Prasanna wadekar 9970248407 36=00



# Core Java Notes

Day 01 (5 hours)

=======

**+ Java Books:**

- Core Java Vol 1 & 2 - Hoarsman
- SCJP Programmer's guide - Khalid Mughal
- Java Programming Language - Jame Gonsling
- Inside JVM -

**+ Java Programming Language:**

- Invented by "James Gosling"
- Company "Sun Microsystem" -> "Oracle Inc".
- Originally Java was invented for embedded systems. In fact first product developed using Java technology was a "Remote Control" named as "*7".
- However in spite lot of efforts from Java team, Java was not considered by industry at that time.
- In 1990's people started using internet. However internet support was limited to static web pages.
- Around 1995 Java use applets as interactive java programs that can be executed on client browsers. In short span of time, applets and hence Java become popular for internet.
- There are three main streams were developed using Java language:

1. Java SE:
   - Desktop based java applications (Console & GUI applications).
   - Core language part of Java i.e. OOP and other features.
   - Versions pulished:
     - Java 1.0
     - Java 1.1
     - J2SE 1.2 -> Added collection framework
     - J2SE 1.3
     - J2SE 1.4 -> another stable and popular Java
     - JavaSE 5.0 -> Added generics, annotations,
     - JavaSE 6
     - JavaSE 7 -> Added Parallel programming,...
     - JavaSE 8
2. Java EE:
   - Web based / Enterprise application
   - Servlets, JSP, EJB, Struts, Spring
   - Versions published:
     - J2EE 1.3
     - J2EE 1.4

PRASANNA

- JavaEE 5.0
- JavaEE 6
- JavaEE 7

3. Java ME:
   - Java for embedded devices.

**+ C/C++ Compilation & Execution:**

[.c/.cpp] -> Compiler -> [.obj] -> Linker -> [.exe]

| Source Code | Object Code | Executable |
|---|---|---|
| | Machine Instru | Machine Instr |
| | | \| |
| | | Operating Sys |
| | | \| |
| | | Processor (HW) |

**+ Java Compilation & Execution:**

Hello.java

---

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

**- COMPILATION COMMAND: javac Hello.java**
   - Name of .java file (with extension)
   - .class file is always created as per class-name (not .java file name).
   - If single .java file contains multiple java classes, then multiple .class files will be created.

**- EXECUTION COMMAND: java Hello**
   - Name of .class file (without extension)
   - It is expected that in this class, main() is prsent

**[Hello.java] -> Compiler -> [Hello.class]**

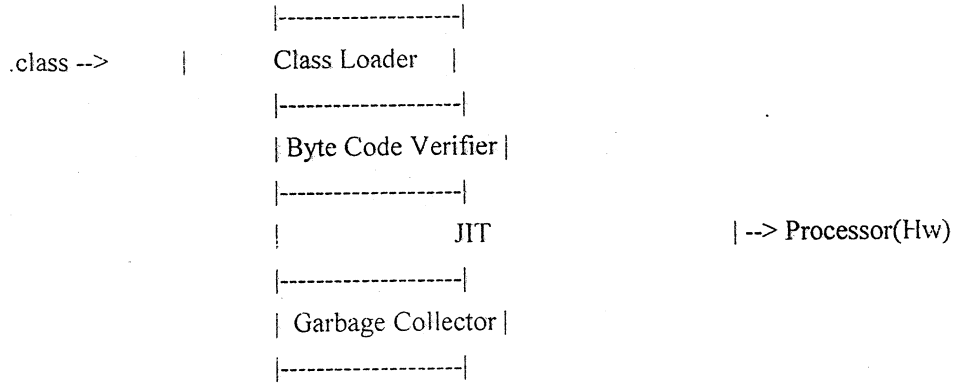
| Source code | Java Executable |
|---|---|
| Java Code | Byte Code |
| | \| |

JVM
|
[Machine Instr]
|
Processor(HW)

**+ JVM, JRE and JDK:**

- Java Virtual Machine:
  - JVM is hypothetical machine assumed by Java compiler and is emulated in software as part of JRE.
  - Machine level instructions of JVM are of one byte each instruction and hence it is called as "byte code".
  - java ClassName
  - Here "java" is application launcher.

- JVM Architecture:




**- Java Runtime Environment:**

- JRE is required for execution of java programs (.class) files.
- JRE = JVM + java tools + java libraries (.jar)
- JRE can be downloaded from oracle site for particular OS platform. In other words, JRE is platform depenedent.

**- Java Development Kit:**

- JDK is required for developing java applications.
- JDK = JRE + java dev tools (compiler, debugger,...)
- JDK can be downloaded from oracle site for particular OS platform. In other words, JDK is platform depenedent.

**+ Java Features:**

- Portable:
- Compile Once and Run Anywhere
- Java code is compiled into .class files. These .class files can be executed on any OS for which JRE is present.
- Even though same .class can run on many platforms (OS), for certain features behaviour will be OS dependent e.g. File IO, Threads, Sockets, AWT, ...
    - Thus java is not purely "Platform Independent".
- Architecture Neutral:
    - JVM (JIT) convert byte code into native code as per underlying CPU architecture.
    - So same java code can be executed on any CPU arch for which JRE is present.

**+ Reading work:**

- Core Java Vol 1
    - Java History
    - Java Buzzwords

**+ Java Code compilation and Execution:**

- PATH is an Ennvironment Variable of the OS. It contains list of directories seperated by semicolon ; (on windows) or colon : (on UNIX/Linux).
- When any program is executed on command line without giving its full path, OS will search it into all directories mentioned in PATH variable.
- If you want to execute any program without giving its full path, you must add its directory path into PATH variable.

**- To see current PATH variable:**

[WIN] set PATH

[LINUX] echo $PATH

**- To add a dir path in in PATH variable:**

[WIN] set PATH=new_dir_path;%PATH%

[LINUX] export PATH=new_dir_path:$PATH

**- To permantly set PATH variable:**

[WIN] My Computer -> Properties -> Advanced -> Environment Variable -> Select PATH -> click EDIT -> add new_dir_path follwed by ; (do not remove/overwrite existing PATH).

[LINUX] in .bashrc file (in your home dir) add cmd:
export PATH=new_dir_path:$PATH

**- To unset/nullify PATH variable:**

[WIN] set PATH=

**[LINUX] export PATH=**

- **To add path of java compiler (javac) into PATH var:**

  [WIN] set PATH=C:\Program Files\Java\jdk1.xyz\bin;%PATH%

  [LINUX] export PATH=/usr/java/jdk1.xyz/bin:$PATH

- **javac -d dirpath FileName.java**
  - In this example, given Java file will be compiled and .class file(s) will be created in the give dirpath.
  - e.g. javac -d . Hello.java
    -> will create Hello.class in current directory.
  - e.g. javac -d D:\temp Hi.java
    -> will create Hi.class in D:\temp directory.

- **java ClassName**
  - This will start java application laucher (java.exe), load JVM in it and it will execute given .class file.
  - After loading and verifying class, execution begin from main() method.
  - If proper main() is not found, JVM will throw error.

**+ Hello world example:**

```
import java.lang.System;
class Hello {
        public static void main(String[] args) {
                System.out.println("Hello, world!");
        }
}
```

- In java, global data/methods are not allowed. Even main() method must be declared within some class.
- When program exxecution begins, JVM calls/executes main() method.
- Here we expect that main() method should be invoked from outside the class (JVM) and hence we declare it as public.
- JVM invoke main() method directly without creating object of the class and hence main() should be declared as static.
- main() does not return any value, so its return type is void.
- main() method can take command line arguments and hence it has an argument : String[] args
- System.out.println("Hello, world!");
  - Here "System" is a pre-defined class
  - "out" is public static member of that class.
  - "out" is of type of "PrintStream" class.

- println() is a method in "PrintStream", which is used to print a given message on the screen followed by a newline character.

- System.in encapsulates stdin.

- System.out encapsulates stdout.

- System.err encapsulate stderr.

- System.out.printf() can also be used to print formatted output on the screen just like C language printf() function. Supported format specifiers are %d, %f, %s, %c.

**+ Java primitive types:**

- Primitive types are not classes in java to improve the performance.
- Integer types:
  - byte (1 byte) - unsigned : 0 to 2^8-1
  - short (2 byte) - signed : -2^15 to +2^15-1
  - int (4 byte) - signed : -2^31 to +2^31-1
  - long (8 byte) - signed : -2^63 to +2^63-1
- Real (numbers) types:
  - float (4 bytes)
  - double (8 bytes)
- character types:
  - char (2 bytes) : unicode characters : 'A'
  - * String- set of characters
    - "Hello" - string const or string literal
- boolean type:
  - boolean (1 bit) : true or false

- Since java is strictly type-checking only a few conversions are allowed (not all).
- Conversion from boolean to integer type and vice-versa is not supported directly.
- Conversion from char to integer type and vice-versa is not supported directly.
- Conversion from narrow data type to wider data type does not need type-casting. However conversion from wider data type to narrow data type needs type-casting.

```
short a = 10;
int b = a;              // OKAY
int x = 10;
short y = x;    // ERROR
short z = (short)x;     // ALLOWED
```

**+ Java Classes and Objects:**

- Class is user-defined data type.
- Class contains fields and methods.
- Variables of class type are called as "Objects".
- In java, objects can be created only on heap using "new operator". In other words, objects cannot be created on stack.

```
- public static void main(String[] args) {
        Person p1 = new Person();
        // ...
}
```

- "p1" is a reference, keeping address of Person object created on heap.
- In java reference can be "null" or can be initialized after declaration.
- If local reference is used before initialization, compiler will raise error.

```
Person p1;
p1.display();     // error
```

- Accessing members on null reference will raise "NullPointerException" (at runtime).

```
Person p1=null;
p1.display();     // exception
```

- **Methods:**
  - **Constructor:**
  - Same name as that of class, no return type.
  - Used to initialize fields.
  - If not implemented, each class will have a default paramterless constructor given by the compiler.
    - Parameterless constructor
    - Parameterized constructor

```
Person() {
        this("", "", 0); // call to param ctor
}
Person(String name, String address, int age) {
        // ...
}
```

- In java, one constructor of class can be accessed from another constructor of that class using "this" keyword.
- However, such constructor call must be first line of the constructor; otherwise compiler raise error.
  - In java, if fields are not initialized:
- Primitive type variables will be initialized to default value e.g. int=0, boolean=false, etc.

- reference variables will be initialized to null.

- In java, fields can be initialized at the point of declaration in the class.

```
class Person {
        String name = "nitin";
        int age = 40;
        // ...
}
```

- Object initializer syntax/block:

```
class Person {
        String name;
        int age;

        // object initializer block
        {
                this.name = "something";
                this.age = 10;
        }

        // ...
}
```

- When a new object is created, all values initialized at point of declaration of fields will be assigned.
- Then object initializer block(s) will be executed in the order of their declaration in the class.
- Finally constructor is executed as per creation of the object.

```
p1 = new Person(); // paramless
p2 = new Person("","",2); // parameterized
```

* Java uses Hungarian notation for class names, while camel notation for field/method names.

- getters/inspectors:

```
int getAge() {
        return this.age;
}
```

- setters/mutators:

```
void setAge(int age) {
```

```
            this.age = age;
        }
```

- facilitators:

    display() / accept()

+ User input:

- Using "Scanner" class one can get input from end user.
- This class is defined in "java.util" package (not in default "java.lang" package).
- Hence before using this class, importing package is mandetory (as shown in example).

- import java.util.Scanner;

```
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.printf("Enter full name : ");
        String name = sc.nextLine();
        System.out.printf("Enter name : ");
        String name = sc.next();
        System.out.printf("Enter roll : ");
        int roll = sc.nextInt();
        System.out.printf("Enter marks : ");
        double marks = sc.nextDouble();
        sc.nextLine(); // flush \n char
        System.out.printf("Enter School name : ");
        String school = sc.nextLine();
        // ...
        sc.close();
    }
}
```

Day 02 (5 hours)

========

+ Access Modifiers:

- Class members can have one of the following access modifier:
    1. public
    2. protected
    3. private

4. default (package)

- Usually fields are declared as private and methods are declared public.

```
- class Person {
        private String name;
        private int age;
        public Person() {
                // ...
        }
        public void display() {
                // ...
        }
}
```

**+ Compilation and Execution Standard Practice:**

[project]
|- [src]
|  |- *.java
|- [classes]
   |- *.class

0. Create directory structure as above & save .java files into "src" directory.
1. Open command prompt and go to "src" directory.
2. CMD> javac -d ..\classes Main.java
3. CMD> cd ..\classes
4. CMD> java Main

**+ static keyword in java:**

- Static members do not belong to the object, they belong to the class.
- Static members can be accessed directly by using class name and dot (.) operator.
- Hence to access static members there is no need to create object of the class.
- However, Like C++, java allows access to the static members using dot operator on objects.

**+ static fields**

- Static fields do not contribute to size of object.
- The data that is common for all objects of the class should be declared as static.

**+ static methods**

- Static methods do not receive implicit "this" pointer and hence they cannot access non-static members of the class directly.

- Static methods can access static members of the class.
- Static methods are written to manipulate static fields in the class.
- Static methods are sometimes implemented as entry-point to the class. In other words, they are used to create objects of class.
    - e.g. Calendar cal = Calendar.getInstance();
        - Creating object using such static method in the class is called as "static factory pattern".

```
+ static block
    - class Sample {
        // ...
        static {
            // ...
        }
        // ...
    }
```

- When class is loaded first time into JVM, class-loader invokes static block(s) declared in the class.
- There can be one or more number of static blocks declared in the class.
- When class is accessed first time, all these static blocks will be invoked in the order of their declaration in the class.
- Static blocks are used to perform one-time initialization for the "class".
- Initialization of static members when class is accessed first time.
- JDBC driver registration must be done once before establishing database connection.

e.g. **Singleton class:**

```
- class Singleton {
    static Singleton obj = null;
    public static Singleton getInstance() {
        if(obj==null)
            obj = new Singleton();
        return obj;
    }
    // fields
    private Singleton() {
    }
    // methods
}

- class Singleton {
```

```
        static Singleton obj = null;
        static {
            obj = new Singleton();
        }
        public static Singleton getInstance() {
            return obj;
        }
        // fields
        private Singleton() {
        }
        // methods
    }
    class Main {
        public static void main(String[] args) {
            Singleton s1 = Singleton.getInstance();
            Singleton s2 = Singleton.getInstance();
            // ...
        }
    }
    + static member class
        - Will be discussed with "inner classes".

+ Composition:
    - Represents "has-a" relationship between the objects.
    - class Date {
        private int day, month, year;
        // ...
    }
    - class Book {
        private String name, author;
        private double price;
        private int pages;
        private Date publish;
        public Book() {
            this.name = "";
            this.author = "";
            this.publish = new Date();
```

```
        }
        public Book(String name, String author, double price, int pages, int
                day, int month, int year) {
            this.name = name;
            this.author = author;
            this.price = price;
            this.pages = pages;
            this.publish = new Date(day,month,year);
        }
        public Book(String name, String author, double price, int pages,
                                                        Date publish) {
            this.name = name;
            this.author = author;
            this.price = price;
            this.pages = pages;
            this.publish = publish;
        }
        public void display() {
            System.out.printf("%s\n%s\n", name, author);
            System.out.printf("%f\n%d\n", price, pages);
            publish.display();
        }
}
class Main {
        public static void main(String[] args) {
            Book b1 = new Book();
            b1.display();

            Book b2 = new Book("book1", "au1", 100.00, 30, 1, 1, 2014);
            b2.display();

            Date d = new Date(2,2,2014);
            Book b3 = new Book("bk2", "au2", 200.00, 40, d);
            b3.display();
            Book b4 = new Book("bk4", "au4", 20.0, 10, new Date(3,32014));
            b4.display();
        }
}
```

+ **Arrays:**

- In java, arrays are objects. So they must be created on heap using "new" operator.
- Syntax:
  - `int[] arr;`
    - Just declares a reference to array of ints.
  - `int[] arr = new int[5];`
    - By default array elements are initialized to zero.
  - `int[] arr = new int[5] { 11, 22, 33, 44, 55 };`
    - Initializing array elements at point of declaration.
- Internally array is object of "java.lang.Array" class and it holds array elements and also a "length" field to represent number of elements in the array.
- To access array elements:

```
- for(i=0; i < arr.length; i++)
        System.out.printf("%d\n", arr[i]);
- for(int n : arr)
        System.out.printf("%d\n", n);
```

- If you try to access array element at invalid index (index < 0 or index >= length), then exception will be thrown.
- When we create array of objects, internally array of references is created. By default each reference is "null".
- Programmer must initialized each reference to a valid object to avoid "NullPointerException".

```
- Date[] arr = new Date[5];
  arr[0] = new Date(1,1,2000);
  arr[1] = new Date(1,1,2001);
  arr[2] = new Date(1,1,2002);
  arr[3] = new Date(1,1,2003);
  arr[4] = new Date(1,1,2004);
  for(int i=0; i<arr.length; i++)
      arr[i].display();
```

- Each java program receives a set of command line arguments given by user while executing program on command line, in form of an array argument to main() function.
- Unlike C/C++, 0th element of the array is not name of executable / .class file.
- Main.java -> class Main ->

```
public static void main(String[] args) {
    for(int i=0; i<args.length; i++)
        System.out.printf("%s\n", args[i]);
}
```

- If program is executed like:

  java Main Pune Karad Satara Sangli

- In this case, "args" will be array of 4 elements

  0 -> Pune

  1 -> Karad

  2 -> Satara

  3 -> Sangli

+ Ragged Arrays:

- In java, we can create array of array objects. Size of each array object may or may not be same.

```
for(int i=0; i<arr.length; i++) {
    for(int j=0; j<arr[i].length; j++) {
        System.out.printf("%d ", arr[i][j]);
    }
    System.out.println();
}
```

+ Java Method Arguments:

- In java primitive types are always passed by value.

```
public static void swap(int a, int b) {
    int t = a;
    a = b;
    b = t;
    System.out.printf("a:%d, b:%d\n", a, b);//20 10
}
public static void main(String[] args) {
    int x=10, y=20;
    System.out.printf("x:%d, y:%d\n", x, y);//10 20
    swap(x, y);
    System.out.printf("x:%d, y:%d\n", x, y);//10 20
}
```

- In java objects are always passed by reference.

```
public static void change(Date d) {
    d.setDay(24);
}
public static void main(String[] args) {
    Date d1 = new Date(12, 6, 1980);
```

```
        d1.display();    // 12/6/1980
        change(d1);
        d1.display();    // 24/6/1980
    }
```

- Variable Arity Method:

```
    public static int add(int... arr) {
        int sum=0;
        for(int i=0; i<arr.length; i++)
            sum += arr[i];
        return sum;
    }
```

- Above declared function can take variable number of integer arguments.
- add(1,2);
- add(1,2,3,4);
- Internally compiler creates an array of all arguments passed to the method and that array is passed as an argument to method.
- We can also pass array of ints to this method.

- To pass variable number of args of any data type the method should be declared as :

```
void method(Object... arr) {
    // ...
}
```

- Since "java.lang.Object" is base/super class for all java classes, it can hold objects of any type.

**+ Inheritance:**

- Inheritance represent "is-a" relationship.
- All members of "super" class are inherited to the "sub" class. Sub-class can have its own additional members.
- **Inheritance Types:**
  - Single
  - Multiple
  - Java does not support "multiple implementation inheritance". In other words, a class cannot be inherited from multiple super classes.
  - Java does support "multiple interface inheritance". In other words, a class/interface can be inherited from multiple "interfaces".
  - multi-level
  - heirarchial
- **Inheritance modes:**

- Java does not support mode of inheritance like C++.
- Inheritance in java is similar to "public" mode of inheritance in C++.

- Syntax:

```
class Person {
	private String name, address;
	private int age;
	public Person() {
		this.name = "";
		this.address = "";
		this.age = 0;
	}
	public Person(String name, String address, int age) {
		this.name = name;
		this.address = address;
		this.age = age;
	}
	// getters/setters
	// input() / display()
}
class Employee extends Person {
	private int empno;
	private double salary;
	public Employee() {
		this.empno = 0;
		this.salary = 0.0;
	}
	public Employee(String name, String address, int age, int
	 empno, double salary) {
		super(name, address, age);
		this.empno = empno;
		this.salary = salary;
	}
	// getter/setter
	// input
	public void display() {
		super.display();
		System.out.printf("%d\n%d\n", empno, salary);
```

```
        }
    }
    class Main {
        public static void main(String[] args) {
            Person p1 = new Person("p1", "a1", 20);
            p1.display();

            Employee e2 = new Employee("e2", "a2", 30, 1001, 40000.00);
            e2.display();
        }
    }
```

**Day 03 (2 hours)**

=======

**+ Inheritance:**

- In java "sub-class" "extends" functionality of "super-class".
- All public and protected members of super-class are directly accessible from "sub-class" methods.
- However if name of member is super-class and sub-class is same, then super-class member can be accessed from sub-class method using "super" keyword.

```
- e.g. Employee class
        public void display() {
            super.display();
            System.out.printf("%d\n%f\n", empno, salary);
        }
```

- Whenever object of sub-class is created, first super-class constructor is executed and then sub-class constructor is executed.
- If not specified, from sub-class constructor paramless constructor of super-class is called.
- We can invoke parameterized constructor of super-class from sub-class using "super" keyword.

```
- e.g. Employee class:
        public Employee(String name, String address, int age, int empno,
double salary) {
            super(name, address, age);
            this.empno = empno;
            this.salary = salary;
        }
```

- Super-class constructor can be invoked from the first line of sub-class constructor. Otherwise compiler raise error.

+ Object Slicing:

- When sub-class object is assigned to a super-class reference, using that reference we can access only super-class methods. Accessing sub-class methods on such reference will raise compile time error.

- e.g.

```
Person p1 = new Employee();
p1.setName("Nilesh"); // Allowed
p1.setSalary(30000.00);     // Error
```

**+ Method Overriding:**

- However if a method is redefined in the sub-class with the same signature as that of super-class; it is called as "method overriding".

- Unlike C++, java doesn't need "virtual" keyword; as each method in java is by default treated like virtual.

- In such case, method is always invoked depending on type of object (not depending on type of reference). This is called as "dynamic method dispatch".

- e.g.

```
class Person     -> public void display() {...}
class Employee   -> public void display() {...}
Person p1 = new Employee();
p1.display();    // Employee's display() is called
```

- While overriding a method, its access specifier can be widen in sub-class. Reverse is not allowed (compile time error).

- In other words, if method is "public" in super-class it must be overridden in sub-class as "public". Attempt to make it protected or private in sub-class raise compile time error.

- While overriding method in sub-class, return-type of method can remain same or can be of sub-class type (of return type).

**+ Package:**

- Package is collection of sub-packages, classes and other types.

- Package is a physical container (i.e. directory) containing sub-packages, classes and other types.

- A type/class can be added into the package by giving package declaration on the first line of .java file.

```
package pkg1;
```

- Packages are used to avoid name-clashing (just like namespaces in C++).

- Class access modifiers:

1. default:
   - Classes are accessible into the same package.
2. public:
   - Classes can be accessed in the same package as well as outside the package.
   - To access a class outside the package, use import statment into that .java file.

- import java.util.Scanner;
- The one or more "import" statements can be written after "package" statement in the .java file (if present).
- One .java file can contain one or more java classes; however only one class can be "public" (in a .java file).
- Name of "public" class must match with .java file name. Otherwise compiler will raise error.

**+ Example of packages:**

- Person.java

```
==========
package p1;
public class Person {
        // ...
}
```

- Main.java

```
==========
import p1.Person;
//import p1.*;
public class Main {
        public static void main(String[] args) {
                Person p1 = new Person();
                // ...
        }
}
```

---

+ javac -d . Person.java
  - create "p1" package (dir) and Person.class file in that dir.
+ javac -d . Main.java
  - create Main.class in cur directory.
+ java Main
  - will execute program.

**- import p1.*;**

- This will allow to access all public classes in "p1" package into the .java file (in which above line is written).
- This will not allow classes under sub-packages of "p1".
- However it is recommended practice to import each class individually as required.
- A class without any package cannot be accessed in class with some package.

- **Nested packages:**

- It is possible to have package within another package. In fact, it is recommended practice to have hierarchial package names.

- Example:

Person.java

==========

```
package com.sunbeam.wimc.pkg1;
public class Person {
	// ...
}
```

```
Main.java
==========
package com.sunbeam.wimc.pkg2;
import com.sunbeam.wimc.pkg1.Person;
public class Main {
	// ...
}
```

---

CMD> javac -d . Person.java

CMD> javac -d . Main.java

CMD> java com.sunbeam.wimc.pkg2.Main

* Note last command, where "Main" class is kept in some package. While executing giving full name of class (pkgname + classname) is expected.

+ CLASSPATH:

- CLASSPATH is an environment variable containing list of directories separated by ; (win) or : (UNIX).

- Java tools (like java compiler, java app launcher, java debugger, etc) and JVM search java classes (.class) and packages into all directories mentioned in the CLASSPATH variable.

- To see current CLASSPATH variable:

[WIN] set CLASSPATH

[LINUX] echo $CLASSPATH

- To add a dir path in in CLASSPATH variable:

[WIN] set CLASSPATH=new_dir_path;%CLASSPATH%

[LINUX] export CLASSPATH=new_dir_path:$CLASSPATH

- To unset/nullify CLASSPATH variable (usually done before staring new assign):

[WIN] set CLASSPATH=

[LINUX] export CLASSPATH=

- If CLASSPATH is not defined, then by default java tools will search classes (.class) and packages in current directory.

**Day 04 (3 hours)**

**+ Members access modifiers:**

- public:
  - in class
  - in package
  - in sub-class
  - out-side class
- protected:
  - in class
  - in package
  - in sub-class
- default:
  - in class
  - in package
- private:
  - in class

**+ "instanceof" operator**

- boolean flag = ref instanceof ClassName;
- If object addressed by given reference is of given class or its sub-class, then instanceof operator results in "true"; otherwise it results "false".
- e.g. Person p1 = new Manager();

```
flag = p1 instanceof Manager; // true
flag = p1 instanceof Employee; // true
flag = p1 instanceof Person; // true
flag = p1 instanceof SalesManager; // false
```

```
flag = p1 instanceof Student; // false
```

**+ java.lang.Object class:**

- All java classes are directly or indirectly inherited from "Object" class.
- "Object" class is root of java class hierarchy.
- It provides following methods which can reused/overridden into the derived classes:

1. public String toString();
   - This method returns String representation of the object.
   - Programmer should override this method to return appropriate string reprsentation (as per his/her requirement).
   - If not overridden, Object class toString() will be invoked. This method returns a string containing "classname@hashcode".

```
- class Person {
        private String name, address;
        private int age;
        // ...
        public String toString() {
                return String.format("name:%s, addr:%s, age:%d", name, address, age);
        }
}
- class Main {
        public static void main(String[] args) {
                Person p1 = new Person("A","b",3);
                System.out.println( p1.toString() );
                System.out.printf("%s\n", p1);
        }
}
```

2. public boolean equals(Object obj);
   - equals() method compares current object to the given object and return true if they are equivalent. Otherwise it returns false.
   - Programmer should override this method to compare objects of class type.
   - If not overridden, Object class equals() will be called. It returns true if references are pointing to the same object; otherwise it returns false.

```
- class Employee {
        private int empid;
        private double salary;
        // ...
```

```
        public boolean equals(Object obj) {
                if(obj==null)
                        return false;
                if(this==obj)
                        return true;
                if(obj instanceof Employee) {
                        Employee other = (Employee)obj;
                        if(this.empid==other.empid)
                                return true;
                }
                return false;
        }
}
- class Main {
        public static void main(String[] args) {
                Employee e1 = new Employee(...);
                Employee e2 = new Employee(...);
                if(e1.equals(e2))
                        System.out.println("same");
                else
                        System.out.println("not same");
        }
}
```

- equals() method characteristics:
- reflexive: obj.equals(obj) must return true.
- symmetric: obj1.equals(obj2) return true, then obj2.equals(obj1) must return true.
- trasitive: obj1.equals(obj2) return true and obj2.equals(obj3) return true, then obj1.equals(obj3) must return true.
- consistent: obj1.equals(obj2) should produce same results until any one of the object is modified.
- Comparision with null, must return false.
- equals() method should not throw any exception (even if given object is of incompatible type).

3. protected void finalize();
4. public final int hashCode();
5. protected Object clone();
6. public void wait();
7. public void notify();
8. public void notifyAll();

Day 05 (5 hours)

==========

+ Assignment Discussion

```
  + Employee class
public double static calculateTotalSalary(Employee emps[]) {
      double total = 0.0;
      for(int i=0; i<emps.length; i++) {
            total += emps[i].getTotalSalary();
      }
      return total;
}
  + Main class
      Employee[] arr = new Employee[5];
      for(i=0; i<5; i++) {
            sysout("1.labor, 2. mgr, 3. vp, 4. salesmgr, 5. mktmgr : ");
            choice = sc.nextInt();
            switch(choice) {
                  case 1:
                        arr[i] = new Labor();
                        break;
                  // ...
            }
            arr[i].accept();
      }

      double expense = Employee.calcTotalSalary();

+ Enum in java:
      - enum in C/C++:
            - Enum is used to increase readability of the program.
            - Enum constants are internally integer values.
            - enum color {
                  red, green, blue
            };
            - enum color c = red;
            - switch(c) {
```

```
            case red:
                  break;
            case green:
                  break;
            case blue:
                  break;
      }
```

- In above example, enum constants red, green and blue are equivalent to ints 0, 1 and 2 respectively.

- In java, enum is added from version j2se 5.0.
- Like C/C++, enum is user defined type.
- Enum can be declared as:

```
enum Operation {
      Add, Subtract. Multiply, Divide
}
```

- By default all user defined enum types are inherited from java.lang.Enum class.
- All enum constants are internally public static final (constant) members of defined enum type.
- List of all enum objects can be accessed using static method in enum type:

```
public static Operation[] valucs();
```

- Each enum object can invoke two non-static methods.
  - public int ordinal();
    - return position of enum const in enum declaration
  - public String name();
    - return String representation/name of enum constant.
- Stringized name of enum constant can be converted into enum const object using static method in enum type:

```
public static Operation valueof(String str);
```

+ abstract class and method:

- If you donot want to create object of a class, that class can be declared as "abstract".
- public abstract class ClassName {

```
// ...
```

}

- If you donot want to implement certain method in the class, then method can be declared as "abstract".
- As abstract method doesn't have implementation, the class is incomplete. So we cannot create object of such class.
- Thus, if a class contains atleast one abstract method, class must be declared as "abstract".

- public abstract class Employee ... {
    // ...
    public abstract double getTotalSalary();
}
- If method is abstract in the class, sub-class is forced to override that method. If not overridden, method will remain abstract in sub-class and sub-class should also be declared as abstract.
- The abstract class can have fields, constructors and other methods too. All these members can be reused from the sub-class.

**+ final keyword:**

- final local variables:
  - Local variables (within method) can be declared as final.
  - The final variables can be initialized only once (may or may not be at the point of declaration).
  - The variable cannot be modified once initialized.
  - void fun1() {
      final int num = 100; // primitive type
      // ...
    }
  - void fun2() {
      final Person p1 = new Person();
      // p1.setName("New Name");  // allowed
      // p1 = new Person(); // error
    }
- In above example, p1 is constant reference and cannot be modified (i.e. cannot point to another object).
- final arguments:
  - void fun(final Person p1) {
      // ...
    }
  - p1 is constant reference and cannot be modified.
- final fields:
  - Non-static final fields can be initialized at the point of declaration in the class or within constructor.
  - Static final fields can be initialized at the point of declaration in the class or static block.
  - However final fields cannot be modified after initialization.
  - class Circle {
      public static final double PI = 3.14;
      // ...
    }

```
- class Employee {
        private final int empid;
        // ...
        public Employee(int empid, ...) {
                this.empid = empid;
                // ...
        }
}
```

**- final methods:**

- If method is declared as final, it cannot be overridden in sub-class. Otherwise compiler raise error.
- e.g. public final Class getClass();
  is a final method in java.lang.Object and hence it cannot be overridden.

**- final class:**

- The final class cannot be inherited. If tried so, compiler raise error.
- e.g. String class is final class.
- e.g. public final class VicePresident {

```
        // ...
}
```

- All methods in final class are treated as final methods.

**+ interface:**

- Interface contains only method declarations (no implementations, no fields, no constructors, etc).
- All methods in interface are public and abstract, which force all inherited classes to implement them.
- Thus the caller can invoke any of the interface method on the implementing class safely (as method will be definately available there).

- In java, interface is declared using "interface" keyword.
- All methods are public and abstract. Declaration of "public" and "abstract" keywords is optional.
- Interface fields (if present) are public static and final. (Declaring public static final is optional).
- While inheriting class from interface, "implements" keyword is used.
- A class can be inherited from one or more interfaces.

**- Syntax:**

- If A, B, C are classes and I1, I2, I3 are interfaces:

```
class A extends B                       // valid
class A extends B, C                    // invalid
class A implements I1                   // valid
class A implements I1,I2                // valid
class A implements I1 extends B         // invalid
```

```
class A extends B implements I1                    // valid
class A extends B implements I1,I2      // valid
interface I3 implements I1                         // invalid
interface I3 extends I1                            // valid
interface I3 extends I1, I2                        // valid
interface I3 extends C                             // ????
```

- Interfaces are used to give standard/specification.

```
interface Shape {
	public static final double PI = 3.14;
	// double PI = 3.14;
	public abstract double calcArea();
	double calcPeri();
}
- class Circle implements Shape {
	private double radius;
	// constructor
	public double calcArea() {
		return Shape.PI * radius * radius;
	}
	public double calcPeri() {
		return 2 * Shape.PI * radius;
	}
}
- class Main {
	public static void main(String[] args) {
		Shape ob = new Circle(7);
		System.out.printf("area : %f\n", ob.calcArea());
	}
}
```

Day 6

=========

```
+ Q. equals() method use??
	public static int search(Object[] arr, Object p) {
		for (int i = 0; i < arr.length; i++) {
			if(p.equals(arr[i]))
```

```
                return i;
            }
            return -1;
        }
        public static void main(String[] args) {
            Student[] arr = new Student[5];
            arr[0] = new Student( ... );
            arr[1] = new Student( ... );
            arr[2] = new Student( ... );
            arr[3] = new Student( ... );
            arr[4] = new Student( ... );
            System.out.println("Enter roll to search : ");
            int roll = sc.nextInt();
            Student s = new Student();
            s.setRoll(roll);
            int index = search(arr, s);
            if(index==-1)
                System.out.println("Record Not Found");
            else
                System.out.println("Record Found");
        }

+ swapping two objects:
        public static void swap(Distance d1, Distance d2) {
            int temp;
            temp = d1.feet;
            d1.feet = d2.feet;
            d2.feet = temp;
            temp = d1.inches;
            d1.inches = d2.inches;
            d2.inches = temp;
        }
        public static void main(String[] args) {
            Distance d1 = new Distance(11, 6);
            Distance d2 = new Distance(8, 10);
            System.out.printf("d1:%s, d2:%s\n", d1, d2);
            swap(d1, d2);
```

```
        System.out.printf("d1:%s, d2:%s\n", d1, d2);
    }
```

+ java.util.Comparable interface:

- This is standard way of comparing two java objects.
- To sort array of objects using

  Arrays.sort() method

  The object's class must be inherited from Comparable interface.
- This interface has single method:

  int compareTo(Object obj);

  - If "this" object is equal to "obj", return 0.
  - If "this" object > "obj", return +1, else return -1.
- class Student implements Comparable {

```
    // ...
    public int compareTo(Object obj) {
        Student other = (Student) obj;
        if(this.roll == other.roll)
            return 0;
        if(this.roll > other.roll)
            return +1;
        else
            return -1;
    }
}
```

- The sort() method can be called as follows:

```
Student[] arr = new Student[5];
arr[0] = new Student(...);
// ...
arr[4] = new Student(...);

Arrays.sort(arr);
```

- Arrays.sort() method sorts the array of objects. Whenever comparision of two objects is required during sorting process, it internally calls compareTo() method on the object.
- If class is not inherited from Comparable interface, Arrays.sort() throws ClassCastException.

+ user-defined sorting method:

```
public static void sort(Object[] arr) {
    for(int i=0; i<arr.length; i++) {
        for(int j=i+1; j<arr.length; j++) {
            Comparable arri = (Comparable)arr[i];
            if(arri.compareTo(arr[j]) > 0) {
                Object temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

**+ Marker interfaces:**

- In java, interfaces may not have any method declarations (i.e. empty interfaces). Such interfaces are called as "marker interfaces".

- They are used to indicate that the class is associated with certain functionality.

- e.g. Serializable is a marker interface. If class is inherited from "java.io.Seriablizable" interface, then JVM ca convert object details into sequence of bytes (using ObjectOutputStream class's write object method). If class is not inherited from Serializable, JVM will throw exception.

- e.g. Cloneable is a marker interface. If class is inherited from "Cloneable", then Object class's clone method can create copy of the object of that class. Otherwise it throws CloneNotSupportedException.

- example: class Date, class Person, ...

**+ Exception Handling:**

- Runtime problems are known as exceptions. Exception handling is flexible way of handling runtime problems in the execution of the program.
- When a function cannot handle a problem detected at runtime, it will "throw" it in form of object of class.
- The calling function should use "try-catch" block for handling such runtime problems.
- If no runtime problem occur, all statements in try block will be executed.
- If runtime problem occurs, all statements thereafter in try block are skipped and catch block is executed.

- In java, runtime problems can be thrown in form of objects of the classes inherited from "java.lang.Throwable" class.
- There two main types of Throwables:
- **Error**
  - Represents non-recoverable JVM-internal problems (usually).

- All error classes are inherited from java.lang.Error.
- e.g. java.lang.OutOfMemoryError, etc.

- **Exception**
  - Represents runtime problems that can be handled
  - There are two types of exceptions:
  - **unchecked exceptions**
    - Mostly occur due to programmer/end user's mistakes.
    - All such exception classes are inherited from java.lang.RuntimeException.
    - Compiler do not force to handle these exception.
    - e.g. NullPointerException, ArrayIndexOutOfBoundsException, etc.
  - **checked exceptions**
    - Represents runtime problems raise within JVM or outside the JVM.
    - Compiler force to handle these exceptions in one of the following ways:
    - Calling function should have try-catch block.
    - Calling function has throws clause specifying exception class.
    - All class inherited from java.lang.Exception (except RuntimeException and its sub-classes) represents checked exceptions.
    - e.g. IOException, SQLException, CloneNotSupportedException, etc.

- **Java Exception Hierarchy:**
  - Throwable
    - |- Error
      - |- OutOfMemoryException
      - |- etc.
    - |- Exception
      - |- IOException
      - |- SQLException
      - |- CloneNotSupportedException
      - |- etc.
      - |- RuntimeException
        - |- NullPointerException
        - |- ArrayIndexOutOfBoundsException
        - |- ClassCastException
        - |- etc.

- If exception is not handled by the calling function, then it will propogated to its calling function. However if exception is not handled in the entire chain of functions. finally it is given to main() function.

- If main() function doesn't handle the exception, it will be caught by the JVM; however program will be terminated.

- One try-block can have multiple catch blocks. However exception sub-class catch block should appear first and then exception super-class catch block. Otherwise compiler will raise error.

- e.g.

```
try {
        // ...
} catch(ArrayIndexOutOfBoundsException e) {
        // ...
} catch(RuntimeException e) {
        // ...
} catch(Exception e) {
        // ...
} catch(Throwable e) {          // generic catch
        // ...
}
```

- **In java, finally block can be written after catch block(s). It is optional block.**

```
try {
        // ...
} catch(Exception e) {
        // ...
} finally {
        // ...
}
```

- The finally block will always execute irrespective of exception occur or not.

- Even if control is returning/jumping from try or catch block (as a result of return, break or continue like statements), finally block is still executed.

- The only case in which finally block will not be executed in execution System.exit() within try or catch block.

- example:

```
class Main {
      public static int divide(int num, int den) {
            if(den==0)
                  throw new RuntimeException("Den cant be 0.");
            return num / den;
      }
      public static void main(String[] args) {
            int num, den, res;
```

```
                Scanner sc = new Scanner(System.in);
                try {
                        num = sc.nextInt();
                        den = sc.nextInt();
                        res = divide(num, den);
                        System.out.printf("Result : %d\n", res);
                } catch(RuntimeException e) {
                        System.out.println("Error occured!");
                } finally {
                        System.out.println("Always executed!");
                }
        }
}
```

- java.lang.Throwable:
    - public Throwable(String message);
        - Throwable class object encapsulates a string message, which can be initialized via parameterized constructor. Such constructor is present for all its derived exception types.
    - public String getMessage();
        - Returns a message encapsulated by exception object.
    - public void printStackTrace();
        - Displays info about exception - mainly exception class type and location in the code from where it is generated (along with stack trace).
    - public String toString();
        - returns exception details in form of string which contains exception type, message.
- User-defined exception class:
    - example:

```
class MyException extends RuntimeException {
        private String field;
        private int value;
        public MyException(String msg, String field, int value) {
                super(msg);
                this.field = field;
                this.value = value;
        }
        // getters
}
class Date {
```

```
        // ...
        public void setYear(int year) {
            if(year < 0)
                throw MyException("invalid year", "year", year);
            this.year = year;
        }
    }
```

- throws clause : Exception specification list
    - example:

```
        class Temp {
            public void fun() throws IOException, ServletException {
                // ...
            }
        }
```

- If a method is not handling generated checked excepion (using try-catch block), then it must write throws clause in method prototype to indicate exception types it may throw.
- if class B extends A:

```
A -> void fun() throws E1, E2, E3;
```

then

```
B -> void fun() throws E1, E2, E3; // OK
```

OR

```
B -> void fun() throws E1, E2;          // OK
```

BUT

```
B -> void fun() throws E1, E4;          // Error
```

In other words, an overridden sub-class method can throw same or fewer exceptions than corresponding super-class method; but it cannot throw any additional exception.

- if class D extends C:

```
C -> void fun() throws IOException;
```

then

```
D -> void fun() throws IOException;     // OK
```

OR

```
D -> void fun() throws EOFException;// OK
```

BUT

```
D -> void fun() throws Exception;       // Error
```

In other words, an overridden method can throw exception of the same type or exception sub-class typ corresponding to super-class method; however it cannot throw super-class of exception type.

Day 07

==========

- multi-catch block: (j2se 7)
  - example:

```
try {
    // ...
    fun();
    // ...
} catch(EOFException e) {
    // igonore : do nothing
} catch(IOException | SQLException e) {
    System.out.println("Error Occured!");
    System.Exit(1);
} catch(Exception e) {
    e.printStackTrace();
}
```

- When we want to implement same logic to handle more than one exception, we can use multi-catch block as shown in above example.

**+ Wrapper classes:**

- In java, primitive types are not classes i.e. primitive type variables are not objects.
- However sometimes, there is need to treat prmitive type values as java objects. E.g. java collections can work only with objects (not with primitive types).
- Example: till j2se 1.4

```
ArrayList obj = new ArrayList();
// It has add method to add object of any type
// void add(Object ob);
obj.add(11);      // error: cannot add primitive type
obj.add(new Integer(11)); // no error: Integer obj
```

- Thus "Integer" is wrapper class provided by java to treat "int" values as "objects".
- Java provides wrapper classes for all primitive types as follows:
  - byte -> Byte
  - short -> Short
  - int -> Integer
  - long -> Long
  - float -> Float
  - double -> Double

- char -> Character
- boolean -> Boolean

- Wrapper class objects encapsulate primitive type values. However additionally they provide lot of helper methods (static / non-static). Also they define lot of helpful constants.

+ Autoboxing:

- example;

```
int num1 = 100;
Integer num2 = num1;
```

- This is valid syntax from J2SE 5.0 and onwards.
- Here primitive type value is automatically converted to corresponding wrapper class object type. This feature is known as "autoboxing".
- example:

```
int num3 = num2;
```

- As shown in above example, even wrapper class object can be converted back to the corresponding primitive type value directly.
- However conversion from primitive type to wrapper object is not efficient (as internally memory is allocated, as well as it takes time). For better performace autoboxing should avoided whereever possible.
- example:

```
Integer i1=10, i2=20, i3;
i3 = i1 + i2;
```

- example:

```
ArrayList obj = new ArrayList();
obj.add(11);      // valid from j2se5.0 - but slow
```

**+ String class:**

- String encapsulate set of characters and length. It provides numerous method for manipulating String data.
- String is immutable object. Any method modifying string contents will not change the object's contents; rather it creates a new String object with modified contents.
- example:

```
public static void main(String[] args) {
        String s1 = "SunBeam", s2;
        System.out.printf("s1 : %s\n", s1);
        s2 = s1.concat("Infotech");
        System.out.printf("s1 : %s\n", s1);
        System.out.printf("s2 : %s\n", s2);
}
```

- **String Comparision:**

- Two object's equality can be checked using equals() method.
- They can also be compared using Comparable interface compareTo() method.
- Object references are compared using == operator. If this operator results true, both reference are pointing to same object.

- String class overrides equals() method and implements Comparable interface.
- String s1 = "sunbeam", s2="infotech";
    - Internally two String objects are created with different values.
    - s1==s2 => false
    - s1.equals(s2) => false
- String s1 = "sunbeam", s2=new String("sunbeam");
    - Use of new operator force to create a new object with similar contents so:
    - s1==s2 => false
    - s1.equals(s2) => true
- String s1 = "sunbeam", s2="sunbeam";
    - To minimize memory requirement, java compiler add a single entry into "string table" maintained in .class file. Obviously at runtime single string object is created and both references point to the same.
    - s1==s2 => true
    - s1.equals(s2) => true
- String s1 = "sunbeam", s2="sun"+"beam";
    - Java compiler itself add "sun" and "beam" to create a new string "sunbeam" and found that the entry is already present in string table. So again single object is created and both references point to it.
    - s1==s2 => true
    - s1.equals(s2) => true
- Java does not support operator overloading; however +, += operators are built-in overloaded for String class.
- String methods:
    - String(String s);
    - String(char []arr);
    - int length();
    - char charAt(int index);
    - boolean equals(Object str);
    - boolean equalsIgnoreCase(String str);
    - int compareTo(String str);
    - static String format(String fmt, ...);
    - int indexOf(char ch);
    - int indexOf(String substr);

- String substring(int start, int end);
- String[] split(String regex);
- String concat(String s);
- String replace(String find, String replace);
- String toUpperCase();
- String toLowerCase();

**+ StringBuffer Class:**

- Represents mutable string object i.e. any method modifing contents of object will modify contents of the same object.
- example:

```
public static void main(String[] args) {
    StringBuffer s1 = new StringBuffer("sunbeam");
    System.out.printf("s1 : %s\n", s1.toString());
    s1.append("infotech");
    System.out.printf("s1 : %s\n", s1.toString());
}
```

- StringBuffer class encapsulates a string buffer which can dynamically grow as number of characters in the object are increased (appended).
- The number of characters present can be found using length() method, while maximum number of characters that can be stored in internal buffer can be found using capacity().
- **Stringbuffer Methods:**
  - StringBuffer(String s);
  - int length();
  - int capacity();
  - char charAt(int index);
  - StringBuffer append(String s);
  - StringBuffer reverse();
  - etc.
- All methods in StringBuffer class are synchronized, hence it can be safely manipulated in multi-threaded program. However due to synchronization, efficiency reduces (slower).

**+ Stringbuilder Class:**

- Functionalitywise same as StringBuffer.
- Added into J2SE 5.0.
- Its methods are not synchronized. So used in single threaded program to improve the performance.

**+ Stringtokenizer Class:**

- Use to split string data into multiple parts. Nowdays split() method is preferred over StringTokenizer.

+ java.util.Calendar class:

- Represents a calendar date and time.
- To create the object:
  - Calendar cal = Calendar.getInstance();
  - By default object will contain current date and time values.
  - Calendar is abstract class. On desktop (J2SE), it internally creates object of java.util.GregorianCalendar class.
- To set calendar fields:
  - cal.set(year, month, day);
  - cal.set(year, month, day, hour, min, sec);
  - Imp note: months are counted from 0 to 11.
- To get value of individual field:
  - int dd = cal.get(Calendar.DAY_OF_MONTH);
  - int mm = cal.get(Calendar.MONTH) + 1;
  - int yy = cal.get(Calendar.YEAR);

**+ Java.Util.Date Class:**

- Was used to represent a calendar day & time. Nowdays most of the functions from this class is deprecated.
- Currently this class is used to represent time duration in milliseconds from epoch time i.e. 1-JAN-1900 00:00:00.
- example:

```
Date d2 = new Date();
System.out.println(d2.toString());
```

**+ Nested Classes:**

1. static member class:
   - A class can be declared within another class with "static" keyword. In this case, the inner class is treated as a static member of the outer class.
   - Inner class object is not dependent on outer class object.
   - Inner class can access static members of outer class directly; however it cannot access non-static members directly.
   - If name of static member is same in inner and outer class, then outer class member can be accessed in inner class using syntax "Outer.member".
   - static member class can have public, private, protected or default access modifier.
   - If static member is visible outside the Outer class (due to its access modifier), then it can be accessed using syntax "Outer.Inner" outside the class.

- Outer.Inner obj = new Outer.Inner();
- example:

```
class Outer {
        static class Inner {
                // ...
        }
        // ...
}
```

**2. non-static member class:**

- Non-static inner class object can be created, if and only if it is associated with some outer class object.
- Non-static member class can access static as well as non-static members of the outer class directly.
- Non-static member class have a hidden reference to the associated outer class object i.e. "Outer.this".
- If name of non-static member in outer class is same as that of inner class, then it can accessed in the inner class using syntax "Outer.this.member".
- Object of inner class can be created outside the Outer class, using one of the following way:
  - Outer obj = new Outer();
    Outer.Inner ref1 = obj.new Inner();
  - Outer.Inner ref2 = new Outer().new Inner();
- Inner class object can be created in the outer class non-static method directly:
  - class Outer's non-static method:

```
void nsMethod() {
        Inner obj = new Inner();
        //Inner obj = this.new Inner();
}
```

- example:

```
class Outer {
        class Inner {
                // ...
        }
        // ...
}
```

+ example:

```
class List implements Iterable {
      private static class Node {
            Object data;
            Node next;
            public Node() {
```

```
            this(null);
        }
        public Node(Object val) {
            this.data = val;
            this.next = null;
        }
    }

    private Node head;
    public List() {
        this.head = null;
    }
    public void add(Object val) {
        Node newnode = new Node(val);
        if(head==null)
            head = newnode;
        else {
            Node trav = head;
            while(trav.next!=null)
                trav = trav.next;
            trav.next = newnode;
        }
    }

    private class Iter implements Iterator {
        private Node nodeObj;
        public Iter() {
            this.nodeObj = head;
        }
        public boolean hasNext() {
            return nodeObj!=null ? true : false;
        }
        public Object next() {
            Object val = nodeObj.data;
            nodeObj = nodeObj.next;
            return val;
        }
```

```
            public void remove() {
            }
        }

        public Iterator iterator() {
            return this.new Iter();
        }
    }
    public class Main {
        public static void main(String[] args) {
            List obj = new List();
            obj.add("A");
            // ...
            List.Iter itr = obj.iterator();
            while(itr.hasNext()) {
                String val=(String)itr.next();
                System.out.println(val);
            }
            for(Object val : obj)
                System.out.println(val);
        }
    }
```

Day 8

==========

3. local class:

- In java, classes can be defined within the method. Such classes are treated like local variables and are called as "local classes".
- Their scope is limited to the method in which they are implemented. In other words, the class cannot be accessed outside the enclosing method.
- If class is implemented within static method, it behaves as static member class (i.e. can access static members of outer class directly).
- If class is implemented within non-static method, it behaves as non-static member class (i.e. can access static and non-static members of outer class directly).
- In both cases, local classes can access final local variables of the enclosing method.
- example:

```
class Main {
```

```
        public static void main(String[] args) {
            Student[] arr = new Student[5];
            arr[0] = new Student(...);
            // ...

            class StudNameComparator implements Comparator {
                public int compare(Object o1, Object o2){
                    Student s1=(Student)o1;
                    Student s2=(Student)o2;
                    int diff = s1.getName().compare(s2.getName());
                    return diff;
                }
            }
            StudNameComparator cmp = new StudNameComparator();
            Arrays.sort(arr, cmp);

            for(Student s:arr)
                System.out.println(s);
        }
    }
```

- We can create any number of objects of local class within enclosing method.

**4. Anonymous Inner Class:**

- example:

```
ClassName obj = new ClassName() {
    // ...
};
```

- When you want to create single object of a class inherited from some other class/interface and need to invoke its overridden methods, anonymous class would be the best choice.
- In above syntax example, internally a new class is created inherited from "ClassName" and then its object is created, which is accessed by "obj" reference.
- Obviously with "obj" reference we can access only overridden methods of that inherited anonymous class.
- Usually such classes are written in some method and hence their behaviour is same as "local classes".
- example:

```
Arrays.sort(arr, new Comparator() {
    public int compare(Object o1, Object o2) {
        Student s1 = (Student)o1;
        Student s2 = (Student)o2;
```

```
                int diff = s1.getName().compareTo(s2.getName());
                return diff;
            }
        });
```

**+ Generics in Java:**

- When same logic need to be implemented for different data types, generic (template) programming can be used in java.
- Example: All data structures (e.g. Stack, Queue, List, Tree, etc) logic remains same irrespective of which type of objects are used in that data structure.
- **Limitations of generics in Java:**
  1. Cannot be used with primitive types.
  2. Cannot create array of generic type (T).
  3. Type of generic type cannot be properly detected at runtime using reflection. Also cannot work well with "instanceof" operator.
- Efficiency of generic types is same as efficiency of corresponding "Object" based types in Java.
- **Advantages:**
  1. The same generic class can be reused for different class types.
  2. Generic classes are type-safe. If different type of elements is added in the generic collection, compile time error is raised.
- Generics were added in Java 5.0 version. Till Java 1.4 all collection classes were used as "Object" collections. However 5.0 onwards, all collection classes refined to take generic parameters.

  Ex1.4: ArrayList list = new ArrayList();

  Ex5.0: ArrayList<Person> list = new ArrayList<Person>();

  Ex7 : ArrayList<Person> list = new ArrayList<>();

- example:

```
class Stack<T> {
    private T[] arr;
    private int top;
    public Stack(Class type, int size) {
        arr = (T[])Array.newInstance(type, size);
        top = -1;
    }
    public void push(T ele) {
        arr[++top] = ele;
    }
    public void pop() {
```

```
                top--;
        }
        public T peek() {
                return arr[top];
        }
        public boolean empty() {
                return top==-1;
        }
        public boolean full() {
                return top==arr.length-1;
        }
}
public class Main {
        public static void main(String[] args) {
                Stack<Integer> s1 = new Stack<>();
                s1.push(11);
                s1.push(22);

                int ele;
                ele = s1.peek();
                s1.pop();
                System.out.printf("popped : %d\n", ele);
        }
}
```

- Generic Interfaces:
  - predefined java.util.Comparable<T> interface:

```
public interface Comparable<T> {
        int compareTo(T o);
}
```

  - implementing Comparable<T> in Student class:

```
class Student implements Comparable<Student> {
        // ...
        public int compareTo(Student other) {
                if(this.roll==other.roll)
                        return 0;
                return this.roll > other.roll ? +1 : -1;
        }
```

```
}
```

- Generic Methods:

```
- By default all methods in generic class are generic methods.
- However it is possible to implement generic method in non-generic
  class.
- example:
class Util {
    public static void swap<T>(T o1, T o2) {
        T t = o1;
        o1 = o2;
        o2 = t;
        System.out.printf("o1:%s, o2:%s\n", o1, o2);
    }
}
class Main {
    public static void main(String[] args) {
        String s1="sunbeam", s2="infotech";
        Util.swap(s1,s2);
        // ...
    }
}
```

+ Java Collection Framework:

- Collection Framework = Interfaces + Implementations + Algorithms

+ Java Collection Hierarchy:

- All collection interfaces and classes are kept under java.util package.

+ Collection<E>

|- Set<E>

|- HashSet<E>

|- LinkedHashSet<E>

|- TreeSet<E>

|- List<E>

|- ArrayList<E>

|- LinkedList<E>

|- Vector<E> *

|- Stack<E>

|- Queue<E>

|- PriorityQueue<E>

+ Map<K,V>

|- HashMap<K,V>

|- LinkedHashMap<K,V>

|- TreeMap<K,V>

|- Hashtable<K,V> *

* Before J2SE 1.2 only Vector and Hashtable were supported. When collection framework is introduces with JAVA2, these classes are modified to be compatible with collection framework interfaces.

**+ Java Collection Interfaces:**

1. Collection<E>
   - Root of collection hierarchy.
   - Most of the classes inherited from Collection<> interface.
2. Set<E>
   - Is a collection which does not allow duplicate values.
3. List<E>
   - Is a collection which allows duplicate values, bidirectional traversing and random access.
4. Queue<E>
   - Is a collection which provide data structure queue functionality.
5. Map<K,V>

- Not inherited from Collection interface.
- Stores data in key, value pair so that for a given key, value can be searched in fastest possible time.

+ interface Collection<E>:

- extends Iterable<E>
  - Iterator<E> iterator();
  - All classes inherited from Collection interface, internally implement Iterator.
  - They all can be traversed using for-each loop.

- boolean add(E ele);
- boolean contains(E ele);
- boolean remove(E ele);
- boolean isEmpty();
- int size();

- boolean addAll(Collection<E> c);

- boolean containsAll(Collection<E> c);
- boolean removeAll(Collection<E> c);
- boolean retainAll(Collection<E> c);
- void clear();

**+ interface Set<E>**

- HashSet<E>
  - Elements are stored in any order (based on hashcode).
  - Faster/Efficient search.
  - Equality is based on hashCode() and equals() method.
- LinkedHashSet<E>
  - Elements are stored in the order of insertion.
  - Equality is based on hashCode() and equals() method.
- TreeSet<E>
  - Elements are stored in the sorted order.
  - Equality and comparision is based on compareTo() method of Comparable interface.
  - In other words, objects to be added into TreeSet<> should implement Comparable interface; otherwise ClassCastException will be thrown.
  - Note that TreeSet<> does not depend on equals() and hashCode() method of the containing objects.
  - As a standard practice, if equals() and compareTo() both are implemented, then both should perform comparision on the same fields.

**Day 9**

==========

+ interface List<E>

- Since it is inherited from Collection<E> interface, it has all methods discussed in Collection<E>.

- List<> provides additional searching mechanisms:
  - int indexOf(Object o);
    - Compare given object with all objects in the list using equals() method and if found, return "index" of that element. If not found return -1.
    - As list can have duplicate elements, this method returns first occurance of the element
      - example:

```
roll = sc.nextInt();
s = new Student();
s.setRoll(roll);
```

```
index = list.indexOf(s);
if(index!=-1)
        System.out.println("Found");
else
        System.out.println("Not Found");
```

- int lastIndexOf(Object o);
- Compare given object with all objects in the list in the reverse order using equals() method. If found return index of object, else return -1.
- As list can have duplicate elements, this method returns last occurance of the element.

**+ Random access:**

- Element at any index can be accessed/replaced directly.
- E get(int index);
  - Returns object at given index. Throws exception for invalid index.
  - example:

```
index = sc.nextInt();
s = list.get(index);
System.out.println(s);
```

- E set(int index, E newEle);
  - Set a new object at given index (replacing old object).
  - example:

```
index = sc.nextInt();
s = new Student();
s.accept();
list.set(index, s);
```

- E remove(int index);
- deletes element at given index and all elements after that index will be shifted to index-1.

**+ Bi-directional Traversing:**

- List<> internally implements iterator for traversing. This iterator is inherited from ListIterator<> interface.
  - ListIterator<> interface is inherited from Iterator<>. It has additional methods:
    - boolean hasPrevious();
    - E previous();
    - void set(E ele);
  - Object of ListIterator<> can be received using listIterator() method of List<> interface.
    - ListIterator<> listIterator();
    - ListIterator<> listIterator(int index);

- **Forward traversing:**

```
ListIterator itr = list.listIterator();
while(itr.hasNext()) {
        Student s = itr.next();
        System.out.println(s);
}
```

- **Reverse traversing:**

```
ListIterator itr = list.listIterator(list.size());
while(itr.hasPrevious()) {
        Student s = itr.previous();
        System.out.println(s);
}
```

- By default, iterator points to start of the list, so reverse traversal is not possible.
- So in case of reverse traversing, list iterator should begin from end of list (as shown in above example).

+ **Algorithms which can work on List<> iterface:**
  - All methods given here are static methods of java.util.Collections class.
  - Sorting List<>:
    - void sort(List<T> list);
      - The type T must be inherited from Comparable<>.
    - void sort(List<T> list, Comparator<T> c);
  - Reversing List:
    - void reverse(List<T> list);
  - Randomizing List:
    - void shuffle(List<T> list);
  - Binary Search:
    - List should be initially sorted.
    - The type T must be inherited from Comparable<>.
    - void binarySearch(List<T> list, T obj);

+ List<> Implementations:
- **ArrayList<> class:**
  - Internally implemented as doubly linked list of arrays. Each node contains multiple elements.
    - It gives more efficient random access.
    - Insertion and deletion (at middle of list) is slower operation).
- **LinkedList<> class:**
- Internally implemented as doubly linked list with head and tail pointer.

- Insertion and deletion at ends of list are very fast. Also insertion and deletion at mid of list if efficient.
- However random access is slower.
- **Vector<> class:**
    - Vector<> is implemented as dynamic array (contiguous memory). The array grows or shrink runtime as per requirement.
    - However Vector<> is synchronized class (all methods are synchronized).
    - It is mainly preferred in multi-threaded program. However sync reduce the performance in single threaded program.
        - **Stack<> class:**
            - Inherited from Vector<>.
            - Provides last-in-first-out behaviour using methods: push(), pop(), peek(), ...

**+ Queue<> interface:**

- Implementation class:
    - PriorityQueue<>.

**+ Map<K,V> interface:**

- Map does not allow duplicate keys. If try to add duplicate key, new value will replace old value.
- methods:
    - V put(K key, V val);
    - V get(K key);
    - int size();
    - void clear();
    - void remove(K key);
    - boolean containsKey(K key);
    - boolean containsValue(V value);
    - Set<K> keySet();
    - Collection<V> values();
    - Set<Map.Entry> entrySet();
- **Implementation classes:**
    - HashMap<K,V>:
        - Elements are stored in any order of the keys.
        - Faster searching
    - LinkedHashMap<K,V>:
        - Elements are stored in the order of insertion of keys.
    - TreeMap<K,V>:
        - Elements are stored in the sorted order of keys.
        - The Key class must implement java.util.Comparable<K>.

- **Hashtable<K,V>:**
  - Elemenst are stored in any order of keys.
  - However Hashtable<> is synchronized class (all methods are synchronized).
  - It is mainly preferred in multi-threaded program. However sync reduce the performance in single threaded program.

* Note:

Map<Integer,Student> map = ....;

KEY : Integer

VALUE : Student

- KEY class i.e. Integer class must implement Comparable<> interface for TreeMap<> and must implement equals() & hashCode() for HashMap<>, LinkedHashMap<> and HashTable<>.

* Note:

Map<Student,ProgressCard> map = ....;

KEY : Student

VALUE : ProgressCard

- In this case Student class must implement Comparable<> interface for TreeMap<> and must implement equals() & hashCode() for HashMap<>, LinkedHashMap<> and HashTable<>.

**+ Time Complexity:**

- It is approximate measure of time required to complete any algorithm.
- Many times time complexity is calculated from number of iterations done into the program.
- For example:

  n! = 1 * 2 * 3 * ... * n

  - The program will be having loop repeated n times. So time required is proportional to "n". Thus time complexity is of order of "n" and denoted as O(n).
  - Here "O(n)" is known as Big O notation.
- Few Imp Time complexities:
  - Time required to add or delete element from stack or queue, is constant (independent of number of elements). So time complexity is O(1).
  - Time required to add element at the end of linked list having only head pointer, depends on number of elements (loop is repeated to traverse to last element). So time complexity is O(n).
  - Time complexity of binary search:

    2^i = n

    [where, i is num of iterations and n is number of elements in the sorted array]

    Time complexity O(log n).
  - Time complexity of selection sort:

Number of comparisions = (n-1) + (n-2) + ... + 1

= n (n-1) / 2

So time complexity is O(n^2)

**+ Hashing:**

- Hashing is used for fast searching.
- The data is stored in form of key-value pair, so that for a given key value can be searched in fastest possible time.
- Ideal time complexity of searching in hash table is O(1).

- Hash table is internally an array, holding data and data is accessed by proper index.
- The values are stored into the array, at a particular index. This index is calculated from the given key using some mathematical function, called as "hash function".
- Simplest example:
  - Storing student info so that it can searched for given roll number.
  - Here array of "n" student objects can be created and information of student with roll number "r" can be stored at index "r-1".
  - This "r-1" is example of simple hash function.
  - h(key) = key - 1 => results in index of hashtable.
  - Here student with roll "r" can be immediately found at index "r-1". Time complexity of searching O(1).

- However in the complex examples, multiple keys can result to same index/slot into the hashtable/array. This is called as "collision".
- The collision can handled in one of the following ways:

**1. Open Addressing:**

- If slot calculated using hash function is already occupied by some object, then another mathematical function is used to find alternative slot for the object. This is called as "re-hashing".
- Simplest re-hashing function would be "index+1" i.e. next slot of table (if empty). This simplest mechanism of open addressing is known as "linear probing".
- Load factor = num of elements / num of slots.
- This mechanism can be used only if Load factor <= 1.0.

**2. Chaining:**

- Each slot of hash-table holds a list of objects competing for the same slot. This list is known "bucket".
- As number of collisions are increased, the time complexity of searching will be deviated (increased) from O(1).

**+ hashCode() method:**

- java.util.Object class has hashCode() method, which can be overridden into the derived to return hash code/value of the object.

- This hash code determines the slot of the hashtable, in which object is to be stored.

- If not overridden, Object class hashCode() is called, which return hash value based on memory address of the object.

- **Rules of hashCode() method:**

1. If two objects are equal (by equals() method), then their hash code must be same.
2. If two objects are not equal (by equals() method), then their hash code may be same.

- Hence hash code must be calculated on the same fields, on which equality is compared.

- Statistically it is proven that hash code calculated using prime numbers is uniformly distributed (more unique) and hence most of the hash code calculations are done using prime numbers.

**+ Reflection:**

- Reflection technique is used to get information about the data type at runtime.

- Three applications/uses of reflection:

1. To get complete info about the type at runtime. It is used to develop intellisence feature, tools like javap.exe.
2. Use the retrieved information about the type to create object of class at runtime and invoke its methods. This is used in most of frameworks like struts, servlets, applets, etc internally.
3. Generate the java classes (byte code) at runtime and execute them. This can be done by using ClassLoader.defineClass() method.

- Information about a type can be received in java.lang.Class object using one of the following way:

1. Class c = obj.getClass();
   - Get the info of the class whose object is referred by "obj".
2. Class c = ClassName.class;
   - Get the info of the class with name "ClassName".
3. Class c = Class.forName("pkg.ClassName");
   - Get the info about the class whose name will be taken at runtime (in form of String).
   - This method loads the class in the JVM and return its information.

- java.lang.Class methods:
  - String getName();
  - boolean isAbstract();
  - boolean isInterface();
  - boolean isEnum();
  - Field[] getFields();
  - Method[] getMethods();
  - Constructor[] getConstructors();
  - Annotation[] getAnnotations();

+ GUI programming in JAVA:

- JAVA provides different classes for creating desktop GUI. There are two main ways to create GUI on JAVA:

1. AWT - Abstract Window Toolkit:

- Java UI classes internally depends on the Peer classes, which in turn use OS functions to create UI.
- e.g. java.awt.Button class internally use "ButtonPeer", which in turn calls OS function ("CreateWindow" on Windows OS) to create button UI.
- Due to this AWT UI will be have different look & feel on different platforms (OS).
- Also internal use of peer classes make them heavy components.

2. SWING:

- Java has another set of classes to create platform independent UI. These classes are called as "SWING".
  - The look & feel of UI remains same on all platforms.
  - Swing components internally use MVC architecture i.e. each swing component (ui control class) internally use three objects to split the functionality:
  - Model : keep the data of the component e.g. Strings in the JComboBox.
  - View : keep appearance related info of the component e.g. height, width, back color, fore color, etc.
  - Controller : responsible for calling event handler when event occurs e.g. user click, selection changes in combo box, etc. It also communicate between view and model.

+ SWING:

+ Swing component heirarchy:

- Component : represents a UI control on the screen.
- Container : is collection of one or more components.
- Window:
  - Container that can hold multiple components.
  - It has its own window i.e. title bar and border.
- Panel:
  - Container that can hold multiple components.
  - It does not have its own window. It is always present in some other window to group components.
- JFrame:
  - Frame has its own window.
  - So it is stand-alone desktop based application.
  - Frame application has main() function.
- JApplet:
  - Applet does not have its own window.
  - So it always execute within browser window.

- Applet does not have main() function. Its execution begin with init() method.
- Swing components:
  - JLabel, JTextField, JTextArea, JButton, JComboBox, JListBox, JCheckBox, etc.

**+ Event handling:**

- User action is called as an event. e.g. key press, mouse click, etc.
- We can handle these events to add functionality into the GUI.
- The standard way to handle events is to implement appropriate "Listener" interface.
- Example: Button click handler:
  * step1: Implement ActionListener interface into some class (in one of the following ways)
    - Write a new class inherited from ActionListener interface.
    - Implement ActionListener into the user-defined frame class itself.
    - Anonymous inner class to implement ActionListener interface.

* step2: Pass the object of class inherited from ActionListener to the addActionListener() method of button object, when that button is created.

* step3: Into this event handling class write appropriate logic within actionPerfromed() method.

- When a button click event occurs, at first it is detected by OS and inform to the JVM. Then JVM pass the event to the button object. Finally button object invokes actionPerformed() method of the listener inherited class object(s) registered with it.

**- Event objects:**

- Each listener method receive an Event object inherited from "java.awt.Event" class.
- e.g. ActionEvent, WindowEvent, etc.
- This event object contains complete info about the event, which includes source of event (component for which event occured) and additional info.

**Day 10**

==========

**+ Adapter classes:**

- Certain listener interfaces have more than one methods. e.g. WindowListener has 7 methods, MouseListener has 6 methods.
- Most of the cases programmer is interested in handling one or two methods from such interface. However being an interface, it is compulsory for the user to implement all the methods.
- To simplify this operation, java provides adapter classes for such interfaces. They are abstract classes having empty/default implementations of all methods in such interface. e.g. WindowAdapter and MouseAdapter.
- So programmer can write his own class inherited from adapter class and override only those methods in which he is interested.

- Typically preferred approach is to create anonymous class inherited from adapter and add its object with target component.

- e.g.

```
// In constructor of the user-defined frame class
this.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
                dispose();
        }
});
```

**+ Layout Managers:**

- Java has few builtin managers which are used to place components in the container in proper way.

- The layout manager can be set using Container's setLayout() method. If layout manager is not given (i.e. set to null), then programmer must give appropriate x,y coordinates and width,height using setBounds() method of the component.

```
this.setLayout(null);
```

- There are few imp layout managers as follows:

**1. FlowLayout:**

- Add components in horizontal direction one after another, just like web page.
- this.setLayout(new FlowLayout());
- button = new Button("Caption");
  this.add(button);
- This is default layout of Panel and Applet class.

**2. GridLayout:**

- Add components in the grid defined by the programmer. Components will be places in appropriate cell (row & col) one after another.

- this.setLayout(new GridLayout(2,3));
- button = new Button("Caption");
  this.add(button);

**3. BorderLayout:**

- Add components in certain direction or center of the container as shown in diag.
- this.setLayout(new BorderLayout());
  button = new Button("Caption");
  this.add(button, BorderLayout.CENTER);
- This is default layout of Window and Frame class.

**4. GridBagLayout:**

- Complicated layout manager which provides flexible way of adding components in the container.

**+ Applets:**

+ Applet steps (on command line):

1. Write a java class inherited from java.awt.Applet class and override init() method. Write appropriate code in it.
2. To test applet with appletviewer tool, write comment before class (after import statements).

```
/*
<applet class="pkg.ClassName" width="200" height="200">
</applet>
*/3. Compile applet class.
javac -d . ClassName.java
```

4. Run applet into applet viewer:

```
appletviewer ClassName.java
```

5. To test applet in html file, write HTML file as follows:

```
<HTML>
    <HEAD>
        <TITLE>APPLET DEMO</TITLE>
    </HEAD>
    <BODY>
        <h2> Applet Sample </h2>
        <hr/>
        <applet code="pkg.ClassName.class" width="200" height="200">
        </applet>
    </BODY>
</HTML>
```

6. Open HTML file in the browser.

**+ Applet:**

- Applet is a java class which is loaded into the client browser and executed within client side JRE plugin of the browser.
- Applets are embedded into html web pages using <applet> tag having following attributes:
  - code="pkg.ClassName.class" i.e. name of the applet class with package.
  - codebase="dir path" i.e. path of directory in which applet package is kept. Required if applet package is in diffrent directory than the html page.
  - width="300" i.e. width of applet
  - height="300" i.e. height of applet

**+ Applet Life Cycle:**

- Applet class has five life cycle methods, which are overridden by the programmer as per requirement and called by the JRE/JVM within the client side browser.

**1. public void init();**

- When client requests a web page having embedded applet in it, the .class file is also loaded at client side. Client broser JVM get the info about applet class from <applet> tag (code attribute). It will load applet class into JVM, create its object, default constructor is executed and then init() method is invoked at runtime.
- This method is called only once in the life cycle of applet class object.
- Programmer should override this method to perform one time initialization like creating components on applet, etc.

**2. public void destroy();**

- When applet object is no longer required, browser JVM will garbage collect it.
- However before doing this, it will call destroy() method of the browser.
- Programmer should override this method, if applet is holding some resource like file or network connection and need to be released before object is destroyed.
- This method is called only once in the life cycle of applet object.

**3. public void start();**

- Whenever applet is made visible to end user, this method is invoked.
- Programmer should override this method to perform any operation that to be done when applet is shown to user e.g. initiating animation, if any.
- This method can be get called multiple times in the life-cycle of applet object.

**4. public void stop();**

- When user navigates from the applet web page, (the applet is no more shown to user) stop() method is called.
- Programmer should override this method to stop operations initiated into start() method, if any. i.e. stop animations.

**5. public void paint();**

- When whole applet or part of applet window need to be repainted, JVM invokes paint() method.
- Programmer should override this method to draw customized painting logic, if required.

**+ Java IO:**

**+ File System Operations:**

- File or directory path is represented by java.io.File object.
- This "File" class has several methods to deal with file system.
- boolean canRead();
- boolean canWrite();

- boolean canExecute(),
- boolean setReadable(boolean perm);
- boolean setWritable(boolean perm);
- boolean setExecutable(boolean perm);
- boolean isHidden();

- boolean exists();
- boolean isFile();
- boolean isDirectory();

- boolean createNewFile();
- boolean mkdir();

- boolean delete();

- String[] list();
- long length();

**+ Input and Output:**

- Stream is java object in which data can be written or read from.
- There are two main types of streams:
  - Byte streams : Data is read/written byte by byte.
  - Character streams : Data is read/written char by char (unicode).

**+ Byte Streams**

- There are two types of Byte Streams:

1. OutputStream:
   - The data can be written into the output stream.
   - It is represented by abstract class: "java.io.OutputStream".
   - This class has following important methods:
     - abstract void write(int b);
       - Write single byte of data to the destination
     - int write(byte[] arr, int offset, int length);
       - Write part of byte array to the dest.
     - int write(byte[] arr);
       - Write whole array data to the dest.
     - void flush();
       - Write all data to the dest (if kept in mem).

- void close();
  - Close the stream and release resources.

2. **InputStream:**
   - The data can be read from input stream.
   - It is represented by abstract class: "java.io.InputStream".
   - This class has following important methods:
     - abstract int read();
       - Read single byte of data from destination
     - int read(byte[] arr, int offset, int length);
       - Read part of byte array from the dest.
     - int read(byte[] arr);
       - Read whole array data from the dest.
     - int available();
       - Returns num of bytes to be read.
     - void close();
       - Close the stream and release resources.

**+ OutputStream heirarchy:**

1. **FileOutputStream:**
   - Write the data to the file on the storage device (with help of OS).
2. **ObjectOutputStream:**
   - Converts java object and its info into the sequence of bytes (serialization).
3. **FilterOutputStream:**
   - Process data before writing it on underlying stream.
4. **DataOutputStream:**
   - Converts java primitive types into the object.
5. **BufferedOutputStream:**
   - Stores data in buffer (temp memory) before writing to underlying stream. Reduces number of write operations and thus improves performance.
6. **PrintStream:**
   - Convert java object data into proper (aligned) format. Neatly readable character data is produced
7. **ByteArrayOutputStream:**
   - Store data in form java byte array.

**+ Demo of DataOutputStream / DataInputStream**

Day 11:

+ Serialization:

- Converting java object into sequence of bytes is called as "serialization". These sequence of bytes contains data (contents) as well as metadata (info) of the object.

- Converting seqence of bytes back to the java object is called as "deserialization". It use metadata to recreate the object and restore its contents (using reflection).

- For serialization, the class must be inherited from marker interface "Serializable"; otherwise "NotSerializableException" will be thrown.

- Also class must have a public parameterless constructor.

- If you do not wish to serialize certain data member of the java object, it must be marked as "transient" (java keyword).

```
class Employee implements Serializable {
        private int empid;
        private String name;
        private double basicSalary;
        private transient double totalSalary;
        public Employee() {
                // ...
        }
        // ...
}
```

- Serialization & Deserilization example:

```
public static void writeStudent(Student s) throws Exception {
        FileOutputStream fout = new FileOutputStream("filepath");
        ObjectOutputStream out = new ObjectOutputStream(fout);
        out.writeObject(s);
        out.flush();
        out.close();
        fout.close();
}
public static void readStudent() throws Exception {
        FileInputStream fin = new FileInputStream("filepath");
        ObjectInputStream in = new ObjectInputStream(fin);
        Student s = (Student)in.readObject();
        System.out.println(s.toString());
```

```
            in.close();
            fin.close();
        }
```

+ Character Streams:

- Java stores characters as unicode (two bytes each).
- However char data can saved on disk using diffent encoding schemes e.g. ANSI, UTF-8, Unicode, Big Unicode, etc.
- The character streams are used for interconversion between java characters and encoding schemes.
- There are two main abstract classes:
  - java.io.Reader class
  - java.io.Writer class
- example:

```
FileReader reader = new FileReader(path);
System.out.println("ENCODING : " + reader.getEncoding());
BufferedReader bufReader = new BufferedReader(reader);
while (true) {
    String line = bufReader.readLine();
    if (line == null)
        break;
    System.out.print(line);
}
bufReader.close();
reader.close();
```

**+ Threading:**

- Thread Creation in Java:

1. extends Thread:
   - Programmer should write a class inherited from Thread class and override its "run()" method.
   - Object of that class should be created and its start() method is invoked.
   - example:

```
class MyThread extends Thread {
    // ...
    public void run() {
        // ...
    }
}
```

```
class Main {
        public static void main(String[] args) {
                MyThread t1 = new MyThread();
                t1.start();
        }
}
```

2. implements Runnable:

- Programmer should write a class inherited from "Runnable" interface and implement its "run()" method.
- Object of "Thread" class should be created and object of above created class should be passed to its constructor. Then call "start()" method of thread object.
- example:

```
class MyRunnable implements Runnable {
        // ...
        public void run() {
                // ...
        }
}
class Main {
        public static void main(String[] args) {
                MyRunnable r1 = new MyRunnable();
                Thread t1 = new Thread(r1);
                t1.start();
        }
}
```

- **Thread Life Cycle:**

1. New:
   - A new thread object is created.
   - Thread t1 = new Thread(...);
2. Ready to Run:
   - When start() method is invoked on thread object, the thread is associated with the scheduler.
   - Scheduler can select thread for the execution at any time based on scheduling algorithm.
3. Running:
   - When thread is selected for the execution by the scheduler, JVM invokes appropriate run() method and its execution begins.
4. Non-Runnable:

- Running thread may do IO request, sleep() or wait(). Because of that thread will go to non-runnable state. It is also called as "waiting" or "blocked" state.
- After IO completion, sleep() time completion or notify() done, thread may return back to "ready state" so that scheduler can schedule it again for further execution.
- However thread doing sleep() or wait() can be forcefully unblocked by calling interrupt() on the thread object (from another thread). This will raise InterruptedException into the sleeping/waiting thread.

5. Finally if run() method of a thread is completed, thread will be "dead" or "terminated".

**+ Thread Join:**

- When a thread calls join() method on some other thread object, it (calling thread) is paused until the given thread is terminated.
- If main thread calls t1.join(), then main thread will be paused until t1 is terminated.

+ Some important methods of thread class:

- void start();
- void setDaemon(boolean daemon);
- void join();
- void interrupt();
- static void sleep(long milli);
- void setName(String name);
- String getName();
- void setPriority(int pri);
- int getPriority();
- State getState();
- static Thread currentThread();
  - Get the "Thread" object in its run() method.
- static void yield();
  - Thread calling "Thread.yield()" method will goto ready state (from running) and thus JVM/scheduler gives CPU time for another thread.

**+ Daemon threads:**

- JAVA has mainly two types of threads:
  - non-daemon thread
  - daemon thread
- By default all threads are non-daemon threads.
- To make a thread as daemon, call setDaemon(true) on the thread object before starting that thread.

- When all non-daemon threads are terminated, process (in which JVM is running) is terminated. This will cause all daemon threads to terminate automatically.
- Thus daemon threads keep executing if at least one non-daemon thread is running.
- Daemon threads are also known as service threads, which provides service to non-daemon threads.
- The main thread is a non-daemon thread.

**+ Thread synchronization:**

- When multiple threads try to access same object at the same time, it is called as "race condition".
- This may result in corrupting object state and hence will get unexpected results.
- To avoid this Java provides sync primitive known as "monitor".
- Each java object is associated with a monitor object. The monitor can be locked or unlocked by the thread.
- If monitor is already locked, another thread trying lock it will be blocked until monitor is unlocked.
- The thread who had locked monitor, is said to be owner of monitor and only that thread can unlock the monitor.

- Java provides "synchronized" keyword to deal with monitors / synchronization.
- synchronized block:
  - example:

```
void fun() {
        // ...
        synchronized(obj) {
                // ...
        }
        // ...
}
```

- When certain part of the method should be executed by single thread at a time, we can use "synchronized" block.
- When a thread begin execution of the sync block, it will lock the monitor associated with given object (obj) and continue execute within that block.
- Meanwhile if another thread try to execute same sync block, it will try to lock the object monitor again.
- Since object is already locked, the second thread will be blocked, until first thread release the lock (at the end of sync block).
- When first thread release the lock, the waiting second thread will acquire it and continue to execute the block.

**- synchronized method:**

- When non-static sychronized method is invoked by the thread, it will lock the monitor associate with current object ("this"). When method completes, thread will release the lock.
- When static synchronized method is invoked by the thread, it will lock the monitor object associated with "ClassName.class" object.

- This ensure that two threads cannot execute same synchronized method on the same object.
- example:

```
class Account {
	private double balance;
	// ...
	public synchronized void withdraw(double amt) {
		// get cur balance from db
		// balance = balance - amt;
		// update new balance into db
	}
	public synchronized void deposit(double amt) {
		// get cur balance from db
		// balance = balance + amt;
		// update new balance into db
	}
	public double getBalance() {
		// get cur balance from db
		// return balance;
	}
}
```

- a1.withdraw(...) and a1.withdraw(...) executed by two different threads at the same time will not execute parallely. (As "a1" object is locked by thread1 and thread2 must wait for it).
- a1.withdraw(...) and a1.deposit(...) executed by two different threads at the same time will not execute parallely. (As "a1" object is locked by thread1 and thread2 must wait for it).
- a1.withdraw(...) and a2.withdraw(...) executed by two different threads at the same time can execute parallely. As lock is acquired by two threads on two different objects.
- a1.withdraw(...) and a2.getBalance(...) executed by two different threads at the same time can execute parallely. Because getBalance() method is not synchronized and hence the thread is neither trying lock the object nor will get blocked.
- If all methods in a class are synchronized, only one thread can perform any operation on object of that class. Such class is called as "synchronized" / "thread-safe" class. E.g. StringBuffer, Vector, Hashtable, etc.

**Day 11:**

==========

+ Sockets:

- Socket is an IPC mechanism used to communicate between two processes on the same machine or different machines (connected over network).
- Socket is defined as communication endpoint.
- Socket = IP address + port number
  - IP address : unique identifier for machine over network
  - port number: 16-bit logical identifier for the socket on a machine.
    - Range for port number : 0 to 65535
    - Ports 0 to 1023 are reserved for system use. Programmer should choose any other unused port.
- There are two main protocols:
  1. TCP:
     - Tranmission Control protocol
     - Connection Oriented
     - Acknowledgement is sent to transmitter
     - Reliable
     - Stream based
  2. UDP:
     - User datagram protocol
     - Connection less
     - No acknowledgement
     - Less Reliable
     - Packet based
- In java, UDP protocol communication is done using classes : DatagramSocket and DatagramPacket.
- In java, TCP protocol communication is done using classes : ServerSocket and Socket.

+ Remote Method Invocation:

- "Remote Procedure Call - RPC" is IPC mechanism.

- RMI demo : four coding steps:
  - **step1: MyIntf.java**

```
public interface MyIntf extends Remote {
        int sum(int a, int b) throws RemoteException;
        Student search(int roll) throws RemoteException;
}
```

  - Write a java interface containing methods to be called from client side.

- It must be inherited from marker interface "Remote".
- Each method in interface must throw "RemoteException".

- **step2: MyImpl.java**

```
public class MyImpl extends UnicastRemoteObject implements MyIntf {
        public MyImpl() throws RemoteException {
        }
        public int sum(int a, int b) throws RemoteException {
                return a + b;
        }
        public Student search(int roll) throws RemoteException {
                // ...
                return null;
        }
}
```

- Write java class to implement above remote interface and provide proper logic.
- This class must be inherited from UnicastRemoteObject.
- Class must have a paramless constructor throwing RemoteException.

- **step3: ServerMain.java**

```
public class ServerMain {
        public static void main(String[] args) throws Exception {
                MyImpl obj = new MyImpl();
                Naming.bind("abc", obj);
        }
}
```

- Create object of impl class and associate it with some name (friendly name).

- **step4: ClientMain.java**

```
public class ClientMain {
        public static void main(String[] args) throws Exception {
                MyIntf obj = (MyIntf)Naming.lookup("abc");
                int res = obj.sum(12, 5);
                System.out.println("Sum : " + res):
        }
}
```

- Find the object with name given at server side, type cast to interface ref and then call methods on it.

* Note:

Usually remote interface and other classes which are required for both client and server, are kept in the seperate package. The implementation class and server side main is kept in a package and finally client main class is kept into one more package.

* Note:

In order to execute server and client code from two different machines on the network, Client side Naming.lookup() method should contain complete URL of the server e.g. rmi://server-ip:1099/objname

Also in this case, server and common package should be deployed on the server machine; while client and common package should be deployed on client machine.

- Compilation and Execution of RMI code:
  - step5: Compile all above files:
    command: javac -d . *.java
  - step6: Run "rmiregistry" from the dir in which server package is kept (in command prompt 1).
    command: rmiregistry
  - step7: Run ServerMain from the dir in which server package is kept (in command prompt 2).
    command: java pkg.ServerMain
  - step8: Run ClientMain from the dir in which client package is kept (in command prompt 3).
    command: java pkg.ClientMain

**- RMI:**

- Calling java method from another JVM (process) on same or different machine (over network) is called as "RMI".
- Internally Stub and Skeleton objects are used to execute the method and receive result at the client side as shown in diag.

- Remote interface must be inherited from marker interface "java.rmi.Remote". It allows JVM to create stub for the given remote interface.
- Each method in remote interface and consructor in the implementation class must throw RemoteException. Since RMI internally performs networking, network connection may fail at runtime. So java forces each method to throw RemoteException, so that programmer can implement some logic to handle such exception.
- The implementation class must be inherited from "java.rmi.server.UnicastRemoteObject". The functionality of this class is internally used by the skeleton at the server-side.

**- RMIREGISTRY:**

- The info about impl class object (remote interface methods) is kept into rmiregistry associated with some "name". This is done by the Naming.bind() method at server side.
- When client program calls Naming.lookup(), it internally communicates with rmiregistry running on server and receive info about remote object. This info is used to create stub at runtime.

- Naming.lookup() returns pointer to stub object, on which client program can execute methods. These stub methods internally communicate with server skeleton to receive result.

- In the older java versions, stubs were not created at runtime. They should be created manually using "rmic" (RMI compiler) command.

- **RMI architecture:**

**- Marshalling & Unmarshalling:**

- Sending object contents over the network after serialization, is called as "marshalling".
- Receiving object contents from network (in form of byte sequence) and deserializing it, is called as "unmarshalling".

**+ Java DataBase Connectivity - JDBC:**

- JDBC Driver is a special program used to convert java requests to database understandable form and database response to java understandable form.
- There are four types of JDBC driver:

**1. Type I driver:**

- Also called as "JDBC ODBC Bridge Driver".
- ODBC drivers are implemented in C and can connect to older databases like Foxpro, Access, etc.
- Java has builtin JDBC driver to communicate with ODBC driver which in turn work with database.
- Name of driver class is sun.jdbc.odbc.JdbcOdbcDriver
- Due to multiple layers, it is slower.
- Also ODBC support is not fully present on all platforms.

**2. Type II driver:**

- Partially implemented in C and partially in Java.
- Use JNI to communicate betn Java and C layer.
- Better performance than Type I.
- Different driver for each database.
- Not truely portable (as partially in C).
- Outdated.
- e.g. Oracle OCI driver

**3. Type III driver:**

- JDBC driver communicate with a middleware, which in turn communicate with actual database.
- e.g. Weblogic RMI driver

**4. Type IV driver:**

- Completely implemented in Java and hence truely portable.
- Different for each database.

- Internally use sockets to communiate with database.
- e.g. Oracle Thin Driver.

**+ JDBC is a specification:**

- Specification is given in form of interfaces by "Sun Microsystem". Important Interfaces:
- Driver : Used to create and manage database connections.
- Connection : Encapsulate socket connection which communicate with database.
- Statement : Encapsulate SQL query to be executed on db.
- ResultSet : Represents database response to the query (rows and cols), used to fetch result row by row.
    - PreparedStatement : Used to execute parameterized query on the database.
    - CallableStatement : Used to execute stored procedure on the database.
- The specification given by Sun is implementeb by each JDBC driver. So internally each JDBC driver is set of classes inherited from above given interfaces. This ensures that same methods are present in all driver implementations.
- Thus JDBC program remains same irrespective of which JDBC driver or database you are using.

**+ JDBC Example:**

//0. adding JDBC driver into CLASSPATH

+ ON COMMAND LINE:
    - set CLASSPATH=/path/of/file/driver.jar;%CLASSPATH%
    - javac -d . ClassName.java
    - java pkg.ClassName
+ ON ECLIPSE:
    - Right click on project, Add New Folder. Give name "lib".
    - Copy driver jar file into "lib".
    - Right click on project, go to properties. Click "Java Build Path". Click "Libraries", then "Add Jars". Now select driver jar from "lib" directory within current project.

```
public class Demo46_Main1 {
	// oracle.jdbc.OracleDriver
	public static final String DB_DRIVER = "com.mysql.jdbc.Driver";
	// jdbc:oracle:thin:@adc:1521:oracle
	public static final String DB_URL = "jdbc:mysql://localhost:3306/test";
	public static final String DB_USER = "root";
	public static final String DB_PASS = "nilesh";
	public static void main(String[] args) throws Exception {
		//1. load and register jdbc driver
```

```
            Class.forName(DB_DRIVER);
            //2. create connection object
            Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS);
            //3. create statement object
            Statement stmt = con.createStatement();
            //4. execute query and get result
            String sql = "SELECT DEPTNO, DNAME, LOC FROM DEPT";
            ResultSet rs = stmt.executeQuery(sql);
            //5. process the result
            while(rs.next()) {
                int deptno = rs.getInt("DEPTNO");
                String name = rs.getString("DNAME");
                String loc = rs.getString("LOC");
                System.out.printf("%d, %s, %s\n", deptno, name, loc);
            }
            //6. close connection, statement, ...
            rs.close();
            stmt.close();
            con.close();
        }
}
```

+ JDBC Insert Example:

```
public class Demo46_Main3 {
    public static final String DB_DRIVER = "com.mysql.jdbc.Driver";
    public static final String DB_URL = "jdbc:mysql://localhost:3306/test";
    public static final String DB_USER = "root";
    public static final String DB_PASS = "nilesh";
    public static void main(String[] args) throws Exception {
        //1. load and register jdbc driver
        Class.forName(DB_DRIVER);
        //2. create connection object
        Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS);
        //3. create statement object
        Statement stmt = con.createStatement();
        //4. execute query and get result
        Scanner sc = new Scanner(System.in);
```

```
            System.out.println("enter deptno, name, loc : ");
            int deptno = sc.nextInt();
            String name = sc.next();
            String loc = sc.next();
            String sql = String.format("INSERT INTO DEPT VALUES(%d,'%s','%s')", deptno, name, loc);
            int count = stmt.executeUpdate(sql);
            System.out.printf("Records Inserted : %d\n", count);
            //5. process the result
            //6. close connection, statement, ...
            stmt.close();
            con.close();
        }
}
```

+ Using Statement:

```
        Class.forName("....");
        con = DriverManager.getConnection("...", "..", "..");
        Statement stmt = con.createStatement();
        int dno = sc.nextInt();
        rs = stmt.executeQuery("SELECT * FROM DEPT WHERE DEPTNO="+dno);
        while(rs.next()) {
            // ...
        }
        con.close();
```

+ Using PreparedStatement:

```
        Class.forName("....");
        con = DriverManager.getConnection("...", "..", "..");
        PreparedStatement stmt = con.prepareStatement("SELECT * FROM DEPT WHERE DEPTNO=?");
        int dno = sc.nextInt();
        stmt.setInt(1, dno);
        rs = stmt.executeQuery();
        while(rs.next()) {
            // ...
        }
        con.close();
```

+ Using CallableStatement : executing stored procedure

```
Class.forName(".....");
con = DriverManager.getConnection("", "", "");
CallableStatement stmt = con.prepareCall("{CALL sp_adddept(?,?,?)}");
dno = sc.nextInt();
stmt.setInt(1, dno);
name = sc.next();
stmt.setString(2, name);
loc = sc.next();
stmt.setString(3, loc);
stmt.execute();
```

+ DriverManager:

- Manual driver registration:

```
Class c = Class.forName("pkg.DriverClassName");
Driver drv = (Driver)c.newInstance();
DriverManager.registerDriver(drv);
```

**- Connection Creation:**

```
con = DriverManager.getConnection(DB_URL, ...);
```

DriverManager internally search for appropriate driver object (already registered) and then call its connect() method to create actual jdbc Connection and return it.

**+ Scrollable ResultSet:**

- While creating the statement, type and concurrency of result set can be specified.

- ResultSet types:

**- ResultSet.TYPE_FORWARD_ONLY**

- Only forward access, no reverse and random access.
- This is default type.

**- ResultSet.TYPE_SCROLL_INSENSITIVE**

- Can perform forward, reverse and random access.
- Any changes done in the database while resultset is in use, will not reflect into resultset.

**- ResultSet.TYPE_SCROLL_SENSITIVE**

- Can perform forward, reverse and random access.

- Any changes done in the database while resultset is in use, will reflect into resultset.

- **ResultSet functions:**
  - **To fetch records:**
    - boolean beforeFirst();
    - boolean afterLast();
    - boolean first();
    - boolean last();
    - boolean next();
    - boolean previous();
    - boolean absolute(int row);
    - boolean relative(int relRow);
  - **To check result set position:**
    - boolean isBeforeFirst();
    - boolean isAfterFirst();
    - boolean isFirst();
    - boolean isLast();

**+ Updateable ResultSet:**

- While creating the statement, type and concurrency of result set can be specified.
- concurrency types:

**- ResultSet.CONCUR_READ_ONLY:**

- default type
- using result set records can be only fetched from database.

**- ResultSet.CONCUR_UPDATEABLE:**

- using resultset records can fetched and also manipulated (insert, update, delete operations can be performed on the current record).
  - update related functions:
    - deleteRow();
      - delete current row from resultset and database
    - updateRow();
      - update changes made in current row into the database.
    - insertRow();
      - add newly created row into the database.

**+ CachedRowSet:**

- Inherited from ResultSet interface.
- Fetch all rows into the client memory, so that for accessing each row no need to connect to the database.
- In this case connection can be closed while data is accessed at the client side (after receiving CachedRowSet). Due to this more number of clients can connect to database.

- However if large of number of records are fetched into client memory, performance will be reduced (due to shortage of memory).

**+ Images in database:**

// CREATE TABLE STUDENTS (ROLL INT PRIMARY KEY, NAME VARCHAR(20), MARKS DOUBLE, PHOTO BLOB);

- load and register driver
- get connection
- PreparedStatement stmt = con.prepareStatement("INSERT INTO STUDENTS(ROLL,PHOTO) VALUES (?,?)");

```
stmt.setInt(1, roll);
Blob bl = new BlobClassName();
OutputStream blobOut = bl.setBinaryStream(0);
FileInputStream fin = new FileInputStream("D:\\pic.jpg");
int b;
while((b=fin.read())!=-1)
        blobOut.write(b);
blobOut.flush();
fin.close();
blobOut.close();
stmt.setBlob(2, bl);
int cnt = stmt.executeUpdate();
```

**+ Transactions:**

- Transaction is set of SQL statements to be executed as a single unit. If all statements are executed successfully, then all of them should be commited to database. If any of the statement fails, then already executed statements should be rollbacked.

- example:

```
try {
        con.setAutoCommit(false);
        stmt1 = con.prepareStatement("UPDATE ...");
        stmt2 = con.prepareStatement("UPDATE ...")
        // ...
        stmt1.executeUpdate();
        stmt2.executeUpdate();
        con.commit();
} catch(Exception e) {
        con.rollback();
        // ...
} finally {
        // closing
}
```