Course Content

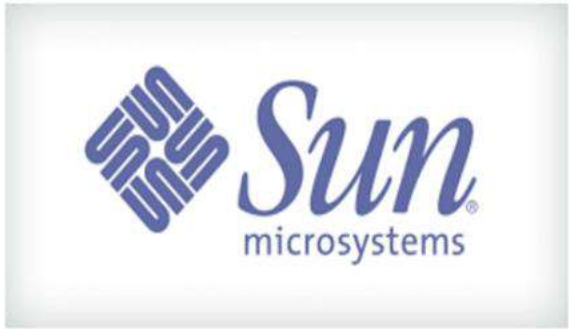
- Introduction and Basics of JAVA
- Class and Object Methodologies
- OOP flavors
- Exception Handling
- Threads in JAVA
- File and I/O
- Logical Data Storage
- JDBC –ODBC Concepts and XML / JSON Parsing

Introduction and Basics of JAVA

- History and overview of JAVA
- JDK , JRE , JVM ,Byte Code .
- Installation of JDK , Eclipse IDE/NetBeans IDE
- Overview of IDEs
- Data Types ,Variables , Keywords ,Operators
- Control Statements ,Looping Statements
- Arrays

JAVA History

- With Java, Sun Microsystems established the first programming language that wasn't tied to any particular operating system or microprocessor.
- Earlier known as "Oak" , later changed to JAVA.
- Recently owned by Oracle , as Sun Microsystems acquired Oracle by in January 2010.



Four Platforms:

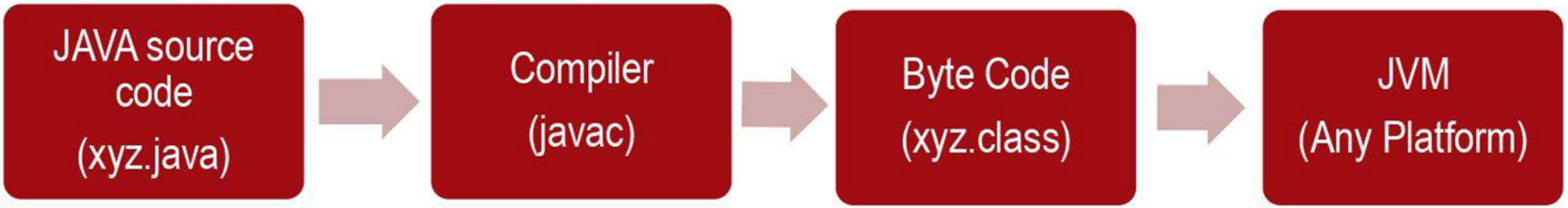
- JAVA SE:
  - It defines everything from the basic types and objects of the Java programming language to high-level classes that are used for networking, security, database access, graphical user interface (GUI) development, and XML parsing.
- JAVA EE:
  - Built on top of the Java SE platform , provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications.
- JAVA ME:
  - Provides an API and a small-footprint virtual machine for running Java programming language applications on small devices, like mobile phones. The API is a subset of the Java SE API, along with special class libraries useful for small device application development.
- JAVA FX:
  - Platform for creating rich internet applications using a lightweight user-interface API. JavaFX applications use hardware-accelerated graphics and media engines to take advantage of higher-performance clients and a modern look-and-feel as well as high-level APIs for connecting to networked data sources.

One must know about them :

- JDK
  - Development kit consist of JAVA classes and compiler.
- JRE
  - A runtime environment to run JAVA Programs.
- JVM
  - A tool on which JAVA program runs – (which makes JAVA platform independent) .

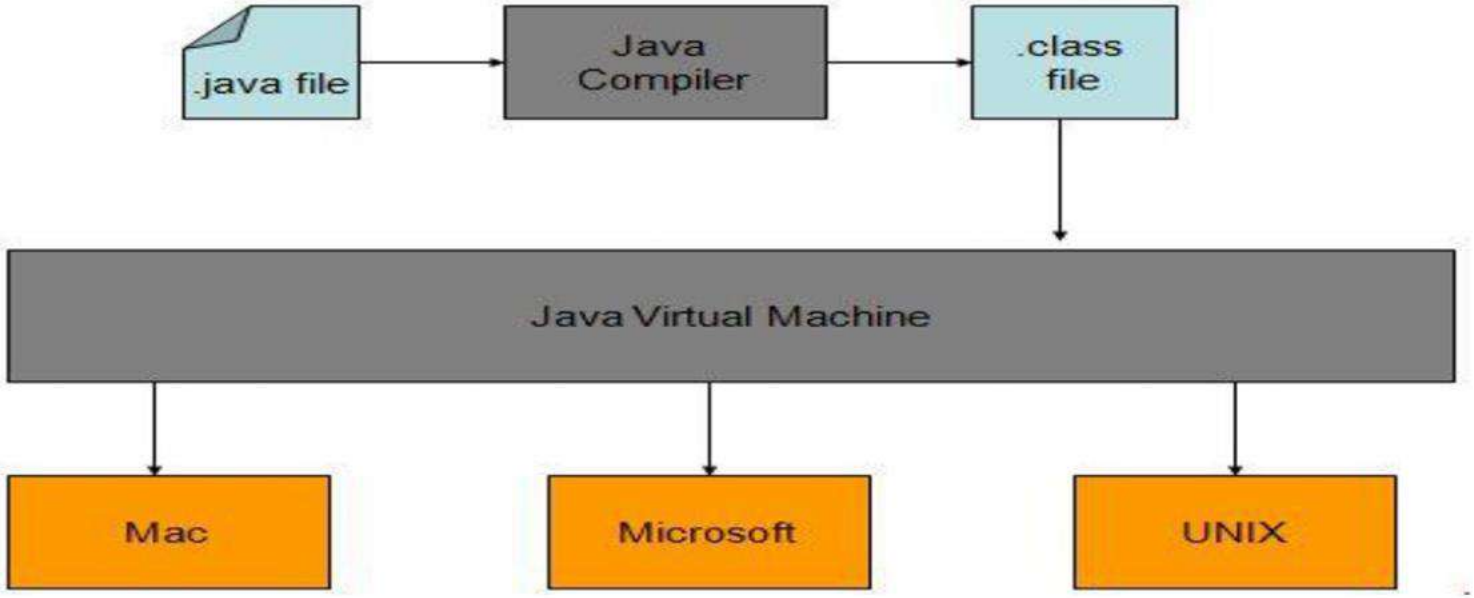
This “TRIO” makes your JAVA program run on any machine.

This is how JAVA does it:



Here JVM is not platform independent but it makes “byte code “ independent to platforms , make sense ?

Platform Independent Byte Code



These machines have their own JVMs installed , which can run same byte code on them.

Setting up the Environment

- JDK can be downloaded form the following link
  - http://www.oracle.com/technetwork/java/javase/downloads/index.html
  - How to install JDK
    - JDK installation Guide

- IDEs (Integrated Development Environment)
  - Eclipse
    - Eclipse Installation Guide
  - NetBeans
    - Netbeans Installation Guide

Data Types, Variables and Keywords

- Basically similar to C or C++ data types.
- Two types of data types:
  - Primitive
  - Non Primitive.

- Primitive – also known as basic data types
  - int , float , char , boolean etc.
- Non Primitive – Derived from primitive data types.(reference)
  - Arrays , List , objects etc.

DataTypes

Data type	Bytes	Min Value	Max Value	Literal Values
byte	1	-2^7	$2^7 - 1$	123
short	2	-2^{15}	$2^{15} - 1$	1234
int	4	-2^{31}	$2^{31} - 1$	12345, 086, 0x675
long	8	-2^{63}	$2^{63} - 1$	123456
float	4	-	-	1.0
double	8	-	-	123.86
char	2	0	$2^{16} - 1$	'a', '\n'
boolean	-	-	-	true, false

Variables

- A variable provides us with named storage that our programs can manipulate.
- Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

    - int data = 25;
    - String name = “java”;

Rules for Identifiers (Variables)

- All variable names must begin with a letter of the alphabet, an underscore, or (_), or a dollar sign ($). The convention is to always use a letter of the alphabet. The dollar sign and the underscore are discouraged.
- After the first initial letter, variable names may also contain letters and the digits 0 to 9. No spaces or special characters are allowed.
- The name can be of any length, but don't get carried away. Remember that you will have to type this name.
- Uppercase characters are distinct from lowercase characters. Using ALL uppercase letters are primarily used to identify constant variables. Remember that variable names are case-sensitive.
- You cannot use a java keyword (reserved word) for a variable name.

Example

Samples of acceptable variable names: **YES**	Samples of unacceptable variable names: **NO**
Grade	Grade(Test)
GradeOnTest	GradeTest#1
Grade_On_Test	3rd_Test_Grade
GradeTest	Grade Test (has a space)

Keywords

Category	Keywords
Access modifiers	**private, protected, public**
Class, method, variable modifiers	**abstract, class, extends, final, implements, interface, native, new, static, strictfp, synchronized, transient, volatile**
Flow control	**break, case, continue, default, do, else, for, if, instanceof, return, switch, while**
Package control	**import, package**
Primitive types	**boolean, byte, char, double, float, int, long, short**
Error handling	**assert, catch, finally, throw, throws, try**
Enumeration	**enum**
Others	**super, this, void**
Unused	**const, goto**

Operators

An operator performs a particular operation on the operands it is applied on.

- Types
  - Assignment Operators
  - Arithmetic Operators
  - Unary Operators
  - Equality Operators
  - Relational Operators
  - Conditional Operators
  - instaceof Operator
  - Bitwise Operators
  - Shift Operators

Operators

- Assignment Operator

Operator	Description	Example
=	Assignment	int i = 10; int j = i;

- Arithmetic Operators

Operator	Description	Example
+	Addition	int i = 8 + 9; byte b = (byte) 5+4;
-	Subtraction	int i = 9 - 4;
*	Multiplication	int i = 8 * 6;
/	Division	int i = 10 / 2;
%	Remainder	int i = 10 % 3;

Operators

- Unary Operators

Operator	Description	Example
+	Unary plus	int i = +1;
-	Unary minus	int i = -1;
++	Increment	int j = i++;
--	Decrement	int j = i--;
!	Logical Not	boolean j = !true;

- Equality Operators

Operator	Description	Example
==	Equality	If (i==1)
!=	Non equality	If (i != 4)

Operators

- Relational Operators

Operator	Description	Example
>	Greater than	if (x > 4)
<	Less than	if (x < 4)
>=	Greater than or equal to	if (x >= 4)
<=	Less than or equal to	if (x <= 4)

- Conditional Operators

Operator	Description	Example
&&	Conditional and	If (a == 4 && b == 5)
||	Conditional or	If (a == 4 || b == 5)

Operators

- instanceof Operator

Operator	Description	Example
instanceof	Instamce of	If (john instance of person)

- Bitwise Operators

Operator	Description	Example
&	Bitwise and	001 & 111 = 1
|	Bitwise or	001 | 110 = 111
^	Bitwise ex-or	001 ^ 110 = 111
~	Reverse	~011 = -10

- Shift Operators

Operator	Description	Example
>>	Right shift	4 >> 1 = 100 >> 1 = 010 = 2
<<	Left Shift	4 << 1 = 100 << 1 = 1000 = 8
>>>	Unsigned Right shift	4 >>> 1 = 100 >>> 1 = 010 = 2

Conditional Statements (if - else)

- **if-else**

Syntax	Example

if (<condition-1>) {
// logic for true condition-1 goes here
} else if (<condition-2>) {
// logic for true condition-2 goes here
} else {
// if no condition is met, control comes here
}
	int a = 10;
	if (a < 10) {
	System.out.println("Less than 10");
	} else if (a > 10) {
	System.out.pritln("Greater than 10");
	} else {
	System.out.println("Equal to 10");
	}
	
	Result: Equal to 10s

Conditional Statements (Switch)

Syntax	Example

switch (<value>) {
  case <a>:
  // stmt-1
  break;
  case <b>:
  //stmt-2
  break;
  default:
  //stmt-3
	int a = 10;
	switch (a) {
	  case 1:
	  System.out.println(“1”);
	  break;
	  case 10:
	  System.out.println(“10”);
	  break;
	  default:
	  System.out.println(“None”);
	
	Result: 10

Flow Control (While , do..While)

- **do-while**

Syntax	Example

do {
// stmt-1
} while (<condition>);
	int i = 0;
	do {
	System.out.println("In do"); i++;
	} while (i < 10);
	
	Result: Prints "In do" 11 times

- **while**

Syntax	Example

while (<condition>) {
//stmt
}
	int i = 0;
	while (i < 10) {
	System.out.println("In while"); i++;
	}
	
	Result: "In while" 10 times

Flow Control (for loop)

- **for**

Syntax	Example

for (initialize; condition; expression)
{
  // stmt
}
	for (int i = 0; i < 10; i++)
	{
	  System.out.println(“In for”);
	}
	
	Result: Prints “In do” 10 times

Arrays in JAVA

Array Declaration

```
int myArray[ ];
int[ ] myArray;
double[ ][ ] doubleArray;
```

Array Construction

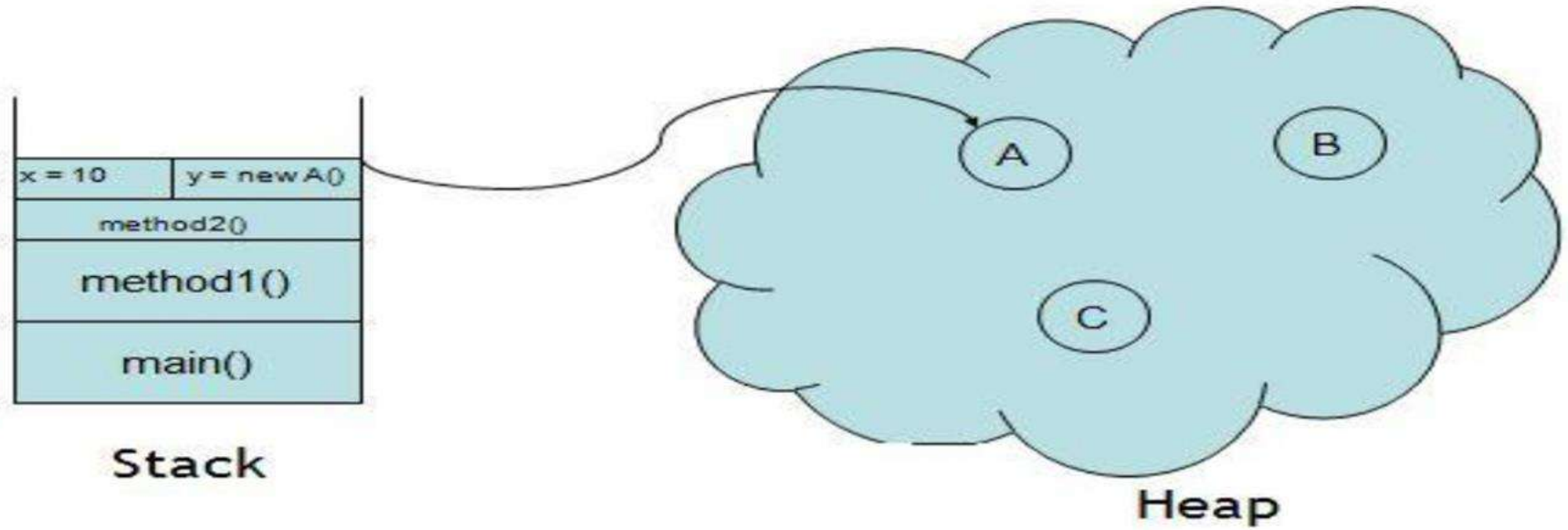
```
int[ ] myArray = new int[5];
int[ ][ ] twoDimArray = new int[5][4]
```

Array Initialization

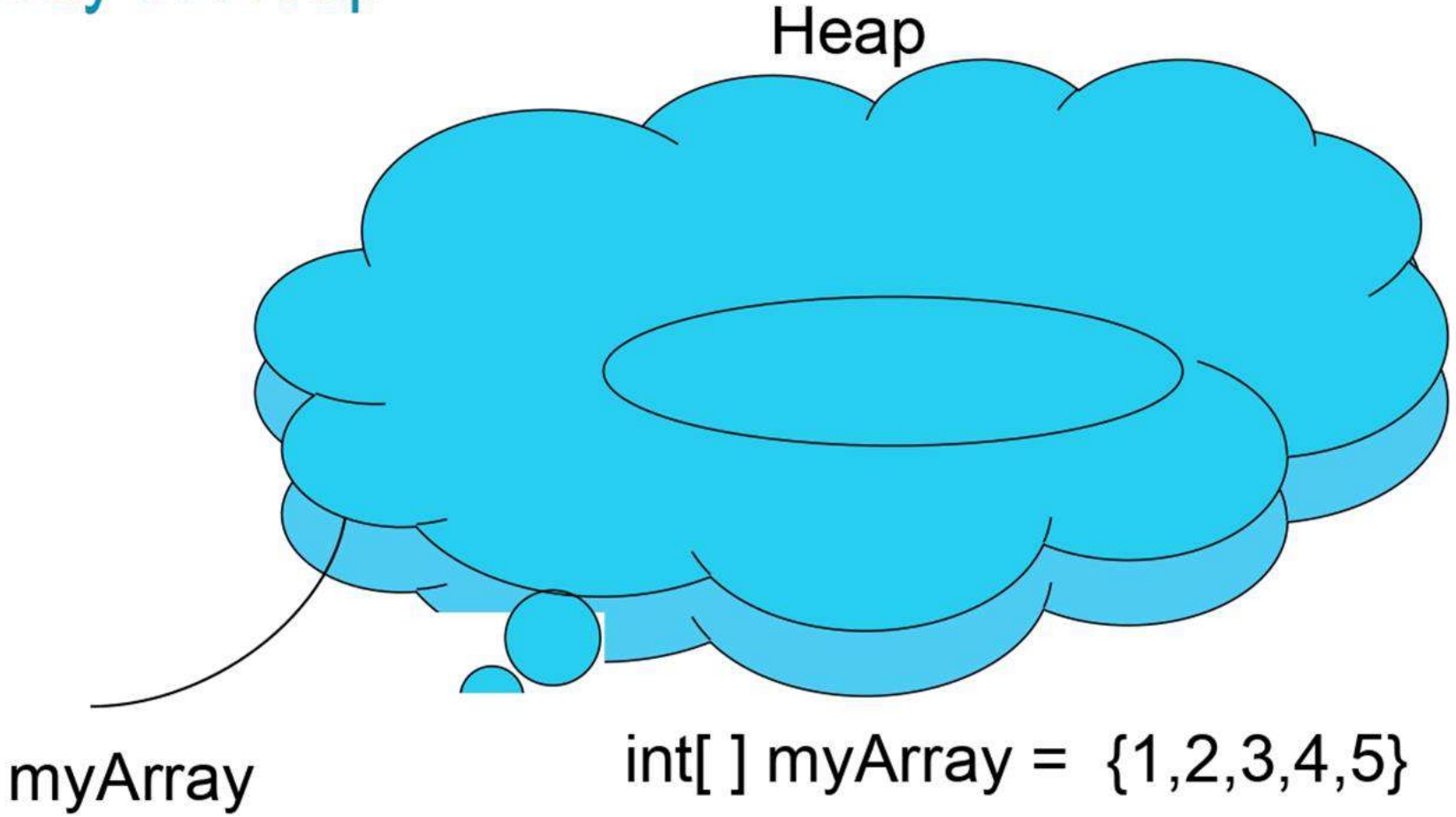
```
int[ ] myArray = new int[5];
for (int I = 0; I < 5; i++) {
   myArray[i] = i++;
}

int[5] myArray = {1,2,3,4,5};
```

Stack vs Heap

- Main difference between heap and stack is that stack memory is used to store local variables and function call, while heap memory is used to store objects in Java.



Array in Heap



Let's Understand this …

```
class TestDemo
{
   public static void main(String[]  args)
   {
      System.out.println("Here we complete the first unit");
   }
}
```

Here We Are

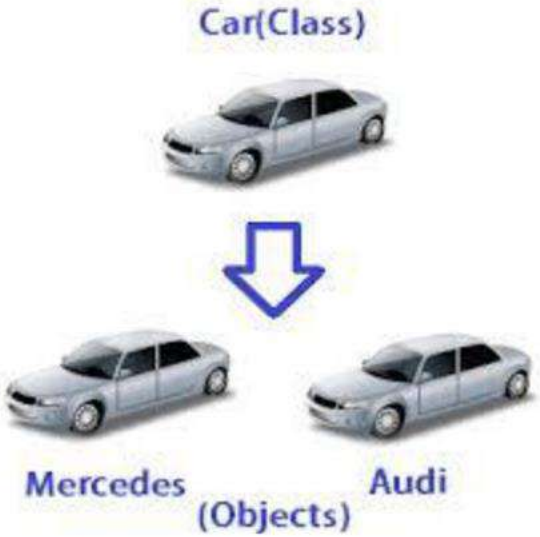
- Introduction and Basics of JAVA
- Class and Object Methodologies
- OOP flavors
- Exception Handling
- Threads in JAVA
- File and I/O
- Logical Data Storage
- JDBC –ODBC Concepts and XML / JSON Parsing

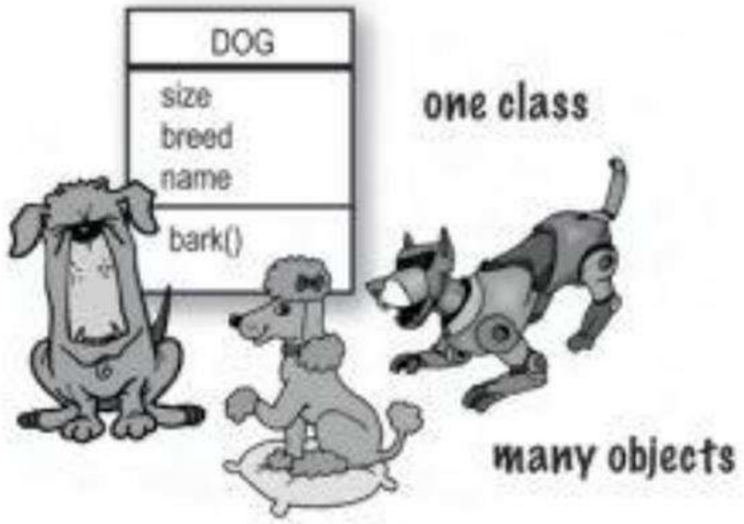
we will go through..

- Classes and Objects
- Constructors
- Destructors and Garbage collection
- this keyword
- Methods in JAVA
- Overloading , Overriding
- Static keyword and methods

Class and Object

- JAVA is pure object oriented language .
- Object ?????
- Let's understand it ..





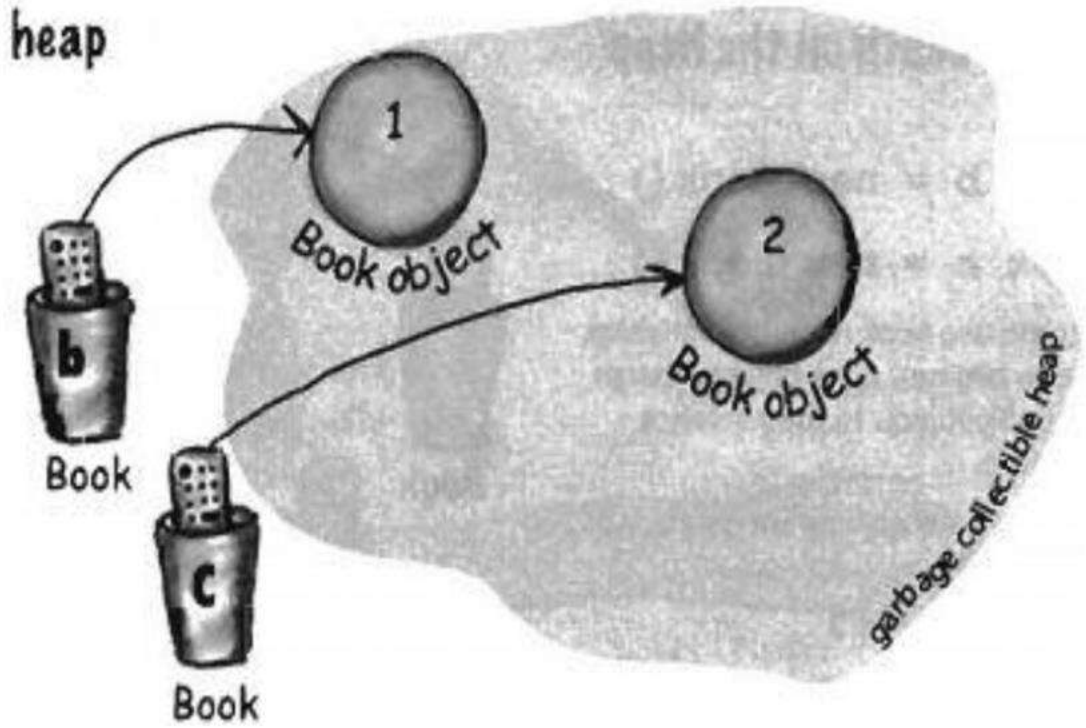
Class and Objects

```
Class Person                                    // Class Declaration
{
    String name;
    int age;
    char gender;
}
Class PersonDemo                                //Demo class with main method
{
    public static void main(String [] args)
    {
                Person p1= new Person();        //Object of class Person created .
                p1.name="Tom";                  //Assign name to p1
                p1.age=23;
                p1.gender='M';
    }
}
```

Object are created in heap

Book b = new Book();

Book c = new Book();

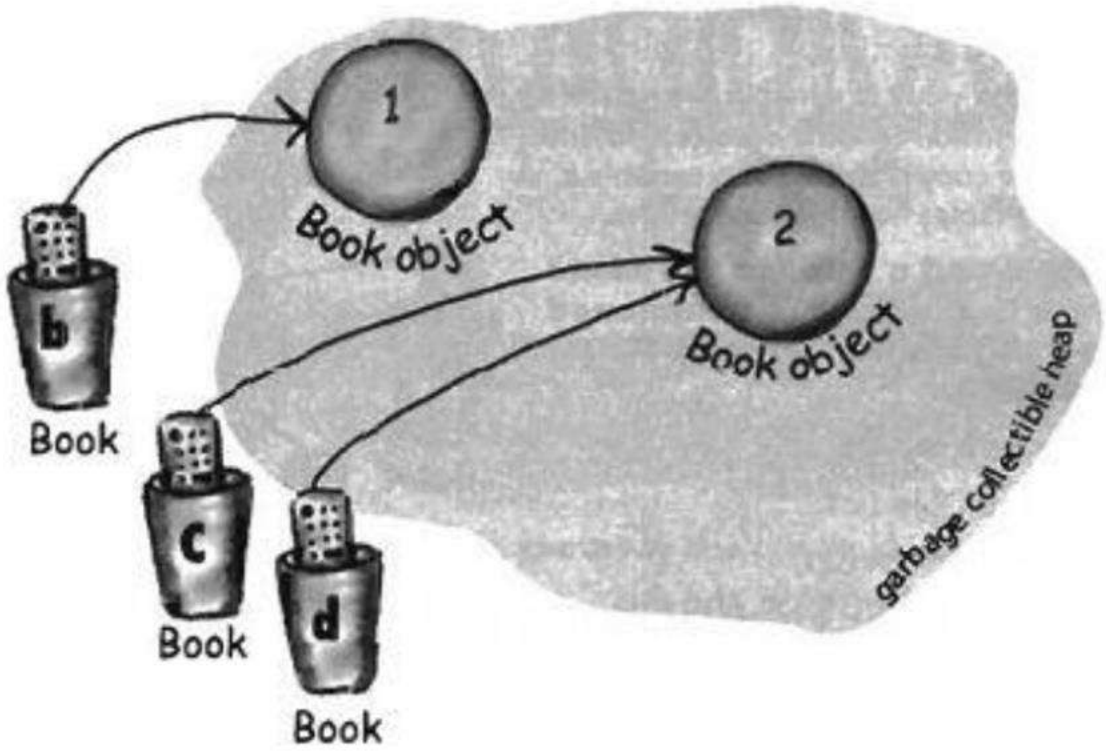


Continue

Book b= new Book();

Book c = new Book();

Book d = c;



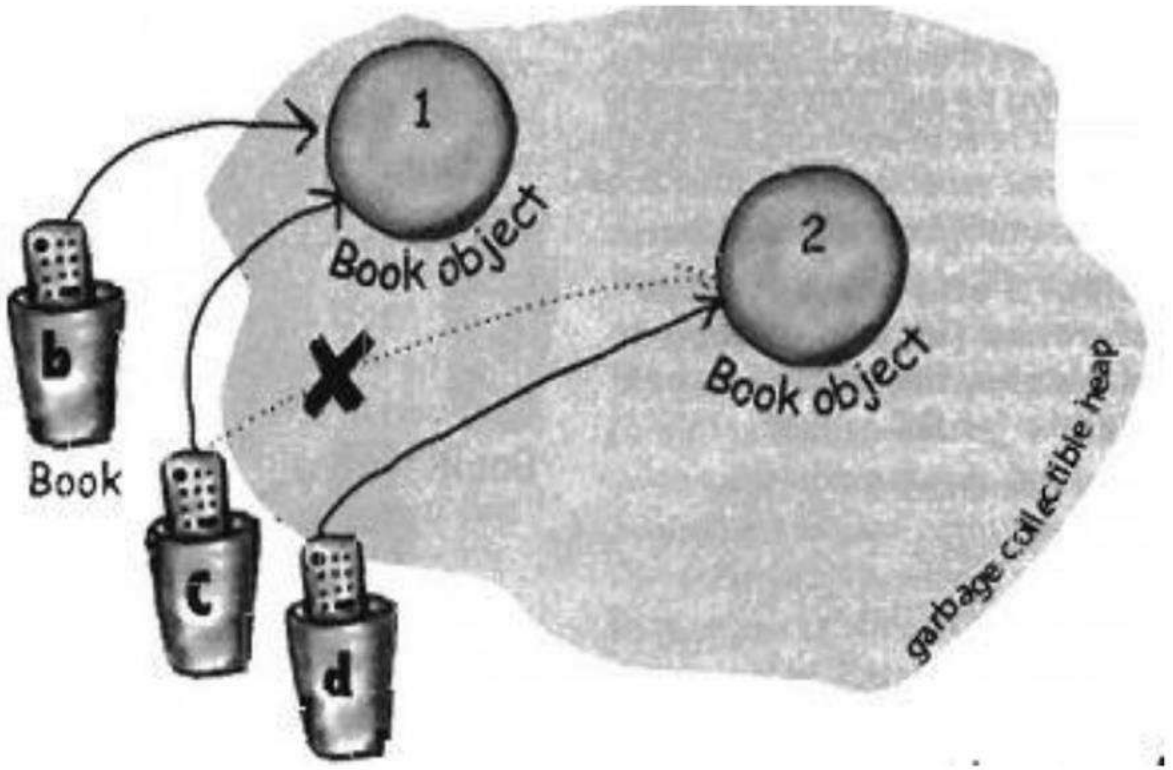
Continue

Book b= new Book();

Book c = new Book();

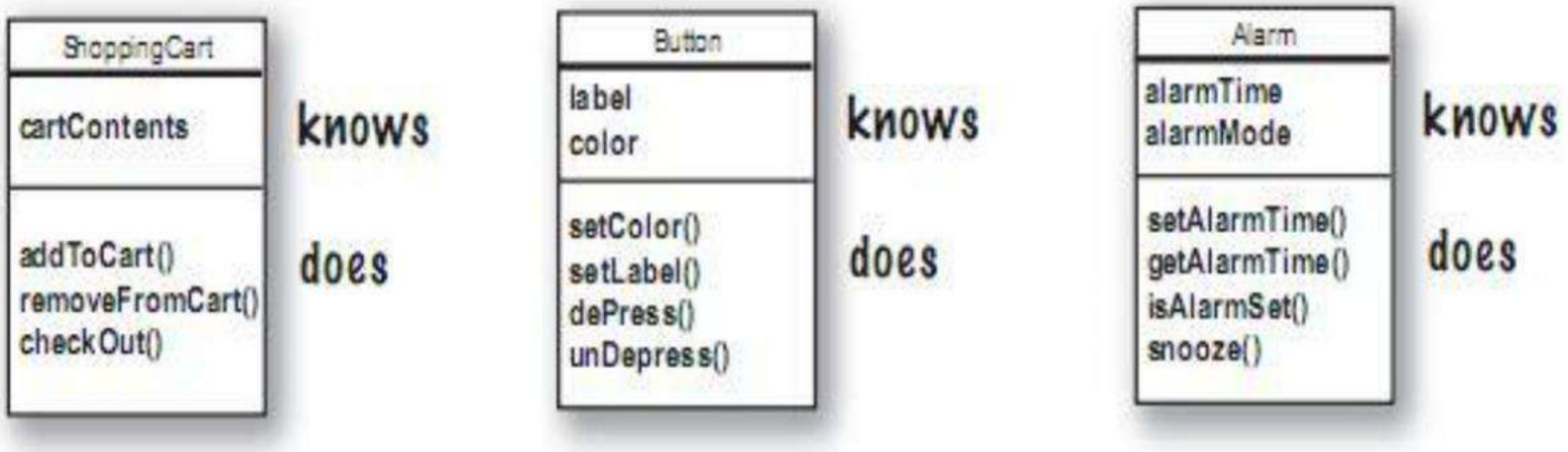
Book d = c;

c=b;



Inside Objects

- When you design a class, think about the objects that will be created from that class type.
- Think about:
  - things the object knows (instance variable)
  - things the object does (method)



Constructors

```
Person p1 = new Person();
```

Looks like we're calling a method named "Person"

Actually it is calling a **constructor** of a class .

Constructor

- Constructor of a class is called when an object of that class is created .
- We didn't write the constructor here , did we ?

```
Class Person                                    // Class Declaration
{
        String name;
        int age;
        char gender;
}
```

Constructor

- It is internally created if we don't create it .
- It has a same name as class .
- A constructor does not have return value unlike methods
- so , we can think previous code like this :

```
Class Person                                    // Class Declaration
  {
            Person()
            {               //blank
            }
            String name;
            int age;
            char gender;
  }
```

Constructor (use)

```
Class Person                                        // Class Declaration
{
    String name;
    int age;
    char gender;
}
Class PersonDemo                        //Demo class with main method
{
    public static void main(String [] args)
    {
                Person p1= new  Person();           //Object of class
    Person created .
                p1.name="Tom";  //Assign name to p1
                p1.age=23;
                p1.gender='M';
    }
}
```

```
Class Person                                                    // Class Declaration
{   String name;
    int age;
    char gender;
    Person(String s)
    {
                name = s ;
    }
}
 Class PersonDemo                       //Demo class with main method
{
    public static void main(String [] args)
    {
                Person p1= new  Person("Tom");  //Object of class Person
    created .
                p1.age=23;
                p1.gender='M';
    }
}
```

Destructors

- JAVA has a special feature called " Garbage Collection " .
- Unused objects are automatically destroyed using this garbage collection mechanism.

```
class Person
{
        Person()
        {
                //Constructor
        }
        ~Person()
        {
                //Destructor
        }
```

Finalize() method

- Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- Simply define the **finalize()** method in class.
- The java run time calls that method whenever it is about to recycle an object of that class.
- Inside the **finalize()** method you will specify those actions that must be performed before an object is destroyed.

finalize()

- The **finalize()** method has this general form:

```
protected void finalize()
{
// finalization code here
}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class.

Methods in JAVA

```
Class Shape
{
        int length;
        int breadth;
        void findArea( )
        {
                System.out.println("Area is  " +(length * breadth) ) ;
        }
}
```

Method in java is what object does .

Methods in JAVA

- Method are having following syntax generally :

```
return-type    method-name  (arg1 , arg2 , ….)
{
                    // method  body
}
```

Let us understand some concepts of methods with programming …..

Method Overloading

- It is known as polymorphism in OOP.
- If **a class** have multiple methods by same name but different parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.

```
class Calculation{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  Calculation obj=new Calculation();
  obj.sum(10,10,10);
  obj.sum(20,20);  }
}
```

Constructor Overloading

- Constructor can also be overloaded because it is also one type of method having same name as class.
- Overloading constructors and methods can increase the readability of the program and make it simpler .

Basically, polymorphism is categorised into 2 parts,i.e,compile time and run time.
Compile time polymorphism can be achieved by 2 ways function overloading and operator overloading
Runtime polymorphism can be achieved by function overriding.

“ this “ keyword

- In java, ‘this’ is a **reference variable** that refers to the current object.
- **Usage of java this keyword**
  - this keyword can be used to refer current class instance variable.
  - this() can be used to invoke current class constructor.
  - this keyword can be used to invoke current class method (implicitly)
  - this can be passed as an argument in the method call.
  - this can be passed as argument in the constructor call.
  - this keyword can also be used to return the current class instance.

Proving this keyword

```
class A5{
void m(){
System.out.println(this);//prints same reference ID
}
public static void main(String args[]){
A5 obj=new A5();
System.out.println(obj);//prints the reference ID

obj.m();
}
}
```

Static Keyword

- The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.
- The static can be:
  - variable (also known as class variable)
  - method (also known as class method)
  - block
  - nested class

Understanding problem without static variable

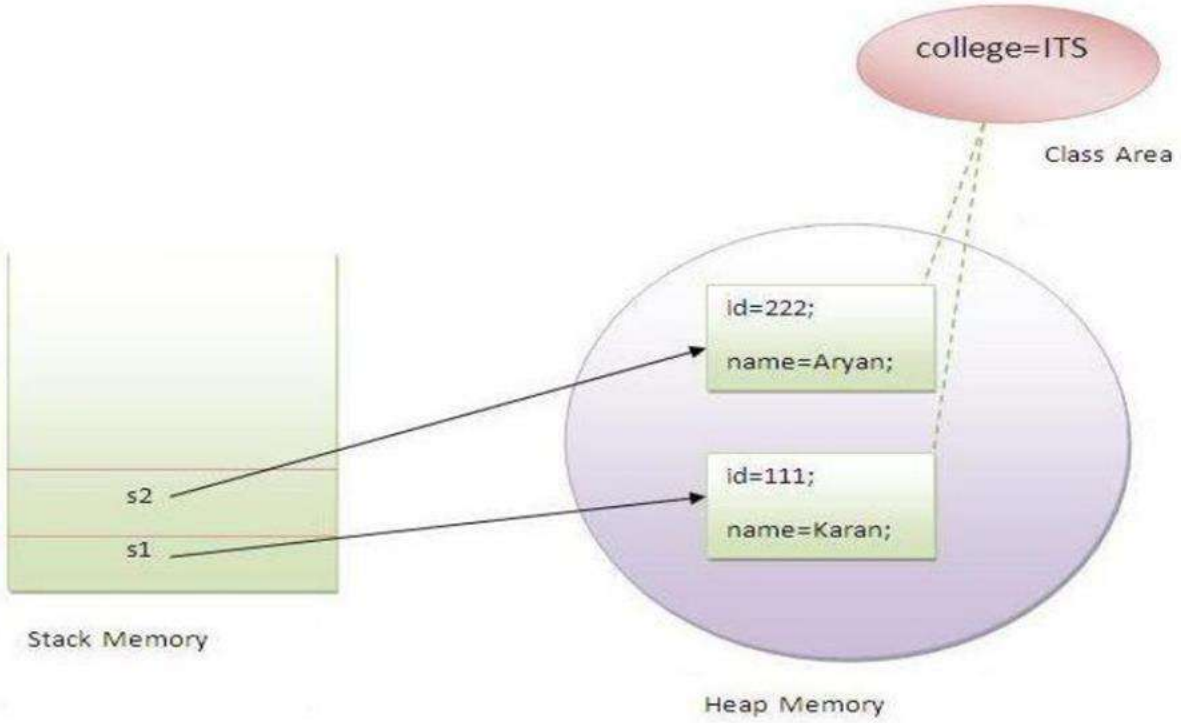
```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

- What if there are 500 students ? (i.e 500 objects ?) , make sense?
- The solution is - ‘static’.

Solution

```
class Student{
                int rollno;
                String name;
                static String college ="ITS";
    Student(int r,String n){
                rollno = r;
                name = n;
                }
    void display ( ) {System.out.println(rollno+" "+name+" "+college);}
    public static void main(String args[]){
                Student s1 = new Student8(111,"Karan");
                Student s2 = new Student8(222,"Aryan");
                s1.display();
                s2.display();
 }
}
```

Static keyword mechanism



Static Method

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

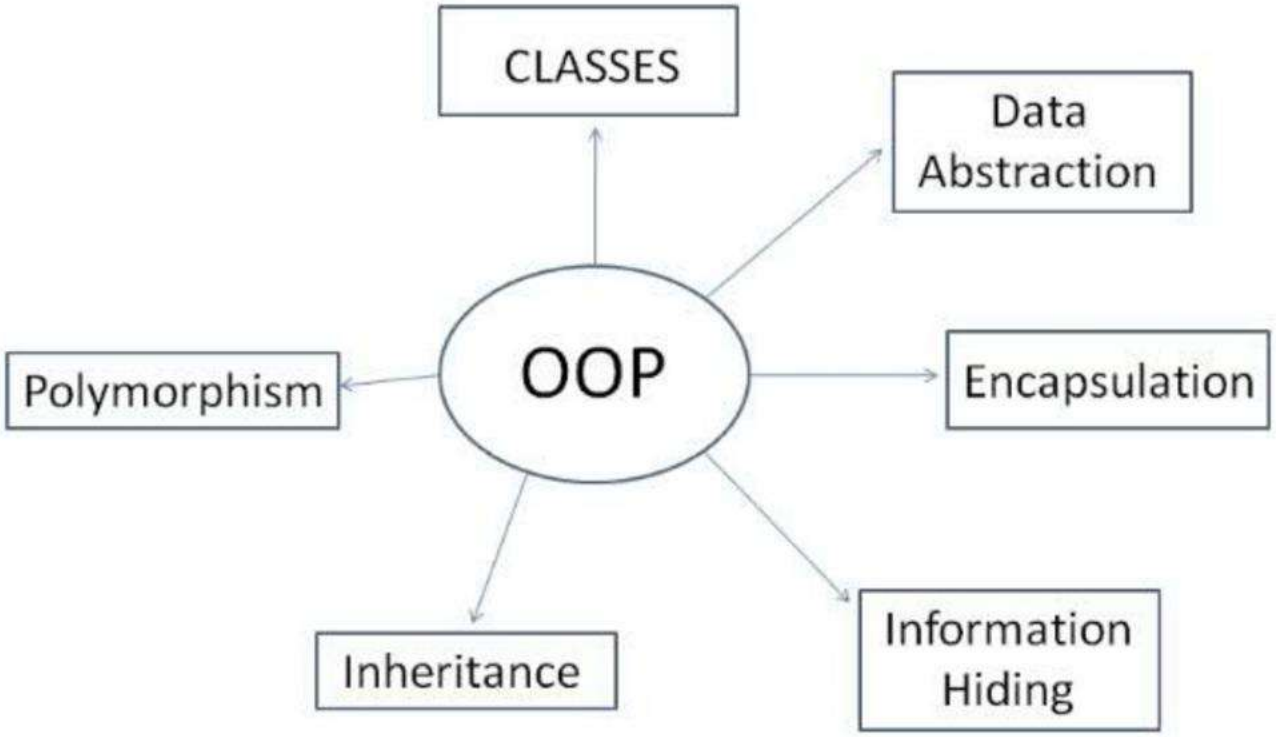
- **Restrictions for static method**
  - The static method can not use non static data member or call non-static method directly.
  - this and super cannot be used in static context.

Here We Are

- Introduction and Basics of JAVA
- Class and Object Methodologies
- OOP flavors
- Exception Handling
- Threads in JAVA
- File and I/O
- Logical Data Storage
- JDBC –ODBC Concepts and XML / JSON Parsing

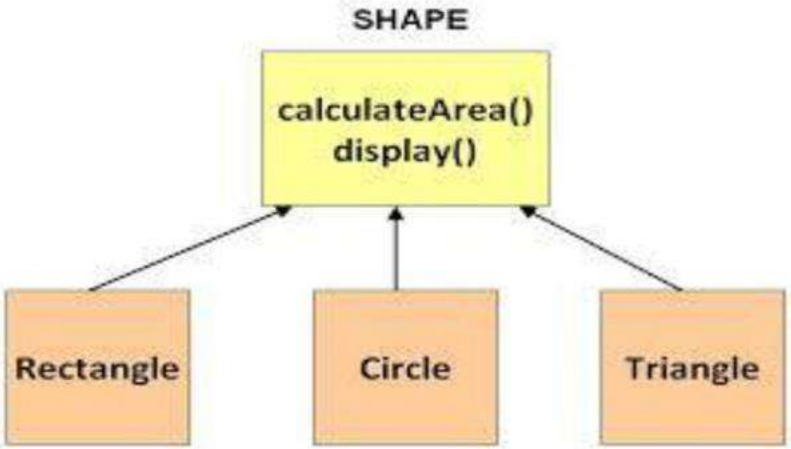
We will go through

- Inheritance
- Super keyword
- Access Specifires
- Packages
- Abstract Class-methods
- Interface
- Enums



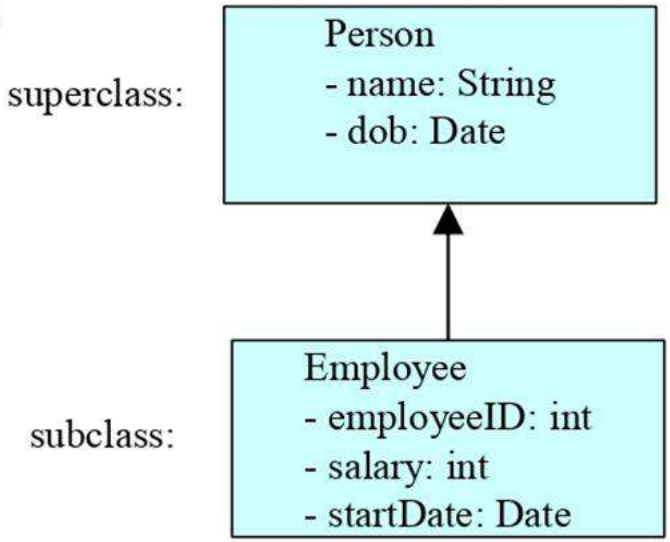
Inheritance

- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.
- The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).



Terminology

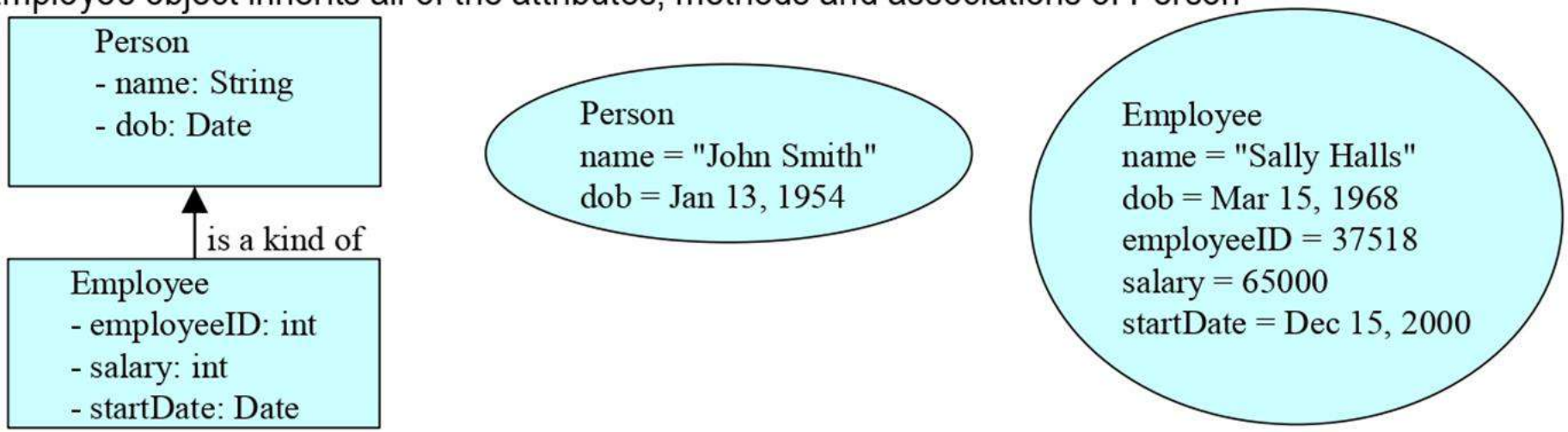
- Inheritance is a fundamental Object Oriented concept
- A class can be defined as a "subclass" of another class.
  - The subclass inherits all data attributes of its superclass
  - The subclass inherits all methods of its superclass
  - The subclass inherits all associations of its superclass
- The subclass can:
  - Add new functionality
  - Use inherited functionality
  - Override inherited functionality



What really happens?

- When an object is created using new, the system must allocate enough memory to hold all its instance variables.
  - This includes any inherited instance variables

- In this example, we can say that an Employee "is a kind of" Person.
  - An Employee object inherits all of the attributes, methods and associations of Person



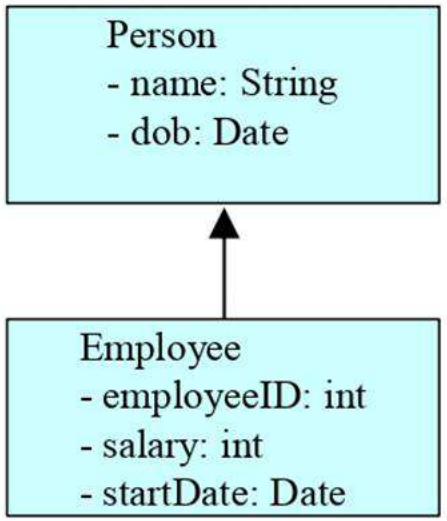
Inheritance in Java

- Inheritance is declared using the "extends" keyword
  - If inheritance is not defined, the class extends a class called Object

```
public class Person
{
    String name;
    Date dob;
   [...]
```

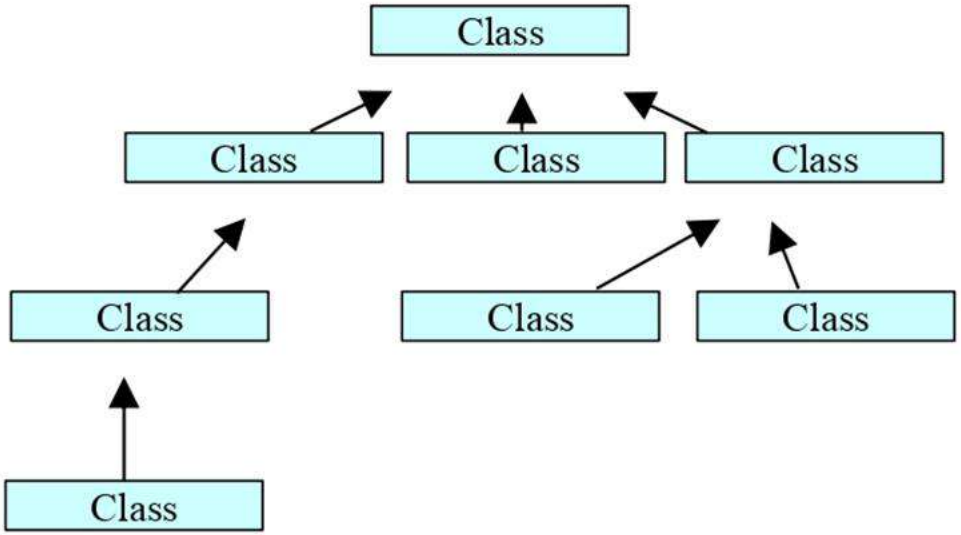
```
public class Employee extends Person
{
   private int employeID;
   private int salary;
   private Date startDate;
   [...]
```

```
Employee anEmployee = new Employee();
```



Inheritance Hierarchy

- Each Java class has one (and only one) superclass.
  - C++ allows for multiple inheritance
- Inheritance creates a class hierarchy
  - Classes higher in the hierarchy are more general and more abstract
  - Classes lower in the hierarchy are more specific and concrete
- There is no limit to the number of subclasses a class can have
- There is no limit to the depth of the class tree.



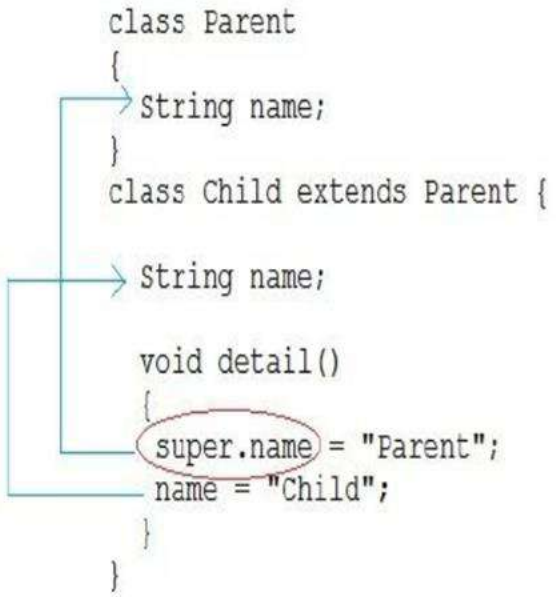
Super keyword

•The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

•Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

- super is used to refer immediate parent class instance variable.
- super() is used to invoke immediate parent class constructor.
- super is used to invoke immediate parent class method.



Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.
- **Usage of Java Method Overriding**
  - Method overriding is used to provide specific implementation of a method that is already provided by its super class.
  - Method overriding is used for runtime polymorphism
- **Rules for Java Method Overriding**
  - method must have same name as in the parent class
  - method must have same parameter as in the parent class.
  - must be a relationship like inheritance.

Access Specifires

- public
  - Data / Method can be accessed from any other class present in any package
- private
  - Data / Method can be accessed from only within the class
- protected
  - Data / Method can be accessed from all classes in the same package and sub-classes.
- default
  - Data / Method can be accessed only from within the same package.

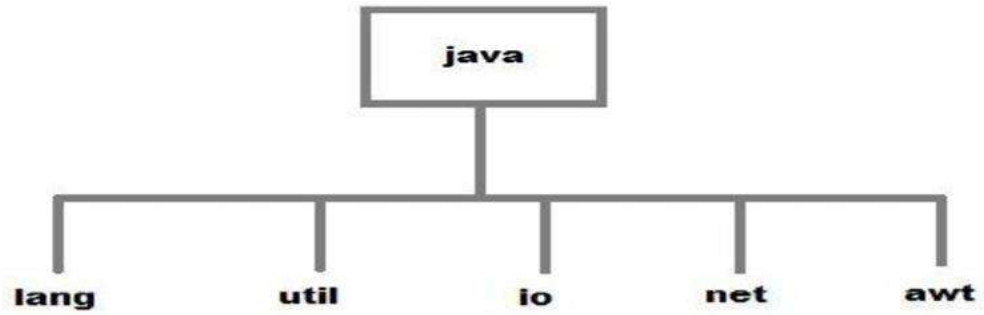
Let us practically understand this .....

Packages

- Package are used in Java, in-order to avoid name conflicts and to control access of class, interface and enumeration etc.
- A package can be defined as a group of similar types of classes, interface, enumeration and sub-package.
- Using package it becomes easier to locate the related classes.

Package are categorized into two forms

- Built-in Package:-Existing Java package for example java.lang, java.util etc.
- User-defined-package:- Java package created by user to categorized classes and interface



Packages

- **Creating a package**
- Creating a package in java is quite easy. Simply include a package command followed by name of the package as the first statement in java source file.

```
package mypack;
 public class employee
{
 ...statement;
 }
```

The above statement create a package called **mypack**.

- Java uses file system directory to store package. For example the .class for any classes you to define to be part of **mypack** package must be stored in a directory called mypack

Packages

- Package is a way to organize files in java, it is used when a project consists of multiple modules.
- Also helps resolve naming conflicts.
- Package's access level also allows you to protect data from being used by the non-authorized classes.

- **Import Keyword :**
- **import** keyword is used to import built-in and user-defined packages into your java source file.
- Your class can refer to a class that is in another package by directly using its name.

Packages

3 different ways to refer to class that is present in different package

- **Using fully qualified name** (But this is not a good practice.)

```
class MyDate extends java.util.Date
        { //statement; }
```

- **import the only class you want to use.**

```
import java.util.Date;
class MyDate extends Date
{ //statement. }
```

- **import all the classes from the particular package**

```
import java.util.*;
class MyDate extends Date
{ //statement; }
```

Packages

- **import statement is kept after the package statement.**
- *Example :*

```
package mypack;
import java.util.*;
```

- But if you are not creating any package then import statement will be the first statement of your java source file.

Static import

- **static import** is a feature that expands the capabilities of **import** keyword.
- Used to import **static** member of a class.
- Using **static import**, it is possible to refer to the static member directly without its class name.

Two general form of static import statement.

1.import only a single static member of a class

import static *package.class-name.static-member-name;*

import static java.lang.Math.sqrt; //importing static method **sqrt** of **Math** class

2 .imports all the static member of a class

import static *package.class-type-name.*;*

- import static java.lang.Math.*; //importing all static member of **Math** class

Abstract Class

Abstraction in Java

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

Abstract class (0 to 100%)

Interface (100%)

Abstract class

Abstract class in Java

- A class that is declared as abstract is known as **abstract class**.
- It needs to be extended and its method should be implemented. It cannot be instantiated.

abstract class :

abstract class A{ }

abstract method

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

abstract method

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

Abstract class

Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
class Honda extends Bike{
    void run()
    {              System.out.println("running safely..");      }

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

Some strict rules

- If there is any abstract method in a class, that class must be abstract.
- If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

Let's try to break them …….

Interface

- An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.
- The interface in java is **a mechanism to achieve fully abstraction**.
- There can be only abstract methods in the java interface not method body.
- It is used to achieve fully abstraction and multiple inheritance in Java.
- It cannot be instantiated just like abstract class.

Interface vs. Abstract Class

oops interface vs abstract class

Interface	Abstract class
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does'n Contains Data Member	Abstract class contains Data Member
Interface does'n contains Cunstructors	Abstract class contains Cunstructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

Interface

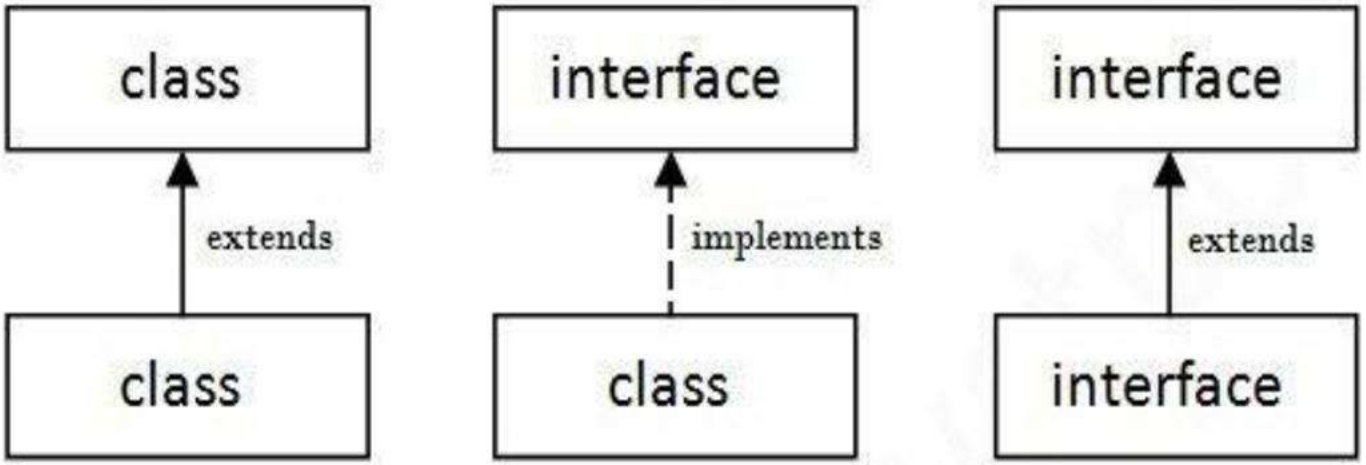
Why interface?

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Let's check it out .....

Interface

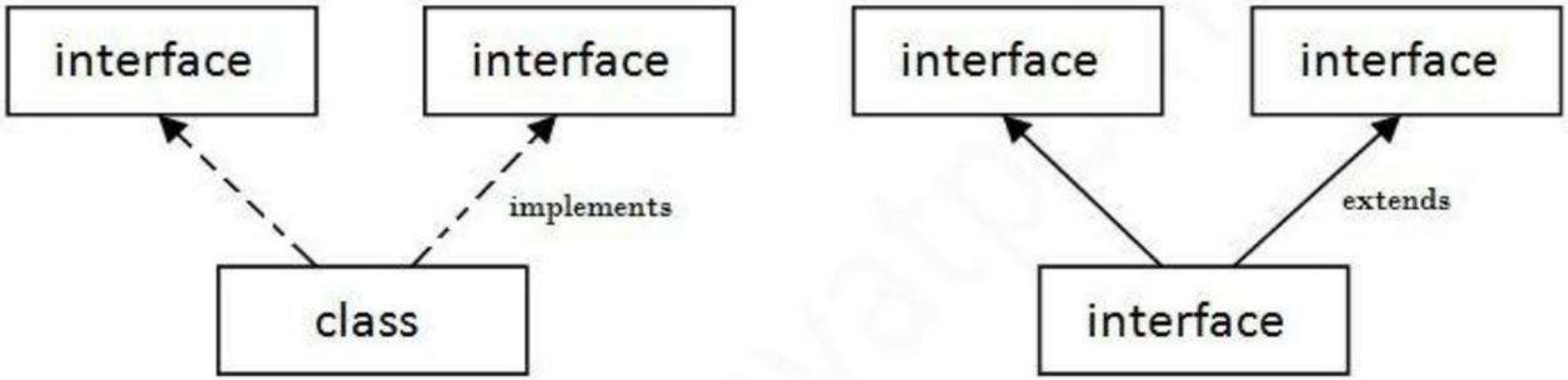
A class extends another class, an interface extends another interface but a **class implements an interface**.



Interface

Multiple inheritance in Java by interface

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



Enums

- A *Java Enum* is a special Java type used to define collections of constants.
- More precisely, a Java enum type is a special kind of Java class.
- An enum can contain constants, methods etc.
- Java enums were added in Java 5.

```
public enum Level
{
        HIGH,
        MEDIUM,
        LOW
}
```

`enum` keyword which is used in place of `class` or `interface`.
The Java `enum` keyword signals to the Java compiler that this type definition is an enum.

Enums with If - else

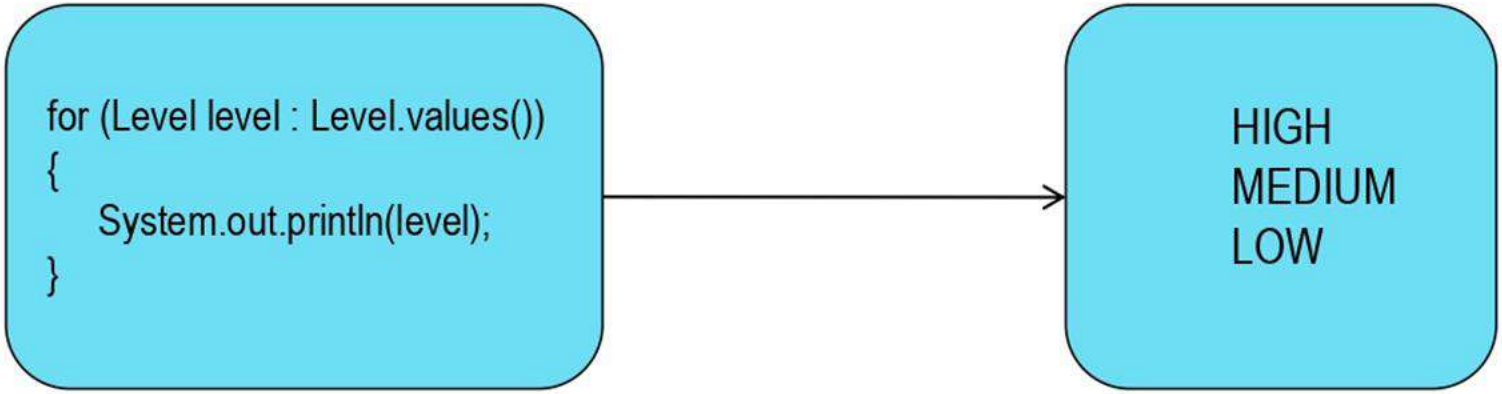
```
Level level = Level.HIGH;
```

The level variable can take one of the Level enum constants as value (HIGH, MEDIUM or LOW). In this case level is set to HIGH.

```
  Level level = ...                  //assign some Level constant to it
  if( level == Level.HIGH)
  { }
else if( level == Level.MEDIUM)
  { }
else if( level == Level.LOW)
  { }
```

Enums – Values() method

- One can obtain an array of all the possible values of a Java enum type by calling its static values() method.
- All enum types get a static values() method automatically by the Java compiler.



Adding Values

```
public enum Level {
        HIGH (3), //calls constructor with value 3
        MEDIUM(2), //calls constructor with value 2
        LOW (1) //calls constructor with value 1
        ;           // semicolon needed when fields / methods follow

        private final int levelCode;
        private Level(int levelCode)
        {
            this.levelCode = levelCode;
        }
    }
```

Methods in Enum

```
public enum Level {
        HIGH (3),      //calls constructor with value 3
        MEDIUM(2),  //calls constructor with value 2
        LOW (1)        //calls constructor with value 1
        ;                    // semicolon needed when fields / methods follow
        private final int levelCode;
        Level(int levelCode)
        {
                this.levelCode = levelCode;
        }
        public int getLevelCode()
        {
                return this.levelCode;
        }
}
```

```
Level level = Level.HIGH;
System.out.println(level.getLevelCode());
```

Course Content

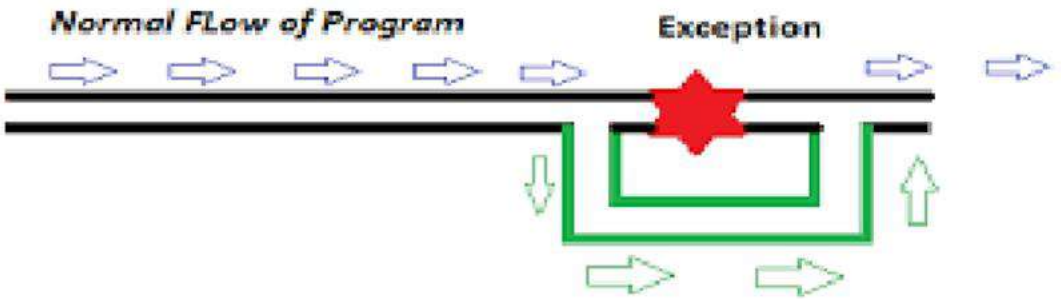
- Introduction and Basics of JAVA
- Class and Object Methodologies
- OOP flavors
- Exception Handling
- Threads in JAVA
- File and I/O
- Logical Data Storage
- JDBC –ODBC Concepts and XML / JSON Parsing

We will go through

- Exceptions
- Types of Exceptions
- Try-catch
- Finally
- throw
- throws
- Custom Exception

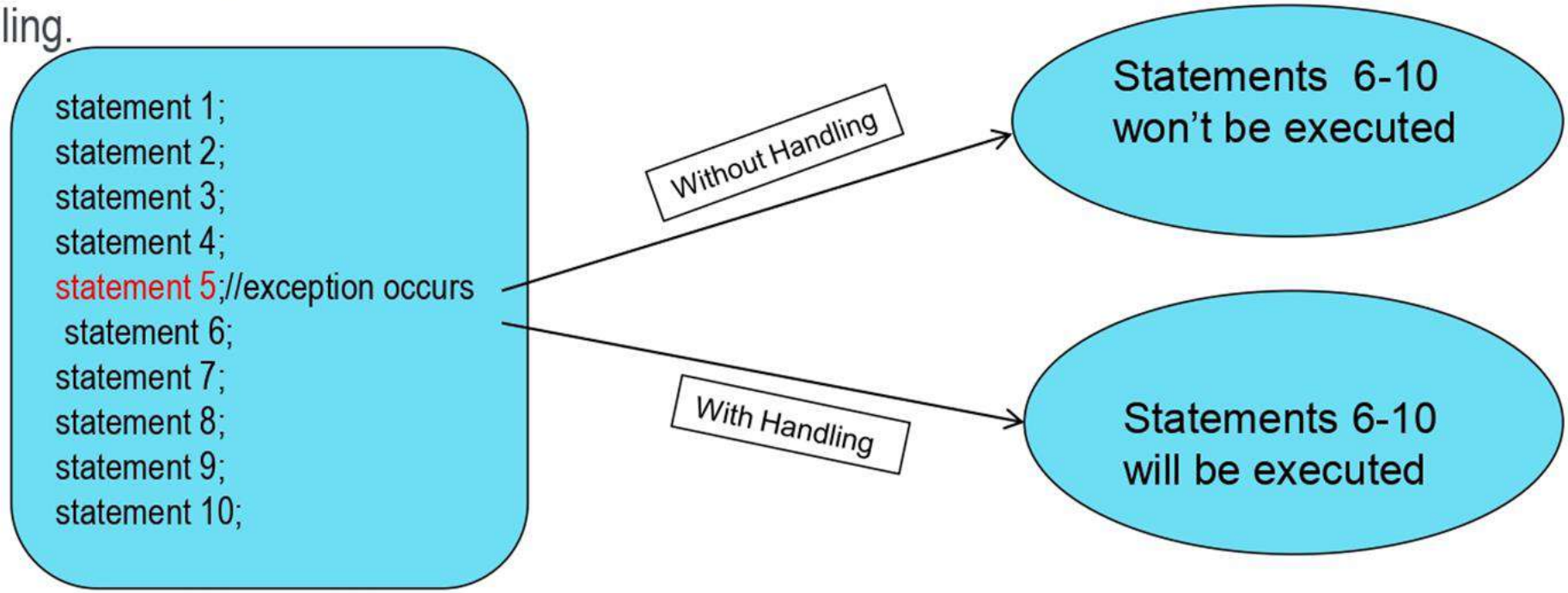
Exception Handling

- The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- **Dictionary Meaning:** Exception is an abnormal condition.
- **What is exception handling**
  - Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

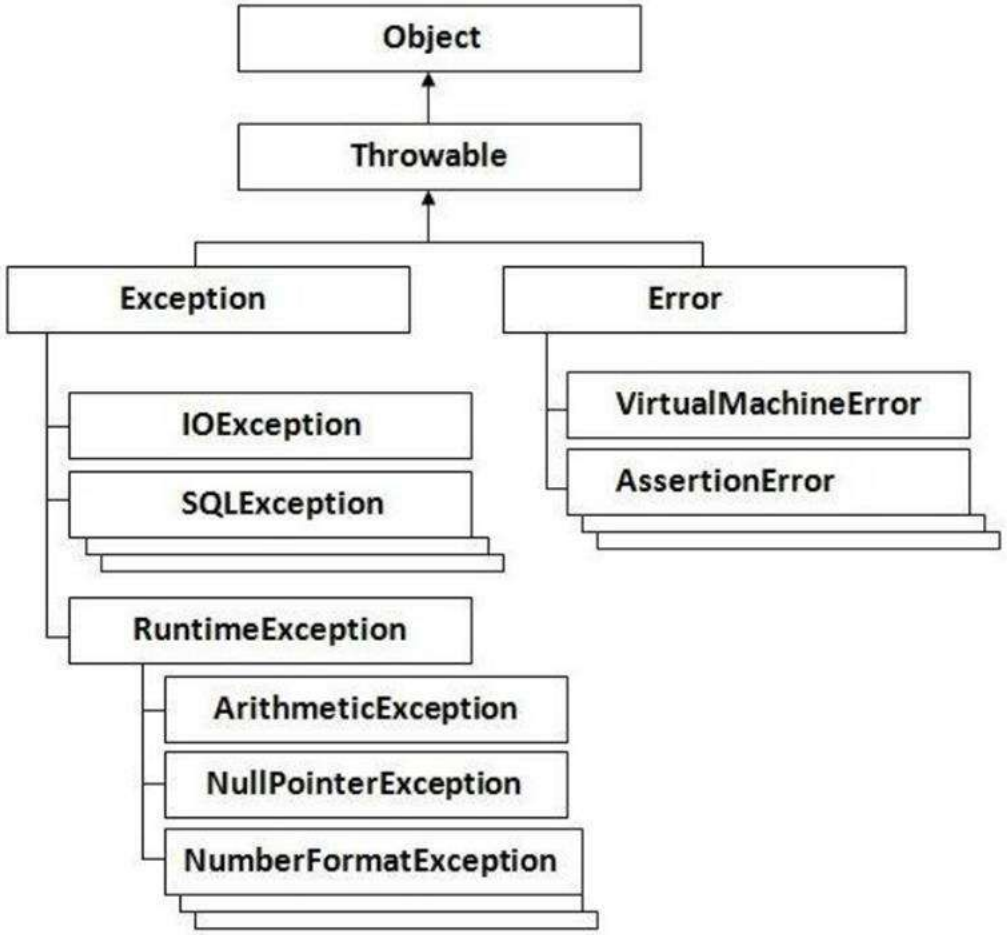


Advantage of Exception Handling

- The core advantage of exception handling is **to maintain the normal flow of the application**.
- Exception normally disrupts the normal flow of the application that is why we use exception handling.



Hierarchy


Object
Throwable
Exception
Error
IOException
SQLException
RuntimeException
VirtualMachineError
AssertionError
ArithmeticException
NullPointerException
NumberFormatException

Types of Exception

- Mainly two types of exceptions:
- **checked** and **unchecked** where **error** is considered as unchecked exception.
- The **sun microsystem** says there are three types of exceptions:
- Checked Exception
- Unchecked Exception
- Error

Checked Exception

- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions

  e.g. IOException, SQLException etc.
- Checked exceptions are checked **at compile-time**.

Continue

Unchecked Exception

- The classes that extend RuntimeException are known as unchecked exceptions
  e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- Unchecked exceptions are not checked at compile-time rather they are **checked at runtime**.

Error

- Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Scenarios where Exceptions occur

1) ArithmeticException occurs

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

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

2) NullPointerException occurs

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

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

Scenarios where Exceptions occur

3) NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);          //NumberFormatException
```

4) ArrayIndexOutOfBoundsException occurs

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

```
int a[]=new int[5];
a[10]=50;                  //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

5 keywords used in java exception handling.

- try
- catch
- finally
- throw
- throws

try catch finally

- Java try block is used to enclose the code that might throw an exception.
- It must be used within the method.
- Java try block must be followed by either catch or finally block.

```
try
{
//code that may throw
exception
}
catch(Exceptionclass
Ref)
{
}
```

```
try
{
//code that may throw
exception
}
finally
{

}
```

try catch finally

Java catch block

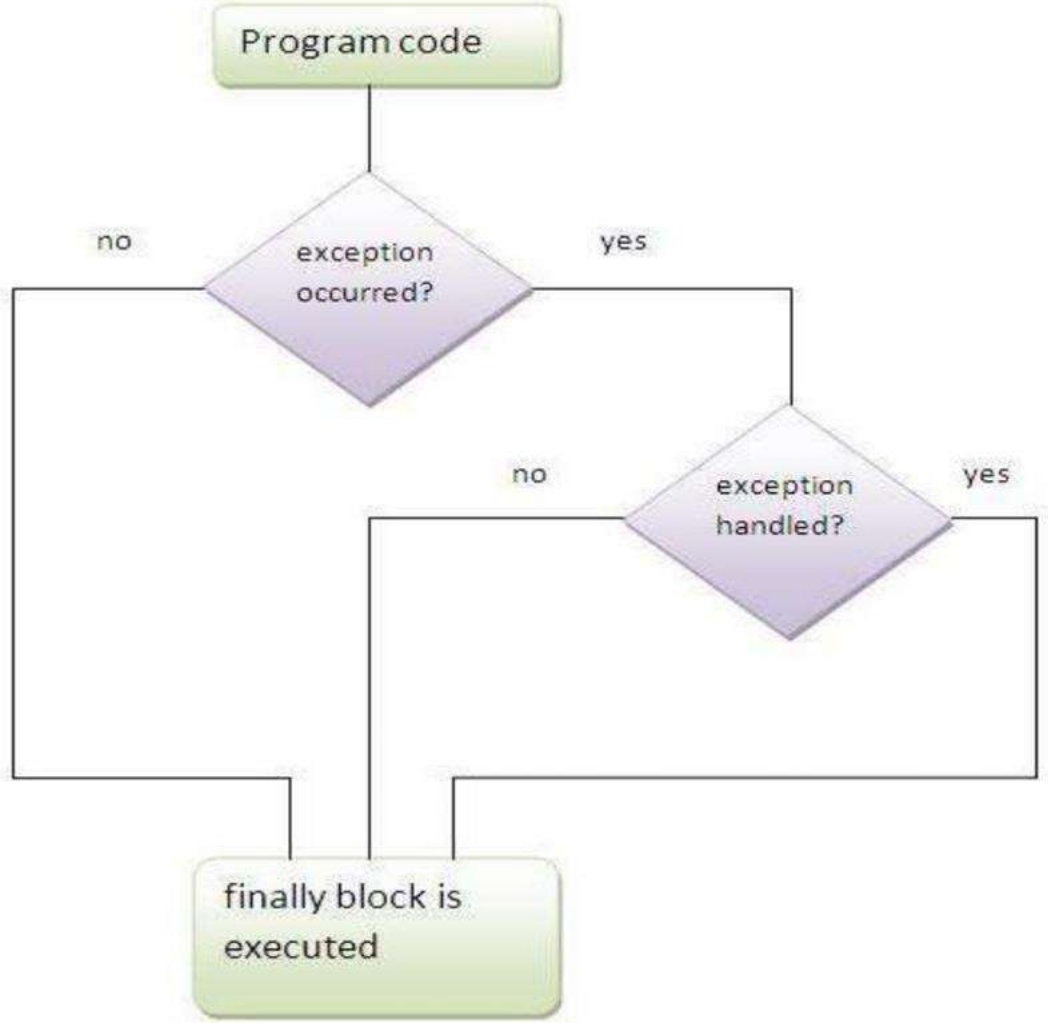
- Java catch block is used to handle the Exception.
- It must be used after the try block only.
- You can use multiple catch block with a single try.

```
try
{
//code that may throw  excepti
on
}
catch(Exceptionclass Ref)
{

}
catch(Exceptionclass2  Ref)
{

}
catch(Exceptionclass3 Ref)
{

}
```

try catch finally

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block.



Some Rules for Exception Handling

- At a time only one Exception is occured and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .
- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

throw

- throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception.

syntax

- throw exception;

Example

throw new IOException("sorry device error);

throw

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    validate(13);
    System.out.println("rest of the code...");
 }
}
```

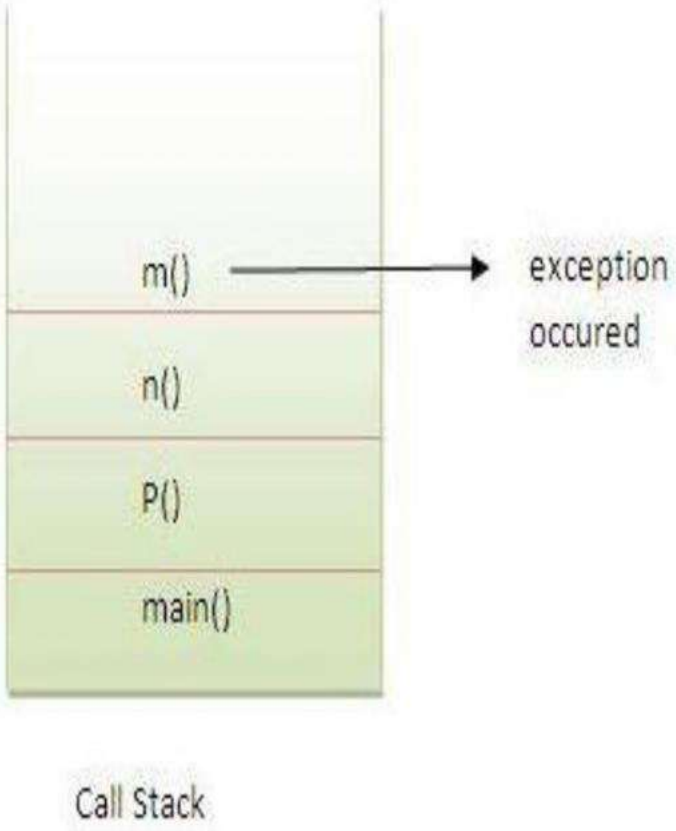
Output

```
Exception in thread main
java.lang.ArithmeticException:not
valid
```

Exception Propagation (Unchecked)

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:
exception handled
normal flow...



Exception Propagation (checked)

```
class TestExceptionPropagation2{
  void m(){
    throw new java.io.IOException("device error");//checked exception
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handeled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation2 obj=new TestExceptionPropagation2();
   obj.p();
   System.out.println("normal flow");
  }
}
```

Output:

Compile time Error

throws

- **throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions.
- If there occurs any unchecked exception such as *NullPointerException*, it is programmers fault that he is not performing check up before the code being used.

Syntax

```
return_type method_name() throws exception_class_name
{
        //method code
}
```

throw

Which exception should be declared

Ans) checked exception only,

because:

unchecked Exception: under your control so correct your code.

error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

throw example

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
    Testthrows1 obj=new Testthrows1();
    obj.p();
    System.out.println("normal flow...");
   }
}
```

Output :
exception handled
normal flow...

Custom Exception

- If you are creating your own Exception that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.

```
class InvalidAgeException extends Exception
{
    InvalidAgeException(String s)
   {
            super(s);
   }
}
```

```
class TestCustomException1{

  static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }

  public static void main(String args[]){
    try{
    validate(13);
    }catch(Exception m){System.out.println("Exception occured: "+m);}

    System.out.println("rest of the code...");
  }
}
```

Output:
Exception occured:
InvalidAgeException:not valid
rest of the code...

Course Content

- Introduction and Basics of JAVA
- Class and Object Methodologies
- OOP flavors
- Exception Handling
- Threads in JAVA
- File and I/O
- Logical Data Storage
- JDBC –ODBC Concepts and XML / JSON Parsing

Threads basics

- Thread is basically a lightweight sub-process, a smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.

Advantage of Java Multithreading

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
2. You **can perform many operations together so it saves time**.
3. Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

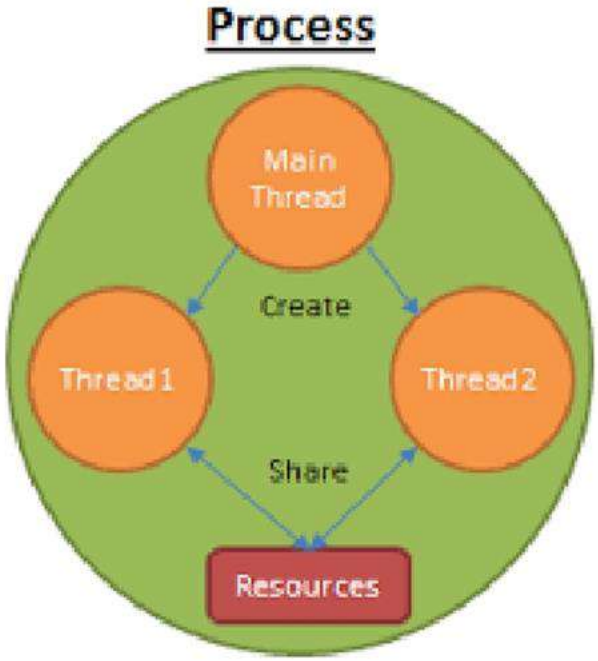
Process vs. Thread

Process

- Each process have its own address in memory
  i.e. each process allocates separate memory area.
- Process is heavyweight.

Thread

- Threads share the same address space.
- Thread is lightweight.

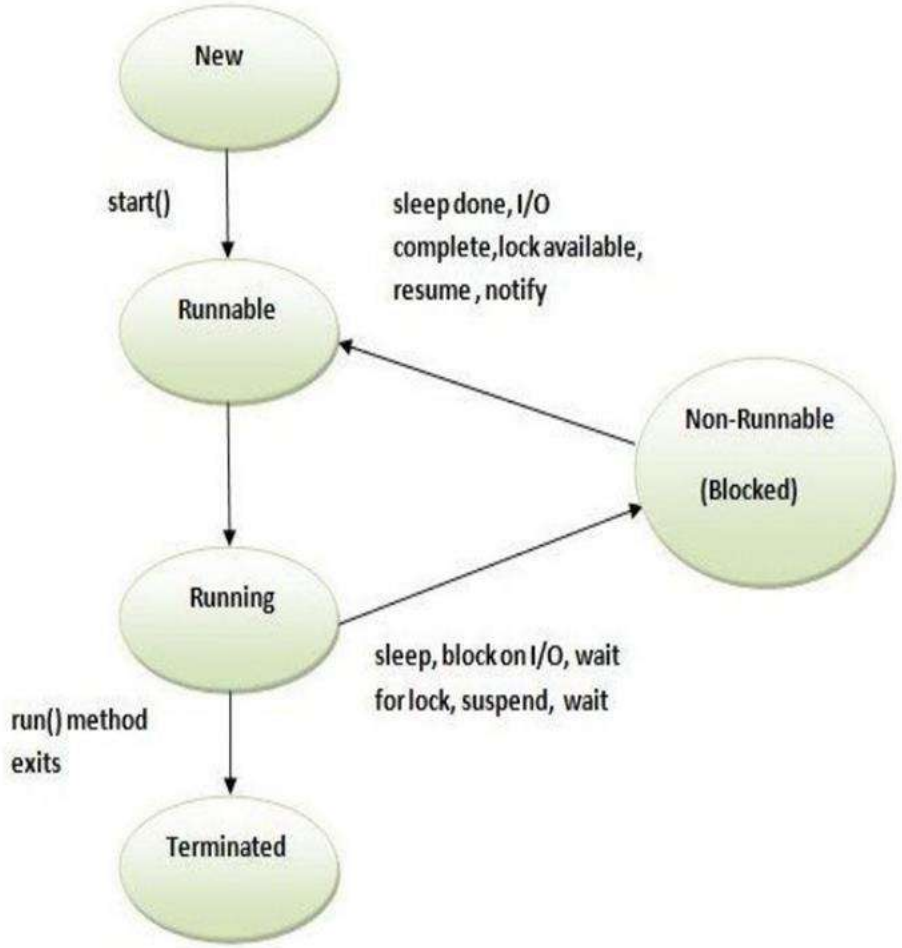


Thread Life Cycle

- A thread can be in one of the five states.
- According to sun, there are 4 states in
  thread life cycle in java
  new, runnable, non-runnable and terminated.
- The life cycle of the thread in java is controlled by JVM.

Thread states

- New
- Runnable
- Running (for understanding)
- Blocked
- Terminated



Continue

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

Creating Threads

Two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

Thread class:

- Thread class provide constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

Thread()

Thread(String name)

Thread(Runnable r)

Thread(Runnable r,String name)

Creating Threads

Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- Runnable interface have only one method named run().

  public void run();

  is used to perform action for a thread.

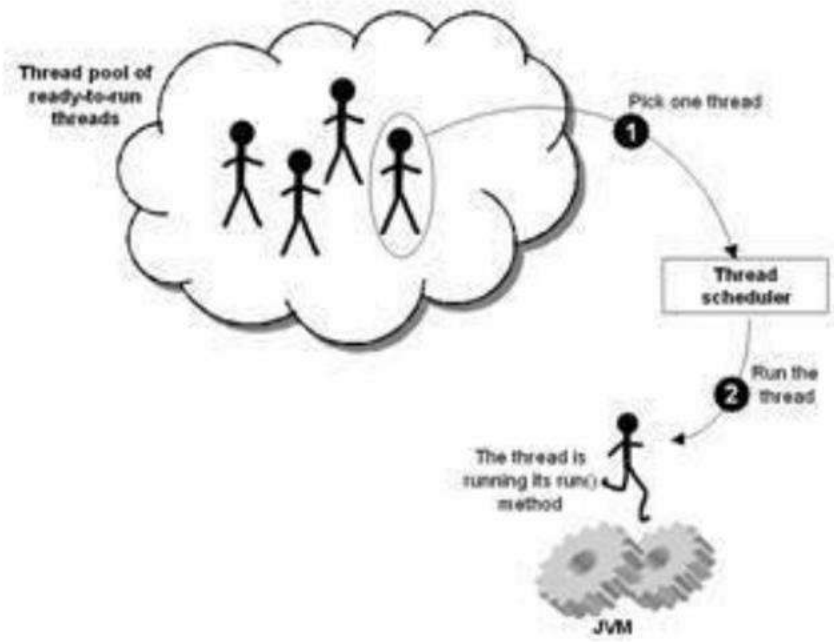
Creating Threads

```
class MyT1 extends Thread{
   public void run()
   {
         System.out.println("thread is running...");
   }
   public static void main(String args[])
   {
              MyT1  t1=new  MyT1();
             t1.start();
   }
}
```

```
class MyT2 implements Runnable{
   public void run()
{
   System.out.println("thread is running...");
}
  public static void main(String args[]){
              MyT2  m1=new  MyT2();
              Thread t1 =new Thread(m1);
              t1.start();
   }
}
```

Thread Scheduler

- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- Only one thread at a time can run in a single process.



Sleep method in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

public static void sleep(long miliseconds)throws InterruptedException

- At a time only one thread is executed.
- If you sleep a thread for the specified time,the thread scheduler picks up another thread and so on.

Sleep

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}
    catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();
  t1.start();
  t2.start();
 }
}
```

Thread Priority

Scheduling:

Execution of multiple threads on a single CPU, in some order.

Priority:

The Java supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling.

When a Java thread is created, it inherits its priority from the thread that created it.

Priority Modification:

setPriority() method.

Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY.

Preemptive Scheduling:

If a thread with a higher priority than the currently executing thread needs to execute, the higher priority thread is immediately scheduled.

Thread Priority

Equal Priority Scheduling:

The scheduler chooses in a round-robin fashion. The chosen thread will run until one of the following conditions is true:

- a higher priority thread becomes "Runnable"
- it yields, or its run() method exits
- on systems that support time-slicing, its time allotment has expired

Then the second thread is given a chance to run, and so on, until the interpreter exits.

Can we start a thread twice

- After starting a thread, it can never be started again.
- If you do so, an *IllegalThreadStateException* is thrown.
- In such case, thread will run once but for second time, it will throw exception.

example

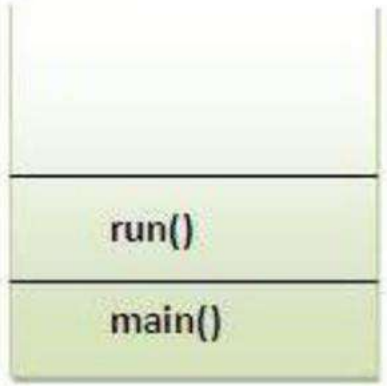
```
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

A Question

What if we call run() method directly instead start() method?

Answer

```
class TestCallRun1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestCallRun1 t1=new TestCallRun1();
  t1.run();      //fine, but does not start a separate call stack
 }
}
```



Stack
(main thread)

Scenario

```
class TestCallRun2 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();

  t1.run();
  t2.run();
 }
}
```

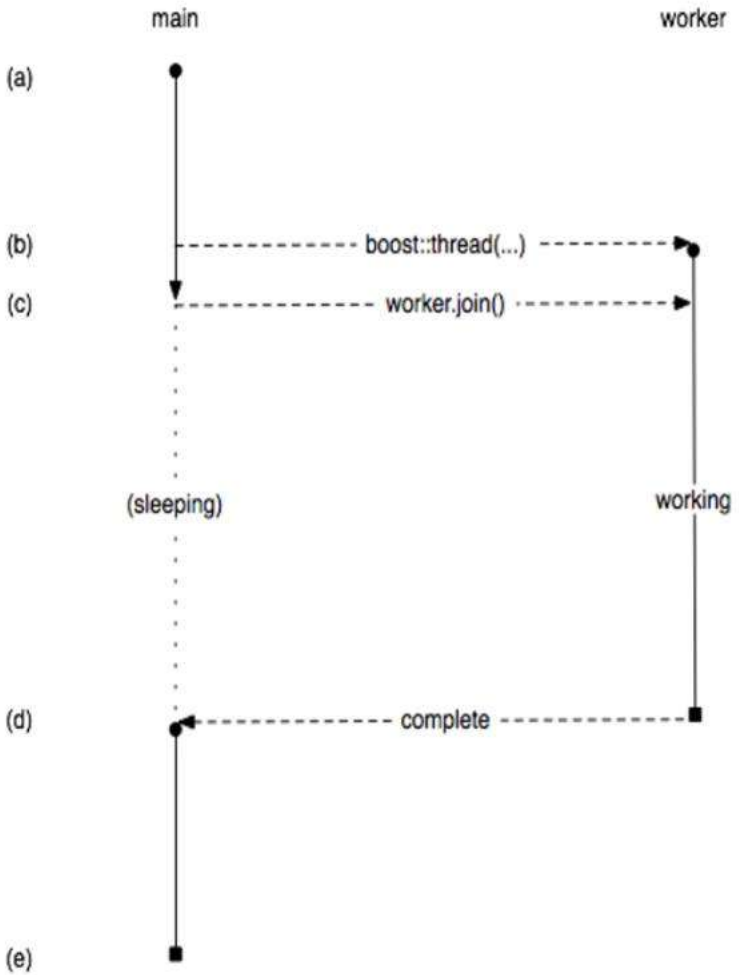
join() method

- The join() method waits for a thread to die.
- In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

public void join() throws InterruptedException

public void join(long milliseconds) throws InterruptedException



```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
}
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
                t1.join();
     }
catch(Exception e)
   {
                System.out.println(e);
   }
    t2.start();
    t3.start();
  }
}
```

Some Methods

- public String getName()
- public void setName (String name)
- public long getId()
- public static Thread currentThread()

Synchronization

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

Why use Synchronization

The synchronization is mainly used to

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

Thread Synchronization

Two types of thread synchronization

Mutual Exclusive

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

Cooperation (Inter-thread Communication in java)(**ITC**)

Mutual Exclusive

- Mutual Exclusive helps keep threads from interfering with one another while sharing data.
  - by synchronized method
  - by synchronized block
  - by static synchronization

Concept of Lock in Java

- Synchronization is built around an internal entity known as the lock or monitor.
- Every object has an lock associated with it.
- By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
- From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Synchronization

Better to visualize it …….so … Let's Go ….

Synchronized Block

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

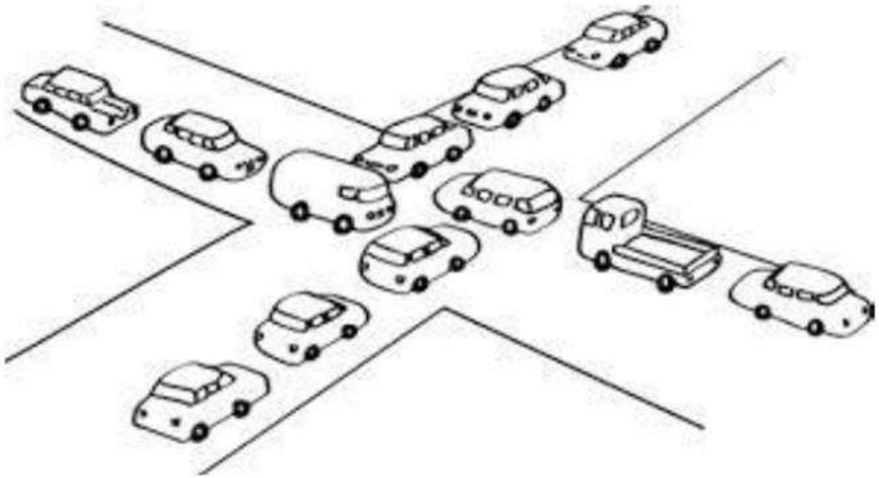
```
synchronized (object reference expression)
{
        / /code block
}
```

Synchronized Block (Example)

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

Deadlock in java

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



Inter-thread communication in Java

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of **Object class**:
- wait()
- notify()
- notifyAll()

ITC

1) wait() method

wait() tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify() or notifyall().

Method Description

public final void wait() throws InterruptedException - waits until object is notified.

public final void wait(long timeout)throws InterruptedException - waits for the specified amount of time.

ITC

2) notify() method

- Wakes up a single thread that is waiting on this object's monitor.
- If any threads are waiting on this object, one of them is chosen to be awakened.
- The choice is arbitrary and occurs at the discretion of the implementation.

```
public final void notify()
```

3) notifyAll() method

- Wakes up all threads that are waiting on this object's monitor.

```
public final void notifyAll()
```

ITC

Let us Understand It.....

Course Content

- Introduction and Basics of JAVA
- Class and Object Methodologies
- OOP flavors
- Exception Handling
- Threads in JAVA
- File and I/O
- Logical Data Storage
- JDBC –ODBC Concepts and XML / JSON Parsing

We will go through

- I/O basics
- Input Output Streams
- Console I/O
- Scanner
- File Handling
- Serialization
- Logging in JAVA

I/O Basics

- **Java I/O** (Input and Output) is used to process the input and produce the output based on the input.
- Java uses the concept of stream to make I/O operation fast.
- The java.io package contains all the classes required for input and output operations.
- We can perform **file handling in java** by java IO API.

Stream

- A stream is a sequence of data. **Stream is composed of bytes.**
- It's called a stream because it's like a stream of water that continues to flow.
- 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

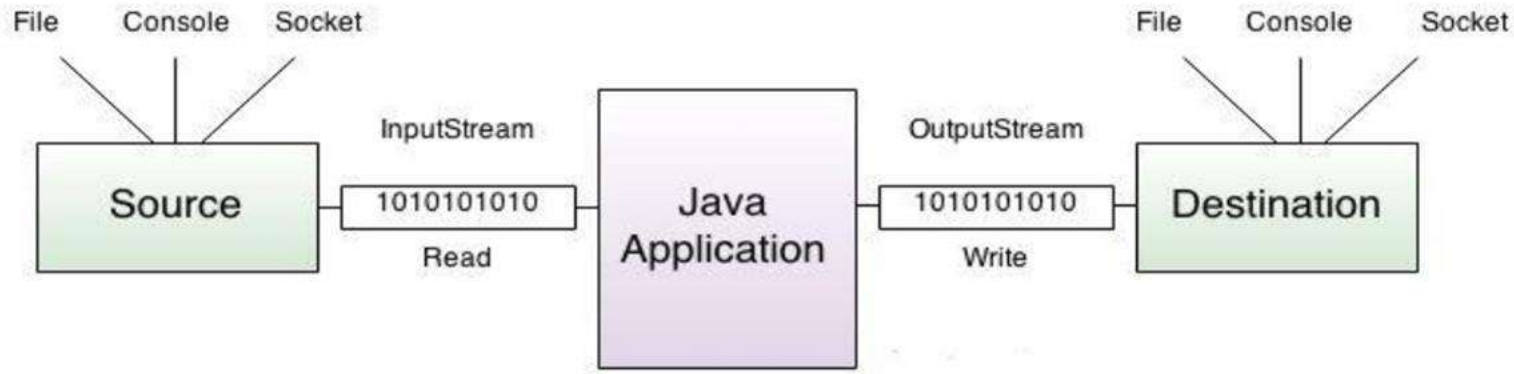
Streams

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.

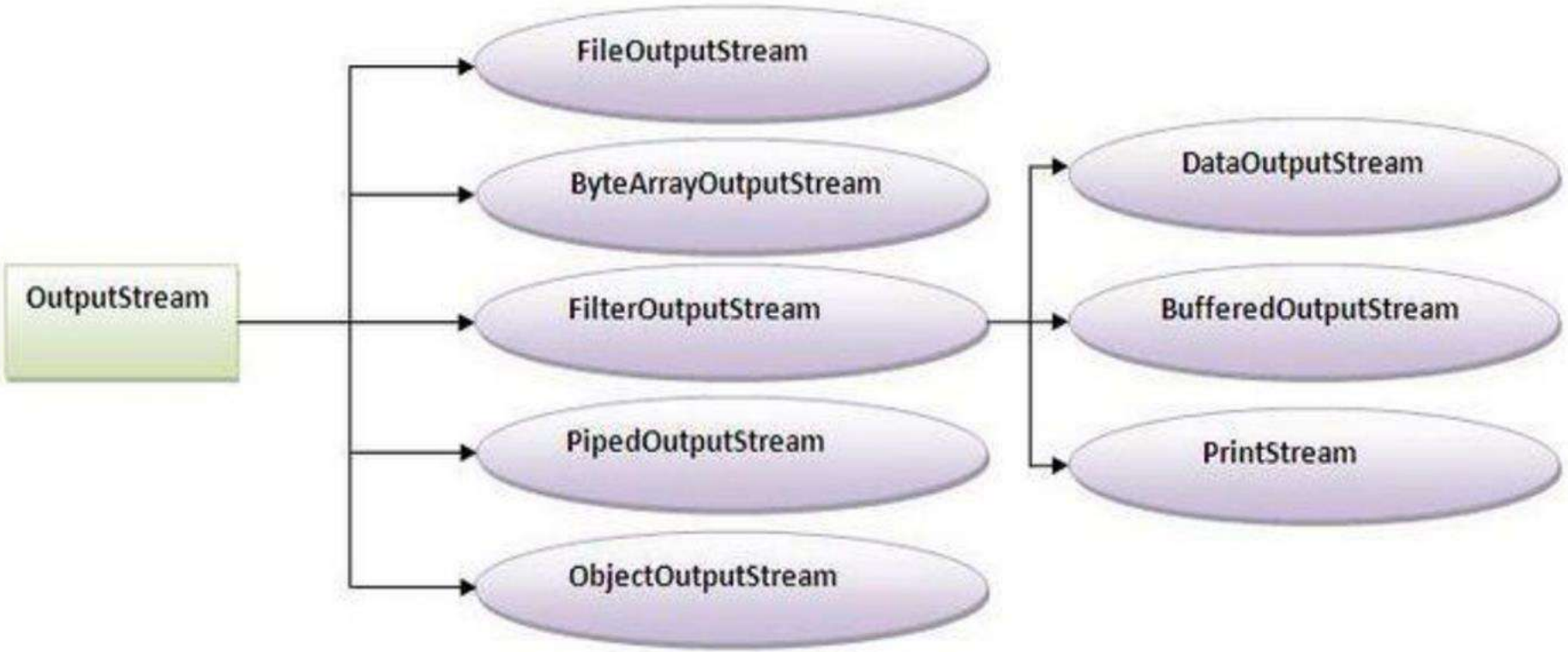


OutputStream class

- OutputStream class is an abstract class.
- It is the superclass of all classes representing an output stream of bytes.
- An output stream accepts output bytes and sends them to some sink.
- **Commonly used methods of OutputStream class**

Method	Description
1) public void write(int)throws IOException:	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException:	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException:	flushes the current output stream.
4) public void close()throws IOException:	is used to close the current output stream.

OutputStream Hierarchy

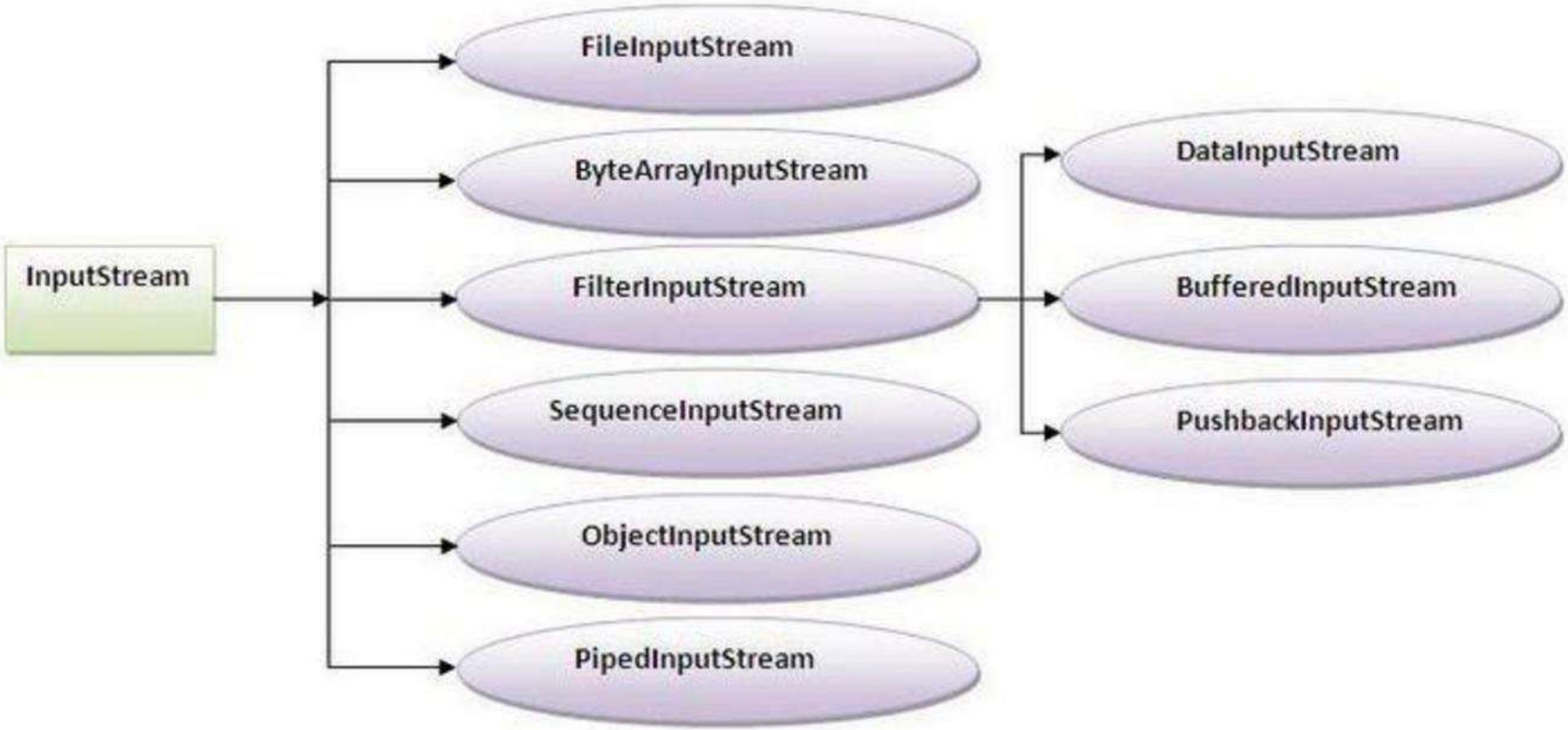


InputStream class

- InputStream class is an abstract class.
- It is the super class of all classes representing an input stream of bytes.

Method	Description
1) public abstract int read()throws IOException:	reads the next byte of data from the input stream.It returns -1 at the end of file.
2) public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException:	is used to close the current input stream.

InputStream Hierarchy



Console I/O

Java Console class

- The Java *Console* class used to get input from console.
- It provides methods to read text and password.

How to get the object of Console

- System class provides a static method console() that returns the unique instance of Console class.

```
public static Console console()
{    }
```

Code to get the instance of Console class.

- Console c=System.console();

Console I/O

Method	Description
1) public String readLine()	is used to read a single line of text from the console.
2) public String readLine(String fmt,Object... args)	it provides a formatted prompt then reads the single line of text from the console.
3) public char[] readPassword()	is used to read password that is not being displayed on the console.
4) public char[] readPassword(String fmt,Object... args)	it provides a formatted prompt then reads the password that is not being displayed on the console.

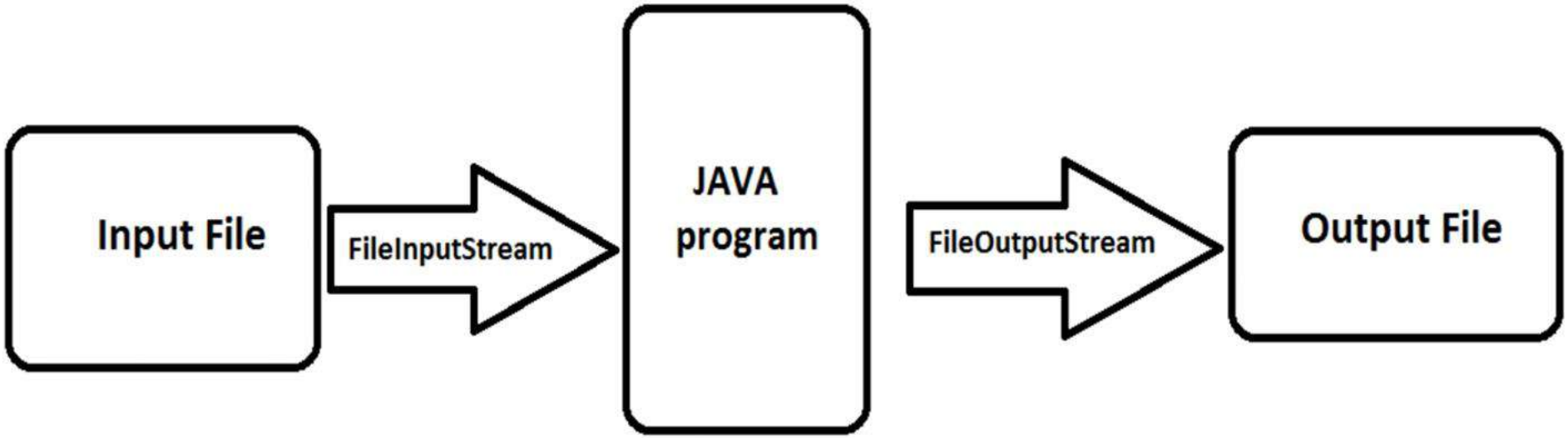
Let's understand this with example ……..

Scanner Class

- The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default.
- It provides many methods to read and parse various primitive values.
- Java Scanner class is widely used to parse text for string and primitive types using regular expression.

Method	Description
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.
public short nextShort()	it scans the next token as a short value.
public int nextInt()	it scans the next token as an int value.
public long nextLong()	it scans the next token as a long value.
public float nextFloat()	it scans the next token as a float value.
public double nextDouble()	it scans the next token as a double value.

File Handling (File I/O)



File Handling

Java FileOutputStream class

- Java FileOutputStream is an output stream for writing data to a file.
- To write primitive values , use FileOutputStream.

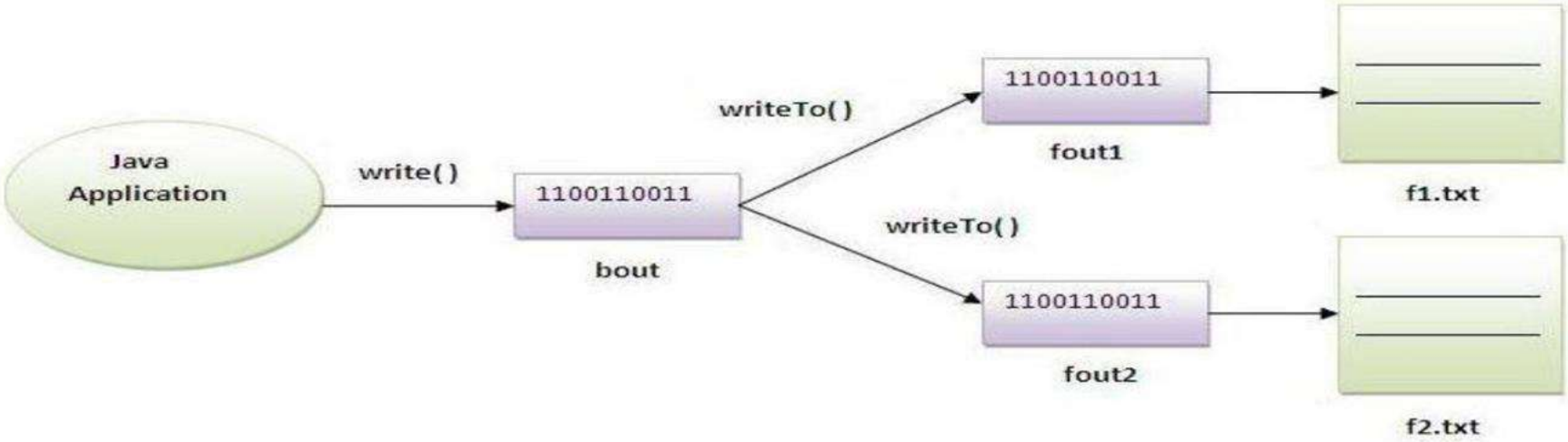
Java FileInputStream class

- Java FileInputStream class obtains input bytes from a file.
- It is used for reading streams of raw bytes such as image data.

File Handling

Java ByteArrayOutputStream class

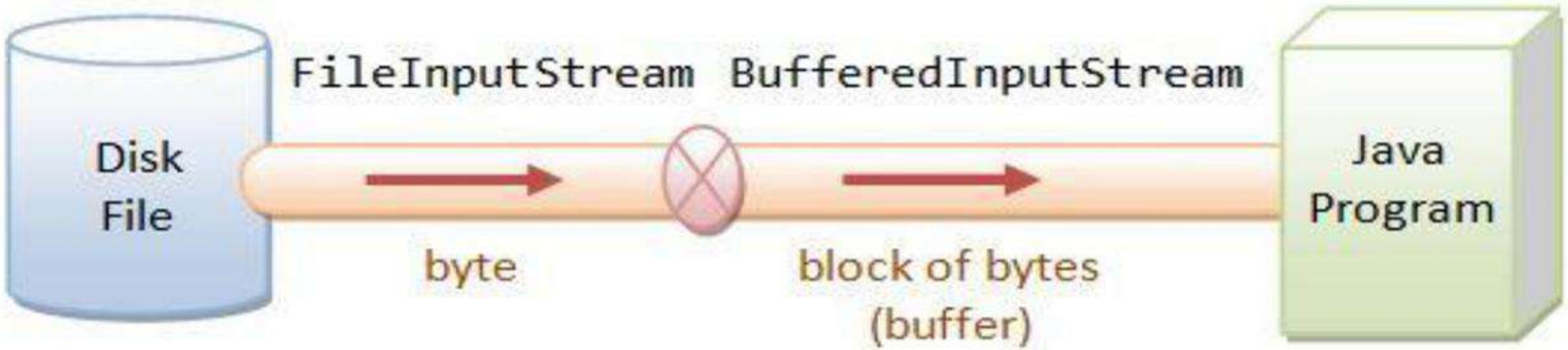
- Java ByteArrayOutputStream class is used to write data into multiple files.
- In this stream, the data is written into a byte array that can be written to multiple stream.
- The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.
- The buffer of ByteArrayOutputStream automatically grows according to data.



Java BufferedOutputStream and BufferedInputStream

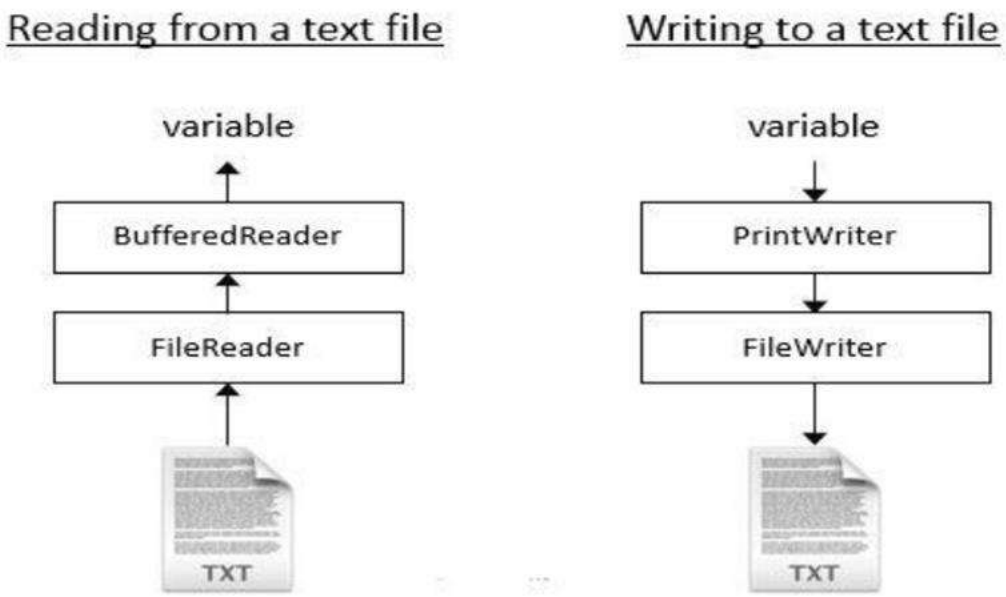
- Java *BufferedOutputStream* class uses an internal buffer to store data.
- It adds more efficiency than to write data directly into a stream.
- it makes the performance fast.

- Java *BufferedInputStream* class is used to read information from stream.
- It internally uses buffer mechanism to make the performance fast.



FileWriter and FileReader

- Java **FileWriter** and **FileReader** classes are used to write and read data from text files.
- These are character-oriented classes, used for file handling in java.



FileWriter

- Constructors of FileWriter class

Constructor	Description
FileWriter(String file)	creates a new file. It gets file name in string.
FileWriter(File file)	creates a new file. It gets file name in File object.

- Methods of FileWriter class

Method	Description
1) public void write(String text)	writes the string into FileWriter.
2) public void write(char c)	writes the char into FileWriter.
3) public void write(char[] c)	writes char array into FileWriter.
4) public void flush()	flushes the data of FileWriter.
5) public void close()	closes FileWriter.

FileReader

- Constructors of FileWriter class

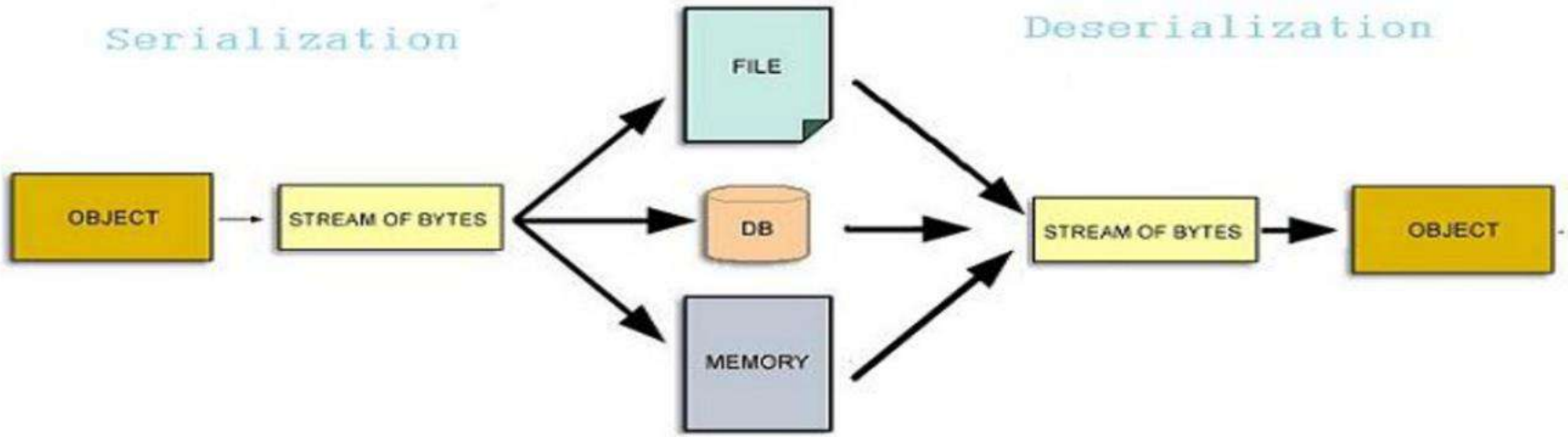
Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

- Methods of FileReader class

Method	Description
1) public int read()	returns a character in ASCII form. It returns -1 at the end of file.
2) public void close()	closes FileReader.

Serialization in Java

- Primary purpose of java serialization is to write an object into a stream, so that it can be transported through a network and that object can be rebuilt again.
- When there are two different parties involved, you need a protocol to rebuild the exact same object again.
- Java serialization API just provides the same.



Serialization

- Most impressive is that the entire process is JVM independent,

> An object can be *serialized* on one platform and *deserialized* on an entirely different platform.

- Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

Serialization

java.io.Serializable interface

- Serializable is a marker interface (has no body).
- It is just used to "mark" java classes which support a certain capability.
- It must be implemented by the class whose object you want to persist.

```
import java.io.Serializable;
public class Student implements Serializable
{
        int id;
        String name;
        public Student(int id, String name)
        {
                this.id = id;
                this.name = name;
  }}
```

Serialization

ObjectOutputStream class

- The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream.
- Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

public ObjectOutputStream(OutputStream out) throws IOException { }

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	flushes the current output stream.
3) public void close() throws IOException {}	closes the current output stream.

Deserialization

- Deserialization is the process of reconstructing the object from the serialized state.
- It is the reverse operation of serialization.

ObjectInputStream class

- An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

public ObjectInputStream(InputStream in) throws IOException { }

Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException{}	reads an object from the input stream.
2) public void close() throws IOException {}	closes ObjectInputStream.

Serialization

If a class implements serializable then all its sub classes will also be serializable.

```
class Person implements Serializable
{
        // code
}
class Student extends Person
{
        //code
}
```

Serialization

- If there is any static data member in a class, it will not be serialized because static is the part of class not object.

```
class Employee implements Serializable
{
        int id;
        String name;
        static String company="ABC IT Pvt Ltd";//it won't be serialized
        public Student(int id, String name)
        {
                this.id = id;
                this.name = name;
        }
}
```

Transient

Java Transient Keyword

- **Java transient** keyword is used in serialization.
- If you define any data member as transient, it will not be serialized.

```
class Student implements Serializable
{
   int id;
   String name;
   transient int age;
}
```

What is logging?

- "Logging" is producing messages that tell you what your program is doing
- It's not much different than using `System.out.println(...)`
- Log messages can go to the console, to a file, or one of several other places (e.g. sent over the Internet)
- You can use logging to help you debug a program
- You can use logging to produce a file when the user runs your program

Why use logging?

- Reality: Large programs always have bugs
  - JUnit testing can greatly reduce this problem, but it's impossible to see in advance all potentially useful tests
- Your program may go to customers (users)
- Typical error report from customer: "It doesn't work."
  - We love it when we have a customer who tries assorted things and gives us a detailed scenario in which the program fails
  - Such customers are rare
- Here's what you want to tell the typical customer:
  - "*Send us the log; it's in such-and-such a place*."

Basic use

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

private static Logger myLogger =
    Logger.getLogger("myPackage");

myLogger.log(Level.SEVERE, "Bad news!");
```

```
October 15, 2015 10:51:09 AM myPackage.Sync main
SEVERE: Bad news!
```

Logging levels and methods

- Level.SEVERE
- Level.WARNING
- Level.INFO
- Level.CONFIG
- Level.FINE
- Level.FINER
- Level.FINEST

- myLogger.severe(String ***msg***);
- myLogger.warning(String ***msg***);
- myLogger.info(String ***msg***);
- myLogger.config(String ***msg***);
- myLogger.fine(String ***msg***);
- myLogger.finer(String ***msg***);
- myLogger.finest(String ***msg***);

- These levels are ordered, so that you can set the level of severity that results in log messages
- However, the levels have no inherent meaning--they are what you make of them

Controlling logging levels

public void Logger.setLevel(Level *newLevel*)

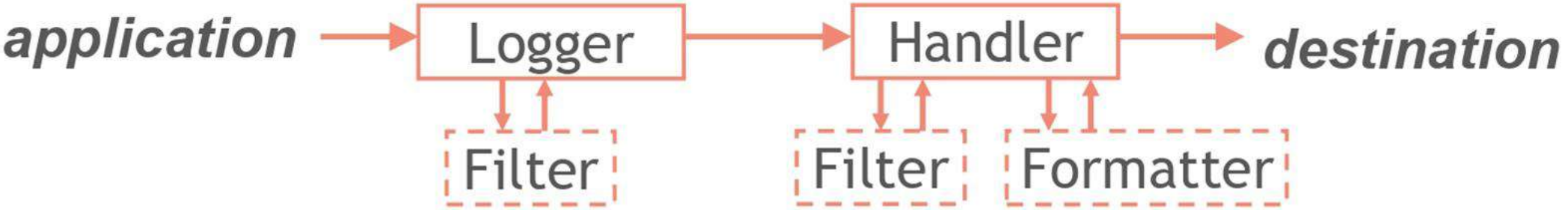
- Sets the logger to log all messages at ***newLevel*** or above
- Logger calls at lower levels don't do anything
- Example: logger.setLevel(Level.WARNING);

- Additional settings:
  - logger.setLevel(Level.ALL);
  - logger.setLevel(Level.OFF);

public Level getLevel()

- Note that this returns a Level, not an int
- Level has intValue() and toString() methods.

Logging flow of control



- You send your message to a Logger
- The Logger checks a Filter to see whether to ignore the message
- The Logger sends the message to a Handler to put the message somewhere
- The Handler checks another Filter to see whether to ignore this kind of message sent to this destination
- The Handler calls a Formatter to decide what kind of a text string to produce
- The Handler sends the formatted message somewhere

Logging formatting and destinations

- The JDK defines five Handlers:
  - StreamHandler: sends messages to an OutputStream
  - ConsoleHandler: sends messages to System.err (default)
  - FileHandler: sends messages to a file
  - SocketHandler: sends messages to a TCP port
  - MemoryHandler: buffers messages in memory
- It also defines two ways to format messages:
  - SimpleFormatter (default)
  - XMLFormatter
  - And, of course, you can define your own Handlers and Formatters
- As you can tell from the “Basic Use” slide earlier, you can ignore all of this and just use the defaults

Using a FileHandler

- try {
  logger.addHandler(new FileHandler("myLogFile"));
  }
  catch (SecurityException e) {
  e.printStackTrace();
  }

- The default Formatter for a FileHandler is XMLFormatter

What to log

- Here's what you *don't* want to log:
  - Everything!
  - Specific methods only *after* an error has occurred
- Here's what you *do* want to log:
  - "Critical events," such as updating a database
  - Results of a long sequence of operations
  - Interactions with other components of a large system
  - Actions known to be complex or error-prone

The rest of the story

You should also know that there are some alternative open source logging packages
Like

- Log4j
- Apache Commons Logging
- SLF4J
- tinylog
- Logback

You can use any of this open source powerful APIs for detailed use of Logging in your code.

Course Content

- Introduction and Basics of JAVA
- Class and Object Methodologies
- OOP flavors
- Exception Handling
- Threads in JAVA
- File and I/O
- Logical Data Storage
- JDBC –ODBC Concepts and XML / JSON Parsing

We are going to cover

- Basics of collection
- Collection Framework
- Hierarchy
- ArrayList
- Iterator
- HashMap
- Some facts
- When to Use

Collection

- **Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.
- All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.
- Java Collection simply means a single unit of objects.
- Java Collection framework provides many

  interfaces (Set, List, Queue, Deque etc.) and

  classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

Collection

What is Collection in java ?

- Collection represents a single unit of objects i.e. a group.

What is framework in java ?

- provides readymade architecture.
- represents set of classes and interface.
- is optional.

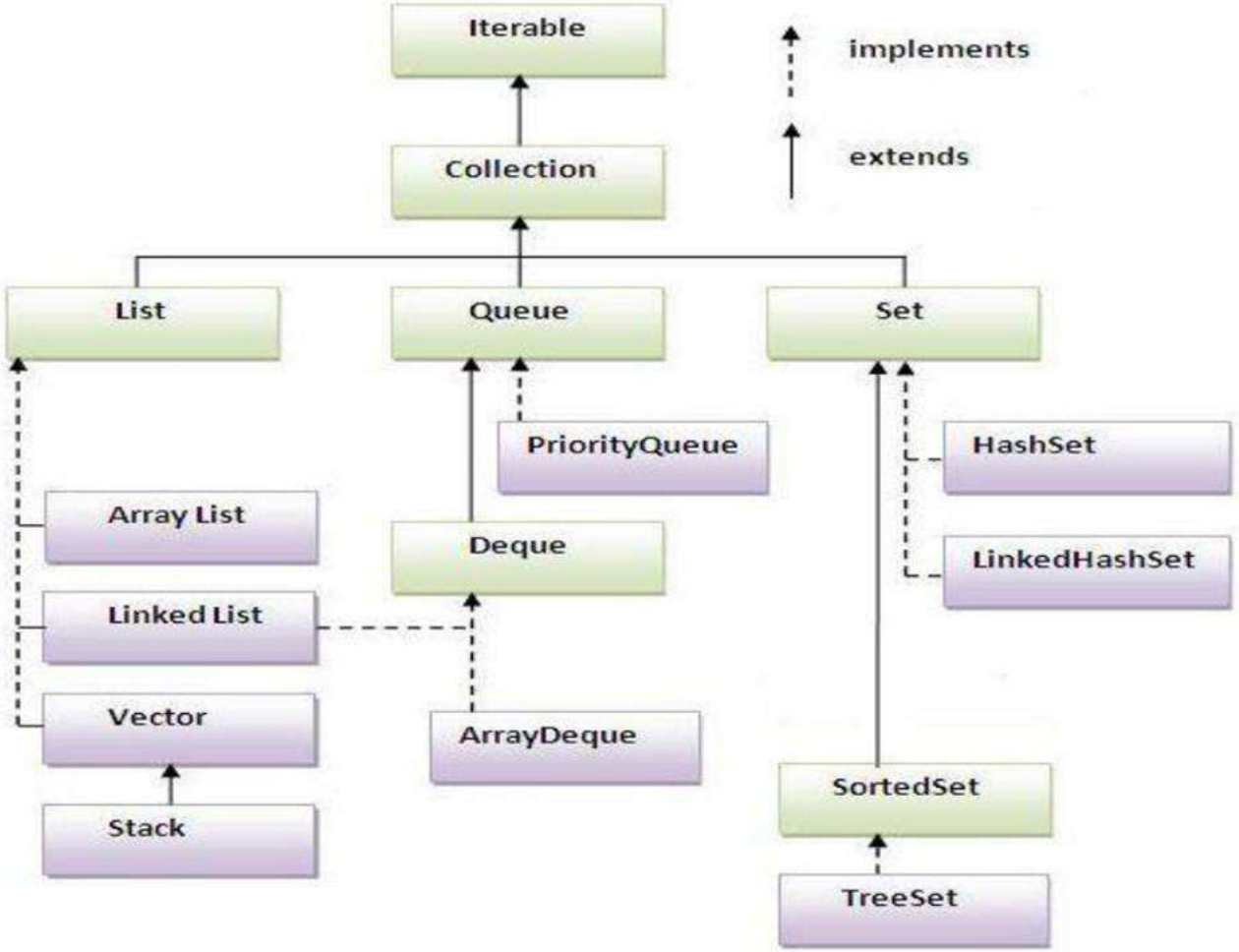
What is Collection framework

- Collection framework represents a unified architecture for storing and manipulating group of objects.

  It has:

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

Hierarchy


Iterable
implements
extends
Collection
List
Queue
Set
PriorityQueue
HashSet
Array List
Deque
LinkedHashSet
Linked List
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Methods in Collections

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element from the collection.

Methods in Collection

8	public boolean contains(object element)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collection.
14	public int hashCode()	returns the hashcode number for collection.

ArrayList

Java ArrayList class

- Java ArrayList class uses a dynamic array for storing the elements.
- It extends AbstractList class and implements List interface.
- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

ArrayList and Generics

Java Non-generic Vs Generic Collection

- Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.
- Java new generic collection allows you to have only one type of object in collection.
- Now it is type safe so typecasting is not required at run time.

Old non-generic example of creating java collection.

```
ArrayList al=new ArrayList();          //creating old non-generic arraylist
```

New generic example of creating java collection.

```
ArrayList<String> al=new ArrayList<String>();       //creating new generic arraylist
```

ArrayList

- In generic collection, we specify the type in angular braces.
- Now **ArrayList** is forced to have only specified type of objects in it.
- If you try to add another type of object, it gives *compile time error*.

Let us do it .....

Iterator

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

Methods of Iterator interface

There are only three methods in the Iterator interface.

public boolean hasNext()

it returns true if iterator has more elements.

public object next()

it returns the element and moves the cursor pointer to the next element.

public void remove()

it removes the last elements returned by the iterator. It is rarely used.

ListIterator

- ListIterator Interface is used to traverse the element in backward and forward direction.
- Commonly used methods of ListIterator Interface:

```
public boolean hasNext();
public Object next();
public boolean hasPrevious();
public Object previous();
```

Map interface

- A map contains values based on the key i.e. key and value pair.
- Each pair is known as an entry.Map contains only unique elements.
- Commonly used methods :

  public Object put(object key,Object value): is used to insert an entry in this map.

  public void putAll(Map map):is used to insert the specified map in this map.

  public Object remove(object key):is used to delete an entry for the specified key.

  public Object get(Object key):is used to return the value for the specified key.

  public boolean containsKey(Object key):is used to search the specified key from this map.

  public boolean containsValue(Object value):is used to search the specified value from this map.

  public Set keySet():returns the Set view containing all the keys.

  public Set entrySet():returns the Set view containing all the keys and values.

Map

Entry

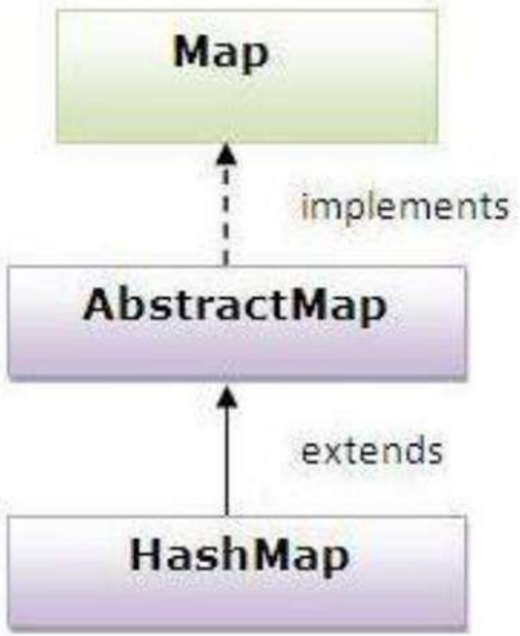
- Entry is the subinterface of Map.
- So we will access it by **Map.Entry** name.
- It provides methods to get key and value.

Methods of Entry interface:

public Object getKey()	: is used to obtain key.
public Object getValue()	:is used to obtain value.

HashMap Class

- A HashMap contains values based on the key.
- It implements the Map interface and extends AbstractMap class.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order.



Keep in Mind

- Some time question is asked as **When to use** List, Set and Map in Java.
- In order to decide when to use List, Set or Map , you need to know what are these interfaces and what functionality they provide.
- **List** in Java provides **ordered and indexed** collection which may contain duplicates.
- **Set** provides an **un-ordered collection of unique** objects, i.e. Set doesn't allow duplicates.
- **Map** provides a data structure based on **key value pair and hashing**.

All three List, Set and Map are **interfaces** in Java and there are many concrete implementation of them are available in Collection API.

When to use List, Set and Map in Java

- If you need to access elements frequently by using index, than List is a way to go. Its implementation e.g. ArrayList provides faster access if you know index.

- If you want to store elements and want them to maintain an **order** on which they are inserted into collection then go for List again, as List is an ordered collection and maintain insertion order.

- If you want to create collection of unique elements and **don't want any duplicate** than choose any Set implementation e.g. HashSet, LinkedHashSet or TreeSet.

- If you store data in form of key and value than Map is the way to go. You can choose from Hashtable, HashMap, TreeMap based upon your subsequent need.

General-purpose Implementations					
Interfaces	**Implementations**				
	Hash table	**Resizable array**	**Tree (sorted)**	**Linked list**	**Hash table + Linked list**
Set	**HashSet**		TreeSet **(sorted)**		LinkedHashSet
List		**ArrayList**		LinkedList	
Queue					
Map	**HashMap**		TreeMap **(sorted)**		LinkedHashMap

Course Content

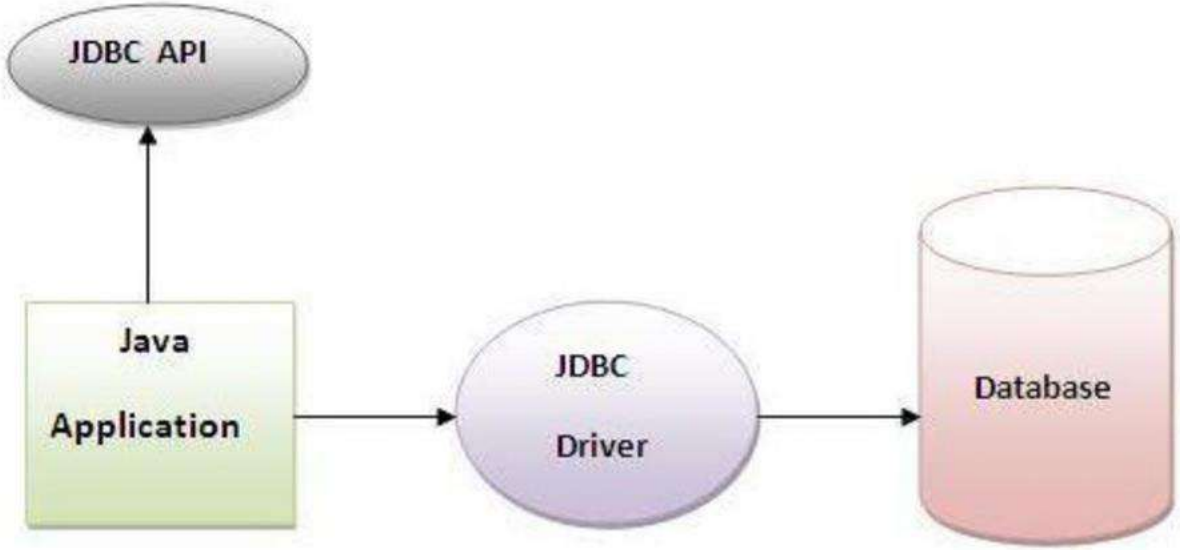
- Introduction and Basics of JAVA
- Class and Object Methodologies
- OOP flavors
- Exception Handling
- Threads in JAVA
- File and I/O
- Logical Data Storage
- JDBC –ODBC Concepts and XML / JSON Parsing

We will see ..

- JDBC concepts
- JDBC Drivers
- Steps to connect to the database in JAVA
- Introduction to SQL
- Basic Operations with MySql and JAVA
- Transaction Management
- Introduction to JSON parsing

JDBC

- Java JDBC is a java API to connect and execute query with the database.
- JDBC API uses jdbc drivers to connect with the database.



JDBC

Why use JDBC

- Before JDBC, ODBC API was the database API to connect and execute query with the database.
- But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured).
- That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

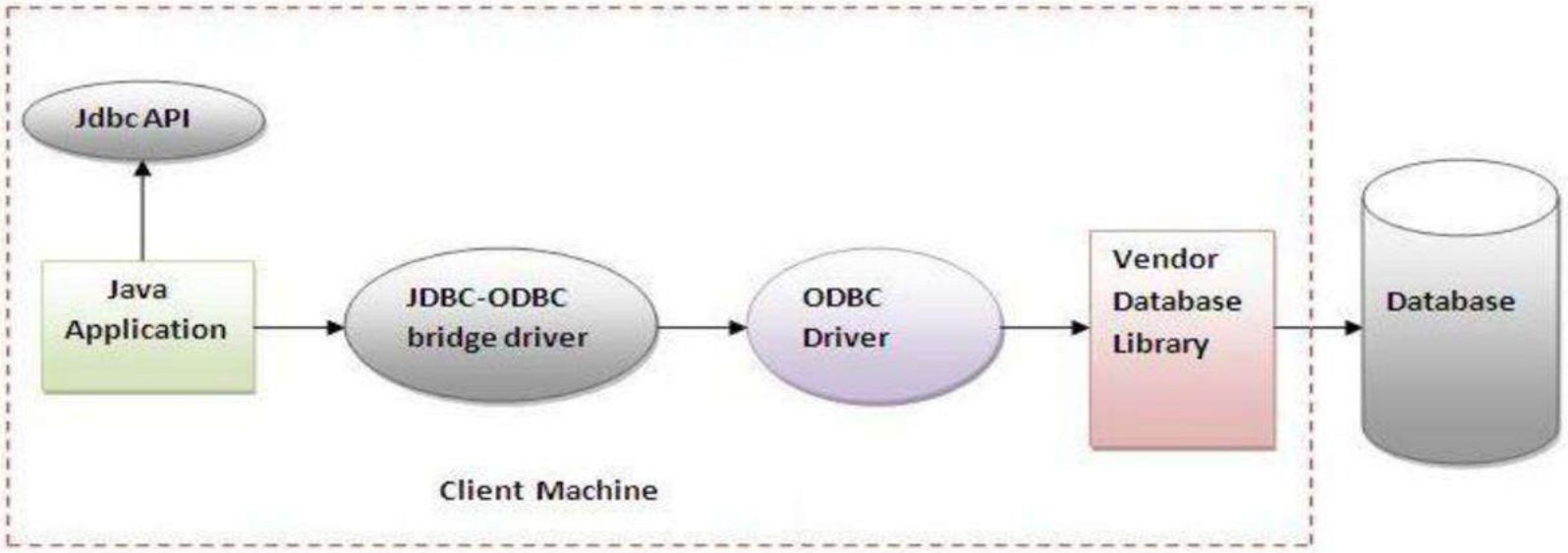
JDBC Driver

- JDBC Driver is a software component that enables java application to interact with the database.

- 4 types of JDBC drivers:
  - JDBC-ODBC bridge driver
  - Native-API driver (partially java driver)
  - Network Protocol driver (fully java driver)
  - Thin driver (fully java driver)

JDBC-ODBC bridge driver

- The JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
- The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
- This is now discouraged because of thin driver.



Continue…

Advantages:

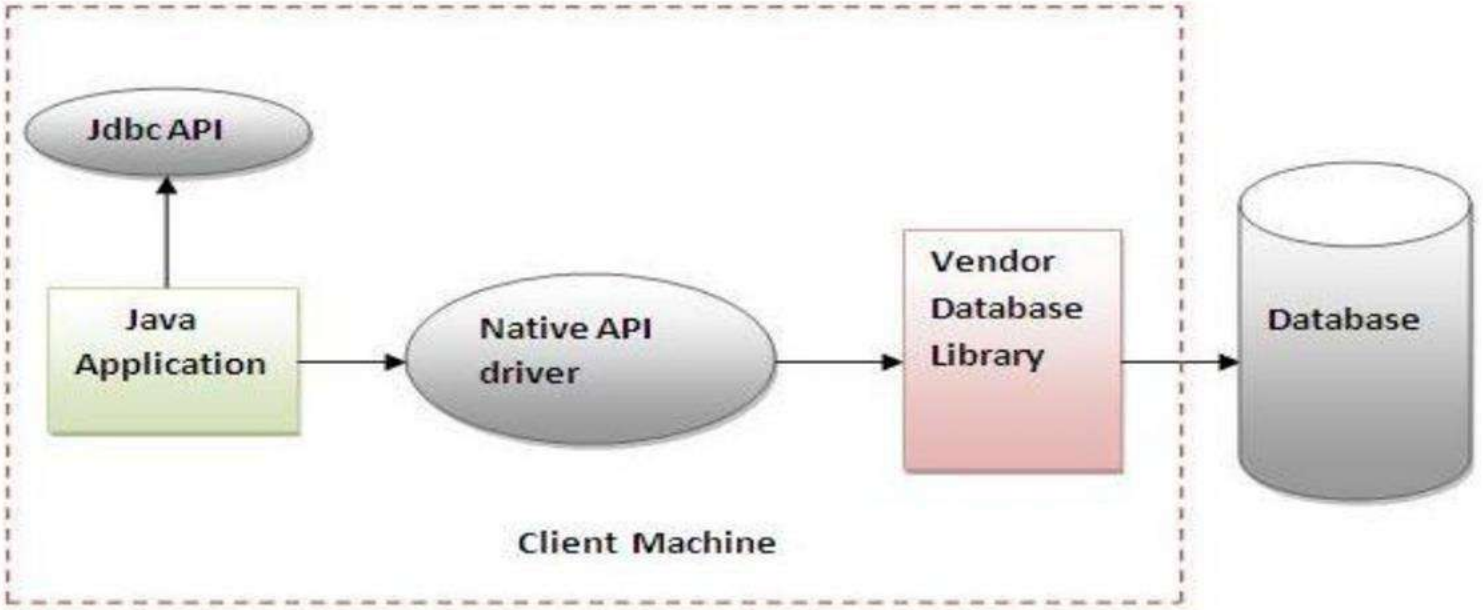
- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Native-API driver

- The Native API driver uses the client-side libraries of the database.
- The driver converts JDBC method calls into native calls of the database API.
- It is not written entirely in java.



Continue

Advantage:

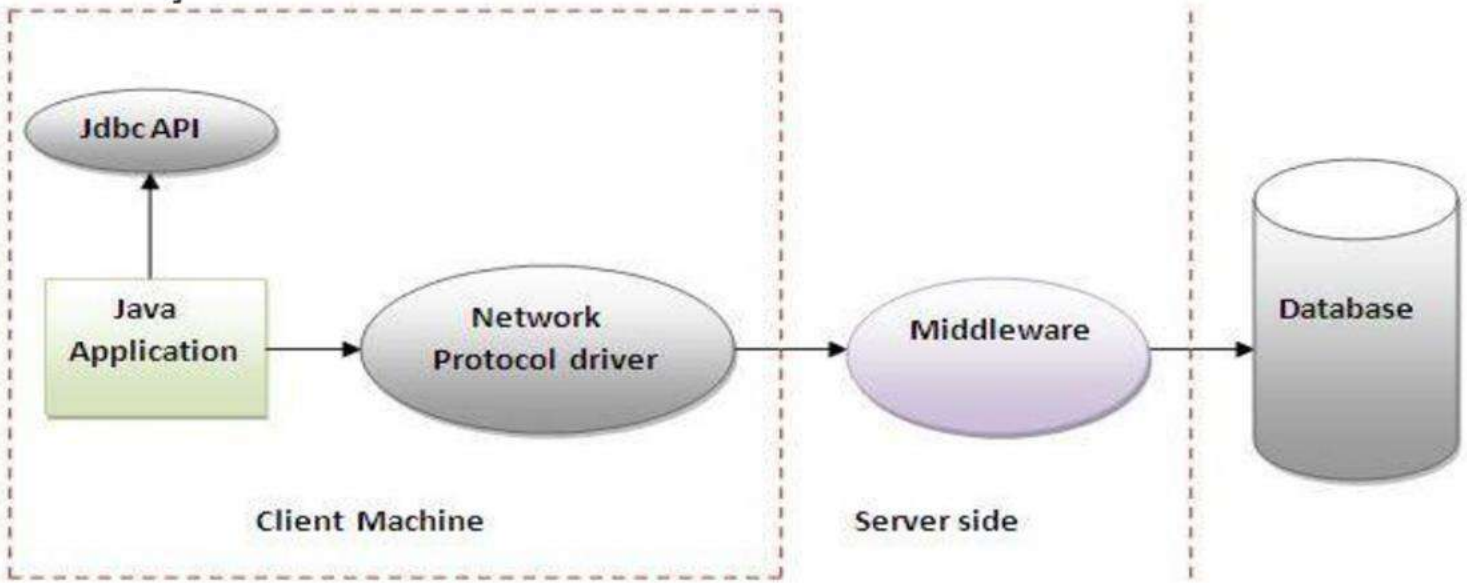
- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Network Protocol driver

- The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol.
- It is fully written in java.



Continue

Advantage:

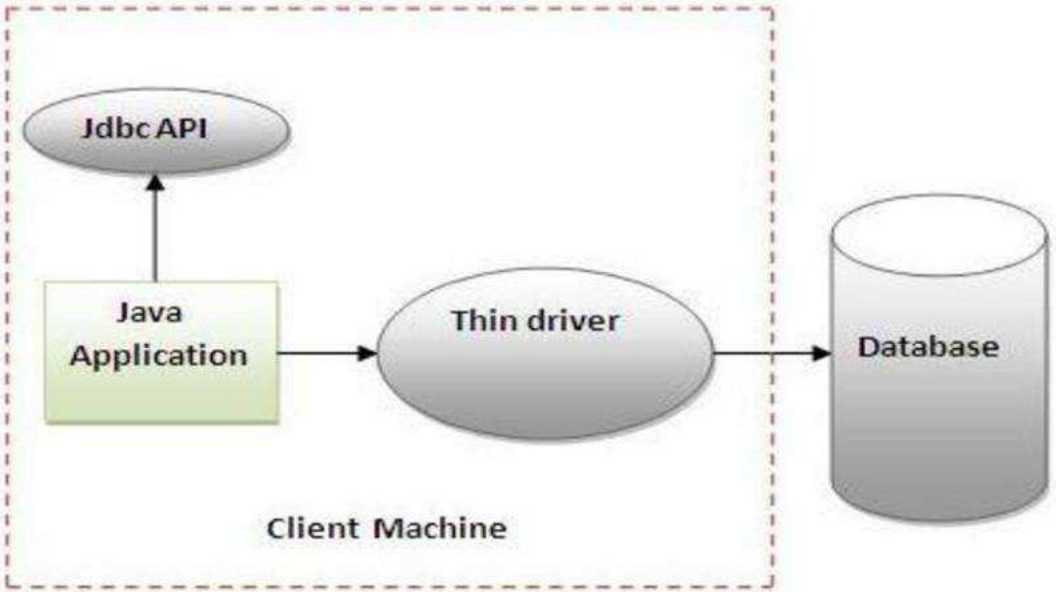
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Thin driver

- The thin driver converts JDBC calls directly into the vendor-specific database protocol.
- That is why it is known as thin driver.
- It is fully written in Java language.



Continue

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depends on the Database.

Steps to connect to the database in java

- There are 5 steps to connect any java application with the database in java using JDBC.
    - Register the driver class
    - Creating connection
    - Creating statement
    - Executing queries
    - Closing connection

Step 1

1) **Register the driver class**

The forName() method of Class class is used to register the driver class.
This method is used to dynamically load the driver class.

Syntax:

- public static void forName(String className) throws ClassNotFoundException

Example to register the OracleDriver class

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Step 2

2) **Create the connection object**

- The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax

public static Connection getConnection(String url)throws SQLException

public static Connection getConnection(String url,String name,String password) **throws** SQLException

Example to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection(  "jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

Step 3

3) Create the Statement object

- The createStatement() method of Connection interface is used to create statement.
- The object of statement is responsible to execute queries with the database.

Syntax

```
public Statement createStatement()throws SQLException
```

Example

```
Statement stmt=con.createStatement();
```

Step 4

4) Execute the query

- The executeQuery() method of Statement interface is used to execute queries to the database.
- This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax

```
public ResultSet  executeQuery (String sql) throws  SQLException
```

Example

```
ResultSet  rs = stmt.executeQuery("select * from emp");
while(rs.next())
{
        System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Step 5

5) Close the connection object

- By closing connection object statement and ResultSet will be closed automatically.
- The close() method of Connection interface is used to close the connection.

Syntax

```
public void close()throws SQLException
```

Example

```
con.close();
```

Introduction to SQL

- **S**tructured **Q**uery **L**anguage
- Originally called **SEQUEL** from Structured English QUEry Language
- SQL has a standard
- SQL provides both
  - Data Definition Language (DDL)
    - Create/alter/delete tables and their attributes
  - Data Manipulation Language (DML)
    - Query one or more tables
    - Insert/delete/modify tuples in tables
- DBMS may provide higher level approach, but SQL is important for tougher queries, using host programming or web programming.

Demo with MySql

- For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database
   is **jdbc:mysql://localhost/testmydb**
   where jdbc is the API,
   mysql is the database,
   localhost is the server name on which mysql is running, (we may also use IP address) and
   testmydb is the database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use "" as the password.

Continue

- To connect java application with the mysql database **mysqlconnector.jar** file is required to be loaded.
- Two ways to load the jar file:
    - paste the **mysqlconnector.jar** file in jre/lib/ext folder
    - set classpath

1) *paste the mysqlconnector.jar file in JRE/lib/ext folder*
    - Download the mysqlconnector.jar file.
    - Go to " jre/lib/ext folder " and paste the jar file here.

2) *set classpath:*
    - There are two ways to set the classpath:
        - temporary
        - permanent

Continue

How to set temporary classpath

- open comman prompt and write:
- C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;.;

How to set the permanent classpath

- Go to environment variable then click on new tab.
- In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;.; as

  C:\folder\mysql-connector-java-5.0.8-bin.jar;.;

Demo

Let us perform basic operations like insert , update and delete using java program into MySql DataBase

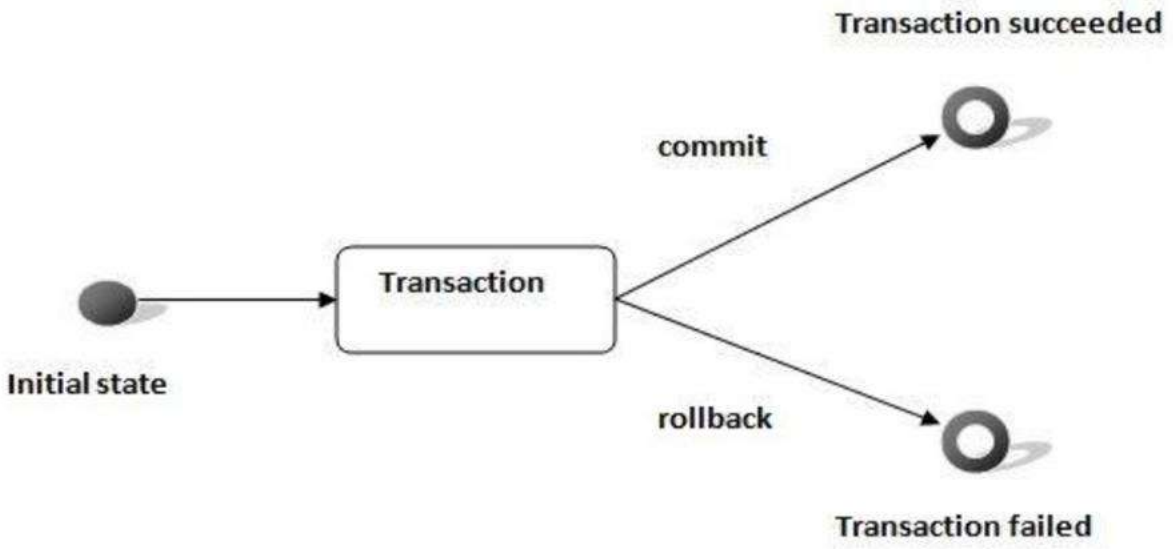
Transaction Management in JDBC

- Transaction represents **a single unit of work**.
- The ACID properties describes the transaction management well.
- ACID stands for Atomicity, Consistency, isolation and durability.
- **Atomicity** means either all successful or none.
- **Consistency** ensures bringing the database from one consistent state to another consistent state.
- **Isolation** ensures that transaction is isolated from other transaction.
- **Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Transaction Management

Advantage of Transaction Management

- fast performance
  - It makes the performance fast because database is hit at the time of commit.



Transaction

In JDBC , Connection interface provides methods to manage Transactions

Method	Description
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

JSON

- **J**ava **S**cript **O**bject **N**otation
- JSON or JavaScript Object Notation is a lightweight text-based open standard designed for human-readable data interchange.
- Conventions used by JSON are known to programmers, which include C, C++, Java, Python, Perl, etc.
- The format was specified by Douglas Crockford.
- It was designed for human-readable data interchange.
- It has been extended from the JavaScript scripting language.
- The filename extension is **.json.**

JSON

Uses of JSON

- It is used while writing JavaScript based applications that includes browser extensions and websites.
- JSON format is used for serializing and transmitting structured data over network connection.
- It is primarily used to transmit data between a server and web applications.
- Web services and APIs use JSON format to provide public data.
- It can be used with modern programming languages.

JSON

Characteristics of JSON

- JSON is easy to read and write.
- It is a lightweight text-based interchange format.
- JSON is language independent.

Let's see how JSON looks ……..

Simple Example in JSON

- The following example shows how to use JSON to store information related to books based on their topic and edition.

```
{ "book":
    [
        {
            "id":"01",
            "language": "Java",
            "edition": "third",
            "author": "Herbert Schildt"
        },
        {
            "id":"07",
            "language": "C++",
            "edition": "second"
            "author": "E.Balagurusamy"
        }
    ]
}
```

Data is represented in name/value pairs.

Curly braces hold objects and each name is followed by ':'(colon),

The name/value pairs are separated by , (comma).

Square brackets hold arrays and values are separated by ,(comma).

JSON encoding and decoding using JAVA

- Before you start with encoding and decoding JSON using Java, you need to install any of the JSON modules available.
- For this demo I have downloaded and installed JSON.simple and have added the location of **json-simple-1.1.1.jar** file to the environment variable CLASSPATH.

- There are lots of JSON Modules available as open source , you can use any of them.

Let's encode – decode JSON using JAVA