OOP with JAVA by Sandeep Sir

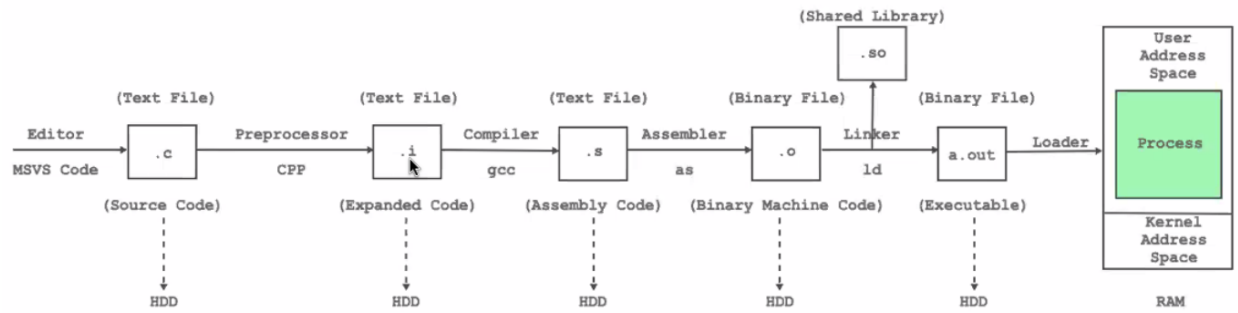
- Flow of execution in C/C++
- Machine Language Code
- Assembly Language Code
- High Level Language Code
- Programming Language
- Framework
- Technology
- Platform
- A short history of Java
- Java Introduction
- Conceptual Diagram of Java
- Java Editions/Development Platforms
- What is API?
- Java Version History
- Software Development Kit
- Java Development Kit
- Java Runtime Environment
- JDK Distributions
- src.zip vs rt.jar vs Java api docs
- "Hello World!!"
- Java Application Execution Flow
- Overview of JVM Architecture

Flow of execution c/c++

main.c → compilation → main.i (Extended code file) → main.asm (Assembly Language) → main.obj (Binary file) [objdump filename] → main.exe (executable binary file)

```c
// File Name: main.c

#include <stdio.h>
int main( void ) {
    printf("Hello, World!\n");
    return 0;
}
```

```
# Preprocessed file
gcc -E main.c -o main.i //main.i is a text file

# Assembly file
gcc -S main.c -o main.s //main.s is a text file

# Object file
gcc -c main.c -o main.o //main.o is a binary file. Use objdump/readelf/nm to view

# Executable
gcc main.c -o main.out //main.out is a binary file. Use objdump/readelf/nm to view
```



read more: https://www.tenouk.com/ModuleW.html

Using Machine Code:

Pros: Direct Execution, Performance

Cons: Complexity, Portability, Maintenance, Development time

Foundation of computing: P K Sinha

Using Assembly Language Code

Pros: Close to Hardware, Performance

Cons: Complexity, Portability, Maintenance, Development Time

Using a High Level Langage Code:

Pros: Easy to use, Faster Dev, Portability, Error Handling

Cons: Performance, Memory Usage, Learning Curve

C can be termed as High Level, Low level or Mid Level Language

Syntax vs Semantics

Unit testing is done by programmer itself (Junit)

Logger adding for unit testing (Apache Log4j)

logger will create a log file in which logs are stored ()

Every language is technology but every technology need not to be a programming language

JRE is a platform

Java language is both technology as well as platform

Statistically type checked

int num = 10 (Specifying at compile time)

Dynamically type checked

object = 10 (Specifying at Run Time)

In strongly type checked, we cannot change datatype once declared

C, C++, Java are strongly type checked languages

Java SE has 2 components: java API, java JVM

In java, java api is nothing but java class. which is readymade or already provided by java.

System API, Scanner API, String, Float API

JAVA LTS Versions - Java 8, 11, 17, 21

cppreferance: refer for c and cpp

JAVA SDK is nothing but java JDK

Java Devlopement Kits is for Devloper

Java Runtime Environment is for clients

https://whichjdk.com/

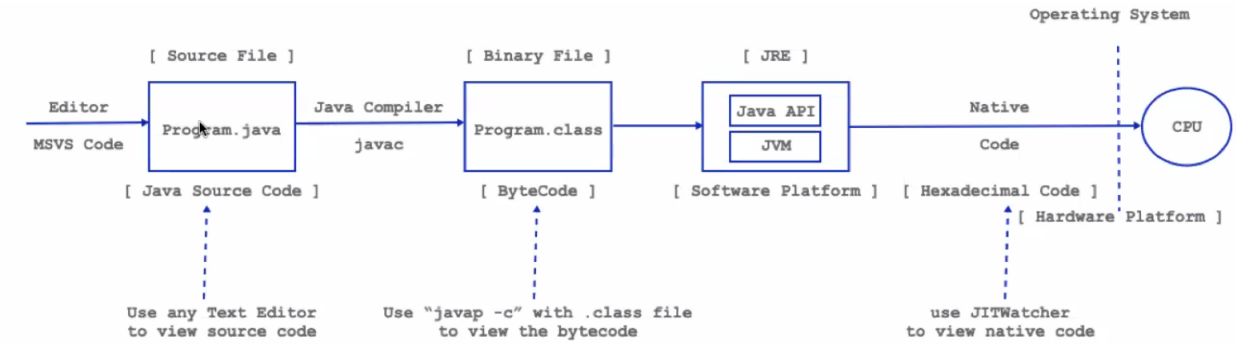
to read bytecode:

javap -c Program.class

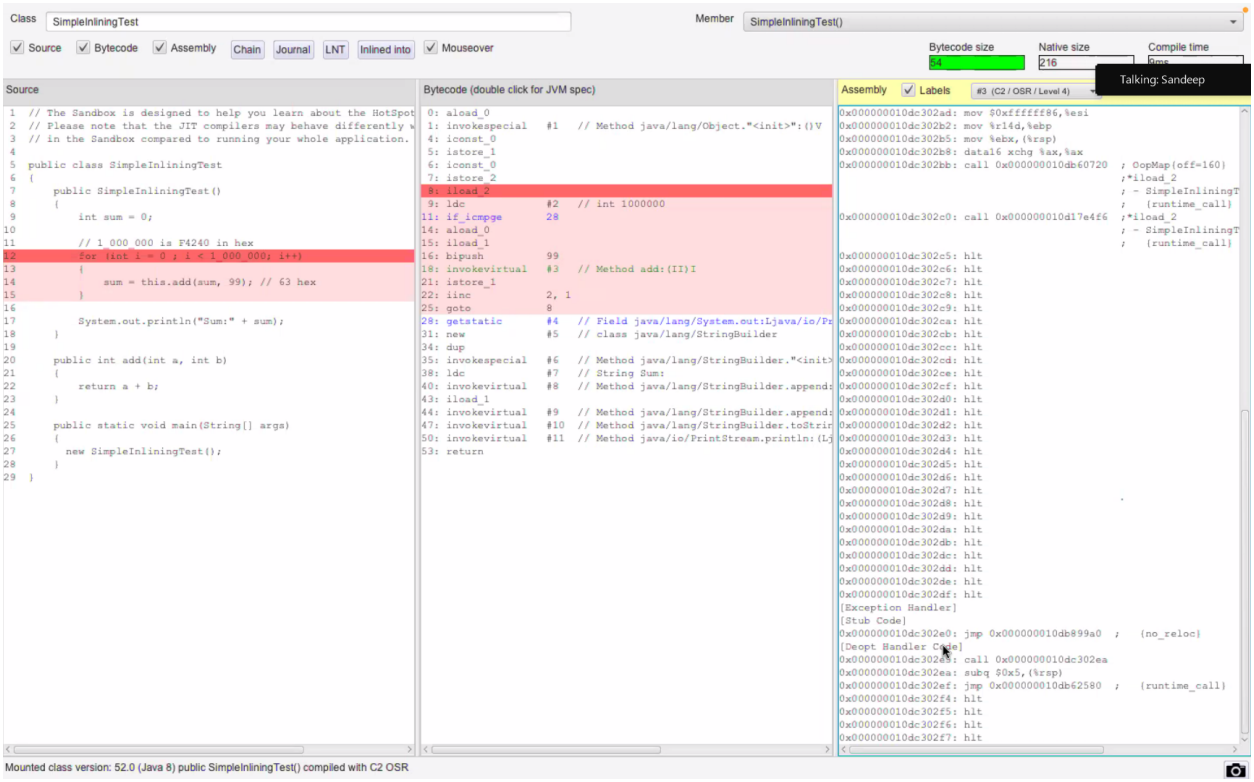
Java Bytecode: It is object oriented assembly code written for JVM

JVM converts Bytecode to native code.

// show native code



JITwatch Tool



- Components of JVM
- Java Buzzwords
- Java Modifier
- Access Modifier
- Java Virtual Machine Threads
- Entry Point Method
- Meaning of System.out.println
- Data Types
- Classification of Data Types
- Primitive Data Types
- Wrapper Class Hierarchy
- Overview of String
- Memory representation
- Widening Conversion
- Narrowing Conversion
- Command line Arguments

Out is static field declared inside System class

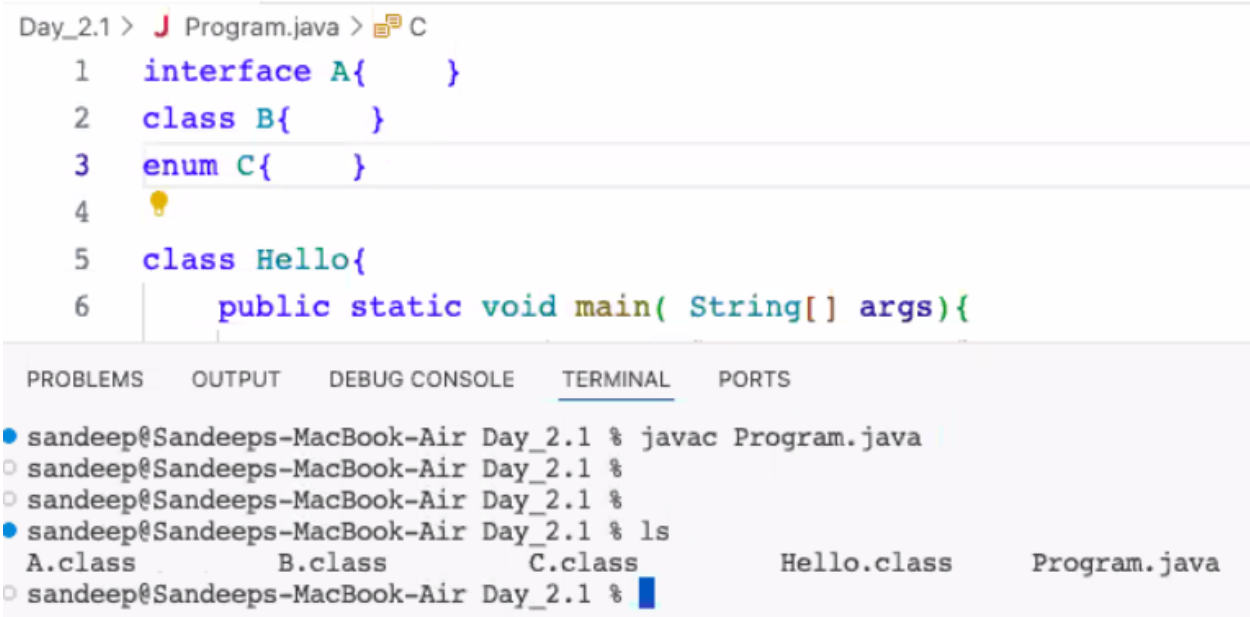
Src file contains source code of java API

compiled SCR files are in JDK>rt.jar

rt.jar: compiled source code of java APIS

Documentation of java APIs: https://docs.oracle.com/javase/8/docs/api/

Workspace → Contains Multiple Projects



Java compiler creates .class file for per class in .java file.

When we create empty file, compiler would not throw error

Good Practice: Same name should be given to the class and file

Bootstrap Path: jre/lib

Extenstion classloader loades jre files from ext directory

(Memory Leakage in JAVA)

Java supports all major and minor pillars of OOP

Bytecode is architecture neutral code that we can ru on any machine by using JVM

Size of datatype on all the platforms is constant, so java is portable

Bytecode is machine independent and portable

Write once run anywhere

JDK is platform Dependent

JRE is platform Dependent

default access modifier in java is package level modifier

java compiler does not check if main method is present or not

java compiler will not check if main method declaration is correct or not

Modifier of the main method must be public

In java, main method must be static

main method must have return type void

name of the main method must be only “main”

main method must take array of string only as a parameter

we can overload main method in java

Widening of data types

C style type casting

Narrowing conversion of data types

Boxing Conversion

```
class Program{
    //public static String toString(int i): Integer class
    public static void main(String[] args) {
        int num1 = 10;
        //String strNumber = Integer.toString( num1 );//Boxing
        String strNumber = String.valueOf(num1);     //Boxing
        System.out.println( strNumber );
    }
}
```

Unboxing Conversion

```
class Program{
    //public static int parseInt(String s)throws NumberFormatException: Integer class
    public static void main(String[] args) {
        String str = new String(original:"123");
        int num1 = Integer.parseInt( str );//UnBoxing
        System.out.println( "Num1    :    "+num1);
    }
}
```

if string do not contain parseable numeric value then parse method throws number format exception

Command Line Arguments

//Java block comments and documentation comment

//Wrapper Class

//Data types

//Multithreads: GC thread

//Print, Printf and Println

// topic+java+oracle

Doubts:

1. Memory allocation in Heap and stack for primitive and non primitive
2. Java virtual machine threads
3. Locations of the classes that we're defining

---

java.lang is by default imported in any java file

Can import classes in 2 ways

1.

```
package org.example;

public class Program {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner( System.in );
    }
}
```

2.

```
package org.example;
import java.util.Scanner;
public class Program {
    public static void main(String[] args) {
        //java.util.Scanner sc = new java.util.Scanner( System.in );
        Scanner sc = new Scanner( System.in);
    }
}
```

```java
package org.example;

import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        //java.util.Scanner sc = new java.util.Scanner( System.in );
        Scanner sc = new Scanner( System.in );

        System.out.print("Name   :    ");
        String name = sc.nextLine();
        System.out.print("Empid :    ");
        int empid = sc.nextInt();
        System.out.print("Salary     :    ");
        float salary = sc.nextFloat();

        System.out.println(name+"    "+empid+"    "+salary);

        sc.close();
    }
}
```

Date class in java.util

Calender class in java

LocalDate class in java

Objected Oriented Programming in Java

Methods in java:

```
public class Demo{
    protected static void add(int a,int b){
            return a+b;
    }
```

```
    public static void main(String[] args)
    {
        System.out.print("The maximum number is: " + Demo.add
(9,7));
    }
}

//Protected: Access Specifier
//Void: Return Type
//add: Method name
//a,b: arguments
```

Procedure Oriented Programming: When we do programming by creating methods to avoid repetitions. eg. c.

Class:

→ If we want to group related data elements together, we should define class

eg. name, empid, salary: Employee

→ We can define class inside method (Local class), class (Nested Class) or outside of class (Top Level Class) also.

→ In java, inside class we can define fields, blocks (initializer block or static initializer block), Constructor, Methods, Nested Types (Interface, class, enum).

→ variables declared in class are called as field

→ non static fields are known as instance variable

→ When we create instance of class, continuous memory is allocated to non static blocks.

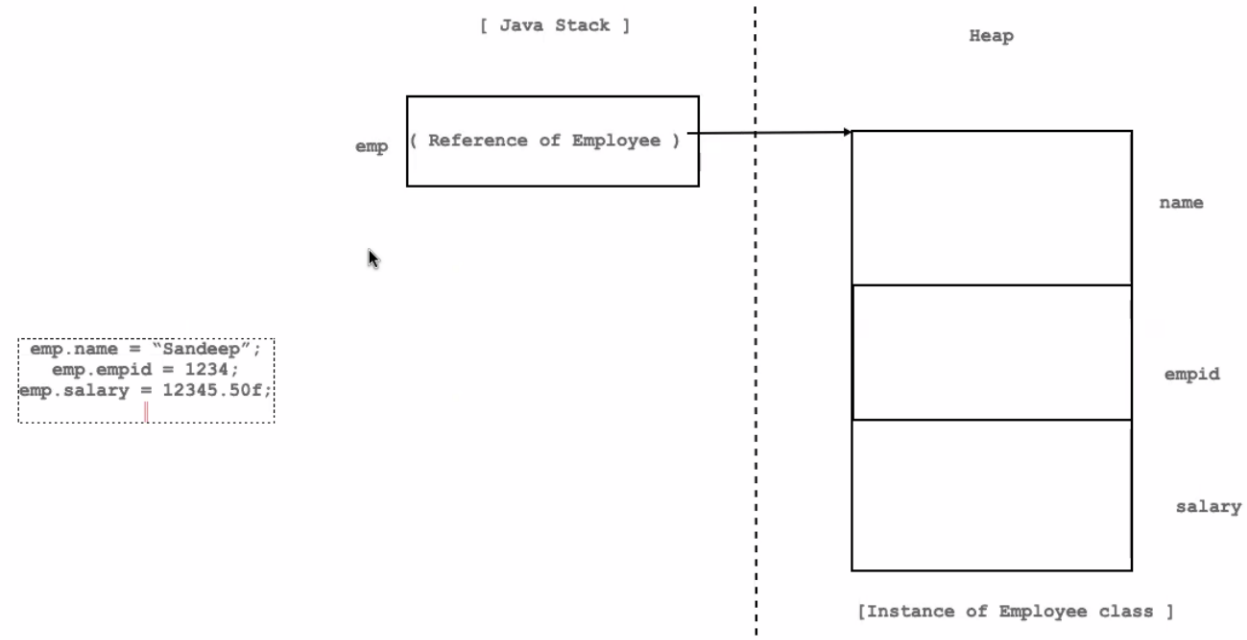
→ class from which we can create instance is called concrete class.

→ Non static block allocates memory once per user in heap

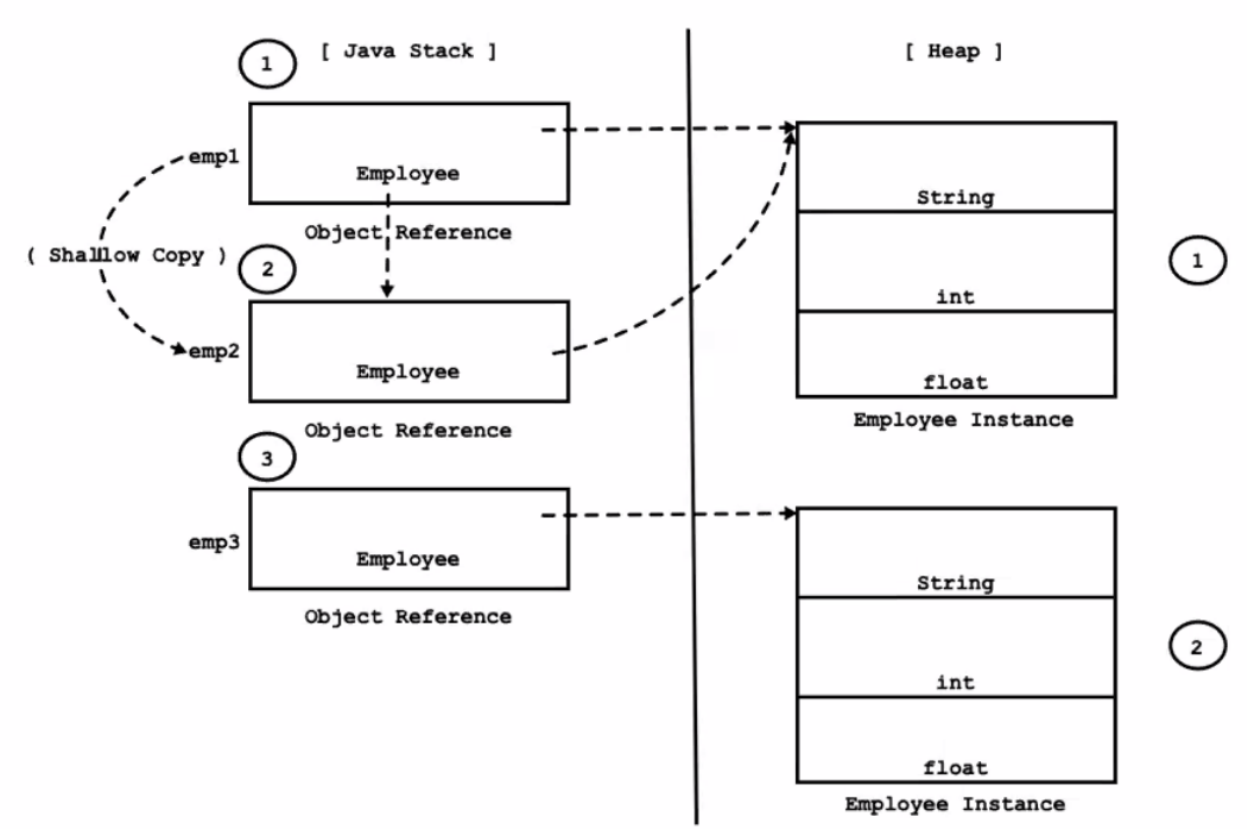
→ new operator is used to allocate memory in heap for instance.

→ Reference of instance is initialized to point the specific instance of the class

→ Reference variable is method local variable which is stored in stack and points towards the object inside heap memory




```
Employee emp1 = new Employee();
Employee emp2 = emp1;
Employee emp3 = new Employee();
```



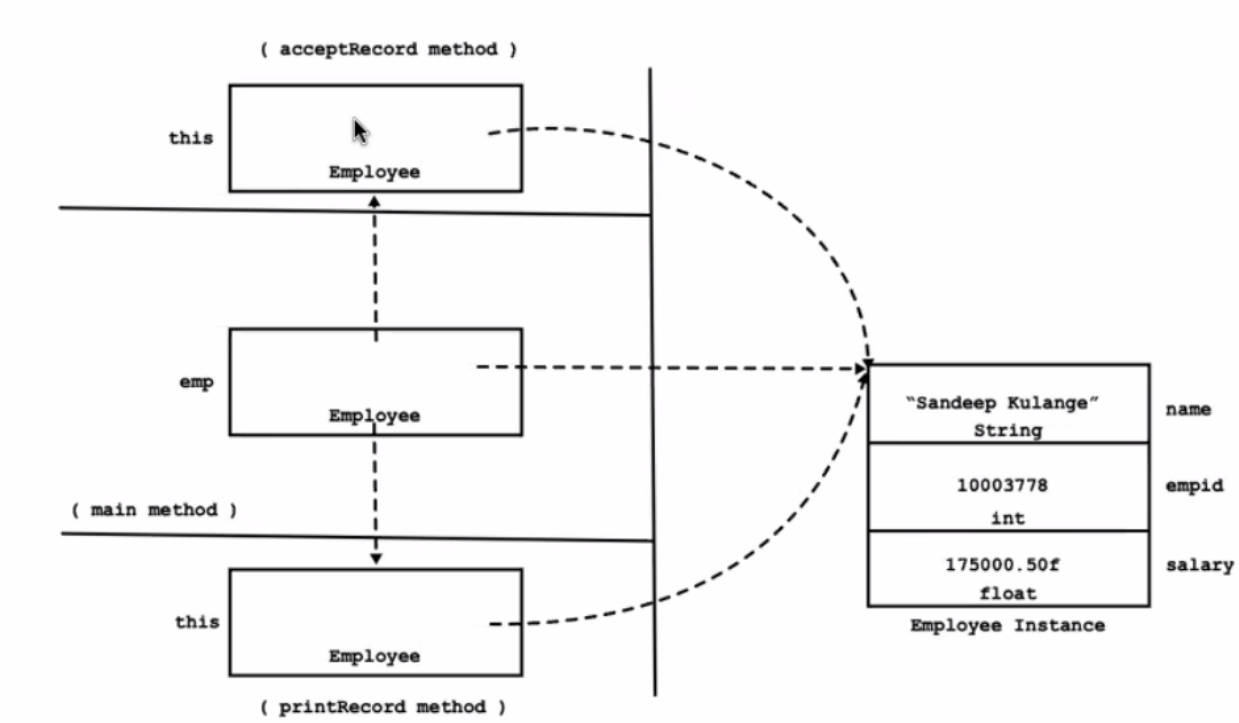
→ Process of calling a method on instance is called as message passing

```
emp.acceptRecord(data);    //Message Parsing
emp.printReccord(data);    //Message Passing
```

→ In java, instance reference (this keyword) is passed by the java compiler automatically.

→ this keyword is connection or link between non static instance variables and methods.

→ this is method parameter which stores reference to current instance.



→ non static fields gets space inside the instance according to order of their declaration in class

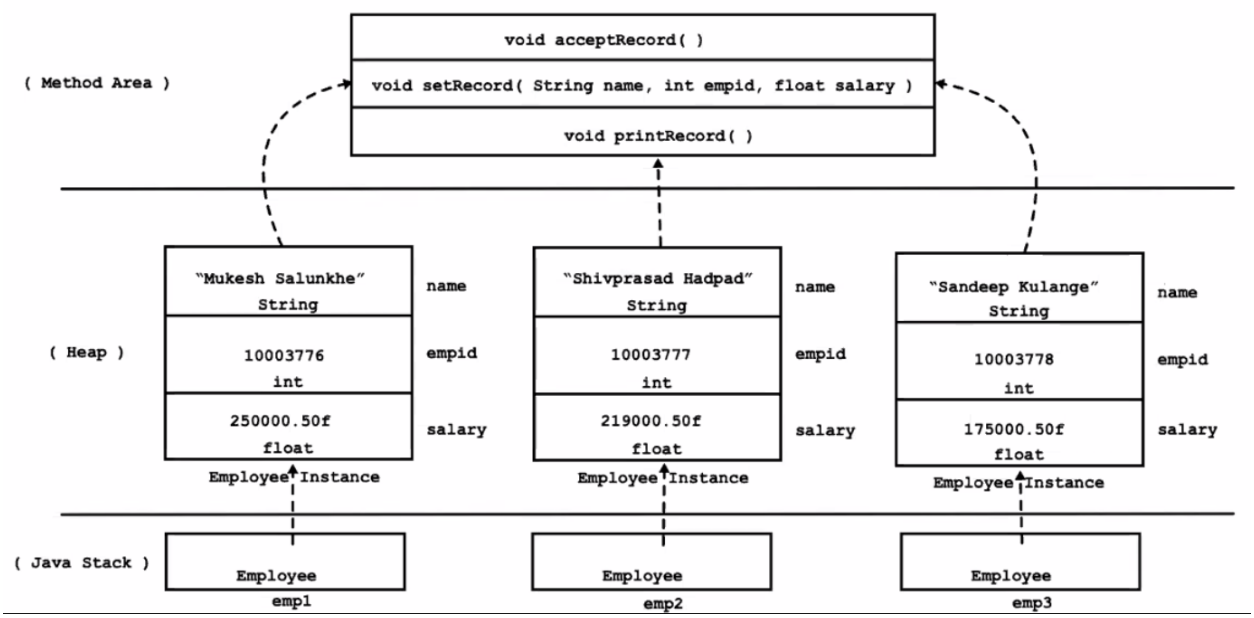
→ In java, object is called as instance

→ Process of creating instance from class is instantiation

→ Sharing of class methods is done by instances by passing reference to it

→ Method do not get space within instances as it is stored once per class and shard through all instances

→ State is nothing but value stored in variable



→ Characteristics of an instance: An instance has state, behavior, and identity

→ Structure of instance depends upon fields declared in class

→ method represents behavior of the instance

→ Class is template/model/blueprint for creating instances

→ Class does not occupy memory, instance does.

→ null is a literal/constant in Java designed to initialize reference variables.

```
Date date = null;    //date is a null object
```

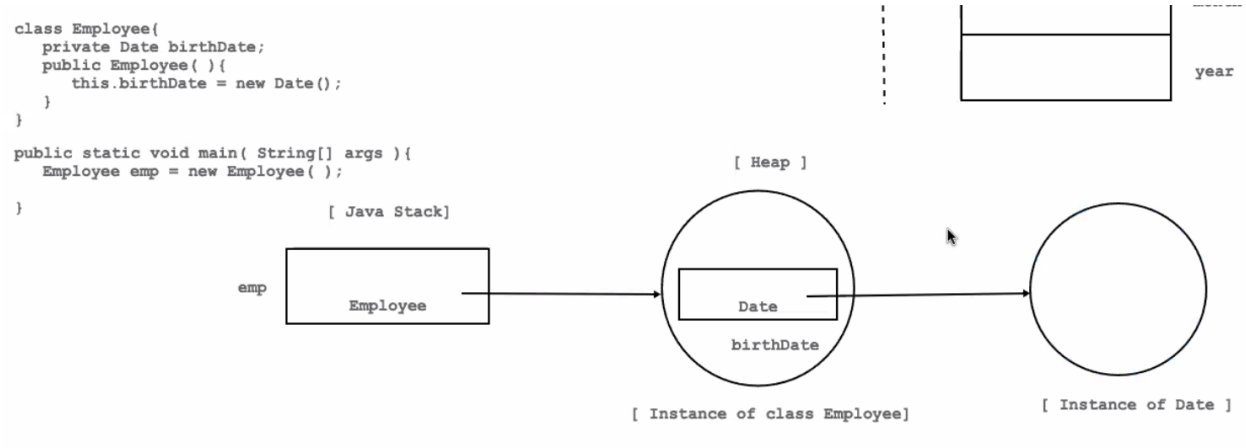
NullPointerException

//Stack trace in java

//what happens when we try to call null object with static methods or variables

→ Primitive Datatype is value type and non primitive is reference type

→ Value type has default types as 0 and Reference Types has default value of null



→ If reference is local it gets stored in stack and if instance is field then it gets stored in heap

→

→ If methods are performing same logic, then their name must be same

for method overloading:

→ For methods with parameter with same type, number of parameters must be different

→ If methods having same number of parameters, at least one parameter must be different

→ if number of parameters and type of parameter is same, order of the parameters must be different

→ only on the basis of different return type, you cannot define method with same name

→ Catching value from method is nor cumpulsary.

→ methods which are participating in overloading are callled overloaded methods

//Date Problem

→ when we want to call method only once per instance, we define constructor

→ Constructor is a java syntax which is used to initialize the instance

→ we cannot call constructor on instance explicitly

→ constructor cannot be called for class

→ we can use any access modifier for constructor

→ Types of Construvtor:

1. Parameter-less Constructor (User Defined Default Constructor)
2. Parameterised Constructor
3. Default Constructor: if we do not define any constructor inside class, compiler generates a constructor for class which is known as default constructor

Process of defining multiple constructors inside class is knows as constructor overloading

// can we make constructor private

```java
package org.example.demo3;

class Date{
    private int day;
    private int month;
    private int year;

    private Date( ) {
        System.out.println("Inside constructor");
        this.day = 0;
        this.month = 0;
        this.year = 0;
    }


    public void printRecord( ) {
        System.out.println( this.day+" / "+this.month+" / "+this.year
        System.out.println();
    }

    public static void test( ) {
        Date dt2 = new Date();
    }
}

public class Program {
    public static void main(String[] args) {
        //Date dt1 = new Date( );
        Date.test();
    }
}
```

This Statement: Using another constructor within one constructor

using this statement, we can reuse body of one constructor

The phenomenon of calling another constructor from one constructor is constructor chaining.

this statement must be the first in constructor body.

Procedural : Object oriented prog,

JVM

1. Classloader
    a. Bootstrap: The bootstrap class loader loads the rt.jar file and other core libraries
    b. In Java, the extension class loader **loads extensions for core Java classes so that they are available to all programs running on the platform**. It's a child of the bootstrap class loader and loads classes from the $JAVA_HOME/lib/ext directory
    c.

//System.out, System.in

//Java compiler do not call main method. With the help of main thread JVM invoke main
method in Java.

// camel case aming convention, pascal case naming convention

https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

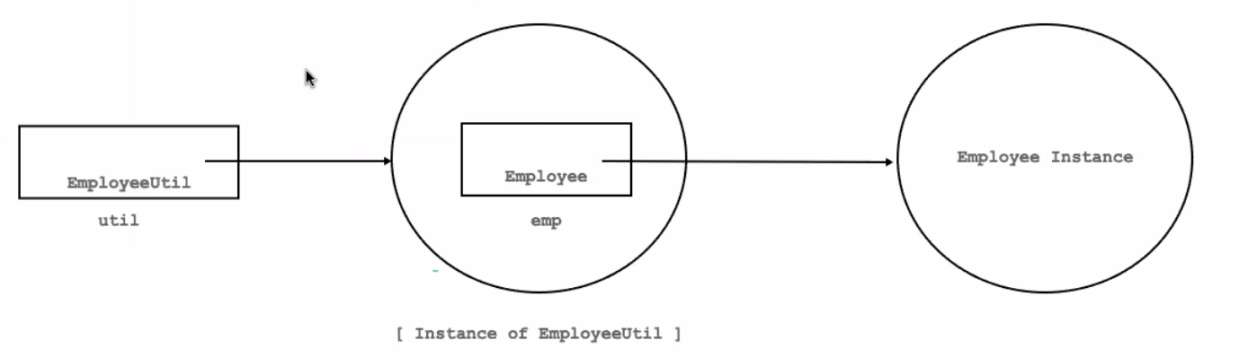
→ we can pass scanner as parameter to the class

→ Setter methods are used to update the values of private fields in class (Mutator/setter method)

→ Getters methods are used to get/return values of the fields in class (Inspector/selector/getter method)

→ After creating instance if we want to do some changes in fields of class, we use getter and setter methods.

→ utility class is defined to include some uilities like enter values



Association

→ Instance field initializer: gets executed before constructor block

if we value of a filed is same for every constructor, instance field initializer is used in that case

→ Instance initializer block: if a block of program is same for every constructor, it is initialized in instance initializer block

```
class Test{
    private int num1 = 10;  //Instance Field Initializer
    private int num2;
    private int num3;

    {//Instance Initializer block
        System.out.println("Instance Initializer block");
        try {
            String pathName = "Sample.txt";
            File file = new File(pathName );
            Scanner sc = new Scanner( file );
            this.num2 = sc.nextInt();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
```

→ By using final keyword to reference of object, we can make changes in instance fields but cannot make change in instance

casing in java

→ Pascal case: first name of each word must be capital (eg. ClassName)

for class, enum, interface

→ Camel case: first word small and next words must start with capital (eg. getValue, parseInt)

method local variable, method parameter, for field, for method

→ snake case: In this case, word is separated by underscores

constant and final variables must be written in capital snake case (eg. MAX_INT)

In java, every class is directly or indirectly is extended from Object class.

Object class is part of object.lang package

→ Java.lang.object is concrete final class, so we can create it's instances directly

→ Object is root of java class hierarchy

→ No nested type in object class

→ No fields available in object class

→ It has only default constructor

→ There are 11 methods in Object class (6 final & 5 non final)

() methods are native)

//Visual VM to analyze JVM

→ Instance variable gets space for once per instance

→ If you want share value of a field in all the instances of the class then we define it as static

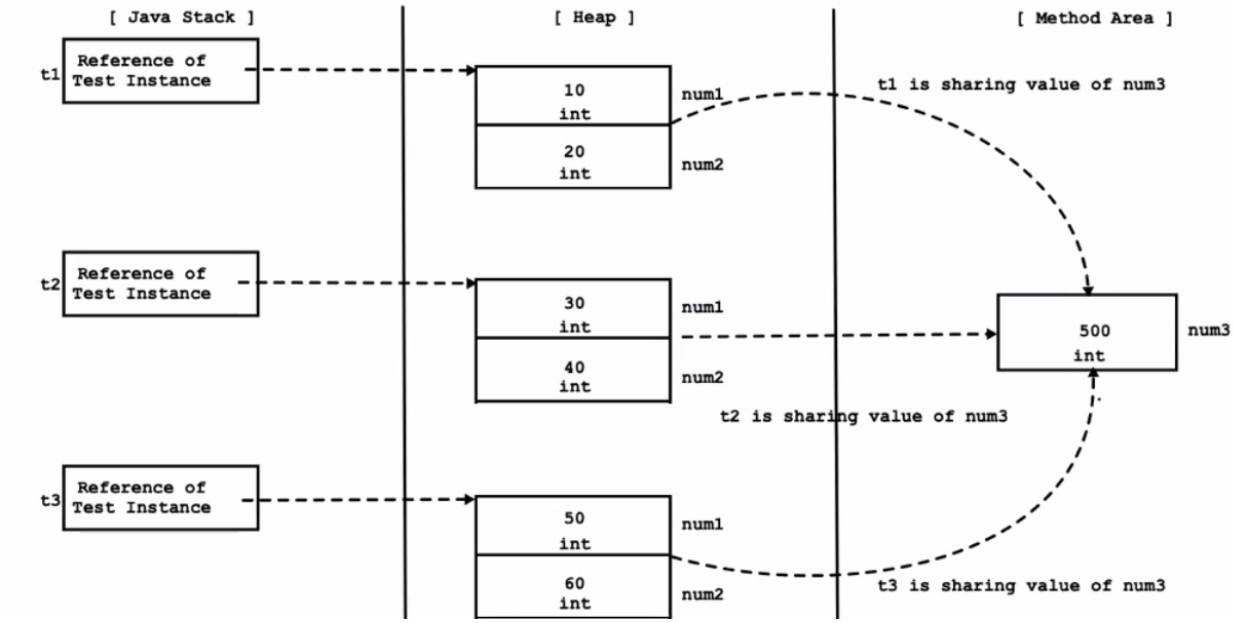
→ static variables are also called class level variables.

→ Static variables gets memory while class loading in method area

→ where, non static variable gets space after creating instances inside heap

→ to access static variables inside methods, we should use Classname.static_field

→



→ We can use static field inside constructor but as a good practice, static field must be initialized outside constructor

→ When multiple values are involved while initializing static field, static initializer block is used

→ In java, when method inside class does note require any instance to run, the method is defined as static .

→ Inside static, we cannot access non static members

→ Static methods can only access static members

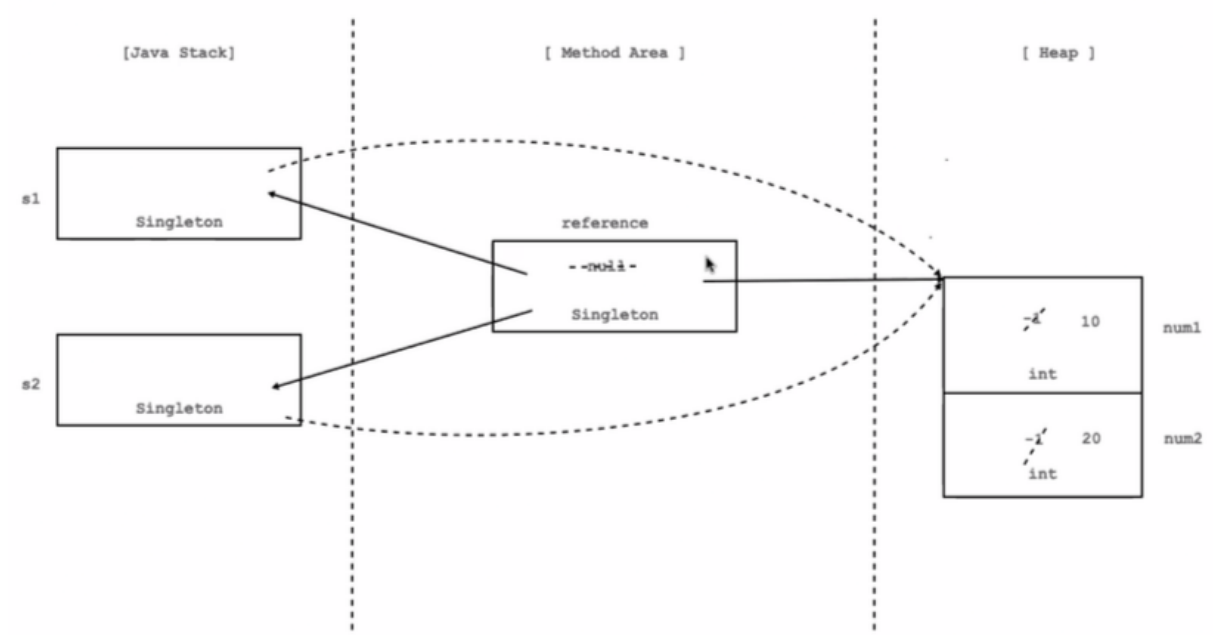
→ inside non static method, we can access non static and static members directly

→ inside static method, if we want to access non static method directly, we need to create instance inside it

→ all the members of math class are static. (that's why it's constructor is private)

→ Singleton Class - A class from which we can create only one class is singleton class

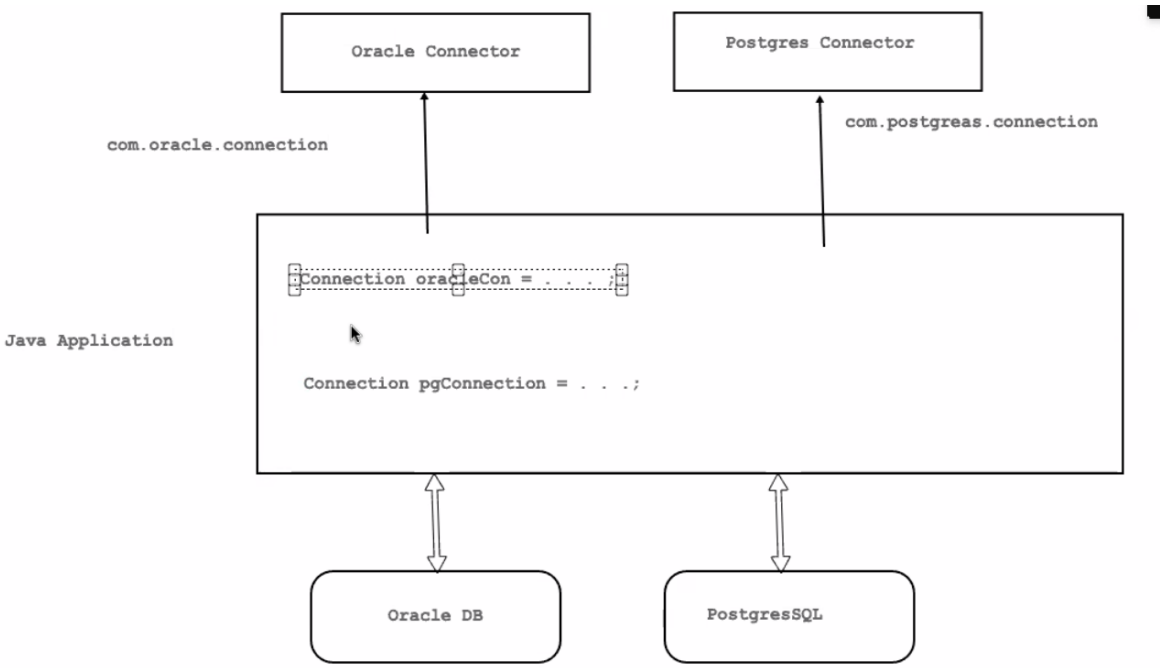
→ In java you cannot declare local variables static



→ Classpath is enviroment variable of java which is used t locate .class file.

→ JDBC Driver:

→ The package in java is used to avoid class name ambiguities/ name collision



→ Package combines functionally related classes.

→ Packages also minimizes learning curve

→ if a class is not member of any package then it is known as unpackaged class

→ Package declaration statement must be the first statement

→ One class cannot be a part of 2 packages

→ After compilation, package name is physically mapped with a directory

→ We can define only one public class in one .java file

→ if we want to use packaged class in unpackaged class, we must import packages.

→ we cannot use classes or types from unpackaged to packaged class.

→ when we do not assign any package to class, it is then set to default class

→ to run the packaged class ⇒ java p1.program

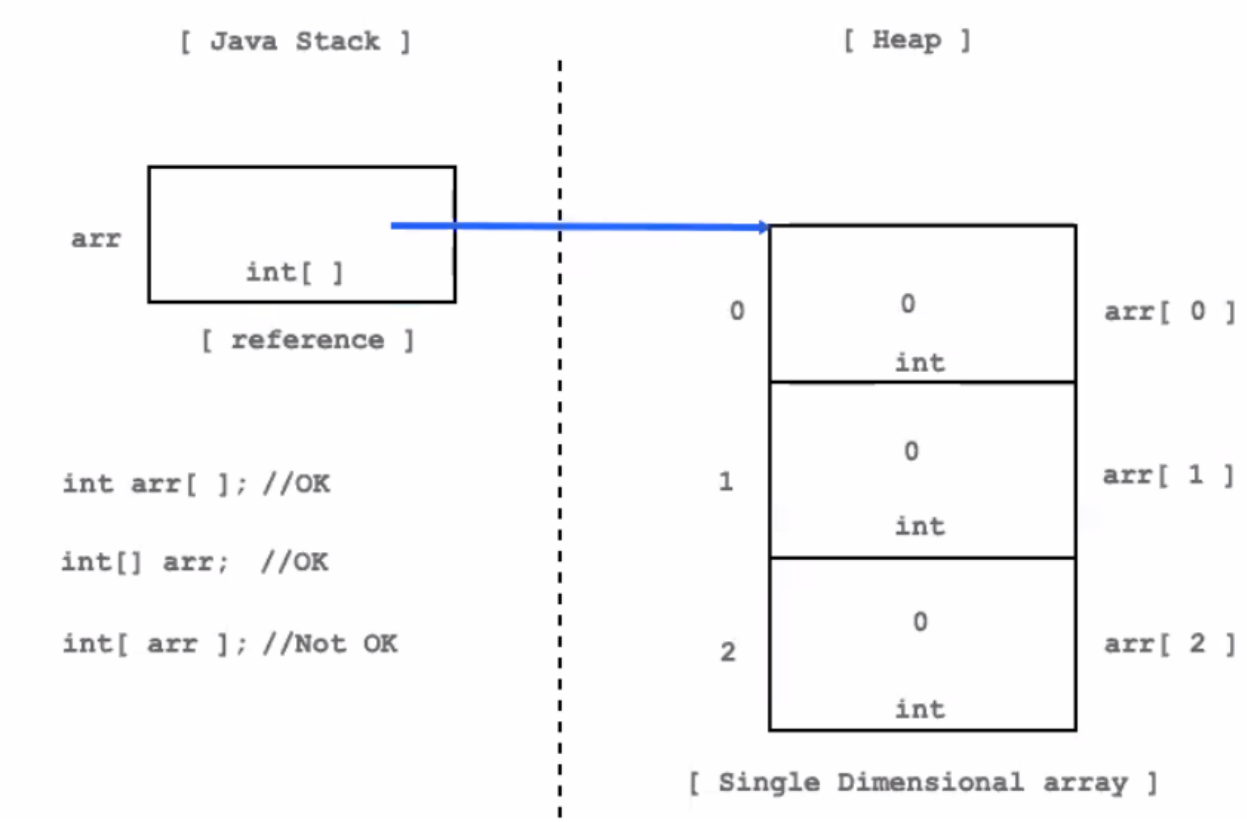
→ Static import is used to access static member of other class inside class.

→ array is a linear data structure that groups elements of same type

→ In java, array is reference type (Non primitive type)

→ Array Types:

1. Single Dimensional
2. Multi Dimensional
3. Ragged Array



→ if we try to create array with -ve size, Exception: Java.Lang.NegativeArraySizeException

→ if we pass invalid index, Exception: Java.LangArrayIndexOutOfBounds

→ Different type of object stored, Exception: Java.Lang.ArrayStoreException

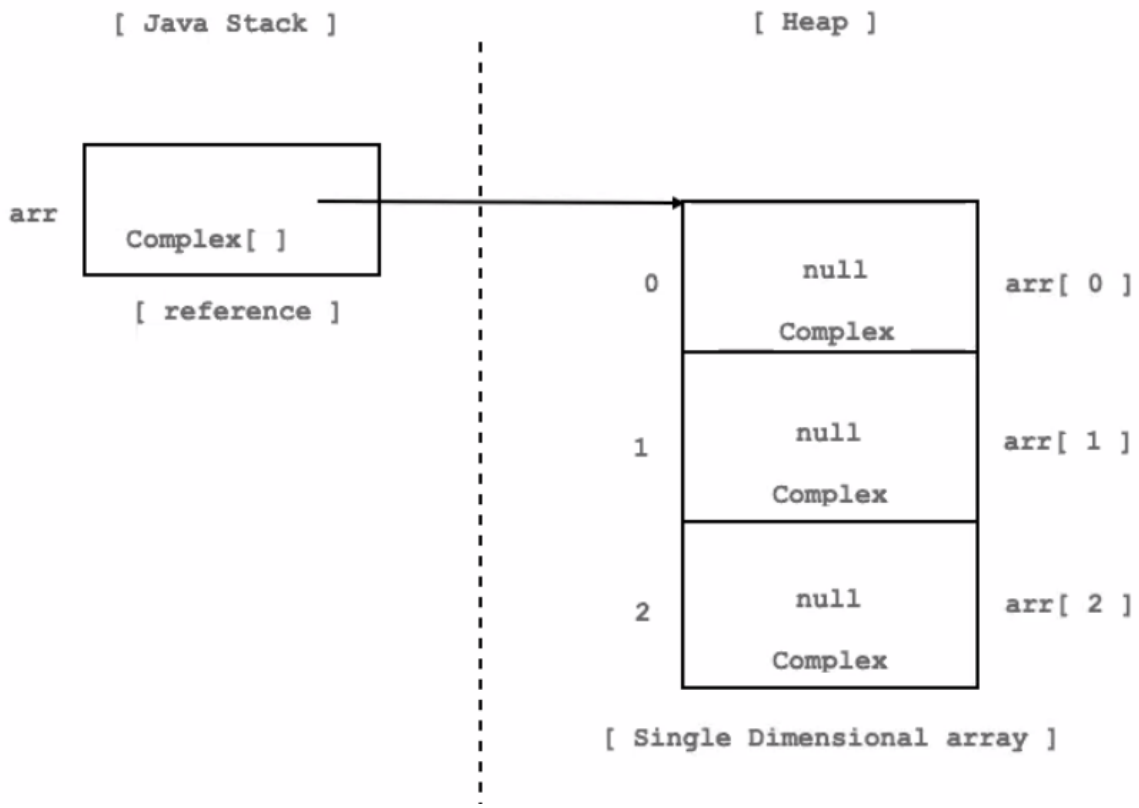
```
int[] arr = new int[size];
int arr[] = new int[size];
int[] arr = new int[] {1,2,3};
```

```
for(int ele : arr){           //Iterator in java
    System.out.println(ele);
}
```

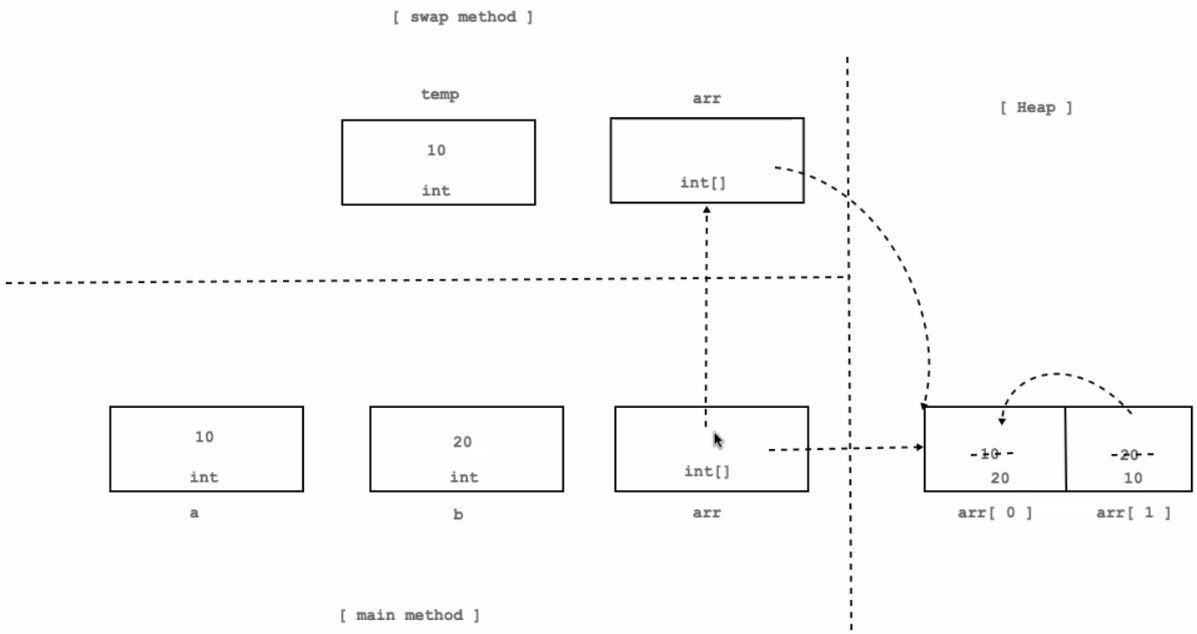
→ Array as field of class

→ array of non primitive datatype

```
complex[] arr = new complex[3];
//array of references of instances
```



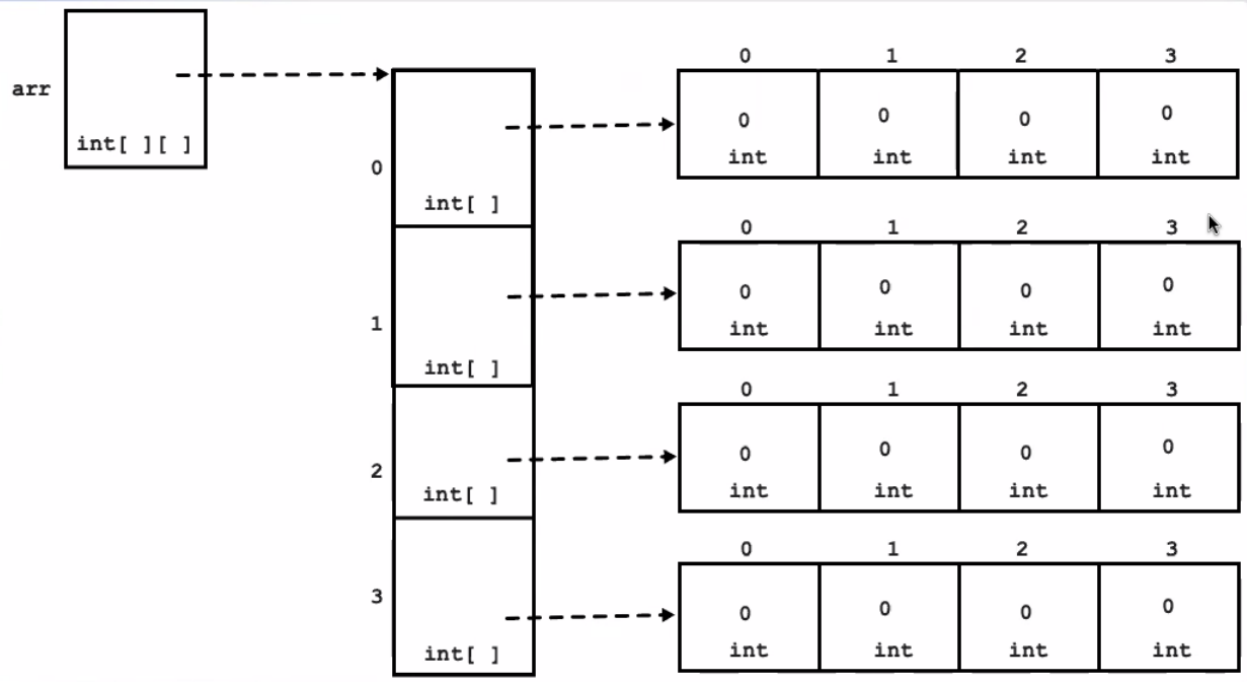
→ Swap method using array as argument



→ Variable argument method:

void sum (int... arr) //variable argument method or variable arity method

→ Multidimentional array is array of arrays whose element arrays are of same length




```
int arr[][] = null;
int[] arr[] = null;
int[][] arr = null;
int[][] arr = new int[][] {{10,20,30}, {40,50,60}};
int[][] arr = new int[4][4];

for (int row = 0; row<arr.length; row++){
    //System.out.println(Arrays.toString(arr[row]));
    for(int col = 0; col<row[row].length; col++){
            System.out.print(arr[row][column]);
     }
    }

for( int[] row: arr) {
        for( int col: row){
            System.out.println(col);
        }
    }
```

//multidimentional array as a field of class

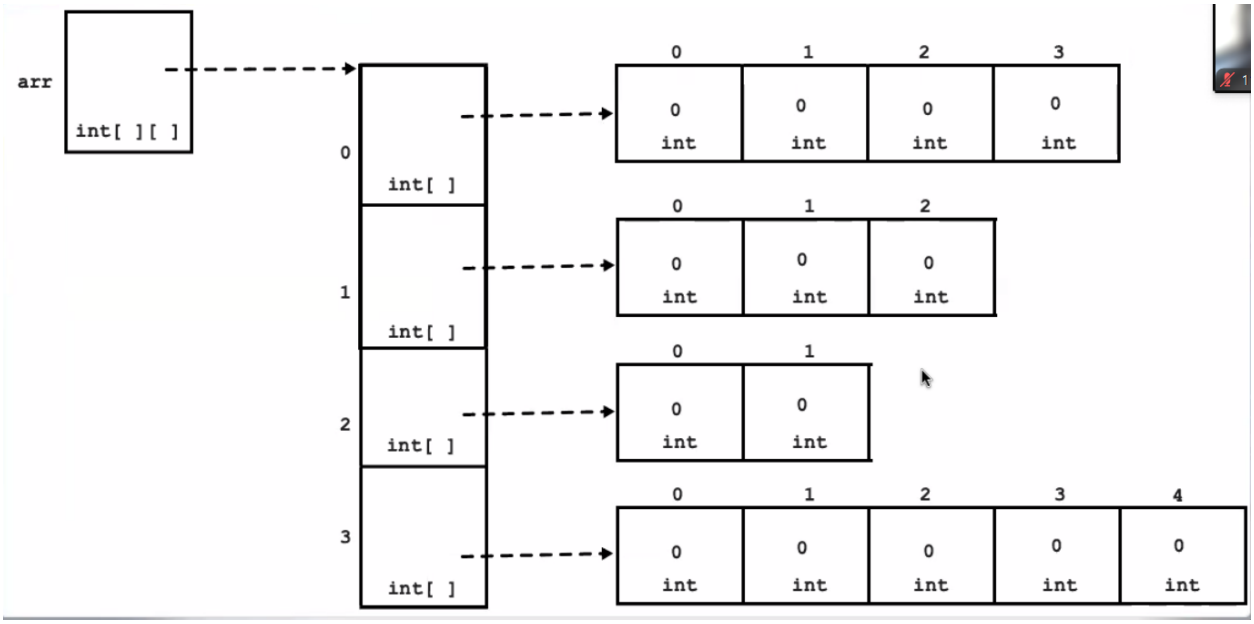
Arrays class is part of Java.util

→ Arrays.deepToString(arr) is used to print multidimentional array as string

→ Arrays.toString(arr[rows]) is used to print single dimentional array

//explore ArrayUtils (MVN repo)

→ Ragged array: It is array of arrays whose element arrays are of variable length.




```
int arr[][] = null;
int[] arr[] = null;
int[][] arr = null;
int[][] arr = new int[][] {{10,20,30}, {40,50,60,70}};

int[][] arr = new int[2][];
arr[0] = new int[5];
arr[0] = new int[6];
```

enum in c/c++

Enum is used to improve user readability of code

Enum is used to give names to the constants

```c
#include<stdio.h>
int sum( int num1, int num2 ){
    return num1 + num2;
}
int sub( int num1, int num2 ){
    return num1 - num2;
}
int multiplication( int num1, int num2 ){
    return num1 * num2;
}
int division( int num1, int num2 ){
    return num1 / num2;
}

typedef enum ArithmeticOperation{
    EXIT, SUM, SUB, MULTIPLICATION, DIVISION    //Enum constants
    //EXIT=0, SUM=1, SUB=2, MULTIPLICATION=3, DIVISION=4    //0,1,2,3,4: Ordinal
}ArithmeticOperation_t;

ArithmeticOperation_t menu_list( void ){
    ArithmeticOperation_t choice;
    printf("0.Exit\n");
    printf("1.Sum\n");
    printf("2.Sub\n");
    printf("3.Multiplication\n");
    printf("4.Division\n");
    printf("Enter choice    :   ");
    scanf("%d", &choice);
    return choice;
}
```

enum in java

```
enum Color{
    RED, GREEN, BLUE;   //references of enum
// 0     1      2   : Ordinal
}

public class Program{
    public static void main(String... args){
            System.out.println(Color.RED); // output: RED
            System.out.println(Color.RED.name()); // output:
```

```
RED
            System.out.println(Color.RED.ordinal()); // outpu
t: 0

            Color[] colors = Colors.values();
            for(Color color: colors){
                System.out.println(Color.name()+" "+Color.ord
inal());
            }

            string name = "RED";
            Color color = Color.valueof(name);
    }
}
```

→ When desired argument is not present in enum: Exception: Java.Lang.IllegelArgumentExecption

→ in java, enum is used to name the literals.

//difference between enum in c++ and java

→ for enum, .class file is created

→ it is a non premitive type

→ Synthetic class: class which we do not define at source code but gets generated at compile time.

→ enum is synthetic class, it is final class

→ since it is final class after compilation, we cannot extend enum

→ In java, all the enums are extended from Java.Lang.Enum

Java.Lang.Enum: common base class of all Java language enumeration types

→ Members of enum are considered as reference of same enum (public static final fields)

```
enum Day{
    SUN("Sunday");
    private String name;
    private Day(String name){
        this.name = name;
    }

    public String getName(){
            return this.name;
    }
}

public class Program{
    public static void main(String... args){
        Day day = Day.SUN;
        System.out.println(day.name()); // output: SUN
        System.out.println(day.ordinal()); // output: 0
        System.out.println(day.getName()); // output: Sunday
    }
}
```

→ use parenthesis to give name to literal

→ there must be a constructor

→ Modifiers for enum: private or packaged level private

→ We cannot change ordinal of enum value;

→ enum cannot extend another enum but can extend interface.

```
Compiled from "MainClass.java"
final class com.classwork.enumpractice.Colors extends java.lang.Enum<com.classwork.enumpractice.Colors> {
  public static final com.classwork.enumpractice.Colors RED;

  public static final com.classwork.enumpractice.Colors GREEN;

  public static final com.classwork.enumpractice.Colors BLUE;

  public static com.classwork.enumpractice.Colors[] values();
    Code:
       0: getstatic     #13                 // Field $VALUES:[Lcom/classwork/enumpractice/Colors;
       3: invokevirtual #17                 // Method "[Lcom/classwork/enumpractice/Colors;".clone:()Ljava/lang/Object;
       6: checkcast     #18                 // class "[Lcom/classwork/enumpractice/Colors;"
       9: areturn

  public static com.classwork.enumpractice.Colors valueOf(java.lang.String);
    Code:
       0: ldc           #1                  // class com/classwork/enumpractice/Colors
       2: aload_0
       3: invokestatic  #22                 // Method java/lang/Enum.valueOf:(Ljava/lang/Class;Ljava/lang/String;)Ljava/lang/Enum;
       6: checkcast     #1                  // class com/classwork/enumpractice/Colors
       9: areturn
```

→ 4 major parts/elements/features/pillers of java oop

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

→ 3 minor pillers

1. Typing (Polymorphism)
2. Concurrency
3. Persistence

Object oriented Programming Structure/System (OOPS) - Alan Kay

→ It is not a syntax, it is process/methodology which is used to solve real world problems using class and objects

→ 3 Phases of object oriented development

1. Object Oriented Analysis (OOA) - pen/paper or UML Diagram
2. Object Oriented Design (OOD) - UML and Design Patterns
3. Object Oriented Programming (OOP) - Syntax of OO Programming Language

→ Object Oriented Analysis and Design with Application: Graddy Booch

→ According to Graddy Booch , there are 4 major pillars of oops: (Essentials for oo language)

1. Abstraction: To achieve simplicity
2. Encapsulation: To hide implementation and provide data security
3. Modularity: To reduce module dependency.
4. Hierarchy: To achieve reusability

→ According to Graddy Booch, there are 3 minor pillars of oops (Features for oo language)

1. Typing / Polymorphism: To reduce maintenance of the System
2. Concurrency: To utilize hardware resources efficiently
3. Persistence: To maintain state of the instance in file/HDD.

→ Abstraction is a major pillar of oops

→ Process of getting essential things from the class is abstraction.

Creating a instance and calling method on it is basically a abstraction

→ Abstraction helps us to achieve simplicity

→ Encapsulation is major pillar of oops

→ Process of providing implementation for the abstraction is called encapsulation.

→ To achieve abstraction, there's need to provide implementations like class, fields and methods for it. This implementation is known as encapsulation.

→ Process of declaring fields private is data encapsulation

→ Process of giving controlled access to the field using methods of class is Data Security

```
Write code to explain encapsulation and abstraction
```

→ Modularity is a major pillar of oops.

→ Modularity: Process of developing complex systems with the help of small modules is known as modularity.

→ Purpose of modularity is to reduce module dependency.

→ Hierarchy is major pillar of oops

→ Level/order/ranking of abstraction is called as hierarchy

→ Main purpose of hierarchy is to achieve code reusability

→ reduce dev time, dev cost, dev efforts.

1. Has-a / Whole part Hierarchy: represents association

   2 forms: aggregation & Composition

   Loose coupling in java

2. Is-a / Kind-of Hierarchy: represents Inheritance

   Inheritance

   Strong Coupling

3. Use-a Hierarchy: Dependency
4. Creates-a Hierarchy: Instantiation

→ Typing is minor pillar of oops

→ also called as polymorphism

→ Ability of any thing to take multiple forms is known as polymorphism

→ Main purpose of polymorphism is to reduce maintenance of the System

→ compile time polymorphism: method overloading

→ run time polymorphism: method overriding

→ Concurrency is minor Pillar of oops

→ concurrency is method of executing multiple processes

→ in java, we implement is using threading

→ useful to utilize hardware resources efficiently

→ Persistence is minor pillar of oops

→ Process of maintaining of state of object java instance in file is called persistence

→ main purpose to maintain the reliability of system

→ Association: when there is a “has a” relationship between types

eg. Room has walls, Library has books, car has a engine.

→ If any object is a part or component of another object the it is called as association

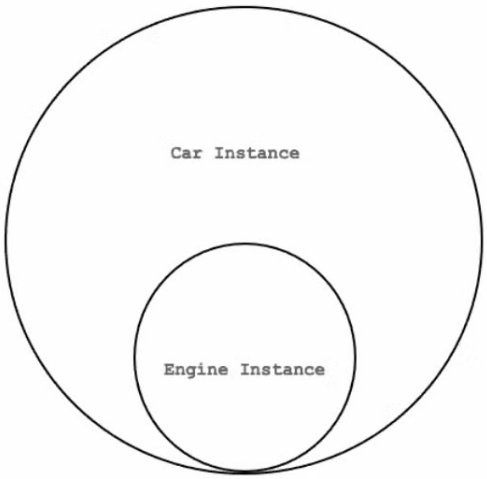
→

Example: Car & Engine

1. Car has a Engine

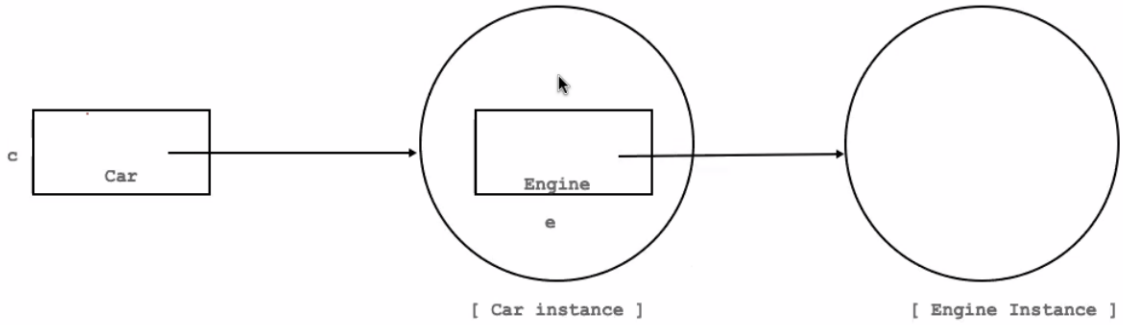
2. Engine is part of car

```
class Engine{
}

class Car{
   Engine e = new Engine( ); //Association
}

Car c = new Car( );
```



→ In java object cannot be part of other object directly. it can be part of it through referance variable.

```
class Engine{

}

class Car{
   private Engine e;
   public Car( ){
      this.e = new Engine( ); //Association
   }
}

class Program{
   public static void main( String[] args ){
      Car c = new Car();
   }
}
```



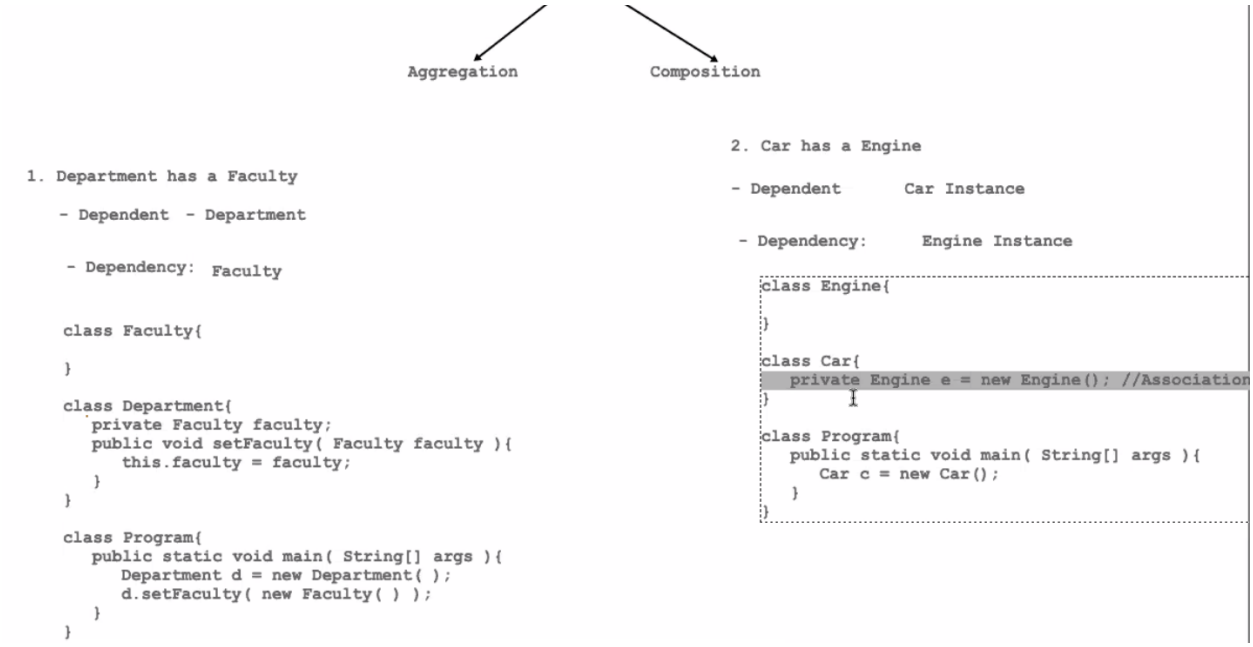
→ Loose Coupling:

Car.setEngine = null;

→ 2 forms: Aggregation, Composition

→ In case of association Dependency instance can exist without dependent instance, it is called aggregation association (Loose coupling)

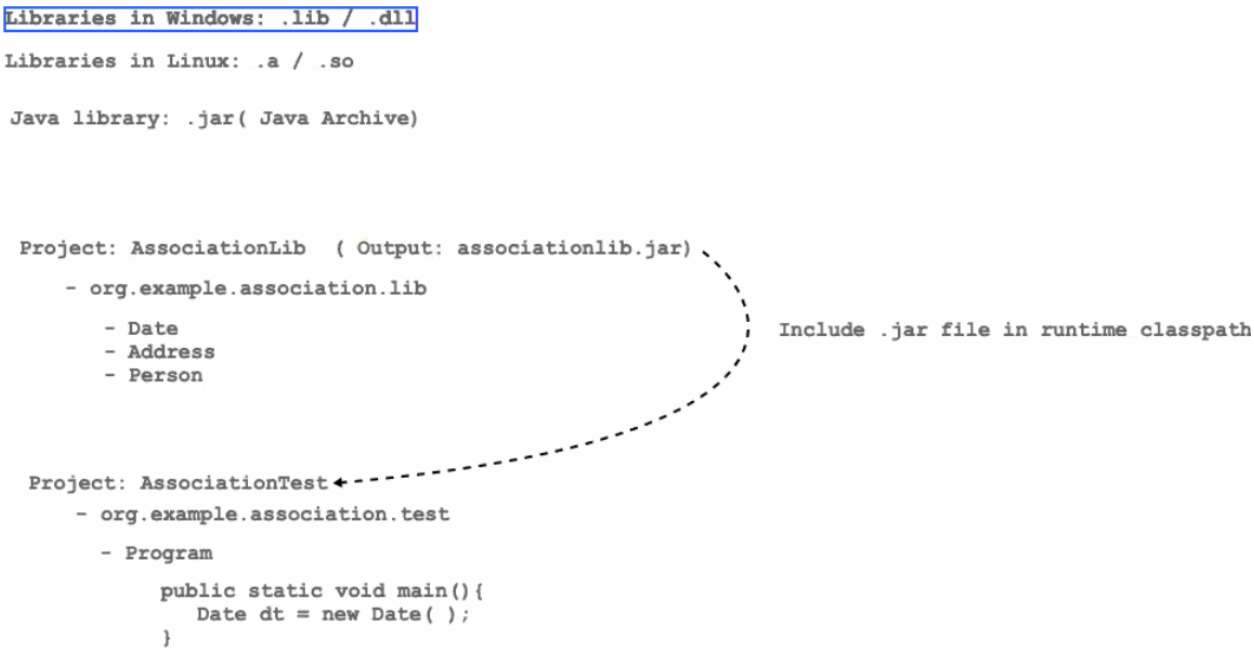
→ In case of association Dependency instance can exist without dependent instance, it is called composition association (Tight coupling)



For java Libraries: jar (java archive)

Windows libraries: .lib/.dll

Linux Libraries: .a/.so



```java
package org.example.association.lib;

public class Person {
    private String name;                        //Association
    private Date birthDate = new Date( );       //Association( Composition )
    private Address currentAddress;             //Association( Aggregation )
}
```

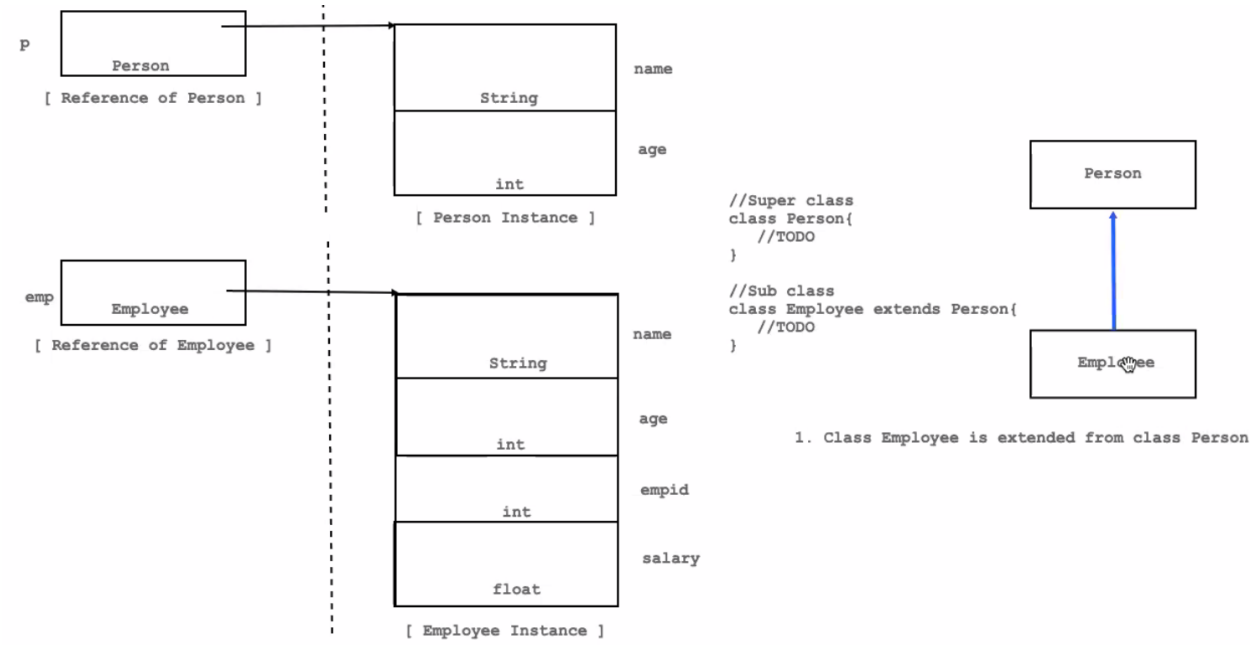
MANIFEST.MF file contains metadata about jar files

//Practice AssociationLib and AssociationTest

//lombok jar file

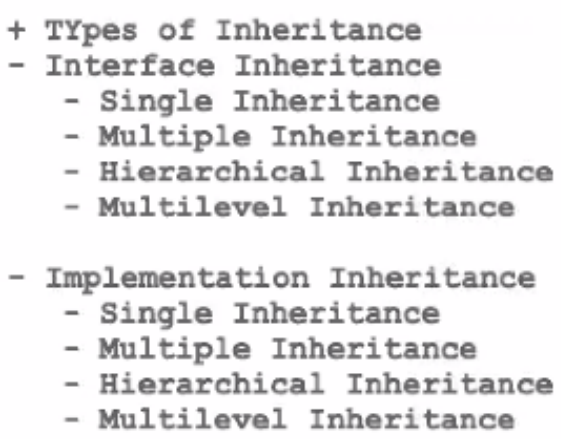
Inheritance

```
//Super class
class Person{

}

//Sub class
Class Employee extends Person{

}
```



→ Multiple Inheritance of Class is not allowed in java

```
Classes: C1, C2, C3, C4

C2 extends C1; //OK

C3 extends C1, C2;  //Not OK

  C2 extends C1; //OK
C3 exetends C2;   //OK

C2 extends C1

C3 extnds c1

C4 extends C1
```



→ In case of inheritance if supertype and subtype are both class, it is implementation inheritance

→ During inheritance, members of superclass inherit into subclass and occupy memory.

→ In java all the members of superclass inherit into subclass except constructor.

→To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

```
Classes c1,c2,c3
c2 extends c1;     //Single Inheritance
c3 extends c2;     //Multilevel Inheritance
c4 extends c1;     //Hierarchical Inheritance

c2 extends c1,c2;  //Multiple Inheritance: Not allowed in Jav
a
```

→ Any Sub class members would not get inherited to superclass

→ We can call methods of superclass for subclass instance

→ When we call sub class constructer, the constructor of parent class gets executed first and then sub class's constructor gets called.

→ by default parameter-less constructor of super class gets called from sub class

→ If we want to call any constructor of super class from constructor of sub class explicitly then we should use super statement.

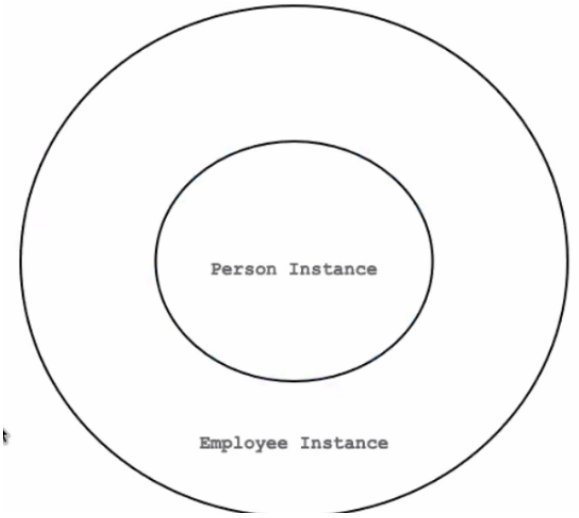
this is also constructor chaining.

super statement call must be the first statement inside the sub class constructor.

→ Nested classes from super class also get inherited into sub class.

→ This process of reusing members of superclass into subclass is called inheritance

→ Inheritance is tight coupling



→ Field is not visible error will be shown when you try to access private members of class

→ Getter and setter is the way to access private members outside of the class

→ Private members gets inherited in subclass

→ Private fields of class can only be accessed by methods of same class only

→ Packaged level private members can only be accessed within the same package

→ Protected fields can be accessed in same package and within subclass in different package.

→ Public fields can be accessed anywhere

	Same Package			Different Package	
Access Modifier	Same Class	Sub Class	Non Sub Class	Sub Class	Non Sub Class
private	A	NA	NA	NA	NA
package level private	A	A	A	NA	NA
protected	A	A	A	A	NA
public	A	A	A	A	A

→ When to decide to use inheritance?

> 💡 According to the client requirement if the implementation of imported package class is logically incomplete or partially complete, we should inherit it in a subclass.

→ Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

→ If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

→ Method overriding is used for runtime polymorphism

→ A static method cannot be overridden

→ It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

→ We cannot override java main method as it is static.

→ Rules for method override

1.
2. Return type must be covarient : Either same or subtype of return type
3. Same method Name, Same number of parameter of methods & Same type of parameters
4. Checked exception list

→ Subclass overrides method of superclass

```
class Person{
    private String name;

    public void getName(){
        System.out.println(this.name);
    }
}

class Employee extends Person{
    private String dept;

    @Override
    public void getName(){
        super.getName();
        System.out.println(this.dept);
    }
}
```

→ When names of super class and sub class members are same, the preference is given to subclass member. in other words, subclass method is shadowing super class method.

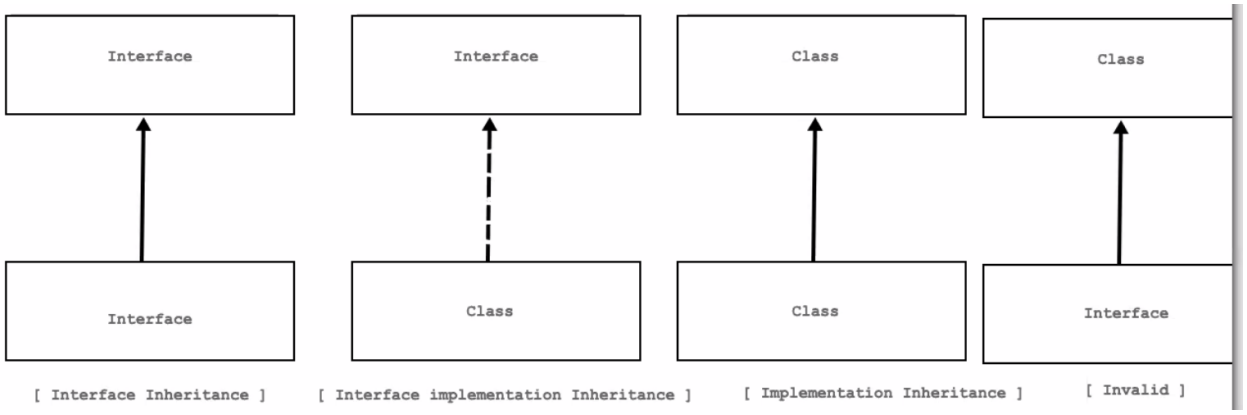
→ super keyword is used to access super-class’s original method into overridden method.

→ Super keyword is used to access any member of superclass into subclass

//shadowing

```java
package org.example.main;

class A{
    int num1 = 10;
    int num3 = 30;
}
class B extends A{
    int num2 = 20;
    int num3 = 40;

    public void print( ) {
        System.out.println("Num1     :    "+this.num1);    //10
        System.out.println("Num2     :    "+this.num2);    //20

        System.out.println("Num3     :    "+super.num3);   //30
        System.out.println("Num3     :    "+this.num3);    //40
    }
}
public class Program {
    public static void main(String[] args) {
        B b = new B( );
        b.print();
    }
}
```

Types of inheritance



During inheritance if super and sub types both are interface, then it is interface inheritance

1. Single Interface Inheritence
2. Multiple Interface Inheritence
3. Hierarchical Interface Inheritence
4. Multilevel ''

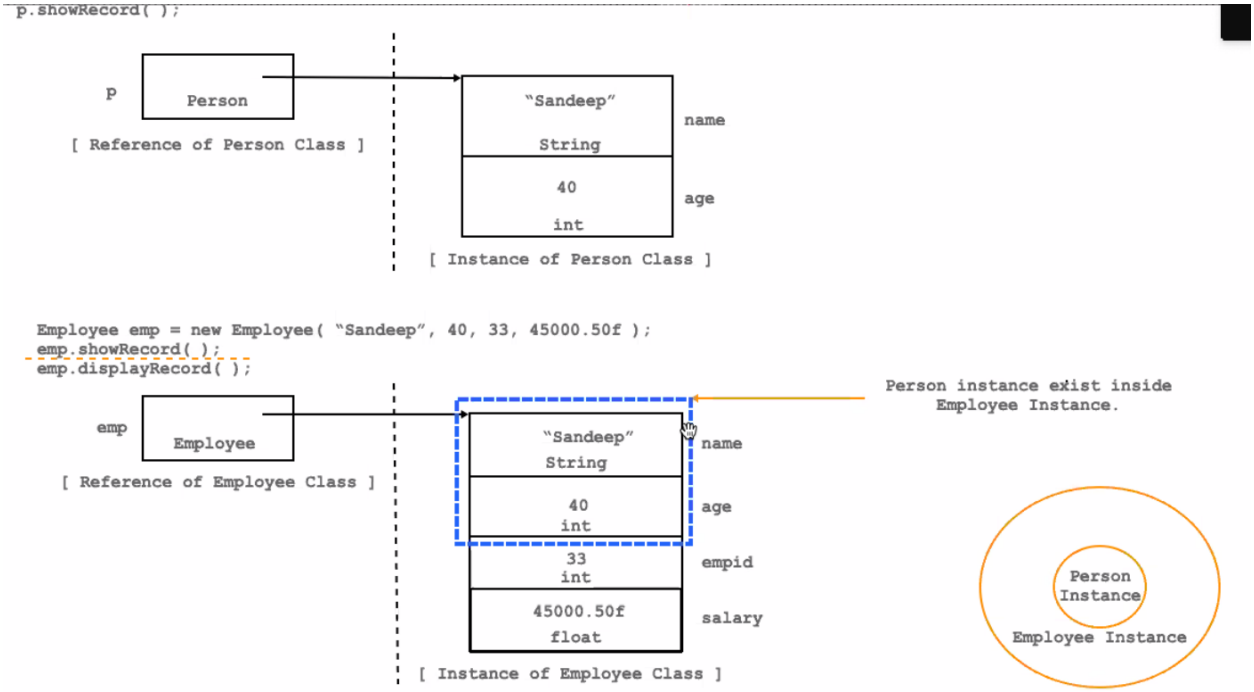
if super and sub type both are class then it is implementation inheritance

1. Single implementation inheritence
2. Multiple '' (Not Allowed)
3. Hierarchical ''
4. Multilevel ''

if super type is interface and sub type is class then it is Interface Implementation Inheritance

A interface cannot be a subtype of class. i.e. it cannot extend class.

//Keshav Dattatray book



→ Employee IS A person

→ As members of super class present in sub class, we can consider subclass instance as superclass instance

```
//Person: Superclass | Employee: Subclass
Employee emp = new Employee();
Person p = emp;      //We can use subclass referance in place
of superclass [Upcasting]
Person p = new Employee();   // [Upcasting]

Person p = new Person();
Employee emp = p    //NO : Type missmatch error (Compilation
Error)
Employee emp = (Employee) p;    // java.lang.ClassCastExcepti
on
//We cannot use superclass instance in place of subclass
```

→ Since members of superclass do not inherit into subclass, we cannot consider instance of superclass as instance of subclass

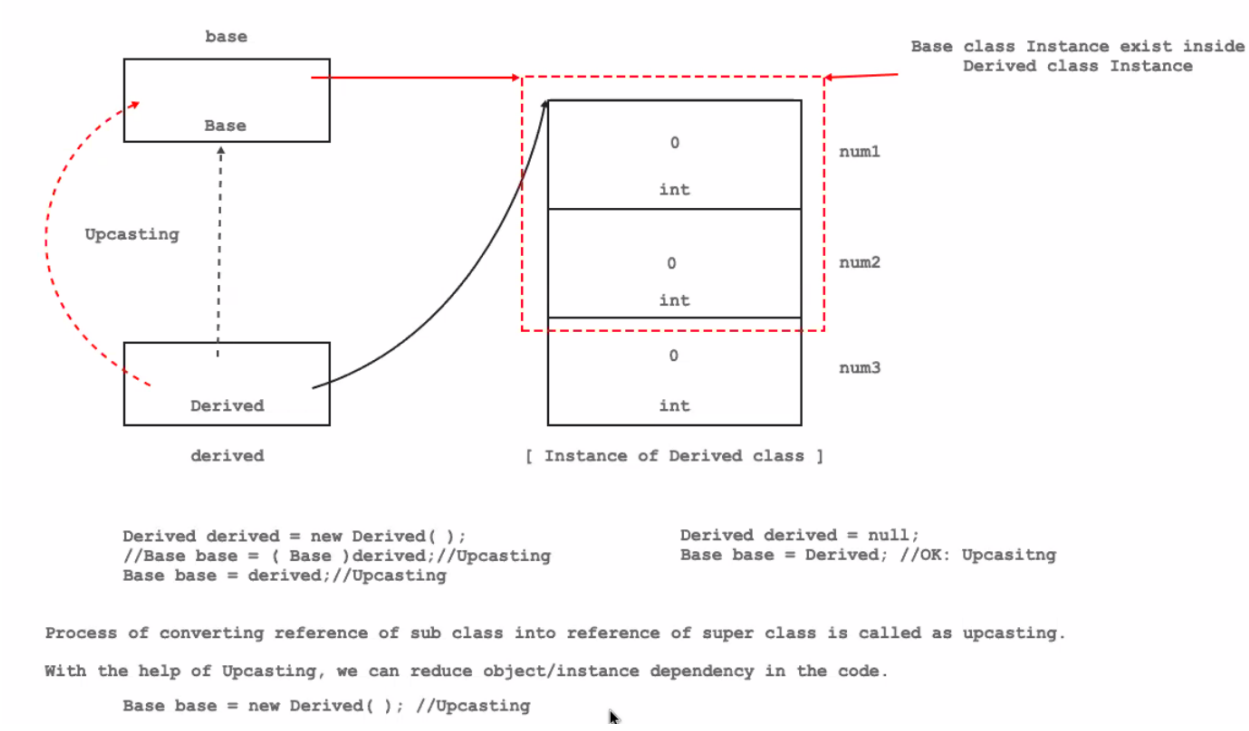
Method Overloading	Method Overriding
Signature must be different	signature should be same
return type is not considered	return type is considered
compile time polymorphism	run time polymorphism (Dynamic method dispatch)
method must be in same class	method must be in superclass and subclass

→ when reference of subclass is used as reference of superclass, the superclass reference can only access the relevant members only

→ it is known as Object slicing

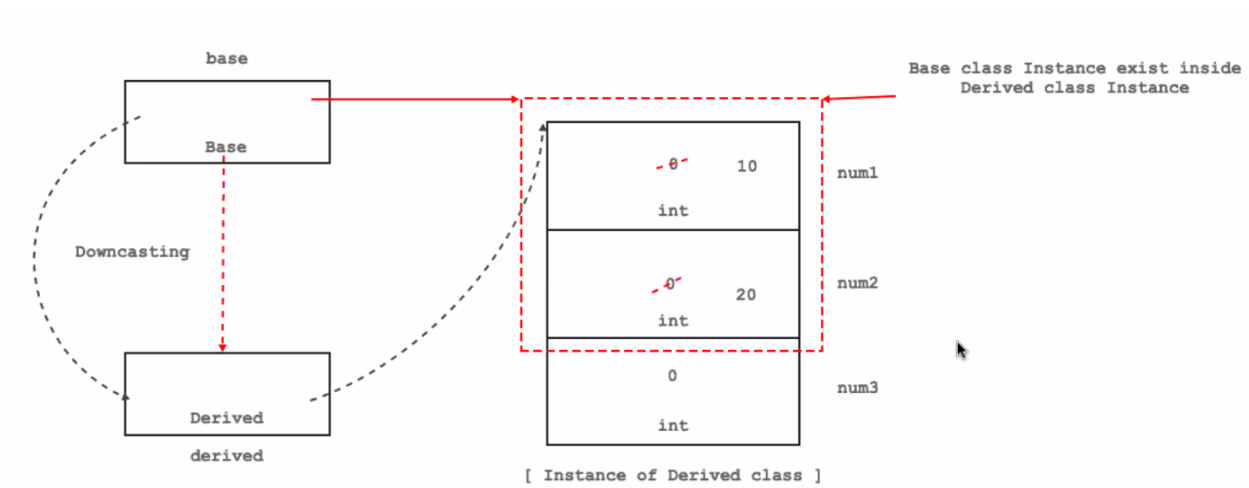
→ Process of assigning instance of subclass to reference of superclass is called as upcasting

→ With the help of upcasting, we can reduce object /instance dependency in the code



→ Downcasting is process of converting reference of super class into reference of sub class.

→ Without upcasting, if we try to do downcasting, we'll get java.lang.ClassCastException

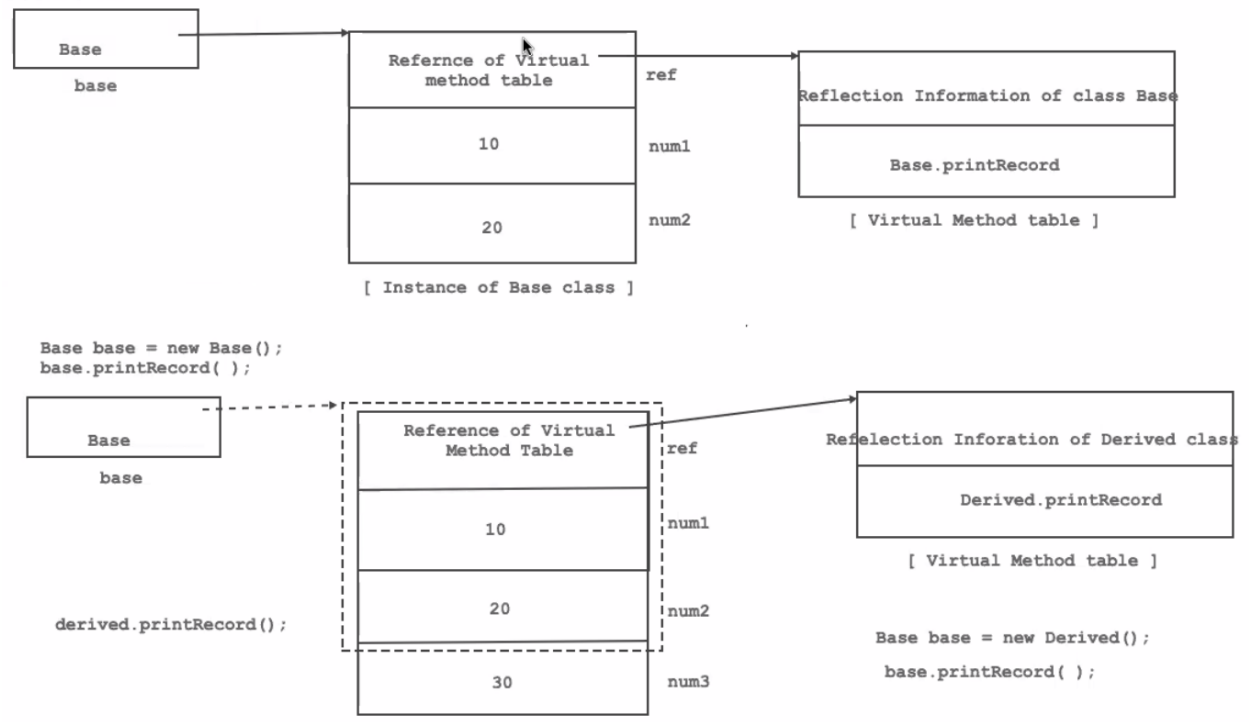



```
Base base = new Derived();//Upcasting
Derived derived = (Derived)base;//Downcasting

Process of converting reference of super class into reference of sub class is called as downcasting.

If downcasting fails then JVM throws ClassCastException.

Base base = new Base();
Derived derived = (Derived )base;//ClassCastException

Base base = null;
Derived derived = (Derived)base;//OK
```

→ Virtual methods: In java all the methods except static methods and constructor are virtual methods.

→ In upcasting, if methods with same sign are present then derived method gets called.

→ In upcasting, process of calling overridden method of superclass in subclass by using reference of subclass is called dynamic method dispatch



instanceof is a operator which checks if reference is instance of given class

→ We use upcasting to reduce object dependency which reduces maintenance of code

→ A class which hides the complex instance implementations from the main method is called as factory class and the method is known as factory method.

→ Exception is instance which is used to send notification to the end user if any exception occur in system.

→ If we want to manage os resources properly we should use exception handling

→ file, thread, socket, i/o devices, os api

Non java or unmanaged resources in java : These are the resources which are not managed by jvm

java.lang.ArithmaticException: Exception that occurs while Arithmatic operations

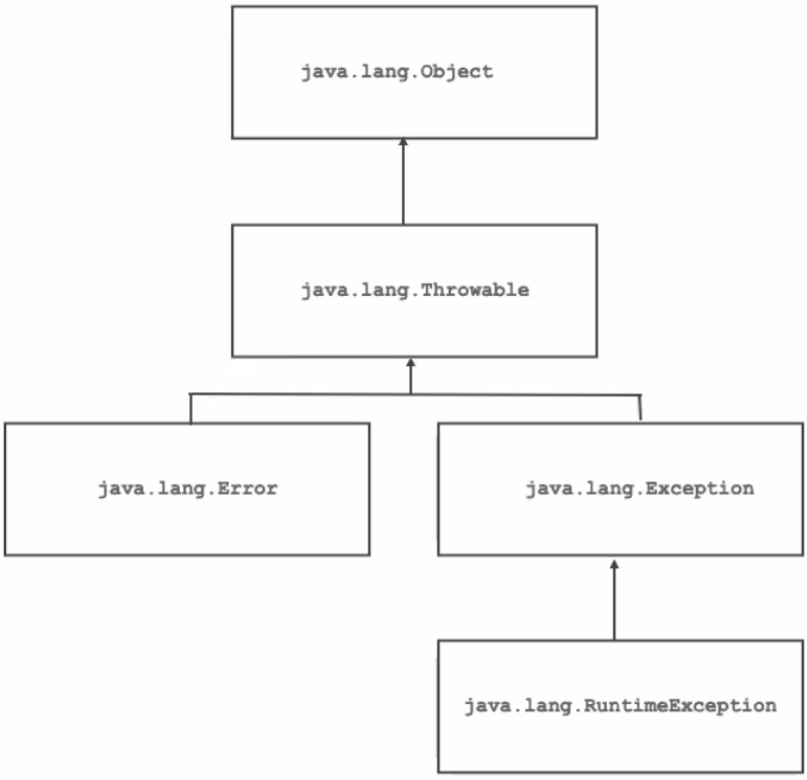
→ File handling, socket programming, jdbc, threading - Exception handling can be used

//setjmp,longjmp

> 💡 In java java.lang.Throwable is considered as superclass for all errors and exceptions

> 💡 java.lang.Error and java.lang.Exception class are subclass of java.lang.Throwable

→ In java, if we want to throw or catch any object then type of that object must be the subclass of throwable class



→ if runtime error generated due to environment or failure of environment, it is Error. e.g. not available memory

→ We can write try catch block to handle Errors but it is advised that we should not catch them as we cannot avoid errors

→ if runtime error generated due to Application or failure of Application , it is Exception. e.g. not available memory

→ Since we cannot recover from Error but we can recover Exception. so it is not recommended to use try-catch block for Errors but the must be used for Exceptions

→ keywords to handle exceptions: try, catch, throw, throws, finally.

→ In java, there are 2 types of exception

1. Checked exception : java.lang.Exception and all of it's subclasses except Runtime ones
2. Unchecked exception: java.lang.Runtime and all of it's subclasses

These types are not designed for JVM but designed for java compiler

→ if a method throws checked exception, it is mandatory to handle it in try-catch block.

→ if a method throws unchecked exception, it is optional to handle it in try-catch block.

→ There are 5 types of throwable constructor.

→ Stacktrace contains complete information about the exception

→ Stacktrace is for developers aid to handle the exceptions

→ If you want to keep watch on statements for possibility of exception causing, we use try block (try handler).

→ You cannot write catch or finally before try.

```
try{
        int result = num1/num2;
        System.out.println(result);
    }
```

→ throw keyword is used to generate exception

→ throw can only throw the objects which are throwable.

→ throw statement is jump statement (Control will get out of try block)

```
try {
    if( num2 == 0 ) {
        ArithmeticException ex = new ArithmeticException("Value of denominator should not be zero");
        throw ex;
    }else {
        int result = num1 / num2;
        System.out.println("Result  :   "+result);
    }
}
```

→ If we want to handle exception, we must use catch block

→ You cannot write catch before try or after finally.

→ we can use loggers in catch block

→ in java if you want to release any local resource we should use finally.

→ Finally is not a method, it's a block.

→ there can be multiple catch for try but only one finally

→ Finally block must appear only after all catch block

→ JVM always executes finally block so, we must use it for releasing resource.

→ finally block can only be avoided from being executed by using system.exit(0).

```
try {
    System.out.println("Inside try block");
    if( num2 == 0 )
        throw new ArithmeticException("Value of denominator should not be zero");
    int result = num1 / num2;
    System.out.println("Result  :   "+result);
}catch( ArithmeticException ex ) {
    System.out.println("Inside catch block");
    //Logger: Logback, Log4j, java.util.Logging
    //System.out.println( ex.getMessage());
    ex.printStackTrace();
}finally {
    System.out.println("Inside finally block");
    sc.close();
    System.out.println("File Close()");
}
```

→ Single try block may have multiple catch blocks.

→ In java 7 update, we can use single catch block for multiple Exceptions (Multi-catch block)

```
catch (ArithmeticException | InputMismatchException ex) {
//Multi-Catch block
            ex.printStackTrace();
```

→ while handling exception using catch, if parent-child class relation is there, then we must handle child class exception first and then superclass exception

→ Exception class reference can contain reference of all checked and unchecked exception.

so, when we implement generic catch block, we can make use of Exception class.

→ There's no compulsion to write catch block

→ We cannot write try block alone, there must be either finalize or catch statement with it

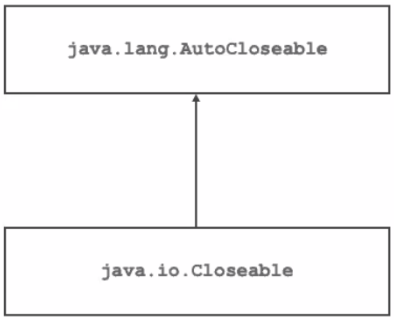
→ A class should override all the methods implemented from parent interface otherwise class will become abstract class.

//Autoclosable and closable

→ any class which implements autoclosable or closable interface, such class is called as resource type in java.

→ instance of resource type is called as resource in java eg. scanner class

```
- AutoCloseable is interface declared in java.lang package.
   - Method: void close( )throws Exception;

- Closeable is interface declared in java.io package.
   - Method: void close( )throws IOException

- A class which implements AutoCloseable / Closeable interface  is called
as "Resource Type" in Java.

 - Instance of resource type is called as "Resource" in Java.
```




```
class Test implements AutoCloseable{ //Resource Type
    private int number;
    private Scanner sc = new Scanner(System.in);
    public Test( ) {
        this.number = 0;
    }
    public void acceptRecord( ) {
        System.out.print("Number   :   ");
        this.number = sc.nextInt();
    }
    public void printRecord( ) {
        System.out.println("Number :   "+this.number);
    }
    @Override
    public void close() throws Exception {
        this.sc.close();
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        try {
            Test t = null;
            t = new Test( ); //Resource
            t.acceptRecord();
            t.printRecord();
            t.close();
        }catch( Exception ex ) {
            ex.printStackTrace();
        }
    }
}
```

→ If we want to keep watch on group of exception then we should put all statements inside try block

→ If we want to define try block then it must have atleast :

1. Catch block
2. finally block
3. Resource

else it throws error.

→ Java introduced the try-with-resources statement in Java 7, which simplifies the process of managing resources that must be closed. The statement allows us to declare one or more resources within the parentheses of a try statement. These resources are automatically closed after the try block is executed.

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

//try with catch
try{
    System.out.println("Hello");
}catch (Exception ex){
  Exception.printStackTrace();
}

//try with finally
try{
    System.out.println("Hello");
}finally(){
  sc.close();
}

//try with resource
try (Scanner sc = new Scanner(System.in)){
    System.out.println("Hello");
}
```

→ if we want to release local resource we must use finally block

→ To avoid resource leakage, try with resource block can be implemented as it directly closes the resource after execution of try block.

→ we can also define multiple resources in try block

→ we can define try catch block inside try catch block, it is known as nested try catch blocks

→ if there is not matching inner catch block, it will look for outer matching catch block

→ if all of the catch block are not matching, the control will go to jvm and it'll throw exception and terminates the program.

→ the exception that occurred in outer try block is handled by outer catch block only

→ it is bad practice to define try catch block in every method

→ throws keyword is used to delegate or transfer the handling of exception towards the caller.

→ throws can be used to handle both checked and unchecked exception

→ we can throw multiple exception using throws

```
public void add(a,b) throws A,B,C{

}

public void add(a,b) throws Exception{

}
```

→ If throws exception has multiple exceptions, we can directly use superclass exception to handle it

//shutdown hook

→ JVM has list of exceptions but in business logic if any exception occurs, JVM cannot handle it

→ if we want to handle exceptions occurred by business logic, we should use custom exception class

→ if runtime error gets due to developers mistake, it is considered as bug of application.

→ to handle bug, we should not write try catch block

→ We should not use try catch block for Error.

Rules of method overriding:

1. Access modifier of subclass overridden method should be same or wider
2. Return category of overidden method must be same or subtype of return of superclass method. (Must be covarient)
3. Method name, Number of parameters and type of parameters must be same
4. During overriding, checked exceptions list in subclass must be same or subset of superclass.

//Question:

```java
package org.example.main;

class Base{
    public void print( ) {
    }
}
class Derived extends Base{
    @Override
    public void print() throws InterruptedException {
        for( int count = 1; count <= 10; ++ count ) {
            System.out.println("Count    :    "+count);
            Thread.sleep(250);
        }
    }
}
public class Program {
    public static void main(String[] args) {
        Base base = new Derived( ); //Upcasting

        base.print();    //Dynamic Method Dispatch
    }
}
```

→ RuntimeException(cause);

cause is a instance of throwable class

→ The process of handling exception by throwing new type of exception is known as exception chaining

1. According to businees requirement, if method defined in superclass is 100% complete then we should declare superclass method as final
2. According to business requirement, if method defined in superclass is 100% incomplete then we should declare superclass method as abstract method
3. According to business requirement, if method defined in superclass is logically incomplete or partially complete then we should declare superclass method as non-abstract & non-final method (Need to override)

→ You cannot override final, static and private method in subclass

→ We can declare overridden method as final

→ Method of class which do not have body is abstract method and method of class which have body is concrete

→ If class contains atleast one abstract method then we have to make the class abstract

→ If we extend abstract then we have to either override abstract method of superclass or we need to define subclass as abstract

→ If the implementation of the class is logically 100% complete, we can define it as final

> 💡 We can create referance of abstract class but we cannot create instance of abstract class

→ we can design a class abstract without making methods abstract

→ Constructor of superclass which is designed to get called from sub class only is called as sole constructor

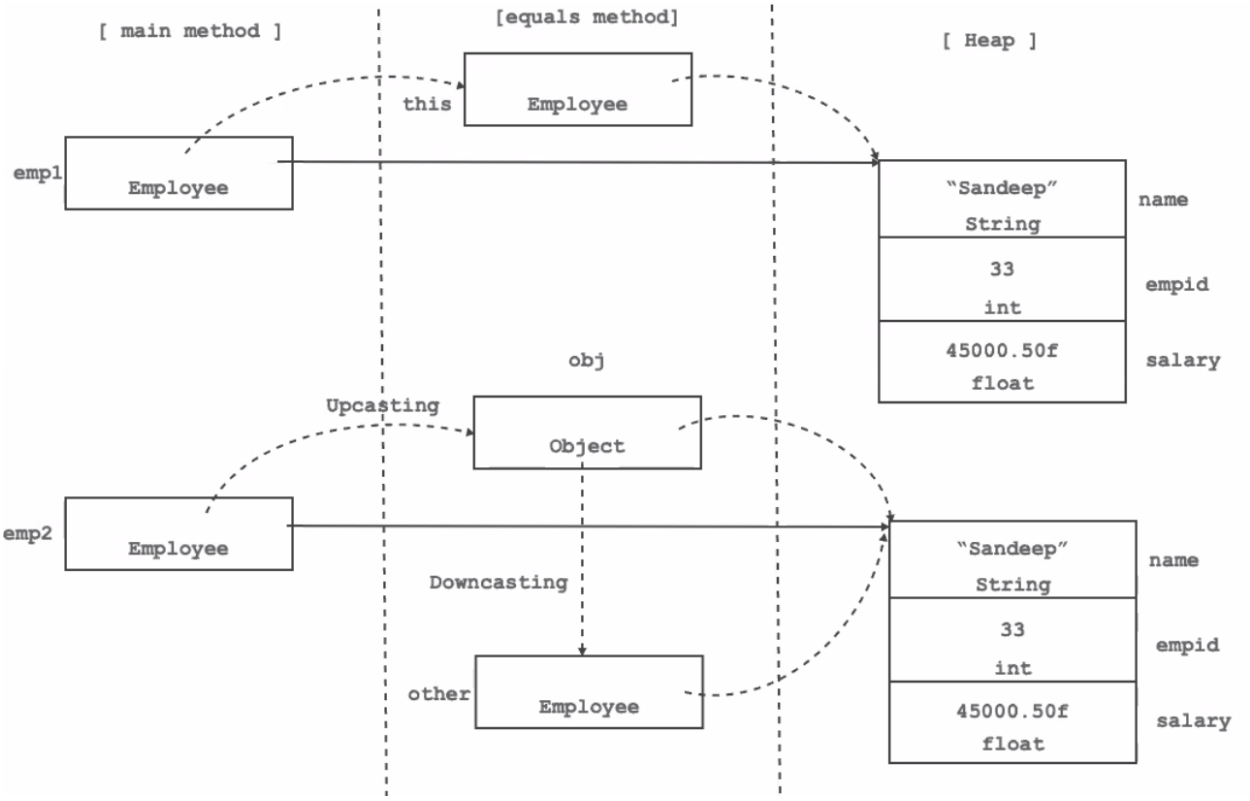
→ java.util.dictionary has sole constructor

→ java.lang.enum has sole constructor

If we want to compare primitive variables, then we should use == operator

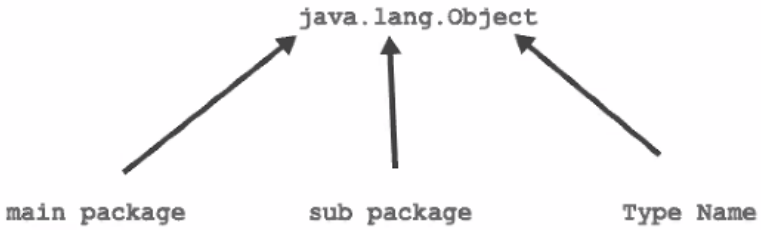
If we want to compare state of reference, then we should use == operator

if we want to compare states of instance which are not primitive type, we can use object.equals object



→ In java. Package can contain

- Sub-package
- Class
- Enum
- Interface
- Error
- Exception
- Annotation type



→ Methods inside java.lang,Object class:

- public String toString();
- public boolean equals(Object obj);
- public native int hashCode();
- Protected native Object clone() throws CloneNotSupportedException;
- protected void finalize() throws Throwable
- public final native class<?> getClass();
- public final void wait() throws InterruptedException;
- public final native void wait(long timeout) throws InterruptedException;
- public final void wait(long timeout, int nanos) throws InterruptedException;
- public final native void notify();
- public final native void notifyAll();

//Questions: java.math : BigInteger, BigDecimal

BigInteger:

→ java.lang.Number >> java.math.BigInteger

→ Range = (-2^(INT_MAX) to 2^(INT_MAX))

```
BigInteger b = BigInteger.valueOf(10);
BigInteger b1 = new BigInteger("99999999999999999");
```

```
System.out.println(b.add(b1))
```

BigDecimal

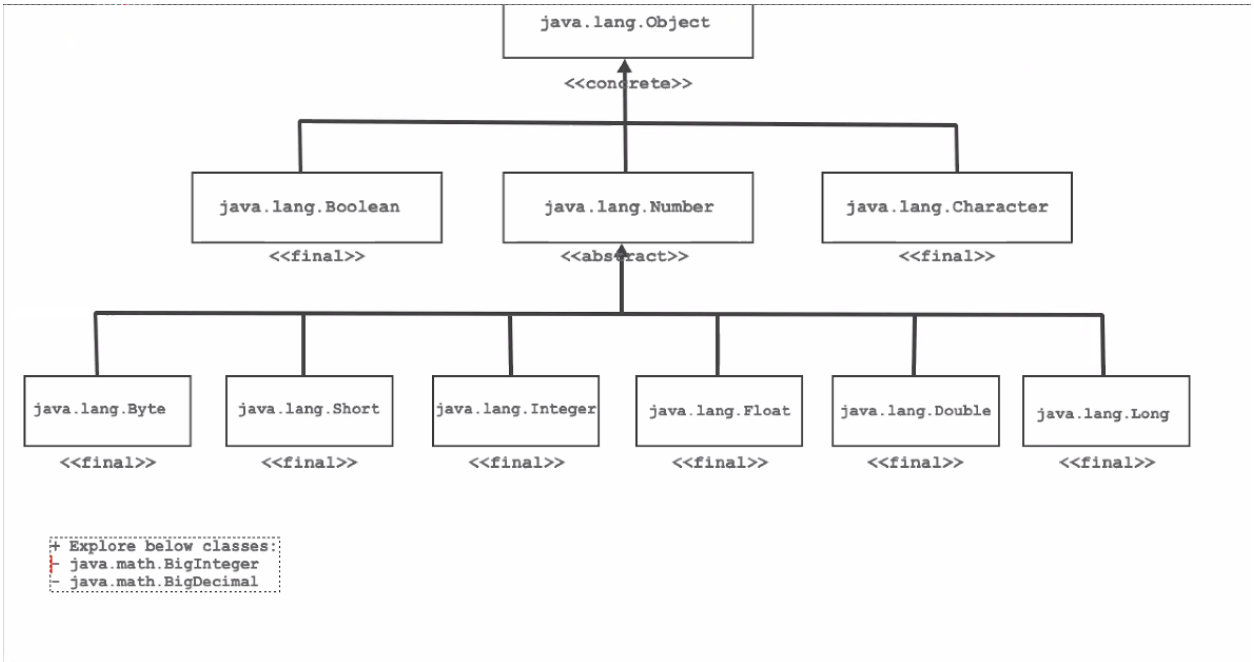
→ float and double are floating point numbers and they create some error while using them but BigDecimal gives no error.

→ java.lang.Number >> java.math.BigDecimal

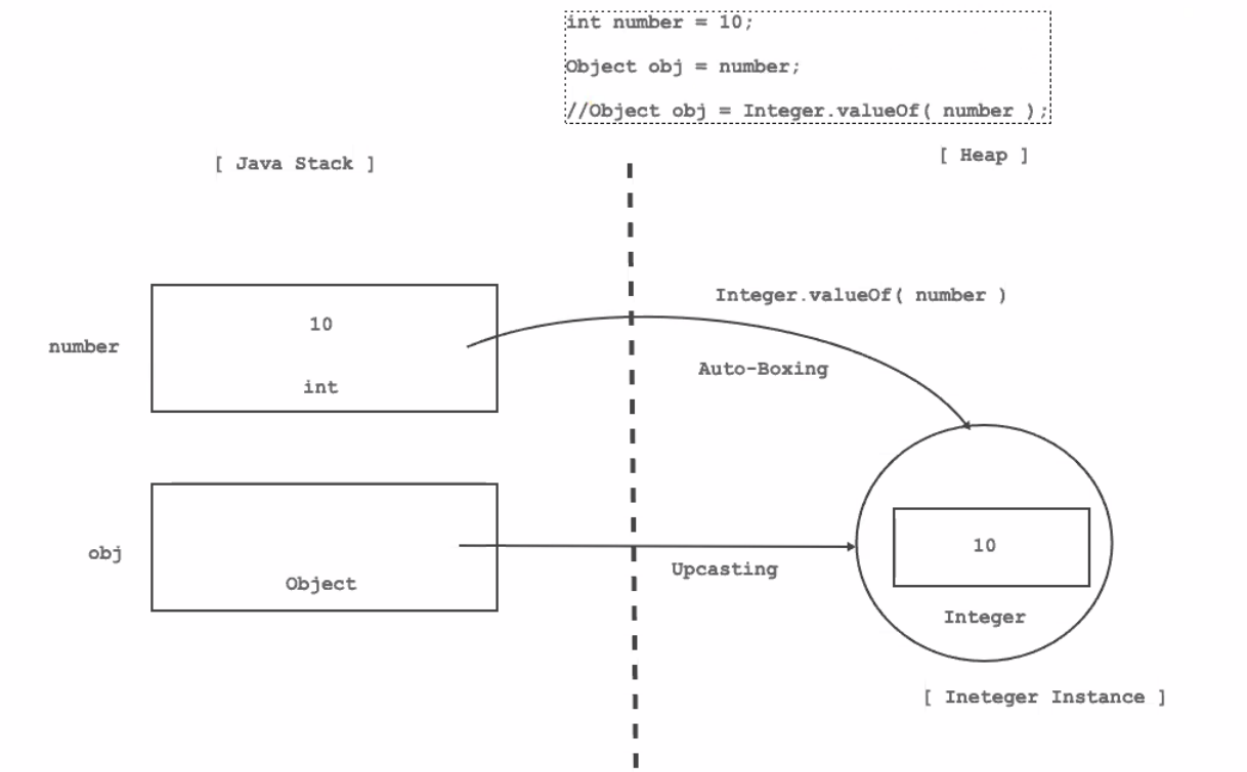
→ Scale : 32 bit integer

→ Range = if 0 and positive.. Up to scale decimal point

else if negative... 10^(-scale)




```
int number = 10;
Object obj = number    //Integer.valueOf(number)
//Auto-boxing and upcasting
```



//Assignment: Create a linear singly linked

//addfirst, addlast, deleteStart, deleteEnd, print

in java if you want to do generic programming

1. using java.lang.object class
2. using generics feature given by java

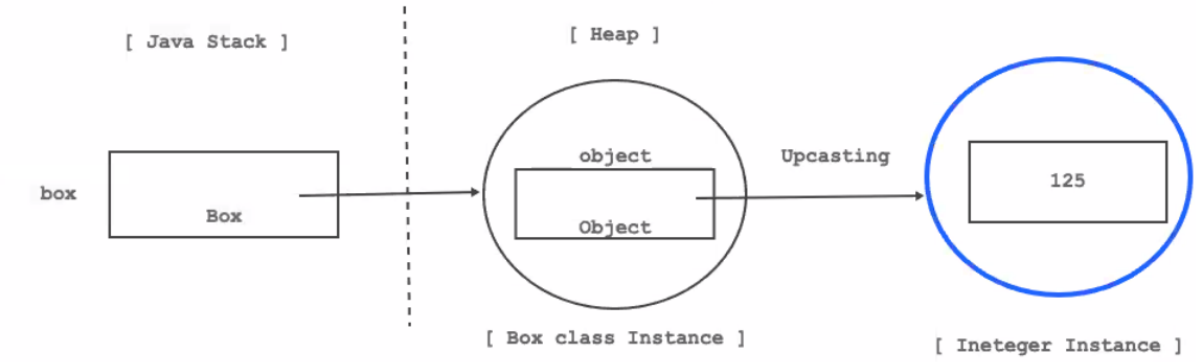
→ The goal of Generic Programming is to design generic algorithms and data structures so that their application scope is the widest possible without sacrificing performance.

Generic class using java.lang.Object

```
class Box{
    private Object object;
```

```
    public Object getObject(){
        return object;
    }

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

public class Program{
    public static void name(String[] args){
        int number = 12;
        Box box = new Box();
        box.setObject(125);      //Boxing and Upcasting

        Object object = box.getObject();
        Integer int = (Integer) object;   //Downcasting
        number = int.intValue();          //Unboxing
        System.out.println(number);
    }
}
```

```
Box box = new Box();
int number = 125;
box.setObject( number );   //box.setObject( Integer.valueOf(number) );
```




```
box.setObject(date);

Object object  = box.getObject();
String string = (String) object;    //ClassCastException: at
Runtime : Not a type safe
System.out.println(string);
```

→ using java.lang.Object class, we can implement generic programming but it does not provide type safety.

→ So, if you want type safe program, we should java generics feature.

using generics feature given by java

→ Diamond operator

```
//Parameterised class or Parameterised type
class Box <T>{              //Type parameter
    private T object;
```

```
    public T getObject(){
        return object;
    }

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

public class Program{
    public static vpid name(String[] args){
        Box<Date> box = new Box <Date>();     //Type Argument
        Box<Date> box = new Box <>();     //Type inference
        Date date = new Date();
        box.setObject(date);

        //String string = box.getObject();   (Cannot right it
now)
        Date dt = box.getObject();
        System.out.println(T);
    }
}
```

→ Generics feature provides us strong type checking at compile time so it provides type safety

→ if we use generics, we don't require explicit type casting

→ With the help of generics, we can create generic data structures and alogrithms

→ Type inference

→ if we do not specify type argument then default type argument is object

→ if we instantiate parameterized type without argument then the class is known as raw type

→ Type argument must be non primitive or reference type

Commonly used type parameters in generics:

> T: Type, E: Element, N: Number, K: Key, V: Value, R: Return Type, US: Second type Parameter

→ We can pass multiple type arguments to the class using generics

→ Bounded type parameter: when we want to add restrictions for declaration of type

```
class Box <N extends Number>{  //N is bounded Type Parameter

}
```

→ Java.util.Arraylist

→ Only on the basis of different type arguments we cannot overload methods

→ We should use wildcard diamond opearator <?>

```
private static void print(ArrayList<?> list) {  //? is called
as wild card which represents unknown type
        for( Object element : list )
            System.out.println( element );
    }
```

wild card types in java (?):

- unbounded wild card: <?>
- upper bounded wild card: <? extends Number>
- lower bounded wild card: <? super Integer>

→ We can also make methods generic

```
public static <T> void displayData(T object){
        System.out.println(object)
    }
```

→ Restriction over generics

- Cannot Instantiate Generic Types with Primitive Type
- Cannot Create Instances of Type Parameter
- Cannot Declare Static Fields Whose Types are Type Parameter
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

→ Type Eraser: Compiler erases type at compile time and considers it as raw type.

Read about restrictions on generics in java

Nested Class:

→ We can define class inside scope of another class, called as nested class

→ outer level class can either be public or default

→ there's no restriction for inner class for access modifier

```
//Top-level class
class Outer{     //Outer class
  //Nested class
    class Inner{   //Inner class
```

```
    }
}
```

→ By defining nested class, we achieve encapsulation

→ Types of nested class

  → Non static nested class / Inner Class

  → Static nested class

For better understanding, consider non static class as non static method of the class

→ Top level class can contain both static and non static members.

→ Inside non static inner class, we cannot define static members if it is non final.

→ Using instance we can access members of non static inner class inside method of top class

→ we can access static and non static members of outer class inside non static inner class.

→ Top level class contains reference to inner class instance

→ To access members of outer class inside inner class: we can use “Outer.this.num1”

→ To create static nested class instance: Outer.Inner = new Outer.Inner()

→ in java, we create static class static but not the top class

→ Non static members of outer class are not accessible in inner class

→ In java, we can define class in method which is called local class

→ We can access this class within method only

→ Local class has 2 types: Method local anonymous inner class, method local inner class

→ When you define non static class inside method, it is known as method local inner class

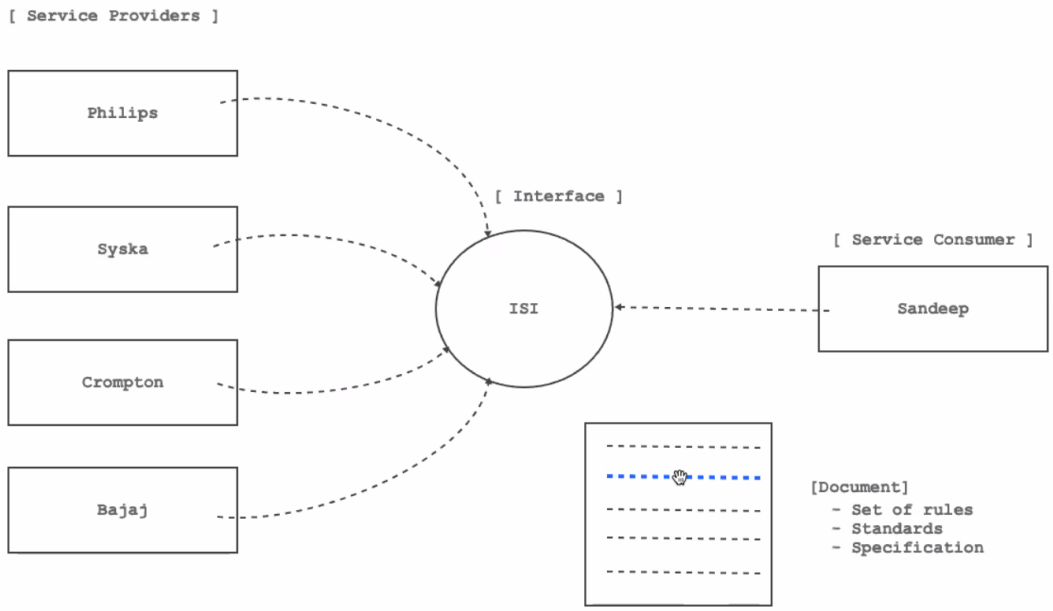
→ We can define class without name

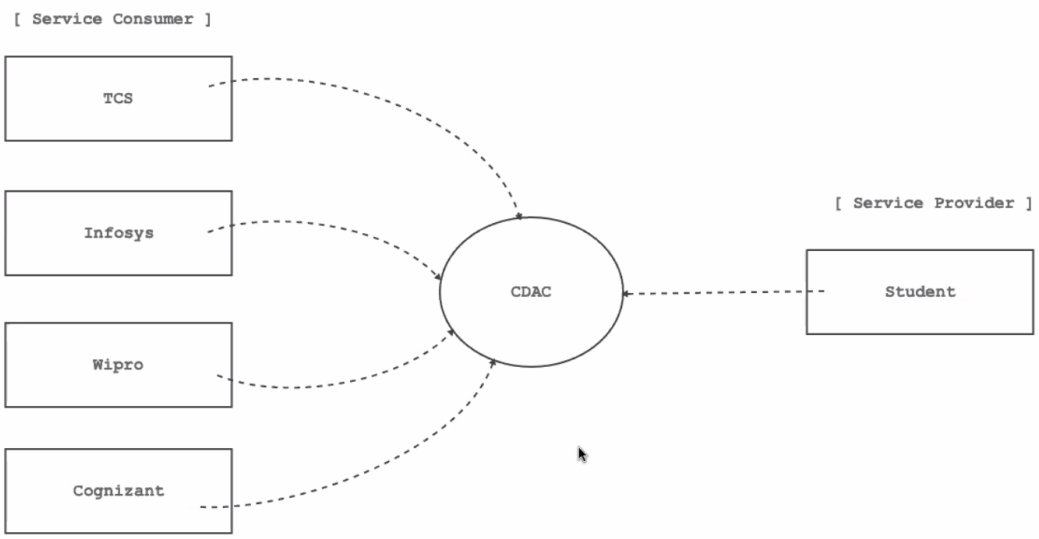
→ In java, every anonymous class is subclass

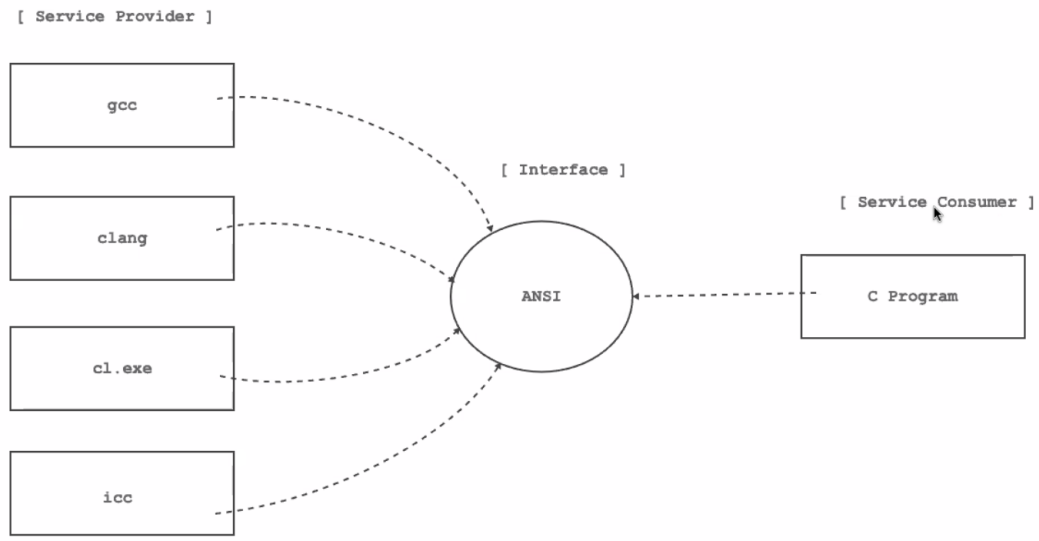
→ if we make changes in super class we need to recompile sub classes also this problem is called as fragile base class problem

→ To overcome this, we need interface in java

Interface in java:







→ Interface is a contract between service provider and service consumers

→ set of rules is considered as specification or standerds

→ if we want to define specification for subclasses, we should use interface in java

→ Interface is non primitive type in java

→ Interface is considered as blueprint for the class

→ Interfaces helps to minimize vendor dependency

→ Interfaces are mainly used to define abstractions.

→ interface is a keyword in java to create interfaces

→ We cannot create instance of interfaces

→ we can create reference of interface

→ can contain nested types, fields, abstract method, default method, static interface method

→ interface fields are by default considered as public static final

→ Interface methods are by default considered as abstract and it is implicitly considered as public abstract.

```
interface Printable{
    int number = 10;    //By default  public static final

    void print();       //Abstract method
}
```

→ we need to override all the methods in the interface otherwise the class will be abstract

```
interface A{
    void f1( );
    //public abstract void f1( );
}

abstract class B implements A{  //Abstract class

}

class C implements A{   //Concrete class
    @Override
    public void f1() {
        System.out.println("C.F1");
    }
}
```

→ Interface does not have constructor as all fields are static in it.

```
Interface: I1, I2, I3                    Class: C1, C2, C3

 1. I2 implements I1  --> Not OK
 2. I2 extends I1  --> OK: Interface Inheritance
 3. I3 extends I1, I2 --> OK: Multiple Interface Inheritance
 4. I1 extends C1  --> Not OK
 5. C1 extends I1   --> Not OK
 6. C1 implements I1   -->OK: Interface Implementation Inheritance
 7. C1 implements I1, I2 -->OK: Multiple Interface Implementation Inheritance
 8. C2 implements C1  --> Not OK
 9. C2 extends C1  --> OK: Implementation Inheritance
 10. C3 extends C1, C2  --> Not OK: Multiple Implementation Inheritance
 11. C2 implements I1 extends C1 --> Not OK
 12. C2 extends C1 implements I1 --> OK
 13. C2 extends C1 implements I1, I2, I3 --> OK
```

→ Interface does not implement other interface, rather it extends

→ Multiple inheritance in interface is allowed

→ Supertype of interface must be interface

→ Class does not extends interface rather it implements it

→ A class can implement multiple inheritance

→ Class do not implement class, it extends

→ Is multiple inheritance is allowed in java?

Supports Multiple Interface Inheritance

Supports Multiple Interface Implementation Inheritance

→ to avoid ambiguity error, try to access the fields of interface by respective interface only (By doing upcasting with subclasses)

→ in java 8, there's no need to define helper abstract method as default methods were added

→ Ideally it recommended to call the methods by name of interface by upcasting them to the class

```
public class Program {
    public static void main(String[] args) {
        A a = new C();  //Upcasting
        a.f1();

        B b = new C( );
        b.f2();
    }
    public static void main1(String[] args) {
        C c = new C();
        c.f1();
        c.f2();
```

→ Design of interface is so important and need not be changed once it is defined. If changed, all the implementing classes will become abstract

→ Default methods are not abstract and need not to be overridden by subclasses

→ can call default method of super-interface from sub interface using InterfaceName.super.method()

```
interface Collection{   //1996
    void acceptRecord( );
    void printRecord( );
    int[] getArray( );
    default void sort( ) {
        int[] arr = this.getArray();
        for( int i = 0 ;i < arr.length - 1; ++ i ) {
            for( int j = 0; j < arr.length - 1 - i; ++ j ) {
                if( arr[ j ] > arr[ j + 1 ] ) {
                    int temp = arr[ j ];
                    arr[ j ] = arr[ j + 1 ];
                    arr[ j + 1 ] = temp;
                }
            }
        }
    }
}
```

→ If a class implements more than one default methods with same name and parameters, we need to override them compulsorily to remove ambiguity (Compile time error)

→ default methods avoid the need of the helper class

→ Default methods are more used by frameworks and less by developers

→ We cannot override static, private and final methods

→ any helper methods in interface can be defined as static and can be used in interface and subclasses also

→ Any interface which contain single abstract method also known as Single Abstract Method interface/ SAM interface/ Functional interface and the method is known as Functional method / Method descriptor

→ A functional interface can have multiple fields, static methods, default methods

→ annotation: @FunctionalInterface

→ To implement functional int: lambda expressions, Method References, Anonymous Classes.

→ java.util.function package contains all java supplied functional interfaces.

Abstract Class	Interface
may and may not contain constructor	No constructor
must or must not contain abstract methods	Methods are abstract by default
Can have only one supertype	can extend one or more interfaces
provides partial data abstraction	full data abstraction

→ When we should abstract class and when to use interface:

abstract class:

→ If “is-a” relationship is between supertype and subtype and want common method design then supertype must be abstract.

→ Using abstract class, we can group related data elements together

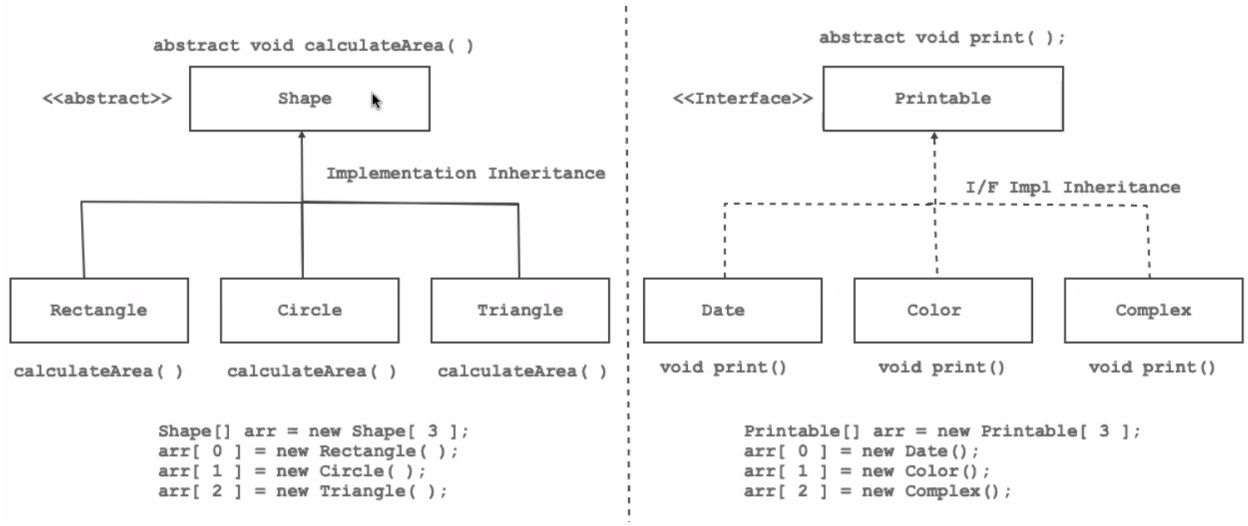
→ If state/field of supertype is involved in all of the subtypes then we use abstract class

Interface:

→ If “is-a” relationship is not there between supertype and subtype and want common method design then supertype must be interface. (can-do relationship)

→ Using interface, we can group instances of unrelated data elements together

→ If state/field of supertype is not involved in subtypes then we use interface



→ Comparable is interface declared in java.lang package, abstract method is compareTo(T o);

→ If we want to sort array of non primitive type then non primitive type should implement comparable interface

→ If we want to sort array of non primitive type in natural order then we should use comparable interface.

→ All wrapper class implements comparable interface.

```
//Employee this: Current Instance
//Employee other: specified Instance
@Override
public int compareTo(Employee other) {
    if( this.empid < other.empid )
        return -1;
    if( this.empid > other.empid)
        return +1;
    return 0;
}
```

→ Logic written in compareTo method is natural ordering

```java
@Override
public int compareTo(Employee other) {
    return this.empid - other.empid;
}
```

```java
@Override
public int compareTo(Employee other) {
    return this.name.compareTo( other.name );
}
```

→ java.util.comparator is also used to sort array of non primitive types

→ int compare(T o1, T o2) is abstract method comparator

→ if you want custom ordering in sort them we should use comparator interface.

//Read about defensive copy

//Effective Java book

//java.util.dateformat

//date and time, systemdate, systemtime, input.

Shallow copy: process of copying content of one variable to other

→ cannot access Date's clone in Program class as it is protected and in different package

→ so, we have to define it as wider modifier in public

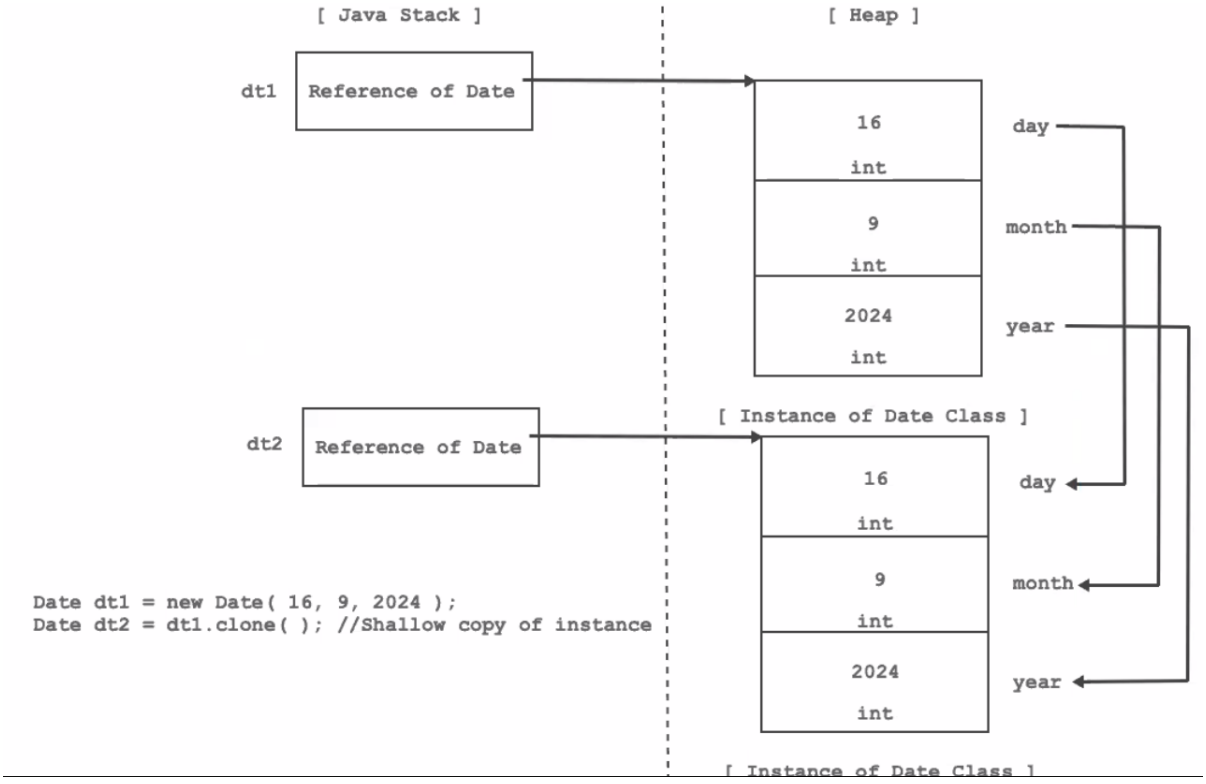
→ clone is a method of object class not clonable interface

→ Clonable is interface in java.lang package

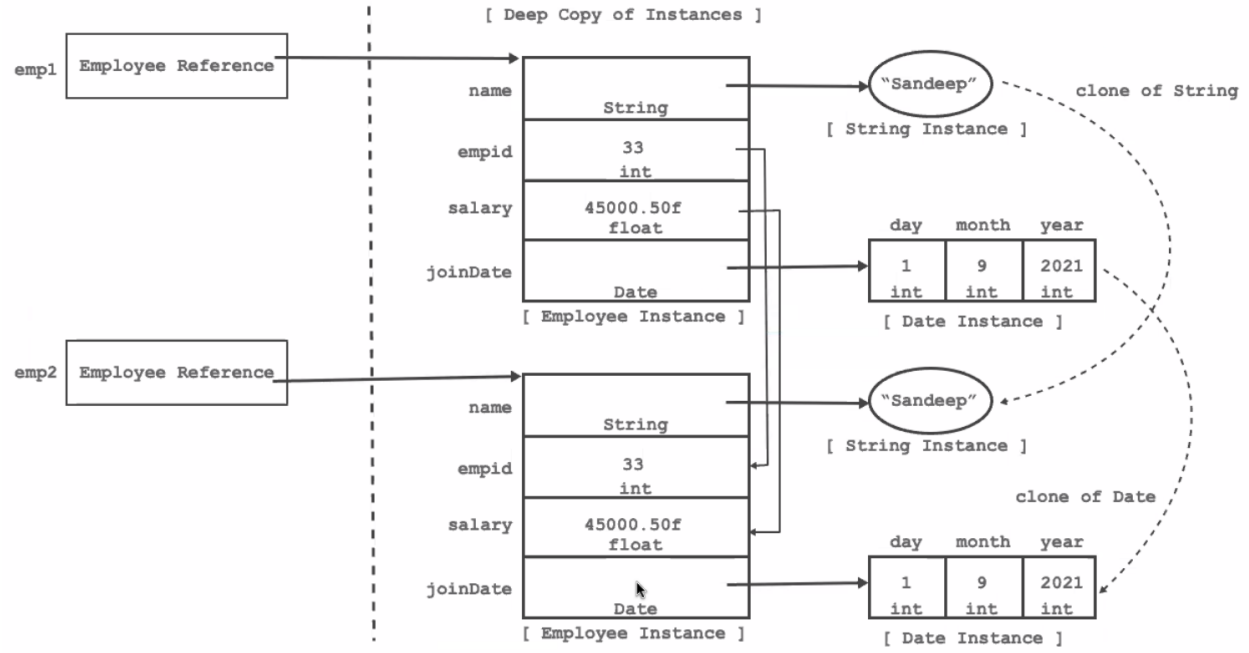
→ Interface without any members known as marker interface/tagging interface eg. java.util.EventListner, java.util.RandomAccess, java.rmi.Remote, java.lang.Clonable

→ Job of annotations and marker interface is to generate metadata. it is for compiler or jvm

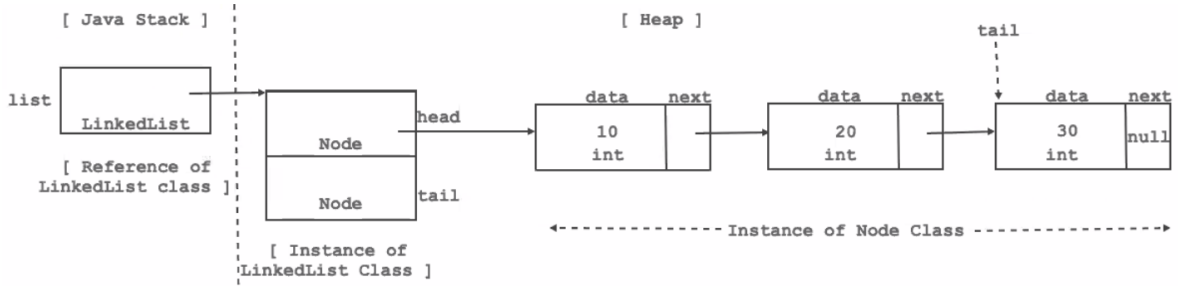
→ without implementing clonable interface if we try to clone a object, it throws clonenotsupported exception



Deep copy of instance:



Linked list implementation:



→ Iterable is interface in java.lang package which is used to iterate over the object

→ abstract method is iterator

Operator:

```
+ Operator
   - Unary Operator
      - ++, – –, ~
   - Binary Operator
      - Arithmetic operator: +, -, *, /, %
      - Relational operator: <, >, <=, >=, ==, !=
      - Logical operator: &&, ||
      - Bitwise operator: &, |, ^, <<, >>
      - Assignment operator: =, short hand operator
   - Ternary Operator
      - Conditional operator( ? : )
```

Arithmetic Expression involves arithmetic operators

Relational expression involves relational operators

An expression which contains Lambda operator (→) is called as lambda expression

→ operator is called as lambda operator whose meaning is goes to.

Syntax:

```
Functional_Interface ref1 = Lambda Expression;

Functional_Interface ref2 = ( LHS )->(RHS)
   - LHS: Input Parameters
   - RHS: Lambda Body


Functional_Interface ref3 = null;
ref3 = ( I/P params)-> Lambda Body;
```

→ in java lambda expression is called as anonymous method

→ To implement functional interface, we can use lambda expression

→ @FunctionalInterface: is a annotation introduced in java8

→ There's no need to specify type in lambda expression

→ There's no need of brackets in single parameter lambda expression

→ if we do not define block for lambda expression, we do not need curly braces'

→ If we define lambda expression, compiler does not generate .class file

→ for lambda expression compiler generates private static method

→ Lambda expression creates synthetic class

→ We can see functional interfaces in java.util.function package

```
+ Functional interfaces from java.util.function package:
   - Predicate<T>: boolean test(T t)
   - Consumer<T>: void accept(T t)
   - Supplier<T>: T get()
   - Function<T,R>: R apply(T t)
```

→ we can pass lambda method body as argument to method, it is called behavior parameterization

→ If existing logic is present in some reference then we can use method reference

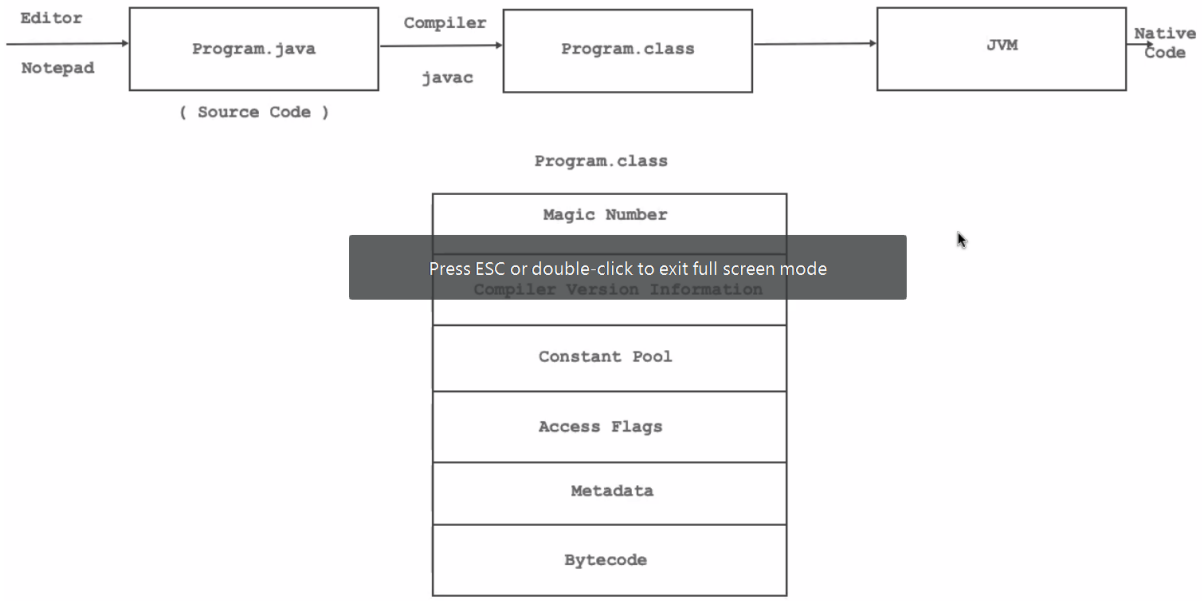
```
interface Addition{
    int addThem(int a, int b);
}

public class HelloWorld{
     public static void main(String []args){
        Addition a = Integer::max;
        //Addition a = (a,b) -> Integer.max(a,b);
        System.out.println(a.addThem(10,20));
     }
}
```

→constructor reference

```
Sample s = Employee::new;
//Sample s = (name, age, salary) -> new Employee(name, age,
```

```
salary);
```

→ .class file contains



→ javap command is java disassembler

→ Magic number: hexdump -c Program.class. We can find the type of file.

→ Compiler version information: It is used to get info about version of compiler

→ Constant Poll: All the literals and constants gets space here

→ Access Flags: All the modifiers

→ Metadata - data which describes other data is metadata. Information about all the classes, methods and fields. (Name, annotations, access modifiers, parameters, etc)

→ Bytecode:

→ Reflection is java language feature which is used to get information of metadata at runtime

→ Instance of java.lang.Class (final class) represents the metadata

→ Class has no public constructor

→ class Class instance contains metadata of loaded classes

→ if n classes gets loaded in jvm memory then n instances are created in class Class

→ Instances of the class Class represent classes and interfaces in a running Java application.

Class is class in package java.lang

it does not contain public constructor so we cannot create it's instance

jvm creates it's instance

we can get it's instance for having metadata

.class syntax:

```
Class<?> c = Number.class;    //abstract class

Date date = new Date();
Class<?> c = date.getClass();   //on instance
Class<?> c = Date.class         //on class
```

Reflection is basically used in spring, springboot and hibernate

/Assignment: to get fields, constructors and methods with parameters by using reflection

→ used in itellisense in ide

→ used in debuggers

→ Using reflection, we can access private fields outside of the class

→ Middlewear application: every information gets managed at runtime

//Assignment try to prepare proxy design pattern

Notes for lambda expressions:

→ Lambda expression is a new and important feature of Java which was included in Java SE 8

→ The Lambda expression is used to provide the implementation of functional interface.

→ Java lambda expression is treated as a function, so compiler does not create .class file.

Ways to implement functional interface:

→ Ways to implement Functional Interface:

→ Method Referance

→ Anonymous Class

→ Lambda Expressions

→ Built in functional interface in java

1. Predicate <T t>: Method: boolean test(arg1). Represents a predicate (boolean-valued function) of one argument.
2. Consumer <T t>: Method: void accept(T t). Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, consumer is expected to operate via side-effects.
3. Supplier<T t>: Method: T get(). This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.
4. Interface Function<T,R>: Method: R apply(T t). Represents a function that accepts one argument and produces a result.

```
import java.util.function.Predicate;
import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.function.Function;
```

```
class Main{
    public static void main(String args[]){
        Function<String, Integer> s = (str) -> Integer.par
seInt(str);
        System.out.println(s.apply("123"));
    }

    public static void main2(String args[]){
        Supplier<String> s = () -> "Hello World";
        System.out.println(s.get());
    }

    public static void main1(String args[]){
        Consumer<String> c = str -> System.out.println(st
r);
        c.accept("Sumant");
    }

    public static void main0(String args[]){
        Predicate<Integer> p = num -> num % 2 !=0 ;

        System.out.println(p.test(10));
    }
```

→ we can pass lambda method body as argument to method, it is called behaviour parameterization