day 1

- ppt day 1

day 2

- ppt day 2

day 3

+ 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 ”);
       ```

+ 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”);
  }
}
```

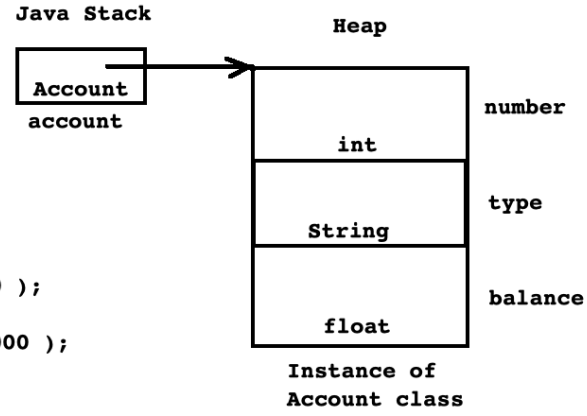
- If we create instance w/o reference then it is called anonymous instance.
- Example:
  new Account( ); //anonymous instance
- If we want to use any instance only once then we should create anonymous instance.
- Example:
  Person p = new Person( );
  p.setName("Sandeep");
  p.setBirthDate(new Date(23,7,1983));

- 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;

day 4

+ 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.**

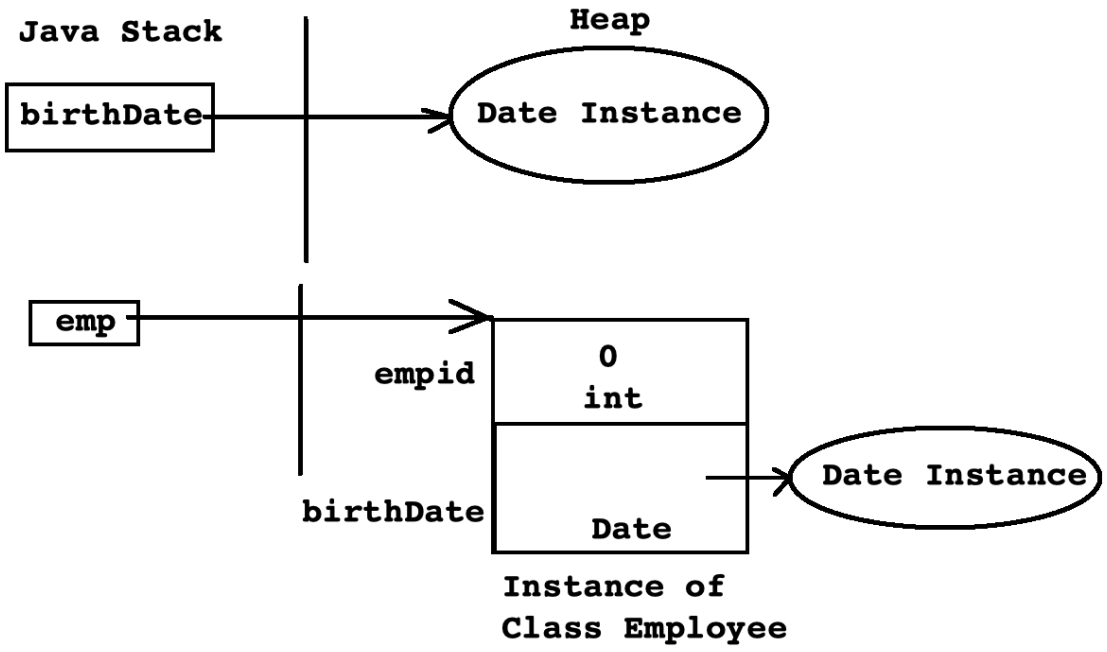
4. **Variable(field) of value type by default contains 0.**
4. **Variable(field) of reference type by default contains null.**

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.

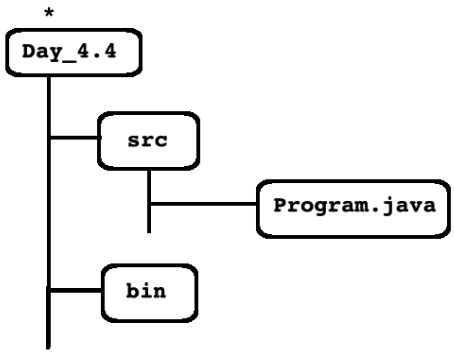
+ Package
  - package is a java language feature which is used:
    1. To group functionally related/equivalent types together.
    2. To avoid name collision/ambiguity.
  - package is keyword in java.
  - If we want to define any type(interface/class) inside package then we should use package declaration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

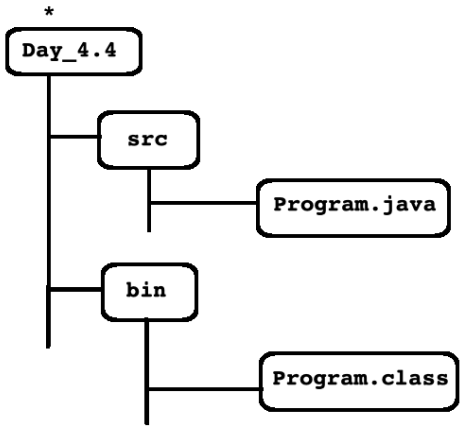
```
package test;
class Program{
  public static void main( String[] args ){
    //TODO
  }
}
```

  - 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

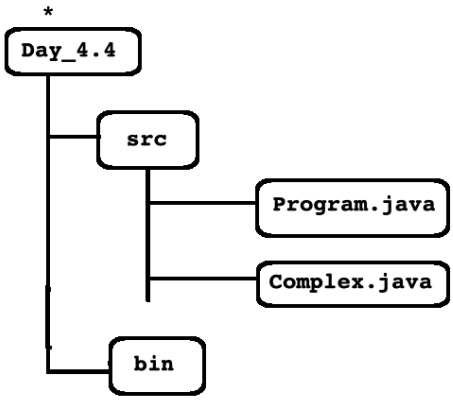


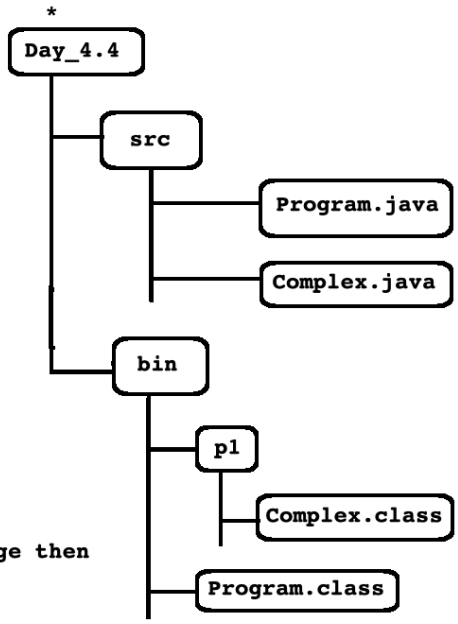



```
export PATH=/usr/bin (optional)
javac -d ./bin/ ./src/Program.java
export CLASSPATH=./bin/
java Program
```

```
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**

day 5

- 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.

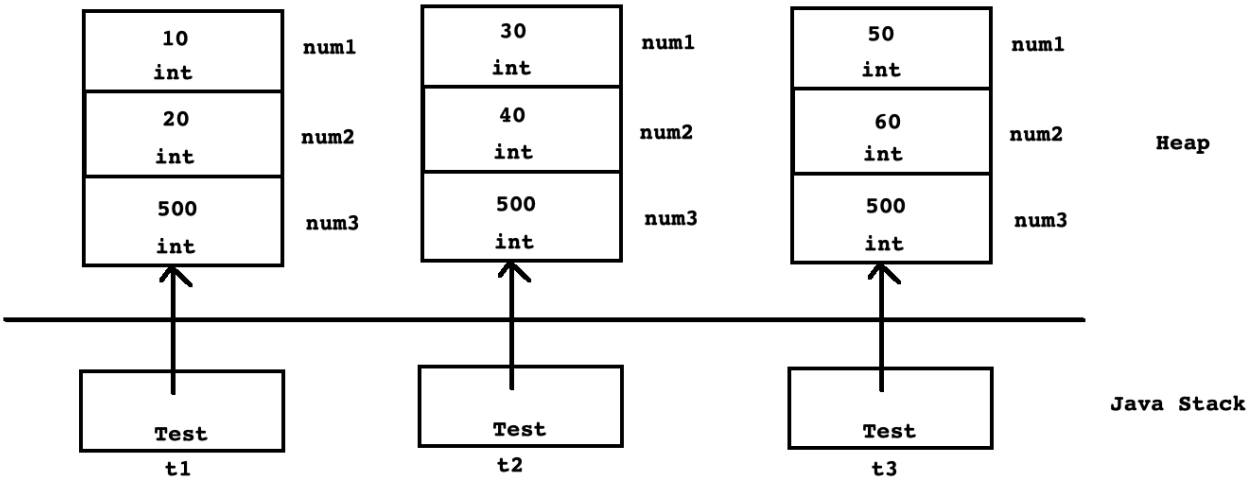
- Static data member declared inside class is called class level variable.
- Static member function declared inside class is called class level method.
- class level members are designed to access using classname and dot operator.

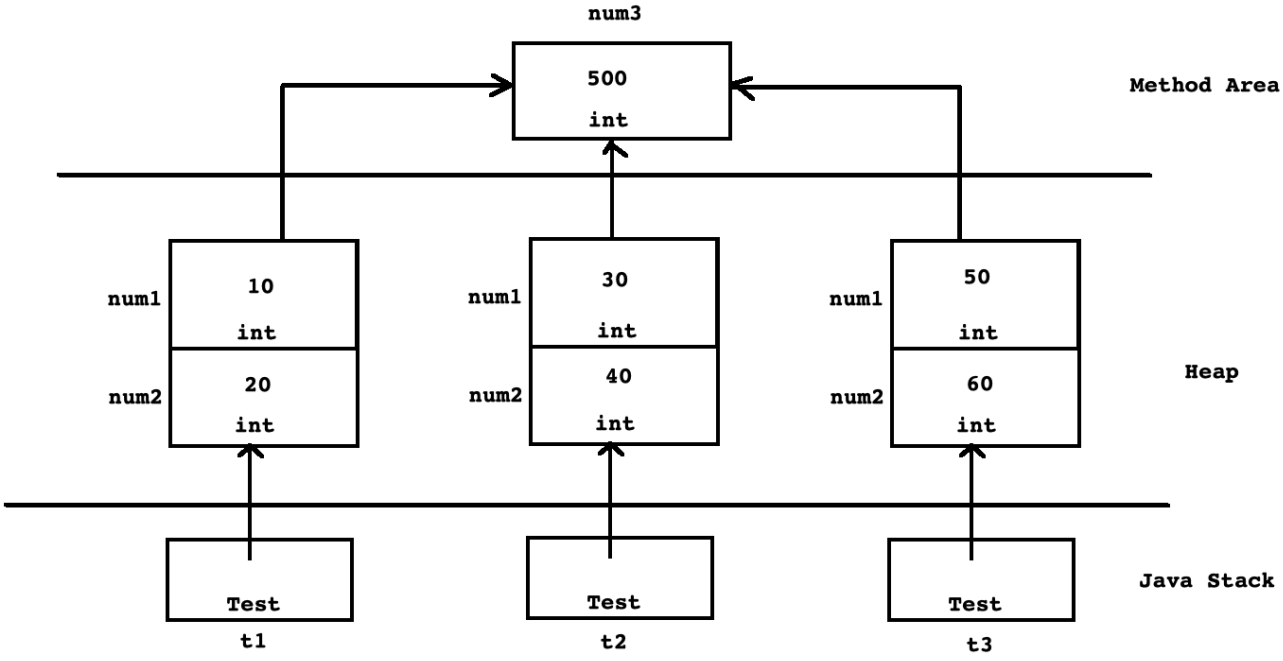
```
class Test{
  static int number;
  public static void print( ){
    System.out.println(Test.number);
  }
}

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

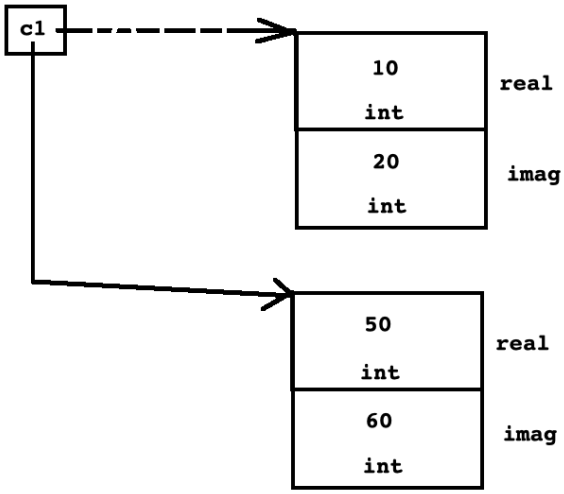
- If we want share value of any field in all the the instances of same class then we should declare such field static.
- static field do not get space inside instance.
- Static field get space once per class during class loading on method area.

```
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.

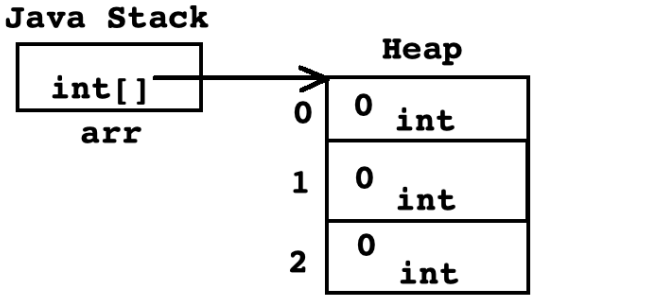

10
int
num1
20
int
num2
500
int
num3
30
int
num1
40
int
num2
500
int
num3
50
int
num1
60
int
num2
500
int
num3
Heap
Test
t1
Test
t2
Test
t3
Java Stack


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

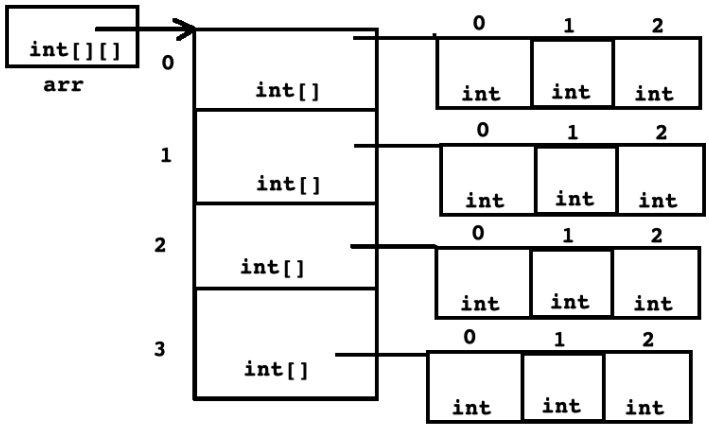


+ Array
  - It is reference type. In other words, to create instance of array it is necessary to use new operator.
  - Three Types of Array:
    1. Single dimensional array
    2. Multi dimensional array
    3. Ragged array
  - Array bounds checking is a job of JVM.
  - If we want to process elements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

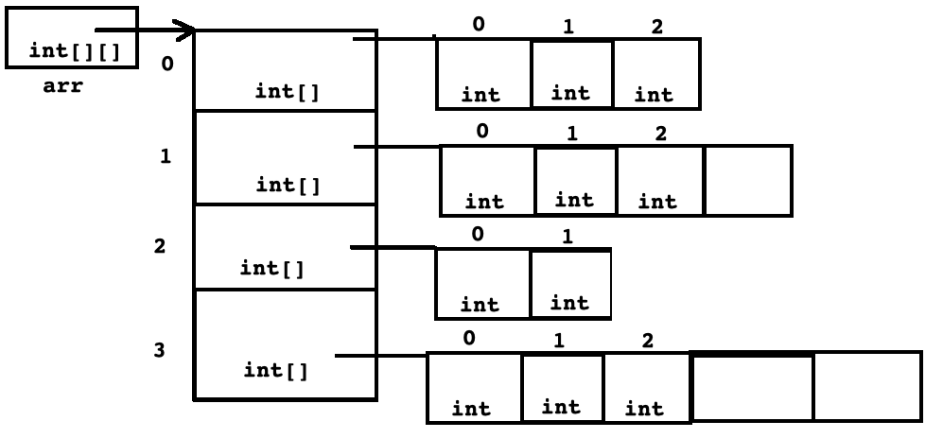
+ Single dimensional array

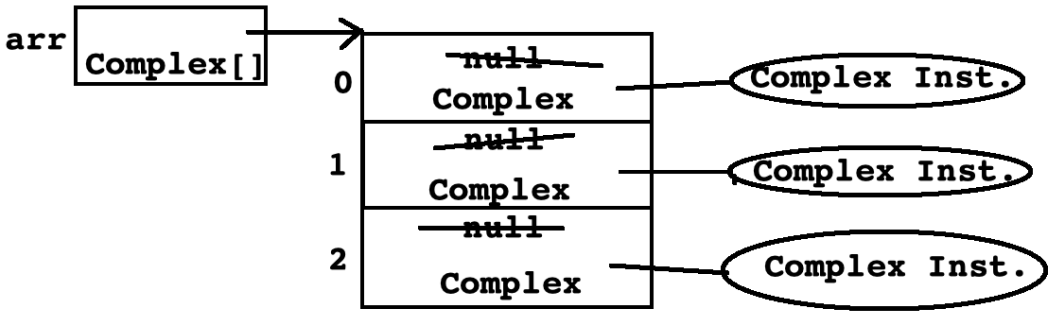


Single dimensional
Array of integer



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();
```

day 6

+ Variable arity method / variable argument method

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

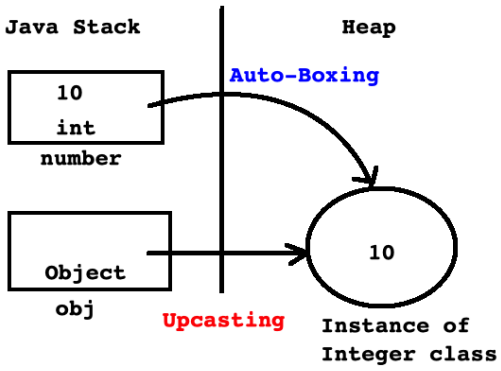
  - Example(Java)
    1. public PrintStream printf(String format, Object... args);

    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++):

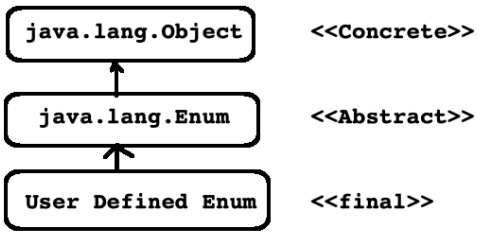
```
enum Color
{
  RED, GREEN, BLUE  //Enum constant/enumerators
  //RED=0, GREEN=1, BLUE = 2
};
enum Day
{
  SUN=1, MON=2, TUES, WED, THURS, FRI, SAT
};
```

  - 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
```

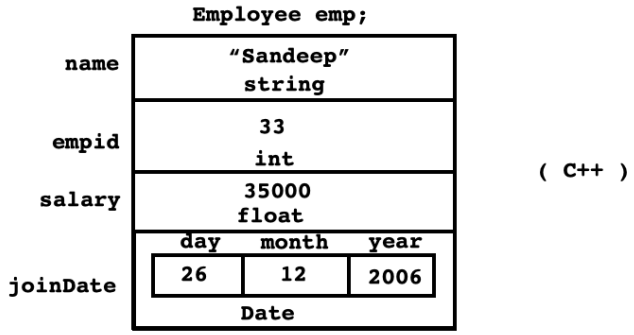
+ Hierarchy
  - Level of abstraction is called hierarchy.
  - Types:
    1. Has-a : Association
    2. Is-a : Inheritance
    3. Use-a : Dependancy
    4. Creates-a : Instantiation

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

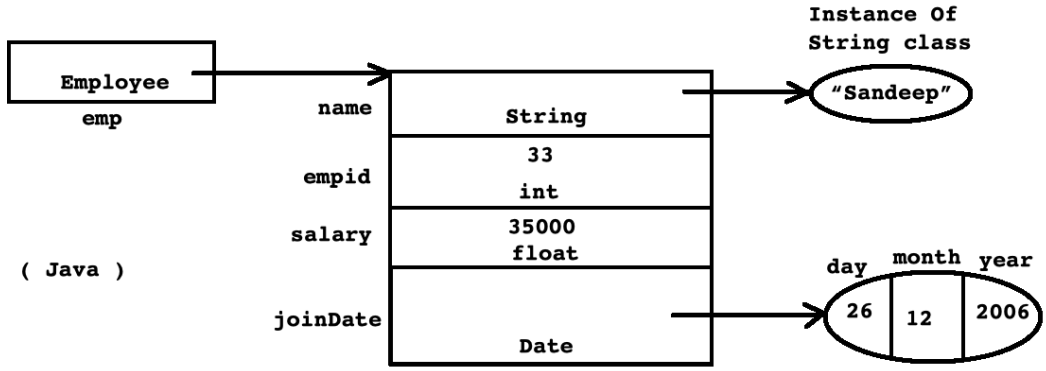
```
class Employee
{
  string name;  //Association
  int empid;
  float salary;
  Date joinDate; //Association
};
Date joinDate(26,12,2006)
Employee emp(“Sandeep”,33,35000,joinDate);
```

```
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.



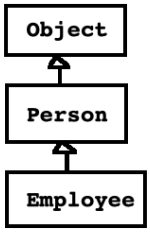
(C++)



(Java)

- 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.



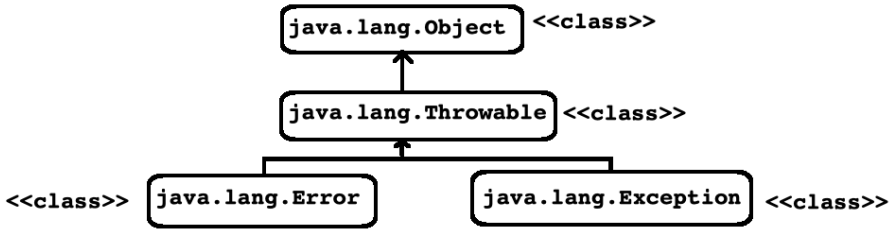
  - Process of converting reference of sub class into reference of super class is called upcasting.

```
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.

day 7

+ 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.**

day 8

+ Final Method

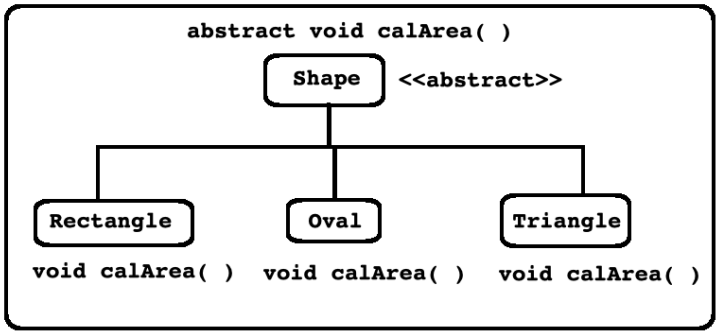
- **If implementation of super class method is logically 100% complete then we should declare super class method final.**
- **Final method is concrete method that we can not override it in sub class.**
- **Final method inherit into sub class hence we can call it on instance of super class as well as 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**

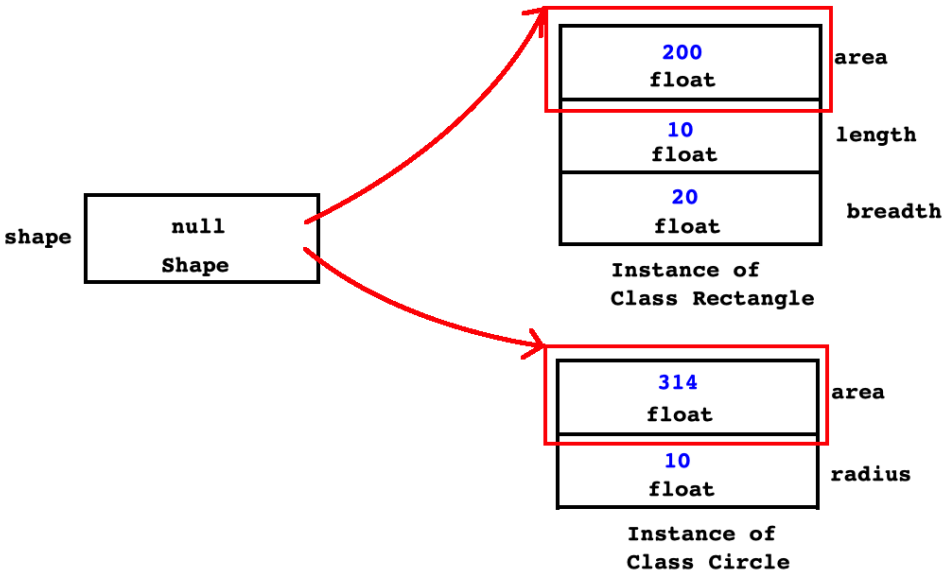


+ Final Class
  - If implementation of any class is logically 100% complete then we should declare such class final.
  - Final class is a concrete class i.e we can instantiate it.
  - We can not extend final class i.e we can not create sub class of final class.
  - Final class can have super class.
  - 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.

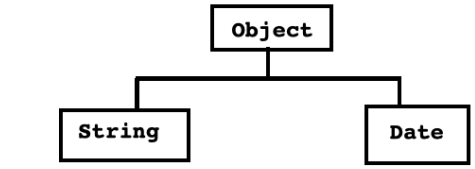


+ In case of upcasting (Shape sh = new Rectangle()), using super class reference:
  1. We can access fields of super class
  2. We can access method of super class
  3. We can not access fields of sub class
  4. We can not access non overriden methods of sub class
  5. We can access overriden methods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

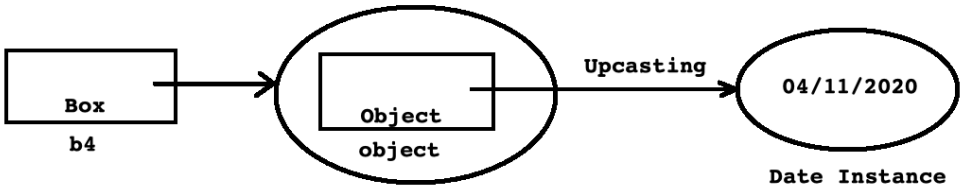
+ **Generics**
  - **If we want to write generic code in Java then we can use:**
    1. **Either java.lang.Object class**
    2. **Or Generics.**

+ **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
   }
}
```

day 9

+ **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);
   }
}
```

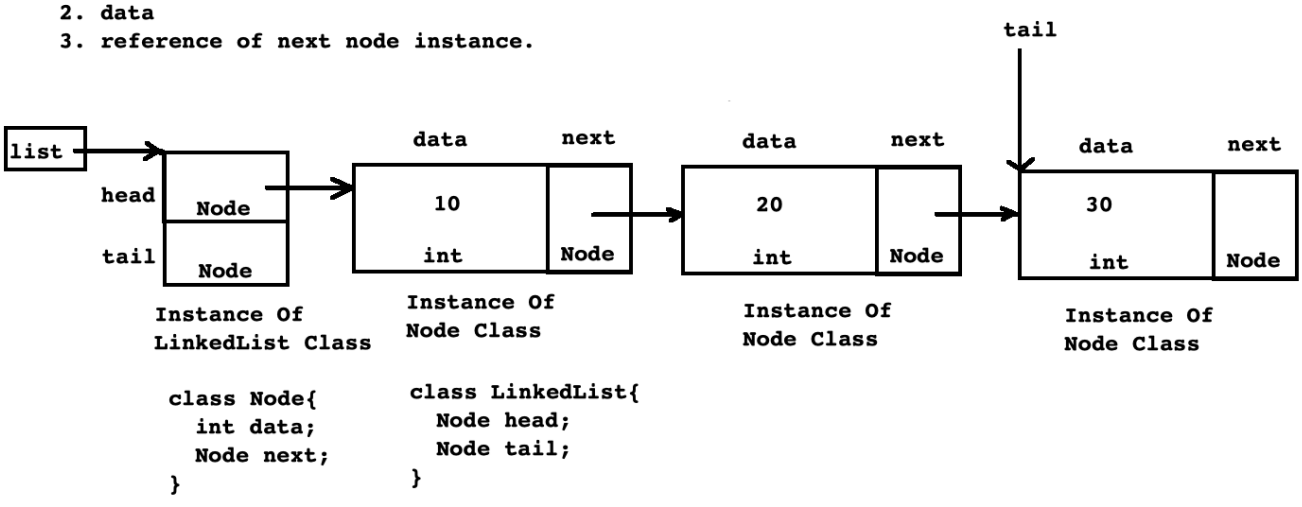
In above code, list can contain reference ArrayList, which can contain Number and its sub type

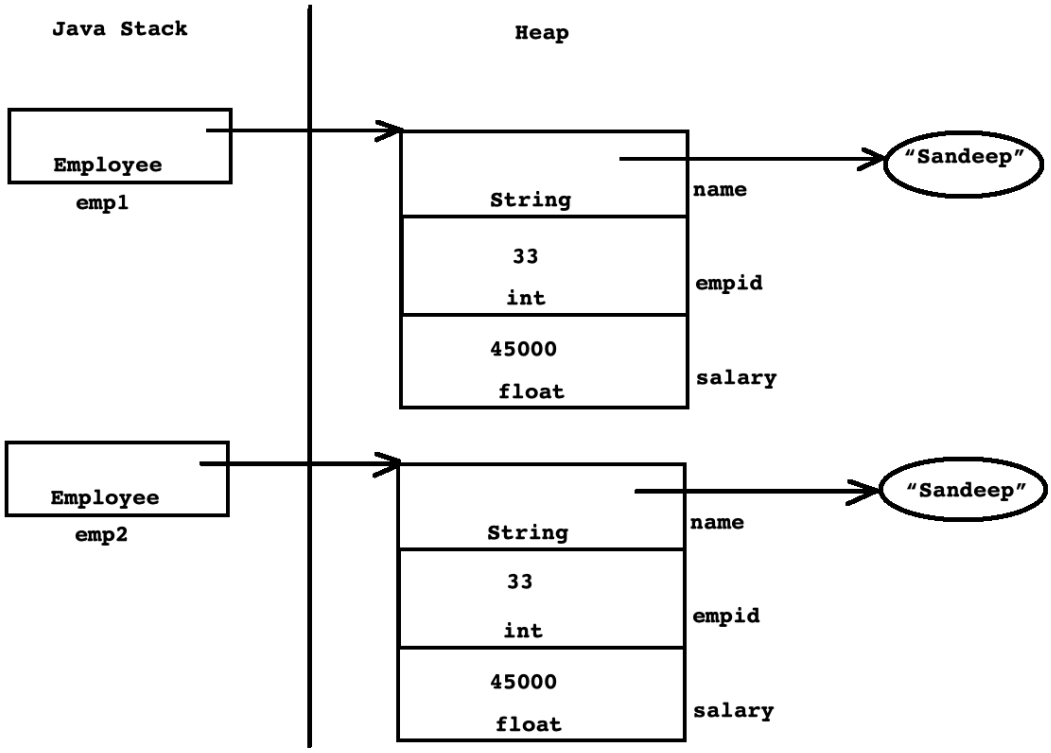
+ **Lower Bounded Wild Card:**

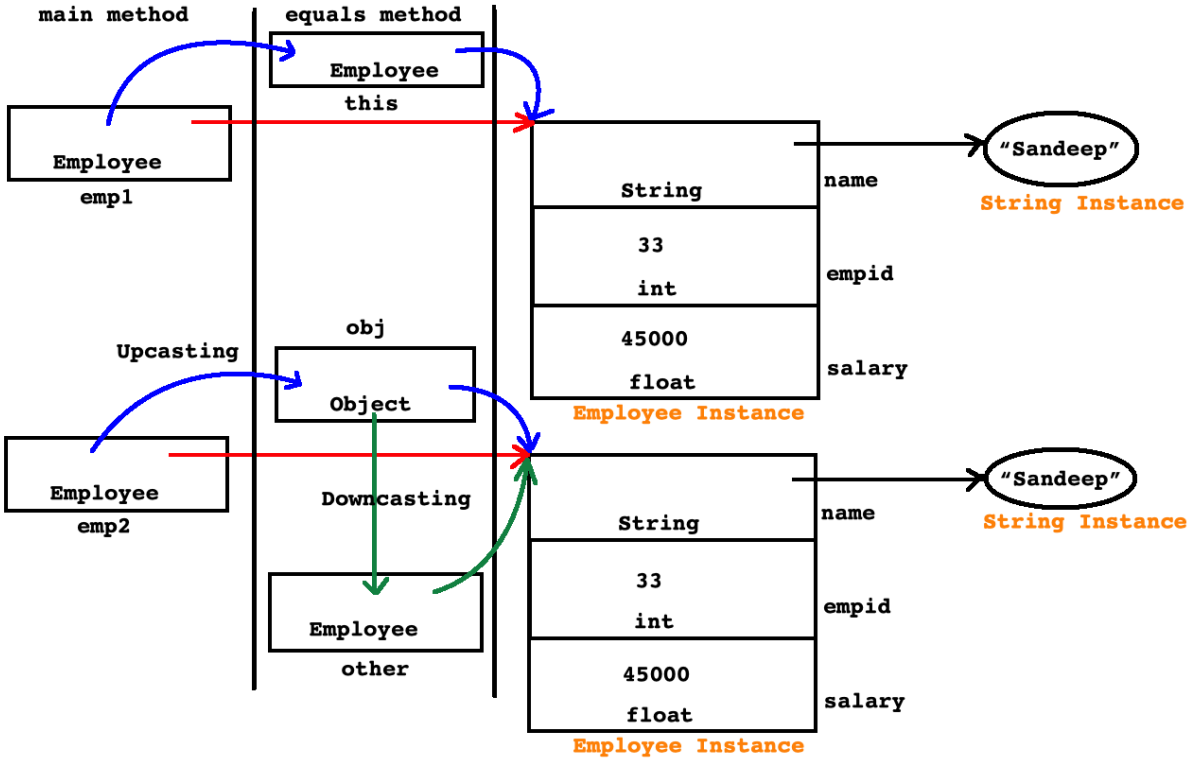
```
private static void printList(ArrayList<? super Integer> list) {
   if( list != null ) {
      for( Object element : list )
         System.out.println(element);
   }
}
```

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.**







+ 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:

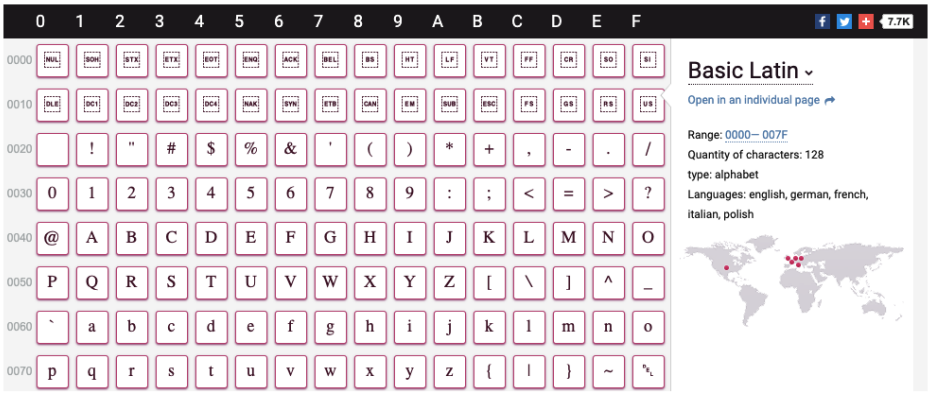
```
public boolean equals(Object obj) {
    return (this == obj); //Compares object references.
}
```

  - 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;
}
```

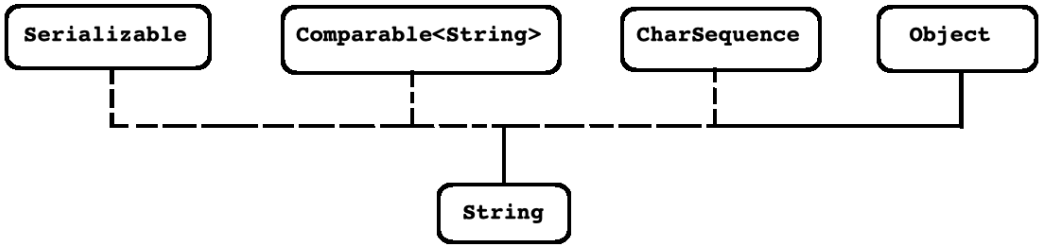
  - In Collection framework, it is used to compare state of instance.

+ 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/



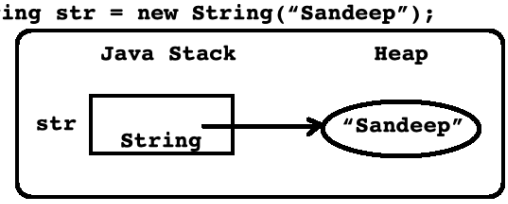
+ String Handling
  - We can use following classes to manipulate String:
    1. java.lang.String
    2. java.lang.StringBuffer
    3. java.lang.StringBuilder
    4. java.util.StringTokenizer
    5. java.util.regex.Pattern
    6. java.util.regex.Matcher

+ String

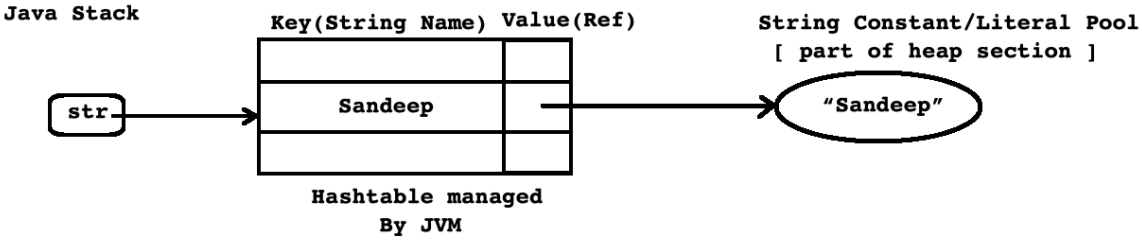


  - java.io.Serializable is a marker interface.
  - "int compareTo(String o)" is a method of java.lang.Comparable interface.
  - java.lang.CharSequence contains following methods:
    1. char charAt(int index);
    2. int length( );
    3. CharSequence subSequence(int start, int end);
  - java.lang.Object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

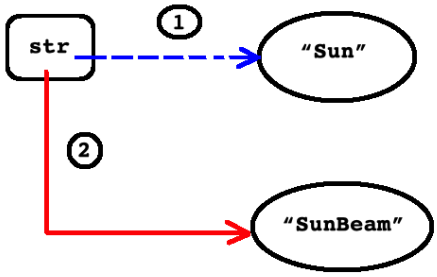
+ java.lang.String class
  - String is not built in type in Java. It is reference type whose instance can be created with and without new operator.
  - Consider Example 1:
    String str = new String("Sandeep");



  - 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
```

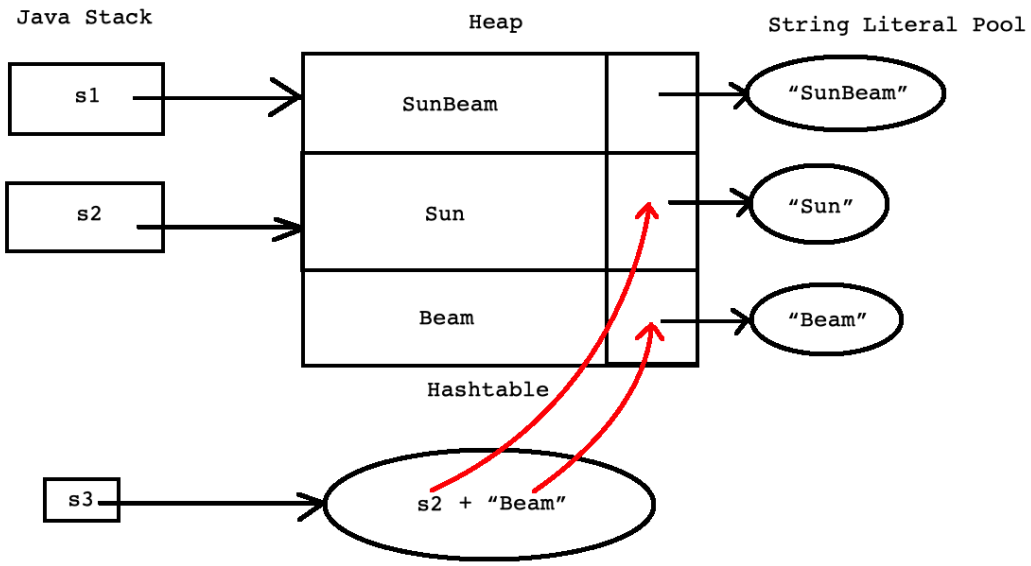
```
String str = “Pune-”;
str = str.concat(411057); //NoT OK
str = str.concat(“, India”); //OK
```

+ 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):

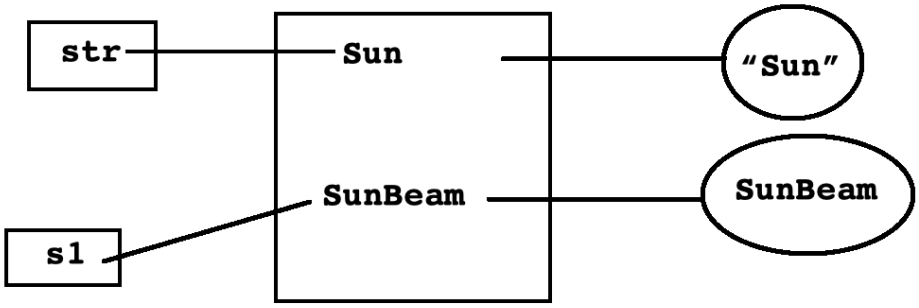
```
1. public String()
//String str = new String( );

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”);
```



day 10

+ 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**

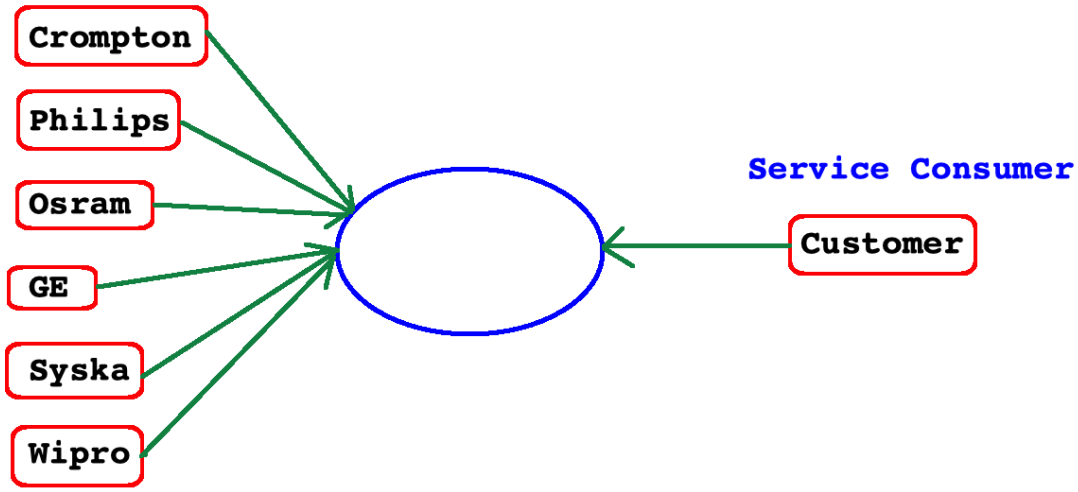
+ Method Local Anonymous Inner class

- **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.**

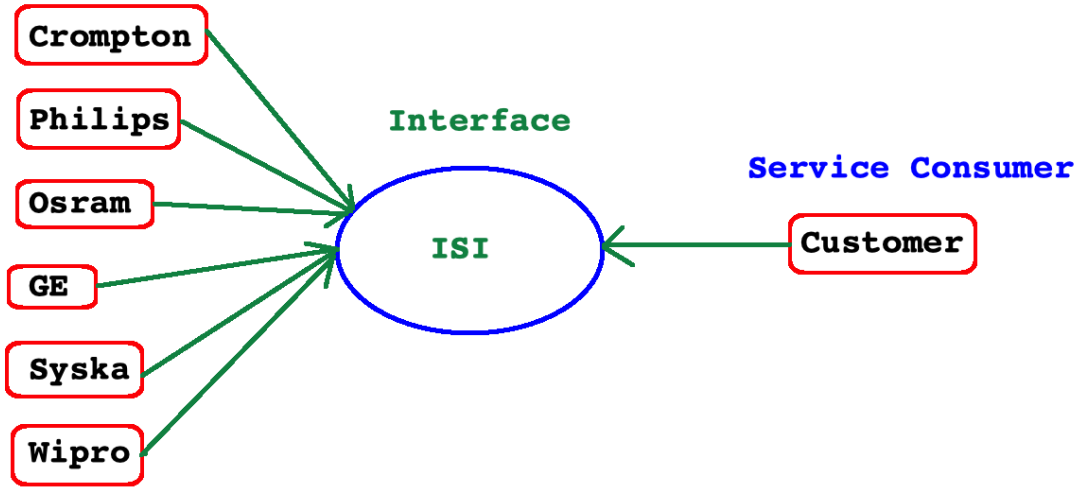
- **Syntax:**

```
Object obj; //reference
new Object( ); //Anonymous instance
Object obj = new Object(); //Instance with reference
Object obj = new Object( ) { }; //Anonymous inner class
```

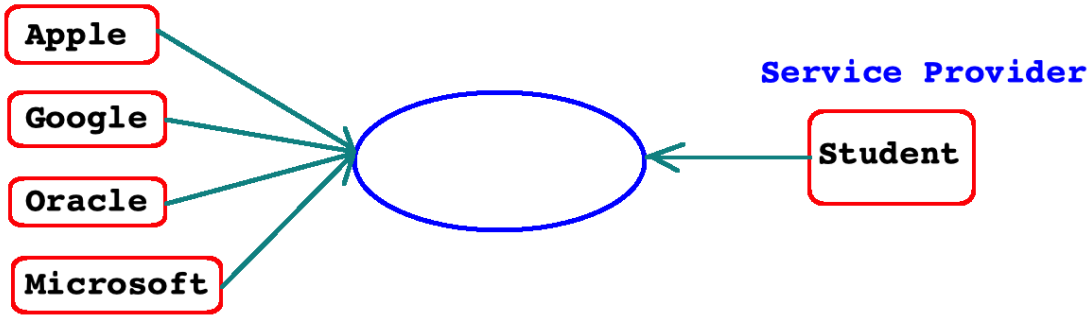
Service Providers



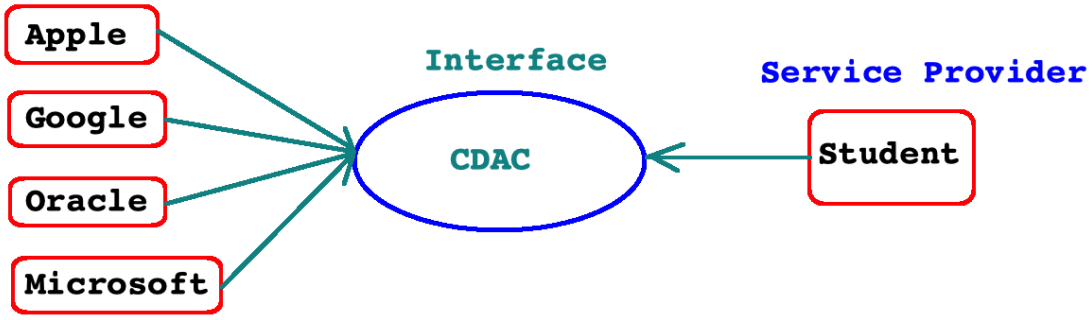
Service Providers

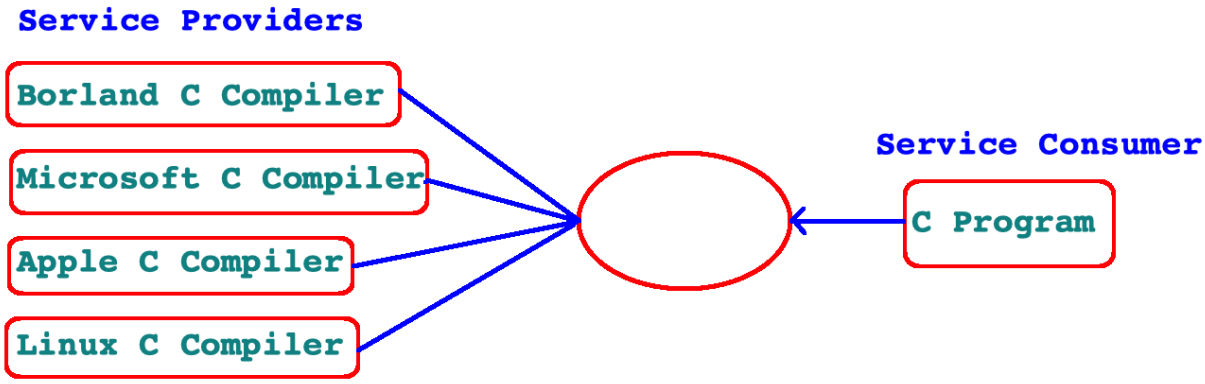


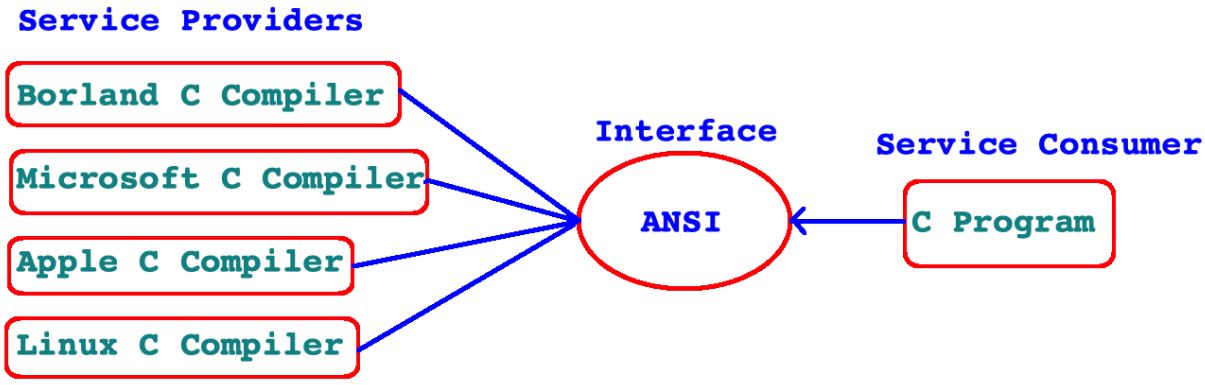
Service Consumer



Service Consumer




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
}
```

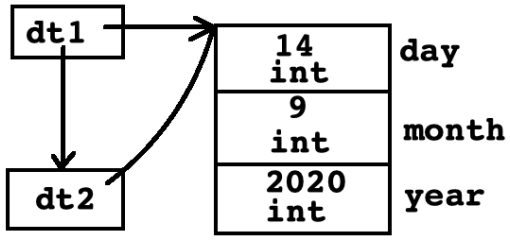
```
class C implements A{
  @Override
  public void print(){
    //TODO
  }
}
```

- **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**

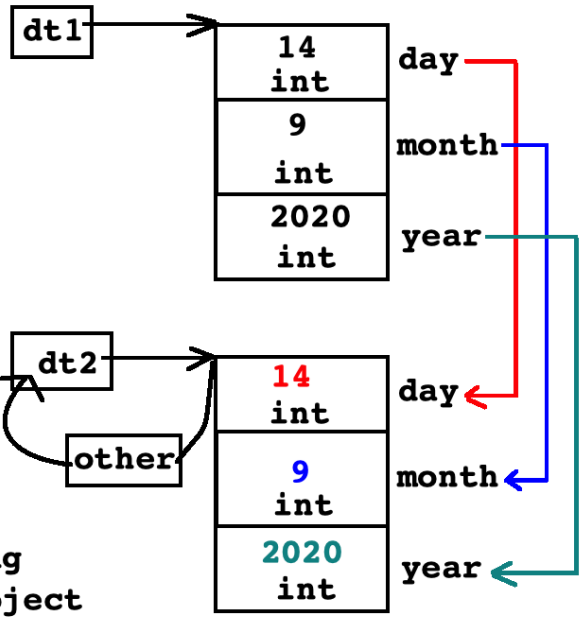
- **Interface can extends one / more than one interfaces.**
- **Class can implement one / more than one interfaces.**
- **Class extend only one class.**
- **First we should extend the class then implement I/F.**

- **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



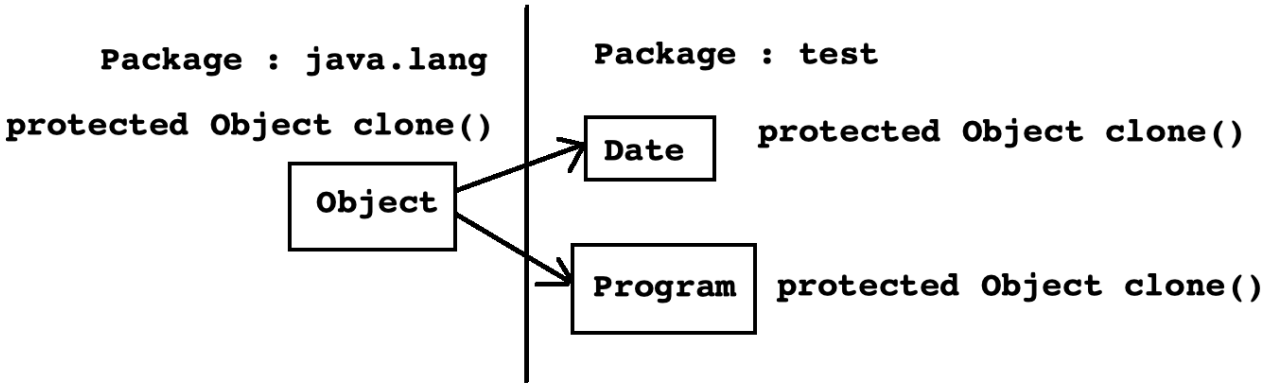
Shallow Copy
of Instances

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();
```

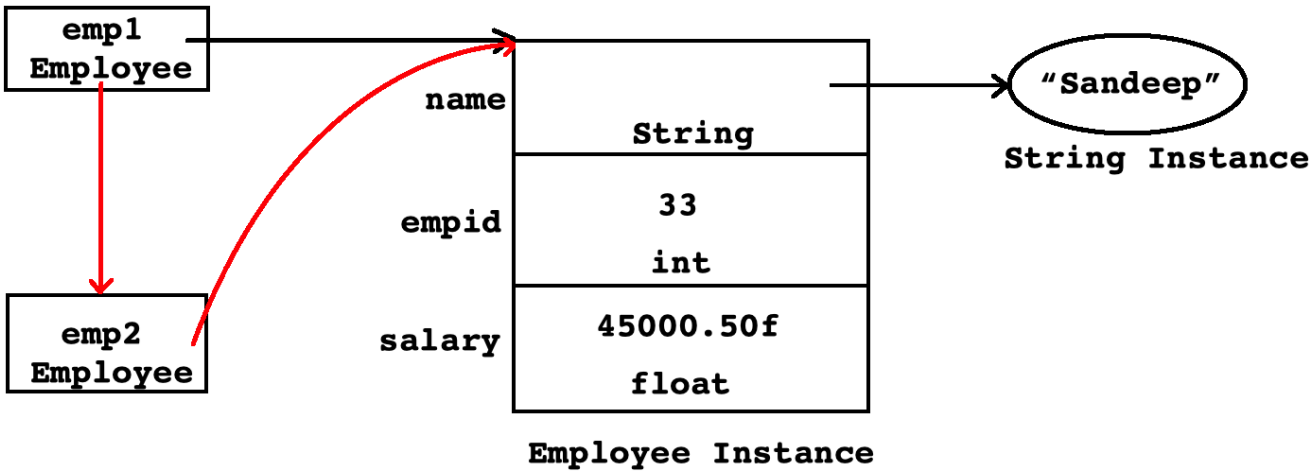
//NOT OK

- If we want to create shallow copy of current instance then we should use “super.clone()” method.

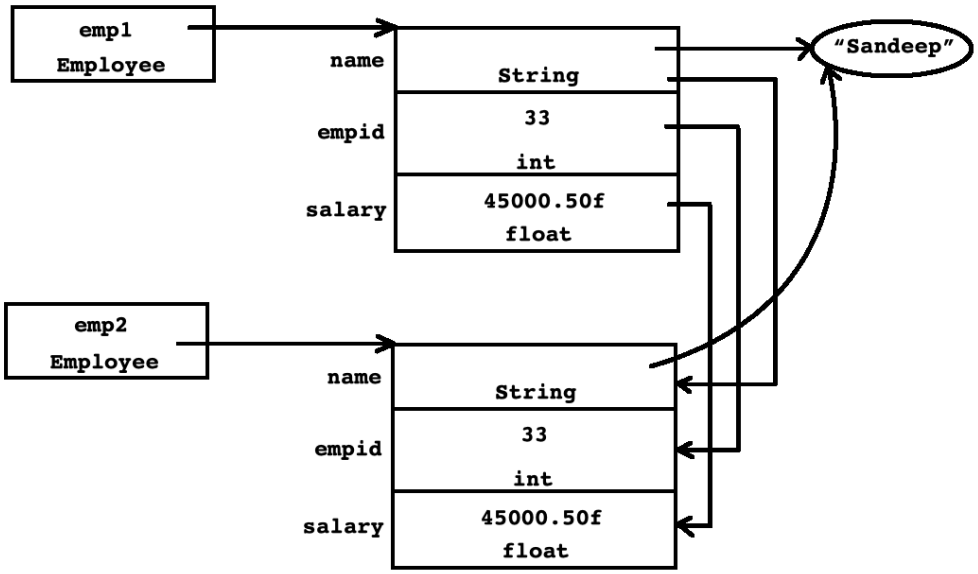
- If we want to clone an object then type of object/instance must implement java.lang.Cloneable interface.
- An interface which do not contain any member is called marker/tagging interface.
- Such interfaces are used to generate metadata for JVM.
- Example:
  1. java.lang.Cloneable
  2. java.util.RandomAccess
  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.

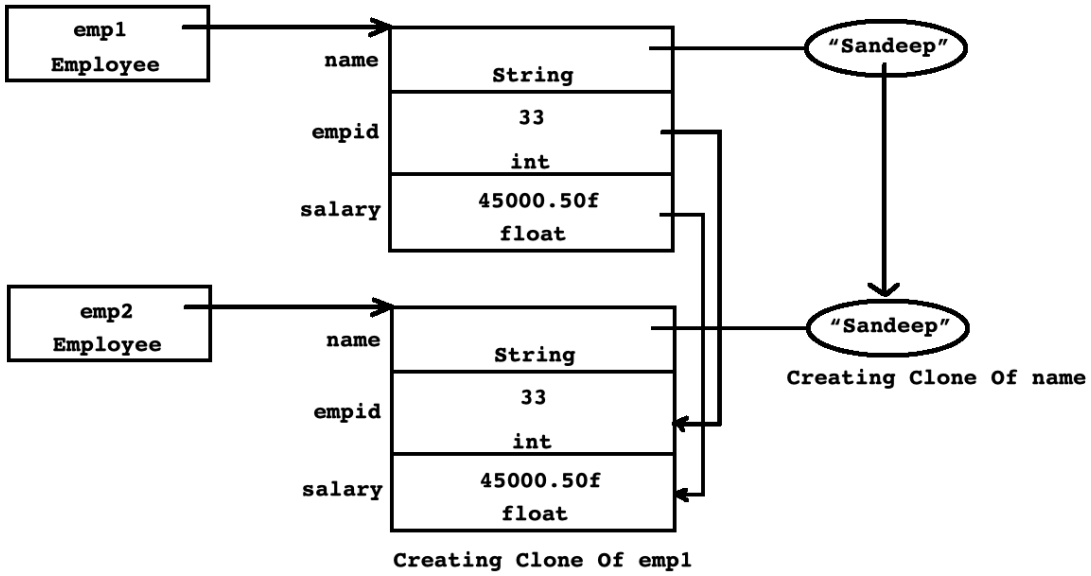
```
Employee emp1 = new Employee(“Sandeep”,33,45000.50f);
Employee emp2 = emp1; //Shallow Copy of reference
```

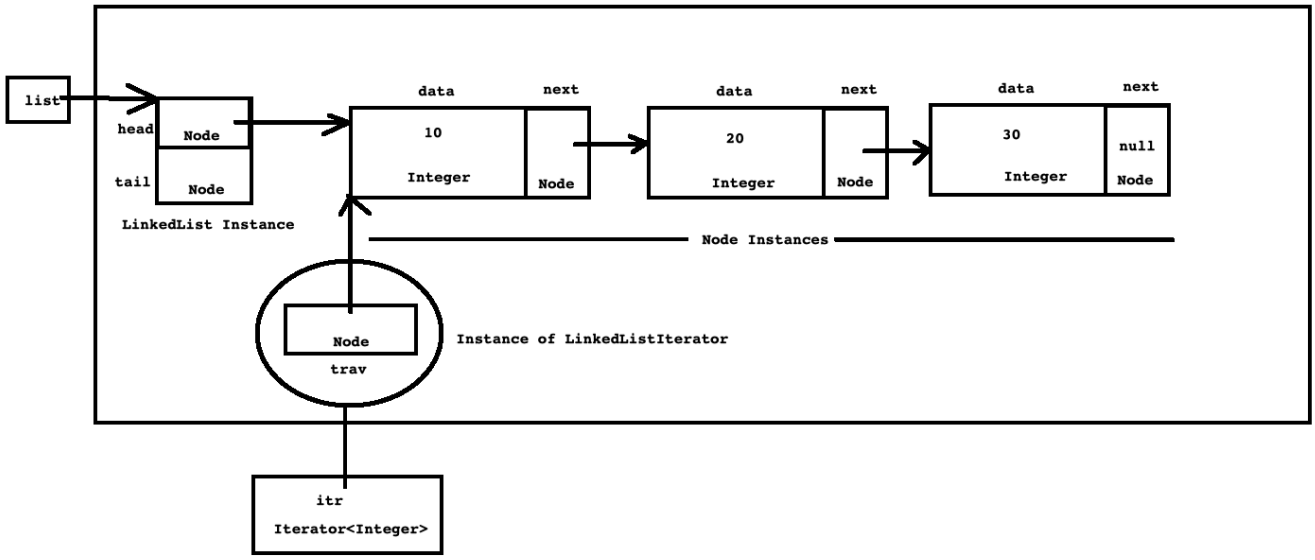


```
Employee emp1 = new Employee(“Sandeep”,33,45000.50f);
Employee emp2 = (Employee)emp1.clone( );//Shallow Copy of Instance
```



Deep Copy Of Instance





- 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.

day 11

+ 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

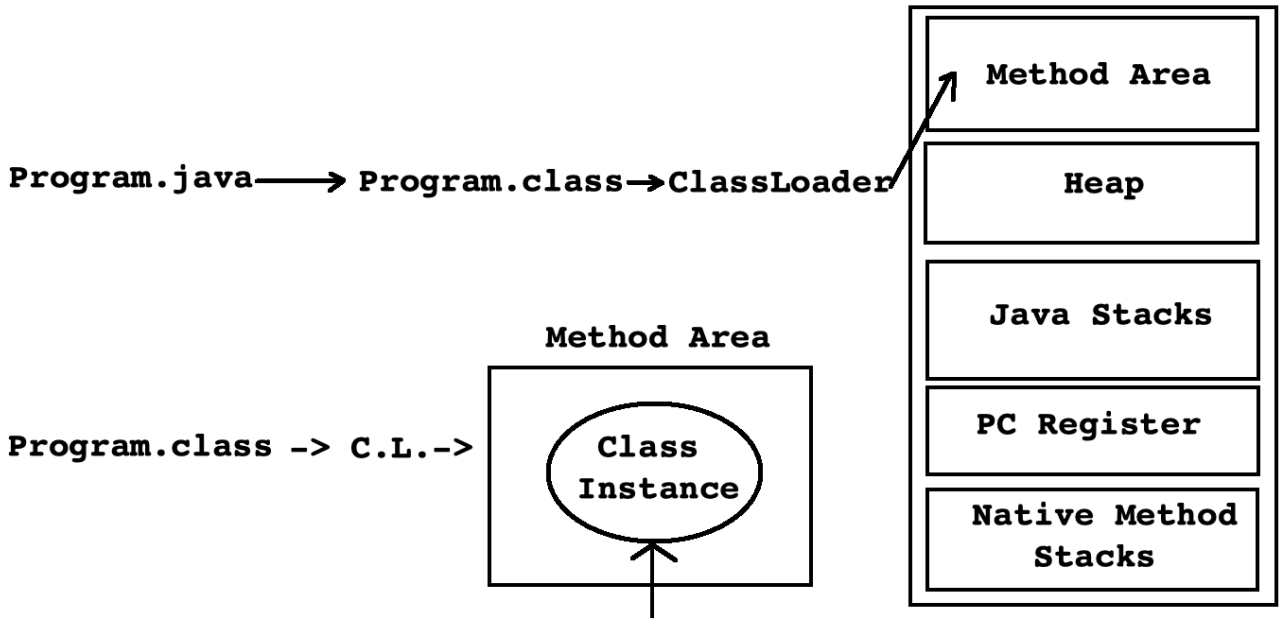
1. **What is name of the field**
2. **Which is modifier of field**
3. **What is the type of field**
4. **Which annotations are used on field**
5. **Whether field is inherited / declared only field**

+ 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

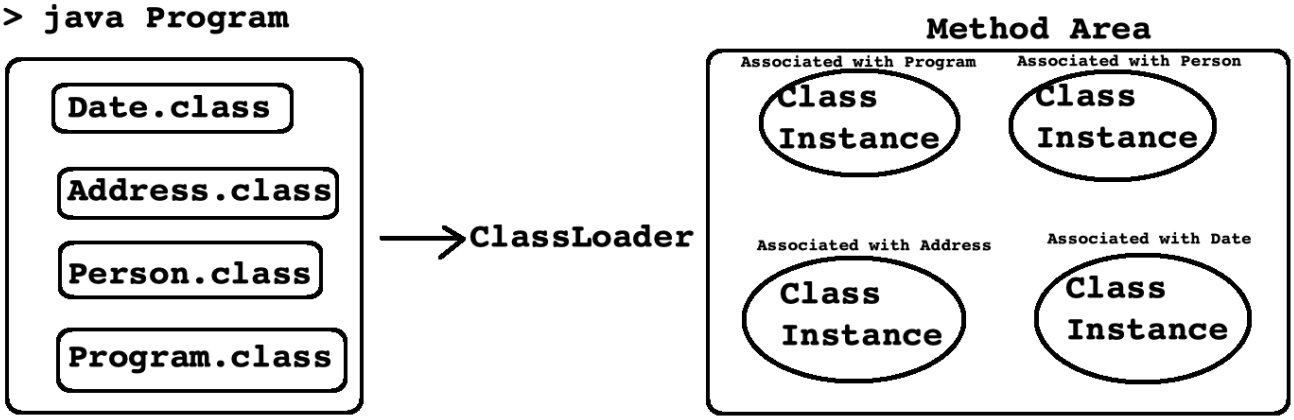


java.lang.Class class instance associated with Program class

```
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);
```

day 12

Collection

+ 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.

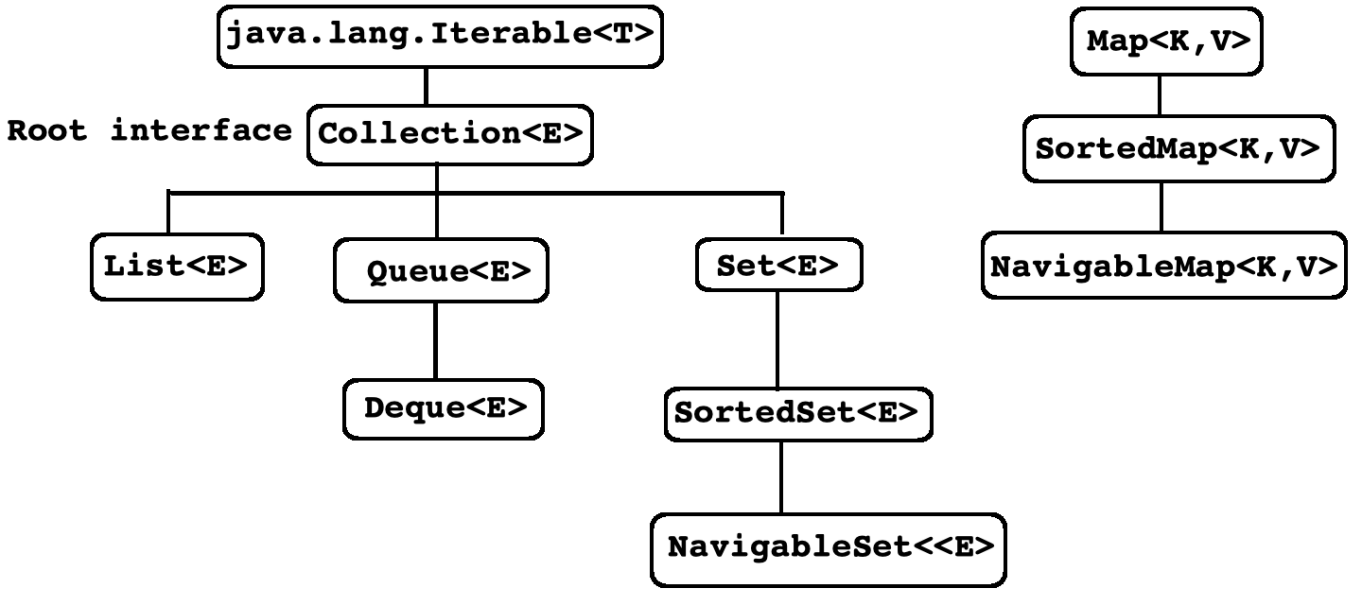
```
@Table(name="employees")
class Employee{
 @Column(name="emp_name", columnDefinition="VARCHAR(100)")
 private String name;
 @Id
 @Column(name="emp_id", columnDefinition="INT")
 int id;
 @Column(name="sal", columnDefinition="FLOAT")
 float salary;
};
```

```
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>



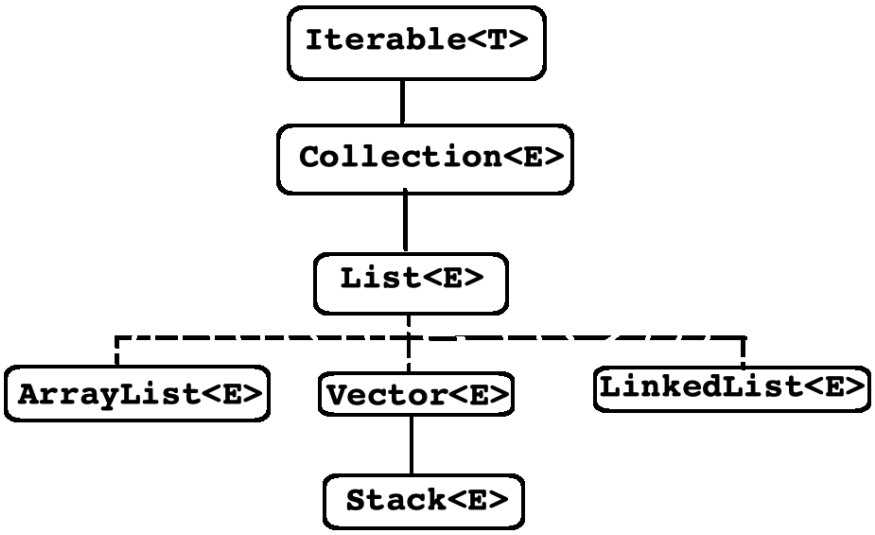
+ Iterable
  - It is an I/F declared in java.lang package
  - Implementing this interface allows an object to be the target of the "for-each loop" statement.
  - Method:
    1. abstract Iterator<T> iterator()
    2. default Spliterator<T> spliterator()
    3. default void forEach(Consumer<? super T> 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)

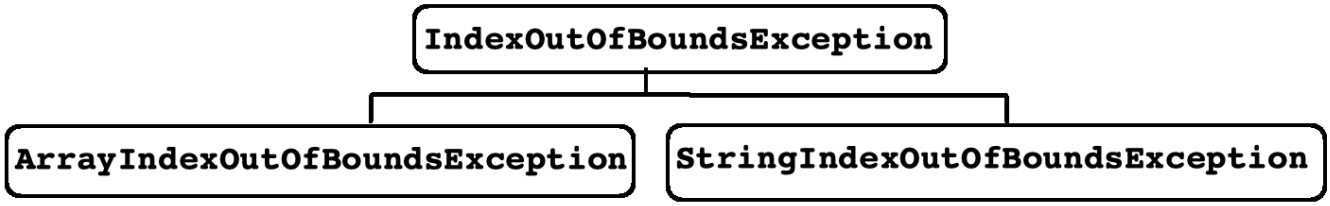
+ ArrayList<E>
  - It is a list collection which implements List<E>, RandomAccess, Cloneable and Serializable interface.
  - ArrayList is resizable unsynchronised collection.
  - Since it is List<E> collection it gets all the properties of List.
  - If we do not specify capacity then its default capacity 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()

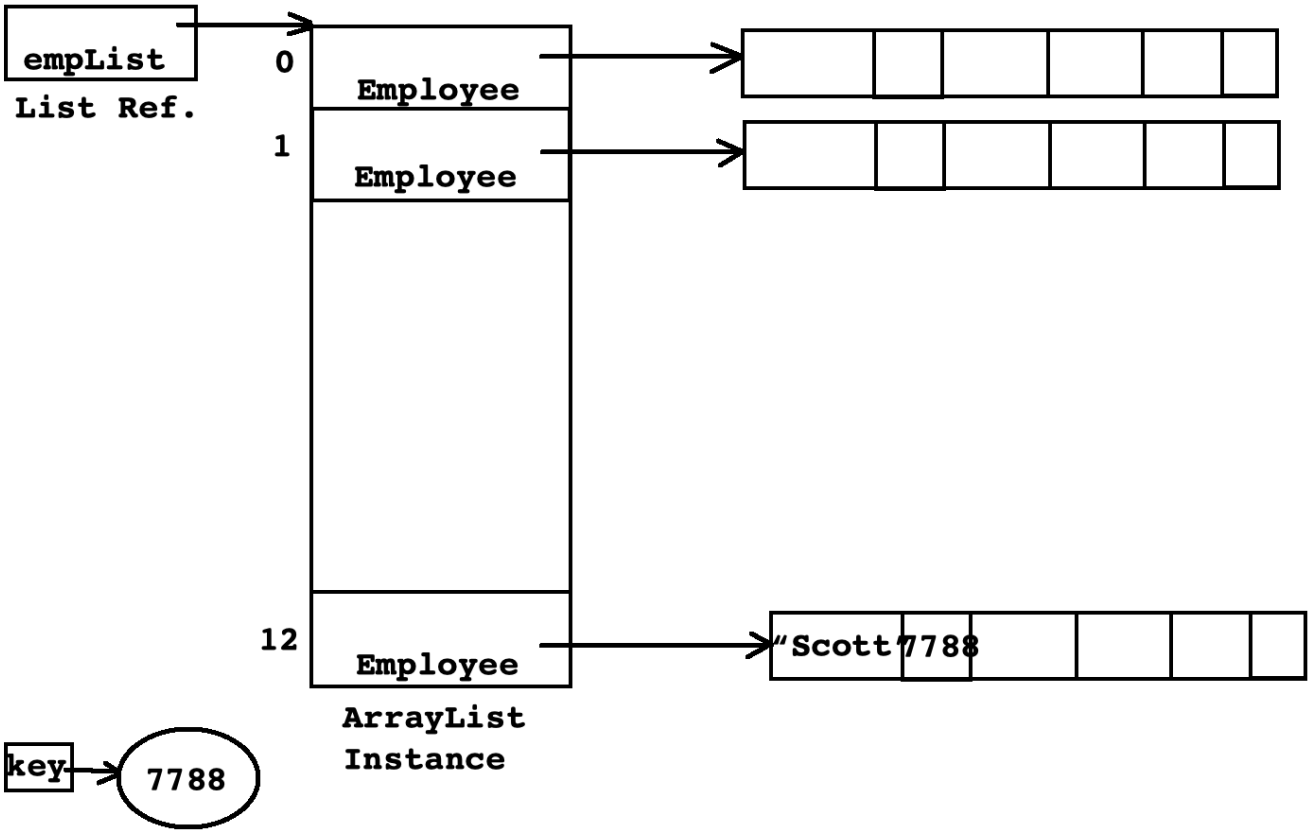
- In Java, Collection is not a group of objects rather it is group of references.

+ 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.

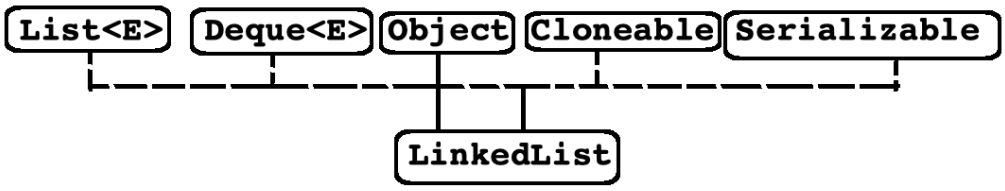
Q. What do you know about ConcurrentModificationException? Or
Q. Which are the types of Iterator?
Ans: 1. Fail-Fast Iterator 2. Not Fail-Fast/Fail-Safe Iterator

+ 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<>( );



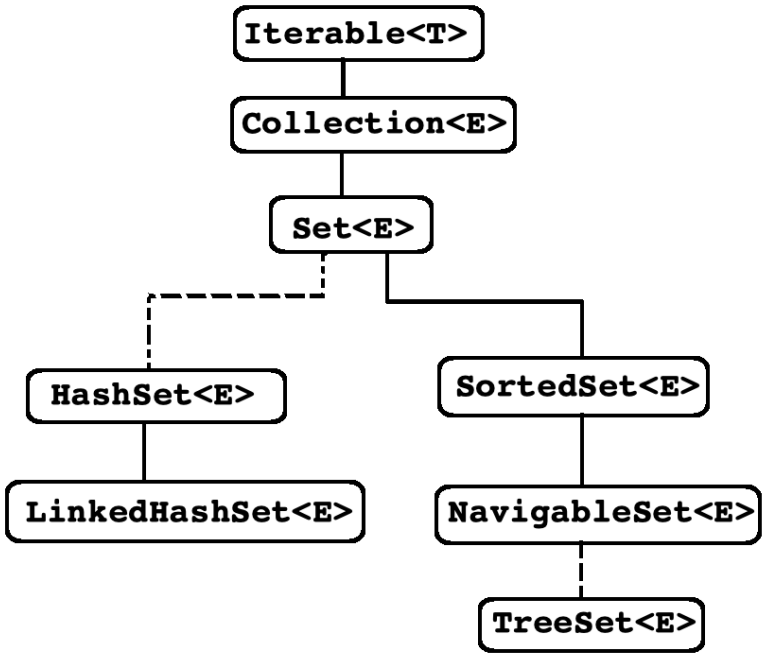
Q. What is the different between ArrayList and LinkedList?

+ 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.**

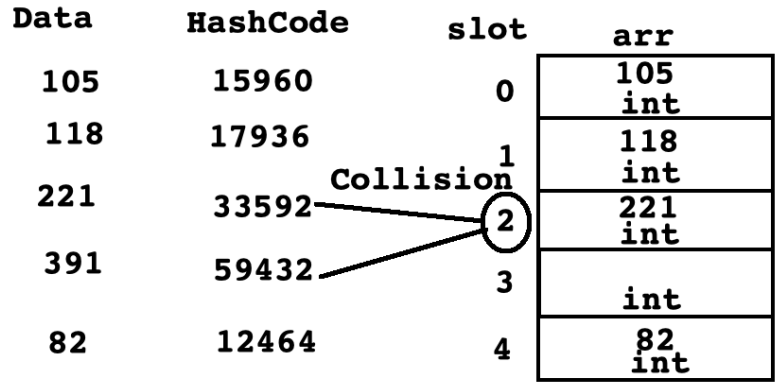
+ Binary Search

- **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

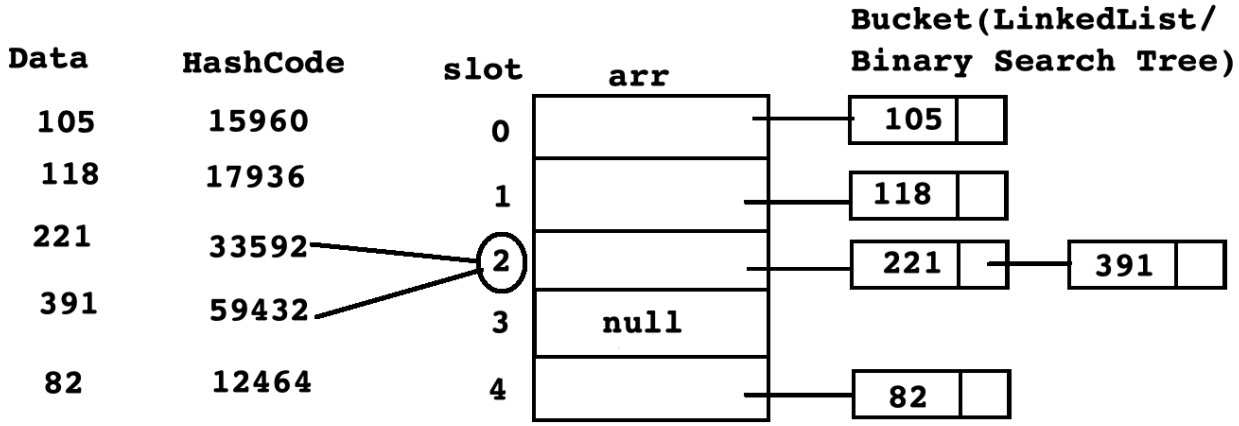
- **If we want to search every element in constant time then we should use hashing. In other words to search element very fast we should use Hashing.**
- **Hashing technique is based on hashcode.**
- **Hashcode is not an index or address/reference.**
- **HashCode is logical integer number that can be 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

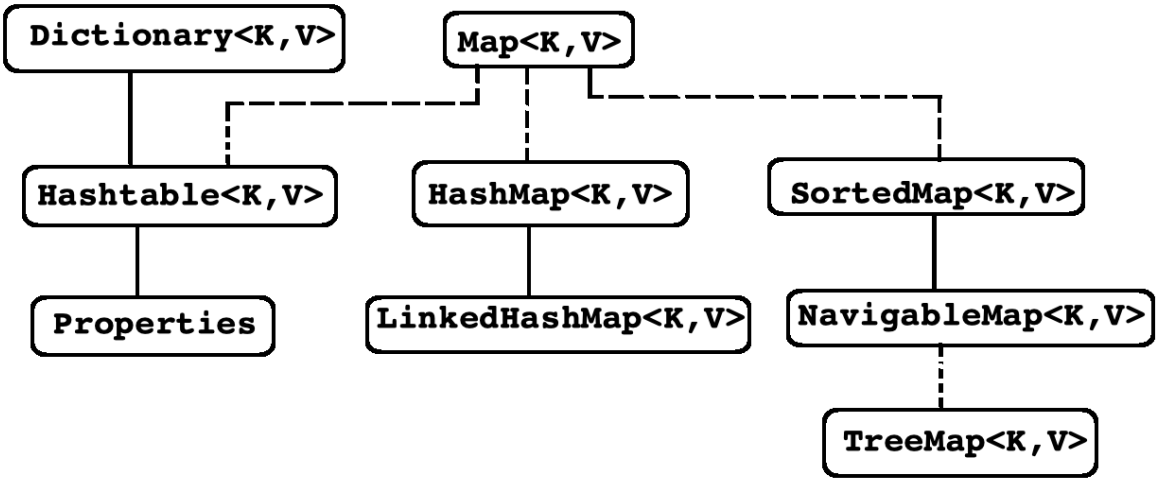
```
public abstract int size();
public abstract boolean isEmpty();
public abstract Enumeration<K> keys();
public abstract Enumeration<V> elements();
public abstract V get(Object key);
public abstract V put(K key, V value);
public abstract V remove(Object key);
```

  - 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.

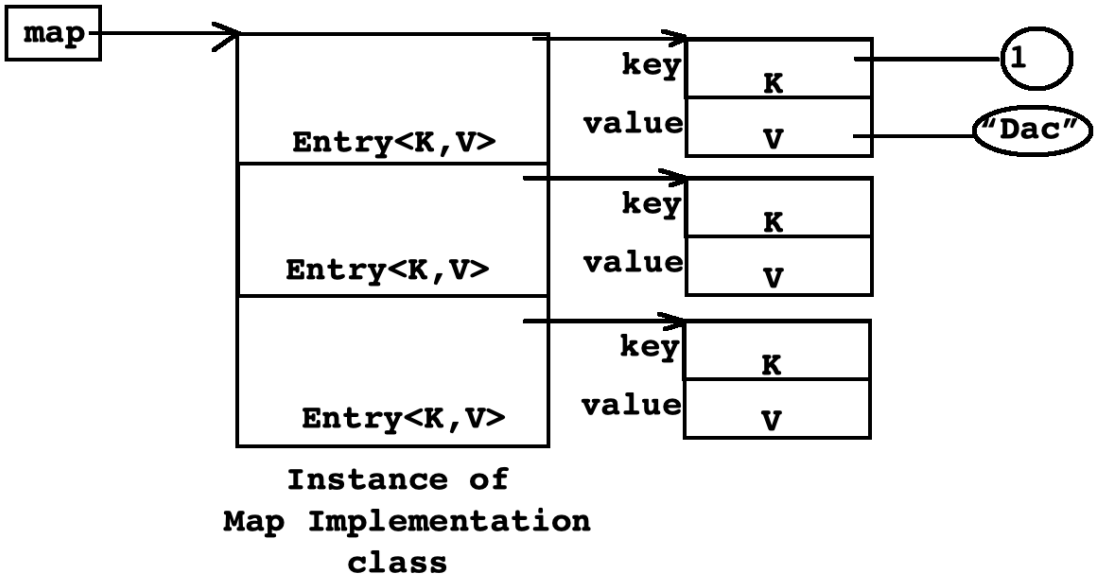
+ Map Interface
  - It is an interface declared in java.util package.
  - It is a member of Java's collection framework but it doesn't extends Collection interface.
  - This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
  - If we want 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.**

day 13

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

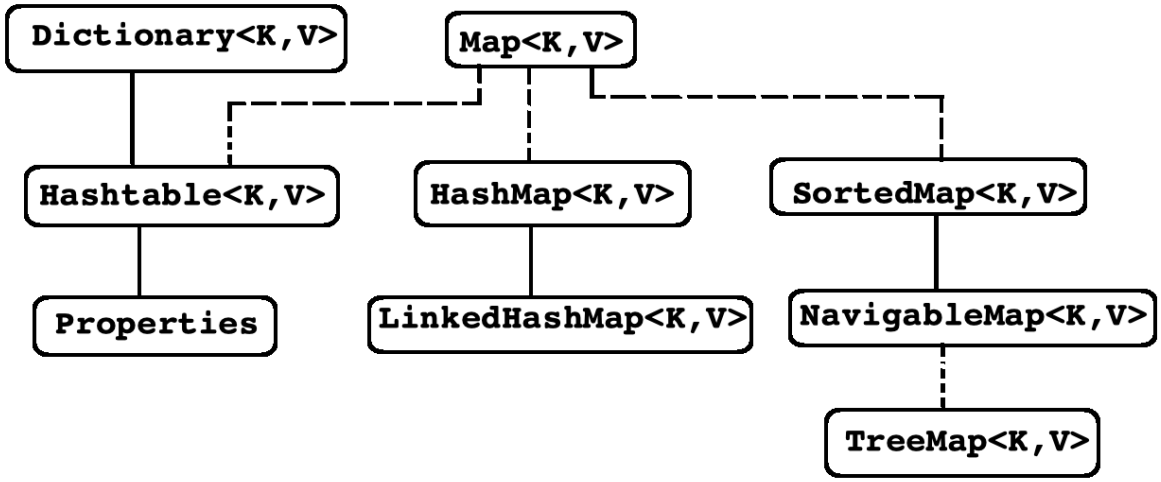
```
public abstract int size();
public abstract boolean isEmpty();
public abstract Enumeration<K> keys();
public abstract Enumeration<V> elements();
public abstract V get(Object key);
public abstract V put(K key, V value);
public abstract V remove(Object key);
```

  - 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.

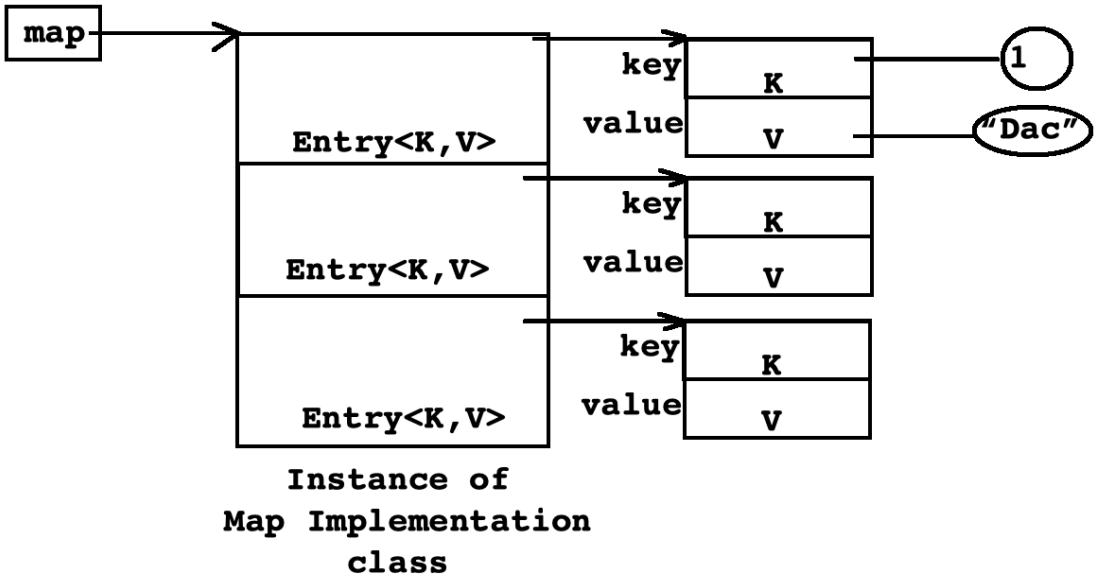
+ Map Interface
  - It is an interface declared in java.util package.
  - It is a member of Java's collection framework but it doesn't extends Collection interface.
  - This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
  - If we want 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.**

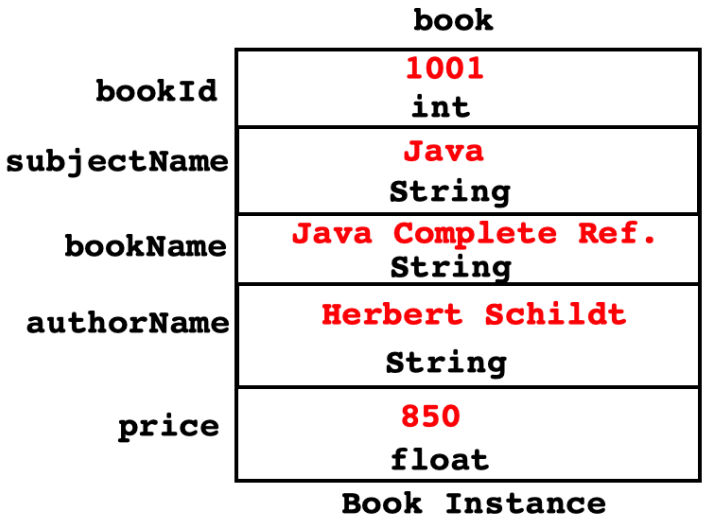
JDBC

jdbc ppt

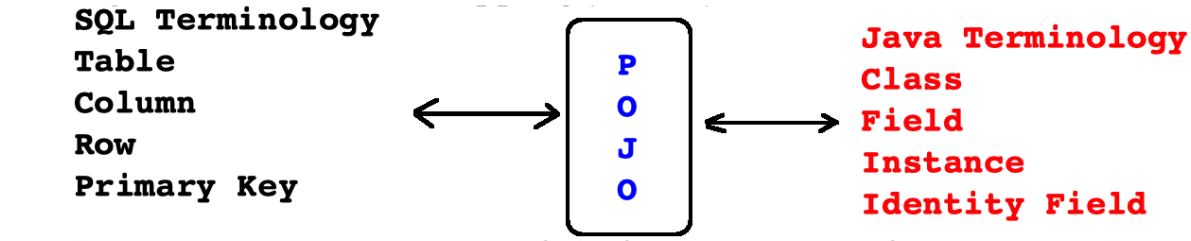
Table : books

book_id	subject_name	book_name	author_name	price
1001	Java	Java Complete Reference	Herbert Schildt	850

columns
Row

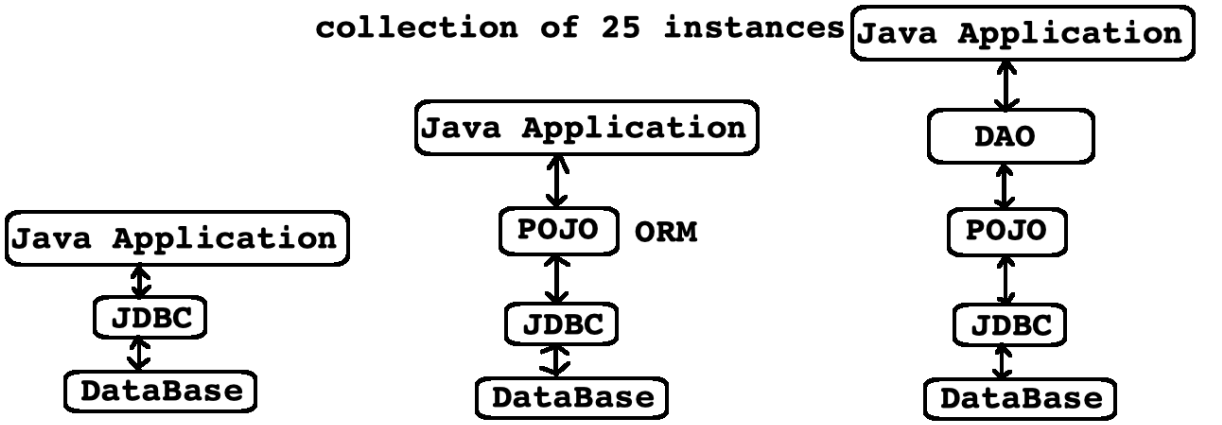


+ Object Relational Mapping(ORM)



- 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

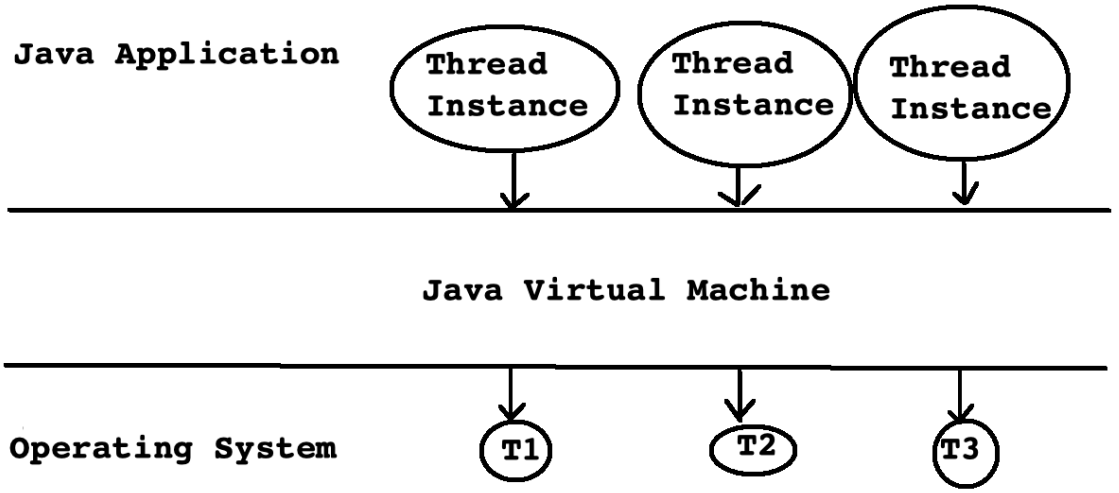
+ ResultSetConcurrency
  1. ResultSet.CONCUR_READ_ONLY
  2. ResultSet.CONCUR_UPDATABLE

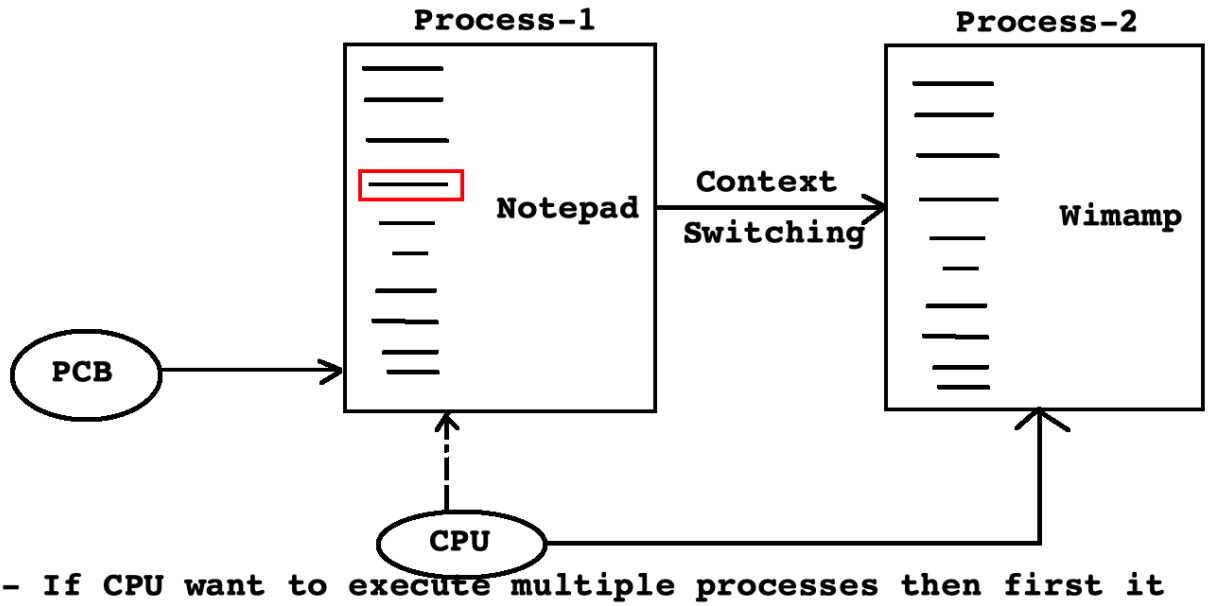
+ ResultSetHoldability
  1. ResultSet.HOLD_CURSORS_OVER_COMMIT
  2. ResultSet.CLOSE_CURSORS_AT_COMMIT

Day 14

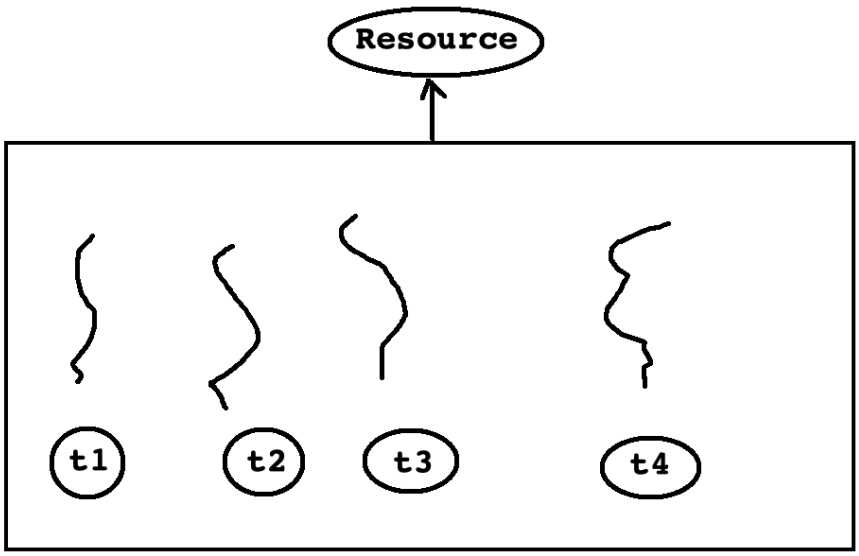
+ Multithreading
  - Process / Task
    1. Program in execution is called process/task
    2. A running instance of a program is called proces.
  - Single-tasking
    * An ability of OS to execute single process at a time is called single-tasking
    * MS-DOS is single-tasking OS
  - Multi-tasking
    * An ability of OS to execute multiple processes at a time is called multi-tasking
    * Modern OS
  - Thread
    1. A lightweight process/sub process is called thread.
    2. Thread is a separate path of execution which runs independently.
  - Single Threaded Application
    * During execution, if application take help of single thread then it is called single threaded application.
  - Multi Threaded Application
    * During execution, if application take help of multiple threads then it is called multi threaded application.

+ Java is multithreaded programming language
  - During execution, JVM take help of main thread and garbage collector hence every java application is multithreaded.
    * Main Thread:
      # It is user thread / non daemon thread
      # It is responsible for invoking main method
      # It's default priority is Thread.NORM_PRIORITY(5).
    * Garbage Collector(GC)/Finalizer
      # It is Daemon thread / Background thread
      # It is responsible for releasing/reclaiming/deallocating memory of unused object.
      # It's default priority is Thread.NORM_PRIORITY + 3; (8).
  - Thread is OS resource. To access OS thread, java developer needn't to do native programming. Rather SUN/ORACLE has given readymade framework to access OS thread hence java application is multithreaded.





- If CPU want to execute multiple processes then first it must save state of current process into process control block(PCB) then control of it can be given to another process. It is called context switching.
- Context switching is a heavy process hence process process based multitasking is called heavy weight multitasking.



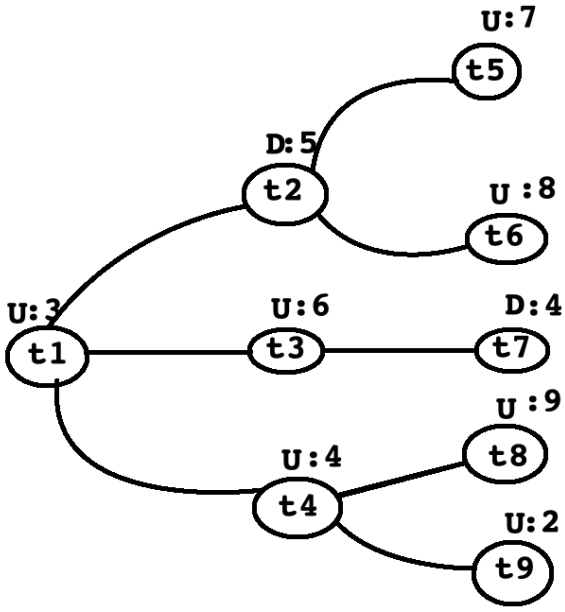
Process

- **Thread always resides within same process hence to access resources assigned to the process threads do not require context switching hence thread based multitasking is faster than process based multitasking.**

+ **Types of Thread**
  1. **User Thread**
  2. **Daemon Thread**
+ **If we create any thread from java application then it is by default considered as user thread.**
+ **“isDamon()” and “setDamon()” are methods of Thread class. Using “setDaemon()” method we can convert user thread into daemon thread.**

```
Thread th = . . .;
if( !th.isDaemon())
 th.setDaemon( true );
```

+ **User thread gets terminated once its job is over. It doesn’t wait for its child thread.**
+ **Daemon thread do not terminate until all its Child thread gets terminated. Hence it is also called as service thread.**



t9
t1
t7
t4
t3
t5
t6
t2
t8

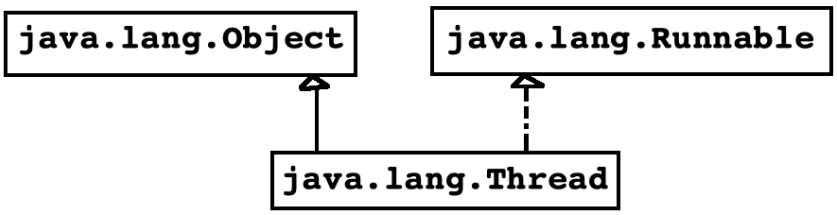
U : User Thread
D : Daemon Thread

+ If we want to manipulate thread in java then we should use types declared in:
  1. java.lang
  2. java.util.concurrent package.

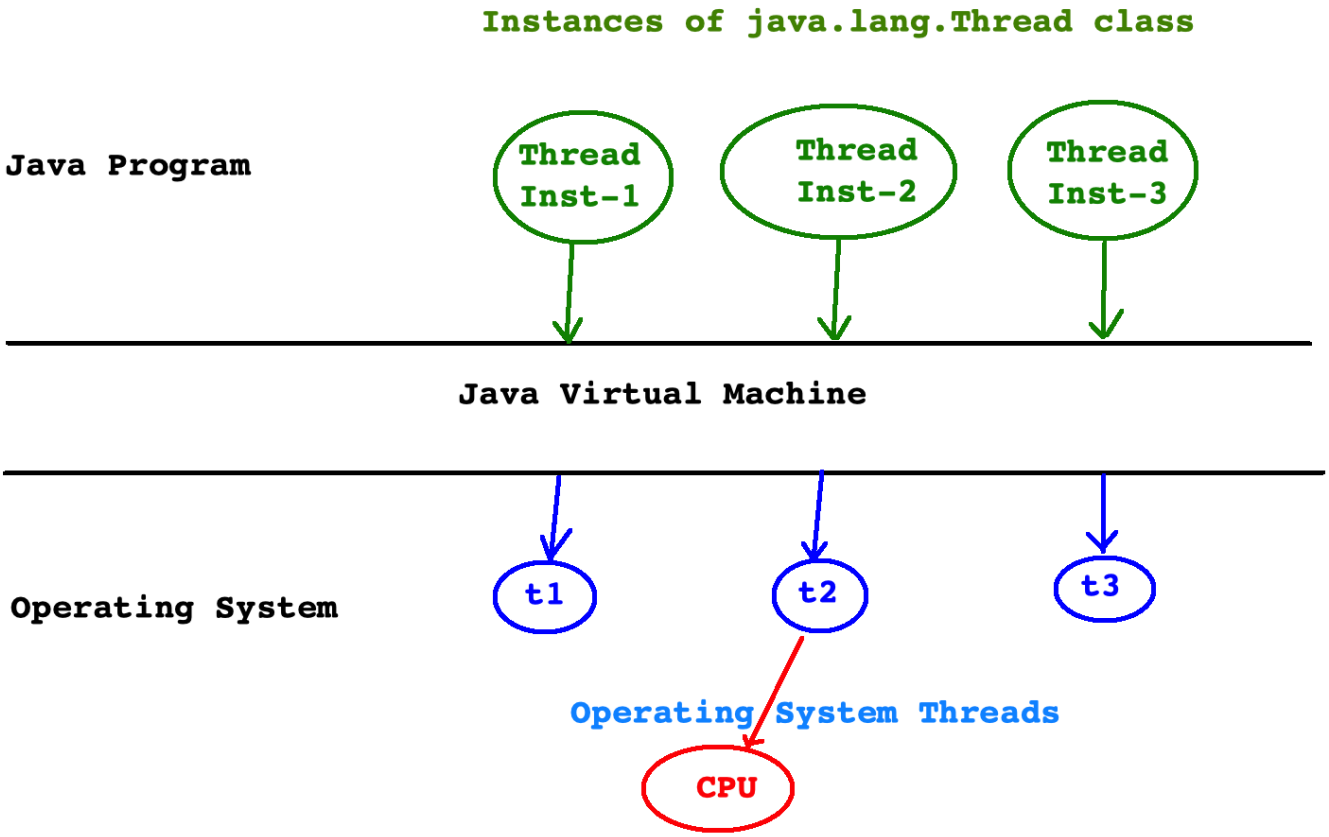
+ Types declared in java.lang package
  - Interface
    1. Runnable
  - Class
    1. Thread
    2. ThreadLocal
    3. ThreadGroup
  - Enum
    1. Thread.State
  - Exception
    1. InterruptedException
    2. IllegallMonitorStateException
    3. IllegalThreadStateException

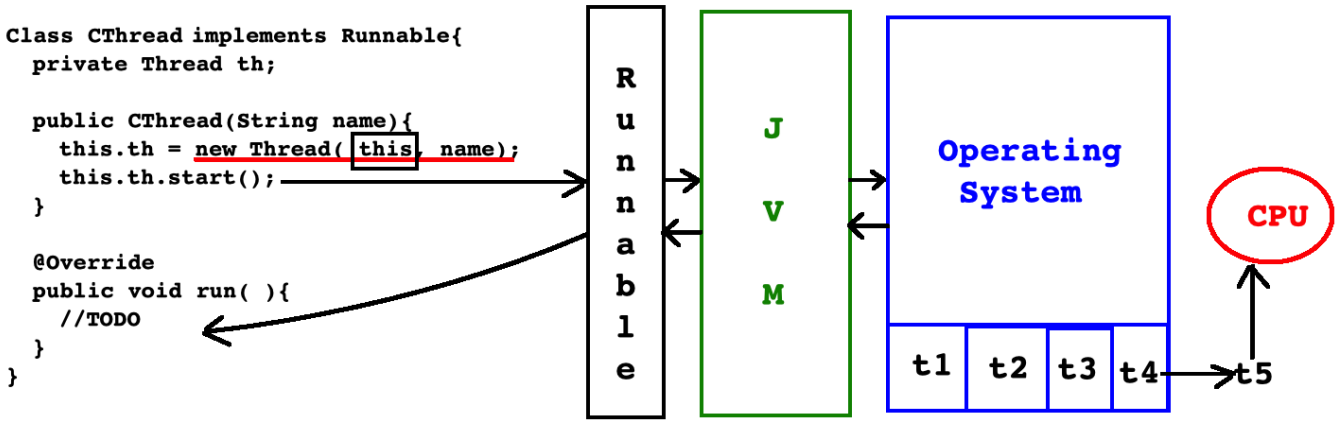
+ Runnable
  - It is a functional interface declared in java.lang package.
  - “void run( )” is a method of Runnable interface.
  - this method is called business logic method in context of thread.
  - If we want to create thread then we can take help of this I?F

+ Thread



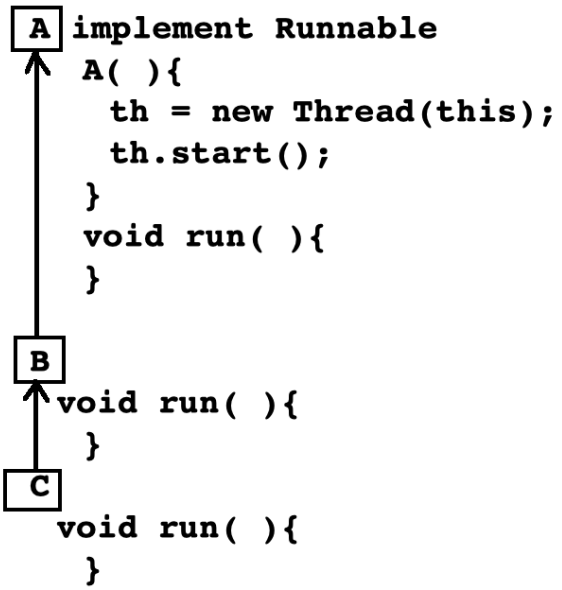
- java.lang.Thread is sub class Object which implements Runnable I/F.
- Instance of Thread class is not a OS thread rather it represents OS thread.
- In other words, It is managed version of unmanned thread.
- If we want to create thread then we can use it.

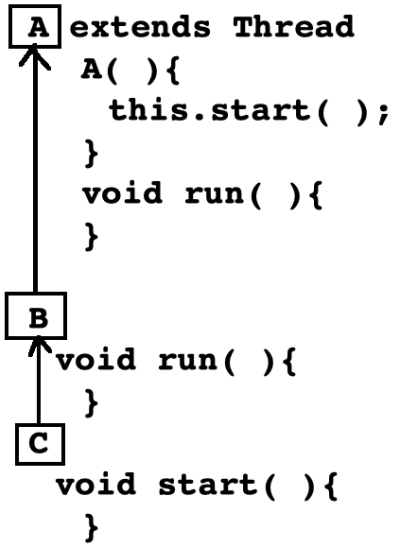




- start() method do not call run() method.
- if we call start() method on Thread instance then it registers thread instance with OS thread via JVM.
- When scheduler assign cpu to the os thread corresponding to thread instance then JVM invoke run method on such a instance which implements Runnable interface.

+ Difference between Runnable and Thread

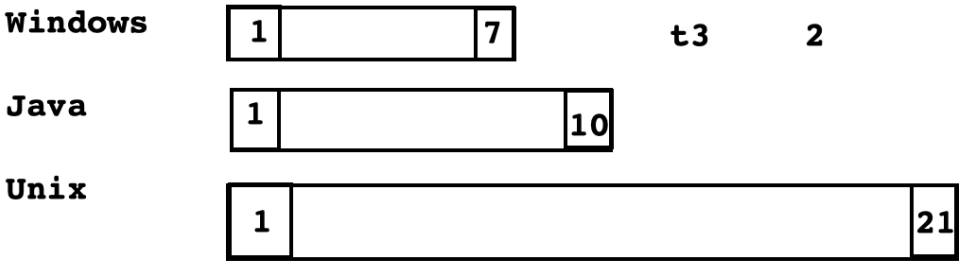


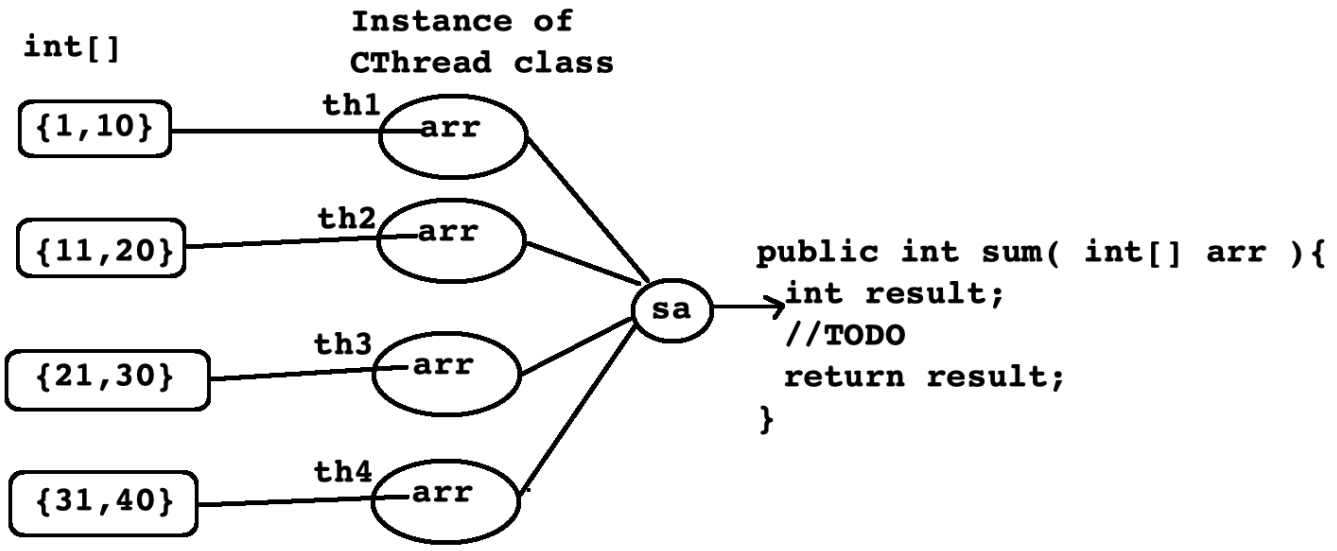


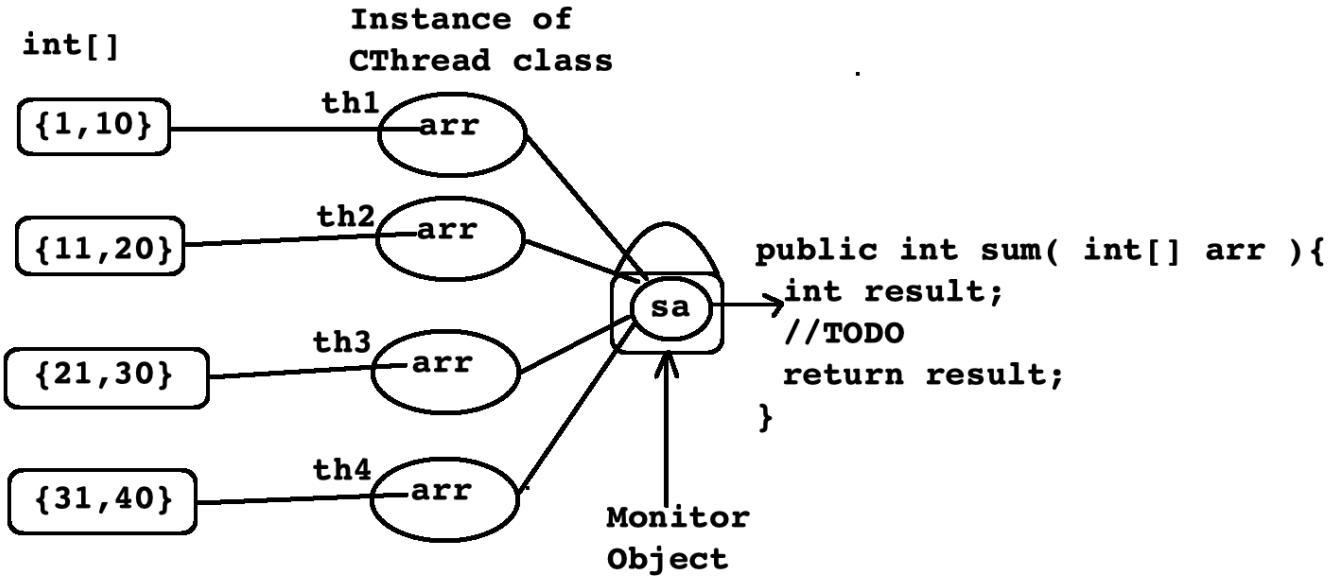
1. If class is extending any other class and if we want to create thread then we should implement Runnable I/F
2. If class is not extending any other class and if we want to create thread then either we should implement Runnable I/F or extend Thread class.

+ If we want to utilise CPU efficiently then we should use thread.
+ Generally, we should avoid use of blocking calls in multithreaded environment.
  1. sleep()
  2. suspend()
  3. join()
  4. wait()
  5. Input operation

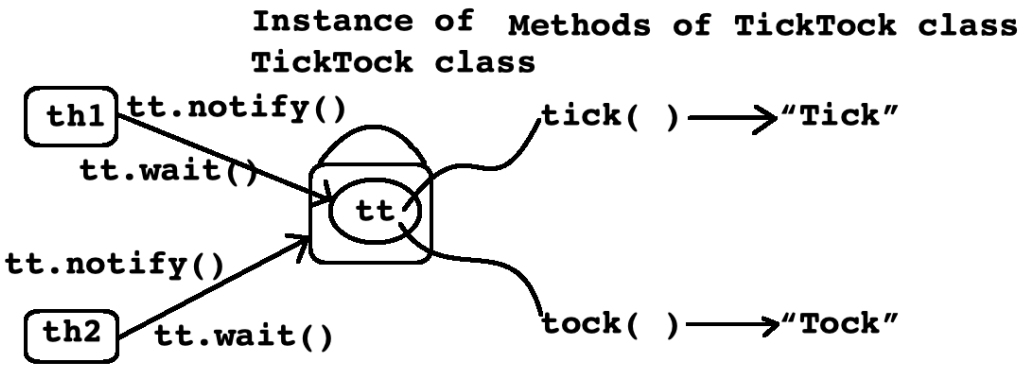
	Java	Windows	Unix
t1	8	7	16
t2	7	7	14
t3	2	4	4







Inter thread communication(Synchronization)



- tt.notify() : send notification to the single waiting thread, which is waiting get access of monitor object(lock) associated with tt.

- tt.wait() : release the lock and move thread into waiting state.

+ Thread Life Cycle