



VARDHAMAN COLLEGE OF ENGINEERING

(AUTONOMOUS)

Shamshabad – 501 218, Hyderabad

DEPARTMENT OF INFORMATION TECHNOLOGY

OBJECT ORIENTED PROGRAMMING



Unit-I

Computer programming Paradigms

1. process-oriented model

In a process-oriented model, **code acting on data**. Example: C .with this approach programs complexity is increased.

2. object-oriented programming

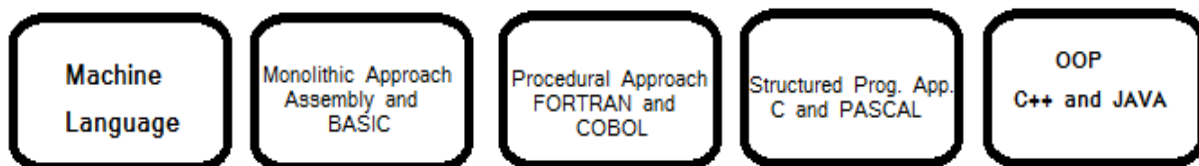
An object-oriented program can be characterized as **data controlling access to code**

Object-oriented programming

Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic.

A program has been viewed as a logical procedure that takes input data, processes it, and produces output data.

Evolution of OOPS



History of OOPS

1. The basis for OOP started in the early 1960s
2. the first programming language to use objects was Simula 67
3. Many consider that the first truly O-O language was Smalltalk, developed at the Learning Research Group at Xerox's Palo Alto Research Center in the early 1970s.
4. C++ was implemented in 1982 under the name C with Classes.
5. Bjarne Stroustrup design C++ by combining some of the features of Simula with the syntax of C.
6. Later Java was introduced.

Difference between Object Oriented Programming and Procedural Programming

Procedural Programming	Object Oriented Programming
<ol style="list-style-type: none">1. In Procedural Programming a program is created step by step instructional format and instructions are executed in order.2. Follow top down approach.3. Less secure because it does not have any proper way of data hiding.4. Does not provide code re-usability feature5. Doesn't provide ability to simulate real-world event much more effectively.6. Slow development	<ol style="list-style-type: none">1. In Object Oriented Programming a program is created in a way as real world works.2. Follow bottom to top approach.3. Secure because it have proper way of data hiding.4. Provide code reusability feature.5. Provide ability to simulate real-world event much more effectively.6. Fast Development.

OOPs Concepts

Object-Oriented Programming is a paradigm to design a program using classes and objects.

Object: Any entity that has state and behavior is known as an object

Example: pen, chair etc..

Class: Collection of objects is called class

Example: Pens is class and pen is object.

OOPS Principles

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

1. Abstraction

Hiding internal details and showing functionality is known as abstraction.

Example: phone call, working of car

Advantages of Abstraction

1. Only show essential details to end user.
2. Hide complexity.
3. Security.

2. Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation.

Example: capsule



Advantages of Encapsulation:

1. A read-only (immutable) or write-only class can be made.
2. Control over the data.
3. It helps in achieving high cohesion and low coupling in the code.

3. Polymorphism

One task is performed by different ways i.e. is known as polymorphism.

Example: God.

Types of polymorphism:

1. Static/compile time polymorphism (by method overloading).
2. Dynamic/run time polymorphism (by method overriding).

4. Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance.

Why inheritance is used?

1. Code re-usability.
2. Run-time polymorphism

Types of inheritance:

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. multiple inheritance
5. Hybrid inheritance

Evolution of Java

1. James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
2. Originally designed for small, embedded systems in electronic appliances like set-top boxes.
3. Firstly, it was called "Greentalk" by James Gosling.
4. Later, it was called Oak and was developed as a part of the Green project.

5. In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies

Why "Java" name

1. Java is an island of Indonesia where first coffee was produced
2. Java is just a name not an acronym
3. In 1995, Time magazine called Java one of the Ten Best Products of 1995.

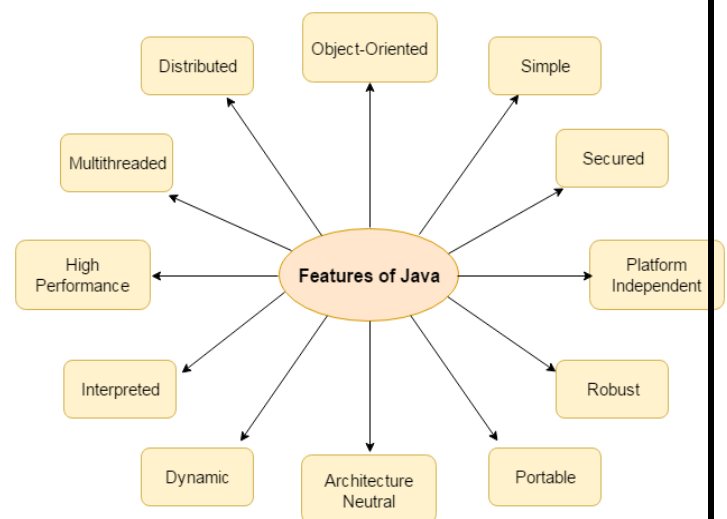
Java Version History

There are many java versions that has been released.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

Java Buzzwords

1. Simple
2. Platform independent
3. Architectural Neutral
4. Portable
5. Multithreaded
6. Distributed
7. Robust
8. Dynamic
9. Secured
10. High Performance
11. Interpreted
12. Object Oriented

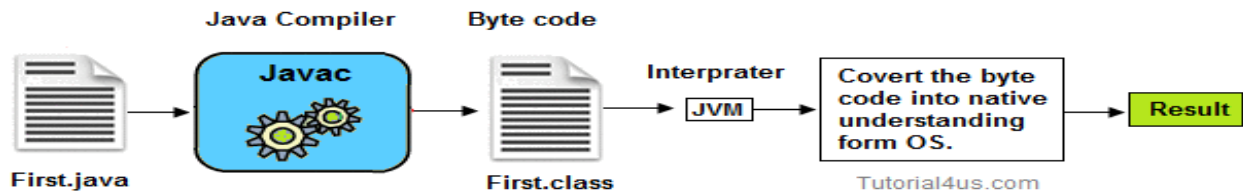


1. Simple

It is simple because of the following factors:

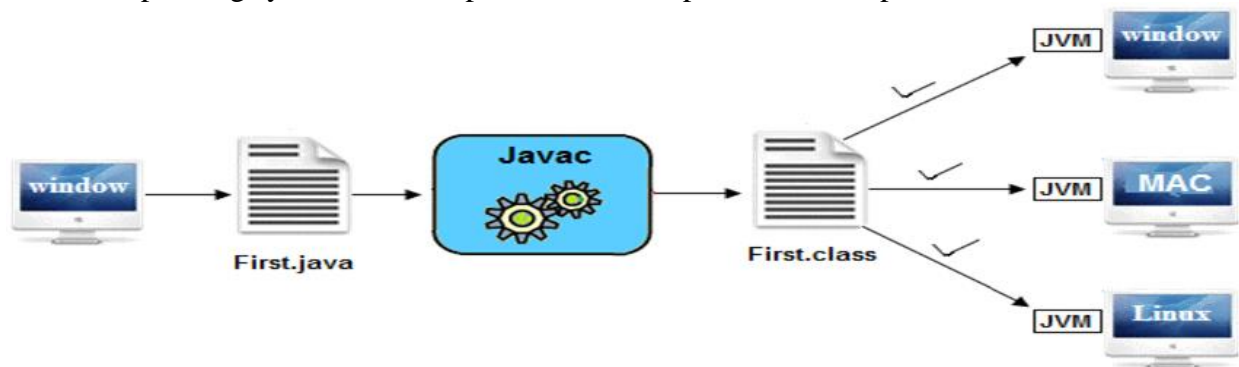
1. It is **free from pointer** due to this execution time of application is improved. [Whenever we write a Java program without pointers then internally it is converted into the equivalent pointer program].
2. It has **Rich set of API** (application protocol interface).
3. It has **Garbage Collector** which is always used to collect un-Referenced (unused) Memory location for improving performance of a Java program.

4. It contains user friendly syntax for developing any applications.



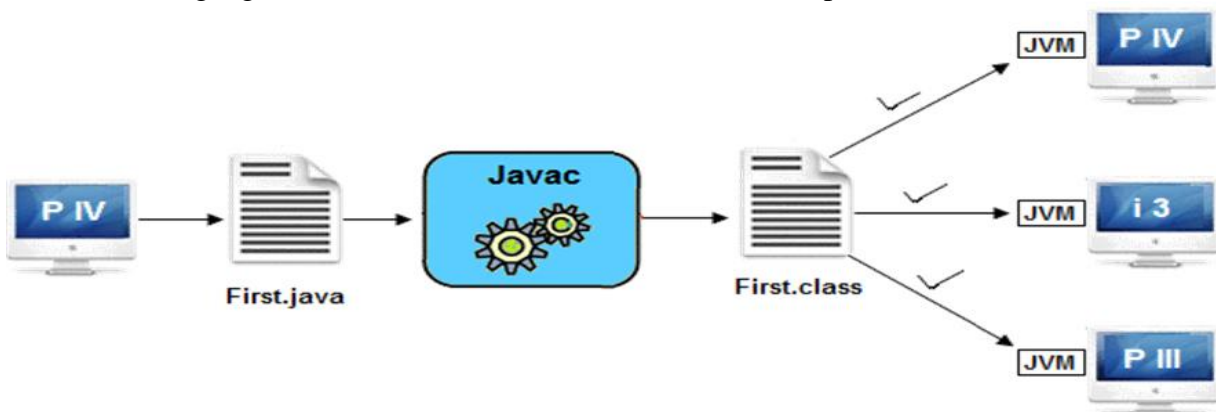
2. Platform independent

A program or technology is said to be platform independent if and only if which can run on all available operating systems with respect to its development and compilation.



3. Architectural Neutral

1. Architecture represents processor.
2. A Language or Technology is said to be Architectural neutral if and only if which can run on any available processors in the real world without considering their development and compilation.
3. The languages like C, CPP are treated as architectural dependent



4. Portable

1. If any language supports platform independent and architectural neutral feature is known as portable.
2. The languages like C, CPP, Pascal are treated as non-portable language.

3. Java is a portable language.

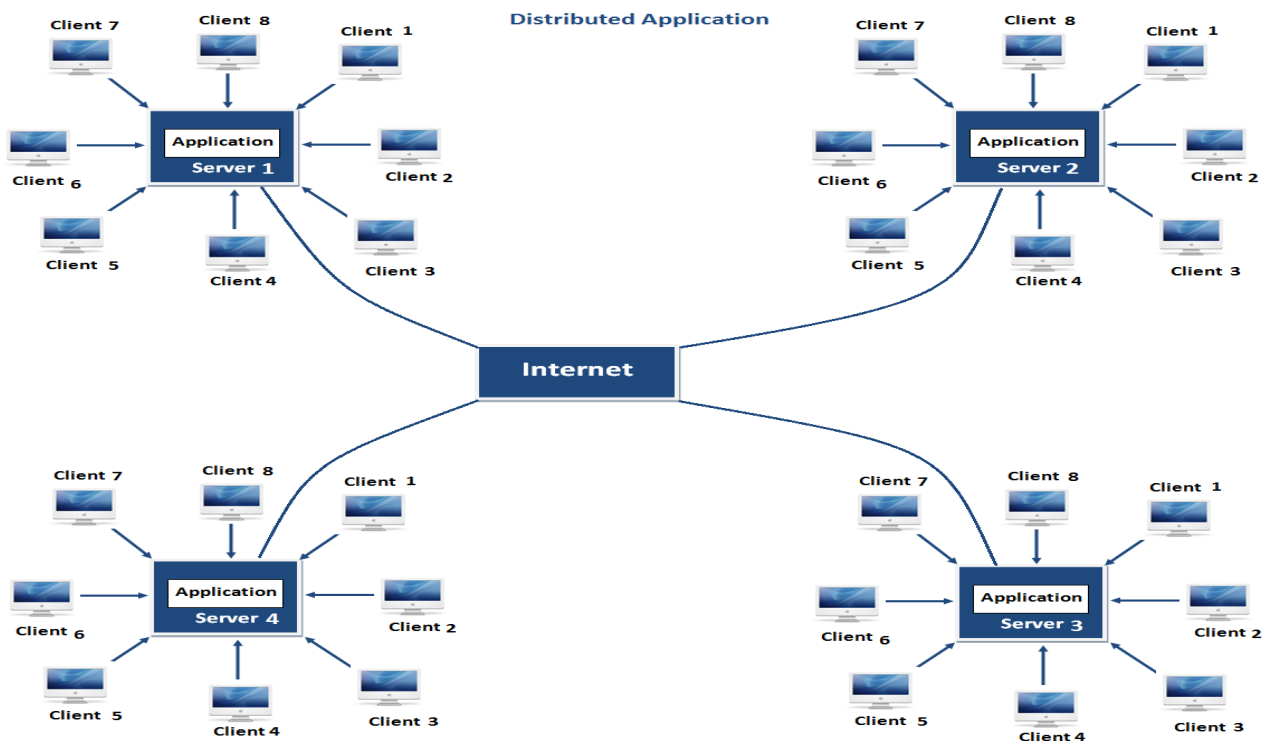
Portability = platform independent + architecture

5. Multithreaded

1. A flow of control is known as a thread.
2. When any Language executes multiple thread at a time that language is known as multithreaded.
3. Java is multithreaded.
4. Using this we can perform multiple tasks at time.

6. Distributed

1. Using Java we can create distributed applications.
2. RMI and EJB are used for creating distributed applications.
3. In distributed application multiple client systems depends on multiple server systems so that even problem occurred in one server will never be reflected on any client system.



7. Robust

1. Robust simply means strong.
2. It is robust or strong Programming Language because of its capability to handle Run-time Error, automatic garbage collection, the lack of pointer concept, Exception Handling. All these points make It robust Language.

8. Dynamic

1. It supports Dynamic memory allocation due to this memory wastage is reduce and improve performance of the application.
2. The process of allocating the memory space to the input of the program at a run-time is known as dynamic memory allocation

9. Secure

1. It is a more secure language compared to other languages.
2. In this language, all code is covered in to byte code after compilation which is not readable by human.

10. High performance

1. Java is an interpreted language, so it will never be as fast as a compiled language like C or C++.
2. But, Java enables high performance with the use of just-in-time compiler.

11. Interpreted

- It is one of the highly interpreted programming languages.

12. Object Oriented

1. In java everything is Object which has some data and behavior. Java can be easily extended as it is based on Object Model.
2. It supports OOP's concepts because of this it is most secure language.

Environment setup

Requirements to write Java Applications

1. For executing any java program, you need to install the JDK if you don't have installed it.
2. Set path of the jdk/bin directory.
3. create the java program
4. compile and run the java program

Path

- Path variable is set for providing path for all Java tools like java, javac, javap, javah, jar, appletviewer. In Java to run any program we use java tool and for compile Java code use javac tool. All these tools are available in bin folder so we set path upto bin folder.

classpath

- classpath variable is set for providing path of all Java classes which is used in our application. All classes are available in lib/rt.jar so we set classpath upto lib/rt.jar.

Java Program Structure

Java Program Structure for developing java application.

package details

```
class className
{
    Data Members
    User_defined Methods

    public static void main(String[] args)
    {
        Block of Staments
    }
}
```

Implementing a Java Program

Creating hello java example

```
import java.io;
class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello Java");
        System.out.println("My First Java Program");
    }
}
```

To compile: javac Hello.java

To execute: java Hello

Understanding Hello java program

1. **class** keyword is used to declare a class in java.
2. **public** keyword is an access modifier which represents visibility, it means it is visible to all.
3. **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
4. **void** is the return type of the method, it means it doesn't return any value.
5. **main** represents startup of the program.
6. **String[] args** is used for command line argument.
7. **System.out.println()** is used to print statements.

Valid java main method signature

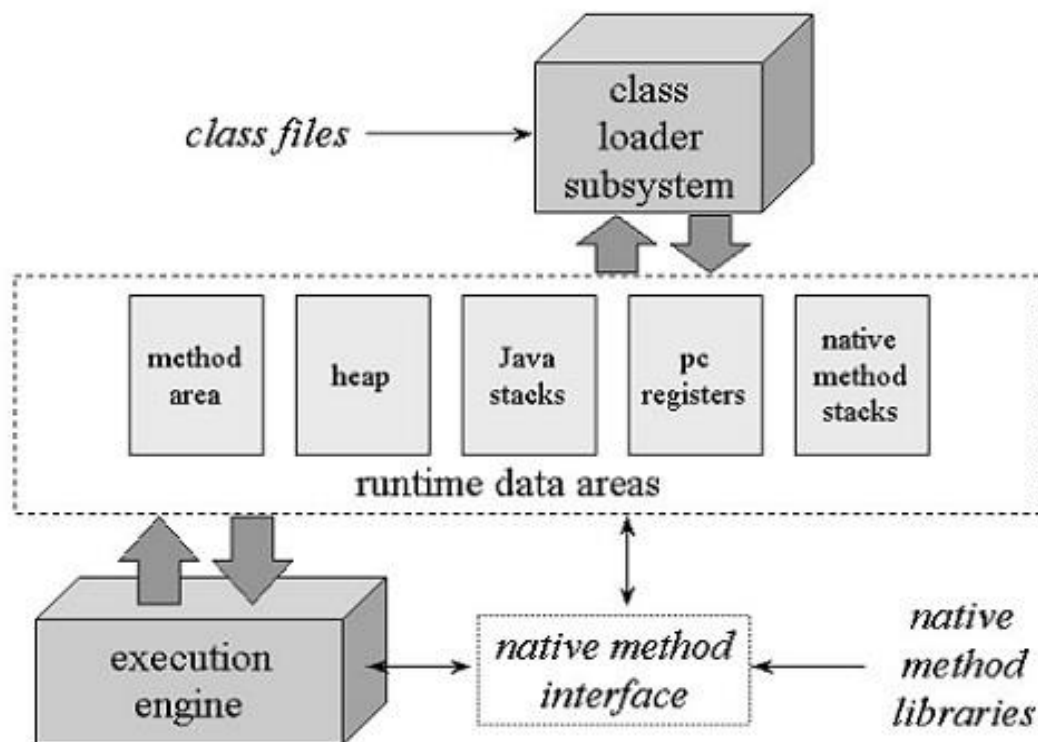
1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String args[])
4. public static void main(String... args)
5. static public void main(String[] args)
6. public static final void main(String[] args)
7. final public static void main(String[] args)

Invalid java main method signature

1. public void main(String[] args)
2. static void main(String[] args)
3. public void static main(String[] args)
4. abstract public static void main(String[] args)

JVM Architecture

JVM is a virtual machine or a program that provides run-time environment in which java byte code can be executed. JVMs are available for many hardware and software platforms. The use of the same byte code for all JVMs on all platforms makes java platform independent.



JVM Diagram

Class loader subsystem

Class loader subsystem will load the **.class** file into java stack and later sufficient memory will be allocated for all the properties of the java program into following five memory locations

- Heap area
- Method area
- Java stack
- PC register
- Native stack

Heap area:

In which object references will be stored.

Method area

In which static variables non-static and static method will be stored.

Java Stack

In which all the non-static variable of class will be stored and whose address referred by object reference.

Pc Register

Which holds the address of next executable instruction that means that use the priority for the method in the execution process?

Native Stack

Native stack holds the instruction of native code (other than java code) native stack depends on native library. Native interface will access interface between native stack and native library.

Execution Engine

It is a part JVM that uses Virtual processor (for execution).which contain interpreter (for reading instructions) and JIT (Just in time) compiler (for performance improvement) for executing the instructions.

How JVM is created

When JRE installed on your machine, you got all required code to create JVM. JVM is created when you run a java program.

Lifetime of JVM

When an application starts, a runtime instance is created.

When application ends, runtime environment destroyed.

If “n” no. of applications starts on one machine then “n” no. of runtime instances are created and every application run on its own JVM instance.

Main task of JVM

1. Search and locate the required files.
2. Convert byte code into executable code.
3. Allocate the memory into ram.
4. Execute the code.
5. Delete the executable code.

Difference between JVM, JRE and JDK

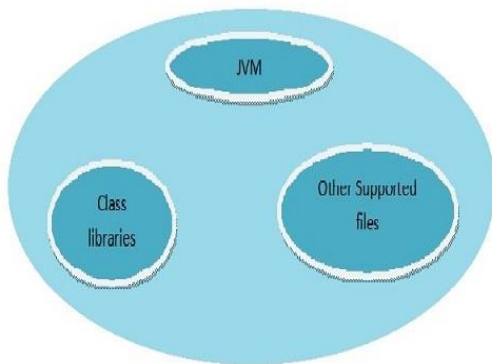
JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is a software. It is a specification that provides runtime environment in which java byte code can be executed. It is not physically exists.



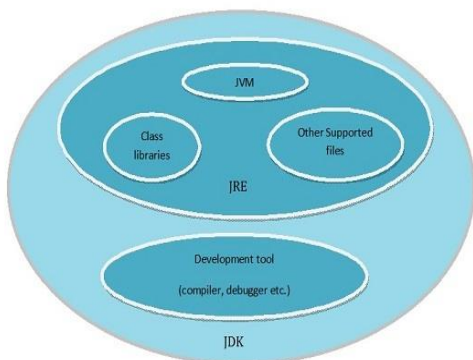
JRE (Java Run Time Environment)

JVM + java runtime libraries + java package classes (e.g. util, lang etc). JRE provides class libraries and other supporting files with JVM.



JDK (Java Development Kit)

JRE+ development tool (compiler, debugger etc.).JDK contains tools to develop the application and JRE to execute the application.

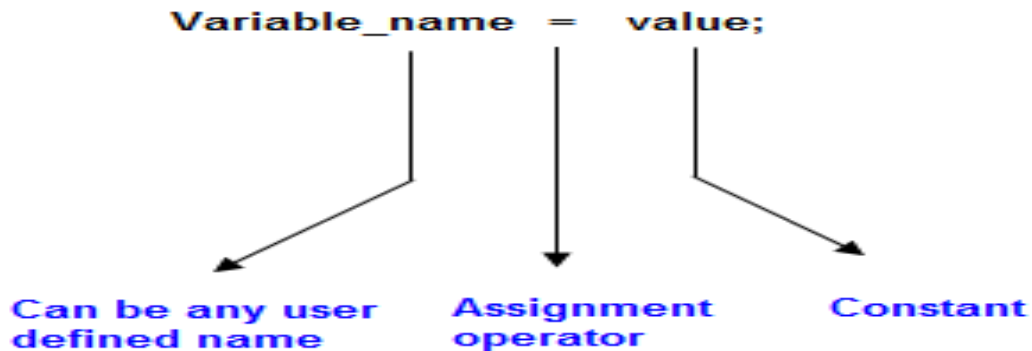


Variables

Variable is an identifier which can be used to identify input data in a program.

Syntax

Variable_name = value;



Rules to write a Variable Name

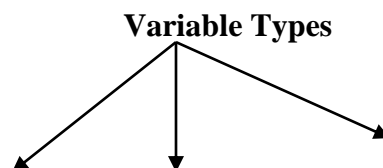
1. Java variable names are case sensitive. The variable name money is not the same as Money or MONEY.
2. Java variable names must start with a letter, or the \$ or _ character.
3. After the first character in a Java variable name, the name can also contain numbers.
4. No space are allowed in the variable declarations.
5. Except underscore (_) no special symbol are allowed in the middle of variable declaration
6. Variable name always should exist in the left hand side of assignment operators.
7. Maximum length of variable is 64 characters.
8. Variable names cannot be equal to reserved key words in Java. For instance, the words int or for are reserved words in Java. Therefore you cannot name your variables int or for.

Java variable naming conventions:

These conventions are not necessary to follow. But it is a good practice to write a java program.

1. Variable names are written in lowercase. Example: variable or apple.
2. If variable names consist of multiple words, each word after the first word has its first letter written in uppercase. Example: variableName or bigApple.
3. Even though it is allowed, you do not normally start a Java variable name with \$ or _ .
4. Static final fields (constants) are named in all uppercase, typically using an _ to separate the words in the name. Example: EXCHANGE_RATE or COEFFICIENT.

Variable Types



Local Instance Static

Local variable

A variable which is declared inside the method is called local variable.

Instance variable

A variable which is declared inside the class but outside the method, is called instance variable .
It is not declared as static.

Static variable

A variable that is declared within the class with static keyword but outside of method, constructor, or block is known as Static/class variable.

Static variables are accessed by **ClassName.VariableName**

Example to understand the types of variables

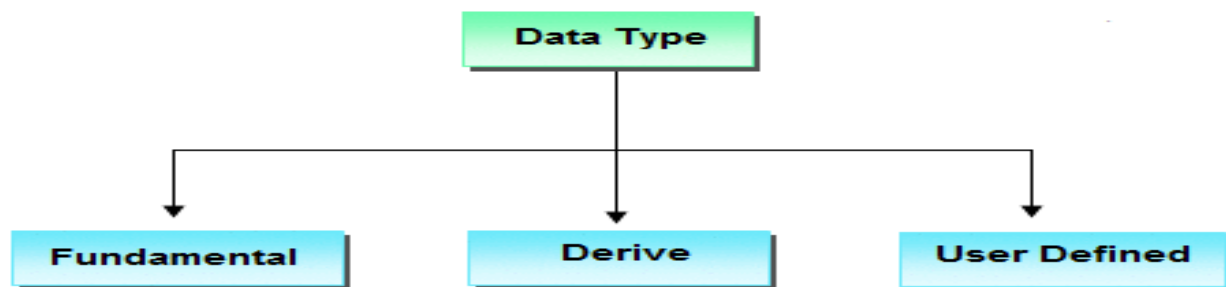
```
class A
{
int data=50;//instance variable
static int m=100;//static variable
void method()
{
int n=90;//local variable
}
} //end of class
```

Data Types

Data type is a special keyword used to allocate sufficient memory space for the data.

Categories of data types

1. Fundamental or primitive data types
2. Derived data types
3. User defined data types



1. Primitive data types

Primitive data types are those whose variables allows us to store only one value but they never allows us to store multiple values of same type.

Example

```
int a; // valid
```

```
a=10; // valid
```

```
a=10, 20, 30; // invalid
```

We have eight Primitive data type which are organized in four groups.

1. Integer category data types
2. Character category data types
3. Float category data types
4. Boolean category data types

1. Integer category data types

Data Type	Size	Range	Default Value
Byte	1	+ 127 to -128	0
Short	2	+ 32767 to -32768	0
Int	4	-2,147,483,648 to 2,147,483, 647	0
Long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L

2. Character category data types

Data Type	Size(Byte)	Range	Default Value
Char	2	232767 to -232768	'\u0000'

This data type takes two byte since it follows Unicode character set.

3. Float category data types

Data Type	Size(Byte)	Range	Default Value
Float	4	+2147483647 to -2147483648	0.0f
Double	8	+ 9.223*1018	0.0d

4. Boolean category data types

Data Type	Size	Default Value
boolean	1 bit	false

2. Derived data types

Derived data types are those whose variables allow us to store multiple values of same type. But they never allow to store multiple values of different types.

Example

```
int a[] = { 10,20,30}; // valid
int b[] = { 100, 'A', "ABC"}; // invalid
```

3. User defined data types

User defined data type related variables allow us to store multiple values either of same type or different type or both.

This is a data type whose variable can hold more than one value of dissimilar type.

Example

```
Student s = new Student();
```

Type Conversion and Casting

Assigning value of one type to variable of another type is known as conversion.

In java we have two types of conversions.

1. Implicit type conversion.
2. Explicit type conversion.

1. implicit type conversion

Java automatically converts from one type to another if it satisfies the following conditions.

- 1 Both types are compatible with each other.
- 2 The size of destination type is more than the source type.

This is also known as “**Widening Conversion**”.

Example

```
int i = 3;           int i=5;//size of int is 32
double f;            long d;//size of long is 64
f = i;               d=i;
```


Example program

```
class TypeConversion
{
    public static void main(String arg[])
    {
        byte b = 50;
        short s = 4125;
        int i = 800000;
        long l = 107343L;

        s = b;
        i = s;
        l = i;

        float f = 25.0f;
        double d = 327.98;

        f = i;
        d = f;

        System.out.println("b = " + b );
        System.out.println("s = " + s );
        System.out.println("i = " + i );
        System.out.println("l = " + l );
        System.out.println("d = " + d );
        System.out.println("f = " + f );

    }
}
```

2. Explicit type conversion

If we want to convert two types which are incompatible or if the size of destination type is less than the size of source type, then we must do the conversion explicitly.

This process is also called as “**type casting**”.

This is also called as Narrowing conversion .

Explicit type cast is requires to Narrowing conversion to inform the compiler that you are aware of the possible loss of precision.

Syntax

```
newValue = (typecast)value;
```

Example

```
double f = 3.5;
int i;
i = (int)f; // it cast double value 3.5 to int 3.
```

Example program

```
class DatatypeCasting
{
    public static void main(String arg[])
    {
        byte b;
        int i = 81;
        double d = 323.142;
        float f = 72.38f;
        char c = 'A';

        c = (char) i;
        System.out.println("i = " + i + " c = " + c);

        i = (int) d;
        System.out.println("d = " + d + " i = " + i);

        i = (int) f;
        System.out.println("f = " + f + " i = " + i);

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

    }
}
```

Operators

Operator is a symbol that is used to perform operations.

Types of operators in java

1. Arithmetic operators
2. Relation operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Ternary operator
7. Unary Operator

Arithmetic operators

Arithmetic operators are used in mathematical expression.

operator	description
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denominator
%	remainder of division

Relation operators

operator	description
==	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Logical operators

operator	description
&&	Logical AND
	Logical OR
!	Logical NOT

Bitwise operators

operator	description
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift
>>>	unsigned right shift(zero-fills from the left)

Assignment Operators

operator	description	example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
=	multiply left operand with the right operand and assign the result to left operand	a=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

Ternary operator

It is also known as Conditional operator and used to evaluate Boolean expression.

expr1 ? expr2 : expr3

If expr1 Condition is true? Then value expr2 : Otherwise value expr3

Unary Operators

operator	description
++	Increment operator increases integer value by one
--	Decrement operator decreases integer value by one

Control Statements

Control statements are the statements which alter the flow of execution and provide better control to the programmer on the flow of execution.

Types of control statements

1. selection control statements.
2. iteration control statements.
3. jump control statements.

1. Selection control statements

Selection statements can be divided into the following categories.

1. The if statements
2. The if-else statements
3. The if-else-if statements
4. The switch statements

1. If statement

if statement performs a task depending on whether a condition is true or false.

Syntax:

```
if (condition)
    statement1;
```

2. The if-else statements.

if the specified condition in the if statement is false, then the statement after the else keyword will execute.

Syntax:

```
if (condition)
    statement1;
else
    statement2;
```

3. The if-else-if statements

This statement following the else keyword can be another if or if-else statement.

Syntax

```
if(condition)
    statements;
```

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

4. The Switch Statements

When there are several options and we have to choose only one option from the available ones, we can use switch statement.

Syntax:

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

2. Iteration Statements

Repeating the same code fragment several times until a specified condition is satisfied is called iteration.

the following loop for iteration statements

1. The while loop
2. The for loop
3. The do-while loop
4. The for each loop

1. The while loop

It continually executes a statement while a condition is true. The condition must return a Boolean value.

Syntax:

```
while (condition)
{
statements;
}
```

2. The do-while loop

The only difference between a while and a do-while loop is that do-while evaluates its expression at the bottom of the loop instead of the top. The do-while loop executes at least one time then it will check the expression prior to the next iteration.

Syntax

```
do
{
    //Statements
}
while ( condition);
```

3. The for loop

A for loop executes a statement as long as the Boolean condition evaluates to true. A for loop is a combination of the three elements initialization statement, Boolean expression and increment or decrement statement.

Syntax:

```
for(<initialization>;<condition>;<increment or decrement statement>)
{
//block of code
}
```

4. The For each loop

This was introduced in Java 5. This loop is basically used to traverse the array or collection elements.

Syntax

```
for(data_type variable : array | collection)
{
}
```

3. Jump Statements

Jump statements are used to unconditionally transfer the program control to another part of the program.

Java provides the following jump statements

1. break statement
2. continue statement
3. return statement

1. Break Statement

The break statement immediately quits the current iteration and goes to the first statement following the loop.

2. Continue Statement

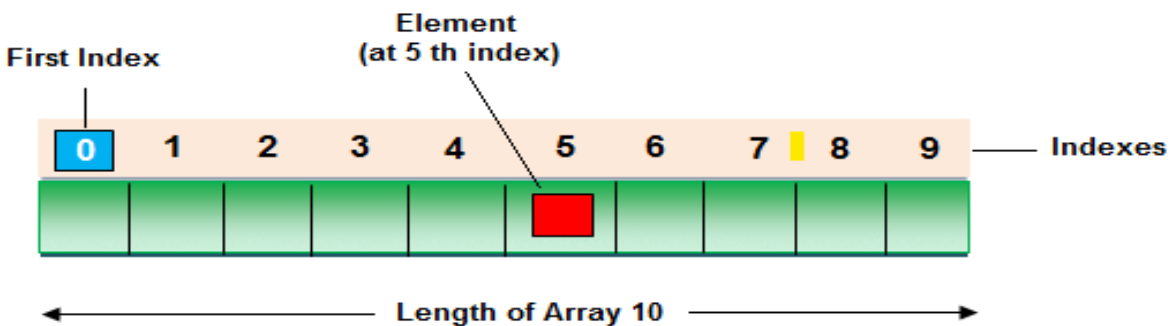
The continue statement is used when you want to continue running the loop with the next iteration and want to skip the rest of the statements of the body for the current iteration.

3. Return Statement

The return statement is used to immediately quit the current method and return to the calling method. It is mandatory to use a return statement for non-void methods to return a value.

Arrays

Array is a collection of similar type of data. It is fixed in size means that you can't increase the size of array at run time. It is a collection of homogeneous data elements. It stores the value on the basis of the index value.



Advantage of Array

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

Disadvantage of Array

1. **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime.

Types of Array

1. Single Dimensional Array
2. Multidimensional Array

Array Declaration

Single dimension array declaration

Syntax

1. `int[] a;`
2. `int a[];`
3. `int []a`

Multidimensional Array declaration

Syntax

1. `int[][] a;`
2. `int a[][];`
3. `int [][]a;`
4. `int[] a[];`
5. `int[] []a;`
6. `int []a[];`

Array creation

Every array in a Java is an object, Hence we can create array by using **new** keyword.

Syntax

```
int[] arr = new int[10]; // The size of array is 10.
```

or

```
int[] arr = {10,20,30,40,50};
```

Accessing array elements

Access the elements of array by using index value of an elements.

Syntax

```
arrayname[n-1];
```

every array type has its corresponding class .

Class arr

```
{
Public static void main(String[] args)
{
Int[] x=new int[3];
S.O.P(x.getClass().getName());// [I
}
}
```

Basic IO

Java I/O (Input and Output) is used to process the input and produce the output.

It uses the concept of streams to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

Stream

A stream is a sequence of data. In Java a stream is composed of bytes.

In java,3 streams are created for us automatically. All these streams are attached with console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and error** message to the console.

```
System.out.println("simple message");
```

```
System.err.println("error message");
```

Let's see the code to get **input** from console.

```
int i=System.in.read();//returns ASCII code of 1st character
```

```
System.out.println((char)i);//will print the character
```

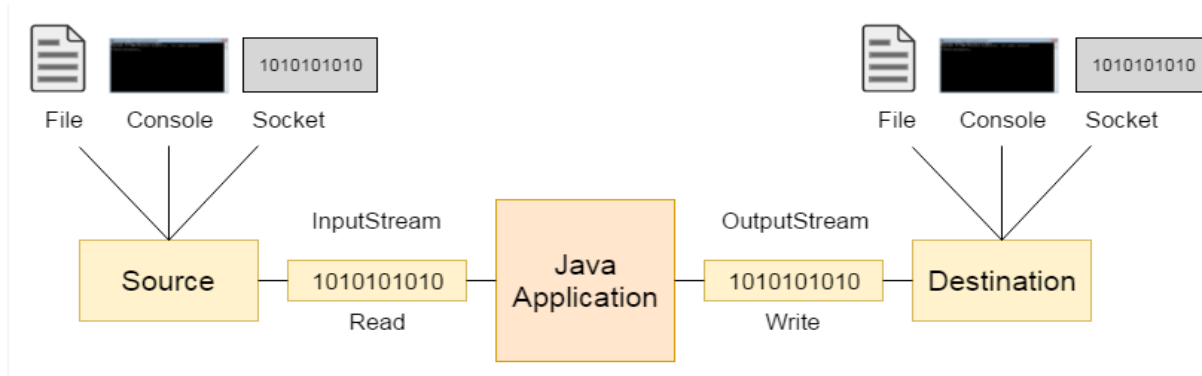
Output Stream vs Input Stream

OutputStream

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.



Reading Console Input

In Java, there are 3 ways to read input from a console.

1. `BufferedReader + InputStreamReader` (Classic)
2. `Scanner` (JDK 1.5)
3. `System.console` (JDK 1.6)

1. `BufferedReader + InputStreamReader`

`InputStreamReader` that can read data from the keyboard.

Syntax

```
InputStreamReader obj = new InputStreamReader (System.in);
```

Connect `InputStreamReader` to `BufferedReader`, which is another input type of stream to read data properly.

Syntax

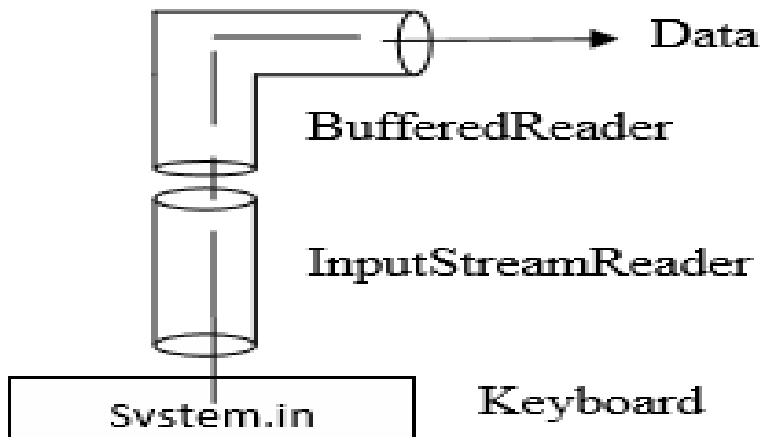
```
BufferedReader br = new BufferedReader (obj);
```

two steps can be combined and rewritten in a single statement

Syntax

```
BufferedReader br = new BufferedReader (new InputStreamReader(System.in));
```

Now, we can read the data coming from the keyboard using `read ()` and `readLine ()` methods available in `BufferedReader` class.



2. Scanner Class

In JDK 1.5, the developer starts to use `java.util.Scanner` to read system input.

Syntax

```
Scanner scanner = new Scanner(System.in);
```

3. System.console

In JDK 1.6, the developer starts to switch to the more simple and powerful `java.io.Console` class to read system input.

It provides methods to read texts and passwords.

syntax

```
Console c=System.console();
```

Writing Console Output

`PrintStream` is an output stream derived from `OutputStream`, it also implements the low-level method `write()`.

`write()` can be used to write to the console.

Syntax

The simplest form of `write()` defined by the `PrintStream` is

```
void write(int byteval)
```

Naming conventions

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

- class name should start with **uppercase letter** and be a **noun** e.g. String, Color, Button, System, Thread etc.
- interface name should start with **uppercase letter** and be an **adjective** e.g. Runnable, Remote, ActionListener etc.
- method name should start with **lowercase letter** and be a **verb** e.g. actionPerformed(), main(), print(), println() etc.
- variable name should start with **lowercase letter** e.g. firstName, orderNumber etc.
- package name should be in **lowercase letter** e.g. java, lang, sql, util etc.
- constants name should be in all **uppercase letters**. e.g. RED, YELLOW, MAX_PRIORITY etc.
-

Introducing Classes and Object

Class

Class is a blue print which is containing only list of variables and method and no memory is allocated for them. A class is a group of objects that has common properties.

A class contains

- Data Member
- Method
- Constructor
- Block
- Class and Interface

Object

Object is an instance of class.

An Object has three characteristics

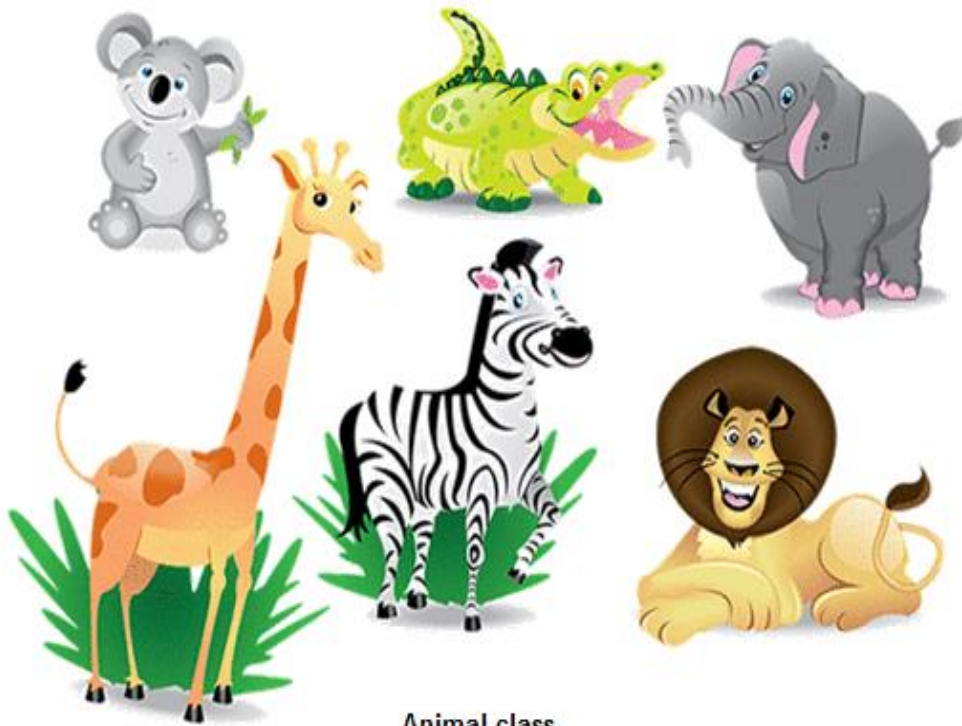
- State
- Behavior
- Identity

State: Represents data of an object.

Behavior: Represents the behavior (functionality) of an object such as deposit, withdraw etc.

Identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

Real life example of object and class



Animal class



Vehicle class

Difference between Class and Object

Class	Object
Class is a container which collection of variables and methods.	object is a instance of class
No memory is allocated at the time of declaration	Sufficient memory space will be allocated for all the variables of class at the time of declaration.
One class definition should exist only once in the program.	For one class multiple objects can be created.

Syntax to declare a Class

```
class Class_Name
{
    data member;
    method;
}
```

Syntax to create an Object

```
Class_name ObjectName=new Class_name();
```

Example

```
class Employee
{
    int eid; // data member (or instance variable)
    String ename; // data member (or instance variable)
    eid=101;
    ename="Vandana";
    public static void main(String args[])
    {
        Employee e=new Employee(); // Creating an object of class Employee
        System.out.println("Employee ID: "+e.eid);
    }
}
```

```
System.out.println("Name: "+e.ename);  
} }
```

Methods

A method is a collection of statements that are grouped together to perform an operation.
Method describes the behavior of an object

Creating Method

Syntax

```
modifier returnType nameOfMethod (Parameter List) {  
// method body  
}
```

Parameter Vs. Argument

Parameter is variable defined by a method that receives value when the method is called.
Parameter are always local to the method they don't have scope outside the method.
argument is a value that is passed to a method when it is called.

Example

```
public void sum( int x, int y )  
{  
    System.out.println(x+y);  
}  
public static void main( String[ ] args )  
{  
    Test b=new Test( );  
    b.sum( 10, 20 );  
}
```

The diagram consists of two red labels with arrows. The label 'parameter' is located at the top right, with an arrow pointing to the parameter 'int y' in the method signature 'sum(int x, int y)'. The label 'argument' is located at the bottom right, with an arrow pointing to the value '20' in the method call 'b.sum(10, 20);'.

Constructors

Constructor is a *special type of method* that is used to initialize the state of object. It provides the values to the data members at the time of object creation that is why it is known as constructor.

Characteristics of constructor

1. A constructor must have the same name as of its class.
2. It is invoked at the time of object creation and used to initialize the state of an object.
3. It does not have an explicit return type.

Types of constructor

1. Default or no-argument constructor.
2. Parameterized constructor.

Default or no-argument constructor

A constructor with no parameter is known as default or no-argument constructor.

If no constructor is defined in the class then compiler automatically creates a default constructor at the time of compilation.

Syntax

```
Class_name()  
{  
//optional block of code  
}
```

Parameterized constructor

A constructor with one or more arguments is known as parameterized constructor.

Parameterized constructor is used to provide values to the object properties.

By use of parameterized constructor different objects can be initialize with different states.

Syntax

```
class ClassName  
{ .....  
ClassName(list of parameters) //parameterized constructor  
{ .....  
}  
..... }
```

Important points Related to Parameterized Constructor

- Whenever we create an object using parameterized constructor, it must be define parameterized constructor otherwise we will get compile time error.
- Whenever we define the objects with respect to both parameterized constructor and default constructor, It must be define both the constructors.
- In any class maximum one default constructor but 'n' number of parameterized constructors.

Difference between Method and Constructor

	Method	Constructor
1	Method can be any user defined name	Constructor must be class name
2	Method should have return type	It should not have any return type (even void)
3	Method should be called explicitly either with object reference or class reference	It will be called automatically whenever object is created
4	Method is not provided by compiler in any case.	The java compiler provides a default constructor if we do not have any constructor.
5	It is used to show behavior of an object.	It is used to initialize the state of an object

Garbage Collection

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

Advantages of Garbage Collection

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

How can an object be unreferenced

There are many ways:

1. By nulling the reference
2. By assigning a reference to another
3. By anonymous object

By nulling a reference

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

By assigning a reference to another

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

By anonymous object

```
new Employee();
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing.

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

Important Points to Remember

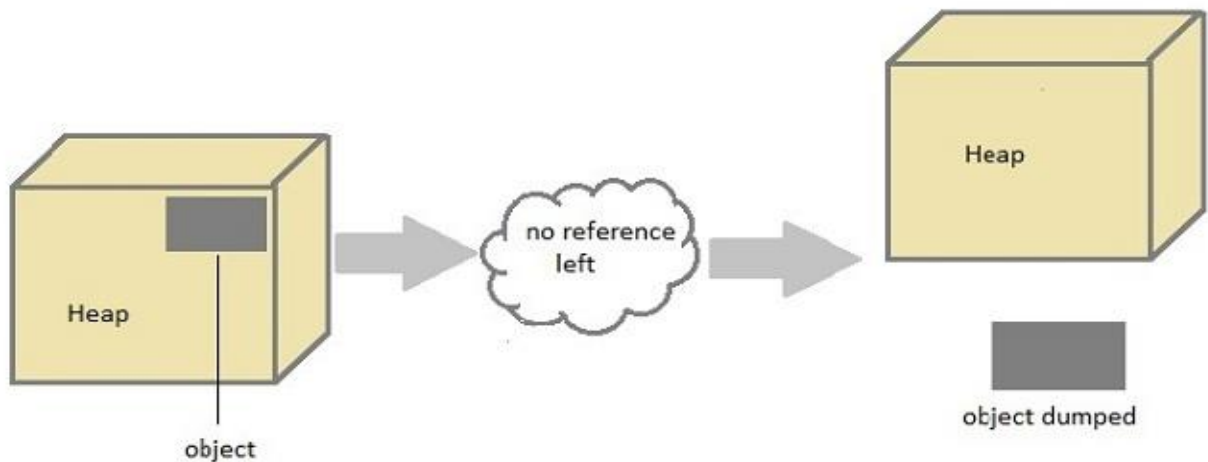
1. finalize() method is defined in **java.lang.Object** class, therefore it is available to all the classes.
2. finalize() method is declare as **protected** inside Object class.
3. finalize() method gets called only once by a Daemon thread named GC (Garbage Collector)thread

gc() method

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

Syntax

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



Overloading Methods

If two or more method in a class has same name but different parameters, it is known as method overloading.

Method overloading can be done by changing number of arguments or by changing the data type of arguments.

If two or more method have same name and same parameter list **but differs in return type are not** said to be overloaded method.

Different ways to overload the method

1. By changing number of arguments or parameters.
2. By changing the data type.
3. Sequence of Data type of parameters.

When an overloaded method is called the compiler looks for exact match.

Sometime when exact match is not found, java automatic type conversion plays a vital role

Constructor Overloading

Whenever same constructor is existing multiple times in the same class with different number of parameters or order of parameters or type of parameters is known as **Constructor overloading**. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Constructor overloading can be used to initialize same or different objects with different values.

Argument Passing

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Recursion

The idea of calling one function from another immediately suggests the possibility of a function calling *itself*. The function-call mechanism in Java supports this possibility, which is known as *recursion*.

Syntax:

```
returntype methodname()  
{  
//code to be executed  
methodname();//calling same method  
}
```

Final keyword

It is used to make a variable as a constant, Restrict method overriding, Restrict inheritance.
final keyword can be used in following way.

1. Final at variable level
2. Final at method level
3. Final at class level

Final at variable level

Final keyword is used to make a variable as a constant.

A variable declared with the final keyword cannot be modified by the program after initialization.

Example

```
public class Circle
{
    public static final double PI=3.14159;
    public static void main(String[] args)
    {
        System.out.println(PI);
    } }
```

Final at method level

It makes a method final, meaning that sub classes cannot override this method.

Example

```
class Employee
{
    final void disp()
    { System.out.println("Hello Good Morning");
    } }
class Developer extends Employee
{
    void disp()
    {
        System.out.println("How are you ?");
    } }
class FinalDemo
{
    public static void main(String args[])
    {
        Developer obj=new Developer();
        obj.disp();
    } }
```

Final at class level

It makes a class final, meaning that the class can not be inheriting by other classes.

example

```
final class Employee
{
    int salary=10000;
}
class Developer extends Employee
{ void show()
{
    System.out.println("Hello Good Morning");
} }
class FinalDemo
{
    public static void main(String args[])
    {
        Developer obj=new Developer();
        obj.show();
    } }
```

Static keyword

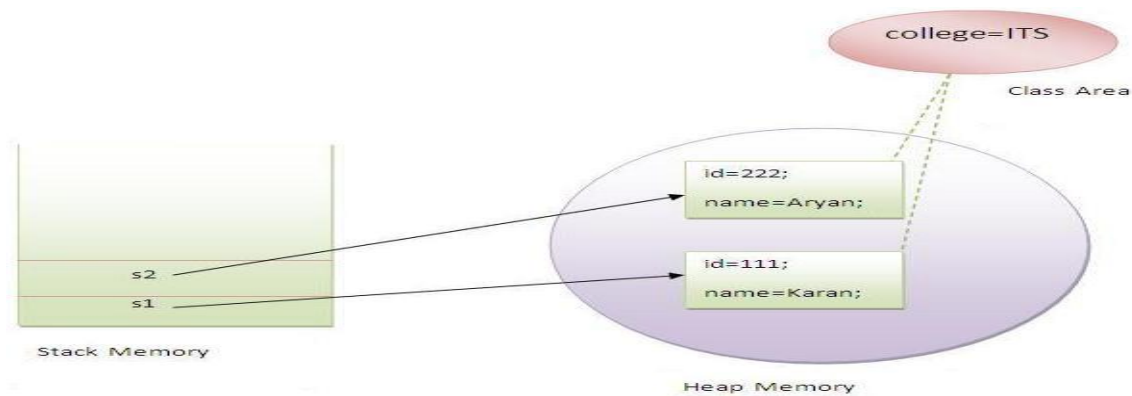
The **static keyword** is used for memory management mainly.

The static can be:

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

Static variable

- The static variable can be used to refer the common property of all objects 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.
- if any object changes the value of the static variable, it will retain its value.
- It makes your program **memory efficient**



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

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.

static block

Is used to initialize the static data member.

It is executed before main method at the time of class loading.

Example

```
class A2{
    static
    {
        System.out.println("static block is invoked");
    }
    public static void main(String args[])
    {
        System.out.println("Hello main");
    }
}
```

Command- Line Arguments

command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

Example

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

}

String

String is probably the most commonly used class in java library. String class is encapsulated under java.lang package. In java, every string that you create is actually an object of type String. One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be altered.

Immutable object

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.

Creating a String object

String can be created by using these ways

1) Using a String literal

String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

```
String str1 = "Hello";
```

2) Using new Keyword

```
String str3 = new String("Java");
```

String class function

Here is the list of methods supported by String class

S.No.	Method & Description
1	char charAt(int index) Returns the character at the specified index.

2	int compareTo(Object o) Compares this String to another Object.
3	int compareTo(String anotherString) Compares two strings lexicographically.
4	int compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences.
5	String concat(String str) Concatenates the specified string to the end of this string.
6	boolean contentEquals(StringBuffer sb) Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	static String copyValueOf(char[] data) Returns a String that represents the character sequence in the array specified.
8	static String copyValueOf(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
9	boolean endsWith(String suffix) Tests if this string ends with the specified suffix.
10	boolean equals(Object anObject) Compares this string to the specified object.
11	boolean equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations.
12	byte getBytes() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	byte[] getBytes(String charsetName) Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
15	int hashCode() Returns a hash code for this string.

16	int indexOf(int ch) Returns the index within this string of the first occurrence of the specified character.
17	int indexOf(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	int indexOf(String str) Returns the index within this string of the first occurrence of the specified substring.
19	int indexOf(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
20	String intern() Returns a canonical representation for the string object.
21	int lastIndexOf(int ch) Returns the index within this string of the last occurrence of the specified character.
22	int lastIndexOf(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	int lastIndexOf(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
24	int lastIndexOf(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
25	int length() Returns the length of this string.
26	boolean matches(String regex) Tells whether or not this string matches the given regular expression.
27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
28	boolean regionMatches(int toffset, String other, int ooffset, int len) Tests if two string regions are equal.

29	String replace(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
30	String replaceAll(String regex, String replacement) Replaces each substring of this string that matches the given regular expression with the given replacement.
31	String replaceFirst(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	String[] split(String regex) Splits this string around matches of the given regular expression.
33	String[] split(String regex, int limit) Splits this string around matches of the given regular expression.
34	boolean startsWith(String prefix) Tests if this string starts with the specified prefix.
35	boolean startsWith(String prefix, int toffset) Tests if this string starts with the specified prefix beginning a specified index.
36	CharSequence subSequence(int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence.
37	String substring(int beginIndex) Returns a new string that is a substring of this string.
38	String substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
39	char[] toCharArray() Converts this string to a new character array.
40	String toLowerCase() Converts all of the characters in this String to lower case using the rules of the default locale.
41	String toLowerCase(Locale locale) Converts all of the characters in this String to lower case using the rules of the given Locale.

42	String toString() This object (which is already a string!) is itself returned.
43	String toUpperCase() Converts all of the characters in this String to upper case using the rules of the default locale.
44	String toUpperCase(Locale locale) Converts all of the characters in this String to upper case using the rules of the given Locale.
45	String trim() Returns a copy of the string, with leading and trailing whitespace omitted.
46	static String valueOf(primitive data type x) Returns the string representation of the passed data type argument.

StringBuffer class

StringBuffer class is used to create a mutable string object i.e its state can be changed after it is created. It represents growable and writable character sequence. As we know that String objects are immutable, so if we do a lot of changes with String objects, we will end up with a lot of memory leak. So StringBuffer class is used when we have to make lot of modifications to our string. It is also thread safe i.e multiple threads cannot access it simultaneously.

StringBuffer defines 4 constructors. They are,

1. StringBuffer ()
2. StringBuffer (int size)
3. StringBuffer (String str)
4. StringBuffer (charSequence []ch)

Important methods of StringBuffer class

S.N.	Method & Description
1	StringBuffer append(boolean b) This method appends the string representation of the boolean argument to the sequence
2	StringBuffer append(char c) This method appends the string representation of the char argument to this sequence.

3	StringBuffer append(char[] str) This method appends the string representation of the char array argument to this sequence.
4	StringBuffer append(char[] str, int offset, int len) This method appends the string representation of a subarray of the char array argument to this sequence.
5	StringBuffer append(CharSequence s) This method appends the specified CharSequence to this sequence.
6	StringBuffer append(CharSequence s, int start, int end) This method appends a subsequence of the specified CharSequence to this sequence.
7	StringBuffer append(double d) This method appends the string representation of the double argument to this sequence.
8	StringBuffer append(float f) This method appends the string representation of the float argument to this sequence.
9	StringBuffer append(int i) This method appends the string representation of the int argument to this sequence.
10	StringBuffer append(long lng) This method appends the string representation of the long argument to this sequence.
11	StringBuffer append(Object obj) This method appends the string representation of the Object argument.
12	StringBuffer append(String str) This method appends the specified string to this character sequence.
13	StringBuffer append(StringBuffer sb) This method appends the specified StringBuffer to this sequence.
14	StringBuffer appendCodePoint(int codePoint) This method appends the string representation of the codePoint argument to this sequence.
15	int capacity() This method returns the current capacity.
16	char charAt(int index) This method returns the char value in this sequence at the specified index.
17	int codePointAt(int index) This method returns the character (Unicode code point) at the specified index

18	int codePointBefore(int index) This method returns the character (Unicode code point) before the specified index
19	int codePointCount(int beginIndex, int endIndex) This method returns the number of Unicode code points in the specified text range of this sequence
20	StringBuffer delete(int start, int end) This method removes the characters in a substring of this sequence.
21	StringBuffer deleteCharAt(int index) This method removes the char at the specified position in this sequence
22	void ensureCapacity(int minimumCapacity) This method ensures that the capacity is at least equal to the specified minimum.
23	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) This method characters are copied from this sequence into the destination character array dst.
24	int indexOf(String str) This method returns the index within this string of the first occurrence of the specified substring.
25	int indexOf(String str, int fromIndex) This method returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
26	StringBuffer insert(int offset, boolean b) This method inserts the string representation of the boolean argument into this sequence.
27	StringBuffer insert(int offset, char c) This method inserts the string representation of the char argument into this sequence.
28	StringBuffer insert(int offset, char[] str) This method inserts the string representation of the char array argument into this sequence.
29	StringBuffer insert(int index, char[] str, int offset, int len) This method inserts the string representation of a subarray of the str array argument into this sequence.
30	StringBuffer insert(int dstOffset, CharSequence s) This method inserts the specified CharSequence into this sequence.

31	StringBuffer insert(int dstOffset, CharSequence s, int start, int end) This method inserts a subsequence of the specified CharSequence into this sequence.
32	StringBuffer insert(int offset, double d) This method inserts the string representation of the double argument into this sequence.
33	StringBuffer insert(int offset, float f) This method inserts the string representation of the float argument into this sequence.
34	StringBuffer insert(int offset, int i) This method inserts the string representation of the second int argument into this sequence.
35	StringBuffer insert(int offset, long l) This method inserts the string representation of the long argument into this sequence.
36	StringBuffer insert(int offset, Object obj) This method inserts the string representation of the Object argument into this character sequence.
37	StringBuffer insert(int offset, String str) This method inserts the string into this character sequence.
38	int lastIndexOf(String str) This method returns the index within this string of the rightmost occurrence of the specified substring.
39	int lastIndexOf(String str, int fromIndex) This method returns the index within this string of the last occurrence of the specified substring.
40	int length() This method returns the length (character count).
41	int offsetByCodePoints(int index, int codePointOffset) This method returns the index within this sequence that is offset from the given index by codePointOffset code points.
42	StringBuffer replace(int start, int end, String str) This method replaces the characters in a substring of this sequence with characters in the specified String.
43	StringBuffer reverse() This method causes this character sequence to be replaced by the reverse of the sequence.

44	void setCharAt(int index, char ch) The character at the specified index is set to ch.
45	void setLength(int newLength) This method sets the length of the character sequence.
46	CharSequence subSequence(int start, int end) This method returns a new character sequence that is a subsequence of this sequence.
47	String substring(int start) This method returns a new String that contains a subsequence of characters currently contained in this character sequence
48	String substring(int start, int end) This method returns a new String that contains a subsequence of characters currently contained in this sequence.
49	String toString() This method returns a string representing the data in this sequence.
50	void trimToSize() This method attempts to reduce storage used for the character sequence.

Inner classes

Definition: A class which is declared inside another class is called inner class.

Situation where we create Inner classes:

Without existing one type of object there is no chance of existing another type of object, then we should go for inner classes.

Example1

```

Class University
{
    Class dept
    {
    }
}

```

Example2

```

class Car
{
    class engine
    {
    }
}

```

Note:

Without existing Outer class object there is no chance of Inner class object.

Inner classes Types

1. Normal or Regular inner classes
2. Method local inner classes
3. Anonymous inner classes
4. Static Nested classes

1. Normal or Regular inner classes:

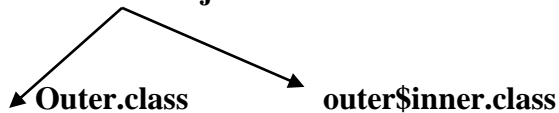
Class outer

```
{
    Class inner
    {
    }
}
```

If we are declaring any named classes directly inside a class without static modifier such type of inner class is called normal or regular inner class.

Compile the file

Javac outer.java



Running the file

Java outer

Err: no such method main

Java outer\$inner

Err: no such method main

Example

Class outer

```
{
    Class inner
    {
    }
    Public static void main(String []args)
    {
        System.out.println("outer class main ");
    }
}
```



```
    }  
}
```

Java outer

Output: outer class main

Java outer\$inner

Err: no such method main

Example 2

Class outer

```
{  
    Class inner  
    {  
        Public static void main(String []args)  
        {  
            System.out.println("outer class main ");  
        }  
    }  
}
```

For this we get compile time error. (Inner classes can't have static declarations)

Accessing inner class code from static area of outer class

Example

Class outer

```
{  
    Class inner  
    {  
        Public void m1()  
        {  
            System.out.println("inner class method");  
        }  
    }  
    Public static void main(String args[])  
    {  
        Outer ob=new outer ();  
        Outer.inner i=ob.new inner();  
        }  
        i.m1();  
    }  
}
```

Accessing inner class code from instance area of outer class

Example

Class outer

```
{
    Class inner
    {
        Public void m1()
        {
            System.out.println("inner class method");
        }
    }
    Public void m2()
    {
        Inner i=new inner();
        i.m1();
    }
    Public static void main(String args[])
    {
        Outer ob=new outer ();
        ob.m2();
    }
}
```

Accessing inner class code from outside of outer class**Example**

Class outer

```
{
    Class inner
    {
        Public void m1()
        {
            System.out.println("inner class method");
        }
    }
}
Class test
{
    Public static void main(String args[])
    {
        Outer ob=new outer();
        Outer.inner i=ob.new inner();
        i.m1();
    }
}
```

From normal or regular inner classes we can access static and non static members directly.

Example

Class outer

```
{
    int x=10;
    Static int y=20;
    Class inner
    {
        Public void m1()
        {
            System.out.println(x);
            System.out.println(y);
        }
    }
    Public static void main(String args[])
    {
        Outer ob=new outer();
        Outer.inner i=ob.new inner();
        i.m1();
    }
}
```

The only applicable modifiers for outer classes are

1. Public
2. Default
3. Final
4. Abstract
5. Strictfp

But for inner classes applicable modifiers are

1. Public
2. Default
3. Final
4. Abstract
5. Strictfp
6. Private
7. Protected
8. static

Nesting of Inner classes is possible

Example

Class A

```
{
    Class B
    {
        Class C
```

```

        {
            Public void m1()
            {
                System.out.println("inner most inner class method");
            }
        }
    }
}
Class test
{
    Public static void main(String[] args)
    {
        A a=new A();
        A.B b=a.new B();
        A.B.C c=b.new C();
        c.m1();
    }
}

```

2. Method Local Inner classes

Sometimes we can declare a class inside a method such types of inner classes is called method local inner classes.

The main purpose of method local inner classes is to define method specific repeatedly required functionality.

Method local inner classes are best suitable to meet nested method requirements.

We can access method local inner classes only within the method where we declare outside of the method we can't access because of its less scope method local inner classes are most rarely used type of inner classes.

Example

```

Class test
{
    Public void m1()
    {
        Class inner
        {
            Public void sum(int x,int y)
            {
                System.out.println("the sum"+(x+y));
            }
        }
        Inner i=new inner();
        i.sum(10,20);
        ;;;;
    }
}

```

```

        i.sum(100,200);
        ;;;
        i.sum(1000,2000);
    }
    Public static void main(String args[])
    {
        Test t=new Test();
        t.m1();
    }
}

```

We can declare method local inner class inside both instance and static methods.

If we declare inner class inside instance method then from that method local inner class we can access both static and non static members of outer class directly.

If we declare inner class inside a static method then we can access only static member of outer class directly from that method local inner class.

Accessing Instance and Static variables


Example

Class Test

```

{
    Int x=10;
    Static int y=100;
    Public void m1()
    {
        Class inner
        {
            Public void m2()
            {
                System.out.println(x);// we will get compile time error if method is static
                System.out.println(y);
            }
        }
        Inner i=new inner();
        i.m2();
    }
    Public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
    }
}

```



Accessing Local Variable

Example:

```
Class Test
{
    Public void m1()
    {
Final ← Int x=10;
        Class Inner
        {
            Public void m2()
            {
                System.out.println(x);
            }
            Inner i=new Ineer();
            i.m2();
        }
    Public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
    }
}
```

Output: Compile time error

From Method local inner class we can't access local variable of the method in which we declare inner class. If the local variable declared as **Final** then we can access.

Note: the applicable modifiers for method local inner classes are **final**, **abstract**, **strictfp**

3. Anonymous Inner classes

Sometimes we can declare inner classes without name such type of inner classes are called anonymous inner classes.

The main purpose of anonymous inner classes is just for instant use (one time usage).

Based on declaration and behavior

Types of anonymous inner classes

1. Anonymous inner class that extends a class
2. Anonymous inner class that implements an interface
3. Anonymous inner class that defined inside arguments

1. Anonymous inner class that extends a class

Class PopCon

{

Public void taste()

{

System.out.println("salty");

}

}

Class Test

{

Public static void main (String[] args)

{

PopCon p=new PopCon()

{

Public void taste()

{

System.out.println("spicy");

}

};

p.taste();

PopCon p1=new PopCon();

P1.taste();

PopCon p2=new PopCon()

{

Public void taste()

{

System.out.println("spicy");

}

};

P2.taste();

}

}

The generated class files are

PopCon.class

Test.class

Test\$1.class

Test\$2.class

Analysis

1. PopCon p=new PopCon()
2. PopCon p=new PopCon()
{
};
 1. We are declaring a class that extends PopCon without Name (Anonymous inner class)
 2. For that child class we are creating an object with parent reference.
3. PopCon p=new PopCon()
{
 Public void taste()
 {
 Syytem.out.println("spicy");
 }
};
 1. We are declaring a class that extends PoCon without name(Anonymous inner class)
 2. In that child class we are over riding taste method.
 3. For that child class we are creating an object with parent reference.

4. Static Nested class

Sometimes we can declare inner class with static modifier such types of inner classes are called static nested classes.

In the case of normal or regular inner without existing outer class object there is no chance of existing inner class object that is inner class object is strongly associated with outer class object.

But in the case of static nested classes without existing outer class object there may be a chance of nested class object. Hence static nested class object is not strongly associated with outer class object.

Class Outer

```
{  
    Static class Nested  
    {  
        Public void m1()  
        {  
            System.out.println("static nested class");  
        }  
    }  
}
```



```

    public static void main (String[] args)
    {
        Nested n=new Nested ();
        n.m1();
    }
}

```

If you want to create nested class object from outside of outer class then we can create as follows.

```
Outer.Nested n=new Outer.Nested()
```

UNIT-II

INHERITANCE

Inheritance Basics

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

Important terminology:

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

The keyword used for inheritance is **extends**.

Syntax:

```

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

```

Example

```

// A class to display the attributes of the vehicle
class Vehicle {
    String color;
}

```

```

int speed;
int size;
void attributes() {
    System.out.println("Color : " + color);
    System.out.println("Speed : " + speed);
    System.out.println("Size : " + size);
}
}

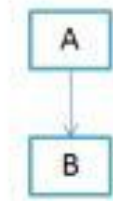
// A subclass which extends for vehicle
class Car extends Vehicle {
    int CC;
    int gears;
    void attributescar() {
        // The subclass refers to the members of the superclass
        System.out.println("Color of Car : " + color);
        System.out.println("Speed of Car : " + speed);
        System.out.println("Size of Car : " + size);
        System.out.println("CC of Car : " + CC);
        System.out.println("No of gears of Car : " + gears);
    }
}

public class Test {
    public static void main(String args[]) {
        Car b1 = new Car();
        b1.color = "Blue";
        b1.speed = 200 ;
        b1.size = 22;
        b1.CC = 1000;
        b1.gears = 5;
        b1.attributescar();
    }
}

```

Types of Inheritance

1. **Single Inheritance** : In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



(a) Single Inheritance

Example

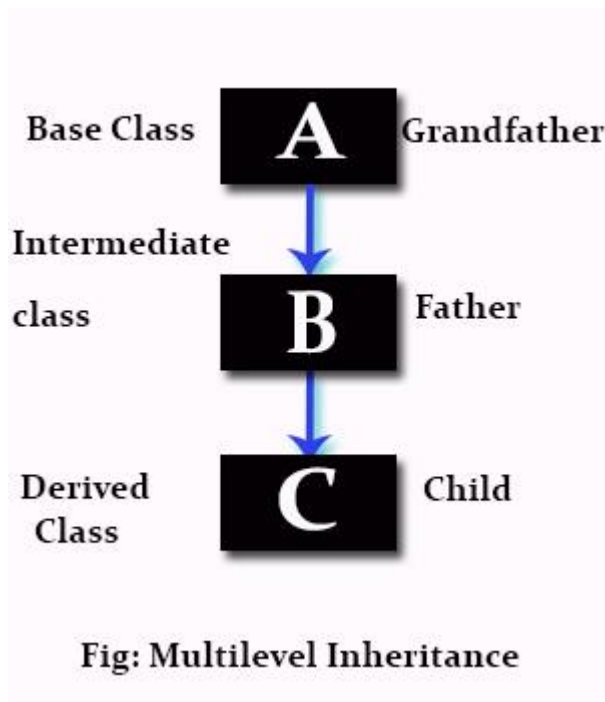
Class A

```
{
    public void methodA()
    {
        System.out.println("Base class method");
    }
}
```

Class B extends A

```
{
    public void methodB()
    {
        System.out.println("Child class method");
    }
    public static void main(String args[])
    {
        B obj = new B();
        obj.methodA(); //calling super class method
        obj.methodB(); //calling local method
    }
}
```

2. **Multilevel Inheritance :** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the **grandparent's members**.



Example

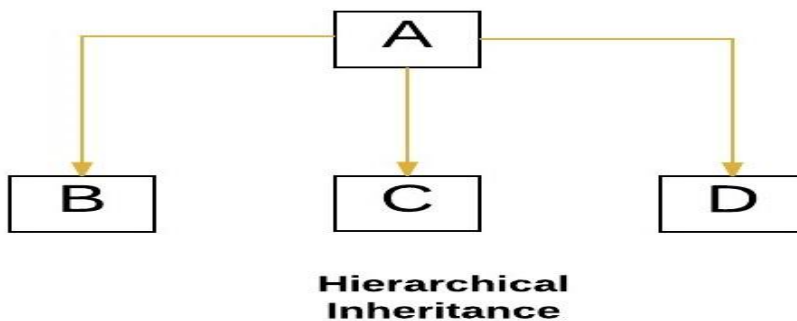
```
Class X
{
    public void methodX()
    {
        System.out.println("Class X method");
    }
}
Class Y extends X
{
    public void methodY()
    {
        System.out.println("class Y method");
    }
}
Class Z extends Y
{
    public void methodZ()
```

```

{
    System.out.println("class Z method");
}
public static void main(String args[])
{
    Z obj = new Z();
    obj.methodX(); //calling grand parent class method
    obj.methodY(); //calling parent class method
    obj.methodZ(); //calling local method
}
}

```

3. **Hierarchical Inheritance :** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class. In below image, the class A serves as a base class for the derived class B, C and D.



Example

Class A

```

{
    public void methodA()
    {
        System.out.println("method of Class A");
    }
}

```

Class B extends A

```

{
    public void methodB()
    {
        System.out.println("method of Class B");
    }
}

```

Class C extends A

```

{
    public void methodC()
    {
        System.out.println("method of Class C");
    }
}

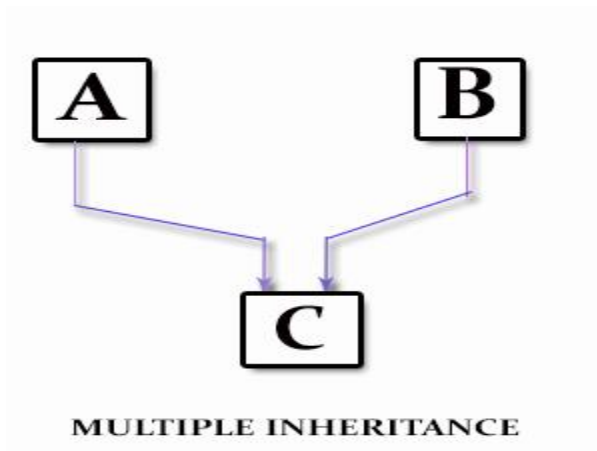
```

```

}
Class D extends A
{
    public void methodD()
    {
        System.out.println("method of Class D");
    }
}
Class MyClass
{
    public void methodB()
    {
        System.out.println("method of Class B");
    }
    public static void main(String args[])
    {
        B obj1 = new B();
        C obj2 = new C();
        D obj3 = new D();
        obj1.methodA();
        obj2.methodA();
        obj3.methodA();
    }
}

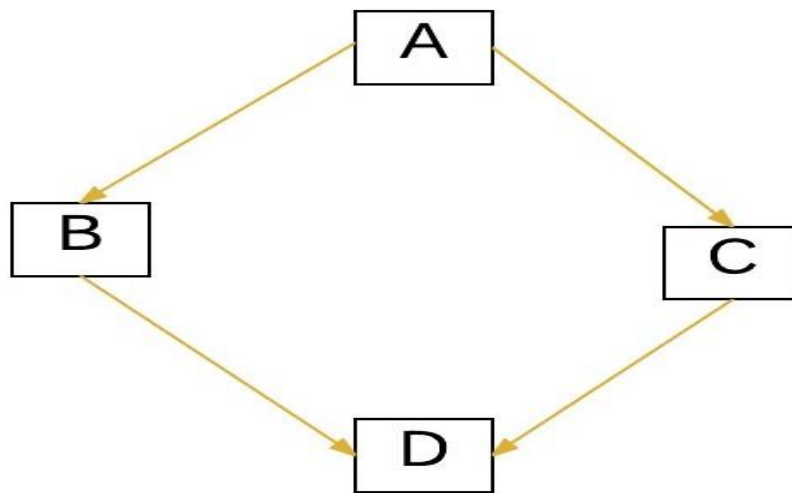
```

4. **Multiple Inheritance (Through Interfaces)** : In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



5. **Hybrid Inheritance (Through Interfaces)** : It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritances with

classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance

Member access and Inheritance

A subclass includes all of the members of its super class but it cannot access those members of the super class that have been declared as private. Attempt to access a private variable would cause compilation error as it causes access violation. The variables declared as private, is only accessible by other members of its own class. Subclass has no access to it.

Example

/* In a class hierarchy, private members remain private to their class.
This program contains an error and will not compile.

*/

// Create a superclass.

class A

{

int i; // public by default

private int j; // private to A

void setij(int x, int y)

{

i = x;

```

    j = y;
    }
    }
    // A's j is not accessible here.
    class B extends A
    {
    int total;
    void sum()
    {
    total = i + j; // ERROR, j is not accessible here
    }
    }
    class Access
    {
    public static void main(String args[])
    {
    B subOb = new B();
    subOb.setij(10, 12);
    subOb.sum();
    System.out.println("Total is " + subOb.total);
    }
    }

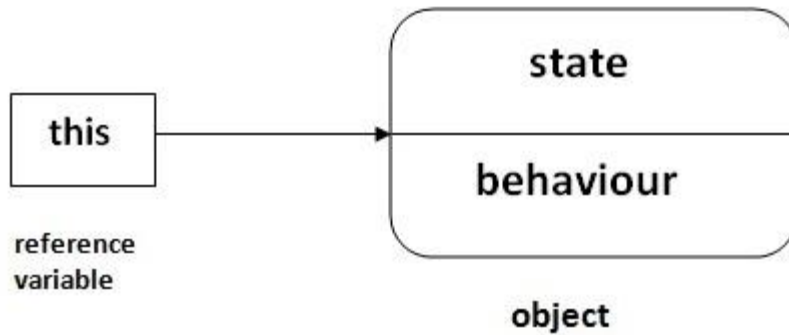
```

This Keyword

This is a reference variable that refers to the current object. It is a keyword that represents current class object.

Uses of This Keyword

1. It can be used to refer current class instance variable.
2. It can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.



1. This: to refer current class instance variable

This keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Example without using this keyword

```
class Employee
{
    int id;
    String name;

    Employee(int id, String name)
    {
        id = id;
        name = name;
    }
    void show()
    {
        System.out.println(id+" "+name);
    }
    public static void main(String args[])
    {
        Employee e1 = new Employee(111,"Harry");
        Employee e2 = new Employee(112,"Jacy");
        e1.show();
        e2.show();
    }
}
```

Output

```
0 null
0 null
```

To differentiate between formal parameter and data member of the class, the data members of the class must be preceded by "**this**".

Example of this keyword in java

```
class Employee
{
    int id;
    String name;

    Employee(int id,String name)
    {
        this.id = id;
        this.name = name;
    }
    void show()
    {
        System.out.println(id+" "+name);
    }
    public static void main(String args[])
    {
        Employee e1 = new Employee(111,"Harry");
        Employee e2 = new Employee(112,"Jacy");
        e1.show();
        e2.show();
    }
}
```

Output

```
111 Harry
112 Jacy
```

Program where this keyword is not required

```
class Employee
{
    int id;
    String name;

    Employee(int i,String n)
    {
        id = i;
        name = n;
    }
}
```

```

void show()
{
    System.out.println(id+" "+name);
}
public static void main(String args[])
{
    Employee e1 = new Employee(111,"Harry");
    Employee e2 = new Employee(112,"Jacy");
    e1.show();
    e2.show();
}
}

```

Output

```

111 Harry
112 Jacy

```

2. This: to invoke current class method

By using this keyword you can invoke the method of the current class. If you do not use the **this** keyword, compiler automatically adds this keyword at time of invoking of the method.

```

class Student
{
    void show()
    {
        System.out.println("You got A+");
    }
    void marks()
    {
        this.show(); //no need to use this here because compiler does it.
    }

    void display()
    {
        marks(); //compiler act marks() as this.marks()
    }
    public static void main(String args[])
    {
        Student s = new Student();
        s.display();
    }
}

```

Output

You got A+

3. this () : to invoke current class constructor

Which can be used to call one constructor within the another constructor without creation of objects multiple times for the same class.

Rules to use this()

1. this() always should be the first statement of the constructor.
2. One constructor can call only other single constructor at a time by using this().

Calling default constructor from parameterized constructor:

```
class A
{
A()
{
System.out.println("hello a");
}
A(int x)
{
this();
System.out.println(x);
}
}
class TestThis1
{
public static void main(String args[])
{
A a=new A(10);
}
}
```

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor:

```
class A
{
A()
{
this(5);
System.out.println("hello a");
}
A(int x)
{
System.out.println(x);
}
```

```

}
}
class TestThis2
{
public static void main(String args[])
{
A a=new A();
}
}

```

Output:

5
hello a

Super keyword

The super keyword is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1. Super: to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

Example

```

class Student
{
String name="Hari";
}
class Marks extends Student
{
String name="Ram";
void display()
{
    System.out.println("child class Name: "+name);//print child class ID
}
}

```

```

        System.out.println("parent class Name: "+super.name);//print base class ID
    }
}
class Supervariable
{
public static void main(String[] args)
{
Marks obj=new Marks();
obj.display();
}
}

```

2. Super: to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class.

Example

```

class Student
{
void message()
{
System.out.println("Good Morning Sir");
}
}

class Faculty extends Student
{
void message()
{
System.out.println("Good Morning Students");
}

void display()
{
message();//will invoke or call current class message() method
super.message();//will invoke or call parent class message() method
}

public static void main(String args[])
{
Facultyent f=new Faculty();
f.display();
}
}

```

```
}
```

3. Super: to invoke parent class constructor.

The super keyword can also be used to invoke or call the parent class constructor. To establish the connection between base class constructor and derived class constructors JVM provides two implicit methods they are:

1. Super()
2. Super(...)

Example

```
class Person
{
    int id;
    String name;
    Person(int id,String name)
    {
        this.id=id;
        this.name=name;
    }
}
class Emp extends Person
{
    float salary;
    Emp(int id,String name,float salary)
    {
        super(id,name); //reusing parent constructor
        this.salary=salary;
    }
    void display()
    {
        System.out.println(id+" "+name+" "+salary);
    }
}
class TestSuper
{
    public static void main(String[] args)
    {
        Emp e1=new Emp(1,"ankit",45000f);
        e1.display();
    }
}
```

Creating Multilevel Hierarchy

In simple inheritance a subclass or derived class derives the properties from its parent class, but in multilevel inheritance a subclass is derived from a derived class. One class inherits only single class. Therefore, in multilevel inheritance, every time ladder increases by one. The lower most class will have the properties of all the super classes'.

It is common that a class is derived from another derived class. The class student serves as a base class for the derived class marks, which in turn serves as a base class for the derived class percentage. The class marks is known as intermediates base class since it provides a link for the inheritance between student and percentage.

The chain is known as inheritance path. When this type of situation occurs, each subclass inherits all of the features found in all of its super classes. In this case, percentage inherits all aspects of marks and student.

Program : Multi_Inhe.java

```
class student
{
    int rollno;
    String name;

    student(int r, String n)
    {
        rollno = r;
        name = n;
    }

    void dispdatas()
    {
        System.out.println("Rollno = " + rollno);
        System.out.println("Name = " + name);
    }
}

class marks extends student
{
    int total;
    marks(int r, String n, int t)
    {
        super(r,n); //call super class (student) constructor
        total = t;
    }
    void dispdatam()
    {
        dispdatas(); // call dispdatap of student class
    }
}
```



```

        System.out.println("Total = " + total);
    }
}

class percentage extends marks
{
    int per;

    percentage(int r, String n, int t, int p)
    {
        super(r,n,t); //call super class(marks) constructor
        per = p;
    }
    void dispdatap()
    {
        dispdatam(); // call dispdatap of marks class
        System.out.println("Percentage = " + per);
    }
}

class Multi_Inhe
{
    public static void main(String args[])
    {
        percentage stu = new percentage(161289, "Naveen", 350, 70); //call constructor percentage
        stu.dispdatap(); // call dispdatap of percentage class
    }
}

```

Output:

```

Rollno = 161289
Name = Naveen
Total = 350
Percentage = 70

```

Method Overriding

When a method in a sub class has same name, same number of arguments and same type signature as a method in its super class, then the method is known as overridden method. Method overriding is also referred to as runtime polymorphism. The key benefit of overriding is the ability to define method that's specific to a particular subclass type.

```

class Bank
{
    int getRateOfInterest()
    {

```

```
return 0;  
}  
}
```

```
class SBI extends Bank  
{  
int getRateOfInterest()  
{  
return 8;  
}  
}
```

```
class ICICI extends Bank  
{  
int getRateOfInterest()  
{  
return 7;  
}  
}
```

```
class AXIS extends Bank  
{  
int getRateOfInterest()  
{  
return 9;  
}  
}
```

```
class TestOverRide  
{  
public static void main(String args[])  
{  
SBI s=new SBI();  
ICICI i=new ICICI();  
AXIS a=new AXIS();  
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());  
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());  
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());  
}  
}
```

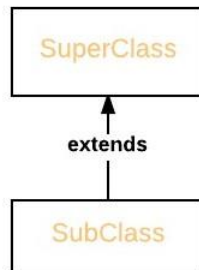
Dynamic method dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

1. When an overridden method is called through a super class reference, Java determines which version (super class/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
2. A super class reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

Upcasting

SuperClass obj = new SubClass



Example

// A Java program to illustrate Dynamic Method
// Dispatch using multilevel inheritance

```
class A
{
    void m1()
    {
        System.out.println("Inside A's m1 method");
    }
}
```

```
class B extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside B's m1 method");
    }
}
```

```
class C extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside C's m1 method");
    }
}
```

```
// Driver class
class Dispatch
{
    public static void main(String args[])
    {
        // object of type A
        A a = new A();

        // object of type B
        B b = new B();

        // object of type C
        C c = new C();

        // ref refers to an A object
        A ref = a;

        // calling A's version of m1()
        ref.m1();

        // now ref refers to a B object
        ref = b;

        // calling B's version of m1()
        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
    }
}
```

Output:

Inside A's m1 method
Inside B's m1 method
Inside C's m1 method

Explanation:

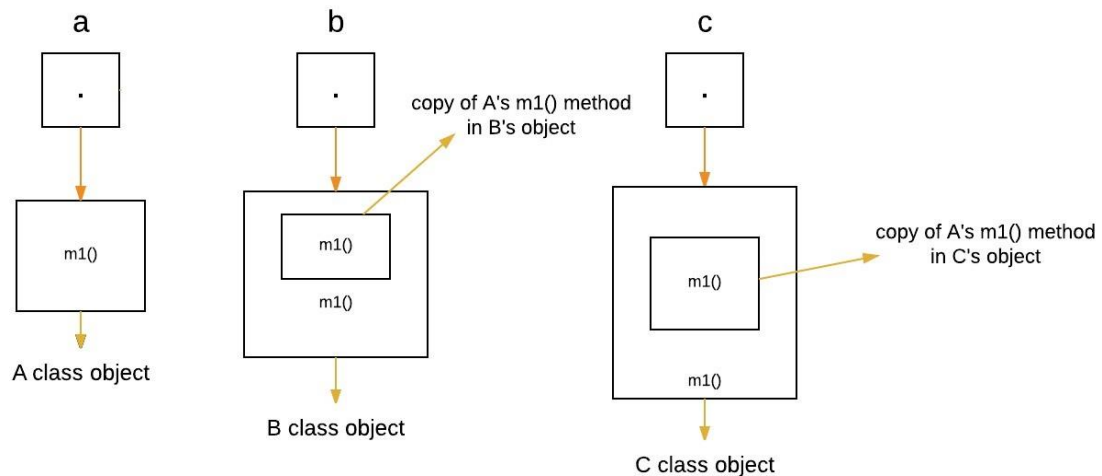
The above program creates one superclass called A and its two subclasses B and C. These subclasses override the m1() method.

1. Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.

```

A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C

```



2. Now a reference of type A, called ref, is also declared, initially it will point to null.

```

A ref; // obtain a reference of type A

```

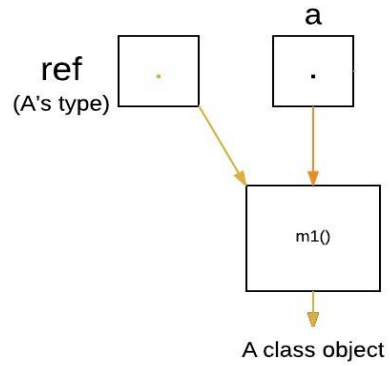
ref
(A's type) null

3. Now we are assigning a reference to each type of object (either A's or B's or C's) to ref, one-by-one, and uses that reference to invoke m1(). As the output shows, the version of m1() executed is determined by the type of object being referred to at the time of the call.

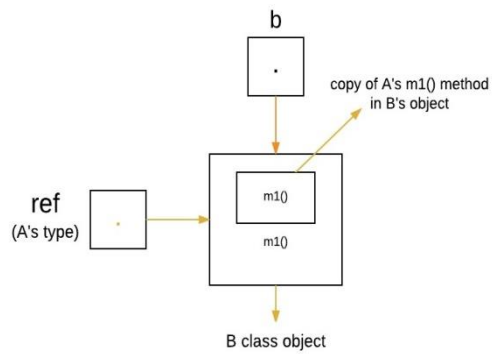
```

ref = a; // r refers to an A object
ref.m1(); // calling A's version of m1()

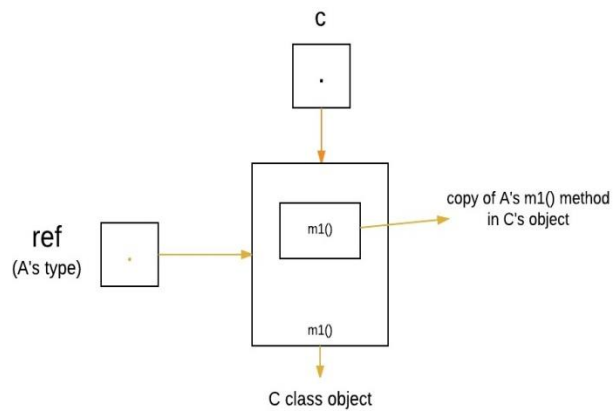
```



`ref = b; // now r refers to a B object`
`ref.m1(); // calling B's version of m1()`



`ref = c; // now r refers to a C object`
`ref.m1(); // calling C's version of m1()`



Abstract class

If a class contains any abstract methods then the class is declared as abstract class. An abstract class is never instantiated. It is used to provide abstraction. it can also have concrete method.

Syntax:

```
abstract class class_name
{
}
```

Abstract method

Methods that are declared without body within an abstract class are called abstract methods. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax :

```
abstract return_type function_name (parameter-list); // No definition
```

Example of Abstract class

```
abstract class A
{
    abstract void callme();
}
class B extends A
{
    void callme()
    {
        System.out.println("this is callme.");
    }
    public static void main(String[] args)
    {
        B b = new B();
        b.callme();
    }
}
```

Output:

this is callme.

Abstract class with concrete(normal) method.

```

abstract class A
{
    abstract void callme();
    public void normal()
    {
        System.out.println("this is concrete method");
    }
}
class B extends A
{
    void callme()
    {
        System.out.println("this is callme.");
    }
    public static void main(String[] args)
    {
        B b = new B();
        b.callme();
        b.normal();
    }
}

```

Output:

```

this is callme.
this is concrete method.

```

// Using abstract methods and classes.

```

abstract class Figure
{
    double dim1;
    double dim2;
    Figure(double a, double b)
    {
        dim1 = a;
        dim2 = b;
    }
    // area is now an abstract method
    abstract double area();
}

```

```

class Rectangle extends Figure
{

```



```

Rectangle(double a, double b)
{
    super(a, b);
}
// override area for rectangle
double area()
{
    System.out.println ("Inside Area for Rectangle.");
    return dim1 * dim2;
}
}
class Triangle extends Figure
{
    Triangle(double a, double b)
    {
        super(a, b);
    }
    // override area for triangle
    double area()
    {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
class AbstractAreas
{
    public static void main(String args[])
    {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref; // this is OK, no object is created
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
    }
}

```

Points to Remember

1. An abstract class may or may not have an abstract method. But if any class has even a single abstract method, then it must be declared as abstract.
2. Abstract classes can have Constructors, Member variables and Normal methods.
3. Abstract classes are never instantiated.
4. When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

Package

A package is a collection of similar types of classes, interfaces and sub-packages.

The package statement defines a name space in which classes are stored. It helps Organize your classes into a folder structure and make it easy to locate and use them.

The general form of the package statement:

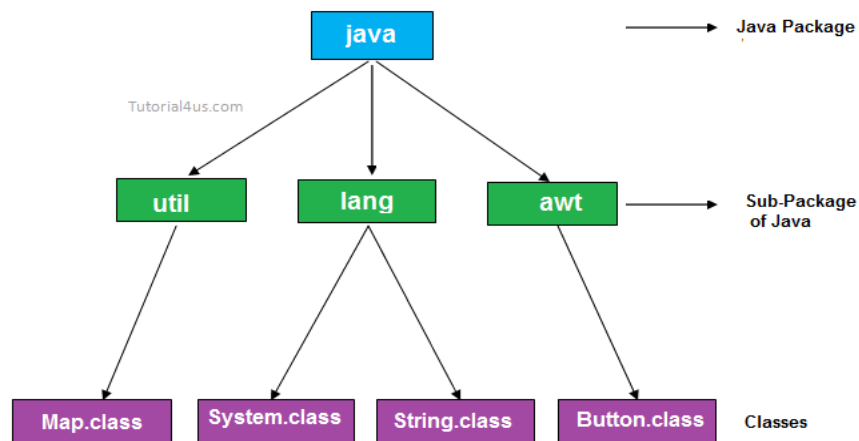
package pkg;

Here, pkg is the name of the package

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]];`

The purpose of package concept is to provide common classes and interfaces for any program separately. In other words if we want to develop any class or interface which is common for most of the java programs than such common classes and interfaces must be place in a package.



Advantage of package

1. Package is used to categorize the classes and interfaces so that they can be easily maintained
2. Application development time is less, because reuse the code
3. Application memory space is less (main memory)
4. Application execution time is less
5. Application performance is enhance (improve)
6. Redundancy (repetition) of code is minimized
7. Package provides access protection.
8. Package removes naming collision.

Type of package

Packages are classified into two type which are given below.

1. Predefined or built-in package
2. User defined package

Predefined or built-in package

These are the packages which are already designed by the Sun Microsystem and supply as a part of java API, every predefined package is collection of predefined classes, interfaces and sub-package.

User defined package

If any package is design by the user is known as user defined package. User defined package are those which are developed by java programmer and supply as a part of their project to deal with common requirement.

Rules to create user defined package

1. The first statement in the program must be package statement while creating a package.
2. While creating a package except instance variables, declare all the members and the class itself as public then only the public members are available outside the package to other programs.

Example Program

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

Finding Packages and CLASSPATH

packages are mirrored by directories. We will find the packages in 3 ways

1. the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
2. We 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 the classes.

Access Protection

Java provides many levels of protection to allow fine-grained control over visibility of the variables and methods within classes, subclasses, and packages.

Classes and packages are means of encapsulating and containing the name space and scope of the variables and methods.

Class Members Visibility

Java addresses the following four categories of visibility for class members:

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

The four access modifiers are:

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

1. Public: Anything declared as public can be accessed from anywhere.

2. Private: Anything declared as private can't be seen outside of its class.

3. Protected: anything declared as protected can be seen outside your current package, but only to the classes that subclass your class directly.

4. Default: a member doesn't have an explicit access specification, then it is visible to the subclasses as well as to the other classes in the same package.

Class Member Access

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

Java Access Protection Example

Protection.java

```
package pkg1;

public class Protection
{
    int n = 1;
    private int n_priv = 2;
    protected int n_prot = 3;
    public int n_publ = 4;

    public Protection()
    {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_priv = " + n_priv);
        System.out.println("n_prot = " + n_prot);
        System.out.println("n_publ = " + n_publ);
    }
}
```

Derived.java

```
package pkg1;

class Derived extends Protection
{
    Derived()
    {
        System.out.println("derived constructor");
        System.out.println("n = " + n);

        /* class only
        * System.out.println("n_priv = " + n_priv); */

        System.out.println("n_prot = " + n_prot);
        System.out.println("n_publ = " + n_publ);
    }
}
```

SamePackage.java

```
package pkg1;
```

```

class SamePackage
{
    SamePackage()
    {
        Protection pro = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + pro.n);

        /* class only
        * System.out.println("n_priv = " + pro.n_priv); */

        System.out.println("n_prot = " + pro.n_prot);
        System.out.println("n_publ = " + pro.n_publ);
    }
}

```

This is Protection2.java file

```

package pkg2;

class Protection2 extends pkg1.Protection
{
    Protection2()
    {
        System.out.println("derived other package constructor");

        /* class or package only
        * System.out.println("n = " + n); */

        /* class only
        * System.out.println("n_priv = " + n_priv); */

        System.out.println("n_prot = " + n_prot);
        System.out.println("n_publ = " + n_publ);
    }
}

```

This is OtherPackage.java file

```

package pkg2;

```

```

class OtherPackage
{
    OtherPackage()
    {
        pkg1.Protection pro = new pkg1.Protection();

        System.out.println("other package constructor");

        /* class or package only
        * System.out.println("n = " + pro.n); */

        /* class only
        * System.out.println("n_priv = " + pro.n_priv); */

        /* class, subclass or package only
        * System.out.println("n_prot = " + pro.n_prot); */

        System.out.println("n_publ = " + pro.n_publ);
    }
}

```

Test for package pkg1

```

/* demo package pkg1 */

package pkg1;

/* instantiate the various classes in pkg1 */
public class Demo
{
    public static void main(String args[])
    {
        Protection obj1 = new Protection();
        Derived obj2 = new Derived();
        SamePackage obj3 = new SamePackage();
    }
}

```

Test for package pkg2

```

/* demo package pkg2 */

```



```

package pkg2;

/* instantiate the various classes in pkg2 */
public class Demo
{
    public static void main(String args[])
    {
        Protection2 obj1 = new Protection2();
        OtherPackage obj2 = new OtherPackage();
    }
}

```

Importing Packages

Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program.

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. Fully qualified name.

1. Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not sub packages.

Example

```

//save by A.java
package pack;
public class A
{
    public void msg()

```

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

```
//save by B.java  
package mypack;  
import pack.*;
```

```
class B  
{  
    public static void main(String args[])  
    {  
        A obj = new A();  
        obj.msg();  
    }  
}
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example

```
//save by A.java
```

```
package pack;  
public class A  
{  
    public void msg()  
    {  
        System.out.println("Hello");  
    }  
}
```

```
//save by B.java  
package mypack;  
import pack.A;
```

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

```
{
    A obj = new A();
    obj.msg();
}
}
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example

//save by A.java

```
package pack;
public class A
{
    public void msg()
    {
        System.out.println("Hello");
    }
}
```

//save by B.java

```
package mypack;
class B
{
    public static void main(String args[])
    {
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```

UNIT – III

1. EXCEPTION HANDLING

1.1 Exception- Handling Fundamentals

An exception is a problem that arises during the execution of a program. When an **Exception(un wanted event)** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally.

Some of the reasons for an exception

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

If the exception Object is not handled properly, the interpreter will display the error and will terminate the program. It is highly recommended to handle exceptions. The main objective of exception handling is graceful termination of program. Exception handling doesn't mean repairing an exception. we have to provide alternative way to continue rest of the program normally is the concept of **exception handling**.

Exception handling can be managed by five keywords:

- 1.try,
2. catch,
- 3.throw,
- 4.throws,
- 5.finally.

This is the general form of an 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
}
```

2. Uncaught Exceptions

Class Test

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

```

        doStuff();
    }
    public static doStuff()
    {
        doMoreStuff();
    }
    public static doMoreStuff()
    {
        System.out.println(10/0);//Here exception occurs
    }
}

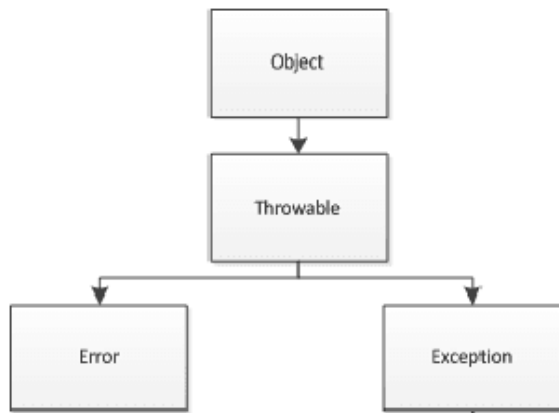
```

Steps to handle Uncaught Exceptions

1. Inside a method if any exception occurs the method in which it is raised is responsible to create exception object by including the following information
 1. name of exception
 2. description of exception
 3. location at which exception occurs (stack trace)
2. After creation of exception object method hands over that object to the JVM.
3. JVM will check whether the method contains any exception handling code or not. If the method doesn't contain exception handling code then JVM terminates that method abnormally and removes corresponding entry from the stack.
4. JVM identifies caller method and checks whether caller method contains any handling code or not. If the caller method does not contain handling code then JVM terminates that caller method also abnormally and removes corresponding entry from the stack. This process will be continued until the main method. If the main method also doesn't contain handling code then JVM terminates main method also abnormally and removes corresponding entry from the stack.
5. Then JVM hands over responsibility of exception handling to default exception handler, which is the part of JVM.
6. Default Exception Handler prints exception information in the following format and terminates program abnormally.

Exception in thread in XXX : name of the exception: Description: stack trace

Hierarchy of Exception classes



Exception

Most of the times exceptions are caused by our program and these are recoverable.

Child classes for Exception

1. IOException
 - i. EOF Exception
 - ii. FileNotFoundException
 - iii. InterruptedIOException
2. SQLException
3. ServletException
4. RuntimeException
 - i. ArithmeticException
 - ii. NumberFormatException
 - iii. ArrayIndexOutOfBoundsException
 - iv. NullPointerException
5. RemoteException
6. InterruptedException

Error

Most of the times errors are not caused by our program and these are due to lack of system resources. Errors are non recoverable.

Child classes for Error

1. VMError
 - i. StackOverflowError
 - ii. OutOfMemoryError
2. AssertionError

3. Types of Exception

There are mainly two types of exceptions

1. Checked Exception
2. Unchecked Exception

Checked Exception:

The exceptions which are checked by compiler for smooth execution of the program are called checked exception.

Example: FileNotFoundException, SQLException

In our program if there is chance of rising checked exception then compulsory we should handle that checked exception either by try, catch or throws key word otherwise we will get compile time error.

Unchecked Exception

The exception which are not checked by compiler whether programmer handling or not such type of exceptions are called Unchecked Exception.

Example

ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Note :

1. Whether it is checked or unchecked exception that occurs at runtime only there is no chance of occurring any exception at compile time.
2. RuntimeException and its child classes, Error and its child classes are unchecked except these remaining are checked.

4. Exception Handling Using Try Catch

It is highly recommended to handle exceptions.

The code which may raise exception is called risky code and we have to define that code inside the try block.

And the corresponding handling code we have to define inside the catch block.

Syntax

Try

{

Risky code

}

Catch(Exception e)

{

Handling code;

}

Example with out Try catch

Class test

{

Public static void main(String args[])

{

System.out.println("statement1");

System.out.println(10/0);

System.out.println("statement3");

}

```
}
```

Out Put:

Statement1

Exception:AE:divide by zero

Example with Try catch

Class test

```
{  
Public static void main(String args[])  
{  
System.out.println("Before Exception ");  
try  
{  
System.out.println(10/0);  
}  
Catch(ArithmeticException e)  
{  
System.out.println(10/2);  
}  
System.out.println("After catch statement.");  
}  
}
```

Out Put:

Statement1

5

Statement3

Control Flow in Try Catch

Try

```
{  
Statement1  
statement2  
statement3  
}  
Catch(Exception e)  
{  
statement4  
}  
Statement5
```

Case1: No Exception

Out: 1,2,3,5 ->normal termination

Case2: if an exception raised at statement 2 and corresponding catch block matched

Out put: 1, 4, 5->normal termination

Case 3: if an exception raised at statement 2 and corresponding catch block not matched

Out Put: 1, --→ abnormal termination

Case 4: if an exception raised at statement 4 or statement 5

Output: Abnormal termination

Note:

1. Within the try block if any where exception raised then rest of the try block won't be executed even though we handled exception. Hence we have to write only risky code into the try block.
2. In addition to try block there may be a chance of raising an exception inside catch and finally blocks.
3. If any statement which is not part of try block and raise an exception then it is always abnormal termination.

Methods to print Exception Information

Throwable class defines the following methods to print Exception Information

Method	Printable Format
printStackTrace()	Name of the Exception : Description : Stack Trace
toString()	Name of the Exception : Description
getMessage()	Description

Note: internally default exception handler will use printStackTrace to print exception information to the console.

Example

```
class Test
{
    public static void main(String args[])
    {
        try
        {
            system.out.println(10/0);
        }
        catch(ArithmeticException e)
        {
            e.printStackTrace();// it will display complete information
            System.out.println(e.toString());//it display partial information
            System.out.println(e.getMessage());// it display only expectation name
        }
    }
}
```

```
}  
}  
}
```

Try with multiple catch blocks

The way of handling an exception is varied from exception to exception hence for every exception type, it is highly recommended to take separate catch block that is try with multiple catch blocks is always possible and recommended to use.

Example

Try

```
{
```

Risky code;

```
}
```

Catch(Exception e)

```
{
```

```
}
```

Bad programming practice

Example:

Try

```
{
```

Risky code;

```
}
```

Catch(Arithmetic Exception e)

```
{
```

Perform any Arithmetic operation;

```
}
```

Catch(FileNotFoundException e)

```
{
```

Use local file instead of remote file;

```
}
```

Catch(Exception e)

```
{
```

//Default Exception Handling;

```
}
```

Best Programming Practice

If try with multiple catch blocks present then the order of catch blocks is very important.

We have to take child first and then parent otherwise we will get compile time error saying

Exception XXX has already been caught

Example

Try

```
{
```

Risky code

```
}  
Catch(Exception e)  
{  
}  
Catch(ArithmeticException e){  
}
```

Output: Compile time error.

Example

```
Try  
{  
Risky code  
}  
Catch(ArithmeticException e)  
{  
}  
Catch(Exception e)  
{  
}
```

Output:

Exception handled .

Nested try Statements

One try-catch block can be present in the another try's body. This is called Nesting of try catch blocks.

Each time a try block does not have a catch handler for a particular exception, the stack is unwound and the next try block's catch handlers are inspected for a match.

If no catch block matches, then the java run-time system will handle the exception.

Syntax of Nested try Catch

.... //Main try block

```
try  
{  
statement 1;  
statement 2;  
//try-catch block inside another try block  
try  
{  
statement 3;  
statement 4;  
}  
catch(Exception e1)  
{  
//Exception Message  
}
```

```

}
catch(Exception e3) //Catch of Main(parent) try block
{
//Exception Message
}

```

5. Throw Keyword

Sometimes we can create exception object explicitly and handover to jvm manually, for this we have to use throw key word.

Throw new ArithmeticException("/ by zero");

The main objective of throw key word is to handover our created exception object to jvm manually.

Example

Class test

```

{
Public static void main(String args[])
{
Throw new ArithmeticException("/ by zero");
}
}

```

Best use of throw keyword is for user defined exceptions or customized exceptions.

6. Throws keyword

In our program if there is a possibility of raising checked exception then compulsory we should handle that checked exception otherwise we will get compile time error saying

Unreported Exception XXX must be caught or declared to be thrown

Syntax

```

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

```

Example

Class test

```

{
Public static void main(String args[])
PrintWriter pw=new PrintWriter("abc.txt");
pw.println("hello");
}
}

```

Compile time error: Unreported Exception java.io.FileNotFoundException; must be caught or declared to be thrown

We can handle this compile time error by using Throws keyword or by using Try catch. The purpose of throws keyword is to delegate the responsibility of exception handling to the caller (it may be another method or JVM) then caller method is responsible to handle that exception.

Example

Class test

```

{
Public static void main(String args[]) throws FileNotFoundException
{
    PrintWriter pw=new PrintWriter("abc.txt");
    pw.println("hello");
}
}

```

Throws key word requires only for checked exception. Throws key word required only to convince compiler and usage of throws keyword doesn't prevent abnormal termination of the program.

7. Finally block

Finally block is a block that is used *to execute important (clean up) code* such as closing connection, stream etc. finally block is always executed whether exception is handled or not.

Example

```

Try
{
    Risky code
}
Catch(Exception e)
{
}
finally
{
    Clean up code;
}

```

8. Java's Built-in Exceptions

1. Unchecked Exceptions

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

2. Checked Exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

9. User defined Exception subclass

We can also create our own exception sub class simply by extending java Exception class.

Example

```

class TooYoungException extends RuntimeException
{
    TooYoungException(String s)
    {
        super(s);
    }
}
class CustException
{
    public static void main(String args[])
    {
        int age=Integer.parseInt(args[0]);
        if(age<18)
        {
            throw new TooYoungException("you are too young so u r not allowed");
        }
        else
        {
            System.out.println("u r allowed");
        }
    }
}

```

10. Chained Exception

Chained Exception was added to Java in JDK 1.4. This feature allow you to relate one exception with another exception, i.e one exception describes cause of another exception.

Two new constructors and two new methods were added to Throwable class to support chained exception.

Throwable(*Throwable cause*)

Throwable(*String str, Throwable cause*)

getCause() and initCause() are the two methods added to Throwable class.

getCause() method returns the actual cause associated with current exception.

initCause() set an underlying cause(exception) with invoking exception.

Example

```

import java.io.IOException;
public class ChainedException
{
    public static void divide(int a, int b)
    {
        if(b==0)
        {

```

```

ArithmeticException ae = new ArithmeticException("top layer");
ae.initCause( new IOException("cause") );
throw ae;
}
else
{
System.out.println(a/b);
}
}
public static void main(String[] args)
{
Try
{
divide(5, 0);
}
catch(ArithmeticException ae)
{
System.out.println( "caught : " +ae);
System.out.println("actual cause: "+ae.getCause());
}
}
}

```

Multi-Tasking

Executing several tasks simultaneously is the concept of multi-tasking.

There are two types of multi-tasking

1. Process based
2. Thread based multi-tasking

Process based multi tasking

Executing several tasks simultaneously where each task is separate independent program(process) is called process based multi tasking.

Example

While writing java program we can listen songs using the same system, at the same time we can download a file from the net.

All these tasks will be executed simultaneously and independent of each other. Hence it is called process based multi tasking.

Process based multi tasking is best suitable at OS level

Thread based multi tasking

Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread based multi tasking and each independent part is called thread.

Thread based multitasking is best suitable at programmatic level.

Differences between Process-based Multitasking & Thread-based Multitasking

Process-based Multitasking

1. Each process have its own address in memory i.e. each process allocates separate memory area.

2. Process is heavyweight.
3. Cost of communication between the process is high.
4. Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

Thread-based Multitasking

1. Threads share the same address space.
2. Thread is lightweight.
3. Cost of communication between the thread is low.

The main important application areas of multithreading are

1. To develop multimedia graphics
2. To develop animations
3. To develop video games
4. To develop web servers and application servers etc.

When compared with old languages developing multithreaded application is very easy. java provides inbuilt support with rich API.(Thread,Runnable,threadGropup)

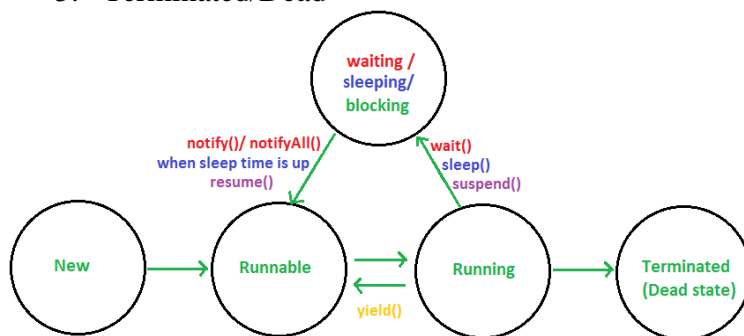
Life cycle of a Thread

What is Thread

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

A thread can be in one of the five states.

1. New/Born
2. Runnable/Ready
3. Running
4. Blocked
5. Terminated/Dead



1. **New** : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. **Runnable** : After invocation of start() method on new thread, the thread becomes runnable.
3. **Running** : A thread is in running state if the thread scheduler has selected it.
4. **Waiting** : A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
5. **Terminated** : A thread enter the terminated state when it complete its task.

Creating a Thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

By extending Thread class

Class Mythread extends Thread

```
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("run method");
        }
    }
}
```

Class Test

```
{
    public static void main(String args[])
    {
        Mythread t=new Mythread();
        t.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("main method");
        }
    }
}
```

By implementing Runnable interface

Class MyRunnable implements Runnable

```
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("child thread");
        }
    }
}
```

Class ThreadDemo

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

```

        Thread t=new Thread(r);
        t.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("main thread");
        }
    }
}

```

Creating Multiple Threads

class NewThread implements Runnable

```

{
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        //System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}

```

class MultiThreadDemo

```

{
    public static void main(String args[])
    {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try
        {
            Thread.sleep(10000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread Interrupted");
        }
    }
}

```

```

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

```

Thread Priorities

Every thread in java has some priority it may be default priority generated by JVM or customized priority provided by programmer

The valid range of thread priorities is 1 to 10.

where

1—MIN_PRIORITY

10---MAX_PRIORITY

Thread class defines the following constants to represent some standard priorities

Thread.MIN_PRIORITY-----1

Thread.NORM_PRIORITY-----5

Thraed.MAX_PRIORITY-----10

Thread scheduler will use priorities while allocating processor.

The Thread which is having highest priority will get chance first.

If two threads having same priority then we can't except exact execution order. it depends on thread scheduler.

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

public final int getPriority().

Public final void setPriority(int p)

We can prevent a thread execution by following methods

1.yield()

2.join()

3.sleep()

1. Yield(): yield method causes to pause current executing thread to give the chance for waiting thread of same priority. If there is no waiting thread or all waiting threads have low priority then same thread can continue its execution.

The thread which is yielded will get the chance once again will depends on thread scheduler and we can't expect exactly.

Syntax

Public static native void yield()

Example

Class mythread extends Thread

```

{
    Public void run()
    {
        For(int i=0;i<10;i++)

```

```

{
    Sop("child thread");
    Thread.yield();
}
}
}

Class ThreadYieldDemo
{
p.s.v.main(String args[])
{
    Mythread t=new Mythread();
    t.strat();
    for(int i=0;i<10;i++)
    {
        Sop("main thread");
    }
}
}

```

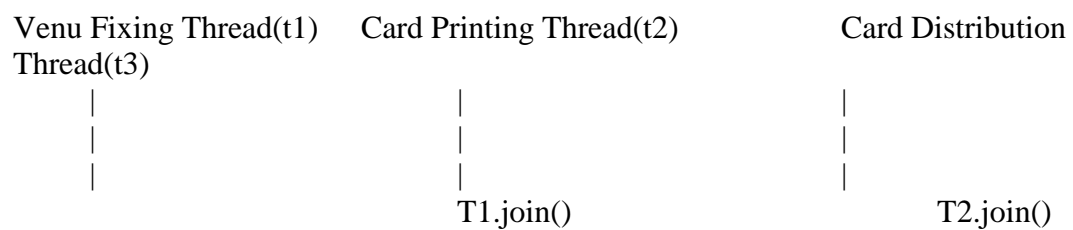
2.Join ()

If a thread wants to wait until completing some other thread then we should go for join() method.

For example If a thread T1 wants to wait until completing T2 then T1 has to call T2.join().

If T1 executes T2.join() then immediately T1 will be entered into waiting state until T2 completes. Once T2 completes then T1 can continue its execution.

Another example



Wedding cards printing thread T2 has to wait until venue fixing thread T1 completion hence T2 has to call T1.join(). wedding cards Distribution T3 has to wait wedding cards printing thread to complete hence T3 has to call T2.join().

Syntax

Public final void join() throws InterruptedException

Public final void join(long ms) throws InterruptedException

Public final void join(long ms, int ns) throws InterruptedException

Note:

Every join() method throws InterruptedException which is checked exception. Hence compulsory we should handle this exception either by using try catch or throws key word otherwise we will get compile time error.

Example

```
class Mythread extends Thread
```

```
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("seetha Thread");
            try
            {
                Thread.sleep(2000);
            }
            catch(InterruptedException e)
            {
            }
        }
    }
}
```

```
class ThreadJoinDemo
```

```
{
    public static void main(String args[])throws InterruptedException
    {
        Mythread t=new Mythread();
        t.start();
        t.join(); -----1
        for(int i=0;i<10;i++)
        {
            System.out.println("rama Thread");
        }
    }
}
```

```

    }
}

```

Case 1: Main thread waiting until completing child thread

1.If we comment line 1 then both main and child threads will be executed simultaneously and we can't expect exact output.

2.If we are not commenting line 1 then main thread calls join method on child thread hence main thread will wait until completing child thread in this case the output is

Seethread ---10

Ramathread ---10.

Case 2: Waiting of child thread until completing main thread.

```

class Mythread extends Thread
{
    static Thread mt;
    public void run()
    {
        try
        {
            mt.join();
        }
        catch(InterruptedException e)
        {
        }
        for(int i=0;i<10;i++)
        {
            System.out.println("child Thread");
        }
    }
}

class ThreadJoinDemo1
{
    public static void main(String args[])throws InterruptedException
    {
        Mythread.mt=Thread.currentThread();
        Mythread t=new Mythread();
        t.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("Main Thread");
        }
    }
}

```

In the above example child thread calls join on main thread object. Hence child thread has to wait until complete main thread

Output:

Main thread -10

Child thread--10

Case 3:

if main thread call join () method on child thread object and child thread call join() method on main thread object then both threads will wait forever and the program will be paused (this something like dead lock).

Case 4: if a thread calls join() method on the same thread itself then the program will be paused(this something like dead lock).

3. Sleep ():

If a thread don't want to perform any operation per a particular amount of time then we should go for sleep() method.

Syntax

Public static native void sleep(long ms) throws InterruptedException

Public static void sleep(long ms, int ns) throws InterruptedException

Note:

Every sleep() method throws InterruptedException, which is checked exception. hence when ever we are using sleep() method compulsory we should handle InterruptedException either by try catch or by throws key word otherwise we will get compile time error.

Example

```
class SlideRotator
{
    public static void main(String args[]) throws InterruptedException
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("Slide:"+i);
            Thread.sleep(5000);
        }
    }
}
```



```

    }
}
}

```

Thread Synchronization

1. Synchronized is the modifier applicable only for methods and blocks but not for class and variables.
2. If multiple threads are trying to operate simultaneously on the same java object then there may be a chance of data inconsistency problem.
To overcome this problem we should go for Synchronized key word.
3. If a method or block declared as Synchronized then at a time only one thread is allowed to execute that method or block on the given object so that data inconsistency problem will be resolved .

The main advantage of Synchronized key word is we can resolve data inconsistency problems but the main disadvantage of Synchronized key word is it increases waiting time of threads and creates performance problems hence if there is no specific requirement then it is not recommended to use Synchronized key word.

Internally Synchronization concept is implemented by using lock, every object in java has a unique lock. Whenever we are using Synchronized key word then only lock concept will come into the picture. If a thread wants to execute Synchronized method on the given object first it has to get lock of that object. Once thread got the lock then it is allowed to execute any Synchronized method on that object. Once method execution completes automatically thread release a lock.

Internally Acquiring and releasing lock takes care by JVM and programmer not responsible for this activity.

While a thread executing synchronized method on the given object then the remaining threads are not allowed to execute any synchronized method simultaneously but the remaining threads are allowed to execute non-synchronized method simultaneously.

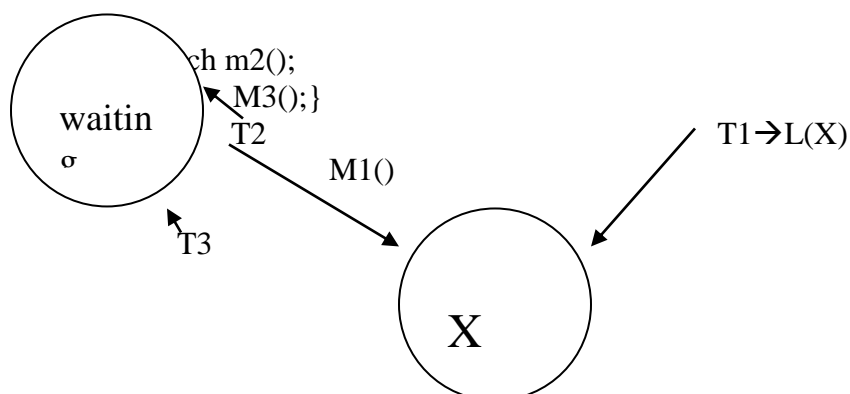
Example

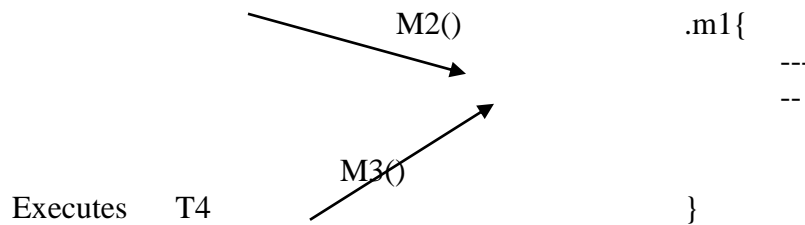
Class X

```

{
    Synchronized m1();

```





Lock concept is implemented based on object but not based on method.

Every object in java contains two areas

1. Synchronized area: this area can be accessed by only one thread at time. Here object state will be changed.

2. Non synchronized area: this area can be accessed by any number of threads at a time. Here object state can't be changed.

Example

```
class Display
{
    public synchronized void wish(String name)
    {
        for(int i=0;i<10;i++)
        {
            System.out.print("Good Morning:");
            Try
            {
                Thread.sleep(2000);
            }
            catch(InterruptedExpection e){ }
            System.out.println(name);
        }
    }
}

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

```

        d.wish(name);
    }
}
class thredSyncDemo
{
public static void main(String args[])
{
    Display d=new Display();
    //Display d1=new Display();
    Mythread t=new Mythread(d,"param");
    Mythread t1=new Mythread(d,"eashwar");
    t.start();
    t1.start();

}
}

```

1. If we are not declaring wish method as synchronized then both threads will be executed simultaneously and hence we will get irregular output.
2. If we are declaring **wish ()** method as synchronized then at a time only thread is allowed to execute **wish ()** method on the given **Display** object hence we will get regular output.

Case Study

```

Display d1=new Display();
Display d2=new Display();
Mythread t1=new Mythread(d1,"param");
Mythread t2=new Mythread(d2,"eashwar");
t1.start();
t2.start();

```

Even though wish method is synchronized we will get irregular output because threads are operating on different java objects.

Conclusion

If multiple threads are operating on same java object then synchronization is required.

If multiple threads are operating on multiple java objects then synchronization is not required.

Classes level Lock

Every class in java has a unique lock which is nothing but class level lock.

If a thread wants to execute a static synchronized method then thread required class level lock.

Once thread got class level lock then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread releases the lock.

While a thread executing static synchronized method the remaining threads are not allowed to execute any static synchronized method of that class simultaneously.

But remaining threads are allowed to execute the following methods simultaneously

1. Normal static methods
2. Synchronized instance methods
3. Normal instance methods.

Another Example on Synchronized methods

class Display

```
{
public synchronized void displayn()
{
    for(int i=0;i<10;i++)
    {
        System.out.print(i);
        try{
            Thread.sleep(2000);
        }catch(InterruptedException e){ }
    }
}
public synchronized void displayc()
{
    for(int i=65;i<75;i++)
    {
        System.out.print((char)i);
        try{
            Thread.sleep(2000);
        }catch(InterruptedException e){ }
    }
}
}
class Mythread1 extends Thread
{
    Display d;
    Mythread1(Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.displayn();
    }
}
```

```

class Mythread2 extends Thread
{
    Display d;
    Mythread2(Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.displayc();
    }
}

class thredSyncDemo1
{
    public static void main(String args[])
    {
        Display d=new Display();
        Mythread1 t1=new Mythread1(d);
        Mythread2 t2=new Mythread2(d);
        t1.start();
        t2.start();
    }
}

```

Synchronized Block

If very few lines of the code required synchronization then it is not recommended declaring entire method as synchronized we have to enclose those few line of the code by using synchronized block.

The main advantage of synchronized block over synchronized method is it reduces waiting time of threads and improves performance of the system.

We can declare synchronized block as follows

1. To get lock of current object

```

Synchronized(this)
{
    ;;;;
}

```

2. To get lock of particular object "B"

```
Synchronized(B)
{
    :::
}
```

3. To get class level lock

```
Synchronized(Display.class)
{
    :::
}
```

Example

```
class Display
{
    public void wish(String name)
    {
        ::::::::::// 1 lack line of code
        synchronized(this)
        {
            for(int i=0;i<10;i++)
            {
                System.out.print("Good Morning:");
                try{
                    Thread.sleep(2000);
                }catch(InterruptedException e){ }
                System.out.println(name);
            }
        }
        :::::::::://1 lack line of code
    }
}

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

```

        d.wish(name);
    }
}
class thredSyncBlockDemo
{
    public static void main(String args[])
    {
        Display d=new Display();
        //Display d1=new Display();
        Mythread t=new Mythread(d,"param");
        Mythread t1=new Mythread(d,"eashwar");
        t.start();
        t1.start();

    }
}

```

Inter thread communication

Inter thread communication is allowed between the synchronized threads.

Two threads can communicate with each other by using wait, notify and notifyAll methods. The thread which is expecting updation is responsible to call wait method. Then immediately the thread will enter into waiting state.

The thread which is responsible to perform updation, after performing updation, it is responsible to call notify method. Then waiting thread will get that notification and continue its execution with those updated items.

Syntax

Public final void wait() throws InterruptedException

Public final native void wait(long ms) throws InterruptedException

Public final void wait(long ms, int ns) throws InterruptedException

Public final native void notify ()

Public final native void notifyAll()

Example

```

class ThraedA
{
    public static void main(String args[]) throws InterruptedException
    {
        ThreadB b=new ThreadB();
        b.start();
    }
}

```

```

synchronized(b)
{
System.out.println("main thread calling wait method");
b.wait();
System.out.println("main thread got notification");
System.out.println(b.total);
}
}
}
class ThreadB extends Thread
{
int total=0;
public void run()
{
    synchronized(this)
    {
        System.out.println("child thread starts calculation");
        for(int i=1;i<=100;i++)
        {
            total=total+i;
        }
        System.out.println("child thread giving notification");
        this.notify();
    }
}
}

```

Producer consumer problem

Producer thread is responsible to produce items to queue and consumer thread is responsible to consume items from the queue.

If queue is empty

The consumer thread will call wait method and entered into waiting state.

After producing items to the queue producer thread is responsible to call notify method.

Stop () method

We can stop a thread execution by using stop method of thread class.

Syntax

Public void stop()

If we call stop method then immediately thread will entered into dead state. Any way the stop method is deprecated and not recommended to use.

Suspend () and resume () methods

We can suspend a thread by using `suspend ()` method of thread class. Then immediately the thread will be entered into suspended state.

We can resume () a suspended thread by using `resume ()` method of thread class. then suspended thread can continue its execution.

Syntax

`Public void suspend ();`

`Public void resume ();`

Any way these methods are deprecated and not recommended to use.

UNIT-IV

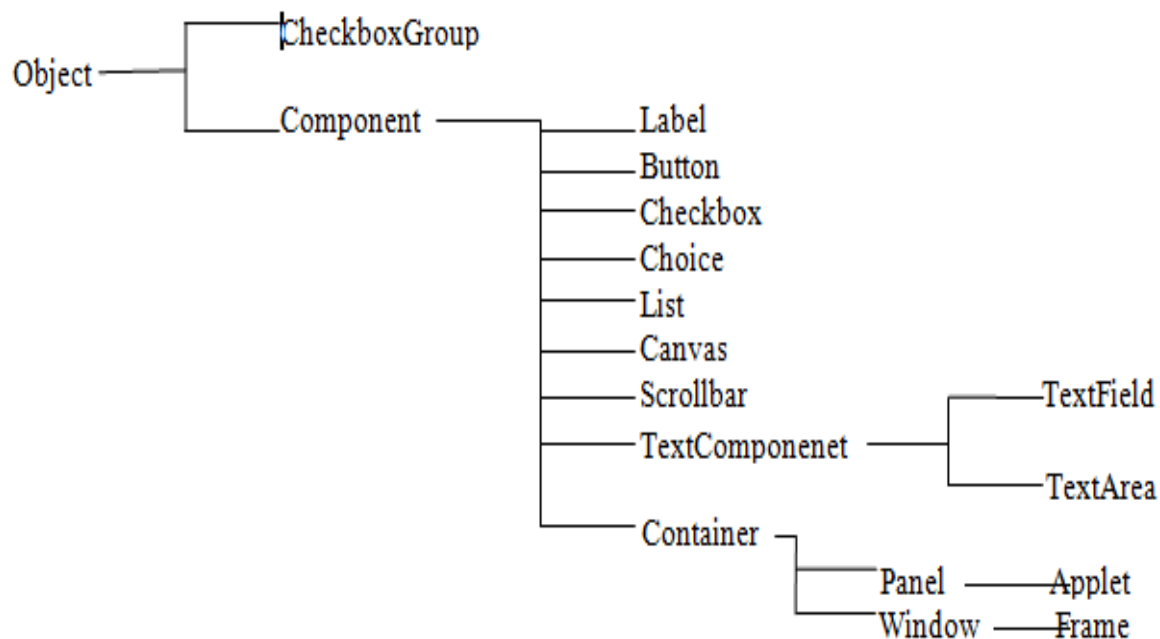
ABSTRACT WINDOWING TOOLKIT (AWT)

The AWT classes are contained in the **java.awt package**.

It is one of Java's largest packages.

It is logically organized in a top-down, hierarchical fashion, it is easier to understand and use

AWT classes Hierarchy



Control Fundamentals

AWT supports the following types of controls.

- Labels
- Push buttons
- Check boxes
- Choice lists

- Lists
- Scroll bars
- Text editing

Label

Label is the user-defined text, which cannot be modified by the end-user.

Constructors

Label() throws HeadlessException

Label(String str) throws HeadlessException

Label(String str, int how) throws HeadlessException

How=> Label.LEFT, Label.RIGHT, or Label.CENTER

To change the text in a label by using the setText() method

void setText(String str)

To obtain the current label by calling getText().

To set the alignment of the string within the label by calling setAlignment().

void setAlignment(int how)

To obtain the current alignment, call getAlignment().

```
import java.awt.*;
public class LabelDemo extends Frame
{
    Label lab1, lab2, lab3;
    public LabelDemo()
    {
        setLayout(new GridLayout(3,1));

        lab1 = new Label("Center aligned text", Label.CENTER);
        lab2 = new Label("Left aligned text");
        lab3 = new Label();

        lab3.setText("Right aligned text");
        lab3.setAlignment(Label.RIGHT);

        add(lab1);
        add(lab2);
        add(lab3);

        lab1.setBackground(Color.yellow);
        lab1.setForeground(Color.blue);
        lab1.setFont(new Font("SansSerif", Font.BOLD+Font.ITALIC, 18));

        System.out.println("lab1 text: " + lab1.getText());
        System.out.println("lab1 alignment: " + lab1.getAlignment());
    }
}
```

```

        setTitle("Labels Do not Have Any Action");
        setSize(450, 200);
        setVisible(true);
    }
    public static void main(String args[ ])
    {
        new LabelDemo();
    }
}

```

Button

Button is a component that contains a label and that generates an event when it is pressed.

Constructors

Button() throws HeadlessException

Button(String str) throws HeadlessException

We can set its label by calling setLabel() method.

we can retrieve its label by calling getLabel() method

Example

```

import java.awt.*;
public class ButtonExample {
    public static void main(String[] args) {
        Frame f=new Frame("Button Example");
        Button b=new Button("Click Here");
        b.setBounds(50,100,80,30);
        f.add(b);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}

```

TextField

The object of a TextField class is a text component that allows the editing of a single line text.

constructors

TextField() throws HeadlessException

TextField(int numChars) throws HeadlessException

TextField(String str) throws HeadlessException

TextField(String str, int numChars) throws HeadlessException

```

import java.awt.*;
class TextFieldExample{
    public static void main(String args[]){
        Frame f= new Frame("TextField Example");
        TextField t1,t2;
        t1=new TextField("Welcome to VCE");
        t1.setBounds(50,100, 200,30);
        t2=new TextField("AWT Notes");
        t2.setBounds(50,150, 200,30);
    }
}

```

```

f.add(t1);
f.add(t2);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
}

```

Layout Manager

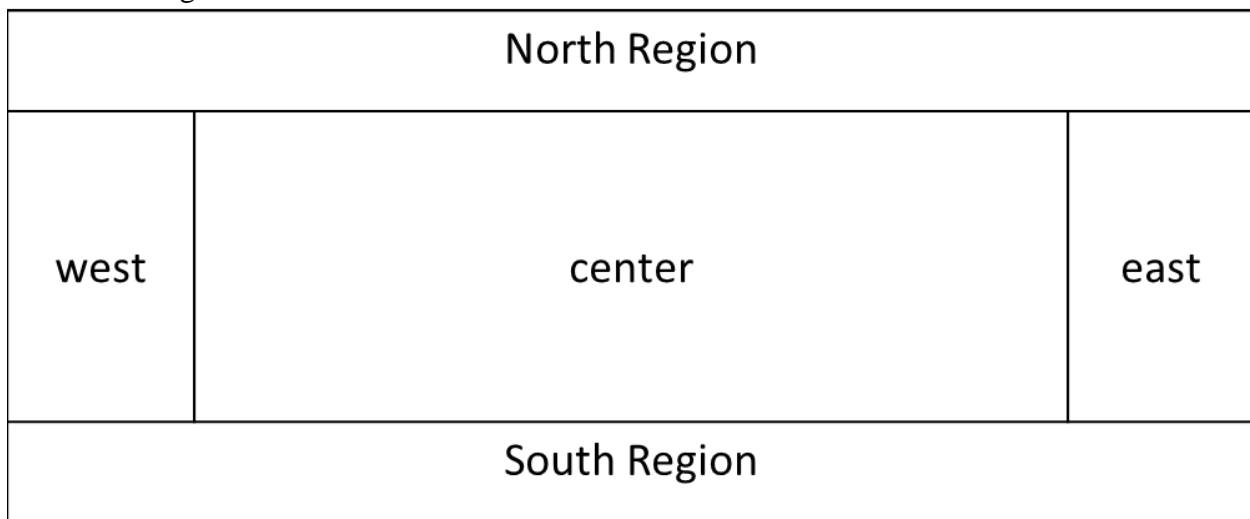
Layout Manager is used to arrange components in a particular manner.

Layout Manager is an interface that is implemented by all classes of layout managers.

The layout manager is set by the **setLayout()** method.

Border Layout Manager

It has five regions.



Each region can contain only one component.

```
Frame f=new Frame("layout");
```

```
f.setLayout(new BorderLayout());
```

If we are creating four buttons and adding them to frame.

```
Button b1=new Button();
```

```
Button b2=new Button();
```

```
Button b3=new Button();
```

```
Button b4=new Button();
```

```
f.add(b1,"North");
```

```
f.add(b2,"South");
```

```
f.add(b3,"East");
```

```
f.add(b4,"West");
```

Flow layout

Used to arrange components in a line , one after the another.

It is a Default layout for panel.

```
Frame f=new Frame();
```

```
Button b1,b2;
```

```
f.setLayout(new Flowlayout());
```

```
f.add(b1);
```

```
f.add(b2);
```

Card layout

Arranges each component in the container as card.

Only one card is visible, and the container acts as a stack of cards.

```
Frame f=new Frame();
```

```
Button b1,b2;
```

```
Cardlayout card;
```

```
Card =new Cardlayout(40,30);
```

```
f.add("card1",b1);
```

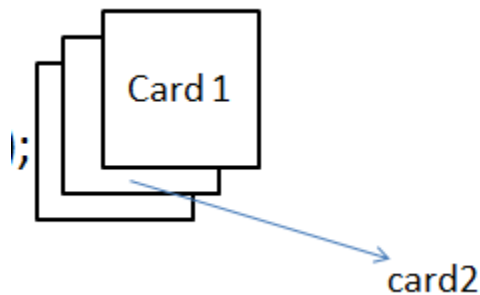
```
f.add("card2",b2);
```

To get which card is to be displayed by using these methods

```
first()
```

```
Last()
```

```
Next()
```



GridLayout

Container is divided into a grid of cells

Each cell accomodates one component.

```
Farne f=new Frame();
```

```
Button b1,b2,b3,b4,b5,b6;
```

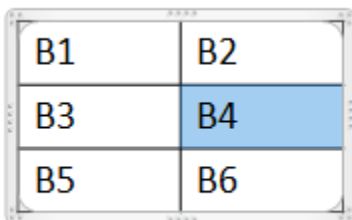
```
f.setlayout(new Gridlayout(3,2));
```

```
f.add(b1);
```

```
:
```

```
:
```

```
f.add(b6);
```



Swings

Introducing Swing

Swing is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*.

With Java 1.1, Swing was used as a separate library.

It is a set of classes which provides many powerful and flexible components for creating graphical user interface. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java. Unlike AWT, Swing provides platform-independent and lightweight components.

Features of Swings

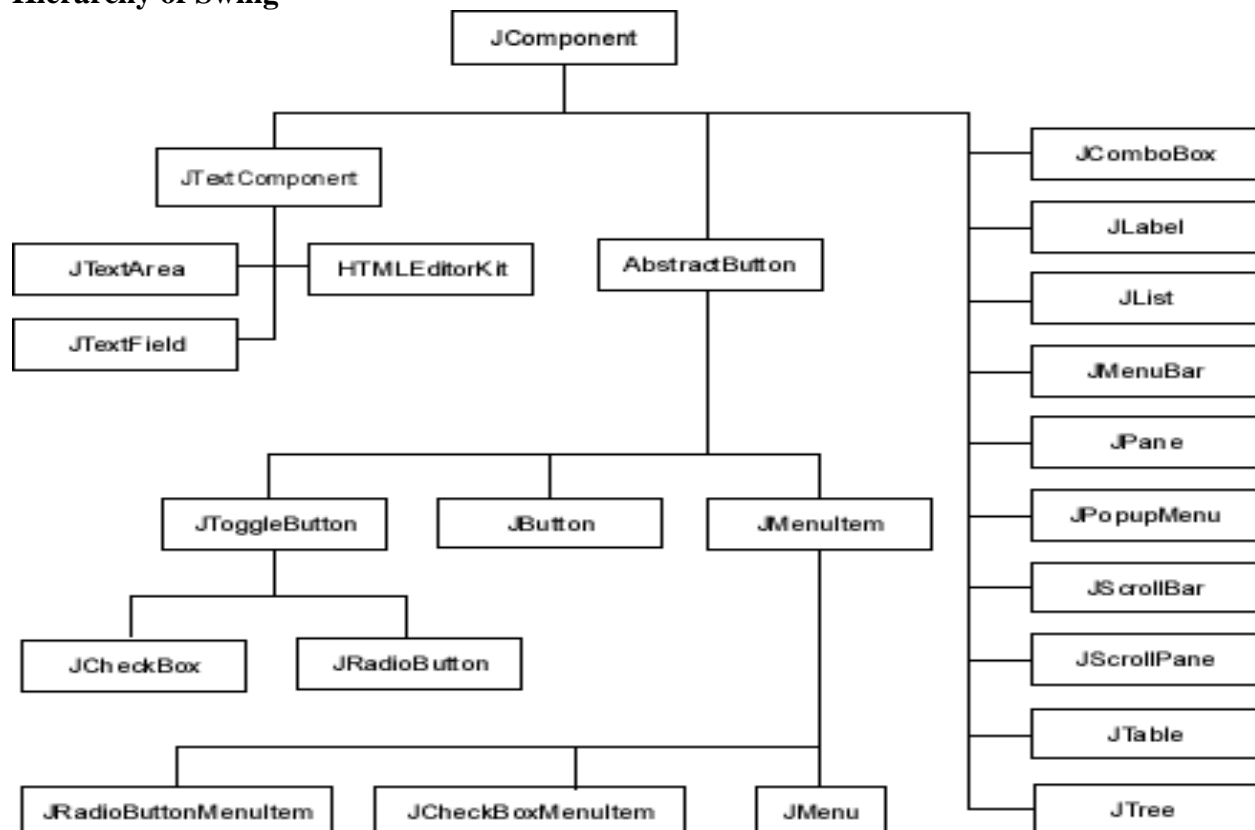
It has two popular features:

1. Lightweight components
2. Pluggable look and feel

Lightweight components

Swing components are lightweight as they are written entirely in Java and do not depend on native peers. They are not restricted to platform-specific appearance like, rectangular or opaque shape.

Hierarchy of Swing



Top-level Containers exist mainly to provide a place for other Swing components to paint themselves. Swing provides four top-level container classes:

1. JFrame - A top-level window with a title and a border.
2. JWindow - As a rule, not very useful. Provides a window with no controls or title.
3. JDialog - The main class for creating a dialog window.

JApplet - Enables applets to use Swing components.

JFrame

```
import javax.swing.*;
class SwingDemo
{
    SwingDemo()
    {

        JFrame jfrm = new JFrame("A Simple Swing Application");

        jfrm.setSize(275, 100);
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel jlab = new JLabel(" Swing means powerful GUIs.");

        jfrm.add(jlab);

        jfrm.setVisible(true);
    }
    public static void main(String args[])
    { // Create the frame on the event dispatching thread.
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                new SwingDemo();
            }
        });
    }
}
```

JApplet

The JApplet class extends the Applet class.

Example

SimpleApplet.java

```
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;

public class SimpleApplet extends JApplet {
    public void init() {
        JPanel p = new JPanel();
        p.setLayout(new GridLayout(2, 2, 2, 2));
        p.add(new JLabel("Username"));
        p.add(new JTextField());
        p.add(new JLabel("Password"));
    }
}
```

```

p.add(new JPasswordField());
Container content = getContentPane();
content.setLayout(new GridBagLayout()); // Used to center the panel
content.add(p);
}
}

```

```

<html>
<head><title>japplet example</title></head>
<APPLET CODE = SimpleApplet WIDTH = 300 HEIGHT = 200>
< PARAM NAME = "bogus" VALUE = "just testing">

< /APPLET>
</html>

```

Light Weight Containers

- Lightweight containers do inherit JComponent.
- Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container.
- One of the examples of lightweight container is JPanel.

Create a Swing Applet

EventJApplet.java

```

import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
public class EventJApplet extends JApplet implements ActionListener{
    JButton b;
    JTextField tf;
    public void init(){

        tf=new JTextField();
        tf.setBounds(30,40,150,20);

        b=new JButton("Click");
        b.setBounds(80,150,70,40);

        add(b);add(tf);
        b.addActionListener(this);

        setLayout(null);
    }

    public void actionPerformed(ActionEvent e){
        tf.setText("Welcome");
    }
}

```



```
}
```

Myapplet.html

```
<html>
<body>
<applet code="EventJApplet.class" width="300" height="300">
</applet>
</body>
</html>
```

Swing Components

JToggleButton

Swing provides a variant of push button called toggle button which has two states: pushed and released.

Toggle button is an object of JToggleButton class.

constructors

JToggleButton ()

JToggleButton(String string)

JToggleButton(String string, boolean state)

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class JToggleButtonExample extends JApplet
{
    public JToggleButtonExample()
    {
        Container Cntnr = getContentPane();
        Icon Icn = new ImageIcon("Button.jpg");
        JToggleButton TglOne = new JToggleButton(Icn);
        JToggleButton TglTwo = new JToggleButton(Icn, true);
        JToggleButton TglThree = new JToggleButton("Toggle It!");
        JToggleButton TglFour = new JToggleButton("Toggle It!", Icn);
        JToggleButton TglFive = new JToggleButton("Toggle It!",Icn,true);
        Cntnr.setLayout(new FlowLayout());
        Cntnr.add(TglOne);
        Cntnr.add(TglTwo);
        Cntnr.add(TglThree);
        Cntnr.add(TglFour);
        Cntnr.add(TglFive);
    }
}
/*<APPLET CODE="JToggleButtonExample.class" WIDTH=400 HEIGHT=400>
</APPLET>*/
```

JTabbedPane

The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon. It inherits JComponent class.

Constructors

JTabbedPane()

JTabbedPane(int tabPlacement)

JTabbedPane(int tabPlacement, int tabLayoutPolicy)

```
import java.awt.*;
```

```
import javax.swing.*;
```

```
/*
```

```
<applet code="JTPDemo.class" width="350" height="300">
```

```
</applet>*/
```

```
public class JTPDemo extends JApplet
```

```
{
```

```
    public void init()
```

```
    {
```

```
        JTabbedPane jt = new JTabbedPane();
```

```
        jt.add("Colors", new CPanel());
```

```
        jt.add( "Fruits", new FPanel());
```

```
        jt.add("Vitamins", new VPanel( ) );
```

```
        getContentPane().add(jt);
```

```
    }
```

```
}
```

```
class CPanel extends JPanel
```

```
{
```

```
    public CPanel()
```

```
    {
```

```
        JCheckBox cb1 = new JCheckBox("Red");
```

```
        JCheckBox cb2 = new JCheckBox("Green");
```

```
        JCheckBox cb3 = new JCheckBox("Blue");
```

```
        add(cb1); add(cb2); add(cb3);
```

```
    }
```

```
}
```

```
class FPanel extends JPanel
```

```
{
```

```
    public FPanel()
```

```
    {
```

```
        JComboBox cb = new JComboBox();
```

```
        cb.addItem("Apple");
```

```
        cb.addItem("Mango");
```

```
        cb.addItem("Pineapple");
```

```
        add(cb);
```

```
    }
```

```
}
```

```
class VPanel extends JPanel
```

```

{
    public VPanel()
    {
        JButton b1 = new JButton("Vit-A");
        JButton b2 = new JButton("Vit-B");
        JButton b3 = new JButton("Vit-C");
        add(b1); add(b2); add(b3);
    }
}

```

JTree

The JTree class is used to display the tree structured data or hierarchical data. JTree is a complex component. It has a 'root node' at the top most which is a parent for all nodes in the tree. It inherits JComponent class.

Constructors

```

JTree()
JTree(Object[] value)
JTree(TreeNode root)

```

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*; // for DefaultMutableTreeNode and JTree
public class JTreeDemo extends JApplet
{
    JTree jt;
    JTextField jtf;
    public void init()
    {
        Container c = getContentPane();
        c.setLayout(new BorderLayout());
        // this is the root node and top in the hierarchy
        DefaultMutableTreeNode rootnode = new DefaultMutableTreeNode("Sports");
        DefaultMutableTreeNode anode = new DefaultMutableTreeNode("Air games");
        rootnode.add(anode); // becomes a file to rootnode
        // create ogames node add to the rootnode (becomes child node of
rootnode)
        DefaultMutableTreeNode ogames = new DefaultMutableTreeNode("OutDoor Games");
        DefaultMutableTreeNode bnode = new DefaultMutableTreeNode("Basket ball");
        DefaultMutableTreeNode vnode = new DefaultMutableTreeNode("Volley ball");
        ogames.add(bnode); // becomes file to ogames
        ogames.add(vnode); // becomes file to ogames
        rootnode.add(ogames); // add ogames to rootnode
// create igames node add to the rootnode(becomes child node of rootnode)
        DefaultMutableTreeNode igames = new DefaultMutableTreeNode("Indoor Games");

```

```

DefaultMutableTreeNode cnode = new DefaultMutableTreeNode("Carroms");
DefaultMutableTreeNode tnode = new DefaultMutableTreeNode("Table Tennis");
igames.add(cnode);           // becomes file to igames
igames.add(tnode);           // becomes file to igames
rootnode.add(igames);         // add igames to rootnode
                               // this node becomes child node to igames
DefaultMutableTreeNode snode = new DefaultMutableTreeNode("Skill Games");
DefaultMutableTreeNode shnode = new DefaultMutableTreeNode("Shooting");
DefaultMutableTreeNode banode = new DefaultMutableTreeNode("Bar Dancing");
snode.add(banode);            // becomes a file to snode
igames.add(snode);            // becomes a file to snode
snode.add(shnode);            // snode is the child node to igames
jt = new JTree(rootnode);     // add root node to the JTree
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(jt , v , h);
c.add(jsp, "Center");         // add scroll pane to the container

}
}
/*
<applet code="JTreeDemo.class" width="350" height="300">
</applet>*/

```

EVENT HANDLING

Event

An event in Java is an object that is created when something changes within a graphical user interface. If a user clicks on a button, clicks on a combo box, or types characters into a text field, etc., then an event triggers, creating the relevant event object.

Delegation Event Model

a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns.

The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.

In the delegation event model, listeners must register with a source in order to receive an event notification.

Events are signals which are fired when the state of a component is changed (eg: when a button is pressed, when a menu is pressed etc.). In the event of a signal firing it is necessary for us to handle the event based on our requirements. For example you would want to open a new window or close it when a button is pressed, or you would want to list a menu when a menu box

is activated (pressed).

Sources:

The previous paragraph only gives you a gist of what happens when a component is activated. Actually when the internal state of the component is modified a **source** is generated. This is nothing but a source to the event.

Listeners:

A single component can take events from different sources. For Example an applet can have sources from the keyboard or from the mouse. So you should make the applet be ready to receive the events from the different sources. This is done by the **Listeners** which are nothing but interfaces with abstract methods which could be implemented on generation of the corresponding event.

Event Handling:

The actions that have to be performed on the component listening to an event like a mouse clicked on an applet are specified. This is called **Event Handling**.

Component type	Events supported by this component
Adjustable	AdjustmentEvent
Applet	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Button	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Canvas	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Checkbox	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
CheckboxMenuItem	ActionEvent, ItemEvent
Choice	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Component	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Container	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Dialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
FileDialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Frame	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Label	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
List	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ItemEvent, ComponentEvent

Menu	ActionEvent
MenuItem	ActionEvent
Panel	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
PopupMenu	ActionEvent
Scrollbar	AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
ScrollPane	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextArea	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextComponent	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextField	ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Window	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

Once you know which events a particular component supports, you don't need to look anything up to react to that event. You simply:

Take the name of the event class and remove the word " Event." Add the word " Listener" to what remains. This is the listener interface you need to implement in your inner class.

Implement the interface above and write out the methods for the events you want to capture. For example, you might be looking for mouse movements, so you write code for the `mouseMoved()` method of the `MouseMotionListener` interface. (You must implement the other methods, of course, but there's a shortcut for that which you'll see soon.)

Create an object of the listener class in step 2. Register it with your component with the method produced by prefixing " add" to your listener name. For example, `addMouseMotionListener()`.

To finish what you need to know, here are the listener interfaces:

Listener Interface Window Adapter	Methods in interface
ActionListener	<code>actionPerformed(ActionEvent)</code>
AdjustmentListener	<code>AdjustmentValueChanged</code> <code>(AdjustmentEvent)</code>
ComponentListener ComponentAdapter	<code>componentHidden(ComponentEvent)</code> <code>componentShown(ComponentEvent)</code> <code>componentMoved(ComponentEvent)</code>

	componentResized(ComponentEvent)
ContainerListener	componentAdded(ContainerEvent)
ContainerAdapter	componentRemoved(ContainerEvent)
FocusListener	focusGained(FocusEvent)
FocusAdapter	focusLost(FocusEvent)
KeyListener	keyPressed(KeyEvent)
KeyAdapter	keyReleased(KeyEvent)
	keyTyped(KeyEvent)
MouseListener	mouseClicked(MouseEvent)
MouseAdapter	mouseEntered(MouseEvent)
	mouseExited(MouseEvent)
	mousePressed(MouseEvent)
	mouseReleased(MouseEvent)
MouseMotionListener	mouseDragged(MouseEvent)
MouseMotionAdapter	mouseMoved(MouseEvent)
WindowListener	windowOpened(WindowEvent)
WindowAdapter	windowClosing(WindowEvent)
	windowClosed(WindowEvent)
	windowActivated(WindowEvent)
	windowDeactivated(WindowEvent)
	windowIconified(WindowEvent)
	windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)
TextListener	textValueChanged(TextEvent)

A simple example will make this clear.

```
//: Button2New.java
/ Capturing button presses import java.awt.*;
import java.awt.event.*; // Must add this import
java.applet.*;

public class Button2New extends Applet { Button
    b1 = new Button("Button 1"),
    b2 = new Button("Button 2"); public
    void init() {
        b1.addActionListener(new B1());
        b2.addActionListener(new B2()); add(b1);
        add(b2);
    }

    class B1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            getAppletContext().showStatus("Button 1");
        }
    }
}
```

```

class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        getAppletContext().showStatus("Button 2");
    }
}

```

Using listener adapters for simplicity

In the previous table, you can see that some listener interfaces have only one method. These are trivial to implement since you'll implement them only when you want to write that particular method. However, the listener interfaces that have multiple methods could be less pleasant to use. For example, something you must always do when creating an application is provide a WindowListener to the Frame so that when you get the windowClosing() event you can call System.exit(0) to exit the application. But since WindowListener is an interface, you must implement all of the other methods even if they don't do anything. This can be annoying.

To solve the problem, each of the listener interfaces that have more than one method are provided with adapters, the names of which you can see in the table above. Each adapter provides default methods for each of the interface methods. (Alas, WindowAdapter does not have a default windowClosing() that calls System.exit(0).) Then all you need to do is inherit from the adapter and override only the methods you need to change. For example, the typical WindowListener you'll use looks like this:

```

class MyWindowListener extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}

```

The whole point of the adapters is to make the creation of listener classes easy.

There is a downside to adapters, however, in the form of a pitfall. Suppose you write a WindowAdapter like the one above:

```

class MyWindowListener extends WindowAdapter
{
    public void WindowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}

```



```
}  
}
```

This doesn't work, but it will drive you crazy trying to figure out why, since everything will compile and run fine – except that closing the window won't exit the program. Can you see the problem? It's in the name of the method: `WindowClosing()` instead of `windowClosing()`. A simple slip in capitalization results in the addition of a completely new method. However, this is not the method that's called when the window is closing, so you don't get the desired results.

UNIT-V

The Stream Classes:

Java's stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

InputStream and **OutputStream** are designed for byte streams.

Reader and **Writer** are designed for character streams.

The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.

Byte Stream Classes:

- **InputStream**

InputStream is an abstract class that defines Java's model of streaming byte input. All of the methods in this class will throw an **IOException** on error conditions.

Some of the methods in this class are:

- `int available()`: Returns the number of bytes of input currently available for reading.
- `void close()`: Closes the input source. Further read attempts will generate an **IOException**.
- `int read()`: Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
- `int read(byte buffer[])`: Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.

- **OutputStream**

OutputStream is an abstract class that defines streaming byte output. All of the methods in this class return a **void** value and throw an **IOException** in the case of errors.

Some of the methods in this class are:

- `void close()`: Closes the output stream. Further write attempts will generate an **IOException**.
- `void write(int b)`: Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write()** with expressions without having to cast them back to **byte**.
- `void write(byte buffer[])`: Writes a complete array of bytes to an output stream.

- **FileInputStream**

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Its two most common constructors are shown here:

`FileInputStream(String filepath)`

`FileInputStream(File fileObj)`

Either can throw a **FileNotFoundException**. Here, *filepath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

- **FileOutputStream**

FileOutputStream creates an **OutputStream** that you can use to write bytes to a file. Its most commonly used constructors are shown here:

`FileOutputStream(String filePath)`

`FileOutputStream(File fileObj)`

`FileOutputStream(String filePath, boolean append)`

`FileOutputStream(File fileObj, boolean append)`

They can throw a **FileNotFoundException** or a **SecurityException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is true, the file is opened in append mode

The Character Streams:

While the byte stream classes provide sufficient functionality to handle any type of I/O

operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the —write once, run anywhere philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed. As explained earlier, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.

- **Reader**

Reader is an abstract class that defines Java's model of streaming character input. All of the methods in this class will throw an **IOException** on error conditions. Table 17-3 provides a synopsis of the methods in **Reader**.

Writer

Writer is an abstract class that defines streaming character output. All of the methods in this class return a **void** value and throw an **IOException** in the case of errors. Table 17-4 shows a synopsis of the methods in **Writer**.

- **FileReader**

The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

`FileReader(String filePath)`

`FileReader(File fileObj)`

- **FileWriter**

FileWriter creates a **Writer** that you can use to write to a file. Its most commonly used constructors are shown here:

```
FileWriter(String filePath)  
FileWriter(String filePath, boolean append)  
FileWriter(File fileObj)  
FileWriter(File fileObj, boolean append)
```

They can throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj*

is a **File** object that describes the file. If *append* is **true**, then output is appended to the end of the file.

Applets

Applets, are Java programs that are developed over the World Wide Web and executed by a Web browser on the reader's machine. Applets depend on a Java-capable browser in order to run.

Creating a Java Applet:

Creating applets is different from creating a simple application, because Java applet rules for how they behave. Because of these special rules for applets in many cases (particularly the simple ones), creating an applet may be more complex than creating an application. For example, to do a simple Hello World applet, instead of merely being able to print a message, you have to create an applet to make space for your message and then use graphics operations to paint the message to the screen.

In the next example, you create that simple Hello World applet, place it inside a Web page, and view the result. First, you set up an environment so that your Java-capable browser can find your HTML files and your applets. Much of the time, you'll keep your HTML files and your applet code in the same directory.

Program:

```
import java.awt.Graphics;  
public class HelloWorldApplet extends  
    java.applet.Applet  
{  
    public void paint(Graphics g)  
    {  
        g.drawString("Hello World!", 5, 25);  
    }  
}
```

Save the file just like with Java applications, give your file a name that has the same name as the class. In this case, the filename would be HelloWorldApplet.java.

Features of Applets:

- ❖ The import line at the top of the file is somewhat analogous to an #include statement in C; it enables this applet to get access to the JDK classes for creating applets and for drawing graphics on the screen.
- ❖ The paint() method displays the content of the applet onto the screen. Here, the

string Hello World gets drawn. Applets use several standard methods to take the place of main(), which include init() to initialize the applet, start() to start it running, and paint() to display it to the screen.

Now, compile the applet just as you did the application, using javac, the Java compiler.

```
javac HelloWorldApplet.java
```

Again, just as for application, you should now have a file called HelloWorldApplet.class in your directory.

To include an applet in a Web page, you refer to that applet in the HTML code for that Web page. Here, you create a very simple HTML file in the directory.

```
<HTML>
<HEAD>
  <TITLE>
    Hello to Everyone!
  </TITLE>
</HEAD>
<BODY>
  <P> My Java applet says:
  <APPLET CODE="HelloWorldApplet.class" WIDTH = 150 HEIGHT =
  25> </APPLET>
</BODY>
</HTML>
```

Java Applications

Java applications are more general programs written in the Java language. Java applications don't require a browser to run, and in fact, Java can be used to create all the kinds of applications that you would normally use a more conventional programming language to create. HotJava itself is a Java application.

Creating a Java Application:

As with all programming languages, your Java source files are created in a plain text editor, or in an editor that can save files in plain ASCII without any formatting characters. On UNIX, emacs, pico, or vi will work; on Windows, Notepad or DOS Edit are both text editors.

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

This program has two main parts:

- All the program is enclosed in a class definition – here, a class called HelloWorld.
- The body of the program is continued in a method(function) called main(). In Java applications, as in a C or C++ program, main() is the first method (function) that runs when the program is executed.

Once you finish typing the program, save the file. Most of the time, Java source files are named the same name as the class they define, with an extension of .java. This file should therefore be called HelloWorld.java. Now, let's compile the source file using the Java compiler. In Sun's JDK, the Java compiler is called javac. To compile your Java program, make sure the javac program is in your execution path and type javac followed by the name of your source file: javac HelloWorld.java.

The compiler should compile the file without any errors. If you get errors, go back and make sure that you've typed the program exactly. When the program compiles without errors, you end up with a file called HelloWorld.class, in the same directory as your source file. This is your Java bytecode file. You can then run that bytecode file using the Java interpreter. This is your Java interpreter is called simply java. Make sure the java program is in your path and type java followed by the name of the file without the .class extension: java HelloWorld

If your program was typed and compiled correctly, you should get the string "Hello World!" printed to your screen as a response.