Slogan of java: write once,run anywhere Slogan of .net : run once ,run always

Syllabus Java Technologies-I(Core Java)

```
Data Types, Operators and Language, Constructs, Inner Classes and
Inheritance, Interface and Package, Exceptions, Collections, Threads,
Java.lang, Java.util,  Java.io , Java Persistent, Servlets, Java Virtual
Machine
```

1. Java intro : 1 day
2. Datatype : 1 day
3. JDBC : 2 days
4. Annotation and reflection : 1 day
5. Multithreading : 2 days
6. FIle handling and socket : 2 day
7. Functional programming : 1 day
8. Collection framework : 2 day
9. Interface : 1 day
10. Abstract class and generics : 1 day
11. Exception Handing : 1 day
12. Package and Array : 1 day
13. Enum, Association ,inheritence : 1 day

Day 1

1. **ppt on basic of java**

Java introduction

1. Java technology is both a programming language and a platform.
2. developed by James Gosling in 1991, at Sun Microsystems
3. slogen "Write once , run anywhere"
4. The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

- Simple
- Object oriented
- Distributed
- Multithreaded

- Dynamic
- Architecture neutral
- Portable
- High performance
- Robust
- Secure
- In the Java programming language, all source code is first written in plain text files ending with the .java extension.
    - Those source files are then compiled into .class files by the javac compiler.
    - A .class file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the JVM1 (Java VM).
    - The java launcher tool then runs your application with an instance of the JVM.
- Because the Java VM is available on many different operating systems,
    - the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS.
    - Some virtual machines, such as the Java SE HotSpot at a Glance, perform additional steps at runtime to give your application a performance boost. This includes various tasks such as - finding performance bottlenecks and recompiling (to native code) frequently used sections of code.

5. **The Java Platform**

- A platform is the hardware or software environment in which a program runs.
- Most platforms can be described as a combination of the operating system and underlying hardware.
- The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.
- The Java platform has two components:
    1. **The JVM**
        - it's the base for the Java platform and is ported onto various hardware-based platforms.
    2. **The Java Application Programming Interface (API)**
        - The API is a large collection of ready-made software components that provide many useful capabilities.
        - It is grouped into libraries of related classes and interfaces; these libraries are known as packages. T
- The API and JVM insulate the program from the underlying hardware.
- As a platform-independent environment, the Java platform can be a bit slower than native code.

- However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

requirement for JAVA :

- developer need for java development

    1. **SDK (Software development kit)** + development tools + runtime environment + Documentation + SUpprting libraries
    ◦
    1. **Java Software developmnt kit (JDK)**

```
+ Java Developmetn tools
+ Java APi Docs (Application programming runtime)
+ rt.jar [its supporting libraries]
   - same as .NET MSCORLIB file
   - core java logic kept as  compiled library files
+ JVM (JVM)
```

    2. JRE(Java Runtime Environtment) = **[rt.jar + JVM]** - now JDK =
        - Java Development tools
        - Java APi Docs
        - JRE

JDK Directory folders

1. bin

- Contains Java language Developement tools
- we are gonna use
    - java,javadoc,javah,javac

2. Include folder contains header file to use JNI

- JNI = Java Native Interface
- Native code = C,C++ code
- we can use JNI to write code in native language and run in java app

3. lib

- contains the files used by development tools
- contains jar files
- JAR (Java ARchive) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution.
- JAR files are archive files that include a Java-specific manifest file.

4. src

- it is a zip file
- .java is extension of java source file
- contain source code of JAVA API

5. docs

- contains documentation of java api
- i.e html files

6. jre

- root directory for java runtime Environmrnt

7. man

- contians documentation of java developemnt tools

Java Package

- contains

1. Interfaces
2. Classes
3. Enum
4. Exceptions
5. Errors
6. Annotation Types
7. Sub classes
8.

command to open java manual

```
> man java

> man javac

> man jar

> man java

> man javap
- Disassembles one or more class files.
```

Java program

1. first program

- hello world program

```
class Program{

 public static void main(String[] args){
//c
//printf("Hello world \n");
// C++
//cout<<"Hello world"<<endl;
//c#
//Console.Readline("Hello world");


  System.out.println("Hello World");
 }



}
```

2. **command to compile and run java files**

- to compile .java file

> javac Program.java

- it creates a .class file, now to run the java file

> java Program

1. flow of java
2. JRE
3. compile classes as output are created for

- per class in .java file
- e.g
- 1. Interfaces
  2. Classes
  3. Enum

Day 2

- (reference) **ppt on Buzz words in java**
- these are design goals of java:
- ten principle of sun microsystem :

1. **Simple**

- java is derived from C/C++
- it is simple in comparison on C and C++
  - on basis of syntax

- has constructor chaining

2. **Object Oriented**

-

3. **Architecture Neutral**
4. **Portable**
5. **Robust**
6. **Multithreaded**
7. **Dynamic**
8. **Secure**
9. **High Performance**
10. **Distributed**

class notes

1. static member function and constructor cant be virtual , all other clases abd member function be virtual

2. older version program can run on later version, features can get deprecated, but still have backward compatibility,

    - this is not in .NET, there Obsolete features are not supported in future version

3. reflection example is debugger for getting private data member , intellisense

4. The Java **Remote Method Invocation** (RMI)

- system allows an object running in one JVM to invoke methods on an object running in another JVM.
- RMI provides for remote communication between programs written in the Java programming language.
- since java supports RMI, it is distributed prog languauge.

5. **Access Modifier**

- 12 types in java
- in comparison to c /c++ , 4 types
-

6. **A Wrapper class**

- is a class whose object wraps or contains primitive data types.
- When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. - In other words, we can wrap a primitive value into a wrapper class object.

practical demos

1. in java we can overload main method (also call one main to another), in other language ??

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

    System.out.println("Hello ");

    //Program::main(10,20); c++
    Program.main(1, 2);
    main(10,20);
}

// in java we can overload main method
public static void main(int num1,int num2)
{
   System.out.println("Num1 :" + num1);
   System.out.println("Num2 : " + num2);
}
```

2. we can write , main method per class, but only one will be starting main method in it , we need to select it
    - calling main method job of main thread .

```
package test;
{}

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

        System.out.println("A.main ");

    }
   }

   public class Program {

      public static void main(String[] args) {

        System.out.println("Program.main ");

    }
}
```

3. different print methods in printstream class

```
public static void main3(String[] args) {

    String name = "suraj porje";
```

```
    int empid = 26;
    float salary = 90000.45f;
    //System.out.println(name + " " + empid + " " + salary);

    System.out.printf("%-25s%-5d%-10.2f \n",name,empid,salary);
    name = "sallu";
     empid = 36;
     salary = 60000.45f;
     System.out.printf("%-25s%-5d%-10.2f \n",name,empid,salary);
    // System.out.println(name + " " + empid + " " + salary);
    }

    public static void main2(String[] args) {

       System.out.println("DAC,"); // new line print

   }


      public static void main1(String[] args) {

       System.out.print("DAC,"); // same line print

   }
```

4. when both primitive datatype need conversion , use c type casting
   - widening conversion, smaller data type to large datatype conversionm, not required explicit casting
   - narrowing conversion, large datatype conversion to smaller data type, explicitely casting in required

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

    double num1 = 10.5; //Initialization
    int  num2 ; // narrowing
    num2 = (int) num1;
    System.out.println("Num2 : " + num2);
    }


public static void main2(String[] args) {

    int num1 = 10; //Initialization
    double num2 = (double)num1; //wideening
    System.out.println("Num2 : " + num2);

    }

 public static void main1(String[] args) {
```

```
    int num1 = 10; //Initialization
   // C/C++ type casting
    double num2 = (double) num1 ; // Initialization

     System.out.println("Num2 : " + num2);


  }
```

3. primitive type are itself not classes, so need wrapper class boxing
    - conversion of primitive type to non primitive type by method it is caleed boxing
    - e.g for string can use ,method
    - toString()
    - valueOf

```
 public class Program{
public static void main(String[] args) {
 String str = "1a2b5"; // Non primitive

 int number = Integer.parseInt(str); // Primitive : Unboxing

 System.out.println("number : " + number);
 //error : seen in stack trace
}


public static void main4(String[] args) {
 String str = "125"; // Non primitive

 int number = Integer.parseInt(str); // Primitive : Unboxing

 System.out.println("number : " + number);
}

public static void main3(String[] args) {
 double number = 10.5;
 String strNumber = Double.toString(number);//Non Primitive : Boxing
 System.out.println(strNumber);
}
public static void main2(String[] args) {
 int number = 10;    //Primitive
 //String strNumber = Integer.toString(number); //Non Primitive : Boxing
 String strNumber = String.valueOf(number);//Non Primitive : Boxing
 System.out.println("Number  :   "+strNumber);
}

}
```

6. use of scanner class to take input from user, demo

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

        Scanner sc = new Scanner(System.in);
        System.out.print("Name : ");
        String name = sc.nextLine();

        System.out.print(" Roll no : ");
        int rollNumber = sc.nextInt();
        System.out.print(" Marks : ");
        float marks = sc.nextFloat();

        System.out.println(" Name : " + name + " roll no : " + rollNumber +
" marks : " + marks);

    }
```

to read

1. JVM

> https://www.artima.com/insidejvm/ed2/jvm.html

2. why main method is static? (an swer as what when it is not static)
3. to compile java program , need atleast one class
4. Package java.lang

> https://docs.oracle.com/javase/8/docs/api/

5. can we write helloworld program , without giving semicolon ?? yes

Day3

- **reference for demo**

```
+ Stream
  - It is an abstraction(object) which either produces(write) or consumes( read)information
    from source to destination.
  - Console = Keyboard(Console Input) + Monitor/Printer(Console Output)

  - Standard Stream objects of C associated with Console.
    1. stdin
    2. stdout
    3. stderr

  - Standard Stream objects of C++ associated with Console.
    1. cin
    2. cout
    3. cerr
    4. clog

  - Standard Stream objects of Java associated with Console.
    1. System.in -> represents -> Keyboard
    2. System.out -> represents -> Monitor
    3. Systeme.err -> Error Stream -> represents -> Monitor

  - Standard Stream objects of C# associated with Console.
    1. System.Console.In
    2. System.Console.Out
    3. System.Console.Error
```

```
+ Reading record from console(keyboard)
  - In C : scanf function
    * Example : scanf("%d", &number );

  - In C++ : std::cin object
    * Example : std::cin >> number;

  - In C# : System.Console.ReadLine() function
    * Example : number = Convert.ToInt32( Console.ReadLine())

  - In Java we can use
    1. Use readLine() method of java.io.Console class
       Console console = System.console( );
       String name = console.readLine( );

    2. Use java.io.BufferedReader class
       BufferedReader reader = new BufferedReader( new InputStreamReader( System.in));
       String name = reader.readLine( );

    3. Use java.util.Scanner class
       Scanner sc = new Scanner( System.in );
       String name = sc.nextLine( );

    4. Use JOptionPane Class
       String name = JOptionPane.showInputDialog("Enter name ");
```

demo on java

- 1. JOptionPane

```
import javax.swing.JOptionPane;



public static void main1(String[] args) {
        String name = JOptionPane.showInputDialog("Enter Name");
        int empid = Integer.parseInt( JOptionPane.showInputDialog("Enter
Empid") );
        float salary = Float.parseFloat(JOptionPane.showInputDialog("Enter
Salary"));

        System.out.println("Name    :   "+name);
        System.out.println("Empid   :   "+empid);
        System.out.println("Salary  :   "+salary);
    }
```

- 2. BufferedReader and InputStreamReader

```
 import java.io.BufferedReader;
import java.io.InputStreamReader;
public static void main3(String[] args)throws Exception {
        BufferedReader reader = new BufferedReader( new
InputStreamReader(System.in));

        System.out.print("Name  :   ");
        String name = reader.readLine();
        System.out.print("Roll Number   :   ");
        int number = Integer.parseInt( reader.readLine() );
        System.out.print("Marks :   ");
        float marks = Float.parseFloat(reader.readLine());

        System.out.println("Name    :   "+name);
        System.out.println("Number  :   "+number);
        System.out.println("Marks   :   "+marks);
    }
```

- 3. scanner

```
import java.util.Scanner;
public static void main4(String[] args) {

        try(Scanner sc = new Scanner(System.in)){
            System.out.print("Name  :   ");
            String name = sc.nextLine();
            System.out.print("Number    :   ");
            int number = sc.nextInt();
```

```
            System.out.print("Balance   :   ");
            float balance = sc.nextFloat();

            System.out.println("Name    :   "+name);
            System.out.println("Number  :   "+number);
            System.out.println("Balance :   "+balance);
        }
    }
```

- 4. demo: Anonymous Instance
- if we create instance without refrence is anonymous instance use when
    - 1. when we want to use instance only once
        - to pass method as a argument

```
Person p = new Person();
p.setName("suraj");
p.setBirthDate(new Date(23,7,1994));
```

    - 2. if we want to use any instance as a method argument or if want to use any instance as a exception then it should by anonymous

```
class Accounts{
    //Field // default -> package level private
  private int number; // 0
  private String type; // null
  private float balance; // 0.0
}

main(){
new Accounts(); //instance without name, is anonymous instance
}
```

- 5. Constructor chaining
    - this it, to achieve constructor reusability, we can call constructor from another constructor
    - for CC we should we this statement inside constructor body
        - this statement must be first statment in constructor body

```
public Account() {
        this(1105, "Current", 85000);   //Constructor Chaining
    }
    public Account(int number, String type, float balance) {
        this.number = number;
```

```
        this.type = type;
        this.balance = balance;
    }
```

- 6. Null Object
- 1. Literal (constant value) in Java:
- literal : datatype

1. true : boolean
2. 'A' : char
3. 12345 : int
4. 3.142 : double
5. "Sunbeam" : String
6. new Date(): Date
7. null : reference variable

```
---

in C/C++
//here null is a macro ,
int *ptr = NULL;
---

in java

// null is a literal, to initialize an object
Account account = null;

//if refernece value contains null value , it its called as null
object/null reference variable
// gives error : NullPointerException
   //Calling the instance method of a null object.

to solve two methods
1.
         if(acc1 != null) { acc1.printRecord(); }
2.  acc1 = new Accounts(); //instantiation
```

- 7. **Creating refrence(instance)**
  - if we want to perform operations on instance then it is neccessary to create reference to it.

```
Account acc1; // in C++ : object
Account acc2; // in java: Reference
Account acc3 = new Account(); // in java: acc3: Reference
```

- consider following statement

```
    Accounts a1 = new Accounts(1001,"Saving",45000);
    Accounts a2 = a1;
   Account a3 = new Account(1002,"Saving",45000);
```

- answer : 2 instance, 3 reference
- 8. Comments in java

```
1. //Single line comment
2. /* Multi line comment */
3. /** Java doc comment **/
```

- 9. in class ,
  - right click,
  - select source --> generate getter/setter
- 10. menu driver program, for account class

```
public class Program {
    static Scanner sc = new Scanner(System.in);

    public static void acceptRecord(Account account) {
        System.out.print("Number    :   ");
        account.setNumber(sc.nextInt());
        System.out.print("Type  :   ");
        sc.nextLine();
        account.setType(sc.nextLine());
        System.out.print("Balance   :   ");
        account.setBalance(sc.nextFloat());
    }

    public static void printRecord(Account account) {
        System.out.println("Number  :   " + account.getNumber());
        System.out.println("Type    :   " + account.getType());
        System.out.println("Balance :   " + account.getBalance());
    }

    public static int menuList() {
        System.out.println("0.Exit");
        System.out.println("1.Accept Record");
        System.out.println("2.Print Record");
        System.out.print("Enter choice  :   ");
        int choice = sc.nextInt();
        return choice;
    }
```

```
    public static void main(String[] args) {
        int choice;
        Account account = new Account();
        while ((choice = Program.menuList()) != 0) {
            switch (choice) {
            case 1:
                Program.acceptRecord(account);
                break;
            case 2:
                Program.printRecord(account);
                break;
            }
        }
    }

    public static void main1(String[] args) {
        Account account = new Account();

        Program.acceptRecord(account);

        Program.printRecord(account);
    }
}
```

- 11. modular approach
  - using 3 class files
  1. Class (contain class data member,cons,getter/setter)
  2. ClassTest(contains method of class to implement, with menuList method)
  3. Program (contains menu driver logic)

class Notes :

+ java.lang.Object
  - It is non final concrete class which is declared in java.lang package.
  - It is ultimate base class/root of java class hierarchy/super cosmic base class.
  - In other words, it doesn't implement any interface and doesn't extends and class.
  - In Java every class(not interface) is directly or indirectly extended from java.lang.Object.

```
class Program{
  p.s.v.m(String[] args ){

  }
}
```

Is same as

```
class Program extends Object{
  p.s.v.m(String[] args ){

  }
}
```

  - Points to remember
    1. Object class do not contain any nested type(interface, class, enum).
    2. Object class do not have any field.
    3. Object class contains only default constructor.
    4. Object class contains 11 methods:
      - 5 non final methods(2 native methods)
      - 6 final methods(4 native methods)
  - A method which is implemented in C++ and used in Java.
  - To check details of all the methods use following command:
    javap java.lang.Object

+ Methods of java.lang.Object class
  - Non final methods of Object class:
    1. public String toString( );
    2. public boolean equals(Object obj);
    3. public native int hashCode( );
    4protectednative Object clone( ) throw CloneNotSupportedException
    5. protected void finalize( ) throws Throwable
  - Final methods of java.lang.Object class:
    6. public final native Class<?> getClass( );
    7. public final void wait( )throws InterruptedException
    8. public final native void wait(long timeout)throws InterruptedException
    9. public final void wait(long timeout, int nanos)throws InterruptedException
    10. public final native void notify( );
    11. public final native void notifyAll( );

+ Class
  - Definition:
    1. It is a collection of fields and methods.
    2. Structure and behavior of an instance depends on class hence class is considered as a template/model/blueprint for instance.
    3. Class represents group/collection of all such subject which is having common structure and common behavior.
  - Class definition represents encapsulation.
  - In java, we can declare/define following members:
    1. Nested Type(Interface, class, enum)
    2. Field(In C++, Data Member)
    3. Constructor
    4. Methods(In C++, Member function)

+ Instance
  - Definition
    1. In java, object is called as instance.
    2. Any entity, which has physical existance or which get space inside memory is called instance.
    3. Any entity, which has state, behavior and identity is called instance.
  - Instance instance only non static field get space.
  - If we want to create instance of a class then it is mandatory to use new operator.
  - In java, instance of class get space on Heap.

+ Procedure to give solution the problem statement using OOP language like C++/C#/Java
  1. Understand and analyse problem statement.
  2. Decide and declare classes with fields.
  3. Instantiate the class i.e create instance of class. Inside instance only non static field will get space.
  4. To initialise instance define constructor inside class.
  5. To process state of the object, define methods inside class.
  6. If we call non static method on instance then this reference is passed to the method.
  7. Using this reference we can process state of instance inside method.

+ Access Modifier:
  - If we want to control visibility of members of a class then we should use access modifier.
  - Access modifier to control visibility:
    1. private
    2. package level private (default)
    3. protected
    4. public

```
class Student{
  private int marks;
  public void setMarks( int marks ){
    if( marks >= 0 && marks <= 100 )
      this.marks = marks;
    else
      throw new IllegalArgumentException("Invalid marks");
  }
}
```

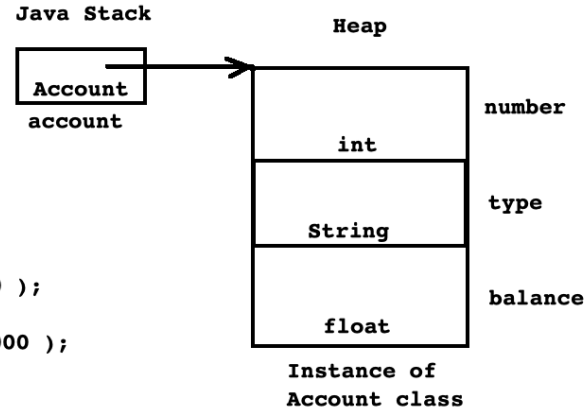
- If we create instance w/o reference then it is called anonymous instance.
- Example:
  new Account( ); //anonymous instance
- If we want to use any instance only once then we should create anonymous instance.
- Example:
  Person p = new Person( );
  p.setName("Sandeep");
  p.setBirthDate(new Date(23,7,1983));

- If we want to use any instance as a method argument or if want to use any instance as a exception then it should be anonymous.

- If we want to perform operations on instance then it is necessary to create reference to it.

- Example:

```
Account acc1; //In C++ : Object

Account acc2; //In Java : Reference
acc2 = new Account( );

Account acc3 = new Account( );
```



- Consider following statements:

```
Account a1 = new Account(1001, "Saving", 45000 );
Account a2 = a1; //Shallow Copy of reference
Account a3 = new Account(1002, "Current", 125000 );
```

- Comments in Java
  1. //Single line comment
  2. /* Multi line comment */
  3. /** Java Documentation comment */

+ this reference
  - this is keyword in java.
  - It is implicit reference variable available in every non static method of a class which is used to store reference of current/calling instance.
  - Using this reference non static field and method can communicate with each other hence it is considered as a line/connection between them.

+ Constructor
  - If we want to intialize instance then we should use constructor.
  - Types:
    1. Parameterless constructor
    2. Parameterized constructor
    3. Default constructor
  - To achieve constructor reusability, we can call constructor from another constructor. It is called constructor chaining.
  - For constructor chaining, we should use this statement inside constructor body.
  - this statement must be first statement inside constructor body.

+ Literals in Java
  1. true : boolean
  2. 'A' : char
  3. 12345 : int
  4. 3.142 : double
  5. "Sunbeam" : String
  6. null : reference variable

+ In C/C++ : int *ptr = NULL;
+ In Java : Account account = null;

extra points

1. in java has no structure and union
2. in java ,object is called as instance

Day4

+ Difference between value and reference Type

1. **Primitive type is called value type.**
1. **Non primitive type is called reference type.**

2. **boolean, byte, char, short, int, long, float, double are value types.**
2. **interface, class, enum, array are reference types.**

3. **Variable of value type contains value.**
3. **Variable of reference type contains reference.**

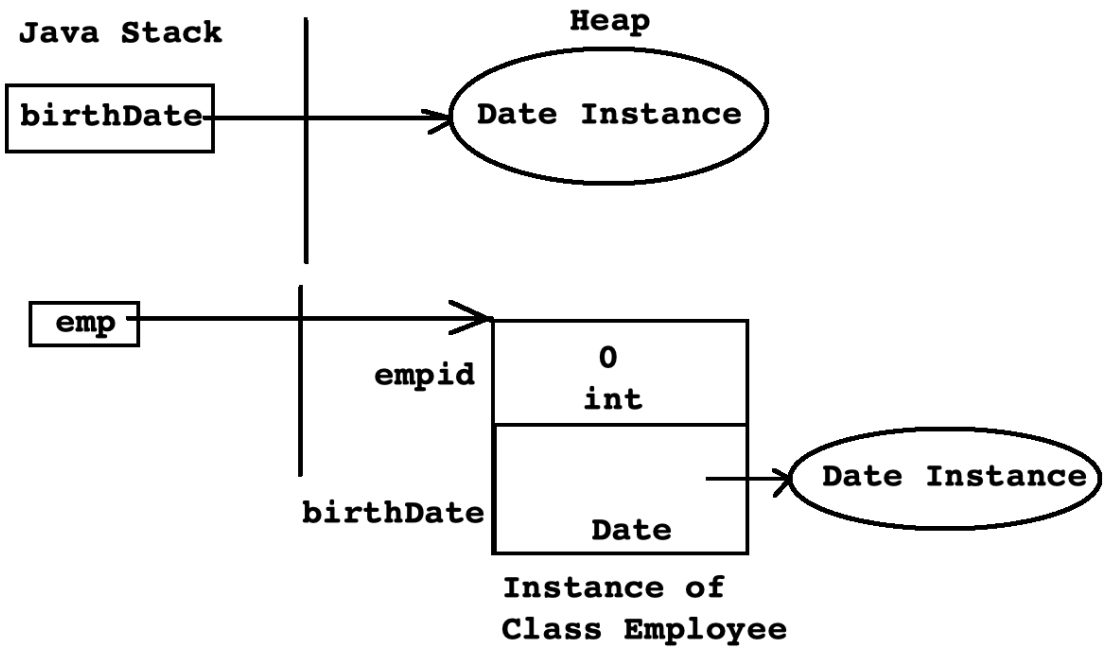
4. **Variable(field) of value type by default contains 0.**
4. **Variable(field) of reference type by default contains null.**

5. **Variable of value type do not contain null value.**
5. **Variable of reference type can contain null value.**

6. **We can not create instance of value type using new operator.**
6. **It is mandatory to use new operator to create instance of reference type.**

7. **Instance of value type get space on Java stack.**
7. **Instance of reference type get space on Heap.**

8. **If we assign variable of value type to the another variable of value type then value gets copied.**
8. **If we assign variable of reference type to the another variable of reference type then reference gets copied.**



+ toString() method

- **It non native / non final method of java.lang.Object class.**
- **Syntax:**
  public String toString( );
- **Implementation inside Object class:**

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- **If we want to return state of instance in String format then we should use this method.**
- **If we do not define toString method inside class then it's super class's toString() method gets called.**
- **toString() method of java.lang.Object class returns String in following format:**
  F.Q.ClassName@HashCode
- **If implementation of super class method is partially complete then we should override method in sub class.**
- **The result in toString() method should be a concise but informative that is easy for a person to read.**
- **It is recommended that all subclasses override this method.**

+ Package

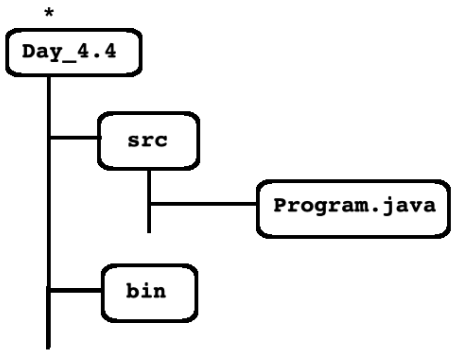
- **package is a java language feature which is used:**
  1. To group functionally related/equivalent types together.
  2. To avoid name collision/ambiguity.
- **package is keyword in java.**
- **If we want to define any type(interface/class) inside package then we should use package declaration statement.**
- **Example:**

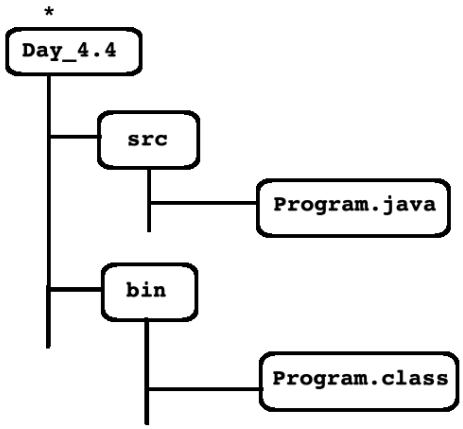
```
package test;
class Program{
  public static void main( String[] args ){
    //TODO
  }
}
```

- **Package declaration statement must be first statement inside .java file.**
- **Example:**

```
package p1; //OK
package p2; //Not OK
class Program{
}
```

- **Package can contains:**
  1. Sub package
  2. Interface
  3. Class
  4. Enum
  5. Exception
  6. Error
  7. Annotation Type



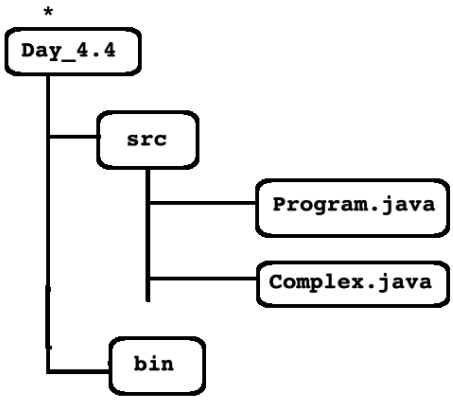


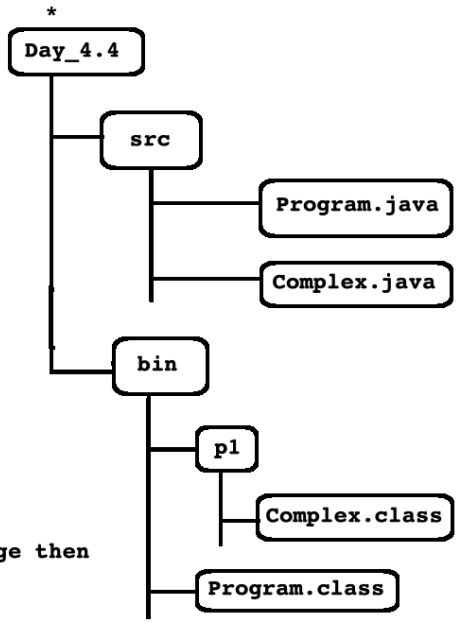

```
export PATH=/usr/bin (optional)
javac -d ./bin/ ./src/Program.java
export CLASSPATH=./bin/
java Program
```

```
set path="c:\Program Files\Java\jdk1.8.0\bin";
javac -d ./bin/ ./src/Program.java
set classpath = ./bin/;
java Program
```

- PATH is os platforms's environment variable which is used to locate java language development tools.

- CLASSPATH is Java platform's environment variable which is used to locate .class file / .jar file.





- Package name is physically mapped to folder.
- If we want to use any type in different package then
  1. Either we should use F.Q.Type name
  2. Or we should use import statement.
- If we define any type inside package then its default access modifier is always package level private.
- According to JLS name of public class and name of .java file must be same.
- Using import statement, we can use any public type inside different package.
- Conclusion : we can use packaged type/class inside unpackaged class/type.

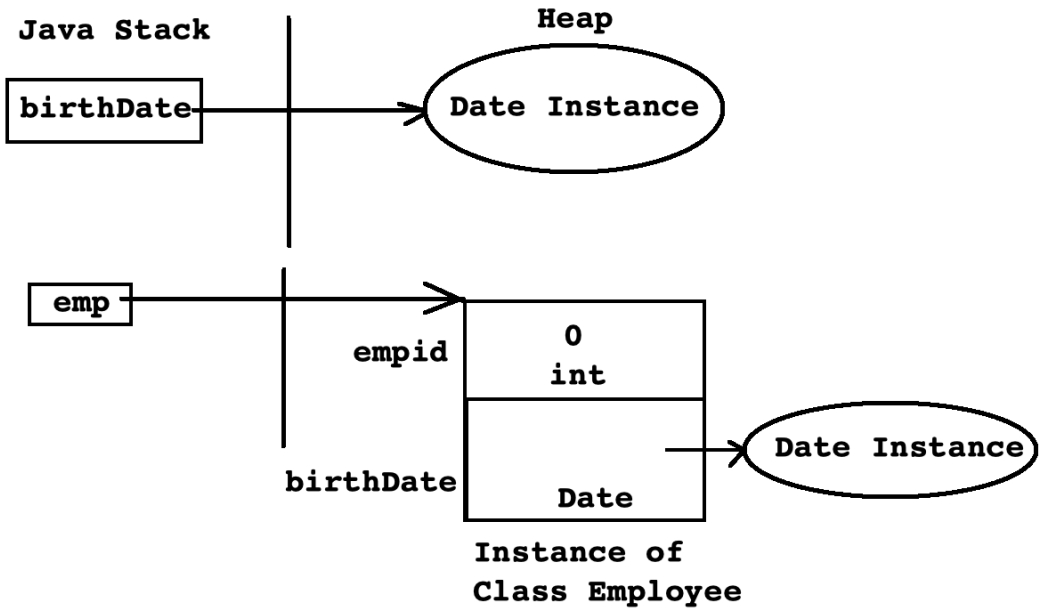
- **If we define any type without package then it is considered as member of default package.**
- **We can not import default package. Hence it is impossible to use unpackaged type inside packaged type.**
- **We can define every type inside different package.**
- **If types belong to the same package then to use it import statement is optional.**
- **java.lang package contains all the fundamental classes of core java.**
- **This package is by default imported in every .java file.**
- **Naming convention for package:**
  1. **com.companyname.projectname**
  2. **org.sunbeam.dac**
  3. **com.mysql.cj.jdbc**

1. Hashcode

- it is a logical integer number.
- that can be generated by using state of an object
-

demo

- reference is always on java stack, instance is always on heap section



1. create a date object, as a date member of Employee class ,so birthDate is a reference

- in C ++ this use of date class as data member of Employee is called Association , i.e tight coupling
- here emp is reference stored on java stack
- instance of Employee containing(empid,bithdate) stored on heap section

```
class Date{
}
class Employee{
    int empid;
    private Date birthDate;
    public Employee() {
        this.birthDate = new Date();
    }
}
public class Program {
    public static void main(String[] args) {
        Date birthDate = new Date();

        Employee emp = new Employee( );
    }
}
```

2. demo best practice to override toString method for object

- to override use annotation over toString method

```
@Override // annotation for overriding method
    public String toString() {


    }
```

- demo to override toString ,generated by using source

```
import java.util.Date;

class Employee{
    private String name;
    private int empid;
    private String department;
    private String designation;
    private float salary;

    // generated by using source
    @Override
    public String toString() {
        return "Employee [name=" + name + ", empid=" + empid + ", salary="
+ salary + "]";
    }


}
public class Program {
    public static void main(String[] args) {
        Employee emp = new Employee("Sandeep", 33, "Training", "Teacher",
25000.45f);
        String str = emp.toString();
        System.out.println(str);
    }

}
```

3. to compile java program from src folder and get .class file in bin folder

- 1. PATH
  - is OS platform environment variable which is used to locate java language development tools.

```
export PATH=/usr/bin   (optional ) -- tu include java development tools
javac -d ./bin/  ./src/Program.java
```

- to execute the .class file in bin
- 2. CLASSPATH
  - is java platform's environment variable which is used to locate .class file /.jarfile

```
// when class file is in other file ,to give its location
   export CLASSPATH=./bin/
   java Program
```

- locate javac

```
whereis is javac
```

4. type in java is referring to

- class, interface
- if we define any type(class) outside a package
- it is member of global package i.e in java called default package
- so we cant access it directly,
  - as we cannot import default package, as it has no name, hence it is impossible to use unpackaged type inside package type.
- demo on default package class

5. Command to be used while dealing with packages in compilation and execution

  - 1. using package p1 class Complex in package p2 class Program file
    - while compiling

```
 -- in Complex.java
 package p1;

  public class Complex{ // Packaged class }

-- in Program.java
   package p2;
   import p1.Complex;
    public class Program{// Packaged class }
```

  - need to use command when in a project package to use is different and needed for compilation
  - command for compilation
    1. javac -d ./bin ./src/Complex.java

2. export PACKAGEPATH=./bin/p1
3. export CLASSPATH=./bin/
4. javac -d ./bin/ ./src/Program.java

- while executing need to use package name . file name

java p2.Program;

- 2. using package p1 class Complex in package p1 class Program file

```
-- in Complex.java
    package p1;

     public class Complex{ // Packaged class }

   -- in Program.java
      package p1;

       public class Program{// Packaged class }
```

- command for compilation

1. javac -d ./bin ./src/Complex.java
2. export CLASSPATH=./bin/
3. javac -d ./bin/ ./src/Program.java

- while executing need to use package name . file name

java p1.Program;

- 3. how to import static method in a java file

```
--- Test class
package test;

 class Test{

public static void print()
{
   System.out.println("hello from test");
}

--- Program class
      package test;
      import static test.Test.print;
       public class Program{
```

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

     System.out.println("Hello ");

        print();
      }
    }
```

- command for compilation
    1. javac -d ./bin ./src/Test.java
    2. export PACKAGEPATH=./bin/test
    3. export CLASSPATH=./bin/
    4. javac -d ./bin/ ./src/Program.java
- while executing need to use package name . file name

```
java test.Program;
```

6. demo for importing static class, static method and variable of package class

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
// import static java.lang.Math.*;
public class Program{

    public static void main(String[] args) {


        float radius = 10;

        float area = (float) (PI * pow(radius,2));

        System.out.println("Area : " + area);

     }
```

- 3. how to import static method in a java file

```
--- Test class
package test;

 class Test{
```

```
public static void print()
{
   System.out.println("hello from test");
}

--- Program class
      package test;
      import static test.Test.print;
       public class Program{

     public static void main(String[] args) {

     System.out.println("Hello ");

        print();
      }
    }
```

- command for compilation
    1. javac -d ./bin ./src/Test.java
    2. export PACKAGEPATH=./bin/test
    3. export CLASSPATH=./bin/
    4. javac -d ./bin/ ./src/Program.java
- while executing need to use package name . file name

```
java test.Program;
```

read

1. can we define multiple classes in single java file?

- NO , as classname and .java file name should be same

2. java.lang.Object

- what are the methods in object ?
- using this command on terminal we can get methods in java Object

```
javap javaObject 'answer'
```

Day5

notes

1. Non static data member declared inside class is called instance variable.

2. Non static member function declared inside a class is called instance method.
3. in java Object is called as instance.

- Ex:

```
Complex c1 = new Complex(); // c: refrence /object reference
```

4. Instance member are designed to access using object reference

- Instance member = {Instance variable ,instance method}

```
class Test{
   int num = 10; // in java and C# we can initializa at time on
   public  void print(){
      System.out.println(num);
   }

   main(){

      Test t1 = null;
      t1 = new Test();
      t1.num = 10;
      t1.print();
   }
}
```

5. Instance variable gets space once per instance according their declaration inside class.
6. to initialize instance variable,we should use constructor,and

- constructor get called once per instance.

7. Static :

- Static data member declared inside class,is called class level variable.
- Static member function declared inside class is called class level method
- class level members are designed to access using class name and dot(.) operator

```
class Test
{
   static int number;
   public static void print()
   {
      System.out.println(Test.number);
   }
}
```

```
main( )
{
Test.number = 10;
Test.print();
}
```

- if we want to share value of field in all the instance of same class then we should declare such field static.
- non static field do not get space inside instance.
- static member get space once per classs during class loading on method area.
- eg. for every instance of class A static member in called only once in first instance , on method area

```
class A{
   static int num;
}
// static variable get called once per class
class B{
   static int num;
}
```

- **in C# static is a constructor (static,)??**
- if we want to initialize static fileds of the class then we should use static initialiser block.

```
static { //static member initiasize
        }
```

```
class Test{
    private int num1;
    private int num2;
    private static int num3;
    static
    {
        System.out.println("Inside static initializer block of class
test");
      Test.num3 = 500;
    }
    public Test( int num1, int num2 ) {
        this.num1 = num1;
        this.num2 = num2;
        System.out.println("Inside test constructor block" );
```

```
    }

    public void print()
    {
        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2 );
        System.out.println("num3 = " +  num3);
    }


}
public class Program {

    static {
        System.out.println("inside static initializer block of program");
    }

    public static void main(String[] args) {

        System.out.println("inside static block of program main");

        Test t1 = new Test( 10, 20 );
        Test t2 = new Test( 30, 40 );
        Test t3 = new Test( 50, 60 );

        t1.print();
        t2.print();
        t3.print();

    }
}
```

this refrence for static and non-static

- non static method are caleed by instance of classm so it gets this refrence/pointer
- static method are called by class name, so it does not this refrence/pointer
- this refrence is link between not static member and nobn static method
- (#) using instance we can refer non static variable in static method
- tatic variable can directly be reffered in static method
- method not containing this refrence ,should be static method

```
public class Program {

    private int num1 = 10;
    private static int num2 = 20;
```

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


         (#) // System.out.println("Num1 : " + num1); // Not Ok

        Program p1 = new Program();
        System.out.println("Num1 : " + p1.num1);



        System.out.println("Num2 : " + num2); // 20



    }
}
```

- 1. when to make static
  - if in method , no need for **this reference** , then declare it static

```
class Test{
    //no need of this reference
    public static int square(int number)
    {
        return number * number;
    }


}
public class Program {

    int number = Test.square(5);

}
```

- 2. cant declare local variable as static variable
  - as static is class level , it cant be local level

```
public class Program {

    private static int count;
    public static void print( ) {
    (#) //int count = 0; // NOT OK

        count = count + 1;
        System.out.println("Count   :   "+count);
```

```
    }
    public static void main(String[] args) {
        Program.print();    //1
        Program.print();    //1
        Program.print();    //1
    }
}
```

- 3.**what is singleton class? write code?**
- create instance of singleton class using reflection ?

```
class Singleton{

    private int number;

    private Singleton()
    {
        this.number = 0;
        System.out.println("Inside constructor");
    }

    public int getNumber()
    {
        return number;
    }

    public void setNumber(int number)
    {
        this.number = number;
    }
    private static Singleton instance = null;
    public static Singleton getInstance() {

        if(instance == null)
        instance = new Singleton();
         return instance;
    }

}

public class Program {


    public static void main(String[] args) {

        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        Singleton s3 = Singleton.getInstance();
```

```
    }
}
```

- 4. final /cosntant
  - read only variable,
  - not mandetory to inilializing , after declaration
  - but once initilize ,cant change the value
  - we can take final value at run time , using scanner, also assign at compile time

```
public static void main1(String[] args) {
        final int number = 10;
        //number = number + 1;
        System.out.println("Number : " + number);
   }

   public static void main(String[] args) {
        final int number;
        number = 10;
        System.out.println("Number : " + number);
   }

   public static void main(String[] args) {
        final int number = 10;//OK
        //number = 20; // Not OK
        System.out.println("Number : " + number);
   }

   public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        final int number = 10;//OK
        //number = 20; // Not OK
        System.out.println("Number : " + number);

        System.out.println("Num2 : ");

        final int number2 = sc.nextInt();

        System.out.println("Number : " + number);
    }
```

- 5. to make a variablee constant , for every instance, and avoid memory loss, by giving static memory is given once

```
class Test{

    final int num1;
```

```
  public static final int num2 = 20;

    public Test()
    {
        this.num1 = 10;
    }
    public void print()
    {
        System.out.println("num1 : " + this.num1);
        System.out.println("num1 : " + this.num2);
    }

}
```

- 6. when final is used at refrence of instance ,
- here refrence is final (c1) , instance is not final
- so qwe cannot declare instance final

```
class Complex{
    private int real;
   private int imag;
}

public static void main(String[] args) {

        // (#)
        final Complex c1 = new Complex(10,20);

        System.out.println(c1.toString());
        c1.setReal(100);
        c1.setImag(200);
        // c1 = new Complex(30,40);
        System.out.println(c1.toString());
    }
```

- 7. refernce type

1. 4 types :
    - 1. Array
        - it is arefence type
        - , in other word to create a array it is neccesary to use new operator
    - 2. class
    - 3. enum
    - 4. Interface

2. Array

- there are three type of Array :
    - 1. Single dimensional array
    - 2. multi dimensional array
    - 3. Ragged array
        - jagged array is not in java
- if we want to process element of array ,then
    - we should use method declared in java.util.Arrays class

3. Single D Array:

- for array of primitive type ,default value of datatype
- for arrray of reference type , default value null
- array refrence , get spack on java stack
- 7. .1 create Single D array in java

```
int arr[]; //OK
        int [] arr2 ; //OK
        //int [arr3]; // Not OK

        int[] arr = null;
        arr = new int[3];//OK

   -- method to use
      int[] arr2 = new int[3];

        //int arr3[] = new int[3];

    int[] arr1 = new int[-3]; // NOT OK // Negative array size exception
```

- printing array, with exception handling for null

```
public static void printArray(int[] arr)
 {
     if(arr != null)
     {
         for (int i = 0; i < arr.length; i++) {

                System.out.println(arr[i]);
            }

     }
```

```
 }

public static void main(String[] args) {

    int[] arr1 = new int[3];

        Program.printArray(arr1);

int[] arr2 = new int[5];

    //  Program.printArray(arr2);

Program.printArray(null);
    }
```

- how to declare array initializer list

```
public static void main(String[] args) {
        //int[] arr = new int[ 3 ] { 10, 20, 30 };  //Not OK // cannot
provide dimension , with array initializer list
      int[] arr2 = new int[ ] { 10, 20, 30 };   //OK
      int[] arr3 = new int[ ]; //Not Ok
        int[] arr3 = new int[ 1];   //OK
        Program.printArray(arr);
    }
```

- how to accept and print array

```
private static void acceptArray(int[] arr) {
        if( arr != null ) {
            for( int index = 0; index < arr.length; ++ index ) {
                System.out.print("arr[ "+index+" ]  :   ");
                arr[ index ] = sc.nextInt();
            }
        }
   }

    private static void printArray(int[] arr) {
        if( arr != null ) {
            for( int index = 0; index < arr.length; ++ index )
                System.out.println(arr[ index ] );
        }
    }

    public static void main7(String[] args) {
        int[] arr = new int[ 3 ];
        Program.acceptArray(arr);
        Program.printArray(arr);
```

```
    }
```

- how to use methods to manipulate array

```
int[] arr = new int[] {50,10,20,30,40};
        Program.printArray(arr);

        Arrays.sort(arr);
      Program.printArray(arr);
```

- how to foreach loop in java
    - forward only and read only loop

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

            int[] arr = new int[] {50,10,20,30,40};
            // for each n java
            for(int element : arr)  // forward only and read only loop
            {
            System.out.println(element);
         }



        }
```

- how to call toString on array

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

            int[] arr = new int[] {50,10,20,30,40};
            // for each n java
        System.out.println(arr.toString()); // output: [I@677327b6 // NOT
OK

        System.out.println(Arrays.toString(arr)); //[50, 10, 20, 30, 40]
//OK

        }
```

- 8. Multi Dimensional Array [MDA]

- how to declare MDA

```
       int arr[] []; //OK
        int[] arr1[]; //ok
        int [] [] arr3; //ok
        int[][] arr4 = null; //ok
      arr4 = new int[4][3];//ok
```

- how to accept and print MDA

```
static Scanner sc = new Scanner(System.in);

    private static void acceptArray(int[][] arr) {
        if( arr != null ) {
            for( int row = 0; row < arr.length; ++ row ) {
                for( int col = 0; col < arr[ row ].length; ++ col ) {
                    System.out.print("arr[ "+row+" ][ "+col+" ] :   ");
                    arr[ row ][ col ] = sc.nextInt();
                }
            }
        }
    }
    private static void printArray(int[][] arr) {
        if( arr != null ) {
            for( int row = 0; row < arr.length; ++ row ) {
                for( int col = 0; col < arr[ row ].length; ++ col ) {
                    System.out.print(arr[ row ][ col ]+"    ");
                }
                System.out.println();
            }
        }
   }

   public static void main2(String[] args) {

    int [][] arr = new int[4][3];

    Program.acceptArray(arr);

    Program.printArray(arr);

    }
```

- how to initialize MDA, using array initializing list

```
    public static void main3(String[] args) {

        int [][] arr = new int[][] {{1,2,3},{4,5,6},{7,8,9}};
        int[][] arr2 = {{1,2,3},{4,5,6},{7,8,9}};
        //int [][] arr = new int[4][3];

        Program.acceptArray(arr);

        Program.printArray(arr);

        }
```

- how to use toString method on MDA

```
public static void main4(String[] args) {

        int [][] arr = new int[][] {{1,2,3},{4,5,6},{7,8,9}};
        for (int row = 0; row < arr.length; row++) {

            System.err.println(Arrays.toString(arr[row]));
        }

        }
```

- 9. Ragged array (for java) (in C# jagged array)
  - it is array of arrays such that member arrays can be of different sizes, i.e.,
    - we can create a 2-D arrays but with variable number of columns in each row.
    - These type of arrays are also known as Jagged arrays.
    - A Jagged or also called Ragged array is a n-dimensional array that need not the be reactangular
- how to declare a ragged

```
public static void main(String[] args) {
        int [][] arr = new int[4][];

        arr[0] = new int[] {1,2,3};
        arr[1] = new int[] {4,5};
        arr[2] = new int[] {6,7,8,9};
        arr[3] = new int[] {10};
}
```

- how to initialize ragged array using array inilializer list

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

        int [][] arr = new int[4][];

        arr[0] = new int[] {1,2,3};
        arr[1] = new int[] {4,5};
        arr[2] = new int[] {6,7,8,9};
        arr[3] = new int[] {10};



}
```

- 10. array of primitive type
- how to

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

        double[] arr = new double[3]; //array of primitive type

        System.out.println(Arrays.toString(arr));//
    }
public static void main2(String[] args) {

        int[] arr = new int[3]; //array of primitive type

        System.out.println(Arrays.toString(arr));//
    }
    public static void main1(String[] args) {


        boolean[] arr = new boolean[3]; //array of primitive type

        System.out.println(Arrays.toString(arr));//[false, false, false]



    }
```

- 11. array of non-primitive/reference type
- how to declare array of referencssa
    - here it is array of referneces ,as out shows

```
class Complex{}

public static void main(String[] args) {

        Complex[] complex = new Complex[3];

        System.out.println(Arrays.toString(complex));//[null, null, null]


    }
```

- how to declare array of instances/object

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

        Complex[] complex = new Complex[3];

        for (int i = 0; i < complex.length; i++) {
            complex[i] = new Complex(); //aray of objects
        }

        System.out.println(Arrays.toString(complex));//[Complex [real=10,
imag=20], Complex [real=10, imag=20], Complex [real=10, imag=20]]
    }
```

- 12. ways to pass value to function
  - 1. C - 2 ways, value/address
  - 2. C++ - 3 ways , value/address/reference
  - 3. C# - using unsafe keyword (i.e pointer can be used to pass to function as argument)
  - 4. Java -
    - no concept of pass by reference
    - each argument is passed to function by value in java
    - so how to pass by reference is if required ? e.g (for swap function)
    - answer: can be done using array, it will update values in array , of varailbe we want to pass by reference

```
private static void swap(int a,int b) //NOT WORKING
    {
        int temp = a;
        a = b;

        b = temp;
    }
    private static void swap(int[] arr ) // WORKING
    {
```

```
        int temp =arr[0];
        arr[0] = arr[1];

        arr[1] = temp;
    }
   public static void main(String[] args)// WORKING
    {

        int a = 10;
        int b = 20;

        int[] arr = new int[] {a,b};
        Program.swap(arr);

        a = arr[0];
        b = arr[1];
        System.out.println("a :" +  a);
        System.out.println("b :" +  b);

        }
public static void main1(String[] args) //NOT WORKING
 {

    int a = 10;
    int b = 20;
    //Program.swap(a, b);

    System.out.println("a :" +  a);
    System.out.println("b :" +  b);

    }
```

Slides

- Non static data member declared inside class is called instance variable.
- Non static member function declared inside class is called instance method.
- In java Object is called as instance.
- Example : Complex c1 = new Complex(); //c1 : reference / object reference
- Instance members are designed to access using object reference.

```
class Test{
  int number;
  public void print( ){
    System.out.println( number );
  }
}

  Test t1 = null;
  t1 = new Test();

  t1.number = 10;
  t1.print( );
```

- Instance members = { instance variable, instance method }
- Instance variable get space once per instance according their declaration inside class.
- To initialise instance variable we should use constructor.
- Constructor gets called once per instance.

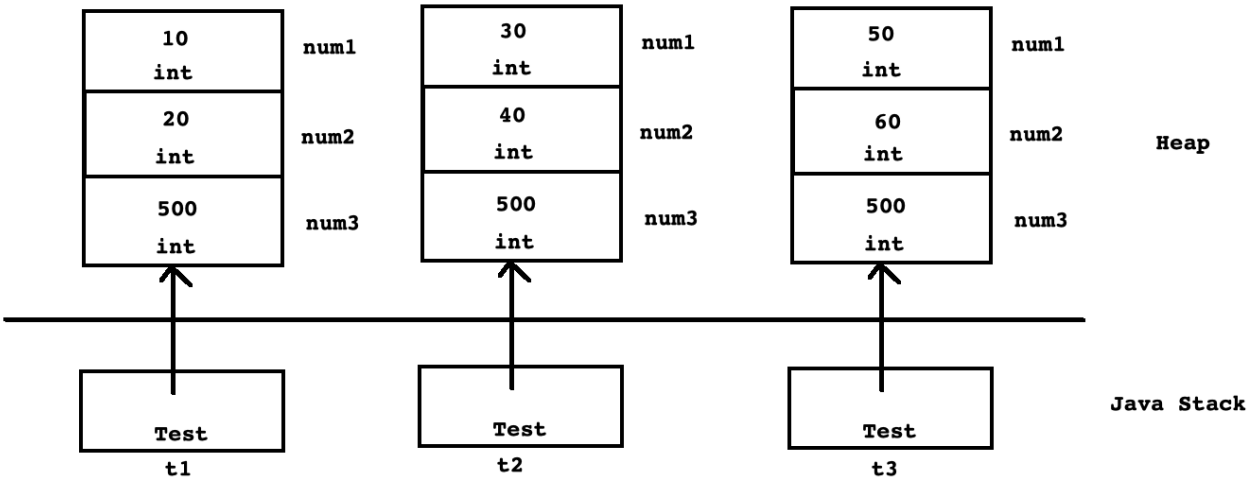
- Static data member declared inside class is called class level variable.
- Static member function declared inside class is called class level method.
- class level members are designed to access using classname and dot operator.

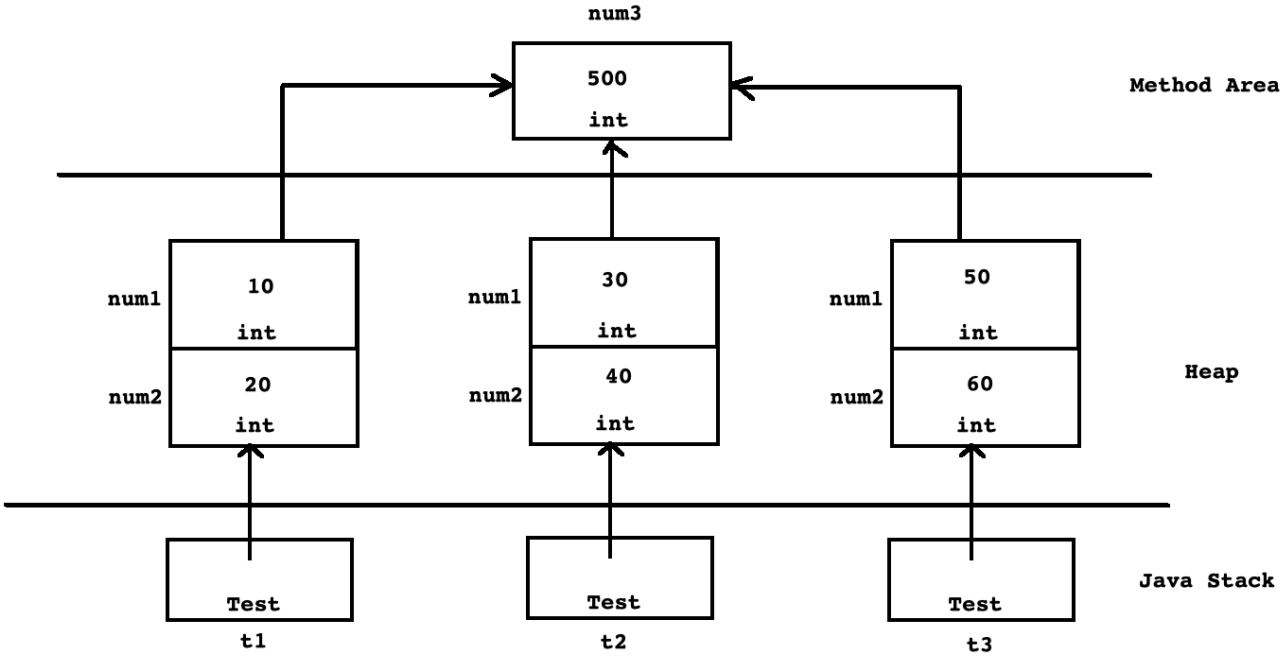
```
class Test{
  static int number;
  public static void print( ){
    System.out.println(Test.number);
  }
}

Test.number = 10;
Test.print( );
```

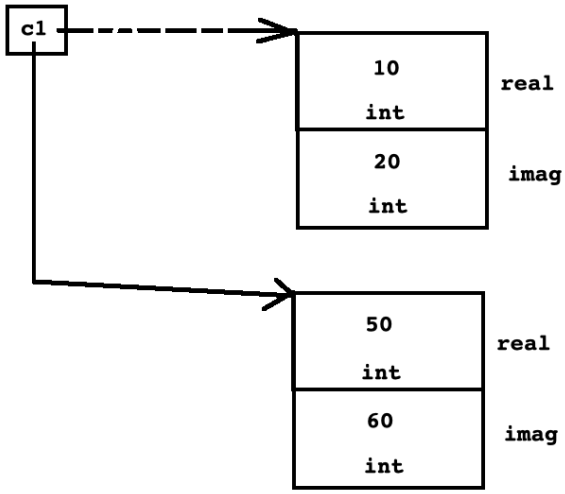
- If we want share value of any field in all the the instances of same class then we should declare such field static.
- static field do not get space inside instance.
- Static field get space once per class during class loading on method area.

```
class A{
  static int number;
}

class B{
  static int number;
}
```

- If we want to initialize static fields of the class then we should use static initialiser block.

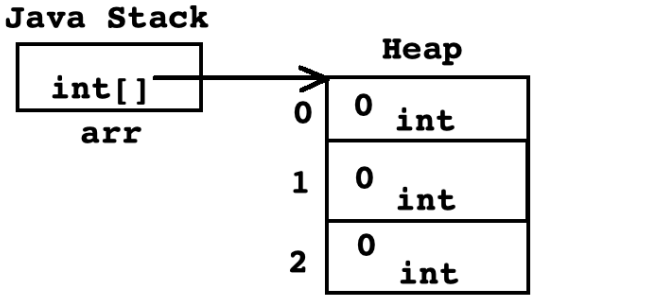

10
int
num1
20
int
num2
500
int
num3
30
int
num1
40
int
num2
500
int
num3
50
int
num1
60
int
num2
500
int
num3
Heap
Test
t1
Test
t2
Test
t3
Java Stack


num3
500
int
Method Area
num1
10
int
num2
20
int
num1
30
int
num2
40
int
num1
50
int
num2
60
int
Heap
Test
t1
Test
t2
Test
t3
Java Stack

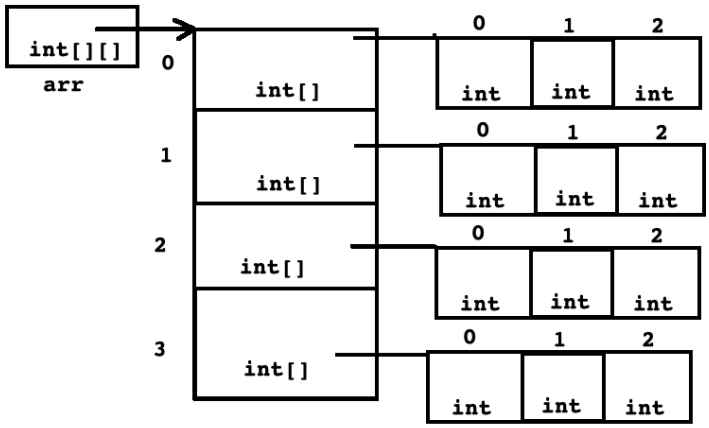


+ Array
  - It is reference type. In other words, to create instance of array it is necessary to use new operator.
  - Three Types of Array:
    1. Single dimensional array
    2. Multi dimensional array
    3. Ragged array
  - Array bounds checking is a job of JVM.
  - If we want to process elements of array then we should use methods declared in java.util.Arrays class.

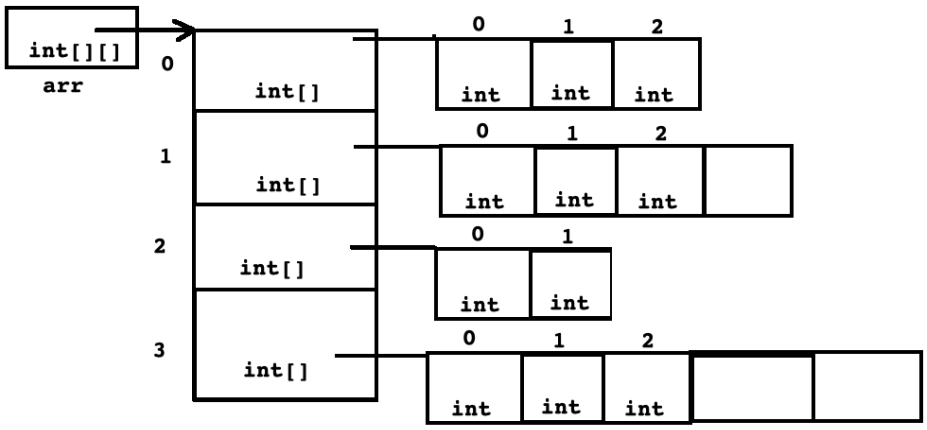
+ Single dimensional array

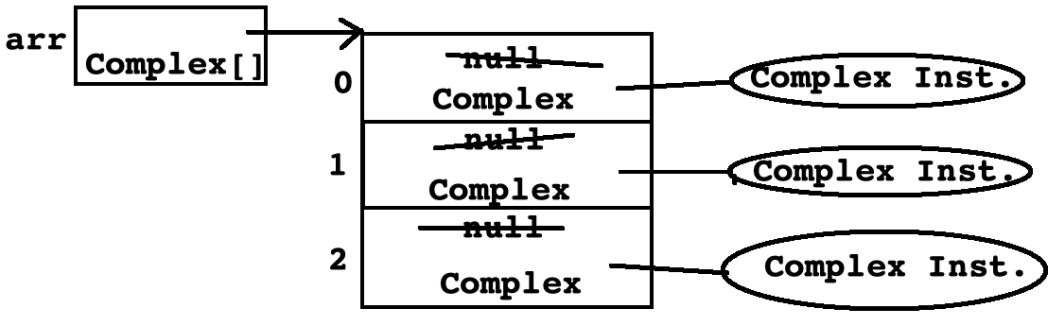


Single dimensional
Array of integer



Multi Dimensional Array






```
Complex *ptr = new Complex[ 3 ]; //C++

Complex[] ptr = new Complex[ 3 ]; //Java
 for( int index = 0; index < 3; ++ index )
    arr[ index ] = new Complex();
```

READ

1. **what is singleton class? write code?**

- create instance of singleton class using reflection ?

2. for sorting The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch

Day6

to read

1. Hierarchy
2. assembly info.css file contains attribute on assembly level in .NET
3. calling method on null object , gives nullpointer exception
4. classpath/runtime class path/build path

notes

+ Variable arity method / variable argument method

  - Example(C)
    1. int printf(char *format, ...);

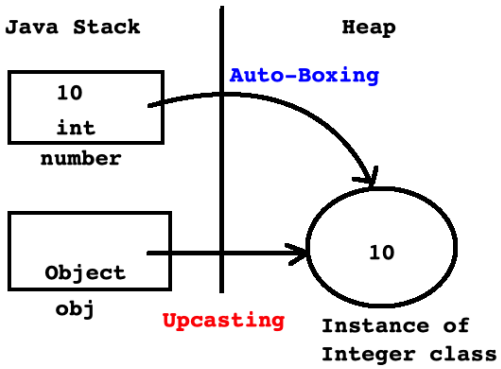
  - Example(Java)
    1. public PrintStream printf(String format, Object... args);

    2. public static String format(String format, Object... args);

    3. public Object invoke(Object obj, Object... args);

```
public static void sum( int... arguments ) {
      int result = 0;
      for( int element : arguments )
        result = result + element;
      System.out.println("Result  :  "+result);
}
```

- **Process of converting state of variable of primitive/value type into non primitive/ reference type is called as Boxing.**
- **Example:**
  int number = 10;
  String str = String.valueOf(number); //Boxing
- **If boxing is done implicitly then it is called auto-boxing.**
- **Example:**
  int number = 10;
  Object obj = number; //Auto-Boxing
  //Object obj = Integer.valueOf(number);



- Process of converting value of variable of non primitive/reference type into primitive/ value type is called unboxing.
- Example:

```
String str = "125";
int number = Integer.parseInt( str ); //Unboxing
```

- If unboxing is done implicitly then it is called as auto-unboxing.
- Example:

```
Integer n1 = new Integer("125");
int n2 = n1.intValue( ); //UnBoxing

int n3 = n1; //Auto-UnBoxing
//int n3 = n1.intValue();
```

+ Enum
  - If we want to give name to constant/literals then we should use enum.
  - Using enum, we can increase code readability.
  - Syntax(C/C++):

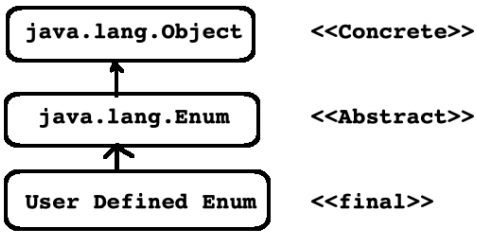
```
enum Color
{
  RED, GREEN, BLUE  //Enum constant/enumerators
  //RED=0, GREEN=1, BLUE = 2
};
enum Day
{
  SUN=1, MON=2, TUES, WED, THURS, FRI, SAT
};
```

  - Enum is reference type in java.
  - java.lang.Enum is abstract class which is considered as super class of all the enums in Java.
  - If define enum in java
    Then it is internally considered as
    Final class. Hence we can not extend enum
  - In Java, Using enum, we can give
    Name to any literal or group of litreals.
  - Example

```
enum Day{
  SUN(1),MON(2)
}
```

```
enum Day{
  SUN("SunDay"),MON("MonDay")
}
```

```
enum Day{
    SUN(1,"SunDay"),MON(2,"MonDay")
}
```



- Consider enum in Java:

```
enum Color{
    RED, GREEN, BLUE  //Enum constants
}
```

- Internal Implementation:

```
final class Color extends Enum<Color> {
    public static final Color RED;
    public static final Color GREEN;
    public static final Color BLUE;

    public static Color[] values();
    public static Color valueOf(java.lang.String);
}
```

- Methods of java.lang.Enum
  1. public final Class<E> getDeclaringClass()
  2. public final String name() //returns name of enum constant.
  3. public final int ordinal()
  4. public static
     <T extends Enum<T>>
     T valueOf(Class<T> enumType, String name)
- values() and valueOf() methods get added into enum at compile time.
- In java, ordinal of enum constant always starts with 0. We can not change ordinal.
- Example:

```
Color.GREEN.name( );      – will return – GREEN
Color.GREEN.ordinal( );   – will return – 1
```

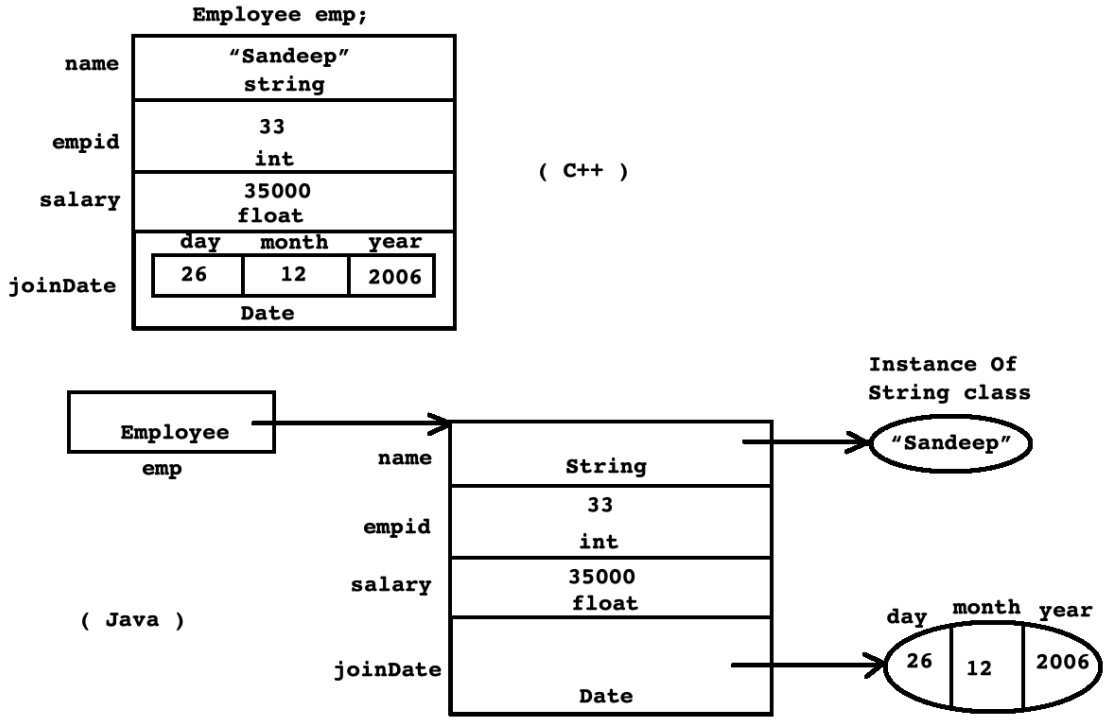
+ Hierarchy
  - Level of abstraction is called hierarchy.
  - Types:
    1. Has-a : Association
    2. Is-a : Inheritance
    3. Use-a : Dependancy
    4. Creates-a : Instantiation

+ Association
  - If “has-a” relationship is exist between 2 types then we should use association.
  - Example:
    * Employee has a joinDate.

```
class Employee
{
  string name;  //Association
  int empid;
  float salary;
  Date joinDate; //Association
};
Date joinDate(26,12,2006)
Employee emp(“Sandeep”,33,35000,joinDate);
```

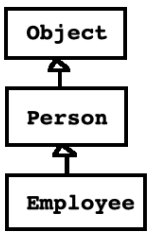
```
class Employee{
  String name; //Association
  int empid;
  float salary;
  Date joinDate; //Association
};
Employee emp = null;
Date joinDate = new Date(26,12,2006);
emp = new Employee(“Sandeep”,33,35000,joinDate);
```

- If we declare object/instance of a class as a data member/field inside another class then it represents association.



- In Java, Instance can contain object reference but instance can not contain another instance.

+ Inheritance
  - It is a process of acquiring properties and behavior of super class by the sub class.
  - W/O modifying implementation of existing class, if we want to extend meaning of that class then we should use inheritance.
  - To extend the class we should use extends keyword.
  - Using super keyword, we can use members of super class inside method of sub class.
  - Using super statement, we can call constructor of super class from constructor of sub class.
  - In Java, any class can extend only one class.
  - Example : Employee is a Person.
  - In other words, Java do not support multi class inheritance.
  - Except Constructor, all the members of super class of any access modifier inherit into sub class.
  - Super statement must be first statement inside constructor body.



  - Process of converting reference of sub class into reference of super class is called upcasting.

```
Employee emp = new Employee();
//Person p = (Person)emp; //OK : Upcasting
Person p = emp; //OK : Upcasting
```

  - Process of converting, reference of super class into reference of sub class is called down casting.

myNotes

1. Hierarchy

- level of abstraction is called hierarchy
- types of hierarchy:

1. Is-a : Inheritence
2. Has-a : Association
3. Use - a : Dependency
4. Creates-a : Instantiation

-

2. Association

- if "has-a" relation between two class, then use Association
- here reference of class ,get memory in object instance
- asso: object part of another object
    - physically only reference to associated object is there in objects instance, so it is
    - object outside object.
    - so if we want to ignore the asso object ,set it as null
- so,in java Instance can oontain object reference, but instance cannot contain another instance

3. Java ARchive (JAR)

- to create a lib

4. inheritence(extends)

- it is a process of acquiring properties and behaviour of super class by the sub class.
- when to use inheritence
    1. without modifying implementation of existing class,if we want to extend meaning of that class then ,use
- to extend the class we should use extends keyword.
- using super statement we can use members of superclass inside method of sub class.
- Using super statement, we can call constructor of super class from constructor of sub class.
- in java any class can extend only one class.
- for every class is extended from Object
- java do not support multi class inheritence
- Except Constructor, all members of super class of any access modifier inherit into sub class.
- super statement must be first statement inside constructor body
- Example : Employee is a Person

```
class Person extends Object{}
class Employee extends Person{}
```

5. upcasting : process of converting of reference of sub class to reference of super class

- it helps us to reduce maintenance of system

6.

demo

1. How to declare dynamic arguments (to get multiple values as argument)

```
public static void sum( int... arguments ) {
        int result = 0;
        for( int element : arguments )
            result = result + element;
        System.out.println("Result  :   "+result);
    }
    public static void main2(String[] args) {
        Program.sum( );
        Program.sum( 0 );
        Program.sum( 10, 20 );
        Program.sum( 10, 20, 30, 40, 50  );
        Program.sum( 10, 20, 30, 40, 50, 60, 70, 80  );
    }
```

2. Auto unboxing

```
public static void main(String[] args) {
        Integer n1 = new Integer("125");
        //int n2 = n1.intValue();   //UnBoxing

        int n3 = n1;    //Auto-UnBoxing
        //int n3 = n1.intValue();
        System.out.println(n3);
    }
```

3. Enum decalaration, calling method using Enum name, enum object, use of name,ordinal,valueOF methods

```
enum Color{
    RED, GREEN, BLUE
}

static Scanner sc = new Scanner(System.in);

// using scanner to print ordinal value
    public static void main(String[] args) {
        System.out.print("Name  :   ");
        String str = sc.nextLine(); //RED,GREEN, BLUE

        Color color = Color.valueOf(str.toUpperCase());
        String name = color.name();
        int ordinal = color.ordinal();

        System.out.printf("%-10s%-5d\n", name, ordinal);
```

```
    }
    //using values method to instantiate enum array
    public static void main5(String[] args) {
        Color[] colors = Color.values();
        for (Color color : colors) {
            String name = color.name();
            int ordinal = color.ordinal();
            System.out.printf("%-10s%-5d\n", name, ordinal);
        }
    }
// using instance of enum to call name,ordinal method
    public static void main4(String[] args) {
        //Color color = Color.RED;
        //Color color = Color.GREEN;
        Color color = Color.BLUE;
        String name = color.name();
        int ordinal = color.ordinal();
        System.out.printf("%-10s%-5d\n", name, ordinal);
    }

    // using className to call name,ordinal method
    public static void main1(String[] args) {
        String name = Color.RED.name();
        int ordinal = Color.RED.ordinal();
        System.out.printf("%-10s%-5d\n", name, ordinal);
    }
```

4. Enum constructor, getter,setter,toString

```
enum Day{
    //SUN,MON,TUES; //OK
    SUN(1,"Sunday"),MON(2,"Monday"),TUE(3,"Tuesday"); //Hint : consider
these are objects like C++
private int dayNumber;
private String dayName;

    /* individual constructor
     * private Day(int dayNumber) { this.dayNumber = dayNumber; }
     *  private Day(String dayName) { this.dayName = dayName; }
     */
private Day(int dayNumber,String DayName)
{
    this.dayNumber = dayNumber;
    this.dayName = DayName;
}

public int getDayNumber() {return dayNumber;}

public void setDayNumber(int dayNumber) {
    this.dayNumber = dayNumber;
```

```
}


public String getDayName() {return dayName;}



public void setDayName(String dayName) {this.dayName = dayName;}

@Override
public String toString()
{
    return String.format("%-10s%-5d", this.dayName,this.dayNumber);
}

}

public static void main(String[] args) {
        Day[] days = Day.values();
        for (Day day : days) {
            String name = day.name();
            int ordinal = day.ordinal();
            System.out.println(name+"   "+ordinal+" "+day.toString());
        }
    }
```

5. using methods of java.util class

- 1. Calender

```
public static void main1(String[] args) {
        Calendar c = Calendar.getInstance();

        int day = c.get(Calendar.DATE);
        int month = c.get(Calendar.MONTH)  + 1;
        int year = c.get(Calendar.YEAR);

        System.out.println(day + "/" + month + "/" + year);

        }
public static void main2(String[] args) {
        // to find system date

        Calendar c = Calendar.getInstance();

        //Date time = c.getTime();

        int hour = c.get(Calendar.HOUR);
        int minute = c.get(Calendar.MINUTE);
        int second = c.get(Calendar.SECOND);
        int millisecond = c.get(Calendar.MILLISECOND);
```

```
        System.out.println(hour + ":" + minute + ":" + second + ":" +
millisecond);

        }
```

- 2. java.time package classes localtime,localdate
  - java.Date class is depricated

```
public static void main(String[] args) {
        // to find system date

        LocalTime lt = LocalTime.now();

        //Date time = c.getTime();

        int hour = lt.getHour();
        int minute = lt.getMinute();
        int second = lt.getSecond();
        int nanosecond = lt.getNano();


        System.out.println(hour + ":" + minute + ":" + second + ":" +
nanosecond);

        }
    public static void main3(String[] args) {
        // to find local date

        LocalDate ld = LocalDate.now();

        int day = ld.getDayOfMonth();
        int month = ld.getMonthValue();
        int year = ld.getYear();

        System.out.println(day + "/" + month + "/" + year);



        }
```

- 3. java.util.Date class demo, but mostly depricated, not to use

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

```
        Date date = new Date();

        int day = date.getDay();
        int month = date.getMonth();
        int year = date.getYear();

        System.out.println(day + "/" + month + "/" + year);



        }
```

6. **inheritence in java, using super keyword**

- default mode of inheritence is public
- in java, except constructor,all member(static,not static,private) of superclass inherit in sub class
- using super keyword we can access member of super class in method of sub class

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

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void printRecord()
    {
        System.out.println("Name: " + this.name);
        System.out.println("Age : " + this.age);
    }
}

class Employee extends Person{


    private int empid;
    private float salary;

    public Employee() {
        this.empid = 0;
        this.salary = 0;
        System.out.println("Employeee cont");
    }

    public Employee(String name,int age,int empid,float salary) {
         // in c++ , constructor base initializing list
        super(name,age);
        this.empid = empid;
        this.salary = salary;
```

```
        System.out.println("Employeee cont");
    }

    public void printRecord()
    {// using super keyword we can access member of super class in method
of sub class
        super.printRecord();

        System.out.println("emp id: " + this.empid);
        System.out.println("Salary : " + this.salary);

    }

    public static void main(String[] args) {

        Employee emp = new Employee("suraj",26,1,23);
        emp.printRecord();
        // output Name: suraj Age : 26 empid  : 1Salary : 23.0
       // ie.base class method also implemented


        }
}
```

7. **upcasting and downcasting in java**

- upcasting : process of asigning/converting of reference of sub class to reference of super class
- Downcasting : process of converting reference of super class into reference of sub class is called
- Exception :**ClassCastException**: Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance

```
public static void main1(String[] args) {
        Employee emp = new Employee("suraj",26,1,23;
        emp.printRecord();
        Employee emp2 = emp; //Shallow copy
        }

public static void main2(String[] args) {
        Employee emp = new Employee("suraj",26,1,23;
        Person p = emp;  //OK //UPcasting
        }

    public static void main4(String[] args) {
        Person p1 = null;

        Employee emp = (Employee) p1; //downcasting
        System.out.println(p1);
        System.out.println(emp);
```

```
            }

    public static void main5(String[] args) {
        Person p1 = new Employee();// upcasting
        Employee emp = (Employee) p1;//downcasting
        System.out.println(p1);
        System.out.println(emp);
            }
public static void main(String[] args) {
Person p1 = new Person();
Employee emp = (Employee) p1;
// ClassCastException
    }
```

8. How to

- runtime polymorphism in c++, in java we have
- **Dynamic method dispatch**
  - it is a process of calling method of sub class using reference of super class
- it works on method printRecord in demo, showing
  - in java, method are by default virtual

```
class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public void printRecord( ) {
        System.out.println("Name    :   "+this.name;
        System.out.println("Age :   "+this.age);
    }
}
class Employee extends Person{
    private int empid;
    private float salary;
    public Employee(String name, int age, int empid, float salary){
        super( name, age );
        this.empid = empid;
        this.salary = salary;
    }
    public void printRecord( ) {
        super.printRecord();
        System.out.println("Empid   :   "+this.empid);
        System.out.println("Salary  :   "+this.salary);
    }
}
```

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

        // Dynamic method dispatch
        // in java, method are by default virtual
        Person p = new Employee(); //upcasting

        p.printRecord();
    }
```

9. How to make a jar(Java ARchieve) file

- make a project ,and write your code in package class, mostly methofs,no main method
- right click on project
    - go to export ---> java---> Jar file(select it)
    - give destination folder for jar file , done

10. How to refer/import a jar file from a folder

- right click on project in which you want to refer the jar file
    - click on build path ---> Add external jar files , done

Day7

to read

1. difference between error and exception
2. types of exception , diff between , designed for java compiler, for jVM all re just exception .
3. stack trace: message we get after exception
4. DRY = Do not Repeat Yourself

- 1. DRY stand for "Don't Repeat Yourself,"
- a basic principle of software development aimed at reducing repetition of information.
- The DRY principle is stated as, "Every piece of knowledge or logic must have a single, unambiguous representation within a system."
- 2. how to achieve it
- divide your system into pieces.
- Divide your code and logic into smaller reusable units and use that code by calling it where you want.
- Don't write lengthy methods, but divide logic and try to use the existing piece in your method.

5. KISS

- 1. KISS: Keep It Simple, Stupid
- 2. The KISS principle is descriptive to keep the code simple and clear, making it easy to understand.
- Keep your methods small.
- Each method should never be more than 40-50 lines.
- Each method should only solve one small problem, not many use cases.
- If you have a lot of conditions in the method, break these out into smaller methods.

- It will not only be easier to read and maintain, but it can help find bugs a lot faster.
- 3. how to achieve it
- Think of many solutions for your problem, then choose the best, simplest one and transform that into your code.
- Whenever you find lengthy code, divide that into multiple methods
- — right-click and refactor in the editor. Try to write small blocks of code that do a single task.

6. page 8 and 9
7. **Assertion** in java docs

> https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

- 1. An assertion is a statement in the Java programming language that enables you to test your assumptions about your program.
- 2. Each assertion contains a boolean expression that you believe will be true when the assertion executes.
  - If it is not true,
  - the system will throw an error.
  - By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.
- 3. benifits of using Assertion:

1. it is one of the quickest and most effective ways to detect and correct bugs.
2. assertions serve to document the inner workings of your program, enhancing maintainability.

- syntax
- the assertion statement has two forms :
- where,
  - Expression1 is a boolean expression.
  - Expression2 is an expression that has a value.
- When the system runs the assertion, it evaluates Expression1 and if it is false throws an AssertionError with no detail message.

```
assert Expression1 ;

assert Expression1 : Expression2 ;
```

tommorow

1. final,abstract,instance,generics
2. c++ template, c# generics

notes

1. any class that implements AutoClossable interface ,

- class is called resource type
- class instance is called Resource
- Non Java resource /OS resources(unmanaged resources)
    - 1. File
    - 2. Thread
    - 3. Socket
    - 4. Network Connection
    - 5. IO devices
- if we want to manage os resources carefully in java application then we should use exception handling
- if we want to handle exception , then we should use 5 keywords:

1. try
2. catch
3. throw
4. throws
5. finally
6. what is Exception?

- Runtime error is called as exception
- **Exception is an instance** which is used to send notification to the end user of the system if any exceptional condition occurs in the program .

3.
```
  java.lang.Object(class)
       |
java .lang.Throwable  (class)
   |                  |
```

(class) java.lang.Error java.lang.Exception (class)

4. Exception Handing in java

- The Throwable class is the superclass of all errors and exceptions in the Java language.
- Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.
- Similarly, only this class or one of its subclasses can be the argument type in a catch clause.

5. Error

- it is an class defined in java.lang package.
- it is a subclass of java.lang.Throwable that indicates serious problems that a reasonable application should not try to catch.
- Most such errors are abnormal conditions.

- Example :
- VirtualMachineError
    - Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.
- OutOfMemoryError
    - Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.
- StackOverflowError
- Thrown when a stack overflow occurs because an application recurses too deeply.
- ERROR

1. we can write try-catch block to handle errors but we should avoid it.
2. we can not recover from error hence we should not write try-catch block
3. generated by environmental condition

EXCEPTION

1. it is a class declared in java.lang package
2. it is sub class of java.lang.Throwable class
3. Exception gets generated due to applications
4. Example

- 1. NumerFormatException
- 2. NullPointerException
- 3. ArrayIndexOutOfBoundException

5. we can recover from exception, by handling it

- to handle exception , we should use try=catch block

6. there are two types of exception ,designed for java compiler, not jVM:

- 1. Checked
- 2. Unchecked
- JVM do not distinguish between checked and unchecked exception.these types are designed for java compiler.

7. Unchecked Exception

- java.lang.RuntimeException and all its sub classes are considered as unchecked exception
- for this, handling unchecked exception is not mandetory.(its optional)
- Example :
- 1. NumberFormatException
- 2. NullPointerException
- 3. ArrayStoreException

- 4. ClassCastException
- 5. IllegalArgumentException

8. Checked Exception

- java.lang.Exception and all its subclasses except java.lang.RuntimeException and all its sub classses are called as checked exception.
- it is mandetory to handle checked exception , or compiler gives compile time error
- 3. Example
- 1. CloneNotSupportedException
- 2. InterruptedException
- 3. CLassNotFoundException

Demo

- throw statement in Java

```
public static void main(String[] args) {
        // Throwable();
    Throwable th1 = new Throwable();
    //Throwable(String message)
    Throwable th2 = new Throwable("this is exception demo");
    //  Throwable(String message, Throwable cause)
    Throwable cause = new Throwable("this is exception");
    Throwable th3 = new Throwable(cause);

    //  Throwable(String message, Throwable cause, boolean
enableSuppression,
    Throwable cause2 = new Throwable();
    Throwable th4 = new Throwable("this is exception",cause);

    }
    public static void main1(String[] args) {

        Test t = new Test();
        //throw t; not ok
    }
```

9. Try

- it is a kryword in java.
- if we want to keep watch on statement then we should use try block/handler.
- we cannot define try block after catch/finally block.
- in java try block must have atleast one
  - catch block
  - finally block
  - resource

```
try{

// try catch
}catch(NullPointerException ex){
ex.printStackTrace();
}

//try finally
---
try{

}finally {

}



//try resource syntax
---
try(Scanner sc = new Scanner(System.in)){


}
```

10. Catch

- it is a keyword in java.
- tf we want to handle exception then we should use catch block/handler.
- we should define catch block after try and before finally block
- Single try block may have multiple catch blocks.
- catch block can handle exception thrown from try block only .
- During arithmetic operation if any exception situation occurs then JVM throws ArithmeticException

```
        try {
        System.out.println("Num 1 : ");
        int num1 = sc.nextInt();

        System.out.println("Num 2 : ");
        int num2 = sc.nextInt();


            int result = num1/num2 ;
```

```
            System.out.println("Result : " + result);
        }
        catch(ArithmeticException ex)
        {

            ex.printStackTrace();
        }
        catch(InputMismatchException ex)
        {
            System.out.println("InputMisMatchException");
            ex.printStackTrace();
        }
```

- multi catch block
- it can handle multiple specific exception

```
try {
        System.out.println("Num 1 : ");
        int num1 = sc.nextInt();

        System.out.println("Num 2 : ");
        int num2 = sc.nextInt();


            int result = num1/num2 ;

            System.out.println("Result : " + result);
        }
        catch(ArithmeticException | InputMismatchException ex)
        { // Multi catch block

            ex.printStackTrace();
        }
```

- EXCEPTION class
- reference of java.lang.Exception class can contain reference of instance of any checked as well as unckecked exception, hence to write generic catch block you should use Exception class

```


// generic catch syntax

try{

} catch(Exception ex)
{ex.printStacktrace();
}
```

```
// reason for use of exception
public static void main(String[] args) {

        InterruptedException ex1 = new InterruptedException("Interrupted");

        Exception ex2 = new InterruptedException("Interrupted"); //
upcasting

       Exception ex3 = new NullPointerException("null pointer exc"); //
upcasting
        }
    public static void main1(String[] args) {

    NullPointerException ex1 = new NullPointerException("Null pointer
Exception");

   RuntimeException ex2 = new NullPointerException("null pointer exc"); //
upcasting

   Exception ex3 = new NullPointerException("null pointer exc"); //
upcasting
    }
```

- CATCH block sequence, in case of inherited exception
- if inheritence exist between exception types then it is mandetory to handle all sub type exception first.

```
   try {
Scanner sc =new Scanner(System.in);

    System.out.println("Num1  :");
    int num1 = sc.nextInt();
    System.out.println("Num2  :");
    int num2 = sc.nextInt();


    int result = num1 / num2;
   }
   catch(ArithmeticException ex)
   {
    System.out.println(" arithmetic Exception");
   }
   catch(RuntimeException ex)
   {
    System.out.println("Runtimee Exception");
   }

   catch(Exception ex)
```

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

- so we can use generic catch block for all exception

```
        try {
     Scanner sc =new Scanner(System.in);

            System.out.println("Num1  :");
            int num1 = sc.nextInt();
            System.out.println("Num2  :");
            int num2 = sc.nextInt();


            int result = num1 / num2;
        }
        catch(Exception   ex)
        {
         System.out.println(" arithmetic Exception");
        }
```

11. throw

- it is a keyword in java
- to generate new exception, we must use throw keyword
- Using throw keyword,we can throw instance of sub class of throwable only.
- throw statement is a jump statement, as it takes control to catch block

```
Scanner sc =new Scanner(System.in);
        try {
            System.out.println("Num1  :");
            int num1 = sc.nextInt();
            System.out.println("Num2  :");
            int num2 = sc.nextInt();
           if(num2 == 0)
            throw new ArithmeticException("divide by zero exception"); //
throw
            int result = num1 / num2;
        }
        catch(Exception ex)
        {
         System.out.println(ex.getMessage());
        }
            sc.close();
```

```
            }
```

-

12. Finally

- it is a keyword in java
- we can not define finally block before try and catch block
- try block can contain only obne finally block
- if we want to release local resources then we should use finally block
- JVM always execute finally block
- before control is coming to finally block,if we forcefully terminate JVM then , the finally block will not get executed

```
Scanner sc = null ; // Local
        try {

            System.out.println("Inside try");
             sc = new Scanner(System.in);
            System.out.println("Num1  :");
            int num1 = sc.nextInt();
            System.out.println("Num2  :");
            int num2 = sc.nextInt();
           if(num2 == 0)
            throw new ArithmeticException("divide by zero exception");


            int result = num1 / num2;

            System.out.println(" result : " + result);
        //  System.exit(0); // forcefull termination
        }
        catch(Exception ex)
        {System.out.println("Inside catch");
         System.out.println(ex.getMessage());

        }
        finally {
            System.out.println("Inside final");
            sc.close();
        }
```

13. try with resource

```
    try(Scanner sc = new Scanner(System.in))
        {
```

```
            System.out.println("Num1  :");
            int num1 = sc.nextInt();
            System.out.println("Num2  :");
            int num2 = sc.nextInt();
            int result = num1 / num2;
            System.out.println(" result : " + result);
            System.exit(0);
        }
        catch(Exception ex) {
         System.out.println(ex.getMessage());

        }
```

14. throws

- if we want to delegate exception from one method to another method ,we use throws clause
- exception should be handled centraly , in main method

```
public static void showRecord()
    {
        try {System.out.println(Thread.currentThread());
        for (int count = 0; count <= 10; count++) {
        System.out.println("Coutn : " + count);

        Thread.sleep(250);

             }
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }



    }
    public static void displayRecord() throws InterruptedException
    {
        System.out.println(Thread.currentThread());
        for (int count = 0; count <= 10; count++) {
        System.out.println("Coutn : " + count);

        Thread.sleep(250);

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

        //Program.showRecord();
        // exception should be handled centraly , in amin method
        try {
```

```
            Program.displayRecord();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

- unchecked exception

```
public static int parseInt(String s )  throws NumberFormat Exc

public static void main(String[] args)  {

        String str = "125";

        int number = Integer.parseInt(str); // no need for try catch as
unchecked

        System.out.println("Number : " + number);


    }
```

- check exception

```
public static native void sleep(long time) throws Interrupted Exception
(checked)

    public static void main(String[] args)  {

        for (int count = 0; count <= 10; count++) {

            System.out.println("Count :" + count);

            try {
                Thread.sleep(250);
            } catch (InterruptedException e) {

                e.printStackTrace();
            }

        }
```

15. Custom Excepton

- JVM cannot understand exceptional conditions generated in business logic
- to handle it we should use user defined/custom exception class.
- 1. Custom Unchecked Exception class

```
class StackOverFlowExcption extends RuntimeException{}
```

- 2. Custom Unchecked Exception class

```
class StackOverFlowExcption extends Exception{}
```

16. Method Overriding

- process of redefining method of super class inside method of sub class is caleed method overriding
- Rules of Method Overridiing -
- 1. Access modifier, of SUB class method should be same or ot should be wider
- 2. return type of sub class method should be same or it should be sub type ,in other word
- 3. Name of method,number of parameters and type of parameters of sub class method
- 4.
- 5. During overriding unchecked exception are not considered.

17. if runtime error(nullpointer,Arrayoutofbound error) get generated due to developers mistake, it is called bug in exception handling case ,

- for this we should not use try catch case

18. Exception : runtime error due to user/client mistake, so we need to use try catch case

- so all checked exception are generated by user/client mistake
- if runtime error ,came due to environmental reason , we cant recover from it, so dont write try catch for it.

19. demo on custom exception

```
@SuppressWarnings("serial")
public class StackOverflowException extends Exception{
    public StackOverflowException(String message) {
        super(message);
    }
}

SuppressWarnings("serial")
public class StackunderflowException extends Exception{
    public StackunderflowException(String message) {
        super(message);
```

```
    }
}
```

20. demo on implementation of stack with custom exception

```
package utils;

import exceptions.StackOverflowException;
import exceptions.StackUnderflowException;

public class Stack {
    private int top = -1;
    private int[] arr;
    public Stack(  ) {
        this( 5 );
    }
    public Stack( int size ) {
        this.arr = new int[ size ];
    }
    public boolean empty( ) {
        return this.top == -1;
    }
    public boolean full( ) {
        return this.top == this.arr.length - 1;
    }
    public void push( int element ) throws StackOverflowException {
        if( this.full())
            throw new StackOverflowException("Stack is full");
        this.arr[ ++ this.top ] = element;
    }
    public int peek( ) throws StackUnderflowException {
        if( this.empty())
            throw new StackUnderflowException("Stack is empty");
        return this.arr[ this.top ];
    }
    public void pop( ) throws StackUnderflowException {
        if( this.empty())
            throw new StackUnderflowException("Stack is empty");
        -- this.top;
    }
}
```

21. demo on main program on stack implement using enum

```
public enum IOOperation {
    EXIT, PUSH, POP
```

```
}

public class Program {
    static Scanner sc = new Scanner(System.in);

    public static void acceptRecord( int[] element ) {
        if( element != null ) {
            System.out.print("Enter element :   ");
            element[ 0 ] = sc.nextInt();
        }
    }
    public static void printRecord( int[] element ) {
        if( element != null )
            System.out.println("Popped element is "+element[0]);
    }
    public static IOOperation menuList() {
        System.out.println("0.Exit");
        System.out.println("1.Push");
        System.out.println("2.Pop");
        System.out.print("Enter choice  :   ");
        return IOOperation.values()[ sc.nextInt() ];
    }
    @SuppressWarnings("incomplete-switch")
    public static void main(String[] args) {
        IOOperation choice;
        int[] element = new int[ 1 ];
        Stack stk = new  Stack();
        while ((choice = Program.menuList()) != IOOperation.EXIT) {
            try {
                switch (choice) {
                case PUSH:
                    Program.acceptRecord( element );
                    stk.push(element[0]);
                    break;
                case POP:
                    element[ 0 ] = stk.peek();
                    Program.printRecord(element);
                    stk.pop();
                    break;
                }
            } catch (StackOverflowException | StackUnderflowException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

22. demo on exception classes accepting sub classes of exception as throw object

```
class A extends Exception{  }
class B extends Exception{  }
class C extends Exception{  }
public class Program {
    //public void print( String str ) throws A, B, C {
    public static void print( String str ) throws Exception {
        if( str == null )
            throw new A( );
        if( str == " ")
            throw new B( );
        if( str == "empty")
            throw new C( );
        System.out.println(str);
    }
    public static void main(String[] args) {
        try {
            Program.print("DAC");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

23. demo on Exception chaining

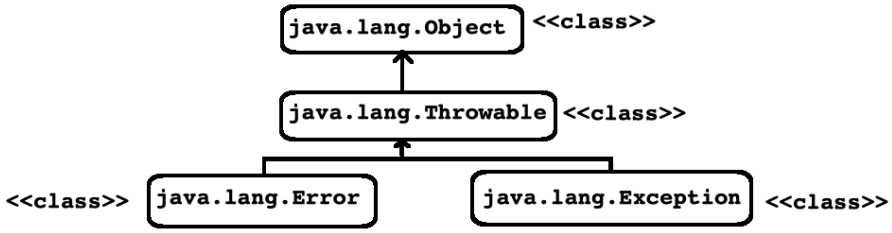
- in case a interfaced method has checked exception in sub class , but not in interface class, so
-
  - in methods catch we can throw another exception method i.e unchecked
  - we can pass the checked Exception as a object in unchecked exception(runtime exception)
  - and implement try catch in main to catch and handle it
- this is known as Exception chaining .

```
interface A{
    void print( );
}
class B implements A{
    public void print()throws RuntimeException {
        try {
            for( int count = 1; count <= 10; ++ count ) {
                System.out.println("Count   :   "+count);
                Thread.sleep(250);
            }
        } catch (InterruptedException cause) {
            throw new RuntimeException(cause); //Exception Chaining
        }
    }
}
public class Program {
```

```
    public static void main(String[] args) {
        try {
            A a = new B( ); //Upcasting
            a.print();
        } catch (RuntimeException e) {
            //Throwable t = e.getCause();
            e.printStackTrace();
        }
    }
}
```

slides

+ Exception Handling
  - Non Java resources / OS resources
    1. File
    2. Thread
    3. Socket
    4. Network Connection
    6. IO Devices
  - If we want to manage os resources carefully in a java application then we should use exception Handling mechanism.
  - If we want to handle exception then we should use 5 keywords:
    1. try
    2. catch
    3. throw
    4. throws
    5. finally
  - What is exception:
    1. Runtime error is called as exception.
    2. Exception is an instance which is used to send notification to the end user of the system if any exceptional condition occurs in the program.



+ Exception handling in C++

```
void setMarks( int marks )
{
  if( marks >= 0 && marks <= 100 )
    this->marks = marks;
  else
    //throw 0;  //OK : C++;
    //throw "invalid marks"; //OK : C++;
    throw Exception("invalid marks"); //OK : C++;
}
```

+ Exception Handling in Java:
  - The Throwable class is the superclass of all errors and exceptions in the Java language.
  - Only objects that are instances of Throwable class (or one of its subclasses) are thrown by the JVM or can be thrown by the Java throw statement.
  - Similarly, only Throwable class or one of its subclasses can be the argument type in a catch clause.
  - Example:

```
throw new Program( ); //Not OK

throw new NullPointerException( ); //OK
```

  - Example

```
try{
}catch( Program p){ //Not OK
}
```

```
try{
}catch( NullPointerException ex){//OK
}
```

+ Error
  - It is a class declared in java.lang package.
  - An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch.
  - Most such errors are abnormal conditions.
  - Example:
    1. VirtualMachineError
    2. OutOfMemoryError
    3. StackOverflowError
  - We can write try-catch block to handle errors but we should avoid it.
  - We can not recover from error hence we should not write try-catch block to handle error.

+ Exception
  - It is a class declared in java.lang package.
  - It is sub class of java.lang.Throwable class.
  - Exception gets generated due to application.
  - Example:
    1. NumberFormatException
    2. NullPointerException
    3. ArrayIndexOutOfBoundsException
  - We can recover from exception.
  - To handle exception, we should use try-catch block.
  - Types of Exception:
    1. Checked Exception
    2. Unchecked Exception

+ JVM do not distinguish between checked and unchecked exception. These types are designed for java compiler

+ Unchecked Exception
  1. java.lang.RuntimeException and all its sub classes are considered as unchecked exception.
  2. It is not mandatory to handle unchecked Exception.
  3. Example:
     1. NumberFormatException
     2. NullPointerException
     3. NegativeArraySizeException
     4. ArrayIndexOutOfBoundsException
     5. ArrayStoreException
     6. IllegalArgumentException
     7. ClassCastException

+ Checked Exception
  1. java.lang.Exception and all its sub classes except java.lang.RuntimeException and all its sub classes are called as checked exception.
  2. It is mandatory to handle checked exception.
  3. Example:
     1. CloneNotSupportedException
     2. InterruptedException
     3. ClassNotFoundException

1. try
   - It is keyword in java.
   - If we want to keep watch on statements then we should use try block/handler.
   - We can not define try block after catch/finally block.
   - In Java, try block must have at least one catch block/finally block/resource.

```
1. try with catch
    try{
      //TODO
    }catch(NullPointerException ex ){
      ex.printStatckTrace();
    }
```

```
2. try with finally.
    try{
      //TODO
    }finally{
      //Clean up code
    }
```

```
3. try with resource.
    try(Scanner sc = new Scanner(System.in)){
      //TODO
    }
```

2. catch
   - It is a keyword.
   - If we want to handle exception the we should use catch block/handler.
   - We should define catch block after try and before finally block.
   - Single try block may have multiple catch blocks.
   - During arithmetic operation if any exception situation occurs then JVM throws ArithmeticException.
   - Catch block can handle exception thrown from try block only.
   - In Java, multi catch block can handle multiple specific exception.
   - Reference of java.lang.Exception class can contain reference of instance of any checked as well as unchecked exception hence to write generic catch block we should use Exception class.

```
try{
  //TODO : Some Code
}catch( Exception ex){
  //Inside generic catch block
  ex.printStackTrace( );
}
```

- If inheritance exists between exception types then it is mandatory to handle all sub type exception first.

```java
Scanner sc = null;
try {
   sc = new Scanner(System.in);
   System.out.print("Num1 :  ");
   int num1 = sc.nextInt();
   System.out.print("Num2 :  ");
   int num2 = sc.nextInt();
   int result = num1 / num2;
   System.out.println("Result  :  "+result);
}catch( ArithmeticException ex ) {
   System.out.println("ArithmeticException");
}catch( RuntimeException ex ) {
   System.out.println("RuntimeException");
}catch( Exception ex ) {
   System.out.println("Exception");
}
```

3. throw
- **It is a keyword in Java.**
- **If we want to generate new exception then we should use throw keyword.**
- **Using throw keyword, we can throw instance of sub class of throwable only.**
- **throw statement is jump statement.**

```
Scanner sc = null;
try {
   sc = new Scanner(System.in);
   System.out.print("Num1 :  ");
   int num1 = sc.nextInt();
   System.out.print("Num2 :  ");
   int num2 = sc.nextInt();
   if( num2 == 0 )
      throw new ArithmeticException("divide by zero exception");
   int result = num1 / num2;
   System.out.println("Result  :  "+result);
}catch( Exception ex ) {
   System.out.println(ex.getMessage());
}
sc.close();
```

4. finally
- **It is keyword in Java.**
- **We can not define finally block before try and catch block.**
- **try block can contain only one finally block.**
- **If we want to release local resources then we should use finally block.**
- **JVM always execute finally block.**
- **before control is coming to finally block, if we forcefully terminate JVM then finally block do not execute.**

5. throws
- **It is keyword in Java.**
- **If want to delegate exception from one method to another method then we should use throws clause.**
- **Example 1:**

  public static int parseInt(String s) throws NumberFormatException

```
public class Program {
   public static void main(String[] args){
      //public static int parseInt(String s)throws NumberFormatException
      String str = "125";
      int number = Integer.parseInt(str); //OK
      System.out.println("Number  :  "+number);
   }
}
```

- **Example 2**

  public static native void sleep(long time)throws InterruptedException

```
public class Program {
   public static void main(String[] args) {
      //public static void sleep(long millis) throws InterruptedException
      try {
         for( int count = 1; count <= 10; ++ count ) {
            System.out.println("Count  :  "+count);
            Thread.sleep(250);
         }
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

3. throw
- **It is a keyword in Java.**
- **If we want to generate new exception then we should use throw keyword.**
- **Using throw keyword, we can throw instance of sub class of throwable only.**
- **throw statement is jump statement.**

```
Scanner sc = null;
try {
   sc = new Scanner(System.in);
   System.out.print("Num1 :  ");
   int num1 = sc.nextInt();
   System.out.print("Num2 :  ");
   int num2 = sc.nextInt();
   if( num2 == 0 )
     throw new ArithmeticException("divide by zero exception");
   int result = num1 / num2;
   System.out.println("Result  :  "+result);
}catch( Exception ex ) {
   System.out.println(ex.getMessage());
}
sc.close();
```

4. finally
- **It is keyword in Java.**
- **We can not define finally block before try and catch block.**
- **try block can contain only one finally block.**
- **If we want to release local resources then we should use finally block.**
- **JVM always execute finally block.**
- **before control is coming to finally block, if we forcefully terminate JVM then finally block do not execute.**

Day8

to read

1. only in case of upcasting , we declare method virtual

- as static member are not designed to call of pointer, it cant be virtual.

```
Shape *ptr = new Rectange();
ptr->calArea();
```

2. what is difference between final,finally,finalize

- 1. final is used with variable, class, method ,we cannot override anything that is final
- 2. finally - it is a block , in exception handling , after catch, used to release local resources
- 3. finalize - it is a protected method, it is used to release class level resources

3. Why generics?
4.

slides

+ Final Method

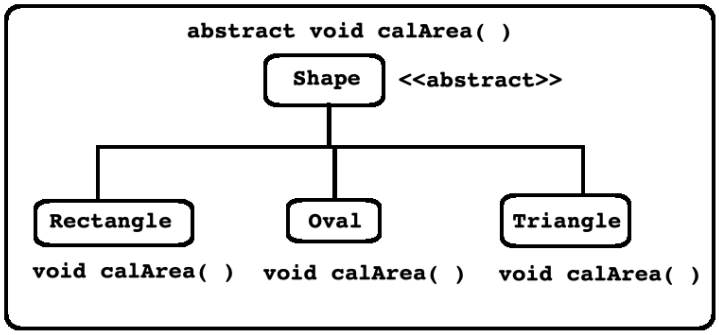
- **If implementation of super class method is logically 100% complete then we should declare super class method final.**
- **Final method is concrete method that we can not override it in sub class.**
- **Final method inherit into sub class hence we can call it on instance of super class as well as sub class.**
- **We can declare overriden method final.**
- **Example:**
  1. **name( ) : Enum**
  2. **ordinal( ) : Enum**
  3. **getClass( ) : Object;**
  4. **wait( ) : Object;**
  5. **notify( ) : Object;**
  6. **notifyAll( ) : Object;**

+ AbstractMethod

- **abstract is keyword in Java.**
- **If implementation of super class method is logically 100% incomplete then we should declare super class method abstract.**
- **Abstract method do not contain body.**
- **If we declare method abstract then it is mandatory to declare class abstract.**
- **We can not instantiate abstract class.**
- **If super class contains abstract method then:**
  1. **Either we should override it in sub class**
  2. **Or we should declare sub class abstract.**
- **W/O declaring method abstract, we can declare class abstract.**
- **Example:**
  1. `abstract int intValue( );`
  2. `abstract float floatValue( );`
  3. `abstract double doubleValue( );`
  4. `abstract long longValue( );`

+ Abstract Class

- **If “is-a” relationship is exist between two types and if we want to use same method design in all the sub classes then we should declare super class abstract.**
- **We can not instantiate abstract class but we can create reference of it.**
- **Abstract class can contain:**
  1. **Field(static as well as non static)**
  2. **Constructor**
  3. **Method(static as well as non static(concrete/abstract))**
  4. **Nested Type**
- **Example:**
  1. **Java.lang.Number**
  2. **java.lang.Enum**
  3. **java.util.Calendar**
  4. **java.util.Dictionary**

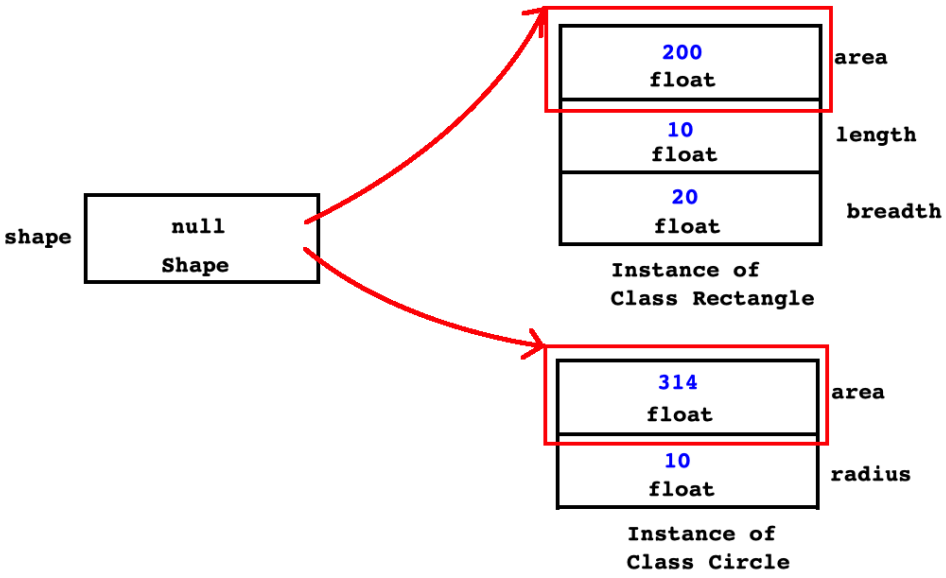


+ Final Class
  - If implementation of any class is logically 100% complete then we should declare such class final.
  - Final class is a concrete class i.e we can instantiate it.
  - We can not extend final class i.e we can not create sub class of final class.
  - Final class can have super class.
  - Example:
    1. java.lang.System
    2. java.lang.Math
    3. java.lang.String/StringBuffer/StringBuilder
    4. All Wrapper Classes
    5. java.util.Scanner
    6. User Defined enum(internally)

+ We can not override following methods in sub class:
  1. Constructor
  2. Private method
  3. Static method
  4. Final Method.

+ We can not use abstract and final keyword together.

- A constructor of super class, which is designed to call from constructor of sub class only is called Sole constructor.

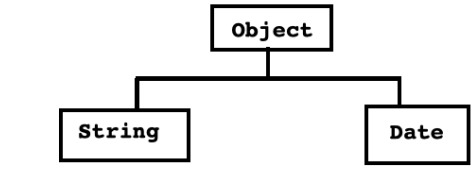


+ In case of upcasting (Shape sh = new Rectangle()), using super class reference:
  1. We can access fields of super class
  2. We can access method of super class
  3. We can not access fields of sub class
  4. We can not access non overriden methods of sub class
  5. We can access overriden methods of sub class

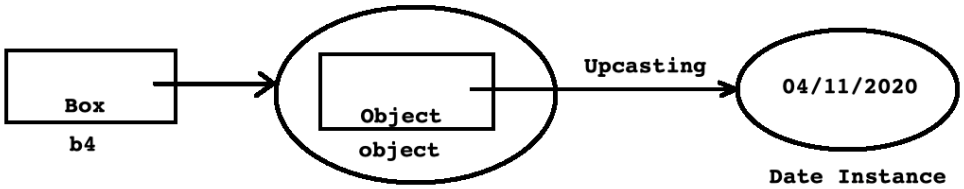
+ **Generics**
  - **If we want to write generic code in Java then we can use:**
    1. **Either java.lang.Object class**
    2. **Or Generics.**

+ **Generic code W/O generics:**

```
class Box{
   private Object object;
   public Object getObject() {
      return object;
   }
   public void setObject(Object object) {
      this.object = object;
   }
}
```




```
Object o = new String( ); //UpCasting
String str = (String)o; //Downcasting : OK

Object o = new Date( ); //UpCasting
Date date = (Date)o; //Downcasting : OK

Object o = null; //UpCasting
String str = (String)o; //Downcasting : null
Date date = (Date)o; //Downcasting : null
```




```
Object o = new Date( ); //UpCasting
String str = (String)o; //Down-casting : ClassCastException
```

```
//Parameterized Type => Generics
class Box<T>{  //T : Type Parameter
   private T object; //null
   public T getObject() {
      return object;
   }
   public void setObject(T object) {
      this.object = object;
   }
}
public class Program {
   public static void main(String[] args) {
      Box<Date> box = new Box<Date>(); //Date : Type Argument

      box.setObject(new Date());

      Date date =  box.getObject(); //OK

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

+ **Why Generics:**

1. Generics gives us stronger type checking at compile time. In other words, using generics we can write type safe generic code.
2. It completely eliminates need of explicit typecasting
3. It helps us to implement generic data structure and algorithm.

+ Frequently used type parameter names:

1. T : Type

2. E : Element

3. N : Number

4. K : Key

5. V : Value

6. U, S : Second Type Parameter names.

+ If we want to put restriction on data type that can be used as a type argument then we should specify Bounded Type Parameter.

+ Specifying bounded type parameter is a job of class implementor.

```
class Box<T extends Number>{ //T extends Number : Bounded Type Parameter
}
public class Program {
   public static void main(String[] args) {
      Box<Number> b1 = new Box<>(); //OK
      Box<Integer> b2 = new Box<>();//OK
      Box<Character> b4 = new Box< >( ); //Not OK
      Box<String> b5 = new Box<>( ); //Not OK
   }
}
```

demos

1. method overriding annotation

```
package test;
class A{
    public void print( int number ) {
        System.out.println("A.print");
    }
}
class B extends A{
    @Override
    public void print( int number ) {
        System.out.println("B.print");
    }
}
public class Program {
    public static void main(String[] args) {
        A a = new B();
        a.print(10);

        B b = new B( );
        b.print( 20 );
    }
}
```

2. final method

- if implementation of super class method is logically 100% complete then,
- we should declare super class method final
- final method is concrete method that we cannot override it in subclass.
- final method get inherited in sub class, hence we can call it on instance of
    - super class as well as sub class.

```
class A{
    // if implementation of super class method is logically complete, then
we should declare method final
    public final void f1() {

        System.out.println("A.f1");
    }
}

class B extends A{
    /* @Override public final void f1() { System.out.println("A.f1"); }
     */  // NOT OK
}

public class Program {
    public static void main(String[] args) {
B b = new B();
b.f1();// final method get inherited in sub class
    }
}
```

- we can declare overrided method as final
- example
- 1. name() : Enum
- 2. orninal ( ) : Enum
- 3. getClass() : Object
- 4. wait(): Object
- 5. notify(): Object
- 6. notifyAll() : Object

```
class A{
    public void f2() {
        System.out.println("A.f2");
    }
}

class B extends A{
    @Override
    public final void f2() {
        super.f2();
    }
}
```

```
class C extends B{
}
public class Program {
    public static void main(String[] args) {
B b = new B();
b.f1();
// final method get inherited in sub class
C c = new C();



    }

}
```

3. Abstract Method

- Abstract is keyword in java
- Dynamic method dispatch : method declare in super class, implemented in sub class
- method should be declared abstract, when in super class implementation of method is 100% logically incomplete
- abstract method do not contain body
- if we declare method abstract, then it is mandetory to declare class abstract.
- f super calss contains abstract method , then
- 1. Either we should override it in sub class
- 2. Or we should declare sub class abstract
- Without declaring method abstract, we can declare class abstract
- Example of abstract class :
- 1. Number class;
  - 1.abstract int intValue();
  - 2.abstract float floatValue();
  - 3.abstract double doubleValue();
- 2. Enum class
  - without declaring method abstract, class is abstract
- 3. abstract class Calendar
  - protected abstract void computeFields()

```
abstract class A{

    // if implementation of super class method is logically complete, then
we should declare method final
    public final void f1() {
        System.out.println("A.f1");
    }
    public abstract void f3();
}

class B extends A{
```

```
    /*
     * @Override public final void f1() { System.out.println("A.f1"); }
     */

    @Override
    public void f3() {
        System.out.println("A.f3");

    }

}
```

4. Abstract Class

- 1. if "is-a" relationship exist between two types
  - and if we want to use same method design in all sub classes
  - then we should declare class Abstract
- 2. we cannot instantiate abstract class but we can create reference of it.
- 3. Abstract class can contain -
  - 1. Field(static as well as non static)
  - 2. Constructor
  - 3. Method(concrete/abstract)(static as well as non static)
  - 4. Nested Type
- 4. Example :
  - 1. java.lang.Number
  - 2. java.lang.Enum
  - 3. java.util.Calender
  - 4. java.util.Dictonary

```
abstract class Test{
    private int num1;
    private static int num2 = 20;
    public Test() {
        this.num1 =10;
    }
    public int getNum1() {
        return num1;
    }
    public static int getNum2() {
        return num2;
    }
    public abstract void print( );
}
class TestImpl extends Test{

    @Override
    public void print() {
        System.out.println("Num1    :   "+this.getNum1());
        System.out.println("Num2    :   "+Test.getNum2());
```

```
    }
}
public class Program {
    public static void main(String[] args) {
        Test test = new TestImpl(); // reference of abstract class,
instance of concrete sub class
        test.print();
    }
}
```

5. Final class

- 1. if implementation of any class is logically 100% complete then we should declare such class final
- 2. Final class is a concrete class i.e we can instantiate it.
- 3. we cannot extend final class i.e we can not create sub class of final class.
- 4. Final class can have super class.
- 5. Example
  - 1. java,lang,System
  - 2. java.lang.Math
  - 3. java.lang.String/StringBuffer/StringBuilder
  - 4. All wrapper class
  - 5. java.util.Scanner
  - 6. User Defined enum()
- 6. we cannot override following method in sub class:
  - 1. Constructor
  - 2. private method
  - 3. Static method
  - 4. final Method
- 7. we cannot use final and abstract keyword together.
- 8. if super class constructor is private, and we use abstract, it gives error

```
abstract class A{


    public final void f1() {
        System.out.println("A.f1");
    }
    public void f2() {
        System.out.println("A.f2");
    }

    public abstract void f3();
}

class B extends A{
```

```
    @Override
    public void f3() {
        System.out.println("B.f3");

    }

}

final class C extends B{

}
//class D extends C{}// not ok

public class Program {


    public static void main(String[] args) {

        C c = new C();

        c.f1();
        c.f2();
        c.f3();
    }

}
```

- 8. if super class constructor is private, and we use abstract, it gives error

```
abstract class A{

    private int num1;

    private A() {
        this.num1 = 10;
    }
    public A(int num1) {
        this.num1 = num1;
    }


}

class B extends A{

    private int num2;
    public B(int num1, int num2) {
        // TODO Auto-generated constructor stub
        super(num1);
        this.num2 = num2;
```

```
    }

}
```

6. Sole Constructor

- abstract class constructor designed for invocation from (extended) sub class constructor
- constructor of super class ,whcih is designed to call from constructor of sub class only is called sole Constructor

```
abstract class A{

    private int num1;
    private int num2;
    public A(int num1, int num2) //sole constructor
    {

        this.num1 = num1;
        this.num2 = num2;
    }

    public void print() {
        System.out.println("Num1 : " + this.num1);
        System.out.println("Num2 : " + this.num2);
    }
}

class B extends A{

    private int num3;

    public B(int num1, int num2,int num3) {
        super(num1, num2);

        this.num3 = num3;
    }

    public void print() {
        System.out.println("Num3 : " + this.num3);
    }


}


public class Program {


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

```
        B b = new B(1,2,3);

        b.print();
        A a = new B(1,2,3);

        a.print();


    }
```

7. In case of upcasting (there is object slicing)

- Example : Shape sh = new Rectangle()
- using super class reference :
    - 1. we can access fileds of super class
    - 2. we can access methods of super class
    - 3. we cannot access fields of sub class
    - 4. we cannot access non overriden methods of sub class
    - 5. we can access overriden methods of sub class
- InstanceOf operator used to get type of object

```
abstract class Shape{

    protected float area;
    public Shape() {
        // TODO Auto-generated constructor stub
    }
    public float getArea() {
        return this.area;
    }

    public abstract void calculateArea();

}



class Rectangle extends Shape{

    private float length;
    private float breadth;

    public void calculateArea() {
        this.area = this.length * this.breadth;
    }
}

class Circle extends Shape{
```

```
    private float radius;

    public Circle() {
        // TODO Auto-generated constructor stub
    }

    public void calculateArea()
    {
        this.area = (float) (Math.PI * Math.pow(this.radius, 2));
    }


}

public class Program {

    static Scanner sc = new Scanner(System.in);

    public static int menuList()
    {
        System.out.println("0. Exit \n 1. Rectangle \n 2. CIrcle \n  Enter
choice ");

        int choice = sc.nextInt();

        return choice;
    }
    private static void acceptRecord(Shape shape) {
        if( shape instanceof Rectangle) // instance of
        {
            Rectangle rect = (Rectangle) shape; // downcasting
            System.out.println("Length : ");
            rect.setLength(sc.nextFloat());
            System.out.println("Breadth : ");
            rect.setBreadth(sc.nextFloat());


        }else {

            Circle c = (Circle) shape;
            System.out.println("radius : ");
            c.setRadius(sc.nextFloat());

        }

    }
    private static void printRecord(Shape shape) {
        String className = shape.getClass().getSimpleName();
        System.out.println("Area of instance of calss " + className + " is
: " + shape.getArea());
    }

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

```
    int choice = 0;

    while((choice = Program.menuList()) != 0)
    {
        Shape shape = null;

        switch (choice) {
        case 1:
            shape = new Rectangle();
            break;
        case 2:
            shape = new Circle();
            break;

        }

        if(shape != null)
        {

            Program.acceptRecord(shape);

            shape.calculateArea();
            Program.printRecord(shape);

        }

    }

    }

}
```

8. reflection for classname

```
private static void printRecord(Shape shape) {

        String className = shape.getClass().getSimpleName();

        System.out.println("Area of instance of calss " + className + " is
: " + shape.getArea());
    }
```

9. use instanceof

```
    if (shape instanceof Rectangle) {

    }
```

10. tester code

```
class ShapeTest {
    private Shape shape = null;

    public void setShape(Shape shape) {
        this.shape = shape;
    }

    static Scanner sc = new Scanner(System.in);

    public void acceptRecord() {
        if (this.shape != null) {
            if (shape instanceof Rectangle) {
                Rectangle rect = (Rectangle) shape; // Downcasting
                System.out.print("Length    :   ");
                rect.setLength(sc.nextFloat());
                System.out.print("Breadth   :   ");
                rect.setBreadth(sc.nextFloat());
            } else {
                Circle c = (Circle) shape; // Downcasting
                System.out.print("Radius    :   ");
                c.setRadius(sc.nextFloat());
            }
        }
    }

    public void printRecord() {
        if( this.shape != null ) {
            String className = shape.getClass().getSimpleName();
            System.out.println("Area of instance of class " + className + "
is : " + shape.getArea());
        }
    }
    public static int menuList() {
        System.out.println("0.Exit");
        System.out.println("1.Rectangle");
        System.out.println("2.Circle");
        System.out.print("Enter choice  :   ");
        return sc.nextInt();
    }
}

public class Program {

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

```
        int choice;
        ShapeTest test = new ShapeTest();
        while ((choice = ShapeTest.menuList( ) )  != 0) {
            test.setShape(ShapeFactory.getInstance(choice));
            test.acceptRecord();
            test.printRecord();
        }
    }
}
```

11.

```
```

12. Generics

- if we want to write generic code in java then we can use:
    - 1. Either java.lang.Object class
    - 2. Or Generics

13. using Object class

```
class Box{
 private Object object;

 public Box(Object object) {
    this.object = object;
 }

 public Object getObject() {
    return object;
 }

 public void setObject(Object object) {
    this.object = object;
 }
 }

 ```
- 2. type safety
   - typecasting done from any primitiva/non primitive type to object,
   - now downcasting must be done back to same primitive /non primitive
type , this is type safety feature
   ```java
   public static void main(String[] args)
    {
    Box b4 = new Box();
```

```
    //Date date = new Date();
    b4.setObject(new Date()); //anonymous object

    // String str = b4.getObject(); // ClassCast Exception

    Date date = (Date) b4.getObject(); //type safe code
    System.out.println(date.toString());

      }
     -- example2
    public static void main(String[] args)
    {
    ArrayList<String> list = new ArrayList<String>();

    list.add("Dac");

    String str = list.get(0);

    System.out.println(str);

     }
```

    ```
14. using Generics
- 1.
```java
// specific class, have data type in class data member
// parametirzed class/Type : Generics
import java.util.ArrayList;
import java.util.Date;

class Box<T>{  // T : Type Parameter
 private T object;
 public Box() {
}

 public Object getObject() {
    return object;
 }

 public void setObject(T object) {
    this.object = object;
 }

}


public class Program{

 public static void main(String[] args)
 {
    //Box b4 = new Box();
    Box<Date> b4 = new Box<Date>(); // Date: Type Argument, for type safety
```
```

```
    b4.setObject(new Date()); //anonymous object


    Date date =  (Date) b4.getObject(); //type safe code
    System.out.println(date.toString());

 }
}
```

15. **Why generics?**
16. Generics gives us stronger type checking at compile time

- in other words,using generics we can write type safe generic code

2. it completely eliminates need fo expilicit typecasting
3. it helps us to implement generic data structure and algorithm

- 1. type inference :
- ability of compiler ,to detect/infer the type of an argument , and return type of argument ,

```
Box<Date> b4 = new Box<>(); /
```

- 2. raw type

```
Box b4 = new Box();  //Raw type : parameteriezed class object,used without
parameterized type
Box<Date> b4 = new Box<>(); // Date: Type Argument, for type safety
```

- 3. use of wrapper class

```
public static void main(String[] args)
    {
        //Box<int> box = new Box<>(); //in generics type argument must be
referenced type
        // so we need wrapper class in generics
        Box<Integer> box = new Box<>();

        box.setObject(10);

        Integer number = box.getObject();
    }
```

- 4. Frequenty used type parameter names:
    - 1. T : Type
    - 2. E : Element
    - 3. N : Number
    - 4. K = Key
    - 5. V = Value
    - 6. U,S = Second ZType Parameter Names
- demo on parameter names

```
interface Entry<K,V>{
    K getKey();
    V getValue();
}

class Pairs<K,V> implements Entry<K,V>{

    private K key;
    private V value;
    @Override
    public K getKey() {

        return this.key;
    }
    @Override
    public V getValue() {
        // TODO Auto-generated method stub
        return this.value;
    }
    public Pairs(K key, V value) {
        super();
        this.key = key;
        this.value = value;
    }
}

public class Program{
    public static void main(String[] args)
    {
        Entry<Integer,String> e = new Pairs(1, "Dac");
        Integer key = e.getKey();
        String value = e.getValue();
        System.out.println("Key : " + key + " \n value : " + value);

    }
}
```

- 5. if we want to put restriction on data type that can be used as a type argument, then ,

    - we should specify **Bounded Type Parameter.**
- to put restriction on type argument ,
- need to put restriction on type parameter, need to use bounded type parameter
- Specifying bounded type parameter is job of class implementor

```
class Box<T extends Number>// bounded type parameter
{}

public static void main(String[] args) {
        Box<Number> b1 = new Box<>();//  ok


        Box<Integer> b2 = new Box<>();//  ok


        Box<Double> b3 = new Box<>();// ok


        Box<Character> b4 = new Box<>(); // not ok

        Box<String> b5 = new Box<>();// not ok


        Box<Date> b6 = new Box<>();// not ok

    }
```

Day9

to read

1. https://unicode-table.com/en/#basic-latin

- read about unicode

2. rules to define immutable class in java

> https://www.journaldev.com/129/how-to-create-immutable-class-in-java

- 1. What is an immutable class in Java? Immutable objects are instances whose state doesn't change after it has been initialized. For example, String is an immutable class and once instantiated its value never changes.
- 2. Benefits of Immutable Class in Java

- An immutable class is good for caching purposes because you don't have to worry about the value changes.

- Another benefit of immutable class is that it is inherently thread-safe, so you don't have to worry about thread safety in case of multi-threaded environment.

- 3. how to create immutable class in java.
- To create an immutable class in Java, you have to do the following steps.

1. Declare the class as final so it can't be extended.
2. Make all fields private so that direct access is not allowed.
3. Don't provide setter methods for variables.
4. Make all mutable fields final so that its value can be assigned only once.
5. Initialize all the fields via a constructor performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.
7. Regular expression (how to validate)

- what is regualar expression for validation of
- 1. email,
- 2. pincode,
- 3. full name,
- 4. ten digit phone number

4. restriction on generics

- To use Java generics effectively, you must consider the following restrictions:

1. Cannot Instantiate Generic Types with Primitive Types
2. Cannot Create Instances of Type Parameters
3. Cannot Declare Static Fields Whose Types are Type Parameters
4. Cannot Use Casts or instanceof With Parameterized Types
5. Cannot Create Arrays of Parameterized Types
6. Cannot Create, Catch, or Throw Objects of Parameterized Types
7. Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

slides

+ **Wild Card** (https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html)
  - **In generics, "?" is called wild card which represents unknown type.**
  - **Types of Wild card:**
    1. **UnBounded Wild Card**
    2. **Upper Bounded Wild Card**
    3. **Lower Bounded Wild Card.**

+ **UnBounded Wild Card:**

```
private static void printList(ArrayList<?> list) {
   if( list != null ) {
      for( Object element : list )
         System.out.println(element);
   }
}
```

In above code, list can contain reference ArrayList, which can contain any(unknown) type of elem

+ **Upper Bounded Wild Card:**

```
private static void printList(ArrayList<? extends Number> list) {
   if( list != null ) {
      for( Object element : list )
         System.out.println(element);
   }
}
```

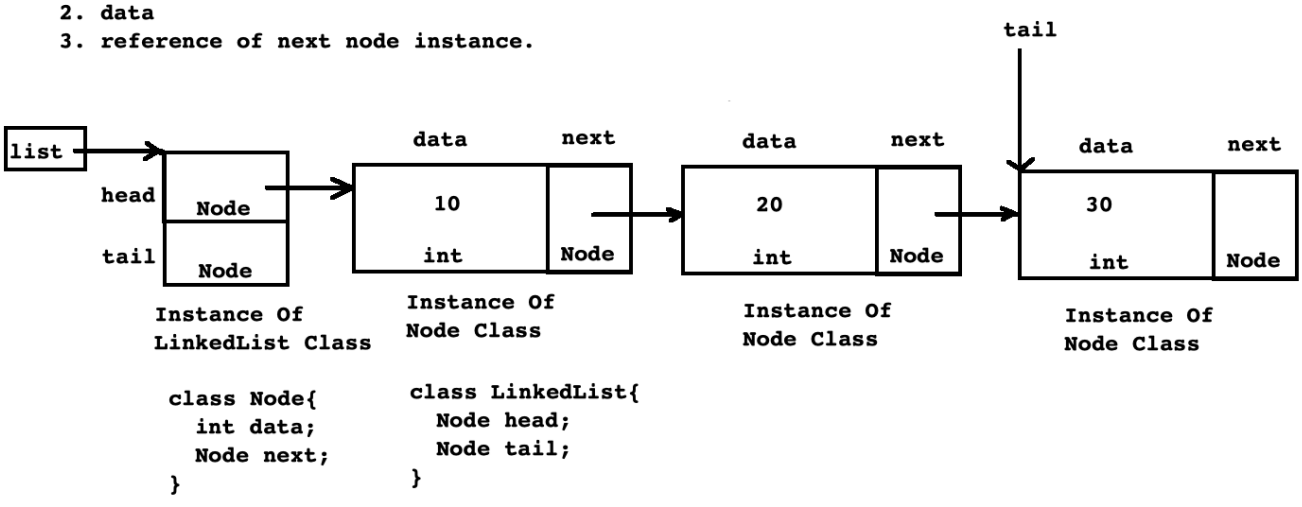
In above code, list can contain reference ArrayList, which can contain Number and its sub type

+ **Lower Bounded Wild Card:**

```
private static void printList(ArrayList<? super Integer> list) {
   if( list != null ) {
      for( Object element : list )
         System.out.println(element);
   }
}
```

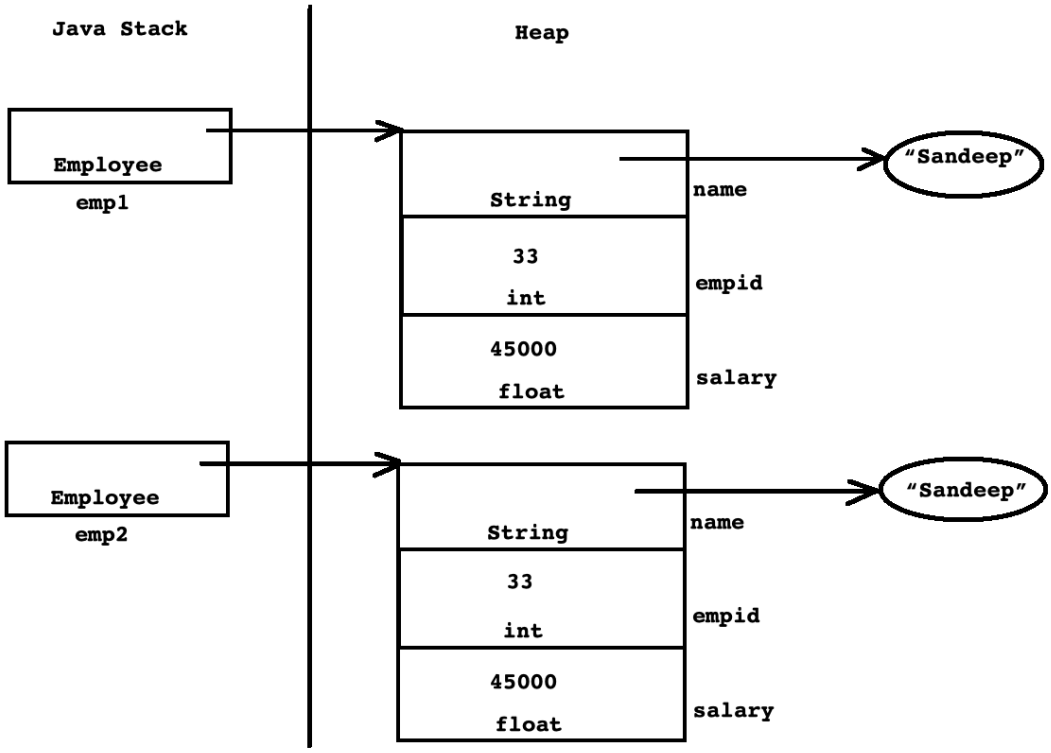
In above code, list can contain reference ArrayList, which can contain Integre and its super type of elements only.

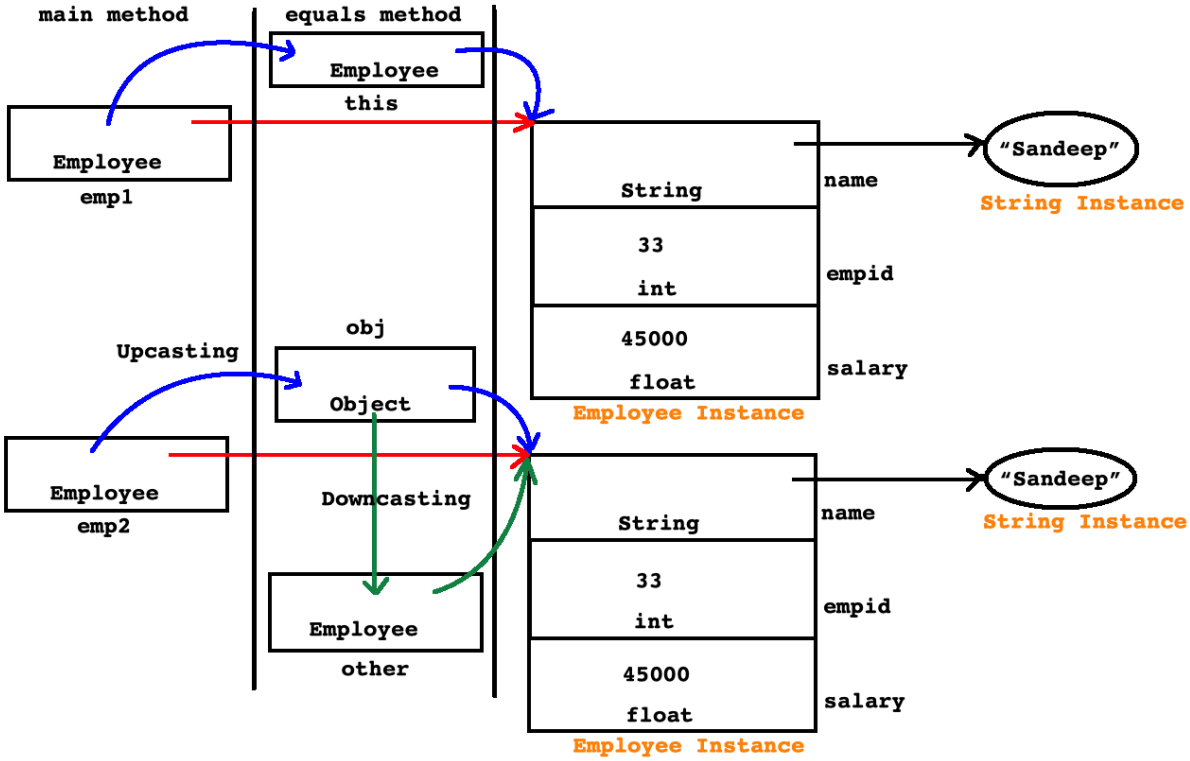
+ **Linked List**
  - **Linked is linear collection.**
  - **It is collection of elements where each element is called as Node.**
  - **Node is instance which may contain 2/3 parts depending on type of linked list.**
  - **Types of linked list are:**
    1. **Singly Linked List**
    2. **Doubly Linked List**
  - **In single linked list, node contains:**
    1. **data**
    2. **reference of next node instance.**
  - **In doubly linked list, node contains:**
    1. **reference of prev node instance.**
    2. **data**
    3. **reference of next node instance.**




```
class Node{
  int data;
  Node next;
}
```

```
class LinkedList{
  Node head;
  Node tail;
}
```





+ Using super class reference, we can not access fields of sub class. To access it we should do down casting.

+ **Equals Method**
  - **“public boolean equals(Object obj)” is a method of java.lang.Object class.**
  - **It is non native, non final method of class.**
  - **If we want to compare state of instance of non primitive type then we should use equals method.**
  - **If we do not define equals method then super class’s equals method gets called.**
  - **equals method of java.lang.Object class do not compare state of instance.**
  - **Consider implementation of equals method of java.lang.Object class:**

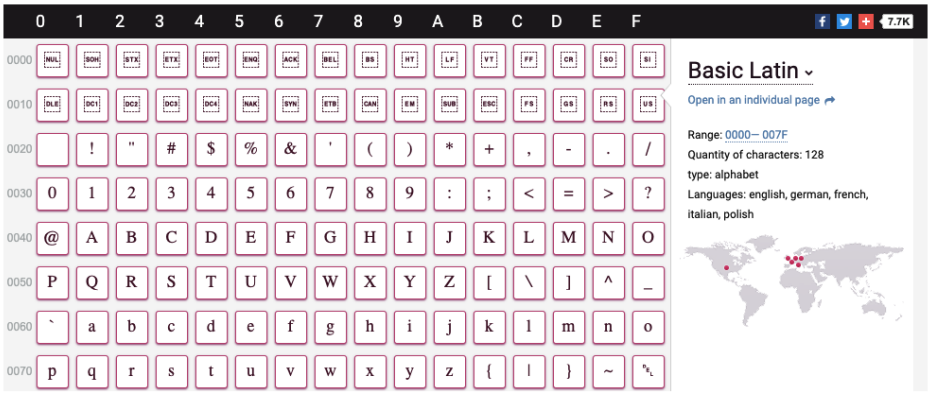
```
public boolean equals(Object obj) {
    return (this == obj); //Compares object references.
}
```

  - **If we want to compare state of instance non final class then it is necessary to override equals method in sub class.**

```
public boolean equals( Object obj ){
  if( obj != null ){
    Employee other = (Employee)obj;
    if( this.empid == other.empid )
      return true;
  }
  return false;
}
```

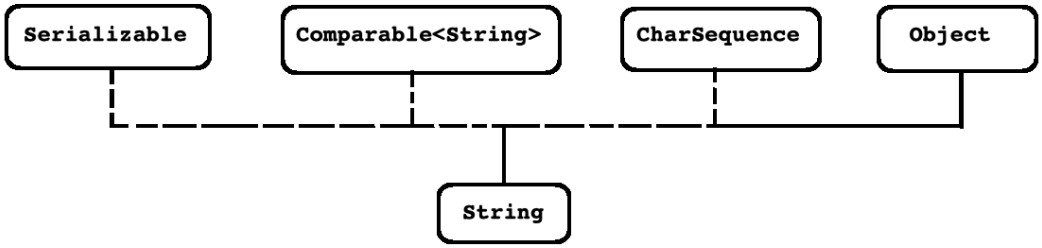
  - **In Collection framework, it is used to compare state of instance.**

+ **Character**
  - **java.lang.Character is wrapper class which wraps value of char type.**
  - **It provides a large number of static methods for determining a character's category (lowercase letter, digit, etc.) and for converting characters from uppercase to lowercase and vice versa.**
  - **The char data type are based on the original Unicode specification, which defined characters as fixed-width 16-bit entities.**
  - **The set of characters from U+0000 to U+FFFF is sometimes referred to as the Basic Multilingual Plane (BMP).**
  - **Reference : http://www.unicode.org/glossary/**



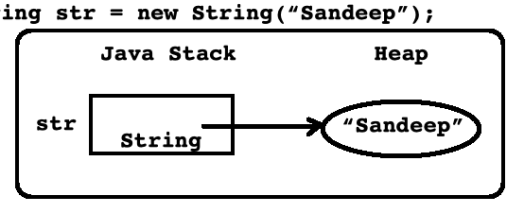
+ String Handling
  - We can use following classes to manipulate String:
    1. java.lang.String
    2. java.lang.StringBuffer
    3. java.lang.StringBuilder
    4. java.util.StringTokenizer
    5. java.util.regex.Pattern
    6. java.util.regex.Matcher

+ String

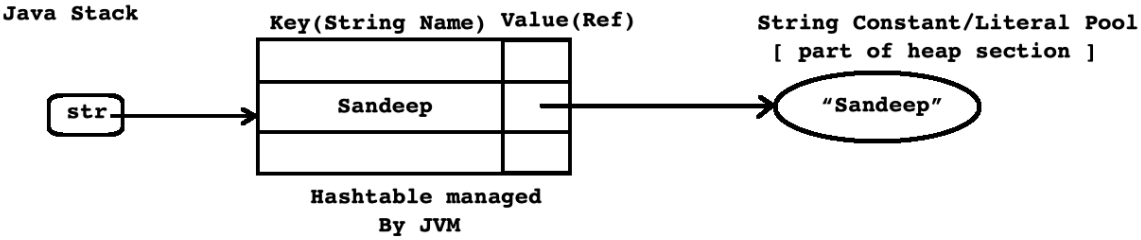


  - java.io.Serializable is a marker interface.
  - "int compareTo(String o)" is a method of java.lang.Comparable interface.
  - java.lang.CharSequence contains following methods:
    1. char charAt(int index);
    2. int length( );
    3. CharSequence subSequence(int start, int end);
  - java.lang.Object is root class which is having 11 methods.
  - java.lang.String is a final class which represents character strings.

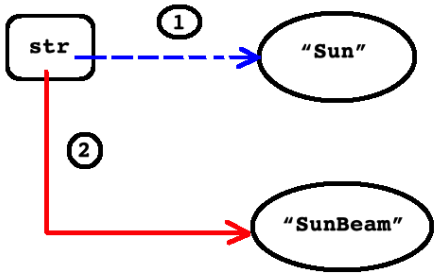
+ java.lang.String class
  - String is not built in type in Java. It is reference type whose instance can be created with and without new operator.
  - Consider Example 1:
    String str = new String("Sandeep");



  - Consider Example 2:
    String str = "Sandeep";
    It is equivalent to
    char[] data = new char[]{'S','a','n','d','e','e','p'};
    String str = new String(data);



+ java.lang.String
  - String instances are constant in other words “immutable”. It means, if we try to modify state of String then JVM creates new String instance.
  - Consider Example:
    String str = “Sun”;
    str = str.concat(“Beam”);



  - “public String concat(String str)” is a method of String class which is used to concat state of string to another string.
  - The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

```
String str = “Pune-”;
str = str + 411057; //OK
str = str + “, India”; //OK
str = str + new java.util.Date(); //OK
```

```
String str = “Pune-”;
str = str.concat(411057); //NoT OK
str = str.concat(“, India”); //OK
```

+ java.lang.String class
  - The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.
  - Constructor(s):

```
1. public String()
//String str = new String( );

2. public String(char[] value)
// char[] chars = {‘D’,’A’,’’C};
//String str = new String( chars );

3. public String(byte[] bytes)
//byte[] bs = new byte[ ]{ 65, 66, 67, 68 };
//String str = new String( bs );

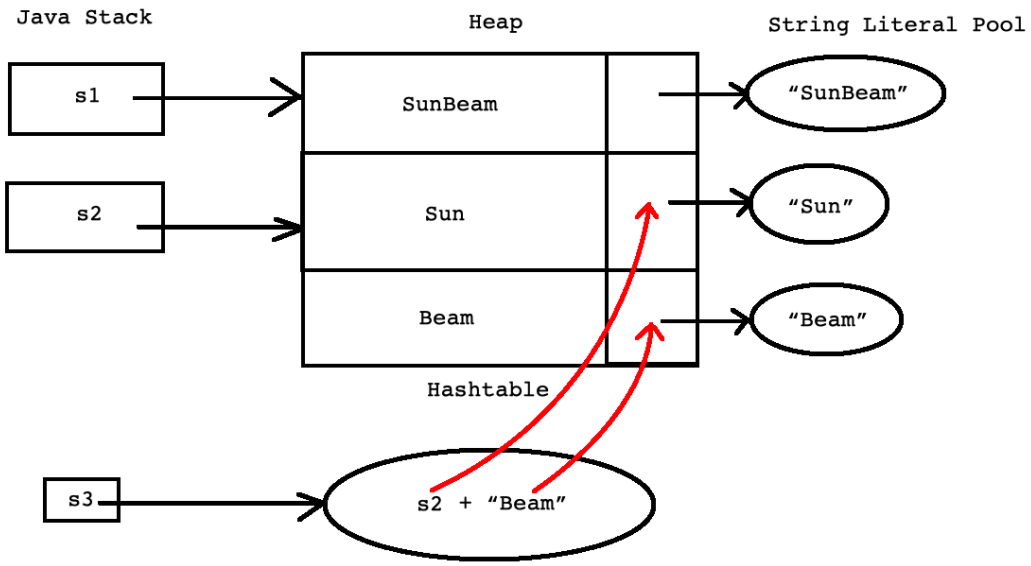
4. public String(String original)
//String str = new String(“SunBeam Pune”);

5. public String(StringBuffer buffer)
//StringBuffer sb = new StringBuffer(“Sandeep Kulange”);
//String str = new String( sb );

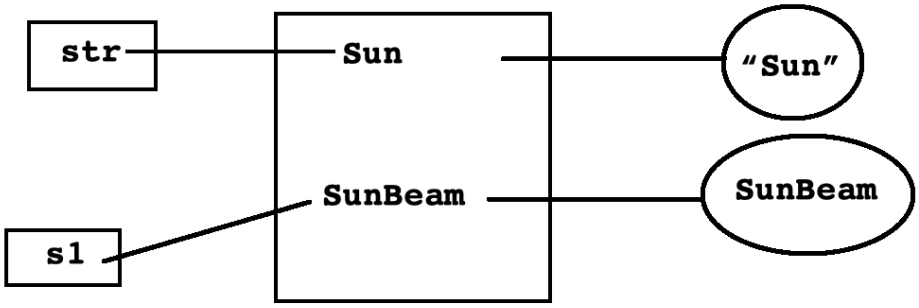
6. public String(StringBuilder builder)
//StringBuilder sb = new StringBuilder(“SunBeam Karad”);
//String str = new String( sb );
```

+ java.lang.String
  - Method(s)
    1. public char charAt(int index)
    2. public int compareToIgnoreCase(String str)
    3. public String concat(String str)
    4. public boolean equalsIgnoreCase(String anotherString)
    5. public static String format(String format, Object... args)
    6. public String intern()
    7. public boolean isEmpty()
    8. public int length()
    9. public boolean matches(String regex)
    10. public String[] split(String regex)
    11. public boolean startsWith(String prefix)
    12. public boolean endsWith(String suffix)
    13. public String substring(int beginIndex, int endIndex)
    14. public char[] toCharArray()
    15. public String toLowerCase()
    16. public String toUpperCase()
    17. public String trim()
    18. public static String valueOf(Object obj)
    19. public int indexOf(String str)
    20. public int lastIndexOf(String str)

```
String s1 = "SunBeam";
String s2 = "Sun";
String s3 = s2 + "Beam";
```



```
String str = “Sun”;
str.concat(“Beam”);
s1 = str.concat(“Beam”);
```



'Day9.14'

notes

1. in java, c, c++, string is not built in datatype , so it gets memory on heap
2. Serializable is an interface in java.io package , it

- is empty interface ,given to generate meta data, it is called marking interface

3. java.lang.string/stringBuffer

demo

1. Demo on generics declaration , solving no reference type for method argument issue in java

```
public class Program {
    private static ArrayList< Integer > getIntegerList( ){
        ArrayList<Integer> list = new ArrayList<>();
        list.add( 10 );
        list.add( 20 );
        list.add( 30 );
        return list;
    }

    private static void showList(ArrayList<Integer> list) {
        if( list != null ) {
            for( Integer element : list )
                System.out.println(element);
        }
    }
```

```
    public static void main(String[] args) {
        ArrayList<Integer> intList = Program.getIntegerList();
        Program.showList( intList );

    }
    public static void main1(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add( 10 );
        list.add( 20 );
        list.add( 30 ); // input

        for( Integer element : list )
            System.out.println(element); //output
    }
}
```

2. demo on using wild card

- 1. Unbounded wild card <?>
- can accept anytype of element in generics
- need to downcast for use

```
private static void printList(ArrayList<?> list) {
        if( list != null ) {
            for( Object element : list )
                System.out.println(element);
        }
    }
```

- 2. Upper bounded wild card <? extends className>

```
private static void printList(ArrayList<? extends Number> list) {
        if( list != null ) {
            for( Object element : list )
                System.out.println(element);
        }
    }
    public static void main(String[] args) {
        ArrayList<Integer> intList = Program.getIntegerList();
        Program.printList( intList );   //OK

        ArrayList<Double> doubleList = Program.getDoubleList();
        Program.printList( doubleList );    //OK
```

```
        ArrayList<String> stringList = Program.getStringList( );
        //Program.printList( stringList );  //Not OK
    }
```

- 3. Lower Bounded wild card <? super className>

```
private static void printList(ArrayList<? super Integer> list)
{
    if(list != null)
    {

        for(Object element : list)
        {
            System.out.println(element);
        }
    }
}



    public static void main(String[] args) {

        ArrayList<Integer> list = Program.getIntegerList();

        Program.printList(list); // ok

        ArrayList<Double> list2 = Program.getDoubleList();

        //Program.printList(list2); // not ok

        ArrayList<String> list3 = Program.getStringList();
    //  Program.printList(list3); not ok

    }
}
```

3. demo on linked list

```
class Node{
    int data;
    Node next; //null

    public Node(int data) {
        this.data = data;
    }
}
public class LinkedList {
```

```
    Node head;
    Node tail;

    public boolean empty() {

        return this.head == null;
    }

    public void addLast(int element)
    {
        Node newNode = new Node(element);
        if(this.empty())
         this.head = newNode;
      else
    this.tail.next = newNode;
this.tail = newNode;
    }
    public void printList() {
        Node trav = this.head;
        while(trav != null)
        {
            System.out.println(trav.data + " ");
            trav = trav.next;
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();
        list.addLast(10);
        list.addLast(20);
        list.addLast(30);
        list.printList();
    }
```

4. methods for comparing primitive type in java

- cannot invoke equals method on primitive data type

```
int num1 = 10;
    int num2 = 10;

    if(num1 == num2)
    System.out.println("equal");

    else
    System.out.println("not equal");
```

```
    // output: equal
```

5. comparing references(emp1,emp2) of reference type object

- as refernce are on java stack, at different location, comparing them gives they are unequal.

```
Employee emp1 = new Employee("suraj", 1, 100);

        Employee emp2 = new Employee("suraj", 1, 100);

        if(emp1 == emp2)//OK : Compares state of references
            System.out.println("Equal");
        else
        System.out.println("not equal");

        //output : not equal

    }
```

6. comparing states of instance of reference type object

```
Employee emp1 = new Employee("Sandeep", 33, 45000);
        Employee emp2 = new Employee("Sandeep", 33, 45000);

        if( emp1.equals(emp2) ) //OK : Compares state of references
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output : Not Equal
```

7. to compare state of instance of reference type,

- we need to overide equals method, and write compare code here
- now we can compare states of instance of object

```
// Employee this = emp1;
    //Object obj = emp2 ; //upcasting
    @Override
    public boolean equals(Object obj) {
        if(obj != null)
        {
            Employee other = (Employee) obj; //downcasting
```

```
            if(this.empid == other.empid)
                return true;
        }
        return false;
    }

    public static void main(String[] args) {

        Employee emp1 = new Employee("suraj", 1, 100);

        Employee emp2 = new Employee("suraj", 1, 100);

        if(emp1.equals(emp2))//OK : Compares state of references
            System.out.println("Equal");
        else
        System.out.println("not equal");

        //output :  equal

    }
```

8. String constructors

- 1. string decalaration using new opertor
  - String in java, doesnt end with null

```
public static void main(String[] args) {
        //public String()
        String s1 = new String( );
        //String s1;    //Object reference: Java Stack
        //new String( );    //String instance : Heap
        System.out.println(s1.isEmpty()+"   "+s1.length());

        }
```

- 2. string declaration without new operator
- converting string to byte

```
public static void main(String[] args) {
        //public String()
    String s1 = "ABC";

    byte[] bs = s1.getBytes();

    //byte : 1 byte : -128 to 127
    System.out.println(Arrays.toString(bs));
    // [65, 66, 67]
```

```
        }
```

- 3. using String(byte[]) constructor

```
public static void main(String[] args) {
    byte[] bs = new byte[] {65,66,67};
    String str = new String(bs);
    //byte : 1 byte : -128 to 127
    System.out.println(str);
    // [65, 66, 67]

        }
```

- 4. using String(char[]) constructor

```
public static void main(String[] args) {
        //public String(char[] value)
        char[] data = new char[ ] {'D','A','C'};
        String str = new String( data );
        System.out.println(str);
    }
```

- 5. using public String(String original) constructor

```
public static void main6(String[] args) {
        //public String(String original)
        String str1 = new String("SunBeam");
        String str2 = new String( str1 );
    }
```

- 6. using public String(StringBuffer buffer)constructor

```
    public static void main7(String[] args) {
        //public String(StringBuffer buffer)
        StringBuffer sb = new StringBuffer("SunBeam");
        String str = new String(sb);
```

```
    }
```

- 7. using public String(StringBuilder builder) constructor

```
    public static void main(String[] args) {
        //public String(StringBuilder builder)
        StringBuilder sb = new StringBuilder("SunBeam");
        String str = new String(sb);
    }
```

9. String memory allocated based on type of decalration

- String Instanced : Heap
- String constant / Literal : String Constant/Literal Pool area (in heap)

```
public static void main1(String[] args) {
        String s1 = new String("SunBeam"); //new String("SunBeam"); :
String Instance
        //String Instanced : Heap

        String s2 = "SunBeam";  //"SunBeam" : String constant / Literal
        //String constant / Literal : String Constant/Literal Pool
    }
```

10. String comparison in case of String constant/literal

- 1. equals methods works
- 2. also comparing reference of string method works ,
- as literal area store all same object ,as single reference, managed by jvm by hash table
    - so both string has same reference

```
public static void main(String[] args) {
        String s1 = "SunBeam";
        String s2 = "SunBeam";
        if( s1 == s2)
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output : Equal
    }

    public static void main5(String[] args) {
        String s1 = "SunBeam";
        String s2 = "SunBeam";
        if( s1.equals(s2))
```

```
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output : Equal
    }
```

11. as concat creates new instance of string, so new reference is created, so boh string have different reference

- with concat, only string objects/constants can be added ,and new string is created

```
public static void main(String[] args) {
        String s1 = "Sun";
        System.out.println(s1); //Sun

        String s2 = s1.concat("Beam");
        System.out.println(s2); //SunBeam

        if( s1 == s2 )
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output :  Not Equal
    }
```

12. string (+) concetination operator can be used to add any object to string

```
public static void main(String[] args) {
        String str = "Pune-";
        str = str + new Date();
        System.out.println(str);    //OK
    }
    public static void main3(String[] args) {
        String str = "Pune-";
        str = str + "India";
        System.out.println(str);    //OK
    }
    public static void main2(String[] args) {
        String str = "Pune-";
        str = str + 411056;
        System.out.println(str);    //OK
    }
```

13. comparison in case on constant Strings, i.e
    - value assigned at compile time itself

```
public static void main8(String[] args) {
        String s1 = "SunBeam";
        String s2 = "Sun"+"Beam";   //"SunBeam"
        if( s1.equals(s2) )
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output : Equal
    }

    public static void main7(String[] args) {
        String s1 = "SunBeam";
        String s2 = "Sun"+"Beam";   //"SunBeam"
        if( s1 == s2 )
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output : Equal
    }
```

14. comparison in case on non constant Strings i.e,

- - value assigned at run time
- in this case using equals method, checkes state gies equal if both string are same

```

```

15. in case of Intern() method, on non constant string , reference comparison , gives equal

- 1. public String intern()
  - A pool of strings, initially empty, is maintained privately by the class String.
  - When the intern method is invoked,

1. if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned.
2. Otherwise, this String object is added to the pool and a reference to this String object is returned.

- here same refernce for s1 and s3

```
public static void main11(String[] args) {
        String s1 = "SunBeam";
        String s2 = "Sun";
        String s3 = ( s2 + "Beam" ).intern();
        if( s1 == s3 )
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
```

```
        //Output : Equal
    }
```

16. demo showing same reference is shared amng all String constant i.e in literal area, by hash table

```
package p1;
class A{
    public static String str = "Hello";
}


package test;
class B{
    public static String str = "Hello";
}
public class Program {
    public static String str = "Hello";
    public static void main(String[] args) {

         String str = "Hello";
         System.out.println(A.str == B.str);    //true
            System.out.println(A.str == Program.str);//true
            System.out.println(A.str == str);//true

            System.out.println(B.str == Program.str);   //true
            System.out.println(B.str == str);//true

            System.out.println(Program.str == str);//true

    }
```

17. in string types

- 1. String reference with same instance-- get same hashcode
- 2. StringBuffer reference with same instance-- get different hashcode
- it get memory on heap, so comparison gives unequal
- 3. StringBuilder reference with same instance-- get different hashcode
- - it get memory on heap, so comparison gives unequal

```

```

18. demo on performing operation on String

- reversing a number
- as String is immutalbe ,
  - we need to convert it to mutable type to perform operation
- so convert it into StringBuffer/StringBuilder

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

        try(Scanner sc = new Scanner(System.in))
        {
            System.out.println("Number : ");
            int number = sc.nextInt();

            String strNumber = String.valueOf(number);

            StringBuilder sb = new StringBuilder(strNumber);

        //  StringBuffer sb = new StringBuffer(strNumber);
            sb.reverse();

            strNumber = sb.toString();

            number = Integer.parseInt(strNumber);
            System.out.println("Number : " + number);

        }

    }

    --- method 2
    public static void main(String[] args) {

        try(Scanner sc = new Scanner(System.in))
        {
            System.out.println("Number : ");

            System.out.println("Number : " + Integer.parseInt(new
StringBuilder(String.valueOf(sc.nextInt())).reverse().toString()));
        }

    }
```

19. StringTokenizer class demo

```
{  // multiple delimeter
        String str = "1a+2b*3c-4d/5e";
        String delim = "+*-/";
        StringTokenizer stk = new StringTokenizer(str,delim, true);
        //System.out.println(stk.countTokens());
        //StringTokenizer stk = new StringTokenizer(str,delim, false);
        //StringTokenizer stk = new StringTokenizer(str,delim);
        String token = null;
        while( stk.hasMoreTokens()) {
            token = stk.nextToken();
            System.out.println(token);
```

```
        }
    }

    {//split operation
        String str = "www.sunbeaminfo.com";
        String delim = ".";
        StringTokenizer stk = new StringTokenizer(str,delim);
        //StringTokenizer stk = new StringTokenizer(str,delim,true);
        String token = null;
        while( stk.hasMoreTokens()) {
            token = stk.nextToken();
            System.out.println(token);
        }
    }



     { // using token class
        String str = "SunBeam Institute of Information Technology";
        StringTokenizer stk = new StringTokenizer(str);
        String token = null;
        while( stk.hasMoreTokens()) {
            token = stk.nextToken();
            System.out.println(token);
        }
    }


     {// using enumeration class

        String str = "Sun Beam INfo Tech";

        StringTokenizer stk = new StringTokenizer(str);

        String token = null;

        while(stk.hasMoreElements())
        {
            token = (String) stk.nextElement();
            System.out.println(token);
        }

    }

    {
        String str = "Sun Beam INfo Tech"
        StringTokenizer stk = new StringTokenizer(str)
        System.out.println("count : " + stk.countTokens());
    }
```

20. Regular Expression for validation email,name.mobile no

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;


class Validator{
    public static String Name_REGEX = "\\p{Upper}(\\p{Lower}+\\s?)";
    public static boolean validateName( String name ) {
        return name.matches(Name_REGEX);
    }
    public static String EMAIL_REGEX = "^[a-zA-Z0-9+_.-]+@[a-zA-Z0-9.-]+$";
    public static boolean validateEmail( String email ) {
        return email.matches(EMAIL_REGEX);
    }
    public static String MOBILE_REGEX = "\\d{10}";
    public static boolean validateMobile( String mobile ) {
        return mobile.matches(MOBILE_REGEX);
    }
}


public class Program {
    public static void main(String[] args) {
        try( Scanner sc = new Scanner(System.in)){
            System.out.print("Mobile no :   ");
            String mobile = sc.nextLine();
            if( Validator.validateMobile(mobile))
                System.out.println(mobile);
            else
                System.out.println("Invalid Mobile number");
        }
    }

    public static void main5(String[] args) {
        try( Scanner sc = new Scanner(System.in)){
            System.out.print("Name  :   ");
            String name = sc.nextLine();
            if( Validator.validateName(name))
                System.out.println(name);
            else
                System.out.println("Invalid Name");
        }
    }


    public static void main4(String[] args) {
        try( Scanner sc = new Scanner(System.in)){
            System.out.print("Email :   ");
            String email = sc.nextLine();


        {
```

```
            if( Validator.validateEmail(email))
                System.out.println(email);
            else
                System.out.println("Invalid Email");
        }// using custom class

        {

            if(email.matches(regex))
            {
                System.out.println(email);
            }
            else
                System.out.println("Invalid email");
        }//using String class method

        {

            if(Pattern.matches(regex, email))
            {
                System.out.println(email);
            }
            else
                System.out.println("Invalid email");
         }// using Pattern class Method
    }
```

Day10

read

1. Specification /Interface

- set of rules is called specification
- specification is also called Standard

2. when to use abstract class and when to use Interface class?

- 1. **Abstract class**
  - Abstract class can extend only one abstract/concrete class
  - abstract class may/may not contain abstract method
  - we can define constructor inside abstract class
  - **use abstract class**
  - 1. when state is involved between super and sub classes, super type must be abstract
    - as we can declare field
    - if there is(is - a) relationship , declare class abstract
    - ex . shape involve area, in rect, circle
- 2. **INTERFACE**
  - interface can extend one or more than one interface

        - insterface method are by default abstract
        - we cannot define constructor inside interface
        - if state is not involved in Super type it should be interface
        - its name end with able, generally it is a interface,as it is used to implement some compulsory logic
        -
    - **use Interface**
    - when state is not involved between classes
    - and same method design must be implemented in all sub classes
    - then (can - do) relationship exist between classes, so make a interface where method is declared , and use interface implementation inheritence

> C implements I

    - we cannot declare field , i.e except final static

3. difference between comparable (in java.lang,same type instance) and comparator (in java.util, different type instance) interface in java?

- Comparable vs Comparator

1. Comparable interface can be used to provide single way of sorting

- whereas Comparator interface is used to provide different ways of sorting.

2. For using Comparable, Class needs to implement it

- whereas for using Comparator we don’t need to make any change in the class.

3. Comparable interface is in java.lang package

- whereas Comparator interface is present in java.util package.

4. We don’t need to make any code changes at client side for using Comparable, Arrays.sort() or Collection.sort() methods automatically uses the compareTo() method of the class.

- For Comparator, client needs to provide the Comparator class to use in compare() method.

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

4. jvm architecture

> https://medium.com/platform-engineer/understanding-jvm-architecture-22c0ddf09722

slides

```
+ Nested class
  - We can define class inside scope of another class. It
    is called nested class.
  - Example:
     //Outer.class
     class Outer{ //Top Level Class
        //Outer$Inner.class
        class Inner{ //Nested class

        }
     }
  - Access modifier of top level class can be either package
    level private or public only. We can use any access modifier
    on nested class.
  - Nested class implementation represents encapsulation.
  - Types of nested class:
     1. Non static nested class( also called as Inner class )
     2. Static nested class


+ Non Static Nested class
  - It is also called as inner class.
  - If implementation of nested class depends on
    implementation of top level class then nested class
    should be non static.
  - Example:
     class LinkedList{
      Node head;
      class LinkedListIterator{//Non static nested class
       private Node trav = LinkedList.this.head;
      }
     }

 - Suggestion:
    For simplicity, consider non static nested class as a
    non static method of a class.
    If we want to invoke non static method then we use
     instance of a class
```

- Syntax

```
//Top Level class
class Outer{ //Outer.class
 //Non static nested class / inner class
 public class Inner{ //Outer$Inner.class
 }
}
```

- Instantiation of Top level class:

```
Outer out = new Outer();
```

- Instantiation of non static nested class:

```
Outer out = new Outer();
Outer.Inner in = out.new Inner( );
```

- or:

```
Outer.Inner in = new Outer().new Inner( );
```

- Top level class can contain static as well as non static members.
- Non static nested class / inner class do not contain static members
- If we want to declare any field static then it must be final.
- Using instance, we can access members of inner class inside method of top level class.
- W/O instance, we can access members of top level class inside method of inner class.
- In case of name clashing/shadowing, to access non static members of top level class inside method of inner class, we should use “TopLevelClass.this” syntax.

+ Static Nested class
  - We can use only public, final and abstract modifier on top level class.
  - We can not declare top level class static but we can declare nested class static.
  - If implementation of nested class do not depends on top level class then we should declared nested class static.
  - If we declare nested class static then it is called static nested class.
  - Suggestion:
    For simplicity, consider static nested class as a static method of a class.

- Syntax

```
//Top Level class
class Outer{  //Outer.class

  //Nested class
  static class Inner{ //Outer$Inner.class

  }
}
public class Program {
  public static void main(String[] args) {
    Outer out = new Outer();

    Outer.Inner in = new Outer.Inner();
  }
}
```

- Static nested class can contain static as well as non static members.
- We can instantiate static nested class.
- Using instance, we can access members of static nested class inside method of top level class.
- W/O instance, We can access static members of top level class inside method of static nested class.
- Using instance, we can access non static members of top level class inside method of static nested class.

+ Local Class
  - We can define class inside function. It is called local class.
  - In java, local class is also called as method local class
  - Types:
    1. Method Local Inner class
    2. Method Local Anonymous inner class
  - In java, we can not declare local variable/class static hence method local class is also called as method local inner class.

+ Method Local Inner class

```
//Top level class
public class Program { //Program.class
   public static void main(String[] args) {
      //Method Local Inner class
      class Complex{ //Program$1Complex.class

      }
      Complex c1 = null;
      c1 = new Complex();
   }
   //Complex c2;  //Error
   //Complex c3 = new Complex(); //Error
}
```

- **Non static class declared inside method is called method local inner class.**
- **We can not create reference or instance of method local inner class outside method**

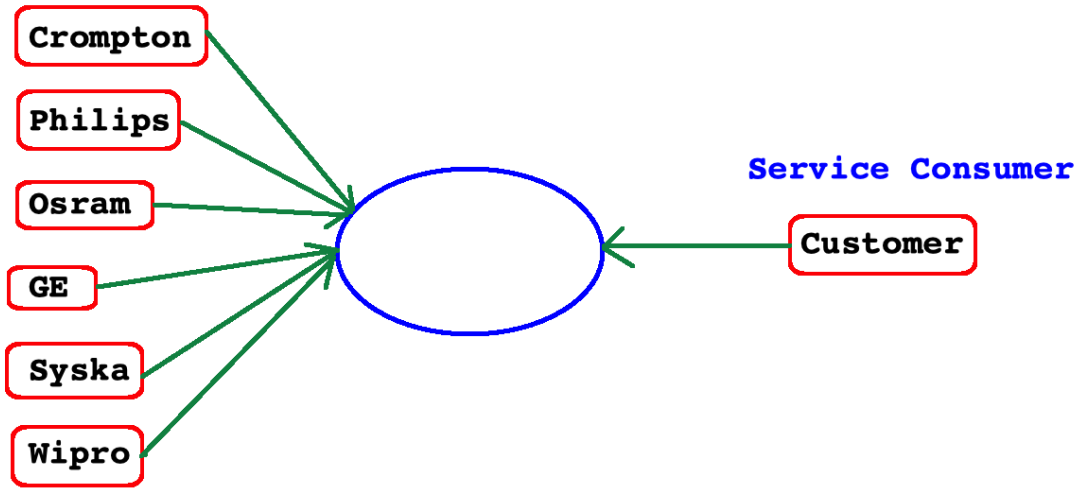
+ Method Local Anonymous Inner class

- **In Java, we can define class W/O name, it is called anonymous class.**
- **In Java, we can define anonymous class inside method only hence it is called method local anonymous class.**
- **Local class can not be static hence it is also called method local anonymous inner class**
- **If we want to define anonymous inner class then it is necessary to take help of existing concrete class, abstract class or interface.**

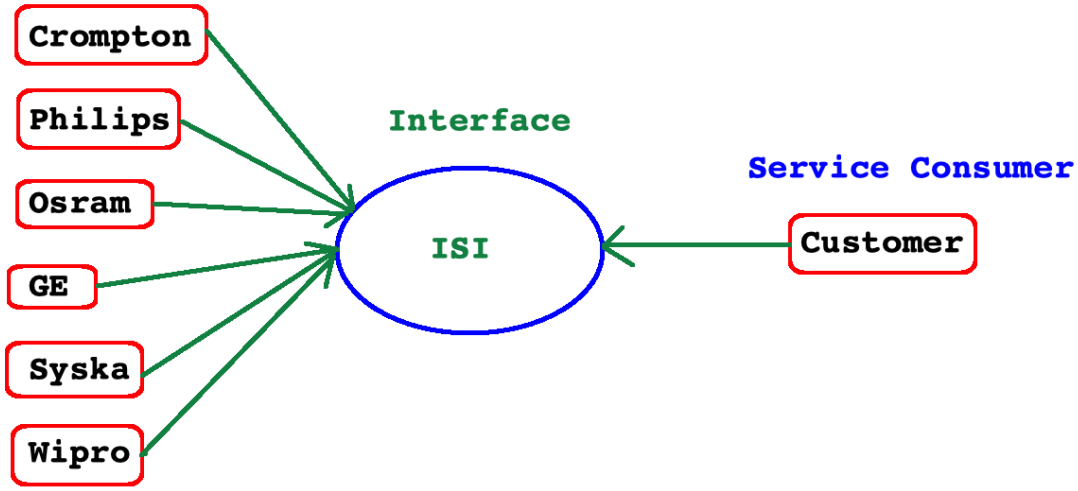
- **Syntax:**

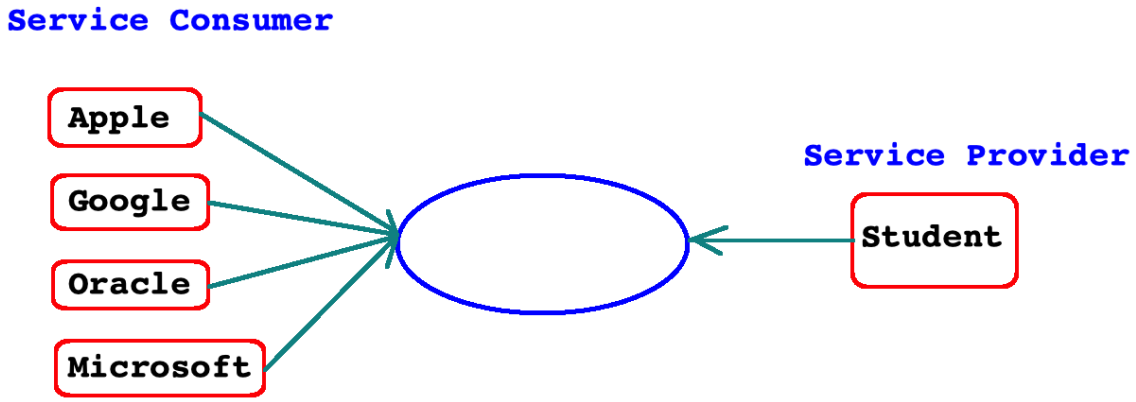
```
Object obj; //reference
new Object( ); //Anonymous instance
Object obj = new Object(); //Instance with reference
Object obj = new Object( ) { }; //Anonymous inner class
```

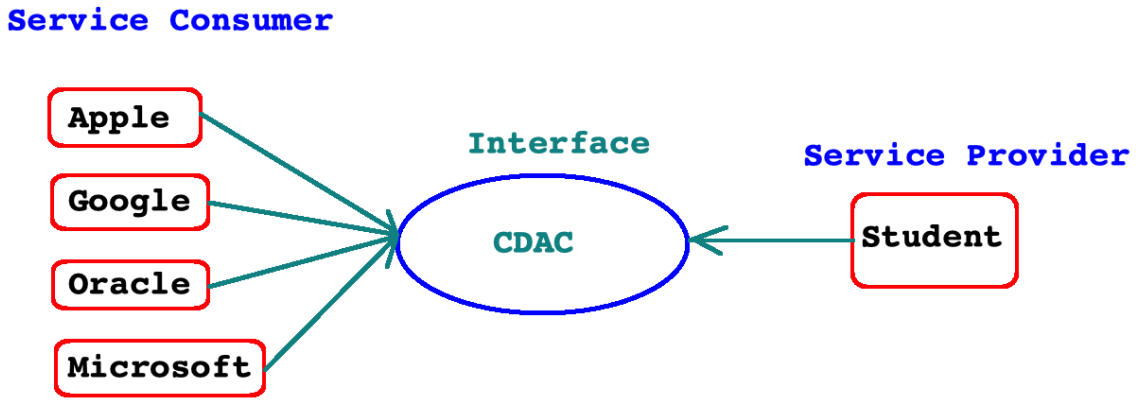
Service Providers

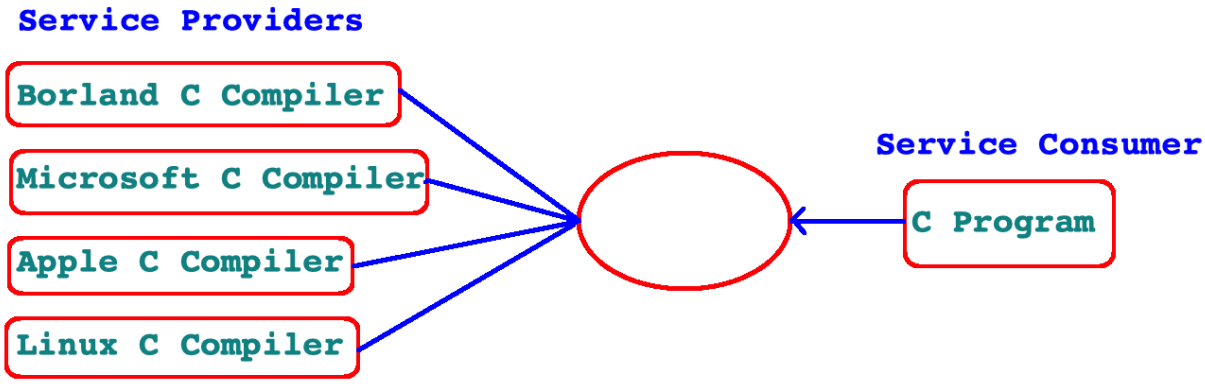


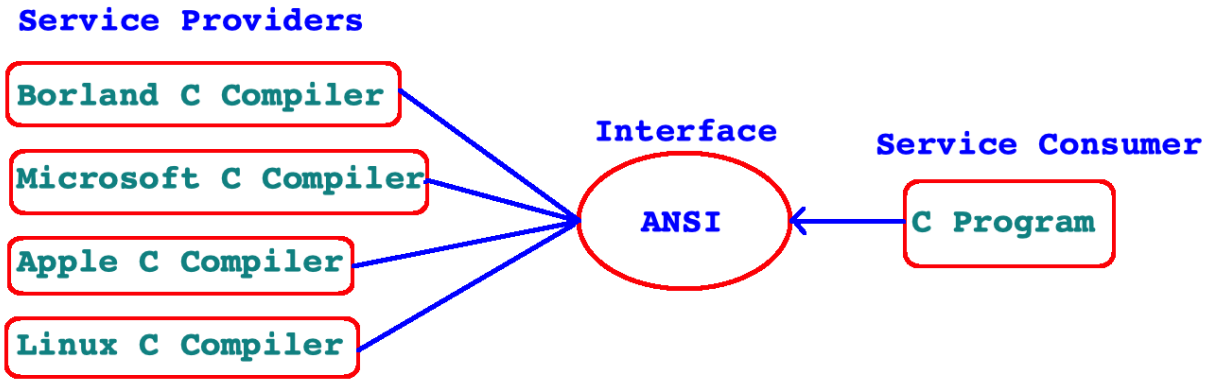
Service Providers




Service Consumer
Apple
Google
Oracle
Microsoft
Service Provider
Student


Service Consumer
Apple
Google
Oracle
Microsoft
Interface
CDAC
Service Provider
Student


Service Providers
Borland C Compiler
Microsoft C Compiler
Apple C Compiler
Linux C Compiler
Service Consumer
C Program


Service Providers
Borland C Compiler
Microsoft C Compiler
Apple C Compiler
Linux C Compiler
Interface
ANSI
Service Consumer
C Program

+ Interface
  - Set of rules is called specification/standard.
  - In java Specification = { abstract classes and interfaces }
  - If we want to define specification for the sub classes then we should use interface.
  - Interface is reference type in java.
  - Interface helps us:
    1. To develop trust between consumer and provider
    2. To reduce dependancy

+ Interface in java
  - It is a reference type.
  - interface is keyword in java.
  - It can contain:
    1. Nested Type
    2. Field
    3. Abstract method
    4. Static Method
    5. Default method

- We can declare field inside interface.
- Interface fields are by default public static and final.
- We can declare method inside I/F. It is by default public and abstract.
- If we want to implement specifications then we should use implements keyword.
- It is mandatory to override all the abstract methods of I/F otherwise sub class will be considered as abstract.
- Example:

```
interface A{
  void print();
}
```

```
abstract class B implements A{
  //TODO
}
```

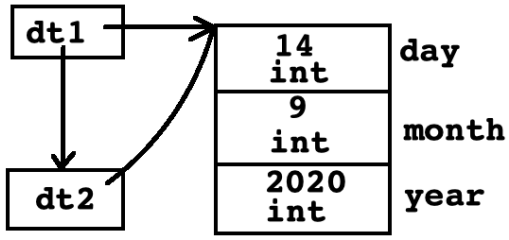
```
class C implements A{
  @Override
  public void print(){
    //TODO
  }
}
```

- **We can not instantiate abstract class and interface. But we can create reference of it.**
- **We can not define constructor inside interface.**
- **I1, I2, I3 : Interfaces**
- **C1, C2, C3 : Classes**

1. **I2 implements I1 //Not OK**
2. **I2 extends I1 //OK**
3. **I3 extends I1,I2 //OK**
4. **I1 extends C1 //Not OK**
5. **C1 extends I1 //Not OK**
6. **C1 implements I1 //OK**
7. **C1 implements I1,I2 //OK**
8. **C2 implements C1 //Not OK**
9. **C2 extends C1 // OK**
10. **C3 extends C1, C2 // Not OK**
11. **C2 implements I1 extends C1 //Not OK**
12. **C2 extends C1 implements I1**
13. **C2 extends C1 implements I1,I2,I3**

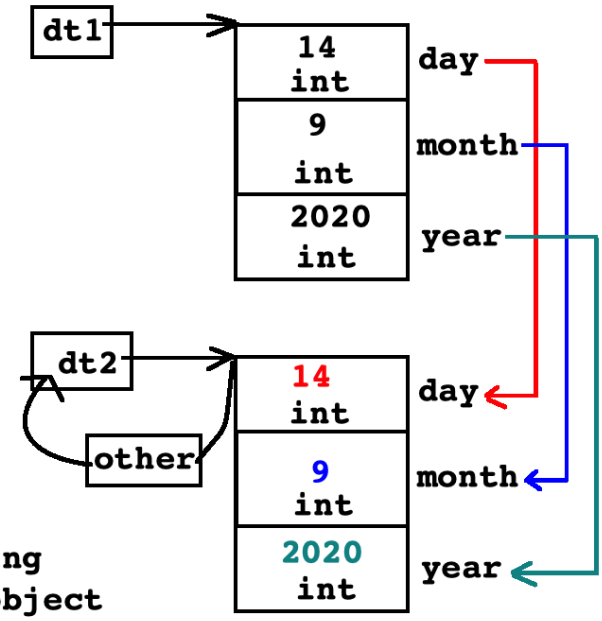
- **Interface can extends one / more than one interfaces.**
- **Class can implement one / more than one interfaces.**
- **Class extend only one class.**
- **First we should extend the class then implement I/F.**

- **Frequently used interfaces**
  1. **java.lang.AutoCloseable**
  2. **java.lang.Cloneable**
  3. **java.lang.Comparable**
  4. **java.util.Comparator**
  5. **java.lang.Iterable**
  6. **java.util.Iterator**
  7. **java.io.Serializable**




```
Date dt1 = new Date();
Date dt2 = dt1;
```

Shallow Copy
of References



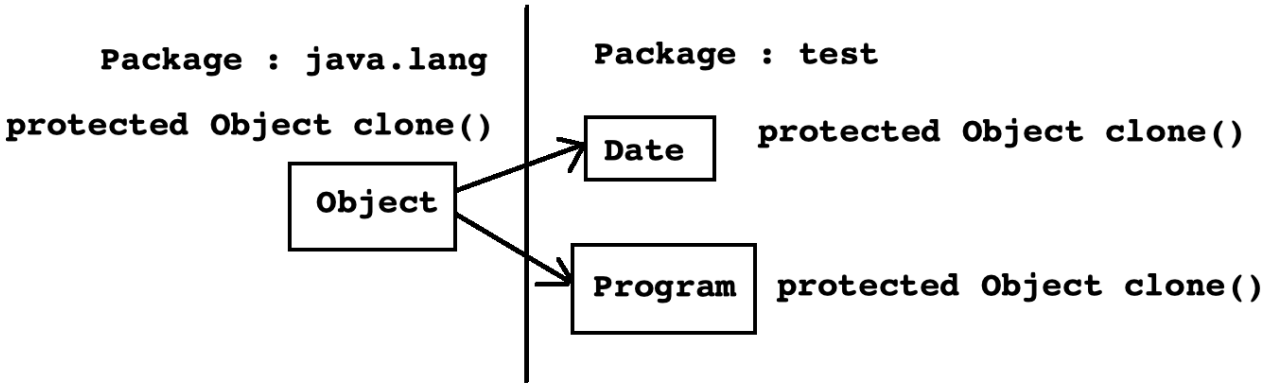
Shallow Copy
of Instances

Shallow Copy is process of copying content of object into another object as it is.
It is also called as bitwise copy.

- Creating instance and copying contents is shallow copy in java.

- If we want to create new instance from existing instance then we should use clone method.
- clone() is native , non final method of java.lang.Object class.
- Syntax:

```
protected native Object clone( )throw CloneNotSupportedException
```




```
Date dt1 = new Date( );
Date dt2 = (Date)dt1.clone();
```

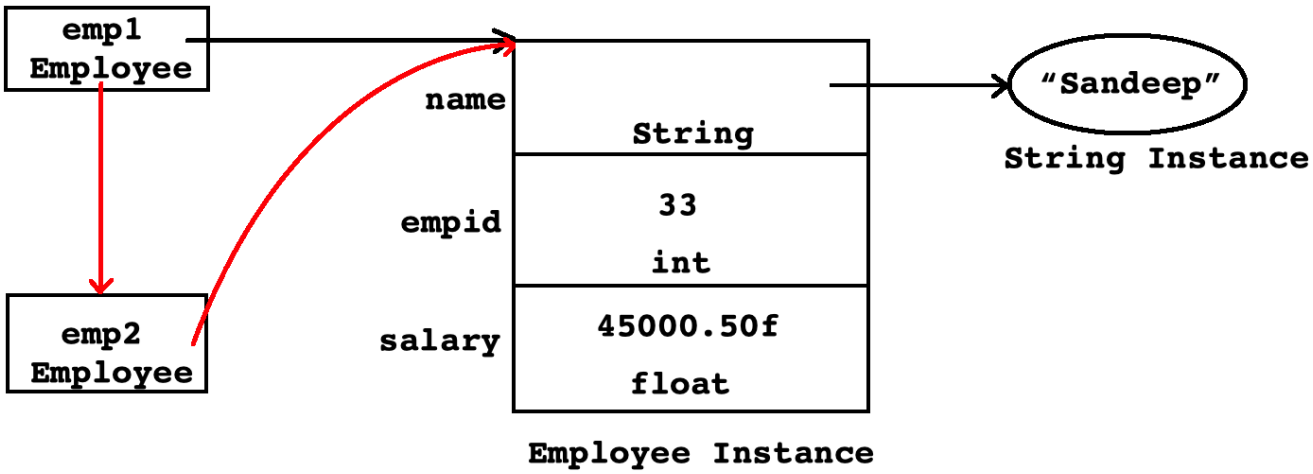
//NOT OK

- If we want to create shallow copy of current instance then we should use “super.clone()” method.

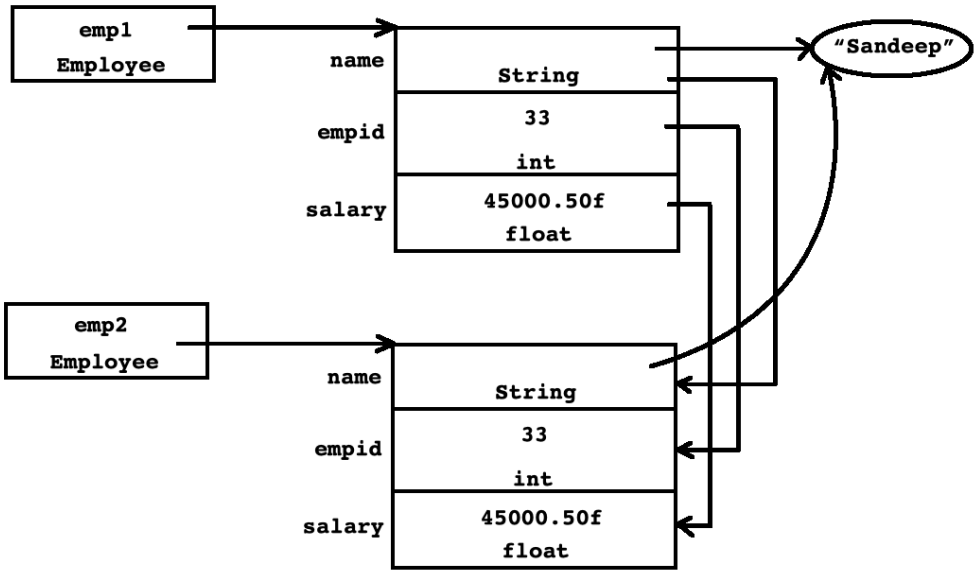
- If we want to clone an object then type of object/instance must implement java.lang.Cloneable interface.
- An interface which do not contain any member is called marker/tagging interface.
- Such interfaces are used to generate metadata for JVM.
- Example:
  1. java.lang.Cloneable
  2. java.util.RandomAccess
  3. java.util.EventListener
  4. java.io.Serializable
  5. java.rmi.Remote

- "Object clone()" is a method of java.lang.Object class.
- W/O implementing Cloneable I/F, if we try to create clone of the object then clone() method throw CloneNotSupportedException.

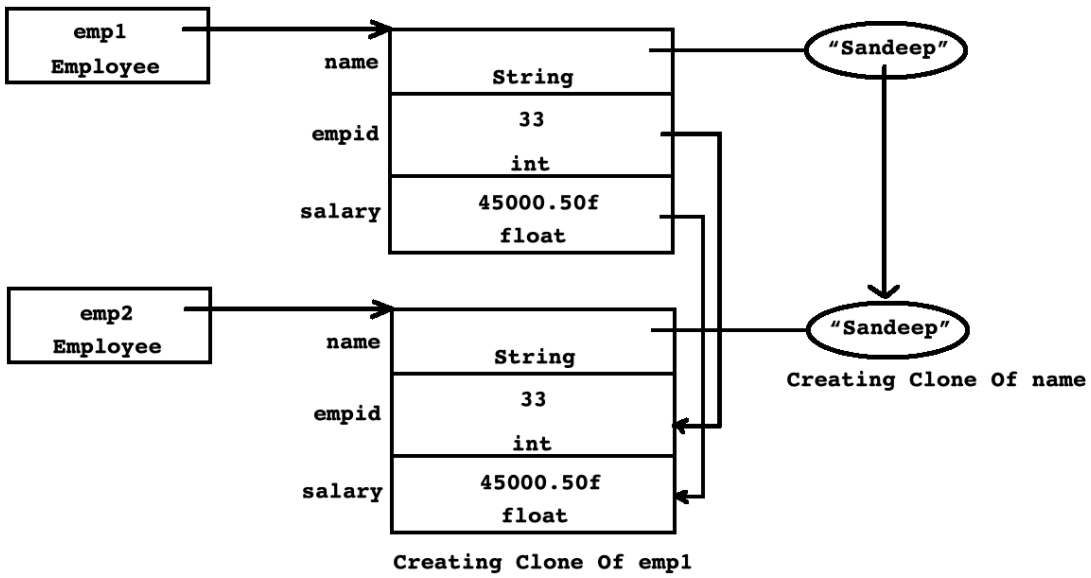
```
Employee emp1 = new Employee("Sandeep",33,45000.50f);
Employee emp2 = emp1; //Shallow Copy of reference
```



```
Employee emp1 = new Employee(“Sandeep”,33,45000.50f);
Employee emp2 = (Employee)emp1.clone( );//Shallow Copy of Instance
```

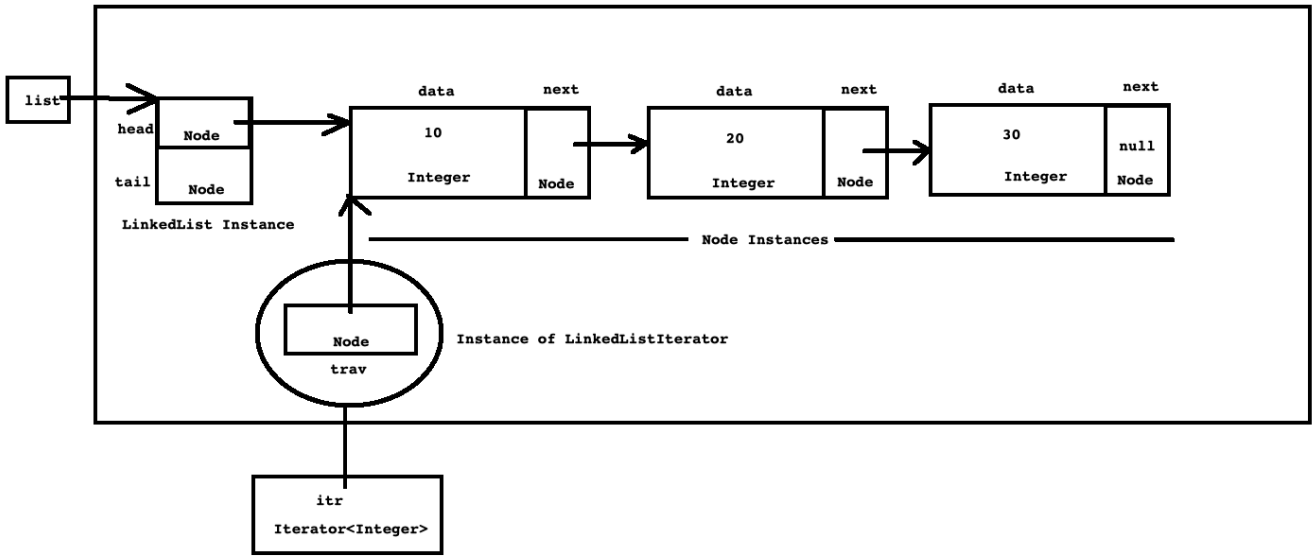


Deep Copy Of Instance



Creating Clone Of name

Creating Clone Of emp1



- **If we use any instance as a pointer/reference then it is called smart pointer.**
- **Iterator is a smart pointer which is used to traverse collection**

- **In for-each loop source must be:**
  1. **Either array**
  2. **Or instance of a class which has implemented java.lang.Iterable interface.**
- **“Iterator<T> iterator( )” is a method of Iterable interface.**
- **Iterator<E> is an interface declared in java.util package.**
- **Methods:**
  1. **boolean hasNext( )**
  2. **E next( )**
- **If any class implements Iterable interface then it is considered as traversible.**

demo

1. **Nested class**

```
//Top Level Class
class Outer{    //Outer.class
```

```
    //Nested Class
    class Inner{    //Outer$Inner.class

    }
}
public class Program {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
    }
    public static void main1(String[] args) {
        Outer out = new Outer();

        Outer.Inner in  = out.new Inner( ) ;

    }
}
```

- 1. Inner class
- accessing in outer class

```
//Top Level class
class Outer{
    private int num1 = 10;  //OK
    private static int num2 = 20;   //OK

    //Non static nested class / Inner class
    class Inner{
        private int num3 = 30;  //OK
        private static final int num4 = 40; //OK
    }

    public void print( ) {
        System.out.println("Num1    :   "+this.num1);
        System.out.println("Num2    :   "+Outer.num2);

        //System.out.println("Num3  :   "+num3); //NOT OK
        //System.out.println("Num4  :   "+num4); //NOT OK

        Inner in = new Inner();
        System.out.println("Num3    :   "+in.num3);
        System.out.println("Num4    :   "+Inner.num4);
    }
}
public class Program {
    public static void main(String[] args) {
        Outer out = new Outer();
        out.print();
    }
}
```

- accessing in inner class

```
//Top Level class
class Outer{
    private int num1 = 10;  //OK
    private static int num2 = 20;   //OK

    //Non static nested class / Inner class
    class Inner{
        private int num3 = 30;  //OK
        private static final int num4 = 40; //OK

        public void print( ) {
            System.out.println("Num1    :   "+num1);    //OK
            System.out.println("Num2    :   "+num2);    //Ok

            System.out.println("Num3    :   "+this.num3);
            System.out.println("Num4    :   "+Inner.num4);
        }
    }
}
public class Program {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.print( );
    }
}
```

- accessing outer class by method in inner class

```
class Outer{
    private int num1 = 10;
    public class Inner{

        private int num1 = 20;

        public void print() {

            int num1 = 30;

            System.out.println("Num1 : "  + Outer.this.num1);
            System.out.println("Num1 : "  + this.num1);
            System.out.println("Num1 : "  + num1);
        }
    }
}
public class Program {
    public static void main(String[] args) {
```

```
        Outer.Inner in = new Outer().new Inner();
        in.print();
    }

}
```

- 2. Static Inner Class

```
class Outer{ // Puter.class
    //static nested class


    static class Inner{ //



    }
}
public class Program {
    public static void main(String[] args) {

        Outer out = new Outer();

        Outer.Inner in = new Outer.Inner();


    }

}
```

- accessing static inner class member in outer class

```
//Top level class
class Outer{
    private int num1 = 10;
    private static int num2 = 20;

    static class Inner{
        private int num3 = 30;//ok
        private static int num4 = 40; //ok
    }



    public void print() {
        System.out.println("Num1 : " + this.num1);
        System.out.println("NUm2 : " + Outer.num2);
```

```
        Inner in = new Inner(); // ok
        System.out.println("NUm3  :" + in.num3);
        System.out.println("NUm4  :" + Inner.num4);
    }

}
```

- accessing by inner class method

```
//Top level class
class Outer{
    private int num1 = 10;
    private static int num2 = 20;

    static class Inner{
        private int num3 = 30;//ok
        private static int num4 = 40; //ok

        public void print() {
            Outer o = new Outer();

            System.out.println("Num1 : " + o.num1);
            System.out.println("NUm2 : " + Outer.num2);

            Inner in = new Inner(); // ok
            System.out.println("NUm3  :" + in.num3);
            System.out.println("NUm4  :" + Inner.num4);
        }

    }
}
public class Program {
    public static void main(String[] args) {

        Outer.Inner in = new Outer.Inner();

        in.print();
    }

}
```

2.**Local Class**

```
public static void main(String[] args) {
        // method local inner class
        // method local anonymous inner class
        //Method Local Inner Class
```

```
        // it is non static
        // outlisde methodw e cannot use Local class
         class MethodInnerClass
         {

          }

         MethodInnerClass c1 = new MethodInnerClass();
         System.out.println(c1.toString());

    }
```

- 1. Method Local Inner Class

```
public static void main(String[] args) {
        // method local inner class
        // method local anonymous inner class
        //Method Local Inner Class
        // it is non static
        // outlisde methodw e cannot use Local class
         class Complex{  // Program$1Complex file name for class

            private int real;
            private int imag;
            public Complex(int real, int imag) {
                super();
                this.real = real;
                this.imag = imag;
            }

            public Complex() {
                this(0,0);
            }

            @Override
            public String toString() {
                return "Complex [real=" + real + ", imag=" + imag + "]";
            }

        }

         Complex c1 = new Complex(10, 20);
         System.out.println(c1.toString());

    }
```

- 2. Method ANonymous Inner Class

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

        //Object obj; // obj => reference
        //new Object(); // => instance w/o refernce => Anonymous Instance
        // Object obj = new Object(); // Instance with reference

        //Method Local Anonymous Inner Class
        // only inside method ,we can declare this

        Object obj = new Object() {

            private String msg = "Inside anonymous inner class";

            @Override
            public String toString() {

                return this.msg;
            }

        }; //Program$1.class // anonymous inner class



        String msg = obj.toString();

        System.out.println(msg);



    }
```

- example of anonymous local method class

```
    public static void main1(String[] args) {
            // anonymous class doesnt have a constructor, even impilict
        Shape sh = new Shape() {

            private float radius = 10;

            @Override
            public void calculateArea() {
                this.area = (float) (Math.PI * Math.pow(radius, 2));

            }
        };

        sh.calculateArea();
        System.out.println("Area : " + sh.getArea());
```

```
    }
```

3. **Interface**

- set of rules are called specification
- specification is also called standard /(i.e interface , which can be implemented)
- example :
- 1. JVM Specification (i.e interface)
- 2. JVM specification implementation (interface implementation)
- 3. JVM runtime instance (instance of specification implementation)
- interface helps to build trust, reduce vendor maintainability
- 1. in java inteface can contain
  - 1. nested type
  - 2. fields
  - 3. static method
  - 4. abstract method
  - 5. default method
- 2. Demo : Interface basic
  - by deault fields in interface are : public static final int num;
  - by deault methods in interface are public abstract void point();
  - in Interface implementation inheritence - it is mandetory to override abstract method of interface , or class becomes abstract
  - we cannot create instance of abstract class and interface, but can create reference for it

```
//Standard : ISI
interface A{
    int num = 1;
    //by deault fields in interface are : public static final int num;

    void point();
    //by deault methods in interface are public abstract void point();

}

//Service Provider
class B implements A
{
    //Interface implementation inheritence
         // it is mandetory to override abstract method of interface , or
class becomes abstract

    @Override
    public void point() {
```

```
        System.out.println("Number : " + A.num);
    }
}

// Service Consumer
public class Program { //Program.class
    public static void main(String[] args) {

        System.out.println(A.num);

        A a = new B(); //upcasting : rules of A are followed by B
    }
    public static void main2(String[] args) {

        System.out.println(A.num);

        B b = new B(); //upcasting : rules of A are followed by B

    }
    public static void main1(String[] args) {

        System.out.println(A.num);

        //A a = new A();
        // we cannot create instance of abstract class and interface, but
can create reference for it

        A a = null;
    }
    }
```

- 3. **different combination of implements and extends keyword on Class and Interface **

> I1,I2,I3 => interface C1,C2,C3 => Classes

- 1. I2 implements I1 // not ok
- 2. I2 extends I1 //ok
- 3. I3 extends I1,I2 // ok : Multiple interface inheri
- 4. I1 extedns C1 // not ok
- 5. C1 extends I1 // not ok
- 6. C1 impleemnts I1 // ok
- 7. C1 implements I1,I2 // ojk : mutiple interface implementation inheritence(shown by dotted lines)
- 8. C2 implemnts C1 // not ok
- 9. C2 extends C1 ; // ok
- 10. C3 extends C1,C2 ; //not ok
- 11. C2 implements I1 extends C12 ; // not ok
- 12. C2 extends C1 Implements I1; //ok

- 4. demo on accessing member from interface

```
//Standard : ISI
interface A{

    int num1 = 10;
    int num4 = 40;
    int num5 = 70;
}

interface B
{
    int num2 = 20;
    int num4 = 50;
    int num5 = 80;
}


interface C extends A, B{
    int num3 = 30;
    int num4 = 60;
}


public class Program { //Program.class
    public static void main(String[] args) {

        System.out.println("num1 : " + A.num1);
        System.out.println("num1 : " + C.num1);
        System.out.println("num2 : " + B.num2);
        System.out.println("num2 : " + C.num2);

        System.out.println("num3 : " + C.num3);

        System.out.println("num4 : " + A.num4); // 40
        System.out.println("num4 : " + B.num4);// 50

        System.out.println("num4 : " + C.num4); // 60


        System.out.println("num5 : " + A.num5);

        System.out.println("num5 : " + B.num5);
    //  System.out.println("num5 : " + C.num5); // ambiguity error // if
same variable in multiple interfaces, implemented
    }

    }
```

- 5. interface method can be

- 1. method same name
- here if method name is same, we can override it only once in sub class
- 2. method different name
- we need to override each method name once

```
//Standard : ISI
interface A{

    void f1();
    void print();
}

interface B
{
    void f2();
    void print();
}


abstract class C{
    abstract void f3();
  public abstract void print();
}

class D extends C implements A,B{

    @Override
    public void f2() {
        System.out.println("D.B.f2");

    }

    @Override
    public void f1() {

        System.out.println("D.A.f1");
    }

    @Override
    void f3() {
        System.out.println("D.C.f3");

    }

    @Override
    public
    void print() {
        System.out.println("Inside print");

    }
```

```
}

public class Program { //Program.class
    public static void main1(String[] args) {

        A a = new D();

        a.f1();
        a.print();

        B b = new D();

        b.f2();

        b.print();
        C c = new D();

        c.f3();

        c.print();

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

        D d = new D();

        // if method name same, we can over ride only once

    }

    }
```

- 6. when to use Abstract class and interface ?
- 1. when state is involved between super and sub classes, super type must be abstract
  - as we can declare field
  - if there is(is - a) relationship , declare class abstract
  - ex . shape involve area, in rect, circle
- 2. when state is not involved between classes
  - and same method design must be implemented in all sub classes
    - then (can - do) relationship exist between classes, so make a interface where method is declared , and use interface implementation inheritence

    > C implements I

  - we cannot declare field , i.e except final static
- 7. **INTERFACE**
    - interface can extend one or more than one interface
    - insterface method are by default abstract

- we cannot define constructor inside interface
- if state is not involved in Super type it should be interface
- its name end with able, generally it is a interface,as it is used to implement some compulsory logic
- demo : using interface we can group object of unrelated classes together

```
Printable[] arr = new Printable[3];

arr[0] = new Date ();
arr[1] = new Stack ();
arr[2] = new String ();
```

```
 - 8. Abstract class
 - Abstract class  can extend only one abstract/concrete class
 - abstract class may/may not contain abstract method
 - we can define constructor inside abstract class

 - demo :  using abstract we can group object of related classes together
```

```
Shape[] arr = new Shape[3];

arr[0] = new Rectangle ();
arr[1] = new Circle ();
arr[2] = new Triangle ();
```

4. **commonly used interfaces in java**

- java.lang
    - interface Cloneable
    - Interface Iterable
    - Interface Comparable
- java.io
    - Interface Serializable
- java.util

5. **Shallow copy** is process of copying content of object into another object as it is

- also called bitwise copy

- 1. shalow copy of references

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

    Date dt1 = new Date(6,11,2020);

    Date dt2 = dt1; // Shallow copy of references
    dt1.setDay(1);
    System.out.println(dt1.toString());



    System.out.println(dt2.toString());


    }
```

- 2. shallow copy of instances
- copy object itself , not only its content, references of fields and method too, so having same references.
-
- to create shallow copy of new instance from existing instance use clone method , object class
- clone method of object class is not accessible outside date class, so we need to override clone method for use in implementing/ Program class

```
protected Object clone()
                throws CloneNotSupportedException
```

- overriding clone method in java

```
class Date implements Cloneable{

    @Override
    public String toString() {
        return "Date [day=" + day + ", month=" + month + ", year=" + year +
"]";
    }

    public Date clone() throws CloneNotSupportedException {

         Date other = (Date) super.clone(); // creating new instance from
existing instance
        return other;
    }

}
// co-varient : same or sub type
```

```
public class Program { //Program.class
    public static void main(String[] args) {

        Date dt1 = new Date(6,11,2020);

        Date dt2;
        try {
            dt2 = dt1.clone();
            if(dt1 == dt2)
            {
                System.out.println("equal");
            }else
            {
                System.out.println(" not equal");
            }
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }

    }
```

- to create new instance of object employeee using clone method
- editing values of other employee(emp2)

```
      public Employee clone( ) throws CloneNotSupportedException {
        Employee other = (Employee) super.clone();
        other.name = new String( this.name);
        other.joindate = this.joindate.clone();
        return other;

        main()
        {
            Employee emp2 = emp1.clone();   //Shallow Copy of references;

        }
    }
```

6. which are marker/tagging interface :
    - an interface which do not contain any member
    - generate metadata fo JVM

- Example :
    - lang.clonable
    - util.RandomAccess in
    - util.EventListener
    - rmi.RMI interface

- util.Remote interface
- io.Serializable

7. Deep copy

- copy object itself , not only its content, but creating new references of fields and method too .

8. **Using Comparable interface**

- 1. to perform array operation
- for reference, in case of array

```
public static void main1(String[] args) {
        int[] arr = { 5,1,4,2,3};
        Arrays.sort( arr );
        System.out.println(Arrays.toString(arr));
    }
```

- 2. **use Array operation like sort on Object Array.**
- need to implement Comparable interface on Object class, to override compareTo method
- comparable interface is for comparing same kind of object
- demo : on comparable interference needed to override compareto method to sort array of object

```
class Employee implements Comparable<Employee>{
    private String name;
    private int empid;
    private float salary;
    //Employee this;
    //Employee other
    @Override // option 1
    public int compareTo(Employee other) {
        return this.name.compareTo(other.name);
    }
    @Override // option 2
    public int compareTo(Employee other) {
        return (int) (this.salary - other.salary);
    }
}
public class Program {
    public static Employee[] getEmployees( ) {
        Employee[] arr = new Employee[ 5 ];
        arr[ 0 ] = new Employee("Prashant", 13, 50000);
        arr[ 1 ] = new Employee("Amol", 11, 40000);
        arr[ 2 ] = new Employee("Rupesh", 15, 30000);
        arr[ 3 ] = new Employee("Umesh", 14, 20000);
        arr[ 4 ] = new Employee("Mukesh", 12, 10000);
        return arr;
    }
```

```
    private static void print(Employee[] arr) {
        if( arr != null ) {
            for( Employee emp : arr )
                System.out.println(emp.toString());
            System.out.println();
        }
    }
    public static void main(String[] args) {
        Employee[] arr = Program.getEmployees();
        Program.print( arr );

        Arrays.sort( arr );
        Program.print( arr );
    }


}
```

- use Array operation like sort on Object Array
- for same and different kind of object

9. **Using Comparator interface**
    1. comparator on employee class to sort of employee array

```
class SortByName implements Comparator<Employee>
{
    @Override
    public int compare(Employee emp1, Employee emp2) {
        return emp1.getName().compareTo(emp2.getName());
    }

}
class SortById implements Comparator<Employee>
{

    @Override
    public int compare(Employee o1, Employee o2) {
        // TODO Auto-generated method stub
        return o1.getEmpid() - o2.getEmpid();
    }

}

public class Program { // Program.class

    public static Employee[] getEMployees() {
        return arr;
    }
```

```
    public static void print(Employee[] arr)
    {
    }

    public static void main(String[] args) {

        Employee[] arr = Program.getEMployees();

        Program.print(arr);

        Comparator<Employee> comparator = null;
        comparator = new SortByName();

        Arrays.sort(arr, comparator);
        System.out.println("using comparator, by name");
        Program.print(arr);

        comparator = new SortById();
        Arrays.sort(arr, comparator);
        System.out.println("using comparator, byid");
        Program.print(arr);
    }
```

2. improving code in print method

```
public static void print(Employee[] arr,Comparator<Employee> comparator)
    {
        if(arr != null)
        {
            Arrays.sort(arr, comparator);

            for(Employee emp : arr)
            {
                System.out.println(emp.toString());
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {

        Employee[] arr = Program.getEMployees();

    System.out.println();

    System.out.println("comparator by name");
    Program.print(arr, new SortByName());
    System.out.println("comparator by id");
    Program.print(arr, new SortById());
    System.out.println("comparator by salary");
    Program.print(arr, new SortBySalary());
```

```
    }
```

3. 'demo on comparator interface, to compare sub classes of Person class, i.e diffrent class Objects

```
  class Person{

    private String name;
     }

class Employee extends Person{

    private int empid;
}

class Student extends Person{
    private int rollNumber;
}

class SortByName implements Comparator<Person>
{
    @Override
    public int compare(Person p1, Person p2) {

        return p1.getName().compareTo(p2.getName());
    }
}

class SortById implements Comparator<Person>
{
    @Override
    public int compare(Person p1, Person p2) {

        if(p1 instanceof Employee && p2 instanceof Employee)
        {
            Employee emp1 = (Employee) p1;
            Employee emp2 = (Employee) p2;

            return emp1.getEmpid() - emp2.getEmpid();
        } else if(p1 instanceof Student && p2 instanceof Student)
        {
            Student s1 = (Student) p1;
            Student s2 = (Student) p2;

            return  s1.getRollNumber() - s2.getRollNumber();
        } else if(p1 instanceof Student && p2 instanceof Employee)
        {
            Student s1 = (Student) p1;
            Employee emp2 = (Employee) p2;
```

```
            return s1.getRollNumber() - emp2.getEmpid();
        } else {

            Student s2 = (Student) p2;
            Employee emp1 = (Employee) p1;

            return emp1.getEmpid() - s2.getRollNumber();
        }
    }
}

public class Program { // Program.class

    public static Person[] getPerson() {
        Person[] arr = new Person[5];
        arr[ 0 ] = new Employee("Prashant", 13);
        arr[ 1 ] = new Student("Amol", 11);
        arr[ 2 ] = new Employee("Rupesh", 15);
        arr[ 3 ] = new Employee("Umesh", 14);
        arr[ 4 ] = new Student("Mukesh", 12);
        return arr;
    }
    public static void print(Person[] arr)
    {
        if(arr != null)
        {
            for(Person p : arr)
            {
                System.out.println(p.toString());
            }
            System.out.println();
        }
    }
    public static void main(String[] args) {

        Person[] arr = Program.getPerson();
    System.out.println();

    System.out.println("comparator by name");
    Arrays.sort(arr,new SortByName());

    Program.print(arr);
    System.out.println("comparator by id");
    Arrays.sort(arr, new SortById());
     Program.print(arr);
    }
}
```

Day 11

to read

slides

+ Metadata
  - Data about data or data which describes other data is called metadata.

+ Metadata of interface

1. **What is name of the interface**
2. **In which package it is declared**
3. **Which is super interface of the interface**
4. **Which annotations has been used in interface**
5. **Which types are declared inside interface**

+ Metadata of Class

1. **What is name of the class**
2. **In which package it is declared**
3. **Which is super class of the class**
4. **Which interface it has implemented**
5. **Which annotations has been used on the class**
6. **Which modifiers used on the class(public abstract, final)**
7. **Which members are declared inside class**

+ Metadata of Field

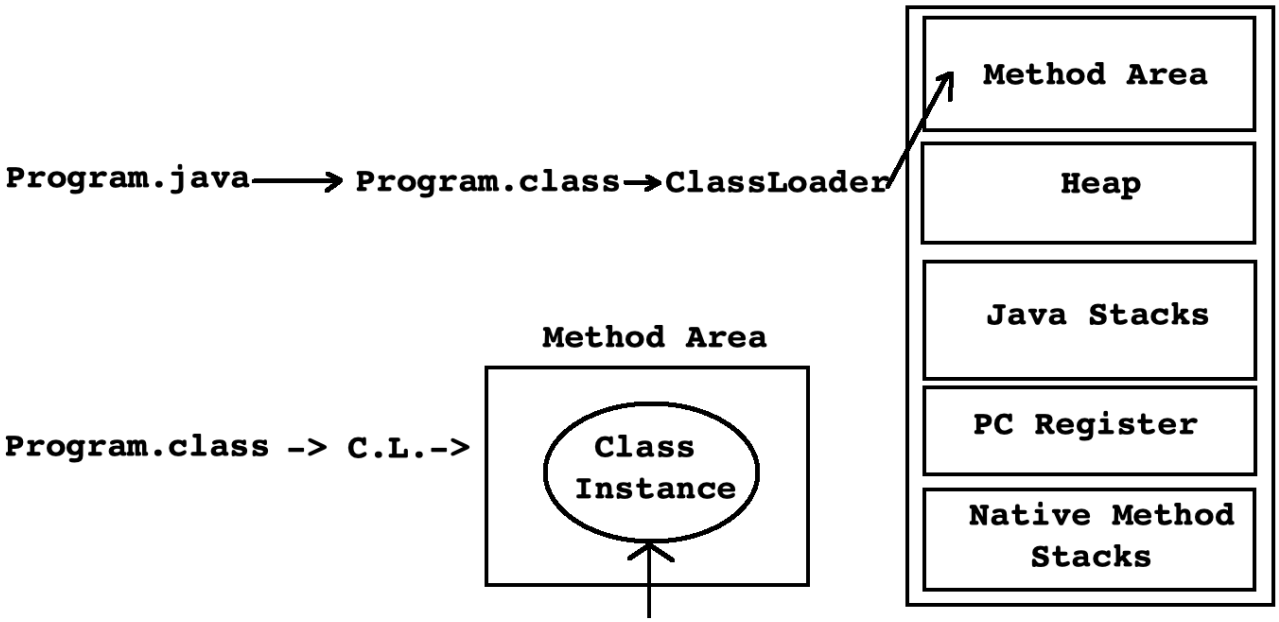
1. **What is name of the field**
2. **Which is modifier of field**
3. **What is the type of field**
4. **Which annotations are used on field**
5. **Whether field is inherited / declared only field**

+ Metadata of method

1. **What is name of the method**
2. **Which is modifier of method**
3. **Which is return type of method**
4. **What is parameter information of method**
5. **Which exception method throws**
6. **Which annotations are used on method**
7. **Whether method is inherited, declared only or overriden method.**

+ Application of metadata
  - After compilation, java compiler generate bytecode and metadata. Due to this metadata, in java, there is no need to include header file in a program.
  - To display information of any type inside intelligence window IDE(e.g eclipse) implicitly read metadata from .class file.
  - Metadata helps JVM to create clone of the instance, serialize instance and marshal/unmarshal instance.
  - Metadata helps garbage collector to keep track of life time of object.
  - Compiler generate metadata implicitly and with the help of annotation programmer can generate it explicitly.

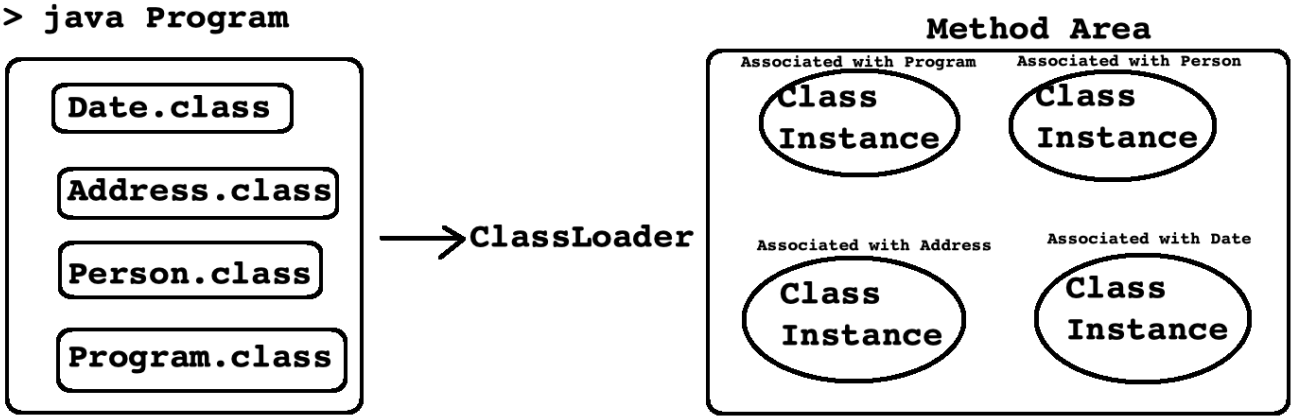
+ Reflection
  - It is a java langue feature which allows us:
    1. To analyze and process metadata
    2. To access private members of the class outside class.
    3. To manage behavior of the application at runtime.
    4. To map object-relation/table.
  - Application
    1. To read metadata from .class file javap and intellisense window implicitly use reflection.
    2. To access private members outside class debugger implicitly use reflection.
    3. To implement drag and drop feature, IDE implicitly use reflection
  - To use reflection, we need to use types declared in:
    1. java.lang and
    2. Java.lanf.reflect package
  - Reading assignment : explore java.lang.Class class
  - Book : Reflection In Action




```
class Date{ }
class Adddres{}
class Person
{
 String name;
 Date dob;
 Address currAddress;
}
```

```
class Program{
 p.s.v.m( String[] args ){
  Date dt = new Date();
  Address a = new Address();
  Person p = new Person( );
 }
}
```

```
-> javac Program.java
//Output : Date.class Address.class Person.class Program.class
-> java Program
```



+ Reflection
  - Class is a final class declared in java.lang package.
  - Class has no public constructor. Instead Class objects are constructed automatically by the Java Virtual Machine.
  - Instances of the class Class represent classes and interfaces in a running Java application.
  - The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the keyword void are also represented as Class objects.
  - Class c = new Class( ); //Not OK

+ Methods of java.lang.Class
  1. public static Class<?> forName(String className)
  2. public T newInstance()
  3. XXX getXXX( )

- How to get reference of instance of java.lang.Class class?
- "public native final Class<?> getClass( )" is a method of java.lang.Object class.
- getClass() method returns reference of instance Class class associated with current instance.

1. Using getClass() method:
```
Integer i = new Integer(0);
Class<?> c = i.getClass();
```
2. Using ".class" syntax
```
Class<?> c =  Number.class;
```
3. Using "Class.forName( )" method
```
String className = "java.io.File";
Class<?> c = Class.forName(className);
```

demo

1. Meta data

- data which describes other data, is metadata
- use of Meta data :

- 1. to implement show help/intellisense
- 2. clone method uses, CLonable marker interface ,which creates meta data ,
  - so meta data helps jvm to
    - create clone of instance
- 3. marshalling (uses Marshall interface)
    1. taking a java object converting it into binary data , to send to a network
    - non marshalling
    1. taking a marshallled binary data of java object, converting it back to java object
- 4. garbage collector takes help of meta data, to get to the resource to be removed , with zero reference
- 5. Serializing

2. Reflection

3. using reflection we can explore meta data , demos

```
```

Day12

to read

1. what is iterable and iterator

2. can we declare static memeber function as virtual and ?

3. Effective java book

4. 4 kind of copy: defensive copy, lazy copy,shallow copy, deep copy

5. assignment on pojo

6. > https://docs.oracle.com/javase/8/docs/api/

7. 4 classes vector, sub clas stack, hash table and sub class property, are by default synchronised for multi threading

8. difference between vectore and arraylist,

- arraylist is asyncronised, capacity increase by half
  - not a legacy class
- vector is synchronised , capicity doubles,
  - we can iterate vector , using itteator,list itertor, and enumerator
  - enumerator is used to traverse collection in forward direction,
    - in which we cant add, or remove elements , use iterator for add and remove in preference of Enumerator.
- vector is a legacy class

9. Dynamic Proxy (for hibernate interface)

- invocationHandler

10. what is difference between list(allow dublicate element) and set(do not allow dublicate element)?

- for set coll need to used iterator only not listiterator

11. hashcode is logical integer number, created by processing state of object .

slides

1. predicate,supplier,consumer function , unaryOperator interface ,in java.util.function

+ Annotation Type
- **An interface which do not contain Any members is called called marker/tagging interface.**
- **Example:**
  1. **java.lang.Cloneable**
  2. **java.io.Serializable**
  3. **java.util.RandomAccess**
- **Marker interfaces are used to generate metadata for the JVM.**
- **Prior to JDK 1.5 marker interfaces were used to generate metadata. JDK 1.5 and onwards, programmer can use annotation to attach metadata.**
- **Annotation Type / annotation is a java language feature, which allows us to attach metadata with source code.**

+ Application of annotations
1. **Annotations can be used to generate error as well to suppress warning in other words it is used to give information to compiler.**
2. **In Java EE, annotations are used to ORM, for generating some code , xml files or to reduce use XML file.**

+ Types of annotation
1. **Marker annotation**
2. **Single valued annotation**
3. **Multi valued annotation**

- Annotation representation
@Override, @SupressWarnings etc.

- Marker Annotation
- **Example:**
  @Override, @FunctionalInterface, @Entity etc.
- **Annotation without any element is called marker annotation.**

- Single Valued Annotation
- **Example:**
  @Table(value=“employees”)
  @SuppressWarnings(value = "serial")
  @Column(name=“emp_id”)
- **An annotation, which is having single element is called single valued annotation.**

- Multi valued annotation
  - Example
    - @Column(name="emp_name", columnDefinition="VARCHAR(100)")
    - @Author(name="Java Notes",date="22/09/2020")
    - @webServlet(urlPatterns="", initParams = "")
  - If annotation is having multiple elements then it is called multi valued annotation.

+ Where we can use annotation?
  1. TYPE(class,interface)
  2. FIELD
  3. METHOD
  4. PARAMETER
  5. CONSTRUCTOR
  6. LOCAL_VARIABLE
  7. ANNOTATION_TYPE
  8. PACKAGE
  9. TYPE_PARAMETER
  10. TYPE_USE

+ Annotations declared in java.lang package
  1. Deprecated(JDK 1.5)
  2. Override(JDK 1.5)
  3. SuppressWarnings(JDK 1.5)
  4. SafeVarargs(JDK 1.7)
  5. FunctionalInterface(JDK 1.8)

+ RetentionPolicy
  - It is enum declared in java.lang.Annotation Package
  - Enum constants:
    1. SOURCE
    2. CLASS
    3. RUNTIME

+ ElementType

- It is enum declared in java.lang.annotation package

- Enum constants :

TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR,
LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE, TYPE_PARAMETER,
TYPE USE

+ Annotations declared in java.lang.annotation package

1. **Target(JDK 1.5)**
2. **Retention(JDK 1.5)**
3. **Documented(JDK 1.5)**
4. **Inherited(JDK 1.5)**
5. **Native(JDK 1.8)**
6. **Repeatable(JDK 1.8)**

- Above annotations are used to implement custom annotations hence these are also called as meta annotations.

+ All the annotations extends java.lang.Annotation interface.

+ Rules to define user defined / custom annotation

1. **Use @ sign and interface keyword.**
2. **To use element with annotation specify element declaration inside annotation**

- Rules to specify annotation type element declaration:

1. **Element declaration can not take parameter.**
2. **Element declaration can not throws clause.**
3. **Return type in element declaration can should be primitive type, String, enum, class or array of any one of these.**

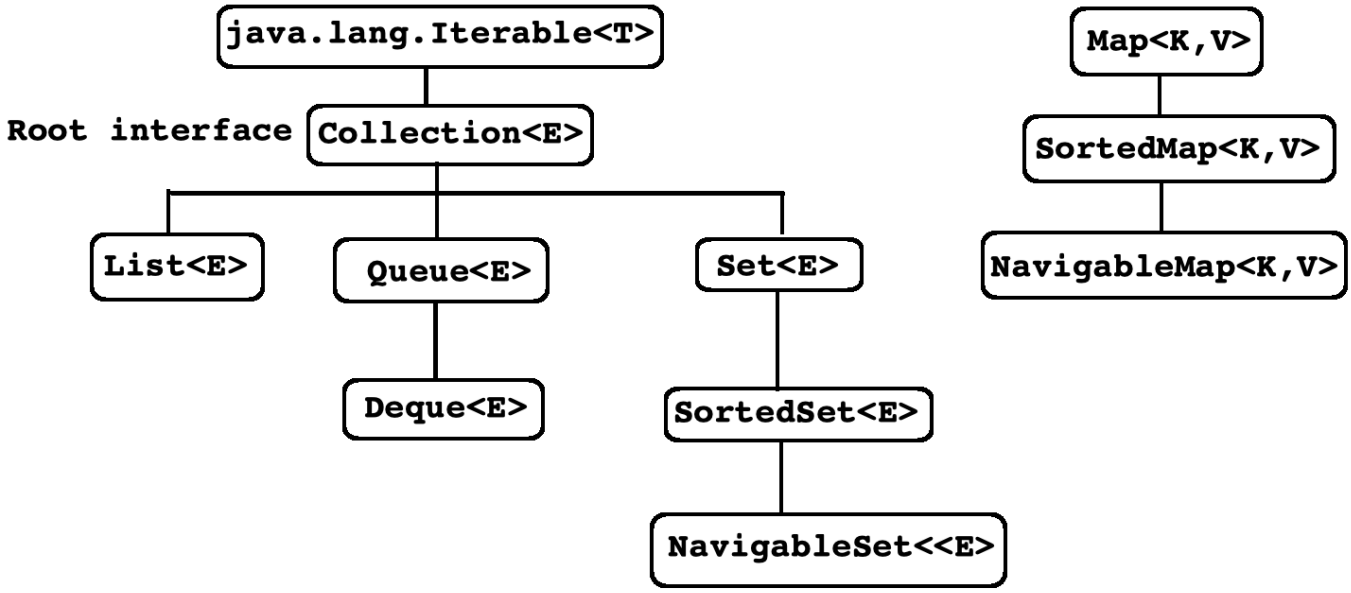
```
@Table(name="employees")
class Employee{
 @Column(name="emp_name", columnDefinition="VARCHAR(100)")
 private String name;
 @Id
 @Column(name="emp_id", columnDefinition="INT")
 int id;
 @Column(name="sal", columnDefinition="FLOAT")
 float salary;
};
```

```
class Program{
 p.s.v.m( String[] args ){
  //TODO
 }
}
```

=>

```
CREATE TABLE employees
(
 emp_name VARCHAR(100),
 emp_id INT PRIMARY KEY,
 sal FLOAT
);
```

+ Collection Framework
  - In Java, Data structure classes are called collection. classes.
  - Collection is an object which represent group of objects.
  - Framework is a library of reusable classes that we can use to develop application.
  - Collection framework is a library of reusable data structure classes that we can use to develop Java application[Standalone, Web, Mobile etc].
  - To use collection framework, we must import java.util package.
  - Data/value stored inside collection is called element.
  - Collection Framework Interface Hierarchy:
    + java.lang.Iterable<T>
      - java.util.Collection<E>
        * java.util.List<E>
        * java.util.Queue<E>
          # java.util.Deque<E>
        * java.util.Set<E>
          # java.util.SortedSet<E>
            $ java.util.NavigableSet<E>



+ Iterable
  - It is an I/F declared in java.lang package
  - Implementing this interface allows an object to be the target of the "for-each loop" statement.
  - Method:
    1. abstract Iterator<T> iterator()
    2. default Spliterator<T> spliterator()
    3. default void forEach(Consumer<? super T> action)

+ Collection<E>
  - The root interface in the collection hierarchy.
  - The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List.
  - This interface is a member of the Java Collections Framework.
  - It is introduced in JDK 1.2

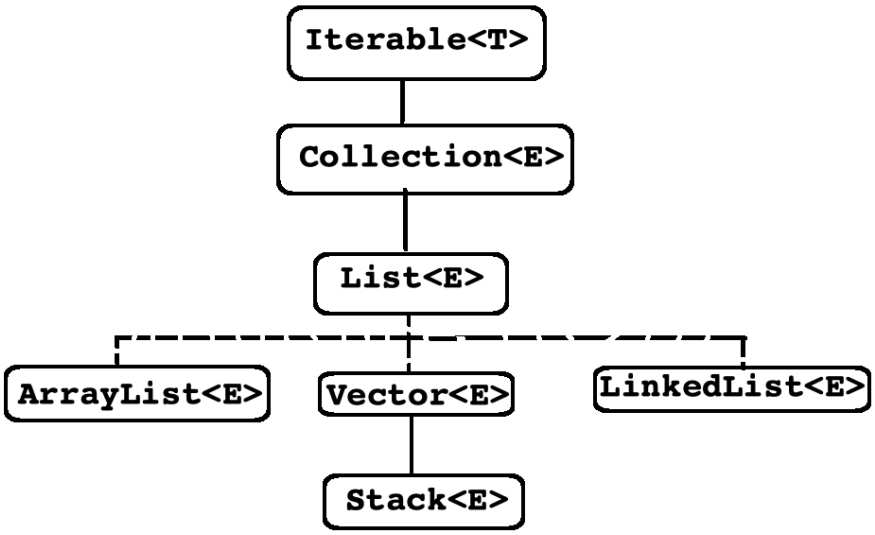
+ Abstract Methods of Collection Interface

1. public abstract boolean add(E);
2. public abstract boolean addAll(Collection<? extends E>);
3. public abstract void clear();
4. public abstract boolean contains(Object);
5. public abstract boolean containsAll(Collection<?>);
6. public abstract boolean remove(Object);
7. public abstract boolean removeAll(Collection<?>);
8. public abstract boolean retainAll(Collection<?>);
9. public abstract boolean isEmpty();
10. public abstract int size();
11. public abstract Object[] toArray();
12. public abstract <T> T[] toArray(T[]);

+ Default Methods Of Collection Interface

1. default Stream<E> stream()
2. default Stream<E> parallelStream()
3. default boolean removeIf(Predicate<? super E> filter)

+ List<E> Interface



List Collections = { ArrayList, Vector, Stack, LinkedList etc. }

+ List<E> Interface
  - It is sub interface of Collection<E> interface.
  - ArrayList<E>, Vector<E>, Stack<E>, LinkedList<E> etc. implements List<E> interface. These are also called as List collections.
  - List collections are ordered/sequential collections.
  - List collections can contain duplicate elements as well as null elements.
  - We can use integer index to access elements from List<E> collection.
  - We can traverse elements of List collection using Iterator as well as ListIterator
  - This interface is a member of the Java Collections Framework.
  - It is introduced in JDK 1.2
  - Hint : All the methods which is returning index, having index parameter or having index in their name are methods of List

+ Abstract Methods of List<E> Interface
  1. void add(int index, E element)
  2. boolean addAll(int index, Collection<? extends E> c)
  3. E get(int index)
  4. E set(int index, E element)
  5. int indexOf(Object o)
  6. int lastIndexOf(Object o)
  7. ListIterator<E> listIterator()
  8. ListIterator<E> listIterator(int index)
  9. E remove(int index)
  10. List<E> subList(int fromIndex, int toIndex)

+ Default methods of List<E> Interface
  1. default void sort(Comparator<? super E> c)
  2. default void replaceAll(UnaryOperator<E> operator)

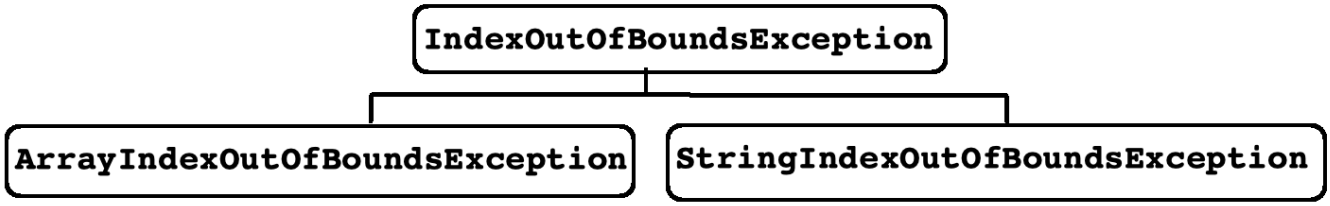
+ ArrayList<E>
  - It is a list collection which implements List<E>, RandomAccess, Cloneable and Serializable interface.
  - ArrayList is resizable unsynchronised collection.
  - Since it is List<E> collection it gets all the properties of List.
  - If we do not specify capacity then its default capacity is 10. If ArrayList<E> is full then it's capacity gets increased by half of its existing capacity.
  - This class is a member of the Java Collections Framework.
  - It is introduced in JDK 1.2
  - Hint : If we want to manage elements of non final type inside ArrayList then it should override equals method.

+ Methods of ArrayList Class:
1. public void ensureCapacity(int minCapacity)
2. protected void removeRange(int fromIndex, int toIndex)
3. public void trimToSize()

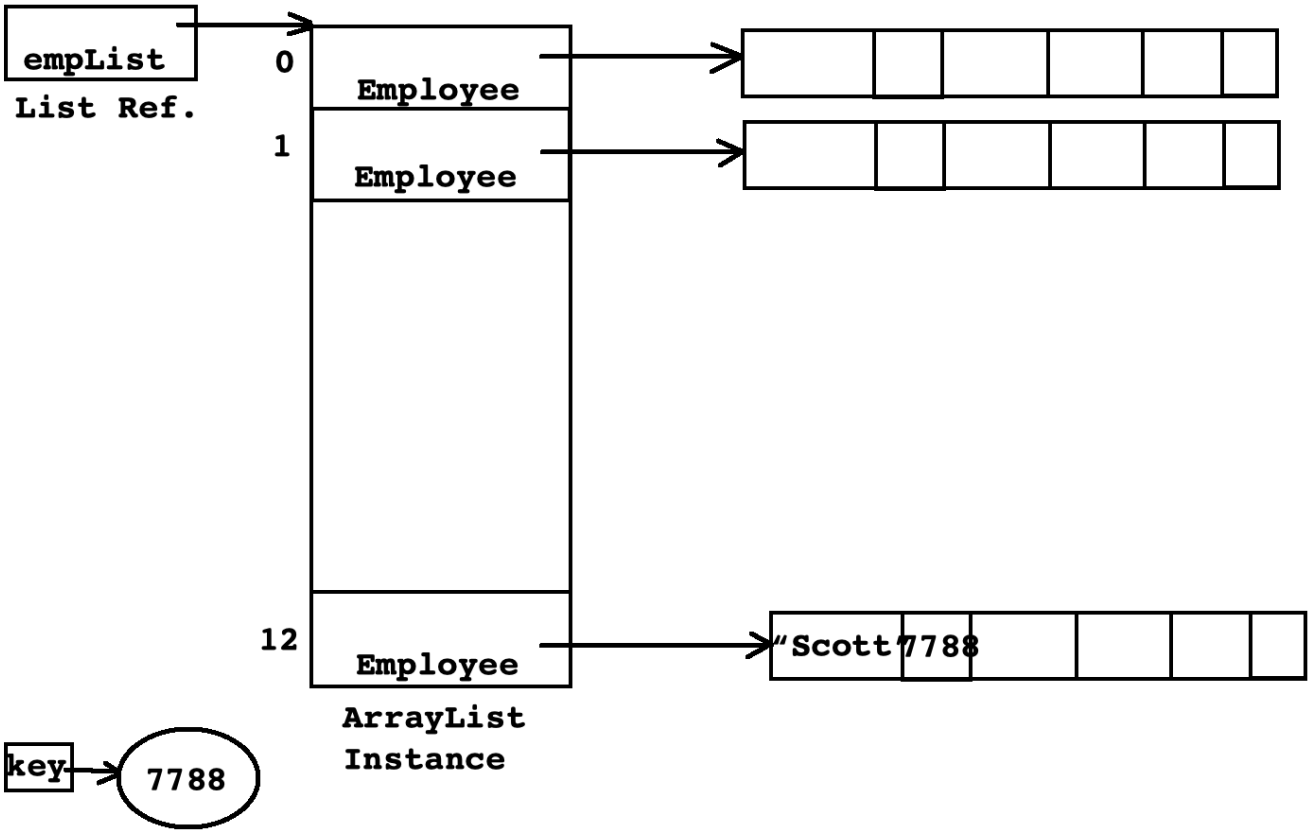
- In Java, Collection is not a group of objects rather it is group of references.

+ Constructor of ArrayList<E> class
  1. public ArrayList( )
     - ArrayList<Integer> list = new ArrayList<>( );
     - List<Integer> list = new ArrayList<>( );
     - Collection<Integer> list = new ArrayList<>( );
  2. Public ArrayList(int initialCapacity)
     - ArrayList<Integer> list = new ArrayList<>(7);
     - List<Integer> list = new ArrayList<>(7);
     - Collection<Integer> list = new ArrayList<>(7);
  3. ArrayList(Collection<? extends E > c)
     - Collection<Integer> c = Program.getCollection( );
     - ArrayList<Integer> list = new ArrayList<>(c);
     - List<Integer> list = new ArrayList<>(c);
     - Collection<Integer> list = new ArrayList<>(c);

```
ArrayList<Integer> list1 = new ArrayList<>(  );
list1.add(10);list1.add(20);
ArrayList<Integer> list2 = new ArrayList<>( list1 );
```



- using illegal index, if we try to access element from any List collection then list method throws IndexOutOfBoundsExceptior
- using illegal index, if we try to access element from any Array then JVM throws ArrayIndexOutOfBoundsException
- using illegal index, if we try to access character from String then String method throws StringIndexOutOfBoundsException



- In context of collection framework, 4 classes are by default synchronised / thread safe:

1. **Vector**
2. **Stack**
3. **Hashtable**
4. **Properties**

+ Vector

- **It is synchronised resizable array.**
- **It implements List<E>, RandomAccess, Cloneable, Serializable interface**
- **It is list collection.**
- **Default capacity of Vector is 10. If it is full then it gets double capacity.**
- **It is a member of Java's collection framework.**
- **It is introduced in JDK 1.0.**
- **This class is roughly equivalent to ArrayList, except that it is synchronized.**

+ What is difference between ArrayList and Vector?

1. **Synchronization**
2. **Capacity**
3. **Traversing**
4. **Introduction in JDK.**

+ Traversal
  1. public Enumeration<E> elements()
  2. public Iterator<E> iterator()
  3. public ListIterator<E> listIterator()

+ Enumeration
  - Enumeration<E> is an interface declared in java.util package.
  - Methods
    1. boolean hasMoreElements()
    2. E nextElement()
  - It is used to traverse collection only in forward direction.
  - During traversing, using Enumeration, we can not add, set or remove element from underlying collection.
  - It is introduced in JDK 1.0
  - NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.

+ Iterator
  - It is interface declared in java.util package.
  - Abstract Methods of Iterator
    1. boolean hasNext( );
    2. E next( );
  - Default Methods of Iterator
    1. Default void remove( )
    2. default void forEachRemaining(Consumer<? super E> action)
  - It is used to traverse collection in forward direction only.
  - During traversing, using iterator, we can not add or set element but we can remove element from underlying collection.
  - It is member of Java collection framework.
  - It is introduced in JDK 1.2
  - We can use it to traverse any collection which implements java.lang.Iterable interface.

+ ListIterator
  - It is sub interface of Iterator I/F which is declared in java.util package.
  - Methods of ListIterator
    1. void add(E e)
    2. void set(E e)
    3. void remove()
    4. boolean hasNext()
    5. E next()
    6. boolean hasPrevious()
    7. E previous()
    8. int nextIndex()
    9. int previousIndex()

- Using ListIterator, we can traverse only List Collections.
- We can use it to traverse collection in bidirectional.
- During traversing, using ListIterator, we can add, set as well as remove element from underlying collection.

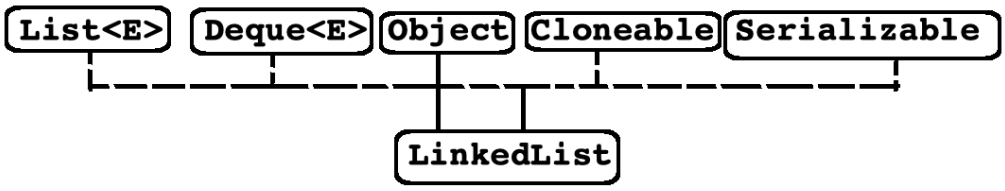
Q. What do you know about ConcurrentModificationException? Or
Q. Which are the types of Iterator?
Ans: 1. Fail-Fast Iterator 2. Not Fail-Fast/Fail-Safe Iterator

+ Fail-Fast Iterator
  - During traversing , using collection reference, we if try to modify state of that collection and if we get ConcurrentModificationException then such iterator is called Fail-Fast Iterator.

+ Not Fail-Fast / Fail-Safe Iterator.
  - During traversing , using collection reference, we if try to modify state of that collection and if we do not get ConcurrentModificationException then such iterator is called Not Fail-Fast / Fail-Safe Iterator.

+ Stack
  - It is sub class of Vector
  - It contains all the methods of Iterable, Collection, List, Vector and Object.
  - Stack Class specific methods:
    1. public boolean empty()
    2. public E push(E item)
    3. public E peek()
    4. public E pop()
    5. public int search(Object o)
  - It is synchronised collection.
  - If we want to perform LIFO operations then we should use Stack.

+ LinkedList
  - It is a List collection which implements List<E>, Deque<E>, Cloneable and Serializable interface.
  - Its implementation is based on Doubly LinkedList
  - This class is a member of the Java Collections Framework.
  - It is introduced in JDK 1.2
  - It is unsynchronised collection.
  - Using Collections.synchronizedList() method we can make it synchronised.
  - Instantiation:
    List<Integer> list = new LinkedList<>( );



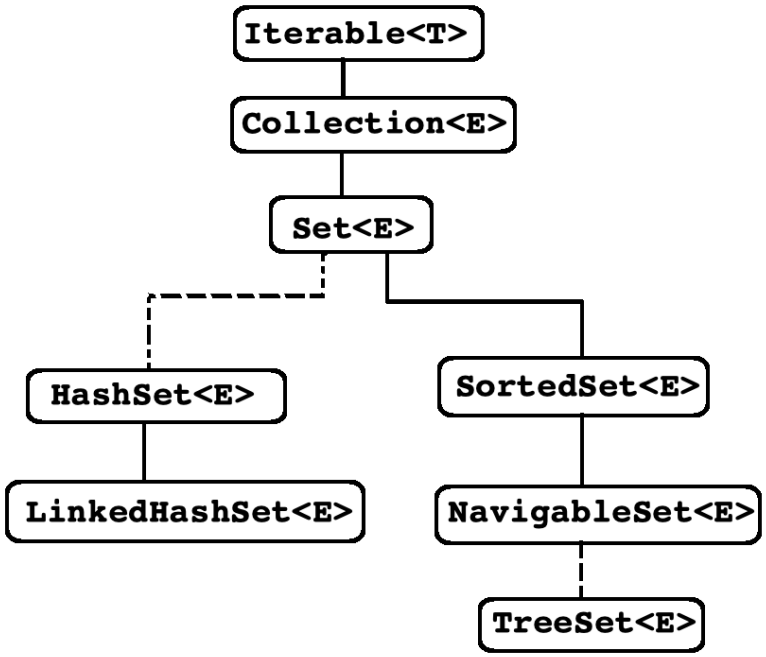
Q. What is the different between ArrayList and LinkedList?

+ Queue
- It is sub interface of Collection declared in java.util Package.

Summary of Methods

Throws exception	Returns special value	
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

+ Deque
  - It is sub interface of Queue interface.
  - The name deque is short for "double ended queue" and is usually pronounced "deck".
  - Summary of Deque methods
  - This interface is a member of the Java Collections Framework.
  - It is introduced in JDK 1.6
  - ArrayDeque, LinkedList etc implements Deque interface.



Set Collections = { HashSet<E>, LinkedHashSet<E>, TreeSet<E> }

+ Set Interface
- **It is a sub interface of Collection interface.**
- **Set collections do not contain duplicate elements.**
- **We can traverse elements of Set collection using iterator only.**
- **This interface is a member of Java's collection framework.**
- **It is introduced in JDK 1.2**
- **Methods of Collection and Set are same.**

+ TreeSet<E>
- **It is set Collection.**
- **It can not contain duplicate element as well as null element.**
- **It is sorted collection.**
- **Its implementation is based on TreeMap.**
- **It is unsynchronised collection. Using Collections.synchrozinedSortedSet(), we can make it synchronize**
- **This class is a member of the Java Collections Framework.**
- **It is introduced in JDK 1.2**
- **Hint : If we want to mange elements of non final class inside TreeSet then non final class should implement Comparable interface or we should provide comparator implementation.**

+ Searching
  - It is the process of finding location(index, address, reference) of an element inside collection.
  - Most commonly used searching techniques:
    1. Linear Search
    2. Binary Search
    3. Hashing

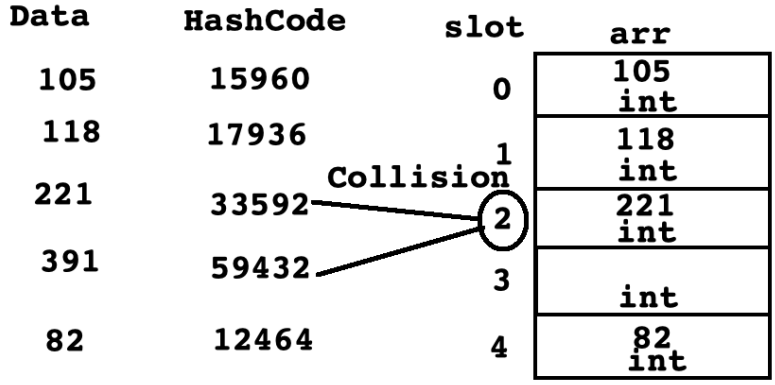
+ Linear Search
  - It is also called as sequential search
  - We can use it to search element inside sorted as well as unsorted collection.
  - It is efficient if collection contains less elements.
  - Drawback : If collection is having large amount of data then it takes more time to search element.
  - Example:
    10 20 30 40 50

    1 2 3 4 5 - no of comparisons
  - time required to search every element is different.

+ Binary Search
  - It used divide and conquer technique.
  - To use this algorithm, collection must be sorted.
  - In comparison with linear search, it takes less time to search element.
  - In binary search algorithm, time required to search every element is also different.

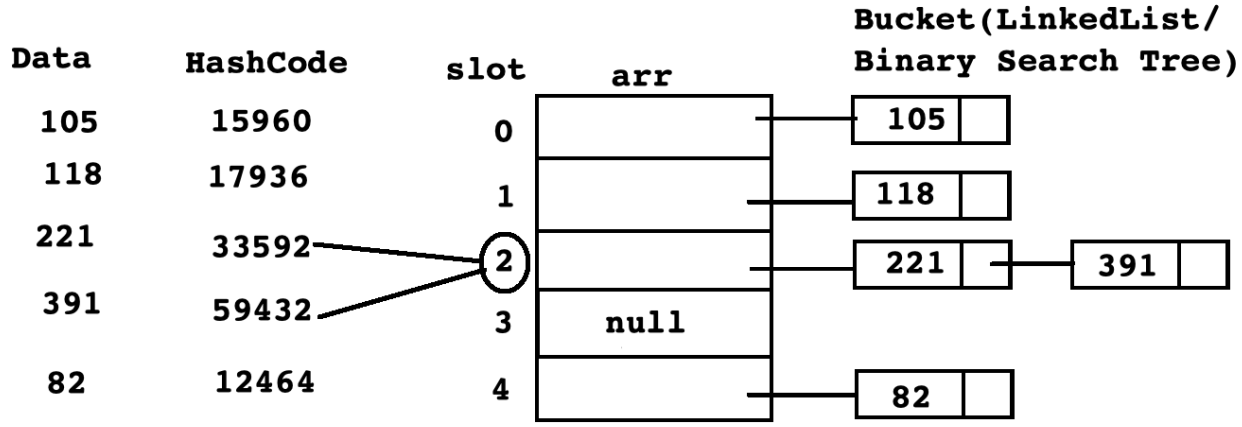
+ Hashing
  - If we want to search every element in constant time then we should use hashing. In other words to search element very fast we should use Hashing.
  - Hashing technique is based on hashcode.
  - Hashcode is not an index or address/reference.
  - HashCode is logical integer number that can be generated by processing state of the object.
  - A function/method, which generates hashcode is called hash function/method.
  - If state of object is same then we get same hashcode.
  - If state of object is different then we get diff hashcode.

- Hashcode is required to generate slot(index).
- We can access elements of array randomly
- 82 12464 4
- 221 33592 2
- 105 15960 0
- 118 17936 1
- 391 59432 2



- By processing hashcode of objects, if we get same slot then it is called collision.

+ Collision resolution techniques:
  1. Separate Chaining / Open Hashing
  2. Open Addressing / Close Hashing
     - Linear Probing
     - Quadratic probing
     - Double Hashing / Rehasing



- A collection maintained per slot is called bucket.

$$\text{Load Factor} = \frac{\text{Number of buckets}}{\text{Total number of elements}} \quad \frac{4}{5} = 0.80$$

- **If we want to manage elements of non final type inside any hashcode based collection then non final type should override equals and hashcode method.**
- **hashcode is native method of java.lang.Object class.**
- **syntax:**
  public int hashCode( );
- **This is typically implemented by converting the internal address of the object into an integer**
- **In case of same state, if we want to generate same hashcode then we should override hashcode method inside class.**

+ HashSet
  - **It is a Set collection.**
  - **It doesn't contain duplicate elements.**
  - **It can contain null element.**
  - **It is unordered collection.**
  - **Its implementation is based on Hashtable**
  - **It is a member of Java's collection framework**
  - **It is introduced in JDK 1.2**
  - **Hint : If we want to manage elements of non final class inside HashSet then it should override equals and hashcode method.**

+ LinkedHashSet
  - **It is sub class of HashSet<E> class.**
  - **Its implementation is based on LinkedList and Hashtable.**
  - **All the properties of HashSet and LinkedHashSet are same except HashSet is unordered and LinkedHashSet is Orderd/ collection.**

+ Dictionary
  - It is an abstract class declared in java.util package.
  - Hashtable is sub class of Dictionary class.
  - If we want to store data in key/value pair form them we should this class.
  - Methods

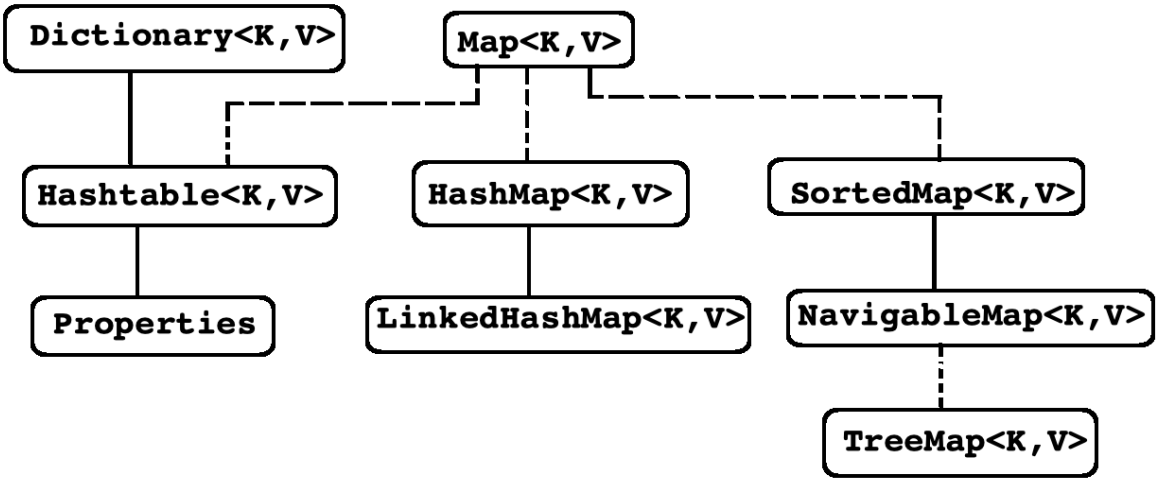
```
public abstract int size();
public abstract boolean isEmpty();
public abstract Enumeration<K> keys();
public abstract Enumeration<V> elements();
public abstract V get(Object key);
public abstract V put(K key, V value);
public abstract V remove(Object key);
```

  - Instantiation

```
Dictionary<Integer, String> d = null;
d = new Hashtable<>( );
```

  - NOTE: This class is obsolete. New implementations should implement the Map interface, rather than extending this class.

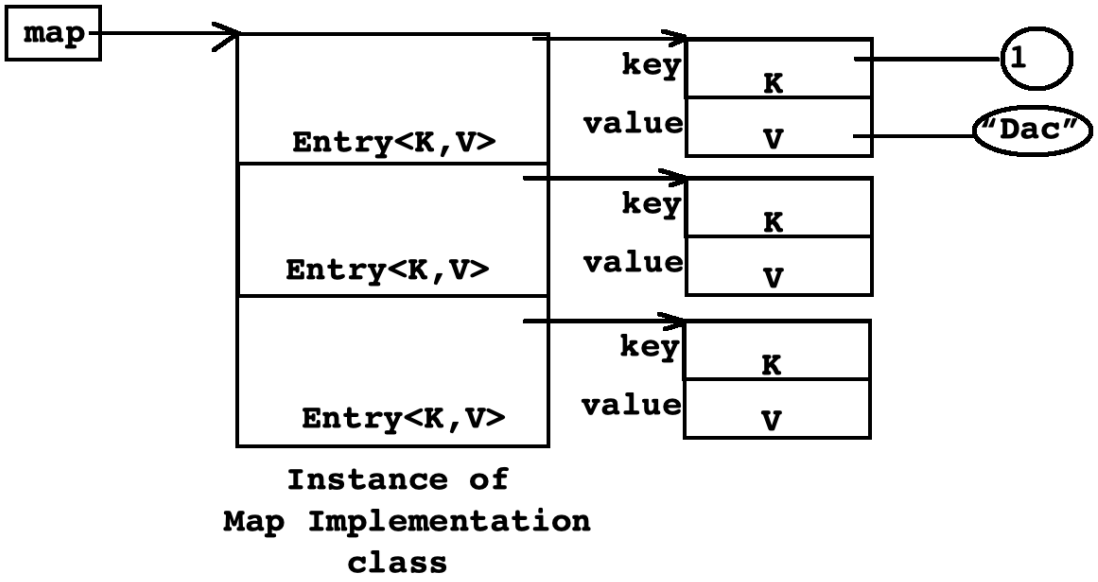
+ Map Interface
  - It is an interface declared in java.util package.
  - It is a member of Java's collection framework but it doesn't extends Collection interface.
  - This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
  - If we want to store data in key/value pair format then we should use Map implementation
  - In map we can not insert duplicate key but we can insert duplicate value.
  - It is a member of Java's collection framework.
  - It is introduced in JDK 1.2
  - Map Collections = { Hashtable, HashMap, TreeMap }



- Map.Entry<K,V> is nested interface of Map<K,V> interface
- Abstract Methods of Map.Entry interface:
- Abstract Methods of Map.Entry interface:
  1. K getKey()
  2. V getValue()
  3. V setValue(V value)
- Abstract Methods of Map Interface

```
public abstract int size();
public abstract boolean isEmpty();
public abstract boolean containsKey(Object);
public abstract boolean containsValue(Object);
public abstract V get(Object);
public abstract V remove(Object key );
public abstract V put(K key, V value );
public abstract void putAll(Map<? extends K, ? extends V>);
public abstract void clear();
public abstract Set<K> keySet();
public abstract Collection<V> values();
public abstract Set<Entry<K, V>> entrySet();
```

- **Map is a collection of entries where each entry contains key/value pair.**



+ **Hashtable**
  - **It is a sub class of Dictionary class which implements Map interface.**
  - **It is a key/value pair collection where key and value can not be null.**
  - **here we can not insert duplicate key but we can insert duplicate values**
  - **It is synchronised collection.**
  - **It is introduced in JDK 1.0**
  - **Instantiation**

    ```
    Dictionary<Integer,String> d = new Hashtable<>( );
    Map<Integer,String> d = new Hashtable<>( );
    ```

  - **Properties is a sub class of Hashtable class where type of key and value is String.**
  - **It is synchronised collection.**

+ HashMap
- **It is a map collection.**
- **In HashMap, key and Value can be null.**
- **It is unsynchronised collection.**
- **It can not contain duplicate keys but it can contain duplicate values.**
- **Hint, If we use instance of non final class as a key then non final class should override equals and hashcode method.**
- **Instantiation**

```
Map<Integer,String> map = new HashMap<>( );
```

+ LinkedHashMap
- **It is a sub class of HashMap class.**
- **During traversing, it maintains order of elements.**
- **It is unsynchronised collection.**
- **Its implementation is based on Hashtable and LinkedList**
- **Instantiation:**

```
Map<Integer,String> map = new LinkedHashMap<>();
```

+ TreeMap
- **It is a map collection**
- **Its implementation is based on Red Black Tree.**
- **In TreeMap, key can not be null but value can be null.**
- **We can not insert duplicate keys but we can insert duplicate values.**
- **It maintains entries in sorted form according to key.**
- **Hint : If we want to use instance of non final class as key then non final class must implement Comparable interface.**

demo

1. iterator , iterable

- 1.

```
public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        list.addLast(10);
        list.addLast(20);
        list.addLast(30);

        Integer element = null;
        Iterator<Integer> itr = list.iterator();
        while( itr.hasNext()) {
            element = itr.next();
            System.out.println(element);
        }
    }
    public static void main1(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        list.addLast(10);
        list.addLast(20);
        list.addLast(30);

        for( Integer element : list )
            System.out.println(element);
    }
```

- 2. demo of using Iterable interface ,for using forech loop

```
class Node{
    int element;
    Node next;
    public Node( int element ) {
        this.element = element;
    }
}
class LinkedList implements Iterable<Integer>{
    private Node head;
    private Node tail;
    public boolean empty( ) {
        return this.head == null;
    }
    public void addLast( int element ) {
        Node newNode = new Node( element );
        if( this.empty( ) )
            this.head = newNode;
        else
            this.tail.next = newNode;
        this.tail = newNode;
    }
    @Override
    public Iterator<Integer> iterator() {
        Iterator<Integer> itr = new LinkedListIterator( this.head );
//Upcasting
```

```
        return itr;
    }
}
class LinkedListIterator implements Iterator<Integer> {
    private Node trav;  //null
    public LinkedListIterator(Node head) {
        this.trav = head;
    }
    @Override
    public boolean hasNext() {
        if( this.trav != null )
            return true;
        return false;
    }
    @Override
    public Integer next() {
        Integer element = this.trav.element;
        trav = trav.next;
        return element;
    }
}
public class Program {
    public static void main(String[] args) {
        LinkedList list =  new LinkedList();
        list.addLast(10);
        list.addLast(20);
        list.addLast(30);

        for( int element : list )
            System.out.println(element);

        /*Integer element = null;
        Iterator<Integer> itr = list.iterator();
        while( itr.hasNext( ) ) {
            element = itr.next();
            System.out.println(element);
        }*/
    }
}
```

2. default , static in Interface

- 1. default method with same name in different interfaces, implemented in a class, needed to be overrided once in that class

```
interface A{
    default void f1( ) {
        System.out.println("A.f1");
    }
    default void f3( ) {
        System.out.println("A.f3");
```

```
    }
}
interface B{
    default void f2( ) {
        System.out.println("A.f2");
    }
    default void f3( ) {
        System.out.println("B.f3");
    }
}
class C implements A, B{
    @Override
    public void f3() {
        //A.super.f3();
        //B.super.f3();
        System.out.println("C.f3");
    }
}
```

- 2. functional interface , i.e onyl one abtract mehod, any one or more default and static metho allowed.
- can use annotation @FunctionalInterface, to check if declard it properly

```
@FunctionalInterface
interface A{    //Functional Interface / Single(S) Abstract(A) Method(M)
Interface / SAM Interface
    void f1( ); //Functional method / method descriptor
}

@FunctionalInterface
interface B{
    void f1( );
    static void f2( ) {
    }
    static void f3( ) {
    }
    default void f4( ) {
    }
    default void f5( ) {
    }
}
@FunctionalInterface
interface B{    //Error
    void f1( );
    void f2( );
}
```

- 3. default and static method use

```
interface Collection{
    void acceptRecord( );
    void printRecord( ); // abstract method
    int[] toArray( );
    static void swap( int[] arr ) { // static : helper/utility method
        int temp = arr[ 0 ];
        arr[ 0 ] = arr[ 1 ];
        arr[ 1 ] = temp;
    }
    default void sort( )  // default :added afterwards to the type
    {
        int[] arr =  this.toArray();
        for( int i = 0; i < arr.length - 1; ++ i ) {
            for( int j = 0; j < arr.length - 1 - i; ++ j ) {
                if( arr[ j ] > arr[ j + 1 ] ) {
                    int[] temp = { arr[ j ], arr[ j + 1 ] };
                    Collection.swap(temp);
                    arr[ j ] = temp[ 0 ]; arr[ j + 1 ] = temp[ 1 ];
                }
            }
        }
    }
}
class Array implements Collection{
    private int[] arr;
    public Array() {
        this( 5 );
    }
    public Array( int length ) {
        this.arr = new int[ length ];
    }
    @Override
    public void acceptRecord() {}
    @Override
    public void printRecord() {}
    @Override
    public int[] toArray() {
        //return Arrays.copyOf(this.arr, this.arr.length); //Defensive Copy
        return this.arr;
    }
}
public class Program {
    public static void main(String[] args) {
        Collection c = new Array(5);
        c.acceptRecord();
        c.printRecord();
        c.sort();
        c.printRecord();
    }
}
```

3. Annotation type

> java.lang
>
> lava.lang.annotation

- 1. demo on declaring a Annotation interface and use it on class
- target : specifies on which type (method,field,class) it is to be used
- retention : specifies till what time keep annotation, runtime,compile time , etc

```
@Documented
@Inherited
@Retention( value = RetentionPolicy.RUNTIME)
@Target( value = ElementType.TYPE )
@interface Author{
    String name( ); //Annotation Type Element declaration
}

@Author(name="ABC") //@Author : Annotation Type, name : element type
class Book{
    //TODO
}
public class Program {
    public static void main(String[] args) {
        Class<?> c = Book.class;
        Annotation[] annotations = c.getDeclaredAnnotations();
        for (Annotation annotation : annotations) {
            if( annotation instanceof Author ) {
                Author author = (Author) annotation;
                System.out.println(author.name());
            }
        }
    }
}
```

- 2. demo on making a repeatable annotation , and get it info by reflection

```
@Retention( value = RetentionPolicy.RUNTIME)
@Target( value = ElementType.TYPE )
@Repeatable( value = Authors.class )
@interface Author{
    String name( ); //Annotation Type Element declaration
    String date( ); //Annotation Type Element declaration
}
@Retention( value = RetentionPolicy.RUNTIME)
@Target( value = ElementType.TYPE )
@interface Authors{
```

```
    Author[] value();
}

@Author(name="ABC", date="09/11/2020")
@Author(name="XYZ", date="09/11/2020")
class Book{
    //TODO
}
public class Program {
    public static void main(String[] args) {
        Class<?> c = Book.class;
        Annotation[] annotations = c.getDeclaredAnnotations();
        for (Annotation annotation : annotations) {
            if( annotation instanceof Authors ) {
                Authors authors = (Authors) annotation;
                for (Author author : authors.value()) {
                    System.out.println(author.name()+"  "+author.date());
                }
            }
        }
    }
}
```

4. Collection framework

- 1.

```

```

- 2.

```

```

- 3.

```

```

- 4.

```

```

```

```

- 5.

```

```

part2

1. fail fast & fail safe iterator

Day 13

to read

1.

slides

collection

+ Dictionary
  - It is an abstract class declared in java.util package.
  - Hashtable is sub class of Dictionary class.
  - If we want to store data in key/value pair form them we should this class.
  - Methods

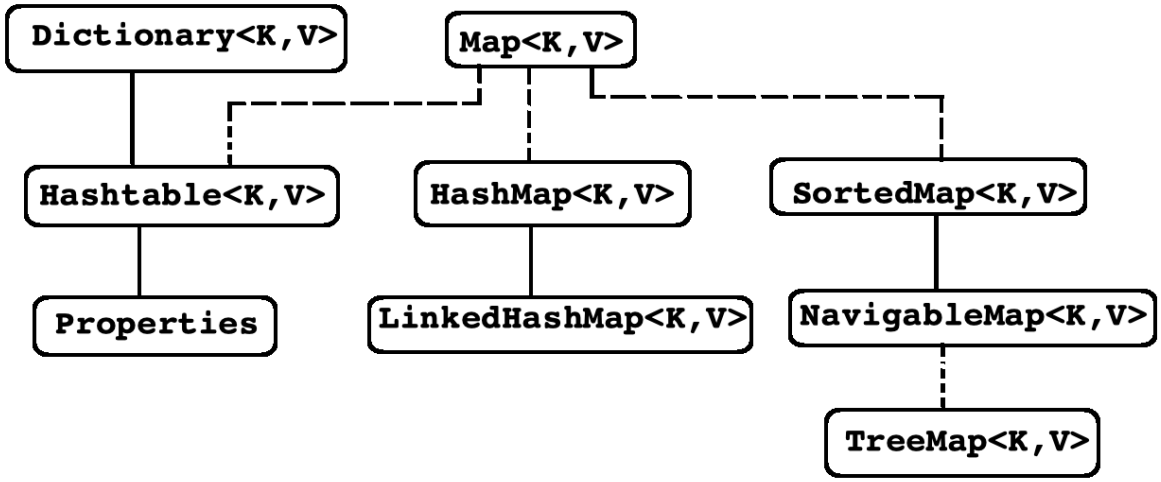
```
public abstract int size();
public abstract boolean isEmpty();
public abstract Enumeration<K> keys();
public abstract Enumeration<V> elements();
public abstract V get(Object key);
public abstract V put(K key, V value);
public abstract V remove(Object key);
```

  - Instantiation

```
Dictionary<Integer, String> d = null;
d = new Hashtable<>( );
```

  - NOTE: This class is obsolete. New implementations should implement the Map interface, rather than extending this class.

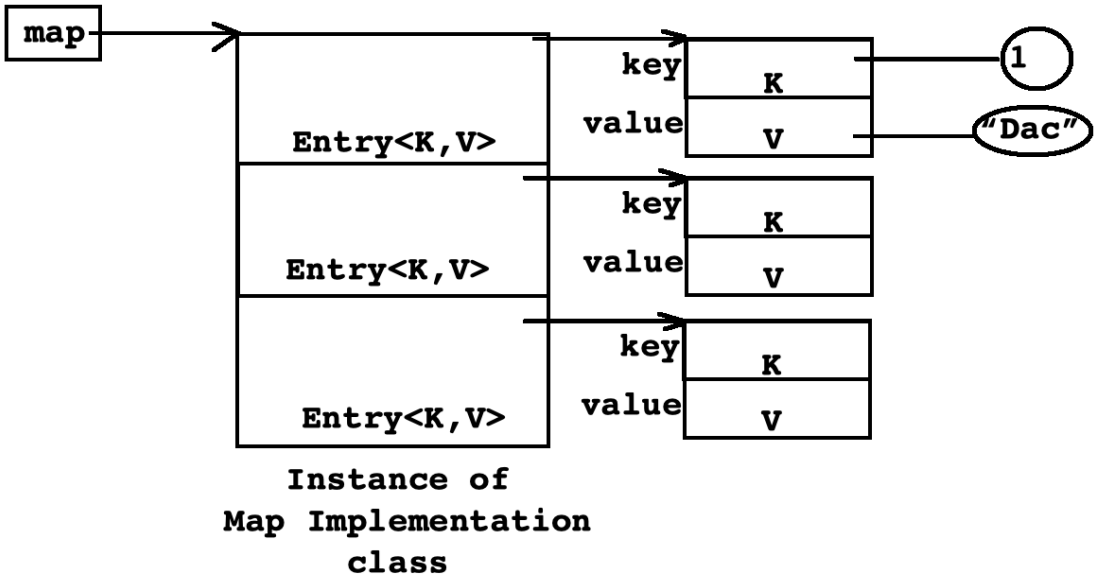
+ Map Interface
  - It is an interface declared in java.util package.
  - It is a member of Java's collection framework but it doesn't extends Collection interface.
  - This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
  - If we want to store data in key/value pair format then we should use Map implementation
  - In map we can not insert duplicate key but we can insert duplicate value.
  - It is a member of Java's collection framework.
  - It is introduced in JDK 1.2
  - Map Collections = { Hashtable, HashMap, TreeMap }



- Map.Entry<K,V> is nested interface of Map<K,V> interface
- Abstract Methods of Map.Entry interface:
- Abstract Methods of Map.Entry interface:
  1. K getKey()
  2. V getValue()
  3. V setValue(V value)
- Abstract Methods of Map Interface

```
public abstract int size();
public abstract boolean isEmpty();
public abstract boolean containsKey(Object);
public abstract boolean containsValue(Object);
public abstract V get(Object);
public abstract V remove(Object key );
public abstract V put(K key, V value );
public abstract void putAll(Map<? extends K, ? extends V>);
public abstract void clear();
public abstract Set<K> keySet();
public abstract Collection<V> values();
public abstract Set<Entry<K, V>> entrySet();
```

- **Map is a collection of entries where each entry contains key/value pair.**



+ **Hashtable**
  - **It is a sub class of Dictionary class which implements Map interface.**
  - **It is a key/value pair collection where key and value can not be null.**
  - **here we can not insert duplicate key but we can insert duplicate values**
  - **It is synchronised collection.**
  - **It is introduced in JDK 1.0**
  - **Instantiation**

    ```
    Dictionary<Integer,String> d = new Hashtable<>( );
    Map<Integer,String> d = new Hashtable<>( );
    ```

  - **Properties is a sub class of Hashtable class where type of key and value is String.**
  - **It is synchronised collection.**

+ HashMap
- **It is a map collection.**
- **In HashMap, key and Value can be null.**
- **It is unsynchronised collection.**
- **It can not contain duplicate keys but it can contain duplicate values.**
- **Hint, If we use instance of non final class as a key then non final class should override equals and hashcode method.**
- **Instantiation**

```
Map<Integer,String> map = new HashMap<>( );
```

+ LinkedHashMap
- **It is a sub class of HashMap class.**
- **During traversing, it maintains order of elements.**
- **It is unsynchronised collection.**
- **Its implementation is based on Hashtable and LinkedList**
- **Instantiation:**

```
Map<Integer,String> map = new LinkedHashMap<>();
```

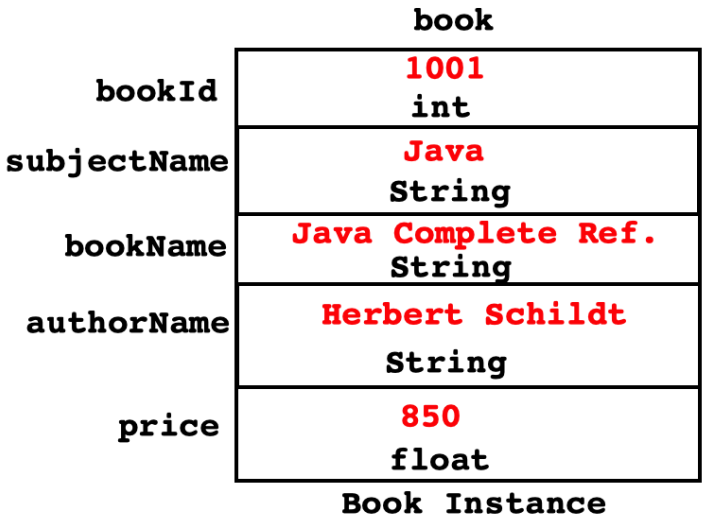
+ TreeMap
- **It is a map collection**
- **Its implementation is based on Red Black Tree.**
- **In TreeMap, key can not be null but value can be null.**
- **We can not insert duplicate keys but we can insert duplicate values.**
- **It maintains entries in sorted form according to key.**
- **Hint : If we want to use instance of non final class as key then non final class must implement Comparable interface.**

JDBC

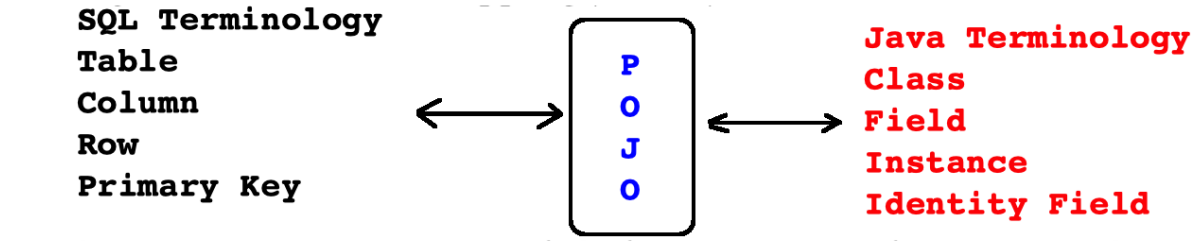
Table : books

book_id	subject_name	book_name	author_name	price
1001	Java	Java Complete Reference	Herbert Schildt	850

columns
Row

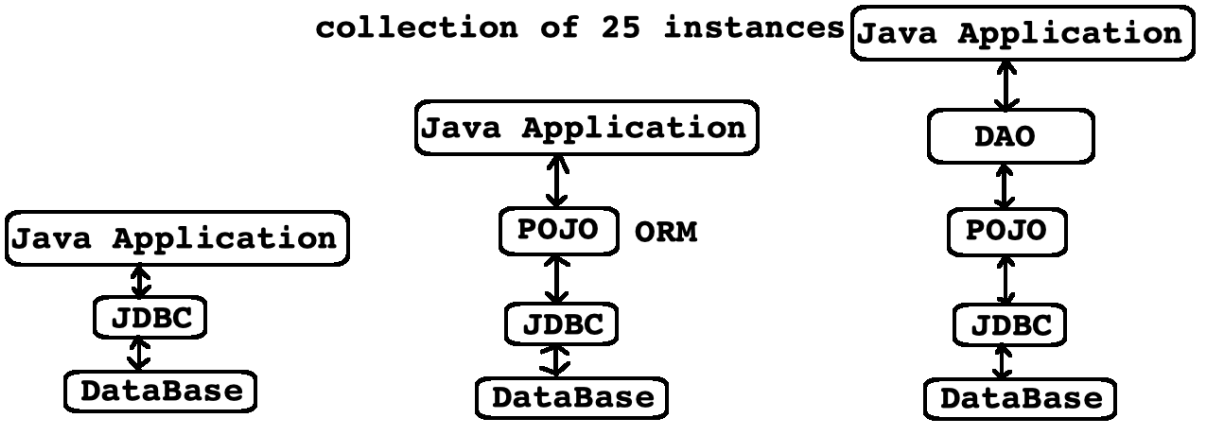


+ Object Relational Mapping(ORM)



- If we want to map rows with instance or vice versa then we should use POJO class.

+ Plain Old Java Object(POJO)
  - It is also called as DTO/VO/BO/entity
  - Rules
    1. It must be packaged public class
    2. It should contain default constructor
    3. For the columns it should contain private fields(use camel case convention)
    4. For every private filed it should contain getter and setter.
    5. It should not contain business logic method
    6. It contain toString(), equals(), hashCode() etc.



+ Data Access Object(DAO) Layer
  - DAO is design pattern which help us to separate data manipulation logic from business logic.
  - Rules to define DAO
    1. It must be packaged public class.
    2. It should contain default constructor.
    3. It must contain CRUD operation.

    C:CREATE/INSERT R:READ/SELECT
    U:UPDATE D:DELETE

1. Statement object can not handle special character in a query.
2. It Compiles query per request.
3. We can not use it to execute stored procedure and stored function.

- If we want to overcome limitation of static queries then we should write parameterized query.
- Example:

  “INSERT INTO books
  (book_id, subject_name, book_name, author_name, price)
  VALUES(?,?,?,?,?)”;
- If we want to execute parameterized query then we should use PreparedStatement object.

- PreparedStatement interface is sub interface of Statement interface.
- Instantiation:

  PreparedStatement = connection.prepareStatement(sql);
- Methods:
  1. void setXXX(int parameterIndex, XXX value)throws SE
  2. int executeUpdate( );
  3. ResultSet executeQuery( );

+ CallableStatement
  - It is a sub interface of PreparedStatement interface.
  - If we want to execute stored procedure and function then we should use CallableStatement object.
  - Instantiation
    String sql =“”;
    CallableStatement stmt = connection.prepareCall(sql);
  - Method:
    boolean execute() throws SQLException
    Returns:
      true if the first result is a ResultSet object;
      false if the first result is an update count or
      there is no result
  - Syntax to call Stored Procedure and Stored Function:

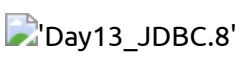
```
{call procedure-name(arg1,arg2, ...)}

{?= call function-name(arg1,arg2, ...)}
```

+ ResultSetType
  1. ResultSet.TYPE_FORWARD_ONLY,
  2. ResultSet.TYPE_SCROLL_INSENSITIVE
  3. ResultSet.TYPE_SCROLL_SENSITIVE

+ ResultSetConcurrency
  1. ResultSet.CONCUR_READ_ONLY
  2. ResultSet.CONCUR_UPDATABLE

+ ResultSetHoldability
  1. ResultSet.HOLD_CURSORS_OVER_COMMIT
  2. ResultSet.CLOSE_CURSORS_AT_COMMIT



demos

1. Collection framework

- 1.

- 2.

- 3.

- 4.

- 5.

2. JDBC (Java DataBase Connectivity)

- 1.

h

- 2.

-

3.

```

```

- 4.

```

```

- 5.

```

```

Day 14

to read

1. why process based multi tasking is heavy weight process and thread based is light weight multitasking?

- 1. in process, context switching is heavy
- 2. in thread, as reource shared in same process of thread , so no context switching

2. types of thread in

- C# for ground and background thread
- java, user thread and daemon thread

3. difference between deamon and user thread?

4. to read

> pdf java certification khalid mughal

5. what is relation between run() and start() method?

6. when to use extends thread and when to implement Runnable?

7. difference between Runnbable I/F and Thread class?

8. thread priority makes java application platform dependent, as different priority ranges are there for windows,java, unix , so behaviour is diff on diff platform

9. difference between locking and syncronizing thread /resource

10. how to achieve synchronisation in java?

- using wait , notify(), notifyall() on threads , in synchronised block only or monitored is asssigned to them .

slides

demo

1. public static enum Thread.State

xtends Enum<Thread.State>

- A thread state. A thread can be in one of the following states:

1. NEW A thread that has not yet started is in this state.
2. RUNNABLE
3. BLOCKED
4. WAITING
5. TIMED_WAITING
6. TERMINATED

- A thread can be in only one state at a given point in time.
- These states are virtual machine states which do not reflect any operating system thread states.

2. thread priority

- Constant Field- Value

1. MAX_PRIORITY - 10
2. MIN_PRIORITY- 1
3. NORM_PRIORITY - 5

-

3. garbage collector is low proprity background thread ,

- as it runs periodically

4. blocking call : sleep(),suspend(),join(),wait(),IO call
5. locking thread resources

- synchronised keyword
    - monitor object get asigned to the block , and it locks thread

6. difference between locking and syncronizing thread /resource

demo

1. demo on method of Thread class

- 1. getting the current running thread using currentThread() method. h

```
Thread thread = Thread.currentThread();

String name = thread.getName();

int priority = thread.getPriority();

String groupName = thread.getThreadGroup().getName();
String state = thread.getState().name();

boolean type = thread.isDaemon();
System.out.println("Type        :   "+( type ? "Daemon" : "User"));

boolean status = thread.isAlive();
System.out.println("Status      :   "+( status ? "Alive" : "Dead"));
```

- 2. using garbage collector to overide finalize method
- but gc cames periodically , so finalize method not used in each program execution

```
class Test{
    public Test() {
        System.out.println("Inside constructor");
    }
    public void print( ) {
        System.out.println("Inside print");
    }
    @Override
    public void finalize(){
        System.out.println("Inside finalize");

    }
}
public class Program {
    public static void main(String[] args) {
        Test t = new Test( );   //Inside constructor
        t.print( );     //Inside print
        t = null;
        System.gc();//Inside finalize
    }
}
```

- 3. blocking call methods
- Blocking Calls :

1. sleep(),
2. suspend(),
3. join(),
4. wait(),
5. io call

```
public class Program {

    //Blocking Calls : sleep(), suspend(), join(), wait(), io call
    public static void main(String[] args) {

        try {
            for (int count = 0; count <= 10; count++) {
                System.out.println(count);

                Thread.sleep(250);

            }
        } catch (Exception e) {
            // TODO: handle exception
        }
    }
}
```

- 4. creating a new thread using java.lang.Thread class Object ,

1. assigning a java.lang.thread class instance mapped to a OS thread, to thread object using

- start () :
  - use to assign thread instance to the object, and making thread state NEW

2. now jvm on start() method assigging the thread instance, calls the run () method ,to execute the logic

- run()
  - it is called by jvm, when it maps a OS thread to thread class instance and assign it to start() object ,and calls the run() method
  - it contains the business logic

```
class Task implements Runnable{

    @Override
    public void run() {

        System.out.println(Thread.currentThread().getName() + " : " +
Thread.currentThread().getState());//New
        System.out.println("Inside business logic method");
    }
```

```
}

public class Program {

public static void main(String[] args) {

    Runnable target = new Task(); // upcasting
    Thread th1 = new Thread(target, "User-Thread-1");
    th1.start();      //RUNNABLE
}
```

2. demo on method to create thread,

- method 1 : implementing Runnable interface Runnable implements : gives target for thread (class for run() method)
- 1. using Runnable interface
- one thread using start() method twice gives exception IllegalThreadStateException

```
    th.start();   //RUNNABLE
        th.start();  //IllegalThreadStateException
```

- 2. using a CThread class, to make code reusable

```
class Task implements Runnable{
    @Override
    public void run() {

        System.out.println("Inside business logic method");
    }
}

class CThread{

    private Thread thread;

    public CThread(String name) {
        Runnable target = new Task();

        this.thread = new Thread(target, name);

        this.thread.start();
    }
}
public class Program {

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

```
    CThread th1 = new CThread("A");
    CThread th2 = new CThread("B");
    CThread th3 = new CThread("C");

}
```

- 3. implementing Runnable interface on Cthread class, where
- Runnable implements : gives target for thread

```
class CThread implements Runnable{

    private Thread thread;

    public CThread(String name) {

        this.thread = new Thread(this, name);

        this.thread.start();
    }

    @Override
    public void run() {

        System.out.println("Inside business logic method");
    }
}
public class Program {

public static void main(String[] args) {
    CThread th1 = new CThread("A");
    CThread th2 = new CThread("B");
    CThread th3 = new CThread("C");

}
```

3. demo on method to create thread,

- method 2 : extending Thread class

- 1. extending java.lang.Thread class

```
class CThread extends Thread{

    public CThread(String name) {
        super(name);
        this.start();
    }

    @Override
```

```
    public void run() {
        System.out.println("Inside Business Logic Method");
    }
}
public class Program {

public static void main(String[] args) {

    Thread t1 = new CThread("A");
    Thread t2 = new CThread("B");
    Thread t3 = new CThread("C");
   }
}
```

- 2. getting thread state while executing application

```
class CThread extends Thread{

    public CThread(String name) {

    super(name); //using thread class constructor

System.out.println(this.getName()+" :   "+this.getState());
this.start();
    }

    @Override
    public void run() {
        System.out.println(this.getName()+" :   "+this.getState());
    try {
    for( int count = 1; count <= 10; ++ count ) {
System.out.println(this.getName()+" :   "+count);
                Thread.sleep(500);
            }
        } catch (InterruptedException cause) {
            throw new RuntimeException(cause);
        }
    }
}

public class Program {

public static void main(String[] args) {

    try {
        Thread thread = new CThread("User Thread");

        while(thread.isAlive())
        {
```

```
System.out.println("inside while" + thread.getName() + " : " +
thread.getState());
    Thread.sleep(100);
    }
    System.out.println("after while" + thread.getName() + " : " +
thread.getState());

    } catch (Exception e) {}
   }
}
```

- 3. thread toString() demo
- when a new thread is created in run() ,
- its priority is same as the thread of start() method thread , which caused jvm to call run() method
- super thread and sub thread have same priority

```
class Task implements Runnable{
    @Override
    public void run() {
        Thread thread = Thread.currentThread();
        //System.out.println(thread.toString());
        //System.out.println("Inside run()");
    }
}
public class Program {
    public static void main(String[] args) {
        Thread thread = Thread.currentThread();
        System.out.println(thread.toString());

        Runnable target = new Task();
        target.run();
    }
}
```

4. demo on thread priority

- 1. get ,set ,and exception on priority h

```
public class Program {
    public static void main4(String[] args) {
        Thread thread = Thread.currentThread();
        //thread.setPriority(Thread.NORM_PRIORITY + 6 );
//IllegalArgumentException
        //thread.setPriority(Thread.NORM_PRIORITY - 6 );
//IllegalArgumentException
    }
    public static void main3(String[] args) {
        Thread thread = Thread.currentThread();
        //thread.setPriority(thread.getPriority() + 3 );
```

```
        thread.setPriority(Thread.NORM_PRIORITY + 3 );
        System.out.println(thread.getPriority());
    }
    public static void main2(String[] args) {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getPriority());
    }
    public static void main1(String[] args) {
        System.out.println(Thread.MIN_PRIORITY);    //1
        System.out.println(Thread.NORM_PRIORITY);   //5
        System.out.println(Thread.MAX_PRIORITY);    //10
    }
}
```

- 2. priority of main thread and default thread of thread class is same ,
- here main in super thread and class Cthread thread is sub/default

```
class CThread extends Thread{

    public CThread() {
    this.start();
    }
    @Override
    public void run() {
//this.setPriority(NORM_PRIORITY + 3 );
System.out.println(this.getName()+" "+this.getPriority());
    }
}
public class Program {

public static void main(String[] args) {
    Thread thread = Thread.currentThread();

thread.setPriority(Thread.NORM_PRIORITY);
System.out.println(thread.getName()+"   "+thread.getPriority());
    CThread cthread = new CThread();
}
}
```

5. demo on locking thread in java

- 1. locking of thread using join() method

```
class CThread extends Thread{
    public CThread( String name ) {
        this.setName(name);
        this.start();
```

```
    }
    @Override
    public void run() {
        try {
            for( int count = 1; count <= 10; ++ count ) {
                System.out.println(this.getName()+" :   "+count);
                Thread.sleep(250);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class Program {
    public static void main(String[] args) {
        try {
            Thread th1 = new CThread("A");
            th1.join();

            Thread th2 = new CThread("B");
            th2.join();

            Thread th3 = new CThread("C");
            th3.join();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 2. thread locking using synchronized block in run()

```
class SumArray{

    public int sum(int[] arr) throws InterruptedException
    {
        int result = 0;
        for(int element : arr)
        {
            result = result + element;
System.out.println("Running total for "+Thread.currentThread().getName()+"
is "+result);
    Thread.sleep(300);
}
        return result;
    }
}

class CThread implements Runnable{

    private int[] arr;
```

```
    private Thread thread;

    public CThread(String name,int[] arr) {
        this.arr = arr;
        this.thread = new Thread(this, name);
        this.thread.start();
    }

    private static SumArray sa = new SumArray();
    @Override
    public void run()// no locking , ramdom thread access sum function
    {

        try {
            // thread locking
            synchronized (sa)  {

                   int result = sa.sum(arr);
                    System.out.println("Result : " + result);
            }

        } catch (Exception e) {
            // TODO: handle exception
        }

    }

    // no locking , ramdom thread access sum function
    public void run()
    { }


}

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

        int[] arr1 = { 1,2,3,4,5,6,7,8,9,10 };
        CThread th1 = new CThread("Th1",arr1);

        int[] arr2 = { 11,12,13,14,15,16,17,18,19,20 };
        CThread th2 = new CThread("Th2",arr2);
    }
}
```

- 3. locking method to be used by thread , for locking the threads

```
class SumArray{
    public synchronized int sum( int[] arr ) throws InterruptedException {
        int result = 0;
```

```
        for( int element : arr ) {
            result = result + element;
            System.out.println("Running total for
"+Thread.currentThread().getName()+" is "+result);
            Thread.sleep(300);
        }
        return result;
    }
}
```

- 4. locking block of code to using synchronized keyword, where
- this represent the class of method

```
class SumArray{
    public int sum( int[] arr ) throws InterruptedException {
        int result = 0;
        synchronized (this) {
            for( int element : arr ) {
                result = result + element;
                System.out.println("Running total for
"+Thread.currentThread().getName()+" is "+result);
                Thread.sleep(300);
            }
            return result;
        }

    }
}
```

- 5.

6. thread Syncronization using Object class method

- wait()
- wait(int timeout)
- notify()
- notifyAll()
- 1. demo on tick tock code using thread syncronizing

```
package test;

class TickTock {
    public void tick() throws InterruptedException {
        synchronized (this) {
            System.out.print("Tick  ");
            this.notify();
            //this.wait();
```

```
                this.wait(1000);
            }
        }

        public void tock() throws InterruptedException {
            synchronized (this) {
                System.out.println("    Tock");
                this.notify();
                //this.wait();
                this.wait(1000);
            }
        }
    }

    class CThread implements Runnable {
        private Thread thread;

        public CThread(String name) {
            this.thread = new Thread(this, name);
            this.thread.start();
        }

        private static TickTock tt = new TickTock();

        @Override
        public void run() throws RuntimeException {
            try {
                if (Thread.currentThread().getName().equals("TickThread")) {
                    for (int count = 1; count <= 5; ++count) {
                        tt.tick();
                        Thread.sleep(250);
                    }
                } else {
                    for (int count = 1; count <= 5; ++count) {
                        tt.tock();
                        Thread.sleep(250);
                    }
                }
            } catch (InterruptedException cause) {
                throw new RuntimeException(cause);
            }
        }
    }

    public class Program {
        public static void main(String[] args) {
            CThread th2 = new CThread("TockThread");
            CThread th1 = new CThread("TickThread");
        }
    }
```

- 2. class/variable which are syncronised will only get OBject class method ,

- wait()
- wait(int timeout)
- notify()
- notifyAll()
- otherwise exception is thrown

> java.lang.IllegalMonitorStateException

```
public class Program {
    private static String str = "Hello";
    public void print() throws InterruptedException {

        synchronized (str) {
System.out.println(str);
//this.wait(1000); //IllegalMonitorStateException
str.wait(1000); //OK
    //System.out.println(str);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Program p = new Program();
        p.print();
    }
}
```

Day 15

to read

1. Not Serializable Exception
2. how to count number of words in a file

slides

demo

1. demo File

- 1. to create a new file

```
public static void main(String[] args) {
    String pathName = "file.txt";
    File file = new File(pathName);
    boolean status  = file.createNewFile();
    System.out.println(status ? "file is created" : "file not created");
```

```
}
```

- 2. delete a file

```
public static void main(String[] args) {
    String pathName = "file.txt";
    File file = new File(pathName);
    boolean status = file.delete();
System.out.println(status ? "file deleted" : "file not deleted");
    }
```

- 3. create a folder

```
    public static void main(String[] args) {
        String pathName ="myFolder";

        File file = new File(pathName);

        boolean status = file.mkdir();

        System.out.println(status ? "folder created": "folder not
created");
    }
```

- 4. delete a folder

```
public static void main(String[] args) {
    String pathName ="myFolder";
    File file = new File(pathName);
    System.out.println(file.getAbsolutePath());
    if(file.isDirectory())
        {
            boolean status = file.delete();
            System.out.println(status ? "folder deleted": "folder not
deleted");
        }
    }
```

- 5. file and folder info

```
private static void FileInfo(File file) {
    if(file.exists())
    {
System.out.println("Name " + file.getName());
```

```
System.out.println(" parent" + file.getParent());
System.out.println("length " + file.length());
System.out.println(" " + new SimpleDateFormat("dd/mm/yyyy").format(new
Date(file.lastModified())) );
}
}
private static void DirectoryInfo(File file) {
File[] files = file.listFiles();
for(File f : files)
{
if(!f.isHidden())
System.out.println(f.getName());
}
}
```

2. **demo on File stream classes(input and output)**

- need for permission for each byte to read/write .

```
public class FileInputStream
extends InputStream
```

- A FileInputStream obtains input bytes from a file in a file system.
- FileInputStream is meant for reading streams of raw bytes such as image data.
- For reading streams of characters, consider using FileReader.

```
public class FilterOutputStream
extends OutputStream
```

- This class is the superclass of all classes that filter output streams.
- The class FilterOutputStream itself simply overrides all methods of OutputStream
- 1. demo java.io.File(Output/Input)Stream
- reading or writing one byte at a time

```
private static void readRecord(String pathname) {
    FileInputStream inputStream = null;
    inputStream = new FileInputStream(new File(pathname));
    char data = (char) inputStream.read();
    System.out.println(data);
    }

 private static void writeRecord(String pathname) {
    FileOutputStream outputStream = null;
```

```
    outputStream = new FileOutputStream(new File(pathname));
    outputStream.write('A');
    }
}
```

- 2. write and read byte string

```
private static void readRecord(String pathname) {
try(FileInputStream  inputStream = new FileInputStream(new File(pathname));
) {
      int data;
      while((data = inputStream.read()) != -1)
       System.out.print( (char)data + " ");
        } catch (Exception e) {}
    }
private static void writeRecord(String pathname) {
try(FileOutputStream outputStream = new FileOutputStream(new
File(pathname)); )
  {
   for(char ch = 'A'; ch <= 'Z'; ++ ch)
    outputStream.write(ch);
  } catch (Exception e) {   }
}
```

3. **demo on Buffer(Input/Output)Stream**

- here String of input is taken at a time ,no need for permission for each byte .

```
public class BufferedInputStream
extends FilterInputStream
```

- A BufferedInputStream adds functionality to another input stream-namely,
  - the ability to buffer the input and to support the mark and reset methods.
- When the BufferedInputStream is created, an internal buffer array is created.

```
public class BufferedOutputStream
extends FilterOutputStream
```

- The class implements a buffered output stream.
- By this, an application can write bytes to the underlying output stream
  - without necessarily causing a call to the underlying system for each byte written.
-
  1.

```
private static void readRecord(String pathname) {
try(BufferedInputStream inputStream = new BufferedInputStream (new
FileInputStream(new File(pathname))); ) {
 int data;
while((data = inputStream.read()) != -1)
{
    System.out.print( (char)data + " ");
}
System.out.println();
} catch (Exception e) {}
}

private static void writeRecord(String pathname) {
try(BufferedOutputStream outputStream =  new BufferedOutputStream(new
FileOutputStream(new File(pathname))); ) {
    for(char ch = 'A'; ch <= 'Z'; ++ ch)
            outputStream.write(ch);
        } catch (Exception e) {
    }
}
```

4. **demo on Data(Input/Output)Stream**

```
public class DataInputStream
extends FilterInputStream
implements DataInput
```

- A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
- DataInputStream is not necessarily safe for multithreaded access.

```
public class DataOutputStream
extends FilterOutputStream
implements DataOutput
```

- A data output stream lets an application write primitive Java data types to an output stream in a portable way.

```
private static void readRecord(String pathname) {
try(DataInputStream inputStream = new DataInputStream(new
BufferedInputStream (new FileInputStream(new File(pathname)))); ) {

String name = inputStream.readUTF();
int num = inputStream.readInt();
float salary = inputStream.readFloat();
```

```
 System.out.println(name + " : " +  num  + " : "+ salary);
} catch (Exception e) {}
}
private static void writeRecord(String pathname) {

try(DataOutputStream outputStream = new DataOutputStream(new
BufferedOutputStream(new FileOutputStream(new File(pathname)))); ) {

    outputStream.writeUTF("suraj");
    outputStream.writeInt(33);
    outputStream.writeFloat(3000);

        } catch (Exception e) {
    }
}
```

5. **demo Object(Input/Output) Stream**

- 1. An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
- The objects can be read (reconstituted) using an ObjectInputStream.
- Persistent storage of objects can be accomplished by using a file for the stream.

```
public class ObjectOutputStream
extends OutputStream
implements ObjectOutput, ObjectStreamConstants
```

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

```
public class ObjectInputStream
extends InputStream
implements ObjectInput, ObjectStreamConstants
```

- 3. demo on on Object Stream

```
public static void writeRecord( String pathname ) throws Exception{
try(ObjectOutputStream outputStream = new ObjectOutputStream(new
BufferedOutputStream( new FileOutputStream(new File(pathname))));)
{
Employee emp = new Employee("Sandeep", 33, 15000.50f);
outputStream.writeObject(emp);
        }
    }
public static void readRecord( String pathname ) throws Exception{
try(ObjectInputStream inputStream = new ObjectInputStream(new
```

```
BufferedInputStream(new FileInputStream(new File(pathname)))))
{
Employee emp = (Employee) inputStream.readObject();
    System.out.println(emp.toString());
        }
    }
```

6. demo on Buffer(Input/Output)Stream

- 1. Writer

```
public abstract class Writer
extends Object
implements Appendable, Closeable, Flushable
```

- Abstract class for writing to character streams.
- The only methods that a subclass must implement are
    - write(char[], int, int),
    - flush(), and
    - close().
- Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.
- 2. Reader

```
public abstract class Reader
extends Object
implements Readable, Closeable
```

- Abstract class for reading character streams. - The only methods that a subclass must implement ar
    - e read(char[], int, int) and
    - close().
- Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.
- 3. demo on reader and writer

```
public static void writeRecord( String pathname ) throws Exception
{
try(FileWriter writer = new FileWriter(new File(pathname));)
{
for(char ch = 'A'; ch <= 'Z';++ch)
```

```
writer.write(ch);
    }
}
public static void readRecord( String pathname ) throws Exception{
try(FileReader reader = new FileReader(new File(pathname)))
{
int data;
while((data = reader.read())!= -1){
System.out.println( (char)data + " ");
}
    }
}
```

7. demo Buffered(Reader/Writer) class

- 1. BufferedWriter class

```
public class BufferedWriter
extends Writer
```

- Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
- 2. BufferedReader class

```
public class BufferedReader
extends Reader
```

- Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

- 3. demo on Buffer classes

```
public static void writeRecord( String pathname ) throws Exception{
    try(BufferedWriter writer = new BufferedWriter(new FileWriter(new
File(pathname))))
    {
writer.write("Diploma In Advanced Computing");
    writer.newLine();
    for(char ch = 'A'; ch <= 'Z';++ch)
        writer.write(ch);
    }
}

public static void readRecord( String pathname ) throws Exception{
try(BufferedReader reader = new BufferedReader(new FileReader(new
File(pathname))))
{
```

```
    int data;
while((data = reader.read())!= -1)
{System.out.println( (char)data + " ");}

String line = null;
while((line = reader.readLine()) != null)
System.out.println(line);
    }
}
```

- 4. demo to implement java p command functionality

```
public static void readRecord( String pathname,String search ) throws
Exception{
try(BufferedReader reader = new BufferedReader(new FileReader(new
File(pathname))))
{
String line = null;
int count = 0;
while((line = reader.readLine())!= null) {
        ++count;
    if(line.contains(search)) {
System.out.println(count + " " + line.trim());
    }
   }
 }
}
public static void main(String[] args) throws Exception {
String pathName = "/home/sunbeam/dac/Core
Java/java_module_classwork/Classwork/Day_15_12_Nov/Day_15.11/src/test/Progr
am.java";

String searchString = "public";
Program.readRecord(pathName,searchString);
 }
}
```

8. demo on java.io class

```
 public class InetAddress
extends Object
implements Serializable
```

- This class represents an Internet Protocol (IP) address.
- An IP address is either a 32-bit or 128-bit unsigned number used by IP,a lower-level protocol on which protocols like UDP and TCP are built.

```
public final class Inet4Address
extends InetAddress
```

- This class represents an Internet Protocol version 4 (IPv4) address.
-
    1.

```
  InetAddress localhost = Inet4Address.getLocalHost();
       String hostname = localhost.getHostName();
       String hostAddress = localhost.getHostAddress();

       System.out.println(hostname);
       System.out.println(hostAddress);
```

9. demo on Socket Programming using TCP protocol,

- for creating messaging between server and client
-
    1. Server side

```
package server;
public class SProgram {
    public static final int port = 5465;

public static void main(String[] args) {
ServerSocket serverSocket = null;
    DataOutputStream outputStream = null;
    DataInputStream inputStream = null;
    Scanner sc = null;
try {
    serverSocket = new ServerSocket(port);
    System.out.println("Server is ready....");
    Socket socket = serverSocket.accept();

inputStream = new DataInputStream(new
BufferedInputStream(socket.getInputStream()));

outputStream = new DataOutputStream(new
BufferedOutputStream(socket.getOutputStream()));
sc = new Scanner(System.in);

    String message = "";
do {
    System.out.print("S:Server  :   ");
                message = sc.nextLine();
    outputStream.writeUTF(message); // Send message to client
    outputStream.flush();
```

```
    message = inputStream.readUTF();
    System.out.println("S:Client    :   " + message); // Read message of
client
    } while (!message.equalsIgnoreCase("end"));

    } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            try {
                sc.close();
                inputStream.close();
                outputStream.close();
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- 2. on client side

```
package client;

public class Program {
    public static String host = "localhost";
    public static final int port = 5465;

public static void main(String[] args) {
Socket socket = null;
DataOutputStream outputStream = null;
DataInputStream inputStream = null;
Scanner sc = null;
try {
    socket = new Socket(host, port);

inputStream = new DataInputStream(new
BufferedInputStream(socket.getInputStream()));

outputStream = new DataOutputStream(new
BufferedOutputStream(socket.getOutputStream()));

sc = new Scanner(System.in);

String message = "";
do {
message = inputStream.readUTF(); // Read message of server
System.out.println("C:Server    :   " + message);

System.out.print("C:Client  :   ");
message = sc.nextLine();
```

```
outputStream.writeUTF(message); // Send message to server
outputStream.flush();
} while (!message.equalsIgnoreCase("end"));

} catch (Exception ex) {
            ex.printStackTrace();
    } finally {
try {
                sc.close();
                inputStream.close();
                outputStream.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

10. demo on socket using threading for increasing performance

- port no : is a logical number assigned to each application process on machine/server machine
- 1. Communication handler

```
package utils;

public class CommunicationHandler implements Runnable{
private Socket socket;
private Thread thread;

public CommunicationHandler(Socket socket) {
    this.socket = socket;
    this.thread = new Thread(this);
    this.thread.start();
    }
@Override
public void run() {

DataOutputStream outputStream = null;
DataInputStream inputStream = null;
Scanner sc = null;
try {
inputStream = new DataInputStream(new
BufferedInputStream(socket.getInputStream()));

outputStream = new DataOutputStream(new
BufferedOutputStream(socket.getOutputStream()));
    sc = new Scanner(System.in);

    String message = "";
```

```
do {
System.out.print("S:Server  :   ");
    message = sc.nextLine();
outputStream.writeUTF(message); // Send message to client
outputStream.flush();
message = inputStream.readUTF();

System.out.println("S:Client    :   " + message); // Read message of client
} while (!message.equalsIgnoreCase("end"));

    } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            try {
                sc.close();
                inputStream.close();
                outputStream.close();

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- 2. changes on server side, Client side remains same

```
package server;

import utils.CommunicationHandler;
public class SProgram {
public static final int port = 5465;

public static void main(String[] args) {
ServerSocket serverSocket = null;

try {
serverSocket = new ServerSocket(port);
System.out.println("Server is ready....");

while(true)
{
Socket socket = serverSocket.accept();
 Runnable target = new CommunicationHandler(socket);
}
} catch (Exception ex) {
            ex.printStackTrace();
    } finally {
    try {
        serverSocket.close();
        } catch (IOException e) {
```

```
                e.printStackTrace();
            }
        }
    }
}
```

11. demo on UDM protocol, for socket programming

- 1. client side

```
package client;
public class Program {
    public static int port = 4576;

    public static void main(String[] args) {
    DatagramSocket socket = null;
        Scanner sc = null;
        try {
            socket = new DatagramSocket(); // Client Socket
            sc = new Scanner(System.in);
            String message = "";
            byte[] buffer = null;

            while (true) {
                System.out.print("C:Client  :   ");
                message = sc.nextLine();
                buffer = message.getBytes();
                DatagramPacket sendPacket = new DatagramPacket(buffer,
buffer.length,
                        InetAddress.getByName("localhost"), port);
                socket.send(sendPacket);

                buffer = new byte[1024];
                DatagramPacket receivePacket = new DatagramPacket(buffer,
buffer.length);
                socket.receive(receivePacket);
                message = new String(receivePacket.getData());
                System.out.println("S:Client    :   " + message);
            }

        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            sc.close();
            socket.close();
        }
    }
}
```

- 2. server side

```
package server;

public class Program {
    public static int port = 4576;

    public static void main(String[] args) {
        DatagramSocket socket = null;
        Scanner sc = null;
        try {
            socket = new DatagramSocket(port); // Server Socket
            sc = new Scanner(System.in);
            String message = "";
            byte[] buffer = new byte[1024];

            while (true) {
                DatagramPacket receivePacket = new DatagramPacket(buffer,
buffer.length);
                socket.receive(receivePacket);
                message = new String(receivePacket.getData());
                System.out.println("S:Client    :   " + message);

                System.out.print("S:Server  :   ");
                message = sc.nextLine();
                buffer = message.getBytes();
                DatagramPacket sendPacket = new DatagramPacket(buffer,
buffer.length, receivePacket.getAddress(),
                        receivePacket.getPort());
                socket.send(sendPacket);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            sc.close();
            socket.close();
        }
    }
}
```

11. Functional Programming using lambada expresison

- 1. method 1 to implement functional interface
- using class implement F/I

```
@FunctionalInterface
interface Printable{
    void print( );
}
```

```
class Test implements Printable{
    @Override
    public void print() {
        System.out.println("Inside print");
    }
}
public class Program {

    public static void main(String[] args) {
    Printable p = new Test();
    p.print();
  }
```

- 2. method 2 to implement functional interface
- using anonymous inner type

```
@FunctionalInterface
interface Printable{
    void print( );
}
public static void main(String[] args) {
        Printable p = new Printable() {

            @Override
            public void print() {
                System.out.println("Inside print");

            }
        };
    }
```

- 3. method 3 to implement functional interface
- using lambada expression

```
@FunctionalInterface
interface Printable{
    void print( );
}
public class Program {
public static void main(String[] args)
{
Printable p = ()-> System.out.println("hello lambada expression");
    p.print();
  }
}
```

- 4. F/I with parameter

```
//Printable p = ( String str )-> System.out.println(str);
//Printable p = ( String message )-> System.out.println(message);
//Printable p = ( str )-> System.out.println(str);
 Printable p = str -> System.out.println("hello " + str);
p.print("raj");
```

- 5. F/I parameter

```
@FunctionalInterface
interface Calci{void sum(int num1,int num2);}
public class Program {
public static void main(String[] args) {
Calci c = (int num1,int num2)->System.out.println("result : "+ (num1 +
num2));
c.sum(20, 30);
  }
}
```

- 6. F/I parameter

```
@FunctionalInterface
interface Calci{int sum(int num1,int num2);}
public class Program {
public static void main(String[] args) {
//Calci c = (int num1,int num2)->System.out.println("result : "+ (num1 +
num2));
Calci c =   ( int num1, int num2 )-> num1 + num2;
    int result = c.sum(20, 30);
    System.out.println(result);
    }
}
```

- 7. F/I by multi line lambada expression using curly braces {}

```
@FunctionalInterface
interface Calculator{
    int factorial( int number );
}
public class Program {
    public static void main(String[] args) {
        Calculator c = number-> {
            int result = 1;
            for( int count = 1; count <= number; ++ count )
                result = result * count;
            return result;
        };
```

```
        int result = c.factorial(5);
        System.out.println(result);
    }
}
```

- 8. using lambada Expression in List

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
public class Program {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>( );
        list.add(10);
        list.add(20);
        list.add(30);
//void accept(T t)
//Consumer<Integer> action = ( Integer number )->
System.out.println(number);
//Consumer<Integer> action = ( number )-> System.out.println(number);
//Consumer<Integer> action = number -> System.out.println(number);
//list.forEach(action);

        list.forEach(number->System.out.println(number));
    }
}
```

- 9. using :: operator

```
public class Program {
    public void displayRecord( ) {
        System.out.println("Inside print");
    }
    public static void main(String[] args) {
        Program prog = new Program();
        Printable p = prog::displayRecord;
        p.print();
    }
    public static void showRecord( ) {
        System.out.println("Inside print");
    }
    public static void main4(String[] args) {
        Printable p = Program::showRecord;
        p.print();
    }
}
```

- 10. using list forEach method using lambada expression

```
public class Program {
    public static void print( Integer number ) {
        System.out.println(number);
    }


    public static void main(String[] args) {

        List<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(20);
        list.add(30);

        //list.forEach(Number->System.out.println(Number));
       //   list.forEach(Program::print);
     list.forEach(System.out::println);
    }
}
```