwise OR assignment	
^=	Bitwise exclusive OR assignment	
>>=	Shift right assignment	
>>>=	Shift right zero fill assignment	
<<=	Shift left assignment	

The Bitwise Logical Operators

The bitwise logical operators are **&**, |, ^, and **~**. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that 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

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result	Example
==	Equal to	5 == 5 → True
!=	Not equal to	5 != 6 →True
>	Greater than	5 > 6 →False
<	Less than	5 < 6 →True
>=	Greater than or equal to	6 > = 5→ True
<=	Less than or equal to	5 < =6 → True

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

Boolean Logical 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
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true = false** and **!false = true**. The following table shows the effect of each logical operation:

A	B	A | B	A & B	A^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

The Assignment Operator

You have been using the assignment operator since Chapter 2. Now it is time to take a formal look at it. The *assignment operator* is the single equal sign, **=**. The assignment operator works in Java much as it does in any other computer language. It has this general form:

var = *expression*;

Here, the type of *var* must be compatible with the type of *expression.* The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
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

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the **?** and it works in java much like it does in C and C++.

The **?:** has this general form:

expression1* ? *expression2* : *expression3

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can't be **void**.

Expressions:

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.

```
int a = 0;
arr[0] = 100;
System.out.println("Element 1 at index 0: " + arr[0]);

int result = 1 + 2; // result is now 3

if(value1 == value2)
System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression a= 0 returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, cadence is an int. As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String.

For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

```
x + y / 100     // ambiguous
```

You can specify exactly how an expression will be evaluated using balanced parenthesis rewrite the expression as

```
(x + y) / 100    // unambiguous, recommended
```

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

x + y / 100

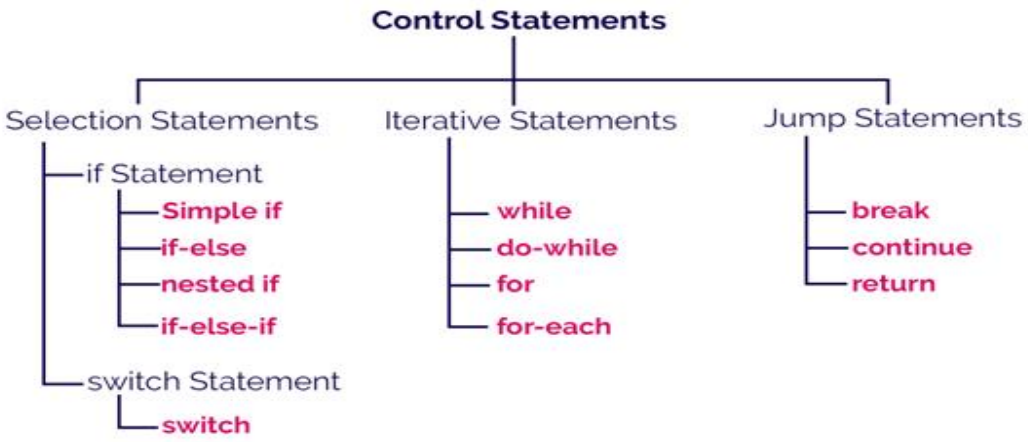
x + (y / 100) // unambiguous, recommended

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first.

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.



Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder

4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

syntax of if-else-if statement is given below.

```
if(condition 1) {
```

```
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate

- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
    case value1:
     statement1;
     break;
    .
    .
    .
    case valueN:
     statementN;
     break;
    default:
     default statement;
}
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

Looping/Iterative/Repetitive Control statement

In looping control instruction, a sequence of statements are executed repeatedly until some condition for termination of loop are satisfied.

Java language supports four different types of looping statements. They are

1. While Loop
2. Do-While Loop
3. For Loop
4. For-each Loop (used for arrays and Collections)

The While Statement

The while statement is used to carry out looping operations.

The General Form of while statement is

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

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

The do-While Statement

When a loop is constructed using the while statement, the test for continuation of the loop is carried out at the beginning of each pass. Sometimes , however ,it is desirable to have a loop with the test for continuation at the end of each pass. This can be accomplished by means of the **do-while** statement. The general form of the Do-While statement is

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

The included statements in the loop will be executed repeatedly, as long as the value of conditional expression true. Here the statements will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop. The statements can be either simple or compound, though most applications will require it to be a compound statement. When condition becomes false, control passes to the next line of code immediately following the loop.

For most applications it is more natural to test for continuation of a loop at the beginning rather than at the end of the loop. For this reason, the Do-while statement is used less frequently than the while statement.

The For statement

The *for statement* is the third and perhaps the most commonly used looping statement in java language. This statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued and a third expression that allows the index to the modifies at the end of each pass.

The general form of for statement is

for(initialization; condition; increment/decrement)

{

//body of loop

}

The execution of the *for* statement is as follows:

1. Initialization of the control variable is done first, using assignment statement such as i=1 or count=0. The variable i, count are known as loop-control variables.
2. The value of the control variable is tested using the condition. The condition is a relational expression , such as i<10 that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, control is transferred back to the *for* statement after evaluating the last statement in the loop. Now the control variable is incremented using an assignment statement such as i=i+1 and new value of the control variable is again tested to see whether it satisfies the loop

condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Differences between while and do-while

In while loop first the condition is checked whether true or false, if the condition is true then the control enters inside the block and executes the statements. Because the testing of the condition is done at the beginning it is also called as *pretest loop. In case of do-while loop the testing of the condition is done at the end after completing the execution of the statements. Hence it is called as post test loop.*

Nested loops

Loops can be nested (i.e., embedded) one within another. The inner and outer loops need not be generated by the same type of control structure. It is essential, however that one loop be completely embedded within the other-there can be no overlap. Also, each loop must be controlled by different index.

examples

1.
```
while(condition-1)
    {
      Statements;
while(condition-2)
      {
      Statements;
      }
    }
```

2.
```
for(initialization; condition-1; increment/decrement)
      {
            Statements;
for(initialization; condition-2; increment/decrement)
      {
            Statements;
      }}
```

Jump Statements

java language permits a jump from one statement to another within a loop as well as a jump out of a loop. The following are the jump statements.

1. Break.
2. Break with a label.
3. Continue.
4. Continue with a label.
5. Return.

Break statement

The break statement is used to terminate loops or to exit from a switch. It can be used within a *while, do-while, for* or a *switch* statement.

The break statement is written simply as

break;

without any embedded expressions or statements.

When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop. Usually we will use the break statement with if statement.

Break with label

Break with a label is used in the case where we can force the control to come out from named block.

The labeled break statement is written simply as

Break label;

Here , label is the name of a label that identifies a block of code. When this form of break executes, control is transferred out of the named block of code. The labeled block of code must enclose the break statement, but it does not need to be the immediate enclosing block. This means that you can use a labeled break statement to exit from a set of nested blocks.

To name a block, put a label at the start of it. A label is nay valid java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a break statement.

/* Demo program on Break with Label*/

```
class sample1
{
public static void main(String args[])
{
L1:
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{
if(i==j)
break L1;
System.out.println(i+"-----"+j);
}}}}
```

Continue statement

The continue statement is used to skip the current iteration i.e, on executing continue statement the continue is transferred to the beginning of the loop. In case of while or do-while the control is transferred to the conditional part. Whereas in for loop the control is transferred to the increment (or) decrement part. Usually we will use the continue statement with if statement.

The continue statement is written simply as

continue;

continue with label :

this is similar to labeled break but instead of making the control to come out from loop , the control is transferred to labeled loop by skipping the current iteration and proceeding with rest of the iterations in the loop.

Return Statement

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. At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus the return statement immediately terminates the method in which it is executed.

Introduction to classes

The class is at the core of java. It is the logical construct upon which entire java language is built. The importance of class is that it defines a new data type. Once

defined, this new type can be used to create objects of that type. Thus class is a template from an object, and an object is an instance of a class.

General Form of class declaration

```
class   classname
{
type instance_variable1;
type instance_variable 2;
……
type instance_variable n;
type methodname_1(parameter list)
{
        // body  of the method
}
type methodname_2(parameter list)
{
        // body  of the method
}
type methodname_n(parameter list)
{
        // body  of the method
}
}
```

The data or variables defined within a class are called instance variables. The code is contained within the method. Collectively, the methods and variables defined within a class are called members of the class. Variables defined within a class are called instance variables because each instance of the class (that is each object of the class) contains its own copy of these variables. Thus the data for one object is separate and unique from the data for other.

Declaring Object

When we create a class, you are creating a new data type. We can use this type to declare objects of that type. Creating objects of a class is a two-step process

1. We must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.

   classname class_variable;

2. We must acquire an actual physical copy of the object and assign it to the variable. We can do this using the new operator. The new operator dynamically

allocates memory for an object and returns a reference to it. This reference is more or less, the address in memory of the object allocated by new. This reference is then stored in the variable.

class_variable=new classname();

Here, class_var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parenthesis specifies the constructor for the class. Two steps can be combined into one steps i.e.

classname class_variable=new classname();

Introducing methods

Classes usually consists of two things: instance variables and methods.

General form of defining method

type name(*parameter-list*)
{
    // body of method
}

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty. Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement;

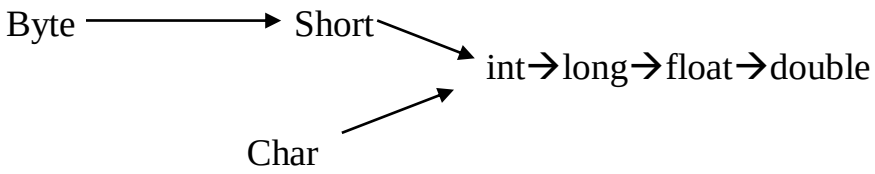
return *value*; Here, *value specifies the va*lue to be returned to the calling function

Method Overloading

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading.* Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

While overloading method resolution, if the compiler unable to find the method with required argument, it won't raise compile-time error immediately. First compiler promotes the argument to the next level and checks whether matched method is available or not. If it is not available then compiler once again promotes arguments to the next level and checks for the matched method. This process will be continued for all possible promotion. Still if the compiler unable to find the matched method then only it will raise compile-time error.
The following is the list of all possible automatic promotions



Constructors

A constructor is similar to a method that is used to initialize the instance variables. The sole purpose of a constructor is to initialize the instance variables.

Characteristics of constructors

1. The constructor's name and class name should be same.
2. A constructor may have or may not have parameters. Parameters are variables to receive data from outside into the constructor. If a constructor does not have any parameters it is called default constructor. If a constructor has 1 or more parameters, it is called parameterized constructor.
3. A constructor does not return any value, not even void.
4. A constructor is automatically called and executed at the time of creating an object. While creating an object, if nothing is passed to the object the default constructor is called and executed. If some value are passed to the object, then the parameterized constructor is called.

   Example Person p1 =new Person(); //default constructor
   Person p2=new Person("a,",22); //parameterized constructor

5. A constructor is called and executed only once per object. This means when we create an object the constructor is called. When we create second object, Again the constructor is called second time.
6. Like method overloading we can overload the constructors.

Access Modifiers

An access modifier is a keyword that specifies how to access the members of a class or a class itself. We can use access modifier before a class and its members. There are four access modifier available in java.

1. **Private :-** private members of a class are not accessible any where outside the class. They are accessible only within the class by the methods of that class.

2. **Public:** public members of class are accessible every where outside the class. So any other program can read them and use them.

3. **Protected:** When a class member is marked as protected, it can be accessed by the members of its own class, its subclasses, and classes in the same package and subclasses of another packages.

4. **Default:** if no access modifier is written by the programmer then the java compiler uses a default access modifier. Default members are accessible outside the class, but within the same package(directory).

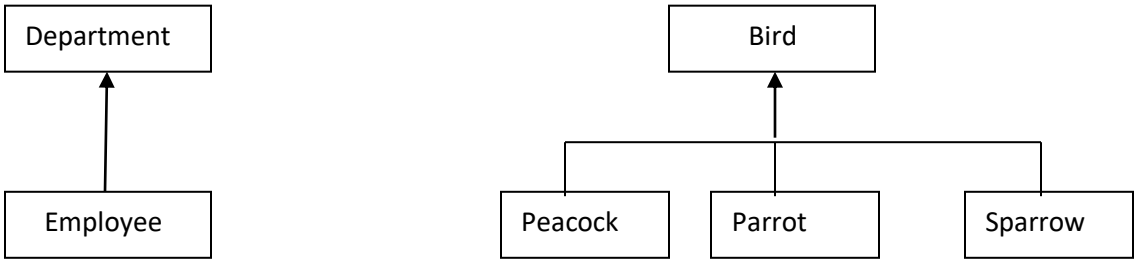
Introduction to Inheritance:

Deriving new class from existing classes such that the classes acquire all the features of existing classes is called inheritance.

Types of inheritance

Single inheritance

Producing sub classes from a single class is called single inheritance.



Class Employee extends Department

class Peacock extends Bird
class Parrot extends Bird
class Sparrow extends Bird

Important points on inheritance

1. A subclass extends a super class.
2. A subclass inherits all public instance variables and methods of the super class, but does not inherit the private instance variables and methods of the super class.
3. Inherited methods can be overridden; instance variables cannot be overridden(although they can be redefined in the subclass, but that; not the same thing and there's almost never a need to do it.
4. Use the **IS-A** test to verify that your inheritance hierarchy is valid. If X extends y, then X IS-A y must make sense.
5. When a method is overridden in a subclass and that method is invoked on an instance of the subclass, the overridden version of the method is called(the lowest on wins).
6. If class B extends A, and C extends B, class B **IS-A** class A and class C **IS-A** class B and class C also **IS-A** class A.

Constructors

A constructor is similar to a method that is used to initialize the instance variables. The sole purpose of a constructor is to initialize the instance variables.

Characteristics of constructors

1. The constructor's name and class name should be same.
2. A constructor may have or may not have parameters. Parameters are variables to receive data from outside into the constructor. If a constructor does not have any parameters it is called default constructor. If a constructor has 1 or more parameters, it is called parameterized constructor.
3. A constructor does not return any value, not even void.
4. A constructor is automatically called and executed at the time of creating an object. While creating an object, if nothing is passed to the object the default constructor is called and executed. If some value are passed to the object, then the parameterized constructor is called.

Example Person p1 =new Person(); //default constructor
Person p2=new Person("a,",22); //parameterized constructor

5. A constructor is called and executed only once per object. This means when we create an object the constructor is called. When we create second object, Again the constructor is called second time.
6. Like method overloading we can overload the constructors.

Difference between default constructor and parameterized constructor.

Default constructor	**Parameterized constructor**
Default constructor is useful to initialize all objects with same data.	Parameterized constructor is useful to initialize each object with different data.
Default constructor does not have any parameters.	Parameterized constructor will have 1 or more parameters.
When data is not passed at the time of creating an object. Default constructor is called.	When data is passed at the time of creating an object, parameterized constructor is called.

Constructors in inheritance

in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used. If super( ) is not used, then the default or parameterless constructor of each superclass will be executed

```
class A
{
A()
{
System.out.println(" A class Constructor");
}

}
class B extends A
{
B()
{
System.out.println(" B class Constructor");
}
}
```

```
class  C  extends  B
{
C()
{
System.out.println("  C  class  Constructor");
}
}
class  D
{
public  static  void main(String  args[])
{
C  c1=new  C();
}
}
```

Method overriding:

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

// demo program on Method overriding.

```
class A
{
int i, j;
 A(int a, int b)
{
i = a; j = b;
 }
 // display i and j
 void show()
{
System.out.println("i and j: " + i + " " + j)
}
}
class B extends A
{
 int k;
B(int a, int b, int c)
```

```
 {
 super(a, b); k = c;
}
// display k – this overrides show() in A
 void show()
{
System.out.println("k: " + k);
 } }
 class Override
{
 public static void main(String args[])
 {
 B subOb = new B(1, 2, 3);
 subOb.show(); // this calls show() in B
}
```

Abstract class

An abstract class is a class that contains 0 or more abstract methods.

(or)

A class that is declared using "abstract" keyword is known as abstract class. It may or may not include abstract methods which means in abstract class you can have concrete methods (methods with body) as well along with abstract methods(without an implementation, without braces, and followed by a semicolon). An abstract class can not be **instantiated** (you are not allowed to create **object** of Abstract class) but we can create reference for the abstract class.

Abstract class declaration

Specifying **abstract keyword** before the class during declaration, makes it abstract.

Abstract Method An abstract method is a method without method body. An abstract method is written when same method has to perform different tasks depending on the object calling it.

Syntax of abstract method:

abstract *type name(parameter-list)*;

Points to remember about abstract method:
1) Abstract method has no body.
2) Always end the declaration with a **semicolon**(;).
3) It must be overridden. An abstract class must be extended and in a same way abstract method must be overridden.
4) Abstract method must be in a abstract class.

Note: The class which is extending abstract class must override (or implement) all the abstract methods. otherwise the class itself becomes abstract.

Example program

```
abstract class Car
{
int regno;
Car(int r)
{
regno=r;
}
void openTank()
{
System.out.println("Fill the tank");
}
abstract void steering(int direction, int angle);
abstract void braking(int force);
}

class Maruti extends Car
{
Maruti(int regno)
{
super(regno);
}

void steering(int direction,int angle)
{
System.out.println(" Take turn");
System.out.println("This is an ordinary steering");
}
void braking(int force)
{
System.out.println("Brakes applied");
```

```
System.out.println("These are hydraulic brakes" );
}
}
class Santro extends Car
{
Santro(int regno)
{
super(regno);
}

void steering(int direction,int angle)
{
System.out.println(" Take turn");
System.out.println("This car uses power steering ");
}
void braking(int force)
{
System.out.println("Brakes applied");
System.out.println("This car uses gas brakes" );
}
}

class Usecar
{
public static void main(String args[])
{
Maruti m=new Maruti(1901);
Santro s=new Santro(5001);

Car ref;
ref=m;
ref.openTank();
ref.braking(500);
ref.steering(1,300);
}
}
```

Differences between method Overloading and method overriding

Method Overloading	Method Overriding
1. Writing two or more methods with the same name but with different signatures is called Method overloading	Writing two or more methods with the same name and same type signatures is called Method overriding.
2. Method Overloading is done in same class	Method overriding is done in super and sub classes.
3. In Method Overloading method return type can be same or different	In Method overriding method return types should be same.
4. JVM decides which method is called depending on the differences in the method signatures	JVM decides which method is called depending on the data types(class) of the object used to call the method.
5. Method Overloading is **code refinement**. Same method is refined to perform a different task.	Method overriding is **code replacement**. The sub class method overrides(replaces) the super class method.

Super keyword

If we create an object to super class, we can access only the super class members, but not the sub class members. but if we create the subclass object, all the members of both super and sub classes are available to it. This is the reason, we always create an object to sub class in inheritance. Some times, the super class members and sub class members may have same names. In that case by default only sub classes ,members are accessible. This shown in the following example program.

Example Program -1

```
class One
{
int i=10;
void show()
{
System.out.println("super class method i="+i);
}
}

class Two extends One
{
```

```
int i=20;
void show()
{
System.out.println("sub class meyhod i="+i);
}
}

class Super1
{
public static void main(String args[])
{
Two t=new Two();
t.show();
}
}
```

Whenever a sub class needs to refer to its immediate super class, irt can do so by use of the keyword super.

Super keyword can be used to refer to

1. Super class constructor.
2. Super class members(data members or methods)

Referring super class constructor

A subclass can call a constructor defined by its super class by use of the following from of super

super(arg-list).

Here, arg-list specifies any arguments needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass' constructor.

Example Program -2

```
class One
{
int i;
One(int i)
{
```

```
this.i=i;
}

}

class  Two  extends  One
{
int  i;
Two(int  a,int  b)
{
super(a);
i=b;
}

void  show()
{
System.out.println("sub  class  i="+i);
System.out.println("super   class  i="+super.i);
}
}

class  Super2
{
public  static  void main(String  args[])
{
Two  t=new Two(10,20);
t.show();
}
}
```

Referring super class members

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.
This usage has the following general form:

super.member

Here, member can be either a method or an instance variable

Example Program -3

```
class  One
{
```

```
int  i;
One(int i)
{
this.i=i;
}
}
class  Two  extends  One
{
int  i;
Two(int  a,int  b)
{
super(a);
i=b;
}
void  show()
{
System.out.println("sub  class  i="+i);
System.out.println("super   class  i="+super.i);
}
}
class  Super2
{
public  static  void main(String  args[])
{
Two  t=new Two(10,20);
t.show();
}
}
```

Packages

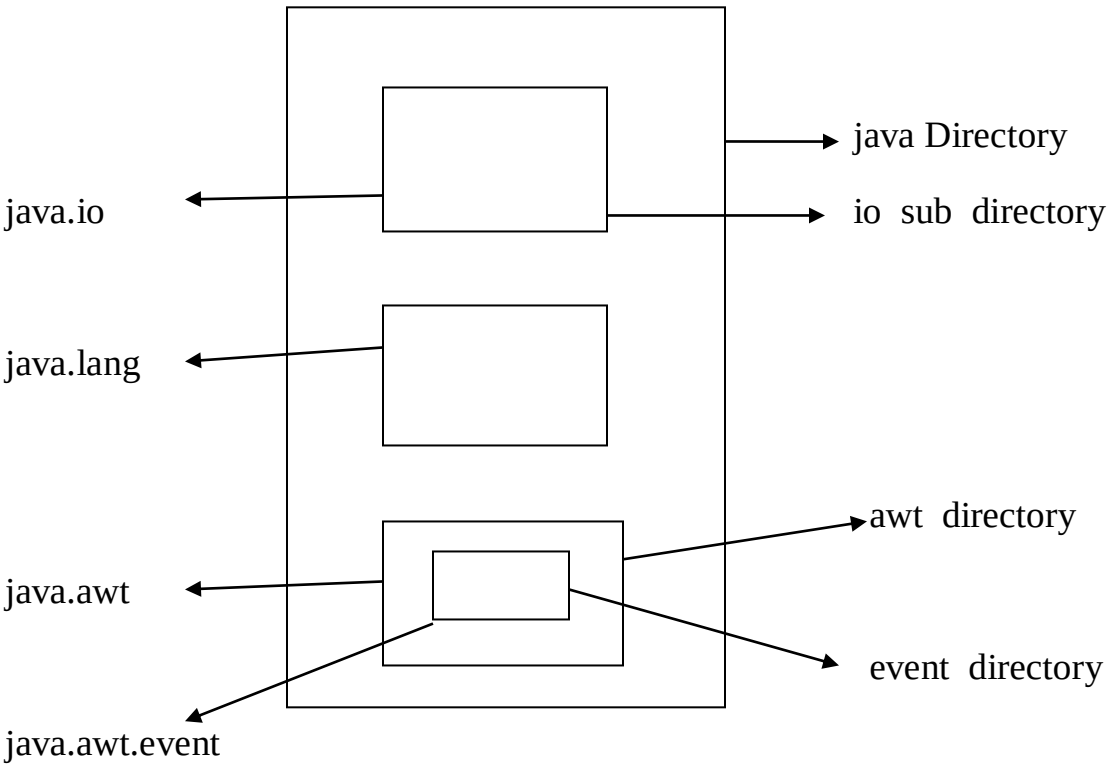
Definition:-Packages are containers for classes that are used to keep the class name space compartmentalized.

(or)

A package represents a directory that contains related group of classes and interfaces. We write the statement like

import java.io.*;

We are importing classes of java.io package. Here java is a directory name and io is another sub directory within it. And the * represents all the classes and interfaces of that io sub directory.



Advantages of packages

1. Packages are useful to arrange relates classes and interfaces into a group. This makes all the classes and interfaces performing the same task to put together in the same package. For example in java all the classes and interfaces which perform input and output operation are stored in java.io package.

2. The classes and interfaces of a package are isolated from the classes and interfaces of another package. This means that we can use same names for classes of two different classes. For example there is a Date class in **java.util** package and also there is another Date class available in **java.sql** package.
3. A group of packages is called library. The classes and interfaces of a package are like books in a library and can be reused several times. This reusability nature of packages makes programming easy.

Different types of packages

There are two different types of packages

1. Built-in packages
2. User-defined packages.

Built–in packages

These are the packages which are already available in java language. These packages provide all most all necessary classes, interfaces and methods for the programmer to perform any task in his program, foe everything there is a method available in java and that method can be used by the programmer without developing the logic on his own. This makes programming easy. Some of the important packages in java SE are

1. **java.lang**:- lang stands for language. This package got primary classes and interfaces essential for developing a basic java program. It consists of wrapper classes which are useful to convert primitive type to objects.
2. **java.io**:- io stands for input and output. This package contains streams. Streams are useful to store data in the for of files and also to perform input and output related tasks.
3. **java.util** :- util stands for utility. This package contains useful classes and interface like Stack. LinkedList, Vector, Arrays etc. these classes are called as collections. There are also classes for handling date and time operations.

User-defined packages:

Just like Built-in packages the use of java language can also create their own packages. They are called User-defined packages. User-defined packages can also

be imported into other classes and used exactly in the same way as in Built–in packages.

Defining a package

To create a package is quite easy: simply include a package command as the first statement. In a java source file. Ant classes declared within that file will belong to the specified packages. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name.

General form of the package statement

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**.
Remember that case is significant, and the directory name must match the package name exactly.
we can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. 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.

Access Specifiers

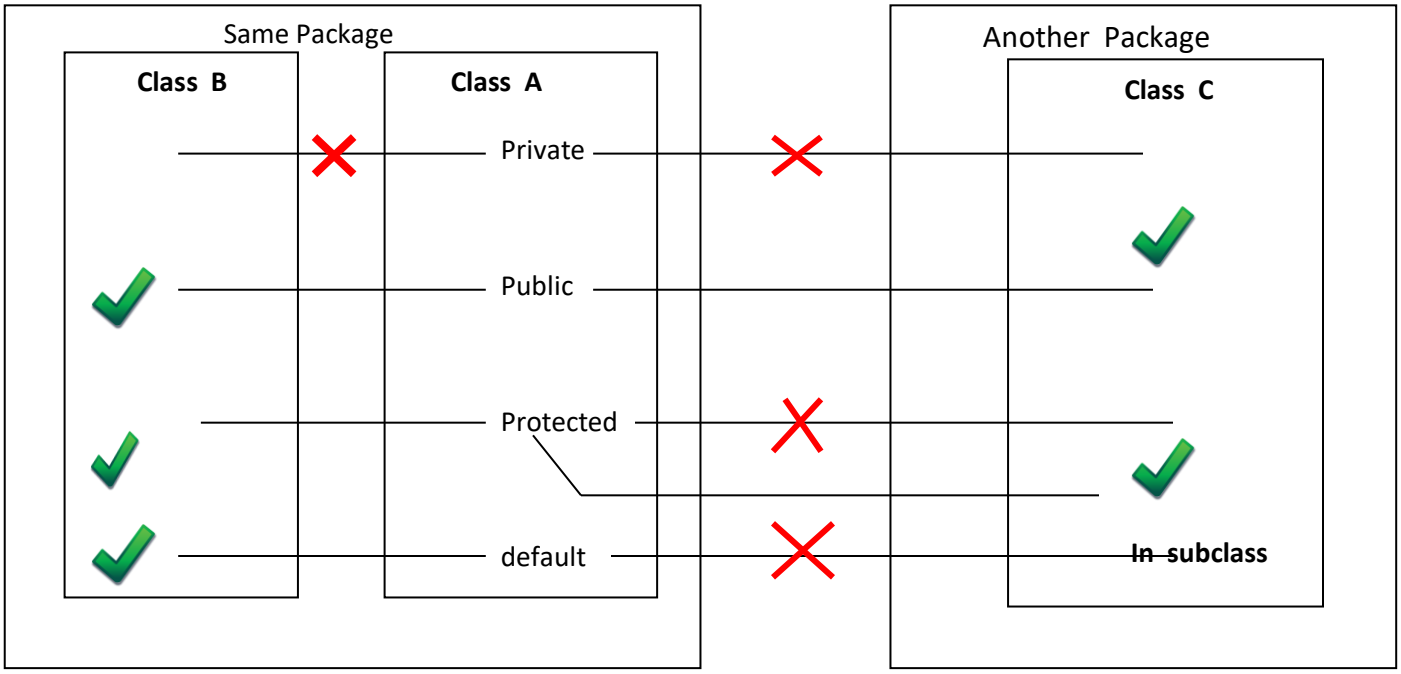
An access modifier is a keyword that specifies how to access the members of a class or a class itself. We can use access modifier before a class and its members. There are four access modifier available in java.

1. Private :- private members of a class are not accessible any where outside the class. They are accessible only within the class by the methods of that class.

2. Public: public members of class are accessible every where outside the class. So any other program can read them and use them.

3. **Protected:** protected members of a class are accessible only to classes in same package but outside the class, but only to classes that subclass your class directly.

4. Default: if no access modifier is written by the programmer then the java compiler uses a default access modifier. Default members are accessible outside the class, but within the same package(directory).



Private members of class A are not available to class B or class C. this means any private member's scope is limited only to that class where it is defined. so scope of private access specifier is class scope.

1. Public members of class a are available to class B and also class C. this means public members are available every where and their scope is global scope.
2. Protected members of class A are available to class B, but not in class c. but if class C is a sub class of class A then the protected members of

class A are available to class c. so protected access specifier acts as public with respect to sub class.

3. When no access specifier is used, it is taken as default specifier. Default members of class A are accessible to class B which is within the same package. They are not available to class C. this means the scope of default members is package scope.

Finding Packages and CLASSPATH:

Java run-time system checks packages that we create in 3 ways. They are

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

Access Protection

Packages are used not only to avoid class name collision but also access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.

Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

1. Subclasses in the same package
2. Non-subclasses in the same package
3. Subclasses in different packages
4. Classes that are neither in the same package nor subclasses.

TABLE 9-1 Class Member Access

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

```
package p1;

public class Protection {

int n = 1;

private int n_pri = 2;

protected int n_pro = 3;

public int n_pub = 4;

public Protection() {

System.out.println("base constructor");

System.out.println("n = " + n);

System.out.println("n_pri = " + n_pri);

System.out.println("n_pro = " + n_pro);

System.out.println("n_pub = " + n_pub);
```

```
}}

package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = "4 + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}

package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
```

```
}
}

package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}

package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
```

```
}
```

This is file OtherPackage.java:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

// Demo package p2.

```
package p2;
// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
```

```
OtherPackage ob2 = new OtherPackage();

}

}
```

Importing Packages in java

in java, the ***import*** keyword used to import built-in and user-defined packages. When a package has imported, we can refer to all the classes of that package using their name directly.

The import statement must be after the package statement, and before any other statement.

Using an import statement, we may import a specific class or all the classes from a package.

importing specific class

Using an importing statement, we can import a specific class. The following syntax is employed to import a specific class.

Syntax

```
Import  packageName.ClassName;
```

Importing all the classes

Using an importing statement, we can import all the classes of a package. To import all the classes of the package, we use * symbol. The following syntax is employed to import all the classes of a package.

Syntax

```
import packageName.*;
```

```
package MyPack;
public class Balance {
String name;
double bal;
public Balance(String n, double b) {
name = n;
```

```
bal = b;
}
public void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}

import MyPack.*;
class TestBalance {
public static void main(String args[]) {
/* Because Balance is public, you may use Balance
class and call its constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show()
}
}
```

Interface in java

Using the keyword interface, you can fully abstract a class‘ interface from its implementation.That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don‘t make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. Each class can determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the ―one interface, multiple methods‖ aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time.

Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and no

extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in a language such as C++.

Defining an Interface:

- An interface is defined much like a class.
- This is the general form of an

interface: access interface name

```
{

        return-type method-name1(parameter-list);

        return-type method-name2(parameter-list);

        type final-varname1 = value;

        type final-varname2 = value;

        return-type method-nameN(parameter-list);

        type final-varnameN = value;

}
```

Here, access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. name is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default

implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public.

An example of an interface definition. It declares a simple interface which contains one method called **callback( )** that takes a single integer parameter.

```
interface Callback
{
        void callback(int param);
}
```

Implementing Interfaces:

Once an interface has been defined, one or more classes can implement that interface.

To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

The general form of a class that implements the interface:

```
access class classname [extends superclass][implements interface
[,interface...]] {

// class-body
}
```

Here, access is either public or not used. If a class implements more than one interface,the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

Example: class that implements the Callback interface shown earlier.

```
class Client implements Callback
{
    public void callback (int p)
    {
    System.out.println("callback called with " + p);
    }}
```

Notice that callback( ) is declared using the public access specifier.When you implement an interface method, it must be declared as public.

For example, the following version of Client implements callback( ) and adds the method nonIfaceMeth( ):

```
class Client implements Callback
 {
 public void callback(int p)
    {
    System.out.println("callback called with " + p);

    }
    void nonIfaceMeth()
    {
    System.out.println("Classes that implement interfaces"+"may also define other members, too.");}}
```

Accessing Implementations Through Interface References:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the ―callee.‖ This process is similar to using a superclass reference to access a subclass object.

Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the callback( ) method via an interface reference variable:

```
class TestIface
{
    public static void main(String args[])
    {
    Callback c = new Client();
    c.callback(42);
    }
}
```

Output:

callback called with 42.

Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback( ) method, it cannot access any other members of the Client class. An interface reference variable only has knowledge of the methods declared by its interface declaration.

Thus, c could not be used to access nonIfaceMeth( ) since it is defined by Client but not Callback.

The preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of Callback, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback
{
```

```
public void callback(int p)
        {

        System.out.println("Another version of
        callback"); System.out.println("p squared is "
        + (p*p)); }

}
class TestIface2
{
        public static void main(String args[])
        {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();
        c.callback(42);
        c = ob; // c now refers to AnotherClient object
        c.callback(42);
        }
}
```

Output:

```
callback called with 42

Another version of callback
p squared is 1764
```

As you can see, the version of callback( ) that is called is determined by the type of object that c refers to at run time.

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

For example:

```
abstract class Incomplete implements Callback
{
        int a, b;
        void show()
        {
        System.out.println(a + " " + b);
        }
}
```

Here, the class Incomplete does not implement callback( ) and must be declared as abstract. Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

Applying Interfaces:

We define a stack interface, leaving it to each implementation to define the specifics. Let‘s look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called IntStack.java. This interface will be used by both stack implementations. Example:

```
// Define an integer stack
interface. interface IntStack
{
        void push(int item); // store an
        item int pop(); // retrieve an item
}
```

The following program creates a class called FixedStack that implements a fixed-length version of an integer stack:

Example:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack
```

```
{
        private int stck[];
        private int tos;

        FixedStack(int size)
        stck = new int[size];
        tos = -1;
}

// Push an item onto the
stack public void push(int
item)
{
        if(tos==stck.length-1) // use length member

        System.out.println("Stack is full.");
        else
        stck[++tos] = item;
}

// Pop an item from the
stack public int pop()
{
        if(tos < 0)
        {

        System.out.println("Stack
        underflow."); return 0;

        }
        else
```

```
        return stck[tos--];
        }
    }
    class IFTest
     {
        public static void main(String args[])
        {
        FixedStack mystack1 = new FixedStack(5);


        // push some numbers onto the stack
        for(int i=0; i<5; i++)
        mystack1.push(i); for(int i=0; i<8;
        i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in
        mystack1:"); for(int i=0; i<5; i++)
        System.out.println(mystack1.pop());
        System.out.println("Stack in
        mystack2:"); for(int i=0; i<8; i++)
        System.out.println(mystack2.pop());

        }
    }
```

Following is another implementation of IntStack that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled. Example:

```
    // Implement a "growable" stack.
    class DynStack implements
    IntStack

    {
    private int stck[];
    private int tos;
```

```
// allocate and initialize
stack DynStack(int size)

{

stck = new int[size];
tos = -1;
}

// Push an item onto the
stack public void push(int
item)
{
// if stack is full, allocate a larger
stack if(tos==stck.length-1)
{

int temp[] = new int[stck.length * 2]; // double
size for(int i=0; i<stck.length; i++)

temp[i] = stck[i];
stck = temp;
stck[++tos] = item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop()
 {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
class IFTest2
{
public static void main(String args[])
```

```
 {
DynStack mystack1 = new DynStack(5);
DynStack mystack2 = new DynStack(8);

// these loops cause each stack to grow
for(int i=0; i<12; i++)
mystack1.push(i); for(int i=0; i<20;
i++) mystack2.push(i);
System.out.println("Stack in
mystack1:");

for(int i=0; i<12; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop());
}
}
```

The following class uses both the FixedStack and DynStack implementations.It does so through an interface reference. This means that calls to push( ) and pop( )are resolved at run time rather than at compile time.

```
/* Create an interface variable and access stacks through it.
*/
class IFTest3

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

IntStack mystack; // create an interface reference
variable DynStack ds = new DynStack(5);
FixedStack fs = new FixedStack(8);

mystack = ds; // load dynamic stack

// push some numbers onto the
stack for(int i=0; i<12; i++)
mystack.push(i);
```

```
mystack = fs; // load fixed
stack for(int i=0; i<8; i++)
mystack.push(i);

mystack = ds;

System.out.println("Values in dynamic
stack:"); for(int i=0; i<12; i++)
System.out.println(mystack.pop());

mystack = fs;

System.out.println("Values in fixed
stack:"); for(int i=0; i<8; i++)
System.out.println(mystack.pop());

}
}
```

In this program, mystack is a reference to the IntStack interface. Thus, when it refers to ds, it uses the versions of push( ) and pop( ) defined by the DynStack implementation. When it refers to fs, it uses the versions of push( ) and pop( ) defined by FixedStack.

These determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Variables in Interfaces:

Interfaces can be used to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values. When we include that interface in a class (that is, when you ―implement‖ the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations. If an interface contains no methods, then any class that includes such an interface doesn‘t actually implement anything. It is as if that class were importing the constant variables into the class name space as final variables

Interfaces Can Be Extended:

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits

another interface, it must provide implementations for all methods defined within the interfaceinheritance chain.

Example:

```
// One interface can extend
another. interface A
 {

void    meth1();
void meth2();

}

// B now includes meth1() and meth2() -- it adds
meth3(). interface B extends A

{

void meth3();

}

// this class must implement all of A and B
class MyClass implements B

{

public void meth1()

 {

System.out.println("Implement meth1().");

}

public void meth2()

{

System.out.println("Implement meth2().");

}

public void meth3()

{
```

```
System.out.println("Implement meth3().");
}
}
class IFExtend
{
public static void main(String arg[])
{

MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}}
```

Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Exception Handling

An exception is an abnormal condition that arises in a code sequence at run time. In other words run-time error.A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

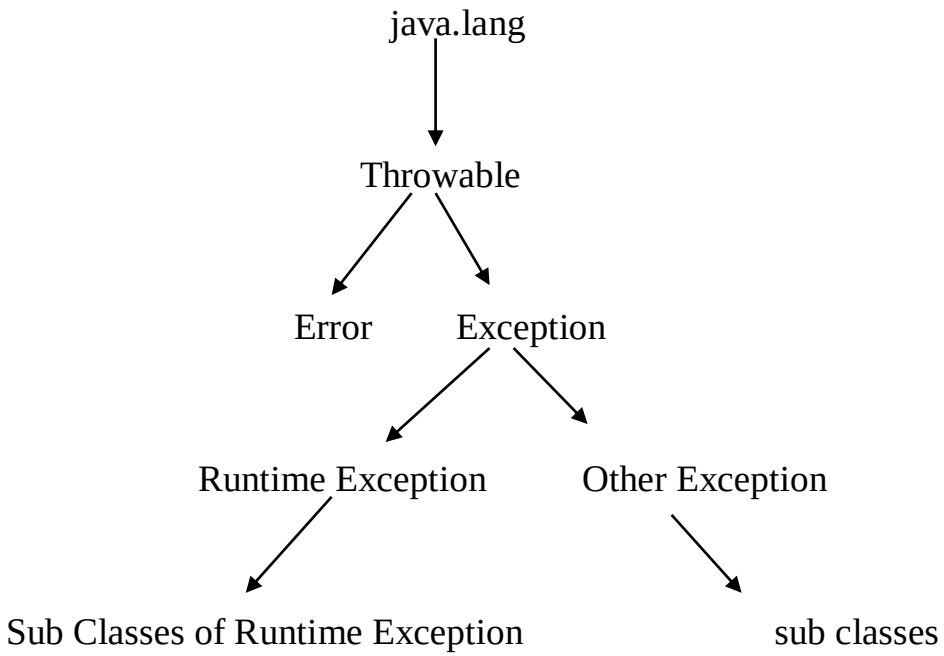
When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the java run-time system, or they can be manually generated by your code.

Advantage of Exception Handling

1. It allows you to fix the error.
2. It prevents the program from automatically terminating.

Exception Types

All the exception types are subclasses of the built-in class Throwable. Thus, Throwable is the top of the exception class hierarchy. This is as shown in the figure.



Java exception handling is managed via five keywords **try, catch, throw, throws and finally.**

General form of Exception Handling block

```
try
{
 // block of code to monitor for errors
}
 catch (ExceptionType1 exOb)
{
 // exception handler for ExceptionType1
 }
 catch (ExceptionType2 exOb)
{
 // exception handler for ExceptionType2
}
…..
 Finally
 {
 // block of code to be executed after try block ends
 }
```

When there is an exception, the user data may be corrupted. This should be tackled by the programmer by carefully designed the program. For this, we should perform the following 3 steps

Step 1: The programmer should observe the statements in his program where there may be possibility of exception. Such statements should be written inside a try block. A try block looks like as follows

```
try
{
Statement
}
```

The greatness of try block is that even if some exceptions arises inside it, the program will not be terminated when JVM understands that there is an exception. It stores the exception details in an exception stack and then jumps into a catch bock. **In simple words to guard against and handle a run-time error. Simple enclose the code that you want to monitor inside the try block**

Step2:

The programmer should write the catch block where he should display the exception details to the user. This helps the user to understood that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error. Catch block look like as follows

```
catch (Exceptionclass ref)
        {
                Statements
        }
```

The **reference ref above** is automatically adjusted to refer to the exception stack where the details of the exception are available. So we can display the exception details using any one of the following

a. using print() or println() methods such as System.out.println(ref);
b. using printStackTrace() method of Throwable class, which fetches exception details from the exception stack and display them

The goal of most well constructed clause should be to resolve the exceptional condition and then continue on as if the error had never happened.

Step :3

Lastly the programmer should perform clean up operations like closing the files and terminating the threads. The programmer should write this code in the finally block.

```
finally
{
Statements
 }
```

The specialty of finally block is that the statements inside the finally block are executed irrespective of whether there is an exception or not. This ensures that all the opened files are properly closed and all the running threads are properly terminated. So the data in the files will not be corrupted and user is at the safe side.

Performing the above three tasks is called 'exception Handling remember in exception handling the programmer is not preventing the exception, as in many cases it is not possible. But the programmer is avoiding any damage that may happen to user data.

throw clause

using throw it is possible for our program to throw an exception explicitly.

The general form of throw is shown below

throw ThrowableInstance;

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

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

throws clause

even if the programmer is not handling runtime exceptions, the java compiler will not give any error related to runtime exception. But the rule is that the programmer should handle checked exceptions. In case the programmer does not want to handle the checked exception he should throw them out using throws clause. Otherwise there will be an error flagged by the java compiler.

General form of a method declaration that includes a throws clause is:

type method-name(parameter-list) throws exception-list

{

// body of method

}

Java's Built-in Exceptions

Checked Exception:-The Exception which are checked by the compiler for smooth execution of the program at runtime are called checked Exception.

Unchecked Exception:- The Exception which are checked at runtime by JVM is called Unchecked Exception

Java's Unchecked RuntimeException Subclasses Defined in java.lang

1. ArithmeticException → Arithmetic error, such as divide-by-zero.
2. ArrayIndexOutOfBoundsException → Array index is out-of-bounds.
3. ArrayStoreException →Assignment to an array element of an incompatible type.
4. ClassCastException →Invalid cast.
5. EnumConstantNotPresentException→ An attempt is made to use an undefined enumeration value.
6. IllegalArgumentException →Illegal argument used to invoke a method.
7. IllegalMonitorStateException →Illegal monitor operation, such as waiting on an unlocked thread.
8. IllegalStateException →Environment or application is in incorrect state.
9. NegativeArraySizeException→ Array created with a negative size.
10. NullPointerException →Invalid use of a null reference.
11. NumberFormatException →Invalid conversion of a string to a numeric format.

Java's Checked Exceptions Defined in java.lang

1. ClassNotFoundException Class not found.
2. CloneNotSupportedException Attempt to clone an object that does not implement the Cloneable interface.
3. IllegalAccessException Access to a class is denied.
4. IllegalArgumentException Illegal argument used to invoke a method.
5. InstantiationException Attempt to create an object of an abstract class or interface.

6. InterruptedException One thread has been interrupted by another thread.
7. NoSuchFieldException A requested field does not exist.
8. NoSuchMethodException A requested method does not exist.

Programs on Exception Handling

/* demo program which causes abnormal termination of program*/

```
class Exam1
{
public  static void main(String  args[])
{
int d=0;
int a=42/d;
}
}
```

Output

```
C:\exception>javac  Exam1.java

C:\exception>java Exam1
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exam1.main(Exam1.java:6)
```

/* Alternative way of above program which uses method */

```
class Exam2
{
static  void subroutine()
{
int d=0;
int a=10/d;
}
public  static void main(String  args[])
{
Exam2.subroutine();
}
}
```

Output

```
C:\exception>javac Exam2.java

C:\exception>java Exam2
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exam2.subroutine(Exam2.java:6)
        at Exam2.main(Exam2.java:10)
```

/* program which uses try catch to avoid abnormal termination*/

```
class  Exam3
{
public  static  void main(String  args[])
{
int a,d;
try
{
d=0;
a=42/d;
System.out.println("This will not  be printed");
}catch(ArithmeticException e)
{
System.out.println("Division  by  Zero");
}
System.out.println("After catch Statement");
}
}
```

Output

```
C:\exception>javac Exam3.java

C:\exception>java Exam3
Division  by  Zero
After catch Statement
```

/* demo program which uses multiple ctach statement which handle multiple exceptions*/

```
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.");
}
}
```

Output

```
C:\exception>javac MultiCatch.java

C:\exception>java  MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

/* program which uses nested try statements */

```
class NestTry { public static void main(String args[])
 { try
{ int a = args.length;
 int b = 42 / a;
System.out.println("a = " + a);
try {
if(a==1)
a = a/(a-a);
if(a==2) { int c[] = { 1 };
c[42] = 99;
}
```

```
} catch(ArrayIndexOutOfBoundsException e) { System.out.println("Array index out-
of-bounds: " + e); }
} catch(ArithmeticException e) { System.out.println("Divide by 0: " + e); }
}
}
```

Output

```
C:\exception>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\exception>
```

/* demo program which uses throw keyword */

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

// This is now correct.
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);
}
}
}
```

/* demo program on finally */

```
// 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();
}
}
```

/* demo program on throws keyword */

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
// This is now correct.
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);
}
}
}
```

Creating User-defined Exception

Sometimes the built-in exception in java are not able to describe a certain situation. In such cases, like the built-in exception, the user can also create his own exception which are called 'user-defined exceptions.. the following steps are followed in creation of user-defined exception.

1. The user should create an exception class as a sub class to Exception class. Since all exceptions are subclasses of exception class, the user should also make his class a subclass to it. This done as

   Class MyException extends Exception
2. The user can write a detail constructor in his own exception class. He can use it. In case he does not want to store any exception details. If the user

does not want to create an empty object to his exception class, he can eliminate the default constructor.

MyException(){}

3. The user can create an parameterized constructor with string as a parameter. He can use this to store exception details. He can call super class constructor from this and send the string.

```
MyException(String str)
{
super(str);
}
```

4. When the user wants to raise his own exception. He should create an object to his exception class and throw it using throw class

```
MyException me=new MyException("Exception details");
    Throw me;
```

Example program

To understand how to create user-defined exception. Let us write a program in which we are creating our own exception class MyException. In this program we are taking the details of account numbers, customer names and balance amounts in the form of three arrays. Then in main() method, we display these details using a for loop. At this time, we check if in any account the balance amount less than the minimum balance to be kept in the account. If so, then MyException is raised and a message is displayed "Balance amount is less"

```
class MyException extends Exception
{
private static int accno[]={1001,1002,1003,1004,1005};
private static String name[]={"Raja Rao","Rama rao","Subba rao","appa
rao","Laxmi Devi"};
private static double bal[]={10000,12000,5500,99.00,55};
MyException()
{
}

MyException(String str)
{
super(str);}

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

```
{
try
{
System.out.println("AccNo"+"\t"+"Customer"+"\t"+" Balance");
for(int  i=0;i<5;i++)
{
System.out.println(accno[i]+"\t"+name[i]+"\t"+bal[i]);
if(bal[i]<1000)
{
MyException me=new   MyException("balance  amount  is less");
throw  me;
}
}
}catch(MyException  me)
{
me.printStackTrace();
}

}
}
```

C:\Users\svit\Desktop\java Notes>javac MyException.java

C:\Users\svit\Desktop\java Notes>java MyException
AccNo Customer Balance
1001 Raja Rao 10000.0
1002 Rama rao 12000.0
1003 Subba rao 5500.0
1004 appa rao 99.0
MyException: balance amount is less
at MyException.main(MyException.java:26)

Chained Exceptions

Chained *Exception* helps to identify a situation in which one exception causes another *Exception* in an application.**For instance, consider a method which throws an *ArithmeticException*** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero.The method will throw the *ArithmeticException* to the caller. The caller would not know about the actual cause of an *Exception*. Chained *Exception* is used in such situations.

To allow chained exceptions, two constructors and two methods were added to Throwable. The constructors are shown here:

Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)

In the first form, causeExc is the exception that causes the current exception. That is, causeExc is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception.

The chained exception methods added to Throwable are getCause( ) and initCause( ).

Throwable getCause( )
Throwable initCause(Throwable causeExc)

The getCause( ) method returns the exception that underlies the current exception. If there is no underlying exception, null is returned. The initCause( ) method associates causeExc with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. Thus, you can call initCause( ) only once for each exception object.

// Demonstrate exception chaining.

```
class ChainExcDemo {
static void demoproc() {
// create an exception
NullPointerException e =
new NullPointerException("top layer");
// add a cause
e.initCause(new ArithmeticException("cause"));
throw e;
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
// display top level exception
System.out.println("Caught: " + e);
// display cause exception
System.out.println("Original cause: " +
e.getCause());
}
}}
```

Multitasking

Executing several tasks simultaneously is the concept of multitasking

There are 2 types of multitasking

1. Process based multitasking
2. Thread based multitasking

Process based multitasking

Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called process based multitasking.

Example while typing a java program in the editor we can able to listen mp3 audio songs simultaneously at the same time we can download a file from the internet. All these are executing simultaneously and independent of each other. It is process based multitasking. This type multitasking best suitable for OS level.

Thread based multitasking

Executing several tasks simultaneously where each task is a separate independent part of the same program. Such type of multitasking is called thread based multitasking.This type of multitasking is best suitable for programmatic level and each independent part is called thread.

Java provides inbuilt support for multithreading by introducing a rich library (Thread, Runnable, ThreadGroup etc) ***Whether it is process based or thread based the main objective of multitasking is to improve performance, by reducing response time.***

The main application areas of multithreading are developing video games and multimedia graphics etc.

Difference b/w process based and thread based multitasking.

Process	Thread
Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called process based multitasking.	Executing several tasks simultaneously where each task is a separate independent part of the same program. Such type of multitasking is called thread based multitasking.
Process is heavy weight component.	Thread is a light weight component.

Each process has a separate memory address	All threads share same memory address
Process is uncontrollable	Thread is controllable
IPC(Inter Process Communication) is expensive and limited	Inter Thread Communication(ITC) is inexpensive.
Context switching from one process to another process is also costly.	Context switching from one thread to another thread is low cost.

The ways to define, instantiate and starting a new thread.

we can define the threads in the following 2 ways

1. By extends thread class
2. By implements Runnable interface.

Method: 1 (by extends from thread class)

```
Public class mythread extends Thread
{
        Public void run()
        {
        for(int i=0;i<10;i++)
        System.out.println("Child Thread");
        }
}
Class ThreadDemo
{
Public static void main(string args[])
        {
        mythread t=new mythread();
        t.start();
        for(int i=0;i<10;i++)
        System.out.println("Parent Thread");
        }

}
```

Method-2(By implements Runnable interface)

We can define a thread even by implementing Runnable interface directly.

Runnable interface present in java.lang package and contains only one method i. e run().

```
interface Runnable()
{
public void run();
}
```

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName)

In this constructor, threadOb is an instance of a class that implements the Runnable interface.This defines where execution of the thread will begin. The name of the new thread is specified by threadName.
After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ). The start( ) method is shown here:

//demonstrating program.

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
```

```
        System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
    }
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on **this** object. Next, **start( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin. After calling **start( )**, **NewThread**'s constructor returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows. (Your output may vary based on processor speed and task load.)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower priority thread.

Every thread in java has some priority. The range of thread priorities is 1 to **10 (1—>least, 10—>highest)**.

Thread class defines the following constants to represent some standard priorities

Thread.MAX_PRIORITY whose value is 10

Thread.NORM_PRIORITY whose value is 5

Thread.MIN_PRIORITY whose value is 1

The thread priorities used by thread Scheduler while allocating CPU. The thread which is having highest priority will get chance first for execution.

Default Priority

The default priority for only main thread is '5'. But for all the remaining threads it will be inherited from parent to child i.e whatever will be the parent thread has the priority the same will be the priority of child thread.

Thread class defines the following methods to get and set priority of a thread.

Public final int getPriority();
Public final int setPriority(int priority);

it should be from 1 to 10 otherwise we will get run time Exception saying **"illegalArgumentException"**

Example

```
class MyThread extends Thread
{
public void run()
{
for(int i=0;i<5;i++)
{
System.out.println ("Child Thread");
}}}

class ThreadPriorityDemo
{
public static void main(String args[])
{
MyThread t=new MyThread();
t.setPriority(10);
t.start();
for(int i=0;i<5;i++)
{
System.out.println ("Main Thread");
}}}
```

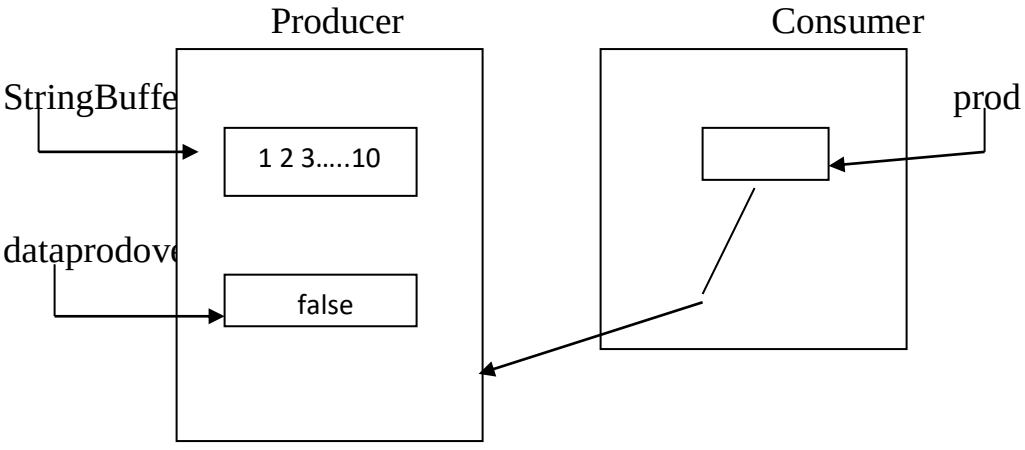
Inter Thread Communication

In some cases, two or more threads should communicate with each other, for example a consumer thread is waiting for a producer to produce the data (or some goods). When the producer thread completes production of data, then the consumer thread should take the data and use it. For example

In the Producer class, we take a StringBuffer object to store data. In this case we take some numbers from 1 to 10. These numbers are added to StringBuffer object. We take another Boolean variable dataprodover, and initialize it to false. The idea is to make this dataprodover true when the production of numbers is completed. Producing data is done by appending numbers to StringBuffer using a for lop. This may take time. When appending we come out of for loop and then store true into dataprodover.

When producer is busy producing the data, now and then the consumer will check if dataprodover is true or not. If dataprodover is true, the consumer takes the data from StringBuffer and uses it. If the dataprodover shows false, then consumer will sleep for some time and then again checks the dataprodover.

In this way the producer and consumer can communicate with each other. But this is not an efficient way of communication because consumer checks the dataprodover at some point of time and finds it false. So it goes into sleep for next 10 milliseconds. Meanwhile the data production may be over. But consumer comes out of sleep after 10 milliseconds and then only it can find dataprodover is true. This means that there may be a time delay of 1 to 9 milliseconds to receive the data after its actual production is completed.



hence to make two threads communicate each other we make use of the following functions

1. **wait()**
   when a running thread calls wait, the thread enters a waiting state where it waits in a queue associated with the particular object on which wait() was called.
   public final void wait() throws interrupted Exception
   public final void wait(long ms) throws interrupted Exception

2. **notify()**
   the first thread in the wait queue for a particular object becomes ready on a call to notify() method, issued by another associated with that object.

   public final native void notify()

3. **notifyAll()**
   every thread in the wait queue for a given object becomes ready.

   public final native void notifyAll()

/* program about ITC without using wait(),notify(),notifyall() methods*/

```
class Producer implements Runnable
{
StringBuffer sb;
boolean dataprodover=false;
Producer()
{
sb=new StringBuffer();
}

public void run()
{
for(int i=1;i<=10;i++)
{
try
{
```

```
sb.append(i+":");
Thread.sleep(100);
System.out.println("Appending.....");
}catch(Exception e){System.out.println(e);}

}
dataprodover=true;
}
}
class  Consumer  implements Runnable
{
Producer  prod;
Consumer(Producer p)
{
prod=p;
}
public void run()
{
while(!prod.dataprodover)
{
try
{
Thread.sleep(10);
}catch(Exception e){System.out.println(e);}
}
System.out.println(prod.sb);

}
}

class syncDemo
{
public  static  void main(String args[])
{
Producer  p=new  Producer();
Consumer  obj=new Consumer(p);
Thread  t1=new  Thread(p);
Thread t2=new  Thread(obj);
t1.start();
t2.start();
}
}
```

/*program about ITC with wait(),notify() methods */

```
class  Producer  implements Runnable
{
StringBuffer  sb;
Producer()
{
sb=new  StringBuffer();
}

public  void run()
{
synchronized(sb)
{
for(int i=1;i<=10;i++)
{
        try
        {
        sb.append(i+":");
        Thread.sleep(100);
        System.out.println("Appending.....");
        }catch(Exception e){System.out.println(e);}
}
}
sb.notify();
}
}
class  Consumer  implements Runnable
{
Producer  prod;
Consumer(Producer p)
{
prod=p;
}
public void run()
{
synchronized(prod.sb)
{
try
{
prod.sb.wait();
```

```
}catch(Exception e){System.out.println(e);}

System.out.println(prod.sb);
}
}
}
class syncDemo1
{
public  static  void main(String args[])
{
Producer  p=new  Producer();
Consumer  obj=new Consumer(p);
Thread  t1=new  Thread(p);
Thread t2=new  Thread(obj);
t1.start();
t2.start();
}}
```

Synchronization:

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

In java we can achieve synchronization in two ways

1. synchronized methods
2. synchronized blocks

Synchronized method

in Java, synchronization is built into the language, and it's easy to use because every object has something called a **monitor**. Think of a monitor as a special lock that controls access to an object by multiple threads.

1. **Implicit Monitor for Each Object**:
   - Every object in Java has its own monitor (lock) automatically.
   - When a thread wants to access a synchronized method of an object, it must first "acquire" the monitor for that object.
2. **Synchronized Methods**:
   - When you mark a method as synchronized, it means that only **one thread** at a time can execute that method on the **same instance** of the object.
   - If another thread tries to call the same synchronized method (or any other synchronized method on the same object), it **must wait** until the first thread finishes and releases the monitor.

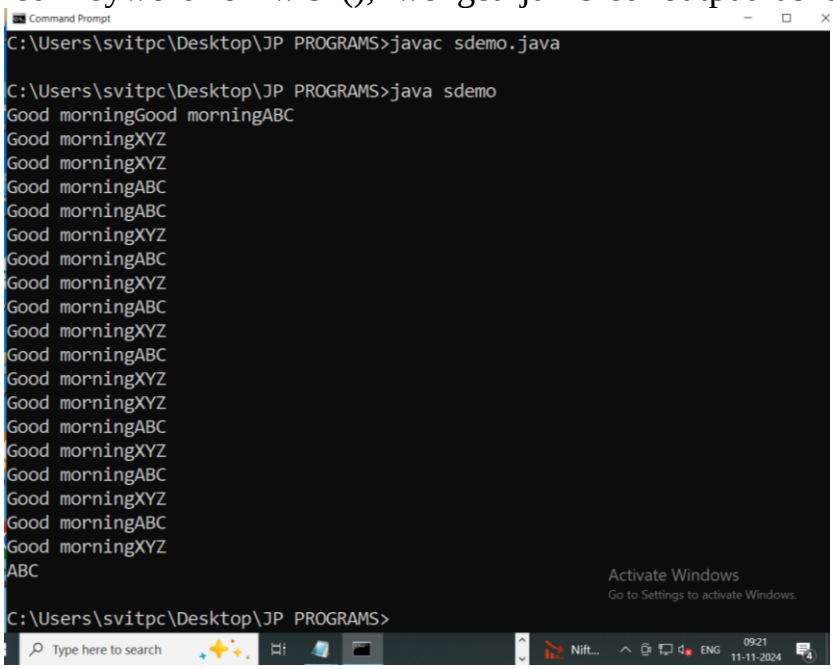
3. **Releasing the Monitor**:
    - Once the thread finishes executing the synchronized method, it **releases the monitor**, allowing other waiting threads to acquire it and execute their synchronized methods.

Programs on thread Synchronization using Synchronized keyword.

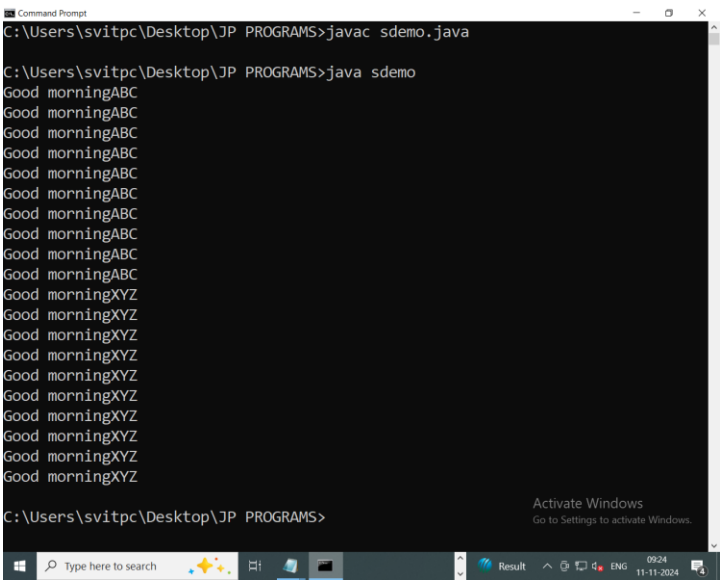
```
class  Display
{
public s
ynchronized  void wish(String  name)
{
for(int  i=0;i<10;i++)
{
System.out.print("Good morning");
try
{
Thread.sleep(500);
}catch(Exception e){}
System.out.println(name);
}}}

class mythread1 extends Thread
{
Display  d;
String name;
mythread1(Display d, String name)
{
this.d=d;
this.name=name;
}
public  void run()
{
d.wish(name);
}}

class sdemo
{
public  static  void main(String args[])
{
Display  d1=new Display();
mythread1 t1=new  mythread1(d1,"ABC");
```

```
mythread1 t2=new  mythread1(d1,"XYZ");
t1.start();
t2.start();
}}
```

Without synchronized keyword for wish(), we get jumbled output as shown below.



By preceding the synchronized keyword before wish method, we get correct output(after completing "ABC" thread then "XYZ thread starts its execution) , which is shown below.



synchronized block.

Why Synchronized Methods Aren't Enough:

If the class you're working with was **not designed for multithreaded access**, it may not have the proper synchronization mechanisms in place. Simply adding synchronized to the methods of your own class won't protect the shared instances of third-party classes from being accessed concurrently in an unsafe way.

Solution:

To synchronize access to an object of this third-party class, you can use a **synchronization block** in your own code. This can be achieved by synchronizing on an external lock object when accessing the third-party object

This is the general form of the **synchronized** statement:

```
synchronized (object) {

        // statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor

```
class  Display1
{
public  void wish(String  name)
{
for(int  i=0;i<10;i++)
{
System.out.print("Good morning");
try
{
Thread.sleep(500);
}catch(Exception e){}
System.out.println(name);
}}}

class mythread2 extends Thread
{
```

```
Display1  d;
String name;
mythread2(Display1 d,String name)
{
this.d=d;
this.name=name;
}
public  void run()
{
synchronized(d)
{
d.wish(name);
}}}

class sdemo1
{
public  static  void main(String args[])
{
Display1  d1=new Display1();
mythread2 t1=new  mythread2(d1,"ABC");
mythread2 t2=new  mythread2(d1,"XYZ");
t1.start();
t2.start();
}}
```

Thread Life Cycle

Life Cycle of a Thread

Threads, the smallest unit of a process, have a life cycle. In Java, this life cycle features six main states that any thread can occupy at a given point in time:

1. New

A thread is in this state when you've created an instance of the Thread class but haven't invoked the start() method yet. It remains in this state until the program starts the thread.

2. Active

This state consists of two sub-states, Runnable and Running. Runnable implies that the thread is ready for execution and is waiting for resource allocation by the thread scheduler. Running means the thread scheduler has selected the thread and is currently executing its run() method.

3. Blocked / Waiting

A thread enters this state when it is temporarily inactive and waiting for a signal to proceed due to reasons like waiting for a resource to become available (Blocked) or waiting for another thread to perform a specific action (Waiting).

4. Timed Waiting

In this state, a thread is waiting for a specified period. A thread might enter this state through methods like Thread.sleep(long millis) or Object.wait(long timeout) where it waits for a particular duration before resuming its activities.

5. Terminated

This is the final state in the thread life cycle. The thread arrives here when it has completed its execution, i.e., its run() method has been completed, or it has been abruptly terminated due to an unhandled exception. Once in this state, the thread cannot be resumed.

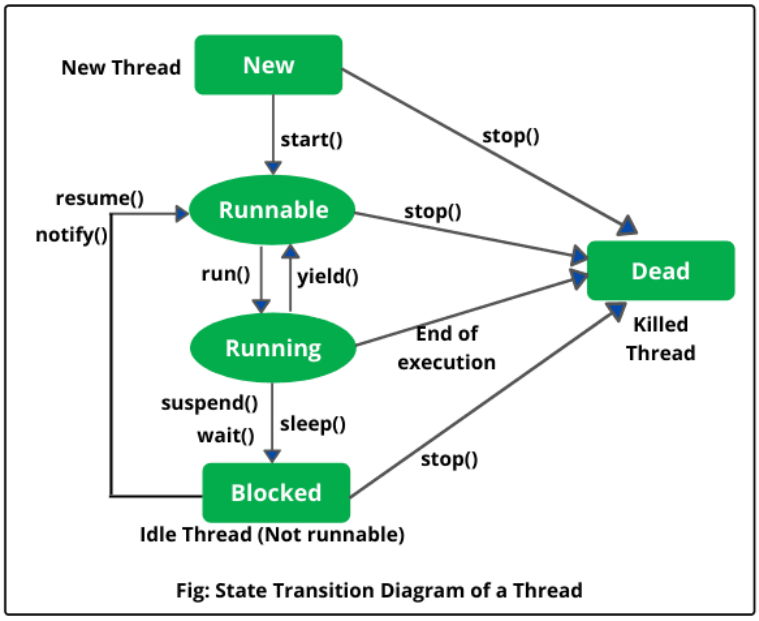


Fig: State Transition Diagram of a Thread

AutoBoxing and Auto unboxing

Until 1.4 version we are not allowed to provide primitive values in the place of wrapper objects and wrapper objects in the place of primitive.. compulsory programmer is responsible to convert

File

Although most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

Files are a primary source and destination for data within many programs. Although there are severe restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

Here, ***directoryPath*** is the path name of the file, ***filename*** is the name of the file or subdirectory, ***dirObj*** is a File object that specifies a directory, and ***uriObj*** is a

URI object that describes a file.

File defines many methods that obtain the standard properties of a **File** object. For example, **getName( )** returns the name of the file, **getParent( )** returns the name of the parent directory, and **exists( )** returns **true** if the file exists, **false** if it does not. The **File** class, however, is not symmetrical. By this, we mean that there are a few methods that allow you to *examine* the properties of a simple file object, but no corresponding function exists to change those attributes. The following example demonstrates several of the **File** methods:

```
// Demonstrate File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");
    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

When you run this program, you will see something similar to the following:

```
File Name: COPYRIGHT
Path: /java/COPYRIGHT
Abs Path: /java/COPYRIGHT
Parent: /java
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified: 812465204000
File size: 695 Bytes
```

Directories

A directory is a **File** that contains a list of other files and directories. When you create a **File** object and it is a directory, the **isDirectory( )** method will return **true**. In this case, you can call **list( )** on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

String[ ] list( )

The list of files is returned in an array of **String** objects.

The program shown here illustrates how to use **list( )** to examine the contents of a directory:

```
// Using directories.
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);

    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```

Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory
```

The listFiles( ) Alternative

There is a variation to the **list( )** method, called **listFiles( )**, which you might find useful. The signatures for **listFiles( )** are shown here:

File[ ] listFiles( )
File[ ] listFiles(FilenameFilter *FFObj*)
File[ ] listFiles(FileFilter *FObj*)

These methods return the file list as an array of **File** objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified **FilenameFilter**. Aside from returning an array of **File** objects, these two versions of **listFiles( )** work like their equivalent **list( )** methods.

The third version of **listFiles( )** returns those files with path names that satisfy the specified **FileFilter**. **FileFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *path*)

The **accept( )** method returns **true** for files that should be included in the list (that is, those that match the *path* argument), and **false** for those that should be excluded.

Creating Directories

Another two useful **File** utility methods are **mkdir( )** and **mkdirs( )**. The **mkdir( )** method creates a directory, returning **true** on success and **false** on failure. Failure indicates that the path specified in the **File** object already exists, or that the directory cannot be created because the entire path does not exist yet. To create a directory for which no path exists, use the **mkdirs( )** method. It creates both a directory and all the parents of the directory.

***/Listing the contents of a Directory using listFiles()*/**

```
import  java.io.*;
class  test
{
 public void list1(String directoryName){
       File directory = new File(directoryName);
       //get all the files from a directory
       File[] fList = directory.listFiles();
       for (File file : fList){
          if (file.isFile()){
             System.out.println(file.getAbsolutePath());
          } else if (file.isDirectory()){
             list1(file.getAbsolutePath());
          }
       }
   }
public  static void main(String args[])
{
test  t=new   test();
```

```
t.list1("E:\\Lab Index");
}
}
```

OUTPUT

```
D:\>javac  test.java

D:\>java test
E:\Lab Index\CPP A.docx
E:\Lab Index\IT B\CPP B.docx
E:\Lab Index\IT B\CPP.xlsx
E:\Lab Index\IT B\ITWS B.docx
E:\Lab Index\IT B\ITWS.xlsx
E:\Lab Index\IT B\JAVA B.docx
E:\Lab Index\IT B\JAVA.xlsx
E:\Lab Index\ITWS.docx
E:\Lab Index\IWS A.docx
E:\Lab Index\JAVA A.docx
E:\Lab Index\New Microsoft Office Word Document.docx
E:\Lab Index\ravali mail id.txt
```

Utility classes

StringTokenizer

The processing of text often consists of parsing a formatted input string. Parsing is the division of text into a set of discrete parts, or tokens, which in a certain sequence can convey a semantic meaning. The **StringTokenizer** class provides the first step in this parsing process, often called the lexer (lexical analyzer) or scanner.

To use **StringTokenizer**, you specify an input string and a string that contains delimiters. Delimiters are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, ",;:" sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage return.

The StringTokenizer constructors are shown here:

1. **StringTokenizer(String str)**

2. **StringTokenizer(String str, String delimiters)**
3. **StringTokenizer(String str, String delimiters, boolean delimAsToken)**

In all versions, str is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, delimiters is a string that specifies the delimiters. In the third version, if delimAsToken is true, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;
class STDemo {
static String in = "title=Java: The Complete Reference;" +
"author=Schildt;" +
"publisher=Osborne/McGraw-Hill;" +
"copyright=2007";

public static void main(String args[]) {

StringTokenizer st = new StringTokenizer(in, "=;");
while(st.hasMoreTokens()) {
String key = st.nextToken();
String val = st.nextToken();
System.out.println(key + "\t" + val);
}
}
}
```

The output from this program is shown here:

```
title Java: The Complete Reference
author Schildt
publisher Osborne/McGraw-Hill
copyright 2007
```

Method	Description
int countTokens( )	Using the current set of delimiters, the method determines the number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
boolean hasMoreTokens( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
Object nextElement( )	Returns the next token as an **Object**.
String nextToken( )	Returns the next token as a **String**.
String nextToken(String *delimiters*)	Returns the next token as a **String** and sets the delimiters string to that specified by *delimiters*.

TABLE 18-1 The Methods Defined by **StringTokenizer**

BitSet

A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed. This makes it similar to a vector of bits. The BitSet constructors are shown here:

BitSet( )
BitSet(int size)

The first version creates a default object. The second version allows you to specify its initial size (that is, the number of bits that it can hold). All bits are initialized to zero.

Method	Description
void and(BitSet *bitSet*)	ANDs the contents of the invoking **BitSet** object with those specified by *bitSet*. The result is placed into the invoking object.
void andNot(BitSet *bitSet*)	For each 1 bit in *bitSet*, the corresponding bit in the invoking **BitSet** is cleared.
int cardinality( )	Returns the number of set bits in the invoking object.
void clear( )	Zeros all bits.
void clear(int *index*)	Zeros the bit specified by *index*.
void clear(int *startIndex*, int *endIndex*)	Zeros the bits from *startIndex* to *endIndex*–1.
Object clone( )	Duplicates the invoking **BitSet** object.
boolean equals(Object *bitSet*)	Returns **true** if the invoking bit set is equivalent to the one passed in *bitSet*. Otherwise, the method returns **false**.
void flip(int *index*)	Reverses the bit specified by *index*.
void flip(int *startIndex*, int *endIndex*)	Reverses the bits from *startIndex* to *endIndex*–1.
boolean get(int *index*)	Returns the current state of the bit at the specified index.
BitSet get(int *startIndex*, int *endIndex*)	Returns a **BitSet** that consists of the bits from *startIndex* to *endIndex*–1. The invoking object is not changed.
int hashCode( )	Returns the hash code for the invoking object.
boolean intersects(BitSet *bitSet*)	Returns **true** if at least one pair of corresponding bits within the invoking object and *bitSet* are 1.
boolean isEmpty( )	Returns **true** if all bits in the invoking object are zero.
int length( )	Returns the number of bits required to hold the contents of the invoking **BitSet**. This value is determined by the location of the last 1 bit.
int nextClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next zero bit), starting from the index specified by *startIndex*.

TABLE 18-2 The Methods Defined by **BitSet**

Method	Description
int nextSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by *startIndex*. If no bit is set, –1 is returned.
void or(BitSet *bitSet*)	ORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.
void set(int *index*)	Sets the bit specified by *index*.
void set(int *index*, boolean *v*)	Sets the bit specified by *index* to the value passed in *v*. **true** sets the bit, **false** clears the bit.
void set(int *startIndex*, int *endIndex*)	Sets the bits from *startIndex* to *endIndex*–1.
void set(int *startIndex*, int *endIndex*, boolean *v*)	Sets the bits from *startIndex* to *endIndex*–1, to the value passed in *v*. **true** sets the bits, **false** clears the bits.
int size( )	Returns the number of bits in the invoking **BitSet** object.
String toString( )	Returns the string equivalent of the invoking **BitSet** object.
void xor(BitSet *bitSet*)	XORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.

TABLE 18-2 The Methods Defined by **BitSet** *(continued)*

Here is an example that demonstrates BitSet:

```
// BitSet Demonstration.
import java.util.BitSet;
class BitSetDemo {
public static void main(String args[]) {
BitSet bits1 = new BitSet(16);
```

```
BitSet bits2 = new BitSet(16);
// set some bits
for(int i=0; i<16; i++) {
if((i%2) == 0) bits1.set(i);
if((i%5) != 0) bits2.set(i);
}
System.out.println("Initial pattern in bits1: ");
System.out.println(bits1);
System.out.println("\nInitial pattern in bits2: ");
System.out.println(bits2);
// AND bits
bits2.and(bits1);
System.out.println("\nbits2 AND bits1: ");
System.out.println(bits2);
// OR bits
bits2.or(bits1);
System.out.println("\nbits2 OR bits1: ");
System.out.println(bits2);
// XOR bits
bits2.xor(bits1);
System.out.println("\nbits2 XOR bits1: ");
System.out.println(bits2);
}
}
```

Date

The **Date** class encapsulates the current date and time. Before beginning our examination of **Date**, it is important to point out that it has changed substantially from its original version defined by Java 1.0. When Java 1.1 was released, many of the functions carried out by the original **Date** class were moved into the **Calendar** and **DateFormat** classes, and as a result, many of the original 1.0 **Date** methods were deprecated. Since the deprecated 1.0 methods should not be used for new code, they are not described here.

Date supports the following constructors:

Date( )
Date(long *millisec*)

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970. The nondeprecated methods defined by **Date** are shown in Table 18-3. **Date** also implements the **Comparable** interface.

Method	Description
boolean after(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is later than the one specified by *date*. Otherwise, it returns **false**.
boolean before(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is earlier than the one specified by *date*. Otherwise, it returns **false**.
Object clone( )	Duplicates the invoking **Date** object.
int compareTo(Date *date*)	Compares the value of the invoking object with that of *date*. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than *date*. Returns a positive value if the invoking object is later than *date*.
boolean equals(Object *date*)	Returns **true** if the invoking **Date** object contains the same time and date as the one specified by *date*. Otherwise, it returns **false**.
long getTime( )	Returns the number of milliseconds that have elapsed since January 1, 1970.
int hashCode( )	Returns a hash code for the invoking object.
void setTime(long *time*)	Sets the time and date as specified by *time*, which represents an elapsed time in milliseconds from midnight, January 1, 1970.
String toString( )	Converts the invoking **Date** object into a string and returns the result.

TABLE 18-3 The Nondeprecated Methods Defined by **Date**

// Show date and time using only Date methods.

```
import java.util.Date;
class DateDemo {
public static void main(String args[]) {

// Instantiate a Date object

Date date = new Date();

// display time and date using toString()
System.out.println(date);
// Display number of milliseconds since midnight, January 1, 1970 GMT
long msec = date.getTime();
System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
}
}
```

Calendar

The abstract **Calendar** class provides a set of methods that allows you to convert a time in milliseconds to a number of useful components. Some examples of the type of information that can be provided are year, month, day, hour, minute, and second. It is intended that subclasses of **Calendar** will provide the specific functionality to interpret time information according to their own rules. This is one aspect of the Java class library that enables you to write programs that can operate in international environments. An example of such a subclass is **GregorianCalendar**.

Calendar provides no public constructors.

Calendar defines several protected instance variables. **areFieldsSet** is a **boolean** that indicates if the time components have been set. **fields** is an array of **int**s that holds the components of the time. **isSet** is a **boolean** array that indicates if a specific time component has been set. **time** is a **long** that holds the current time for this object. **isTimeSet** is a **boolean** that indicates if the current time has been set.

Some commonly used methods defined by **Calendar** are shown in Table 18-4.

Method	Description
abstract void add(int *which*, int *val*)	Adds *val* to the time or date component specified by *which*. To subtract, add a negative value. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**.
boolean after(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is later than the one specified by *calendarObj*. Otherwise, it returns **false**.
boolean before(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is earlier than the one specified by *calendarObj*. Otherwise, it returns **false**.
final void clear( )	Zeros all time components in the invoking object.
final void clear(int *which*)	Zeros the time component specified by *which* in the invoking object.
Object clone( )	Returns a duplicate of the invoking object.
boolean equals(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is equal to the one specified by *calendarObj*. Otherwise, it returns **false**.

TABLE 18-4 Commonly Used Methods Defined by **Calendar**

The following program demonstrates several **Calendar** methods:

```
// Demonstrate Calendar
import java.util.Calendar;

class CalendarDemo {
  public static void main(String args[]) {
    String months[] = {
             "Jan", "Feb", "Mar", "Apr",
             "May", "Jun", "Jul", "Aug",
             "Sep", "Oct", "Nov", "Dec"};

    // Create a calendar initialized with the
    // current date and time in the default
    // locale and timezone.
    Calendar calendar = Calendar.getInstance();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[calendar.get(Calendar.MONTH)]);
    System.out.print(" " + calendar.get(Calendar.DATE) + " ");
    System.out.println(calendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));

    // Set the time and date information and display it.
    calendar.set(Calendar.HOUR, 10);
    calendar.set(Calendar.MINUTE, 29);
    calendar.set(Calendar.SECOND, 22);
     System.out.print("Updated time: ");
     System.out.print(calendar.get(Calendar.HOUR) + ":");
     System.out.print(calendar.get(Calendar.MINUTE) + ":");
     System.out.println(calendar.get(Calendar.SECOND));
   }
 }
```

Sample output is shown here:

```
Date: Jan 1 2007
Time: 11:24:25
Updated time: 10:29:22
```

String Handling

A *string* is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type **String**. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient

The String Constructors

The **String** class supports several constructors. To create an empty **String**, we can use the default constructor. For example,

1. **String s = new String();**

will create an instance of **String** with no characters in it.

To create a **String** initialized by an array of characters, use the constructor shown here:

2. **String(char *chars*[ ])**

Here is an example:

```
char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);
```

This constructor initializes **s** with the string “abc”.

We can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

3. String(String *strObj*)

Here, *strObj* is a **String** object

Methods of String class

1. **String Length**

The length of a string is the number of characters that it contains. To obtain this value, call the

length( ) method,

int length( )

The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

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

2. **charAt( )**

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

char charAt(int *where*)

Here, *where* is the index of the character that you want to obtain. The value of *where* must be
nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value "**b**" to **ch**.

3. **getChars( )**

If we need to extract more than one character at a time, you can use the **getChars( )** method.
It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target.* The index within *target* at which the substring will be copied is passed in *targetStart.* Care must be taken to assure that the

target array is large enough to hold the number of characters in the specified substring.

4. **equals(Object *str*)**

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

5. **compareTo( )**

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order.
The **String** method **compareTo( )** serves this purpose. It has this general form:

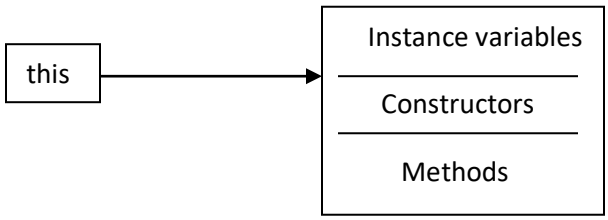
int compareTo(String *str*)

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison
is returned and is interpreted, as shown here:

Value	Meaning
Less than zero	The invoking string is less than str.
Greater than zero	The invoking string is greater than str.
Zero	The two strings are equal.

The keyword this

'*this*' is a keyword that refers to the object of the class where it is used. In other words 'this' refers to the object of the present class. Generally, we write instance variables, constructors and methods in a class. All these members are referenced by 'this'. When an object is created to a class, a default reference is also created internally to the object. This default reference is nothing but 'this' so this can refer to all the things of the present object.



Differences between equals() and "=="

==	Equals()
1. We can apply this operator for both primitive and object references	We can apply only for object references but not for primitives
2. In the case of object references it is always meant for references comparison	By default equals() ,method is also meant for reference comparison(from object class equals())
3. We can't override for content comparison	We can override for content Comparison.
4. In the case of different types of objects it raises compile time saying "**incomparable types"**	In the case of different types of objects this method returns false.
5. For any object references 'r' r==null is always false.	For any object references 'r' r.equals (null) is always false.

Final Keyword

A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared.(final is similar to const in C/C++).

It is common coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant. ***Example:*** final int MAXSIZE=10;

Java static keyword

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

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block

1) Java static variable

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

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

Advantage of static variable

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

2) Java static method

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

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

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. ***this*** and ***super*** cannot be used in static context.

3) Java static block

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