Core Java

Sandeep Kulange

Course Guidelines

- **Batch** : e-DAC September 2020
- **Module Name** : Object Oriented Programming with Java
- **Duration** : 44 classroom hours + 44 lab hours (88 hours)
- **Evaluation Method:**

    1. Theory Exam : 40% weightage
    2. Lab Exam : 40% weightage
    3. Internal Exam : 20% weightage

- **Number of Session** : 22
- Note: Each Session is of 2 hours

Reference Book(s)

1. The Complete Reference Java – Herbert Schildt
2. Head First Java – Kathy Sierra
3. Java 8 Programming Black Book – Dreamtech Press
4. Java, How to program – Paul and Harvey Deitel
5. Core Java Fundamentals, Volume I and II – Cay Horstmann
6. A programmers guide to Java SE 8 – Khalid Mughal
7. Java 8 In Action – Manning Press

Reference Tutorial(s)

1. https://www.artima.com/java/
2. http://tutorials.jenkov.com/
3. https://www.journaldev.com/7153/core-java-tutorial
4. https://www.baeldung.com/java-tutorial
5. https://docs.oracle.com/javase/tutorial/

What is Platform?

- **A *platform* is the hardware or software environment in which a program runs.**
- **Types of platform:**

    1. **Hardware based platforms**
        - Combination of operating system and underlying hardware.
        - Some of the most popular hardware based platforms like Microsoft Windows, Linux, Solaris OS, and Mac OS.
    2. **Software only platform**
        - A platform that runs on top of other hardware-based platforms.
        - Some of the most popular software-only platforms like MS.NET, **Java**.

What is Framework?

- Library may contain interfaces, classes and enum.
- **Libraries of reusable classes /interfaces that is used to develop application is called framework.**
- Consider following examples:

    1. **RMI** : Distributed application development framework.
    2. **AWT/SWING** : GUI application development framework.
    3. **Struts** : MVC based, web application development framework.
    4. **Hibernate** : Automatic persistence framework.
    5. **Junit** : Unit testing framework

What is Language?

- It has syntax.
- It has tokens(keyword, Identifier, constant, operator, punctuator).
- It has data types.
- It has unique built in features. For Ex. pointer, lambda expression etc.
- Example: C,C++, **Java**, C#, Python, GO etc.
- We can use language to develop application but generally it is used to implement business logic.

What is Technology?

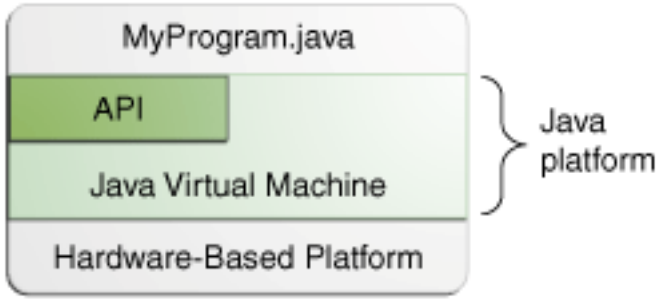
- **Types of application:**
    1. Console User Interface(CUI).
    2. Graphical User Interface(GUI).
    3. Library application.
    4. Web Application
    5. Web Services
- **Technologies helps us to develop an application.**
- Using **java** we can develop standalone application, web application & services, mobile application etc.
- To execute these application, platform is required.

History of the Java programming language

- Java **language** is both **technology** as well as **platform**.
- Java was originally developed by **James Gosling** and his team in 1991 at **Sun Microsystems** and released in 1995.
- Initial name of Java was Oak. But this name was already registered hence later Oak was renamed to Java.
- “Write Once Run Anywhere” (WORA) is a slogan of the Java.
- In 2010, Sun Microsystems was overtaken by Oracle.
- Latest version of Java is **Java 15** which is released in September 2020.
- In 1997, Sun Microsystems approached the ISO/IEC JTC1 standards body and later the ECMA International to formalize Java, but it soon withdrew from the process. Java remains a de facto standard, controlled through the Java Community Process(JCP).

Java SE Platform

- Java SE platform is also called as Core Java.
- The Java platform has two components:
    1. **The Java Application Programming Interface(API).**
    2. **The Java Virtual Machine.**



- Java API = All interfaces, classes and enums which are part of Java development kit.
- The Java SE API's are sub set of Java EE API's.

Java Platform's

- Sun has defined and supports four editions of Java targeting different application environments. The platforms are:

1. **Java Platform, Standard Edition (Java SE)**
    - Also called as Core Java
    - Java SE API's are subset of Java EE API
2. **Java Platform, Enterprise Edition (Java EE)**
    - Now it is Jakarta EE.
    - It is also called as JEE/Enterprise Java/Web Java/Advanced Java.
3. **Java Platform, Micro Edition (Java ME)**
    - For embeded and mobile devices.
4. **Java Card**
    - For smart Cards.

Java SE Naming and Versions

- The Java Platform name has changed a few times over the years.
- Java was first released in January 1996 and was named Java Development Kit, abbreviated JDK.
- Version 1.2 was a large change in the platform and was therefore rebranded as Java 2. Full name: Java 2 Software Development Kit, abbreviated to Java 2 SDK or **J2SDK**.
- Version 1.5 was released in 2004 as J2SDK 5.0 –dropping the “1.” from the official name and was further renamed in 2006. Sun simplified the platform name to better reflect the level of maturity, stability, scalability, and security built into the Java platform. Sun dropped the "2" from the name. The development kit reverted back to the name "JDK" from "Java 2 SDK". The runtime environment has reverted back to "JRE" from "J2RE.”
- JDK 6 and above no longer use the “dot number” at the end of the platform version.

Java SE Naming and Versions

Version	Date
JDK Beta	1995
JDK1.0	January 23, 1996 [40]
JDK 1.1	February 19, 1997
J2SE 1.2	December 8, 1998
J2SE 1.3	May 8, 2000
J2SE 1.4	February 6, 2002
J2SE 5.0	September 30, 2004
Java SE 6	December 11, 2006
Java SE 7	July 28, 2011
Java SE 8	March 18, 2014
Java SE 9	September 21, 2017
Java SE 10	March 20, 2018
Java SE 11	September 25, 2018 [41]
Java SE 12	March 19, 2019
Java SE 13	September 17, 2019
Java SE 14	March 17, 2020
Java SE 15	September 15, 2020

What is SDK, JDK and JRE?

- Software Development Kit(SDK).
- **SDK = Development tools + Documentation + Supporting libraries + Runtime Env.**
- JDK = Java's SDK.
- **JDK = Java Development tools + Java API Docs + rt.jar + JVM;**
- JRE = [rt.jar + JVM];
- **JDK = Java Development tools + Java API Docs + JRE;**
- For Java 8 API Link is : https://docs.oracle.com/javase/8/docs/api/
- For language tools : https://docs.oracle.com/javase/7/docs/technotes/tools/

JDK’s Installation Directory Path

- Microsoft Windows Operating System

  C:\Program Files\Java\jdk1.8.0

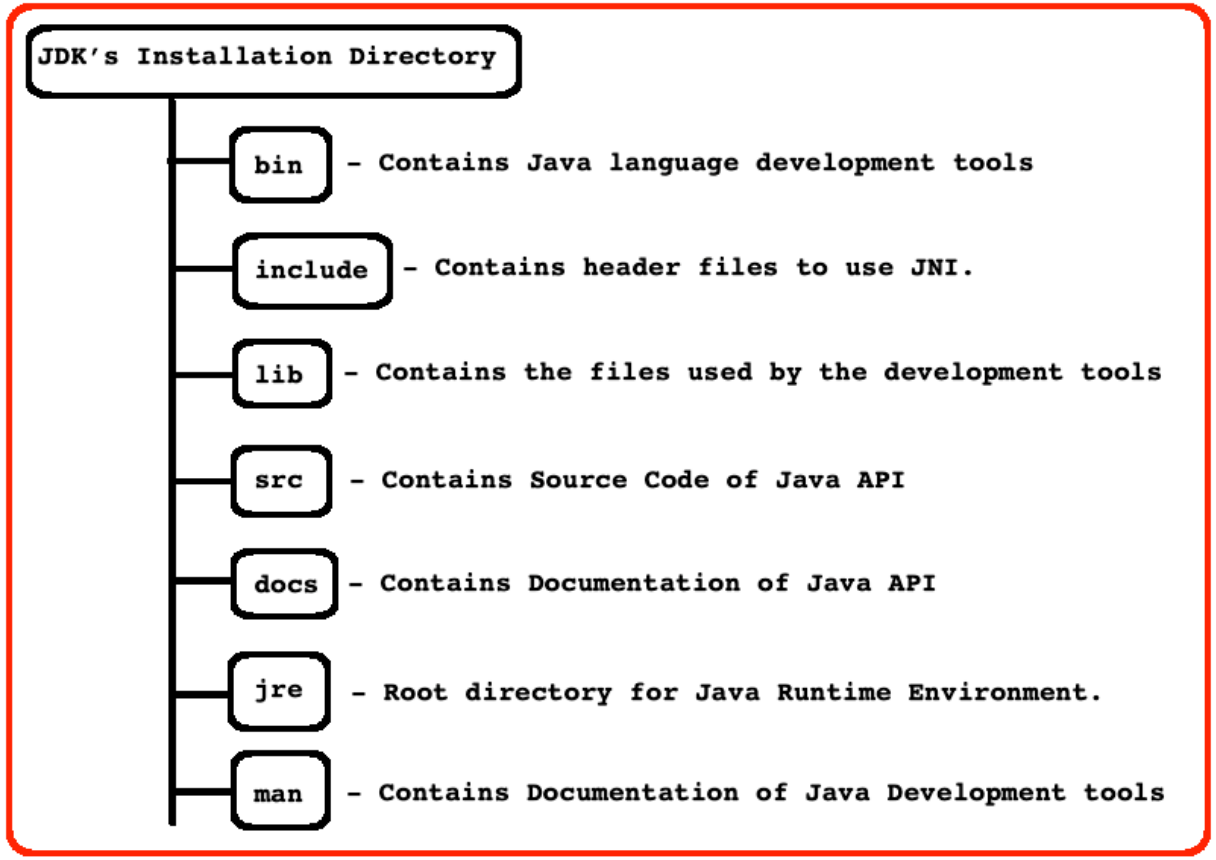
- Ubuntu

  /usr/lib/jvm/java-8-openjdk-amd64

- Mac OS:

  /Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin

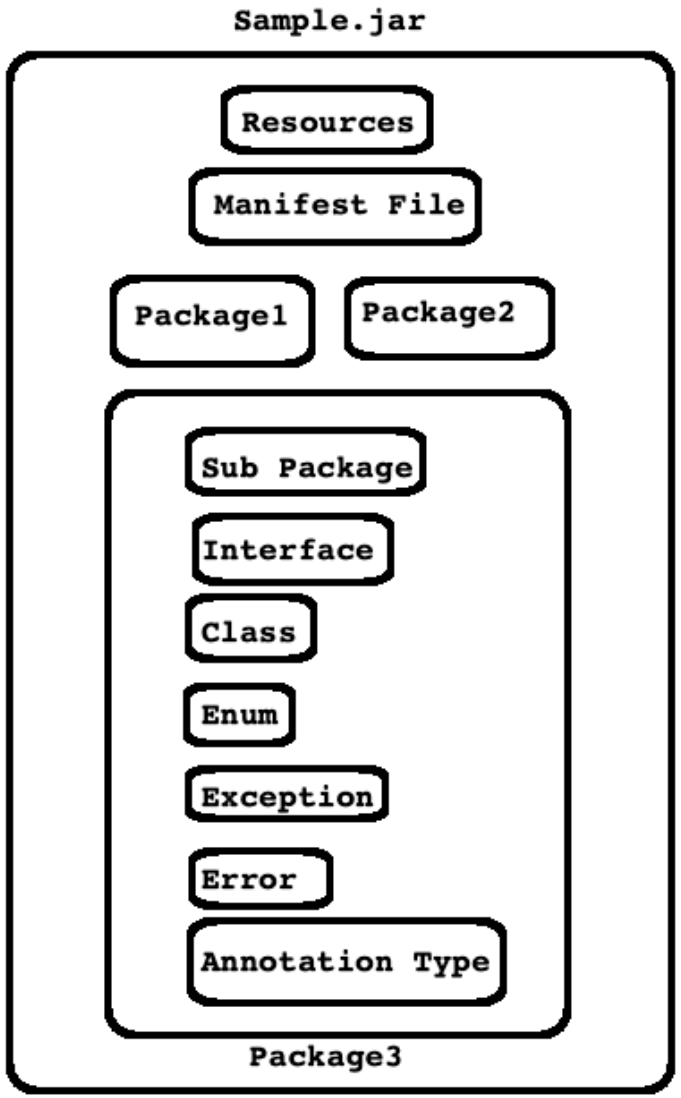
JDK's Installation Directory Structure



- For the JDK and JRE file structure click on following Link:

  https://docs.oracle.com/javase/8/docs/technotes/tools/windows/jdkfiles.html

Structure of Java Archive

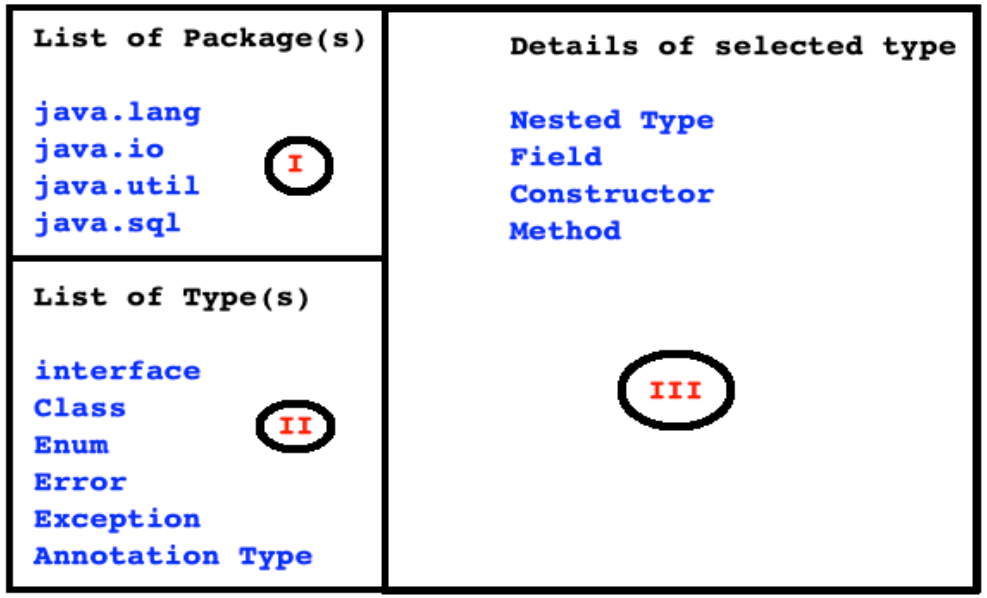


- Java Interface contains:
  1. Nested Interface
  2. Field
  3. Abstract Method
  4. Default Method
  5. Static Interface Method

- Java class contains:
  1. Nested Type
  2. Field
  3. Constructor
  4. Method

Java Documentation

- src.zip file contains source code of Java API.
- Java docs contains documentation of Java API.
- rt.jar file contains compiled code of Java API. It is required for JVM.
- In web browser, you can see help in following format:



“rt.jar” and packages

Main Packages of rt.jar	Sub Packages of java package.		
com	applet	awt	beans
java	io	lang	math
javax	net	nio	rmi
jdk	security	sql	text
sun	time	util	

Some Java Terminologies

- A class from which we can create object is called concrete class. In other words, we can instantiate concrete class.
- A method of a class which is having a body is called concrete method. It can be static/non static.
    1. Non static methods are designed to call on object.
    2. static methods are designed to call on class name.
- A class from which we can not create object is called abstract class. In other words, we can not instantiate abstract class.
- A method of a class which don’t have a body is called abstract method.
- Parent class is called super class and child class is called sub class.
- A class that we can not extend or we can not create its sub class is called final class.

Some Java Terminologies

Sr.No.	C++	Java
1	Data Member	Field
2	Member Function	Method
3	Class	Class
4	Object	Instance
5	Pointer	Reference
6	Access Specifier	Access Modifier
7	Base class	Super Class
8	Derived Class	Sub Class
9	Derived From	Extended from

Simple Hello Application

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

```
export PATH=/usr/bin
javac Program.java
java Program => Output : Program.class
```

Day2

- If .java file do not contain any type(interface/class/enum) then java compiler do not generate .class file.
- Java Compiler generates .class file per type defined in .java file.
- Name of the file and class name can be difrrent. But generally name of the class and name of the file should be same.

System.out.println

- java.lang package contains all the fundamental classes of core java.
- java.lang package is by default imported in every .java file.
- System is a final class declared in java.lang package.
- Variable declared inside function(method) is called local variable(Method Local Variable).
- variable declared inside class is called data member.
- In Java data member is called as field.
- Fields declared inside System class:
    1. public static final InputStream in; //ref
    2. public static final PrintStream out;//ref
    3. public static final PrintStream err;//ref

```
package java.lang;
public final class System extends Object{
    //Fields
    public static final InputStream in; //ref
    public static final PrintStream out;//ref
    public static final PrintStream err;//ref
}
```

- How to access fields of System class?

1. System.in; //represents keyboard
2. System.out; //represents monitor
3. System.err; //Error Stream - represents monitor

- stdin, stdout, stderr are standard stream objects of C language.

```
printf("Hello World");
fprintf(stdout,"Hello World");

int number;
scanf("%d", &number );
fscanf(stdin, "%d", &number );

if( num2 == 0 )
    fprintf(stderr,"/ by zero");
```

```
else
{
    int result = num1 / num2;
    fprintf(stdout,"Result  :   %d\n", result);
}
```

- System.in, System.out and System.err are standard stream objects of Java.
- PrintStream is a class declared in java.io package
- print, printf and println are non static, overloaded methods of java.io.PrintStream class.

Entry Point Method

- In Java, function is also called as method.
- Set or rules and guidelines are called as specification/standard.
- In Java Specification = Abstract Classes + Interfaces.
- Java Language Specification(PDF:Oracle->JCP)
  - It is a specification for Java Language.
- Java Virtual Machine Specification(PDF:Oracle->JCP)
  - It is a specification for JVM implementation.
- According to JVM specification, "main" method must be entry point method.
- Syntax: public static void main(String[] args);
- We can overload main method in java.
- We can define main method per class but only main can be considered as entry point method in Java.
- If sutaible main method is not available inside class then compiler do not generate error but JVM generates error.

Data Type

- Data type of any variable/instance describes 4 things:
  1. Memory : How much memory is required to store the data.
  2. Nature : Which type of data is allowed to store inside memory
  3. Operation : Which operations(functionality) are allowed to perform on data.
  4. Range : Set of values which are allowed to store inside memory.
- Types of data type in Java:
  1. Primitive Data Types
  2. Non Primitive Data Types

Primitive Data Types

- It is also called as Value Type.
- There are 8 primitive/value types in Java. Sr.No Primitive Type Size Field's Default value Wrapper Class

1.
```
 boolean         :   Not Mentioned   :       false           :
Boolean
```

2.
```
 byte          :   1 byte          :       0              :   Byte
```

3.
```
 char          :   2 byte          :        \u0000        :
Character
```

4.
```
 short         :   2 bytes         :       0              :   Short
```

5.
```
 int           :   4 bytes         :       0              :
Integer
```

6.
```
 float         :   4 bytes         :       0.0f           :   Float
```

7.
```
 double        :   8 bytes         :       0.0d           :
Double
```

8.
```
 long          :   8 bytes         :       0L             :   Long
```

- Default value of field of primitive type is generally 0.
- In Java, primitive types are not classes. But for every pritive type class is given it is called Wrapper class.
- All the Wrapper classes are declared in java.lang Package.
- Variable of primitive/value type get space on Java Stack.

```
class Program{
    int num1;   //Field
    public static void main(String[] args) {
        int num2;   //Method Local Variable
    }
}
```

Non Primitive Data Types

- It is also called as Reference Type.
- There are 4 non primitive/reference types in Java.

1. Interface
2. Class
3. Enum
4. Array

- Instance of non primitive/reference type get space on Heap.

Components of JVM:

1. Class Loader Sub System
    - Classloader is responsible for loading .class file from HDD into JVM memory.
    - Types of class loader:
        1. Bootstrap Classloader
        2. Extension Classloader
        3. Application Classloader
2. Runtime Data Areas
    1. Method Area
    2. Heap
    3. Java Stack
    4. PC Register
    5. Native Method Stack
3. Execution Engine
    1. Interpreter
    2. Just In Time (JIT) Compiler

Dynamically Type Checked Language(s)

- Consider example in python

```
number = 10 #OK : Python
number = "DAC" #OK : Python
```

- In python type of variable can be decided by looking toward value

Statically Type Checked Language(s)

- Consider example in Java

```
number = 10; //Not OK
int number = 10;    //OK
```

- Java compiler do not decide type of variable by looking toward value. Rather if we want to use any variable it is mandatory to mention it type.

```
class Program{
    public static void main( String[] args ){
        int number; //OK
        System.out.println("Number  :   "+number); //error: variable
number might not have been initialized
    }
}
```

- We can not use any(primitive/non primitive) type of local variable w/o storing value inside it.

```
class Program{
    public static void main(String[] args) {
        int num1 = 10;  //OK
        int num2;   //OK
        num2 = num1;    //OK
        System.out.println("Num1  :   "+num1); //10
        System.out.println("Num2  :   "+num2); //20
    }
}
```

Initialization

```
int num1 = 10;      //Initialization
int num2 = num1;    //Initialization
```

- Initialization is a process of storing user defined value inside variable during its declaration.
- We can initialize any variable only once.

Assignment

```
int num1 = 10;      //Initialization
int num2;
num2 = num1;    //Assignment
num2 = 20;       //Assignment
```

- Assignment is a process of storing user defined value inside variable after its declaration.
- Assignment can be done multiple times.

Widening

```
class Program{
    public static void main(String[] args) {
        int num1 = 10;  //Initialization
```

```
        double num2 = (double)num1;  //Widening : OK
        double num3 = num1;  //Widening : OK
    }
}
```

- Widening is a process of converting value of variable of narrower type into wider type.
- In case of Widening explicit type casting is optional.

Narrowing

```
class Program{
    public static void main(String[] args) {
        double num1 = 10.5;  //Initialization
        int num2 = ( int )num1; //Narrowing : OK
        int num3 = num1; //Narrowing : NOT OK
    }
}
```

- Narrowing is a process of converting value of variable of wider type into narrower type.
- In case of Narrowing explicit type casting is mandatory.

Boxing

- It is the process of converting state of variable of value type into reference type.
- Example

```
int num1 = 10;
String str = Integer.toString( num1 );  //Boxing
```

```
int num1 = 10;
String str = String.valueOf( num1 );  //Boxing
```

UnBoxing

- It is the process of converting state of instance of reference type into value type.
- Example

```
String str = "125";
String str = Integer.parseInt(str );  //UnBoxing
```

Day3

Parameter versus Argument

```
int sum( int a, int b ) //a, b => Function Parameters / Parameter
{
    int result = 0;
    result = a + b;
    return result;
}
int main( void )
{
    int x = 10;
    int y = 20;
    //int result = sum( 10, 20 ); //10, 20 => Function Argument / Argument
    int result = sum( x, y );//x, y => Function Argument / Argument
    printf("Result  :   %d\n", result);
    return 0;
}
```

Command Line Argument

- Consider code in C language

```
./Main.exe Sandeep 33 45000.50    //Windows
./Main.out Sandeep 33 45000.50    //Linux
//"./Main.out" "Sandeep" "33" "45000.50"
```

```
/*
* argc :
    - argument counter.
    - Function Parameter
    - It keeps counter of arguments including file name
* argv :
    - argument vector.
    - Function Parameter
    - It is array of character pointer which stores address of string.
*/
int main( int argc, char *argv[ ] )
{
    //argv[ 0 ] => "./Main.out"

    //argv[ 1 ] => "Sandeep";
    char *name = argv[ 1 ];

    //argv[ 2 ] => "33"
```

```
    int empid = atoi( argv[ 2 ] );

    //argv[ 3 ] => "45000.50"
    float salary = atof( argv[ 3 ] );

    return 0;
}
```

- Consider command line argument in Java

```
javac Program.java  => Program.class
java Program Sandeep 33 45000.50f
//java Program "Sandeep" "33" "45000.50f"
```

```
class Program{
    public static void main(String[] args) {
        //args[ 0 ]   =>  "Sandeep"
        String name = args[ 0 ];

        //args[ 1 ]   =>  "33"
        int empid = Integer.parseInt( args[ 1 ] );

        //args[ 2 ]   =>  "45000.50f"
        float salary = Float.parseFloat( args[ 2 ]);
    }
}
```

Java Buzzwords / Features

1. Simple
2. Object Oriented
3. Architecture Neutral
4. Portable
5. Robust
6. Multithreaded
7. Secure
8. Dynamic
9. High Performance
10. Distributed

Java is a Simple programming language.

- C language invented during 1969-1972.
- C++ language invented in 1979.
- Java language invented in 1991.

- Java language is derived from C and C++. In others words, Java follows syntax of C and Concepts of C++.
- Syntax of Java is simpler than syntax of C and C++.
    1. No need to include header file.
    2. Do not support structure and union. But it supports enum.
    3. Do not support default argument.
    4. Do not support constructure member initializer list.
    5. Do not support delete operator and destrctor.
    6. Do not support friend function and friend class.
    7. Do not support copy constructor and operator overloading.
    8. Do not support private and protected mode of inheritance.
    9. Do not support multi-class inheritance in other words, It doesn't support multiple implementation inheritance.
    10. We can not delcare/define global variable and function.
    11. Do not support pointer.

Java is a Object Oriented programming language.

- Alan Kay -> Inventor of OOPS and Simula.
- Grady Booch : Inventor of UML : Author of : Object Oriented Analysis and Design With Application.
- According to Grady Booch there 4 major and 3 minor pillars/parts/elements of oops.
- 4 major pillars of oops
    1. Abstraction
    2. Encapsulation
    3. Modularity
    4. Hierarchy
- According to Grady Booch, if we want to consider any language OO then it must support 4 major pillars of OOPS.
- 3 minor pillars of oops
    1. Typing / Polymosphism
    2. Concurrency
    3. Persistence
- If language support to above features then it will be considered as useful but not essential to classify language OO.
- Since Java support to all major and minor pillars of oops hence it is conisidered as object oriented.

Java is Architecture Neutral

- CPU Architectures : X86, X64, ARM, POWER PC, SPARK, APLHA etc.
- Java compiler convert java source code into bytecode.
- Native CPU can not execute bytecode code directly.

- Execution engine of JVM converts bytecode into native code.
- .class file contains bytecode which is CPU neutral code makes Java architecture neutral.
- Since Java is architecture neutral, java developer need not to worry about underlying hardware and operating system.

Java is Portable programming language

- Term Portable is related to executable.
- Java is portable because Java is architecture neutral.
- Size of data types on all the platform is constant/same.
    1. boolean : Not Mentioned
    2. byte : 1 byte
    3. char : 2 bytes
    4. short : 2 bytes
    5. int : 4 bytes
    6. float : 4 bytes
    7. double : 8 bytes
    8. long : 8 bytes
- Since Java is portable, It doesn't support sizeof operator.

Java is Robust programming language.

- Java is robust programming language because of 4 reasons:
    1. Java is architecture neutral.
    2. Java is object oriented programming language.
    3. Java's memory management.
    4. Java's Exception Handling.

Java is Multithreaded Programming language.

- JVM is responsible for managing execution of Java application.
- Thread : Light weight process / sub process is called thread.
- When JVM starts execution of Java application then it also starts execition of 2 thread i.e main thread and garbage collector.
- Because of main thread and garbage collector, every java application is Multithreaded.
- Main Thread
    1. It is a user thread / non daemon thread.
    2. It is responsible for invoking main method.
- Garbage Collector / Finalizer
    1. It is a daemon thread / background thread.
    2. It is responsible for deallocating memory of unused objects.

Scanner Demo : For Input

- Scanner is a final class declared in java.util package.
- Instantiation
    - Process of creating instance from class is called Instantiation.

```
java.util.Scanner sc = new java.util.Scanner(System.in);
```

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
```

- Methods:
    1. public String nextLine()
    2. public int nextInt()
    3. public float nextFloat()
    4. public double nextDouble()
- If we want to call non static method then it is ncessary to use object reference.

Eclipse Introduction

print, println,printf

- print method print o/p on console and keep cursor on same line.
- println method print o/p on console and move cursor to the next line.
- printf is used to print formated output on console.

```
String nm1 = "Prashant Lad";
int id1 = 1234;
float sal1 = 25000.50f;
System.out.printf("%-15s%-5d%-10.2f\n", nm1, id1, sal1 );
```

OOPS Concepts

- Consider following examples:

1. Let us consider Date:
    - "24/11/2020"
    - day, month, year => int
    - day, month and year are of type int which a related to int.

```
class Date{
    int day;
    int month;
    int year;
}
```

2. Let us consider Color
    - Color value is represented using RGB.
    - red, green, blue => int

- red, green, blue are of type int which a related to Color.

```
class Color{
    int red;
    int green;
    int blue;
}
```

3. Let us consider Account
    - number : int
    - type : String
    - balance : float
    - number, type, balance are realated to Account.

```
class Account{
    int number;
    String type;
    float balance;
}
```

4. Let us consider Employee
    - name : String
    - empid : int
    - salary : float
    - name, empid and salary are related to Employee.

```
class Employee{
    String name;
    int empid;
    float salary;
}
```

- If we want to group related data elements together then we should use Class.
- class is a keyword in Java.
- If we want to define class then first it is necessary to understand problem statement.
- A variable declared inside class is caled field.
- If we want to store value inside non static field then we must create object of the class.
- In Java, object is called as instance.
- class is non primitive / reference type.
- If we want to create instance of a class then we should use new operator.
- Java instance get space on Heap.
- Non static field get space once per instance according order of their declaration inside class.
- In java all the instances are anonymous.

- If we want to perform operations on instance then it is necessary to create object reference / reference.
- Instance w/o reference is called anonoymous instance.
- If we want to give controlled access to the field then we should declare field private and give access to it using method.
- If we want to perform operations on instance then we should define method inside class.

Object Oriented Programming With Java

Day3 Agenda

- Class
- Steps for write code for class
- Reference and Instance
- this reference
- Constructor
- Constructor Chaining
- What is null
- Value Type Versus Reference Type.

Major and minor pillars of oops

- 4 Major pillars of oops:
  1. Abstraction
  2. Encapsulation
  3. Modularity
  4. Hierarchy

- Must be required in object oriented programming language.

- 3 minor pillars of oops
  1. Typing/Polymorphism
  2. Concurrency
  3. Persistence

- Useful but not required in object oriented programming language.

Class

- If we want to group functionally equivalent / related elements together then we should use class
- Consider Examples
  - Date
    - Related data elements are day, month and year
  - Address
    - Related data elements are cityname, statename and pincode
  - Color
    - Related data elements are red, green and blue
  - Student
    - Related data elements are name, rollnumber and marks
  - Department
    - Related data elements are id, name and location.
  - Employee
    - Related data elements are name, empid and salary

Class

- class is a keyword in Java which can contain:
  1. Nested types(interface, class, enum)
  2. Field
  3. Constructor
  4. Method

- We can define class inside class it is called nested class.

- Variable declared inside class is called as field.

- Function implemented inside class is called method.

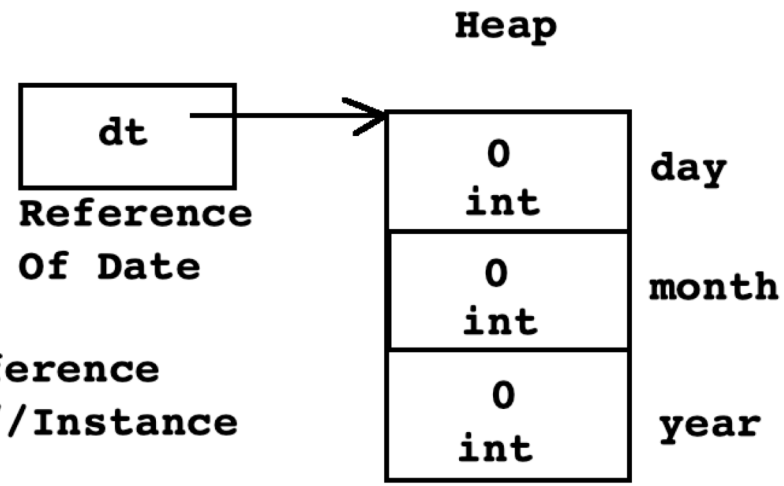
- Constructor is used to initialize instance.

Consider steps while writing a class

1. Understand assignment/problem statement and decide classes and its fields.
2. Define class and declare fields inside it.
3. Create Instance of class.
   - To create instance of a class it is mandatory to use new operator.
   - Employee emp = new Employee( );
   - If we create instance of a class then fields get space inside it.
4. To process(accept/print/set/get) state/value of instance define and invoke method on it.
   - Process of calling method on instance is called message passing.
5. Use this reference inside method to process state of instance.

Reference and Instance

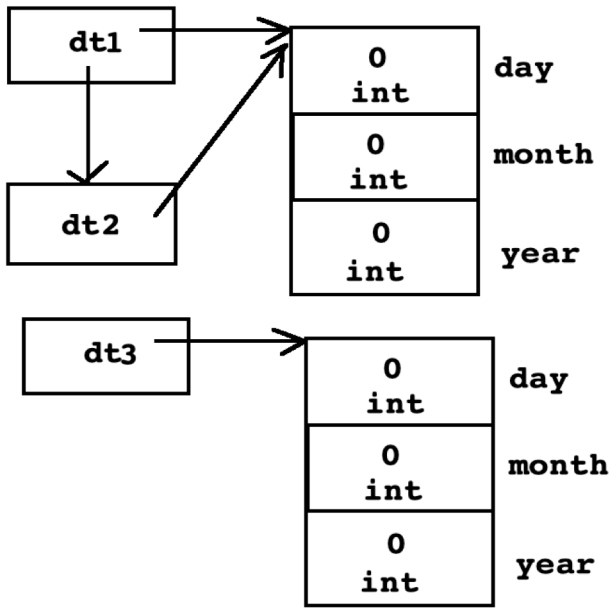
+ non static field gets space once per instance according to order of their declaration inside class.



+ If we want to process state of instance(call method on instance) then we should create reference of a class.

Reference and Instance

```
Date dt1 = new Date( );
Date dt2 = dt1;
Date dt3 = new Date( );
```

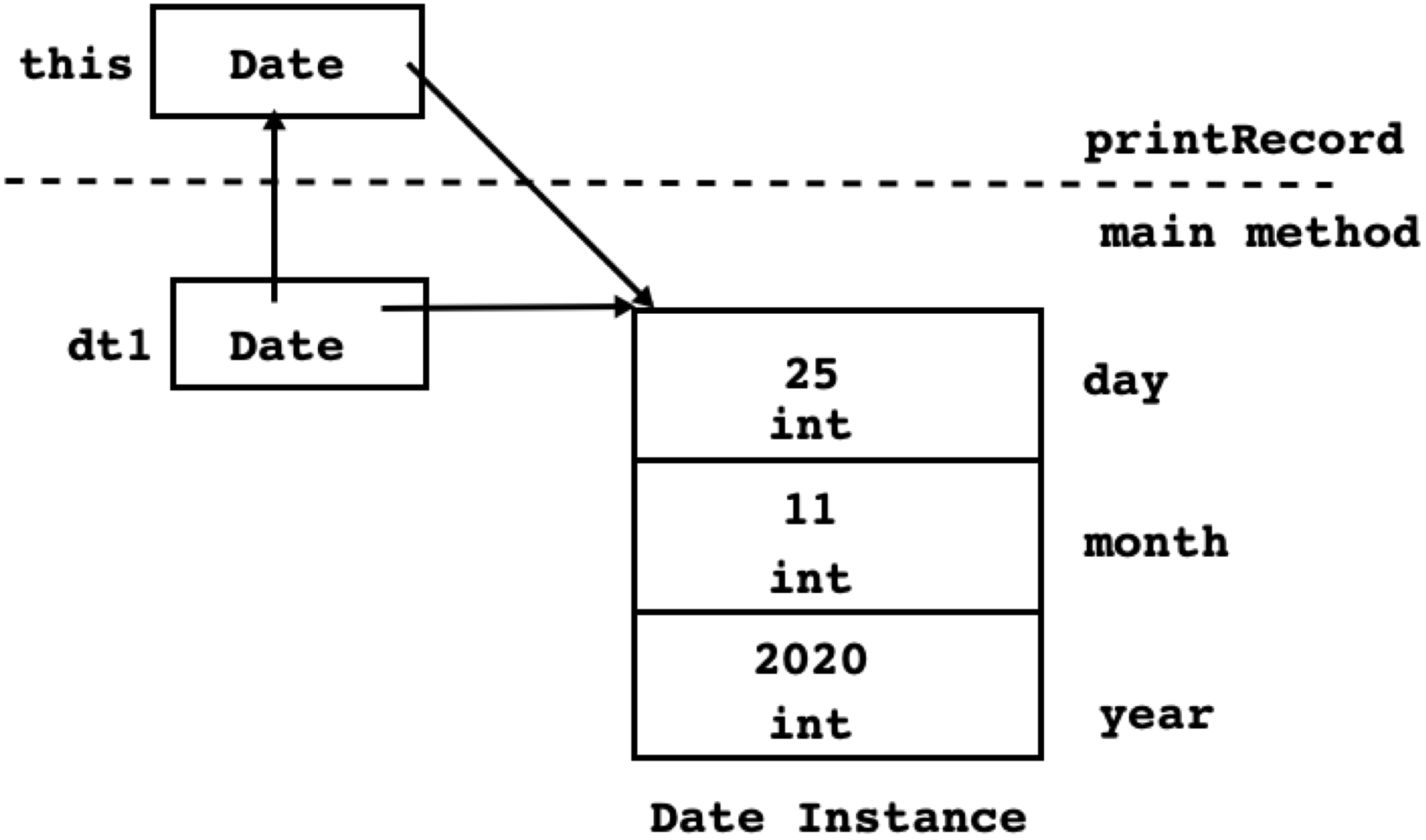


this reference

```
class Date{
	private int day, month, year;
	public void printRecord( /*Date this*/ ){
		//TODO : use this to access state of Date instance.
	}
}
class Program{
	public static void main( String[] args ){
		Date dt1 = new Date( );
		dt1.printRecord( );		// dt1.printRecord( dt1 );

		Date dt2 = new Date( );
		dt2.printRecord( );		// dt2.printRecord( dt2 );
	}
}
```

this reference



this reference

- To process value or state of instance, we should call method on instance.
- If we call non static method on instance then compiler implicitly pass reference of current instance as a argument. And to store value of the argument compiler implicitly declare one parameter inside method. It is called this reference.
- this is a keyword in Java.
- Using this reference, non static field and non static method can communicate with each other. Hence it is also called as link /connection between non static field and non static method.
- We can not declare this reference explicitly. It is implicit parameter available in every method of a class. It is considered as first parameter of a method.

- In simple words, It is a implicit reference variable which is available in every non static method of a class which is used to store reference of current or calling instance.

- If name of local variable and name of field is same then we should use this before field.

Constructor

- If we want to initialize instance then we should use constructor.
- Constructors are special because:
  1. Its name is same as class name.
  2. It doesn't have any return type.
  3. It is designed to call implicitly.
  4. It gets called once per instance.

- Types of constructor
  1. Parameterless constructor
  2. Parameterized constructor
  3. Default constructor

Parameterless Constructor

- A constructor which do not take any parameter is called parameterless constructor.
- Consider Example:

```
class Date{
    private int day, month, year;
    public Date( ){      //Parameterless constructor
           this.day = 25;
           this.month = 11;
           this.year = 2020;
    }
}
```

- If we create instance without passing argument then parameterless constructor gets called.
- Consider Example:

```
Date dt = new Date( ); //Here parameterless constructor will call
```

Parameterized Constructor

- A constructor which take parameter(s) is called parameterized constructor.
- Consider Example:

```
class Date{
    private int day, month, year;
    public Date( int day, int month, int year ){   //Parameterized constructor
            this.day = day;
            this.month = month;
            this.year = year;
    }
}
```

- If we create instance by passing argument then parameterized constructor gets called.
- Consider Example:

```
Date dt = new Date( 25,11,2020); //Here parameterized constructor will call
```

Default Constructor

- If we do not define constructor(no constructor) inside a class then compiler generates once constructor for the class by default it is called default constructor.
- The default constructor is zero argument constructor.

- Consider Example:

```
class Date{
    private int day, month, year;
    public void print( ){
    }
}
```

- Instantiation

```
Date dt1 = new Date( ); //Here Default constructor will call
Date dt2 = new Date( 25, 11, 2020 ); //Compiler Error
```

Constructor Chaining

- If class contains set of constructors then constructor can reuse logic of another constructor.
- If we want to reuse implementation of existing constructor inside another constructor then we should call constructor explicitly. It is called constructor chaining.
- For constructor chaining we should use this statement.
- Consider Example:

```
class Date{
    private int day, month, year;
    public Date( ){
            this(25,11,2020);    //Constructor Chaining
    }
    public Date( int day, int month, int year ){
            this.day = day;
            this.month = month;
            this.year = year;
    }
}
```

- “this” statement must be first statement inside constructor body.

Literals

- Constant is also called as literal.
- Consider examples:
  1. true : Use boolean to store value
  2. ‘A’ : Use char to store value
  3. 123 : Use byte/short/int/long to store value
  4. 3.14f : Use float to store value
  5. 3.142 : Use double to store value
  6. “SunBeam” : Use String to store value
  7. null : Use reference to store value.
- NULL is macro in C which is used to initialize pointer.
- Example : int *ptr = NULL;
- null is literal in Java which is used to initialize reference variable.
- Example : Employee emp = null;

Value Type versus Reference Type

Value Type

1. Primitive type is also called as value type.
2. There are 8 values types in Java.
3. Variable of value type get space on Java Stack.
4. Variable of value type contain value.
5. In case of copy, value gets copied.
6. Variable of value type do not contain null.
7. We can not use new operator to create instance of value type.

Reference Type

1. Non primitive type is also called reference type.
2. There are 4 reference types in Java.
3. Instance of reference type get space on Heap.
4. Variable of reference contain reference.
5. In case of reference, reference gets copied.
6. Variable of reference type can contain null.
7. It is mandatory to use new operator to create instance of reference type.

Thank you

Java Technologies-I (Core Java)

JDBC

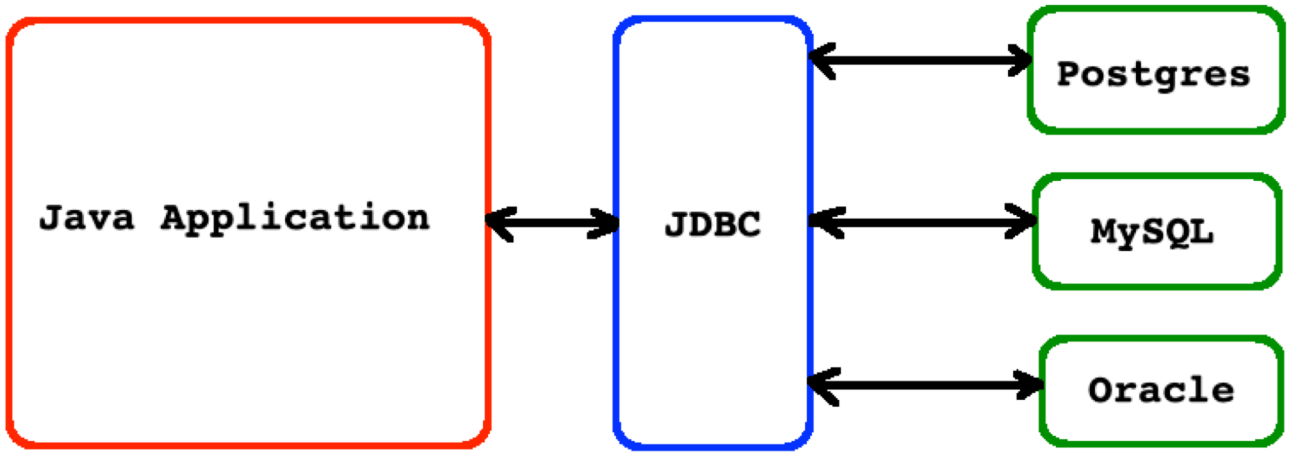
Java DataBase Connectivity

- Specification = { Abstract Class(es) + Interface(s) }
- JDBC is a specification:
  - Vendor : **SUN/ORACLE (java.sql package)**
  - Implementation : **Database vendors (Database Connector)**
  - Client : **Java Application Developers (Include connector in build path)**
- JDBC API version:

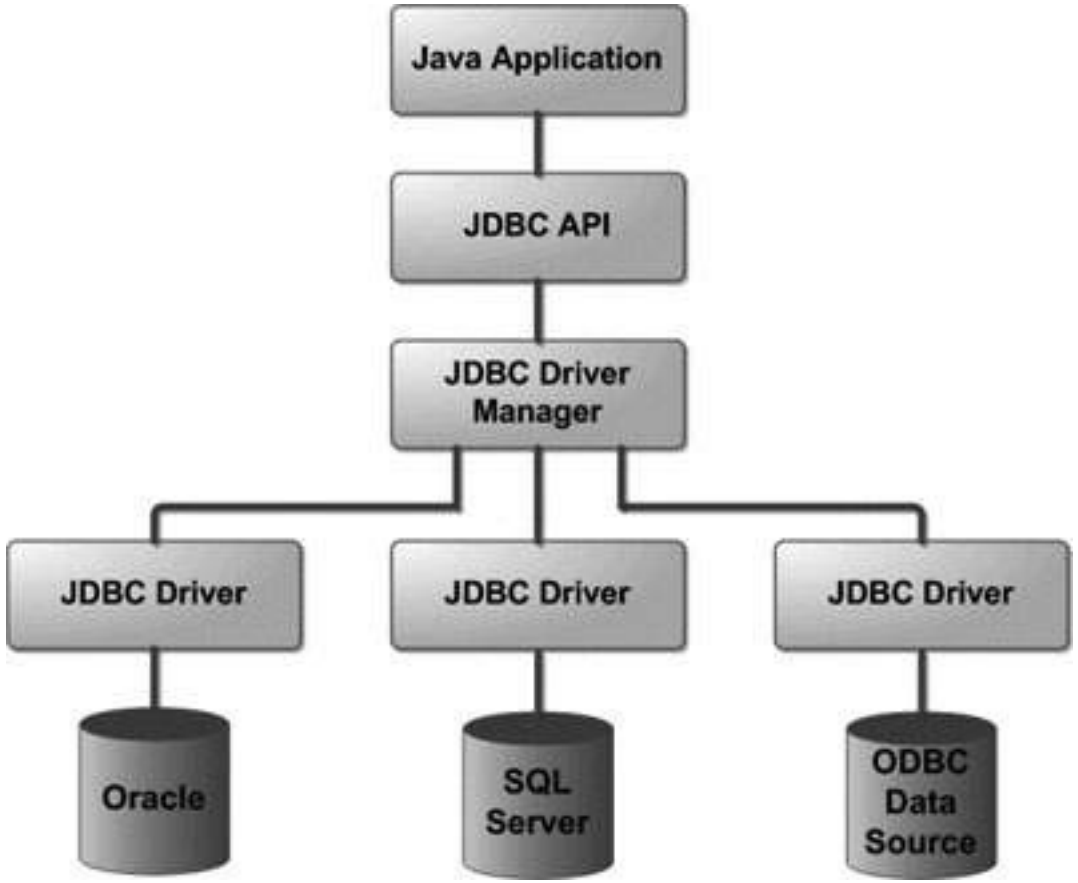
1	JDK 1.1	No Tag
2	J2SE 1.2	JDBC 2.0 API
3	J2SE 1.4	JDBC 3.0 API
4	Java SE 6	JDBC 4.0 API
5	Java SE 7	JDBC 4.1 API
6	**Java SE 8**	**JDBC 4.2 API**

Why JDBC?

- If we want to access and process data stored in relational database management system using Java programming language then we should use JDBC.
- It minimizes database vendor dependency in the Java application.

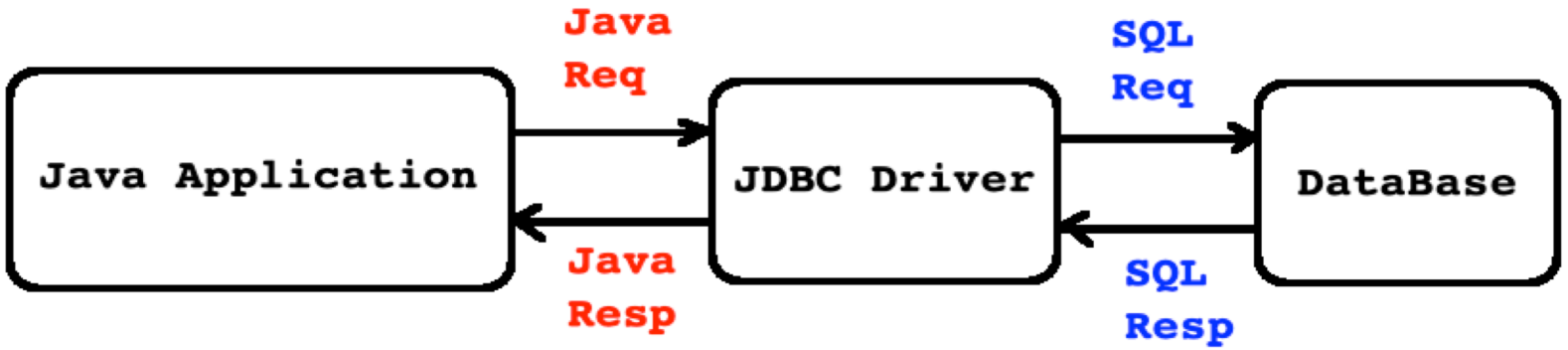


JDBC Architecture



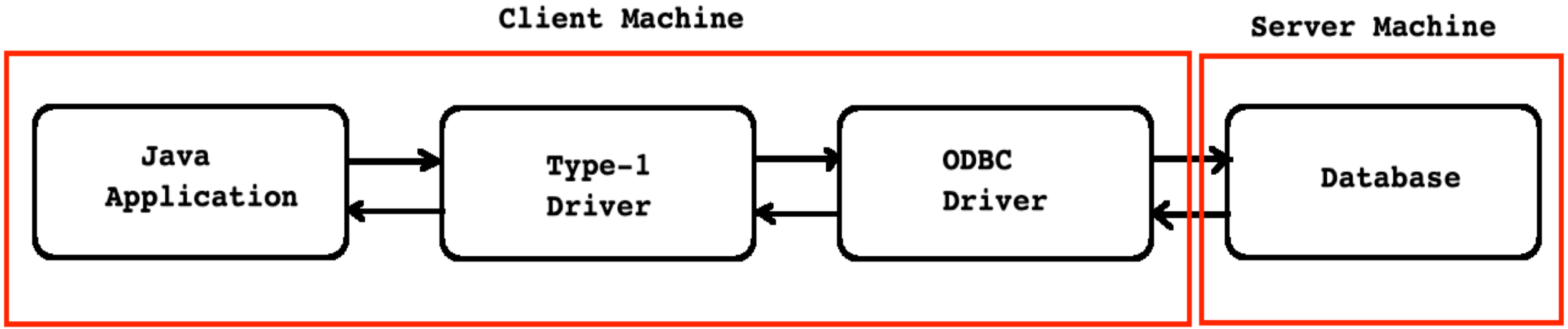
JDBC Driver

- Driver is a program, which converts JDBC API Calls(Java Request) into database specific calls(SQL Request) and vice versa.



- **On the basis of functionality and architecture there are 4 types of driver available:**
  1. Type – 1 Driver (JDBC-ODBC Bridge Driver)
  2. Type – 2 Driver (Native API Driver)
  3. Type – 3 Driver (Network Protocol / Middleware Driver)
  4. Type – 4 Driver (Database Protocol / Thin Driver)
- **Progress Data direct company provides Non standard Type – 5 Driver.**

JDBC Type 1 Driver Architecture



Type-1 Driver : To convert JDBC call into ODBC call or vice versa.

ODBC Driver : To convert ODBC call into database specific call or vice versa.

- Type – 1 driver is also called as JDBC-ODBC bridge driver.
- Example : sun.jdbc.odbc.JdbcOdbcDriver

JDBC Type-1 Driver

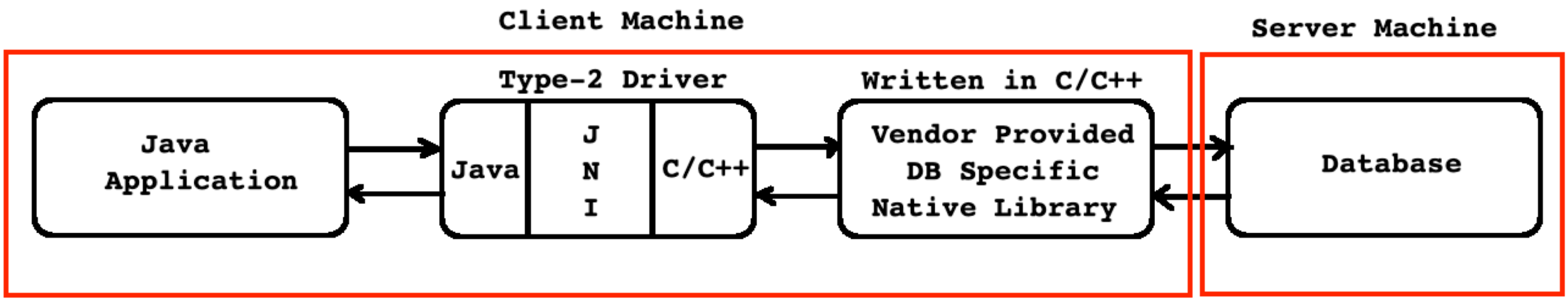
- **Advantages:**

  1. It is very easy to use and maintain.
  2. It comes with JDK hence separate installation is not required.
  3. It is database independent driver hence migration from one database to another is easy.

- **Limitations:**

  1. It requires multiple conversion hence it is slower in performance.
  2. It depends on ODBC driver which work on Microsoft Windows operating system only.
  3. It is obsolete driver. No support from JDK 1.8 onwards.

JDBC Type 2 Driver Architecture



JNI : Calls C/C++ function into Java.

Type-2 Driver : Converts JDBC call into C/C++ Call which can understand by DB.

Typically it is provided by database vendor.

- Type-II driver is also called as Native API Driver.

- Example : Oracle Call Interface (OCI) Driver.
- *Note : MySQL do not support Type-2 Driver.

JDBC Type-2 Driver

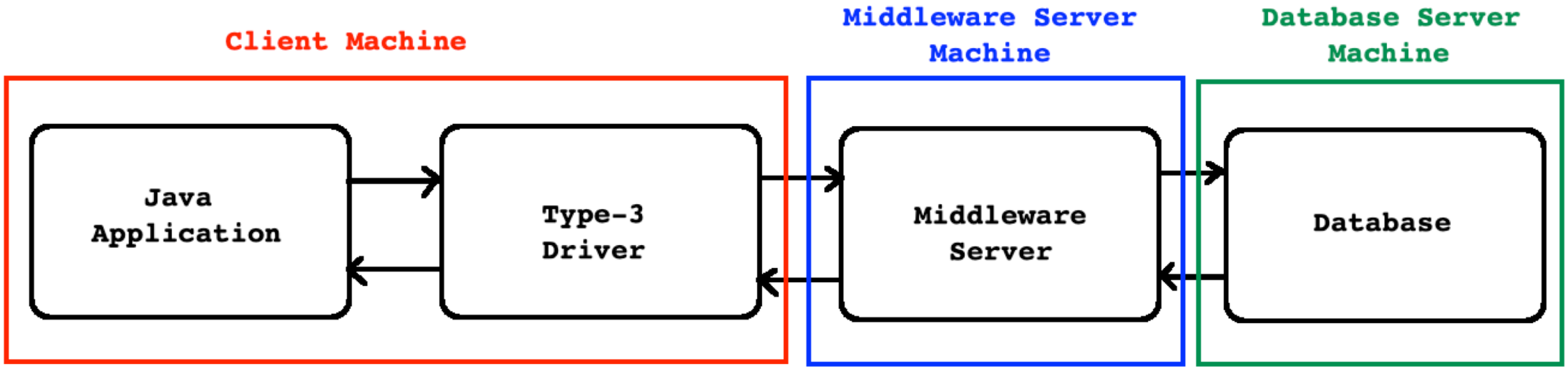
- **Advantages:**

  1. It is based on vendor provided database specific native library hence ODBC support is not required.
  2. In comparison with Type-1 driver, it requires less call to communicate with database hence it is faster in performance.
  3. Available for Windows, Sun Solaris, Linux. Hence more portable than Type-1 Driver.

- **Limitations:**

  1. Since it is database dependant driver, migration from one database to another is difficult.
  2. It is platform dependant driver.
  3. It is necessary to install native library on client machine.
  4. All the database vendors do not provide native library.

JDBC Type-3 Driver Architecture



- Type-3 driver is also called as Middleware driver.
- It is the only driver, which is database independent and platform independent driver.
- Middleware server implicitly use Type-1/Type-2/Type-4 Driver to communicate with database.
- Type-3 driver follows 3 tier architecture.
- Explore : idssoftware.com
- JDBC client use socket to communicate with middleware server.

JDBC Type-3 Driver

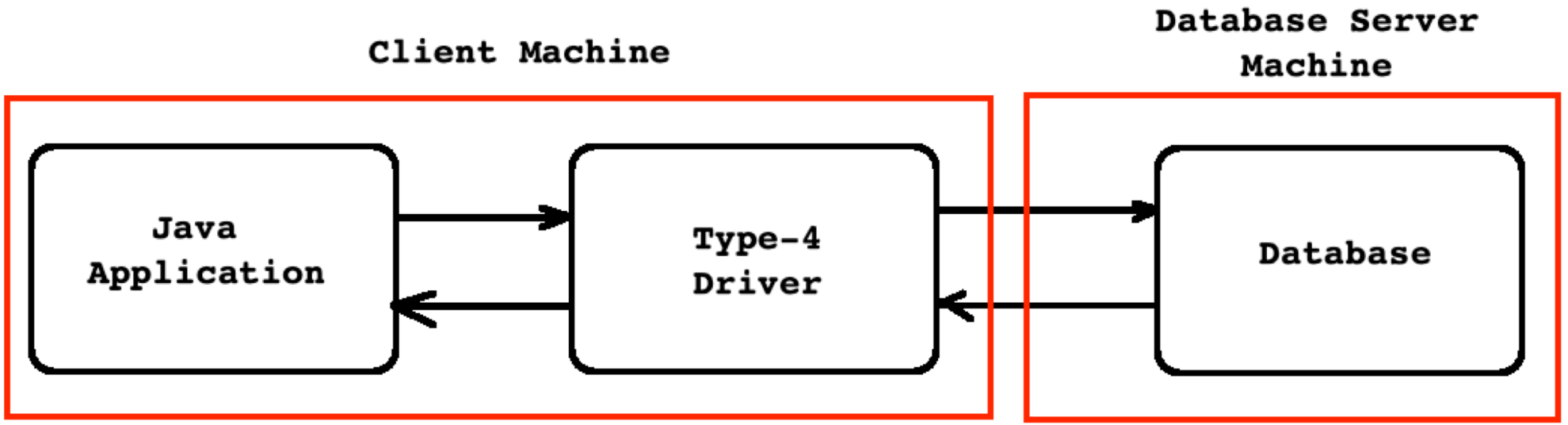
- **Advantages:**

  1. It is a pure Java driver hence truly portable.
  2. No need to install ODBC driver or vendor provided database specific library on client's machine.
  3. Middleware server is a application server which helps us for auditing, load balancing or logging.

- **Limitations:**

  1. Network support is required on client's machine.
  2. It is costly to maintain.
  3. Database-specific coding is required at middleware server.

JDBC Type-4 Driver Architecture



- JDBC Type-4 driver is also called as Database Protocol Driver/Thin Driver /Pure Java Driver
- It is completely written in Java.
- This provides better performance than the type 1 and type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls.
- It directly communicate with database using vendor specific native protocol.

JDBC Type-4 Driver

- **Advantages:**
  1. Completely written in Java to achieve platform independence.
  2. It doesn't translate the requests into an intermediary format such as ODBC.
  3. No need of middleware server. Client application directly connects to database.
  4. Easily available.
  5. Simple to install.
  6. We can use it for standalone application as well as web application.
- **Limitations:**
  1. It is database dependant driver.

Steps to connect Java Application To Database

1. Include database (MySQL) connector into runtime classpath/build path.
2. Import java.sql package.
3. Load and register Driver.
4. Establish connection using users credential(username and password).
5. Create Statement/PreparedStatement/CallableStatement to execute query.
6. Execute query and process result.
7. Close resources.

Configuration Information

1. Username : **sunbeam**
2. Password : **sunbeam**
3. Database : dac_db
4. MySQL : MySQL 8.0.21
5. Port no. : 3306
6. Connector : mysql-connector-java-8.0.21.jar
7. Url : **jdbc:mysql://localhost:3306/dac_db**
8. Driver : **com.mysql.cj.jdbc.Driver**
9. Reference : https://web.mit.edu/java_v1.5.0_22/distrib/share/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html

Driver

1. Driver is interface declared in java.sql package.
2. Every driver class must implement this interface.
3. Driver implementation handles the communication with database.
4. When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager. This means that a user can load and register a driver by calling:

   ```
   Class.forName("com.mysql.cj.JDBC.Driver")
   ```

5. For more details please explore source code available in following file: mysql-connector-java-8.0.21/src/main/user-impl/java/com/mysql/cj/jdbc

MySQL Driver Implementation.

```
public class Driver
    extends NonRegisteringDriver
    implements java.sql.Driver {
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
    public Driver() throws SQLException {
        // Required for Class.forName().newInstance()
    }
}
```

DriverManager

1. The DriverManager class is the traditional management layer of JDBC, working between the user and the drivers.
2. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.
3. In addition, the DriverManager class attends to things like driver login time limits and the printing of log and tracing messages.
4. Note that the javax.sql package, otherwise known as the JDBC 2.0 Standard Extension API, provides the DataSource interface as an alternate and preferred means of connecting to a data source. However, the DriverManager facility can still be used with drivers that support DataSource implementations.
5. For simple applications, the only method in the DriverManager class that a general programmer needs to use directly is DriverManager.getConnection.

```
Connection con = DriverMananager.getConnection( url, user, password );
```

Connection

1. It is an interface declared in java.sql package.
2. Connection implementation instance represents connection with database.
3. A connection session includes the SQL statements that are executed and the results that are returned over that connection.
4. A single application can have one or more connections with a single database, or it can have connections with many different databases.
5. Methods of Connection interface:
    1. Statement createStatement() throws SQLException
    2. PreparedStatement prepareStatement(String sql) throws SQLException
    3. CallableStatement prepareCall(String sql) throws SQLException

Statement

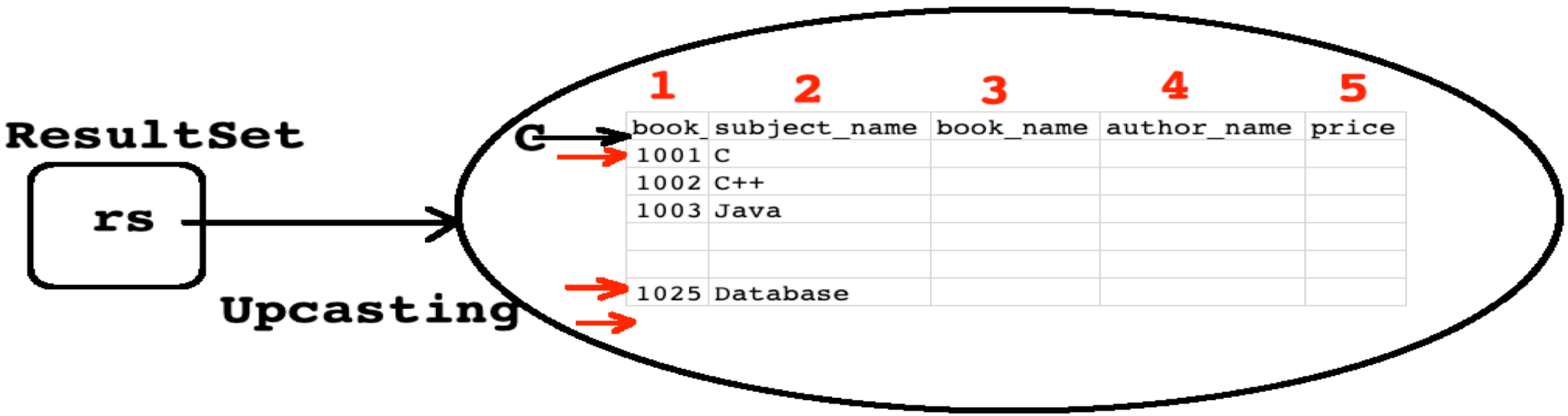
1. It is an interface declared in java.sql package.
2. A Statement object is used to send static SQL statements to a database.
3. The Statement interface provides basic methods for executing statements and retrieving results.
4. Methods of Statement interface:
    1. ResultSet executeQuery(String sql) throws SQLException
    2. int executeUpdate(String sql) throws SQLException
5. Creating Statement Object:

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

ResultSet

1. It is an interface declared in java.sql package.
2. A ResultSet is a Java object that contains the results of executing an SQL query. In other words, it contains the rows that satisfy the conditions of the query.
3. The data stored in a ResultSet object is retrieved through a set of get methods that allows access to the various columns of the current row. The
4. A ResultSet object maintains a cursor, which points to its current row of data. The cursor moves down one row each time the method next is called. When a ResultSet object is first created, the cursor is positioned before the first row, so the first call to the next method puts the cursor on the first row, making it the current row. ResultSet rows can be retrieved in sequence from top to bottom as the cursor moves down one row with each successive call to the method next.

ResultSet



Instance of ResultSet Implementation Class

```
stmt.executeQuery( String sql )
boolean status = rs.next( );
```

Day 5

Components of Java Platform

```
1. Java API
2. Java Virtual Machine
```

Components of JVM

```
1. Class Loader Subsystem
2. Runtime Data Areas
3. Execution Engine
```

Class Loaders

```
1. BootStrap ClassLoader
2. Extension ClassLoader
3. Application ClassLoader
```

Runtime Data Areas

```
1. Method Area
2. Heap
3. Java Stack
4. PC register
5. Native Method Stack
```

Execution Engine

```
1. Interpreter
2. Just In Time Compiler
```

Value Type

- Primitive type is called as value type.
- boolean, byte, char, short, int, float, double, long are primitive / value types.
- We can not create variable/instance of value type using new operator.

```
int num1 = 10;  //OK
int num1 = new int( 10 );//NOT OK
```

- Variable of value type get space on Java stack.
- A variable of value type contains value.
- If we try to initalize/assign variable of value type then value gets copied.

```
int num1 = 10;    //OK
int num2 = num1;    //OK
```

- We can not store null value inside variable of value type.

```
int num1 = null;    //NOT OK
```

- Field of value type, by default contains 0.

```
class Test{
    private boolean b;  //false -> 0
    private int i;      //0
    private float f;    //0.0
    private double d;   //0.0
    private long l;     //0
}
```

Reference Type

- Non primitive type is called as reference type.
- Interface, Class, Enum, Array are non pritive type / reference type.
- To create instance of reference type it is mandatory to use new operator.

```
Complex c1(10,20);  //NOT OK
Complex c1 = new Complex( 10, 20 ); //OK
```

- A variable of reference type contains reference.
- Instance of reference type get space on heap.
- If we try to initalize/assign variable of reference type then reference gets copied.

```
Complex c1 = new Complex(10,20);
Complex c2 = c1;    //OK
```

- We can store null value inside variable of reference type.

```
Complex c1 = null;  //OK
```

- Field of reference type by default contains null value.

```
class Date{
}
class Employee{
    private Date joinDate;  //null
}
```

Characteristics of Instance

- Only Fields get space once per instance and according to order of their declaration inside class.
- Method do not get space inside instance rather all the instances of same class share single copy it.
- Instances share single copy of method by passing this reference.

State

- Data/value stored inside instance is called as state.

```
int num1 = 10;  //state of num1 is 10
Employee emp = new Employee("Sandeep",33,15000);
//State of instance is "Sandeep",33,15000
```

- value of of the field represents state of instance.

Behavior

- Set of operations which are allowed to perorm on instance is called behavior of instance.

```
Employee emp = new Employee();
emp.acceptRecord( );
emp.printRecord( );
//behavior of instance is acceptRecord and printRecord
```

- Method defined inside class represent behavior of instance.

Identity

- It is a property of instance which is used to identify instance uniquely.
- A value of any field which is used to identify instance uniquely represents it identity.

Class

- Class is collection of fields and methods.
- Structrue(skeleton) and behavior of instance depends on class hence class is considered as a template/model/blueprint for an instance.
- Class is collection/set of such instances which is having common structure and common behavior.
- Tata NANO, Maruti 800, Ford Figo -> class -> Car
- Akash, Yogesh, Nitin, Nilesh, Sandeep -> class -> Person
- Let US C, Mastering DBT, Core Java Vol-1 -> class -> Book
- NOKIA 1100, One Plus 5T, iPhone6 -> class -> MobilePhone
- MacBook Air, Lenovo Thinkpad, HP Envy, Dell Inspiron -> class -> Laptop
- Class is a imaginary / logical entity.
- Class implementation represents encapsulation.

Instance

- Any entity which has physical existance is called object.
- In Java object is called as instance.
- An entity which has State, Behavior, Idenitity is called instance.
- Instance is real time / physical entity.

Instantiation

- Process of creating instance from a class is called instantiation.
- It represents abstraction.
- Field of a class which get space inside instance is called instance variable. In short, non static field declared inside class is called instance variable.
- Instance variable get space inside instance during instantiation once per instance according to order of their declaration inside class.
- A method of a class, which is designed to proces state of instance / designed to call on instance is called instance method. In short, non static method of a class is called instance method.
- If we want to access instance members then we should use object reference(this, t1, t2, t3).

Class Level variable

- If we want to share value of any field inside all the instances of same class then we should declare such field static.
- Static field do not get space inside instance rather all the instances of same class share single copy of it hence size of instance depends on size of non static field.
- Field of a class which do not get space inside instance is called class level variable. In short static field is also called as class level variable.
- To access class level variable we should use classname and dot opetator.
- Static field / class level variable get space once per class during class loading on method area.

Constructor

- If we want to initialize non static field/instance variable then we should use constructor.
- Constructor gets called once per instance

- We can call constructor from another constructor. it is called constructor chaining.

Static Initialization Block

- If we want to initialize static field/class level variable then we should use Static Initialization Block.
- Static Initialization Block gets called once per class.
- We can not call Static Initialization Block from another Static Initialization Block.
- We can write multiple static block inside class. In this case JVM execute them sequentially.

Static Method

- To access non static members of the class method should be non static and to access static members of the class method should be static.
- Static method of a class is also called as class level method and class level method is designed to call on class name.
- Why static method do not get this reference?
  - If we call non static method on instance then method get this reference.
  - Static method is designed to call on class name.
  - Since static method is not designed to call on instance it doesnt get this reference.
- Since static method do not get this reference, hence we can not access non static membrs inside it. In other words static method can access only static members of the class.
- Using instance, we can access non static members inside static method.

```
public class Program {
    private int num1 = 10;
    private static int num2 = 20;
    public static void main(String[] args) {
        //System.out.println("Num1  :   "+num1);    //Not OK
        Program p = new Program();
        System.out.println("Num1    :   "+p.num1);  //OK : 10
        System.out.println("Num2    :   "+num2);    //OK : 20
    }
}
```

- Inside method, if there is a requirement to use this reference then, we should delcare that method non static otherwise it should be static.

Day 6

Method Overloading

- Consider code in C

```c
void sum( int num1, int num2 )
{
    int result = num1 + num2;
    printf("Result  :   %d\n", result);
}
void add(int num1, int num2, int num3) {
    int result = num1 + num2 + num3;
    printf("Result  :   %d\n", result);
}
int main( void ) {
    sum( 10, 20 );
    add( 10, 20 , 30 );
    return 0;
}
```

- In C, we can give same name to the multiple functions.
- If implemenation of method is logically same then we should give same name to the method. It helps developer to reduce maintenance of code.
- If we want to give same name to the method then it is necessary to follow some rules.

1. If we want to give same name to the method and if type of all the parameters are same then number of parameters passed to the method must be different.

```java
public class Program {
    public static void sum( int num1, int num2 ) {
        int result = num1 + num2;
        System.out.println("Result  :   "+result);
    }
    public static void sum(int num1, int num2, int num3) {
        int result = num1 + num2 + num3;
        System.out.println("Result  :   "+result);
    }
    public static void main(String[] args) {
        Program.sum( 10, 20 );
        Program.sum( 10, 20 , 30 );
    }
}
```

2. If we want give same name to the method and if number of parameters are same then type of at least one paramters must be different.

```
public class Program {
    public static void sum( int num1, int num2 ) {
        int result = num1 + num2;
        System.out.println("Result  :   "+result);
    }
    public static void sum(int num1, double num2) {
        double result = num1 + num2 ;
        System.out.println("Result  :   "+result);
    }
    public static void main(String[] args) {
        Program.sum( 10, 20 );
        Program.sum( 10, 20.5 );
    }
}
```

3. If we want to give same name to the method and if number of parameters passed to the method are same then order of type of parameters must be diffrent.

```
public class Program {
    public static void sum( int num1, float num2 ) {
        float result = num1 + num2;
        System.out.println("Result  :   "+result);
    }
    public static void sum(float num1, int num2) {
        float result = num1 + num2 ;
        System.out.println("Result  :   "+result);
    }
    public static void main(String[] args) {
        Program.sum( 10, 20.1f );
        Program.sum( 10.1f, 20 );
    }
}
```

- Method can return value but catching that value is optional.

4. On the basis of only diffrent return type we can not give same name the method.

```
public class Program {
    public static int sum( int num1, int num2 ) {
        int result = num1 + num2;
        return result;
    }
    public static void sum( int num1, int num2 ) { //Error
        int result = num1 + num2;
        System.out.println("Result  :   "+result);
    }
    public static void main(String[] args) {
        int res = Program.sum(10, 20);
```

```
        System.out.println("Result  :   "+res);

        Program.sum(50, 60);
    }
}
```

- Process of defining method using above rules is called as method overloading. In simple words, process of defining method with same name and diffrent signature is called method overloading.
- Methods, which take part in overloading are called overloaded methods.
- We can overload main method in Java

```
public class Program {
    public static void main(String message ) {
        System.out.println(message);
    }
    public static void main(String[] args) {
        Program.main("Hello Dac");
    }
}
```

- We can define multiple constructors inside class. It is called constructor overloading.

```
class Complex{
    private int real, imag;
    public Complex( ){
        this(0,0);
    }
    public Complex( int real ){
        this.real = real;
    }
    public Complex( int real, int imag ){
        this.real = real;
        this.imag = imag;
    }
}
```

- We can overload static as well as non static methods.
    - valueOf() is overloaded static method of java.lang.String class.
    - print, println, printf are non static overloaded methods of java.io.PrintStream class.
- To overload method, minimum 2 definitions are required.
- If implemenation of methods are logically same/equivalent then we should overload method.
- For method overloading, methods must be exist inside same scope.
- Since catching value from method is optional, return type is not considered in method overloading.

Final

- final is keyword in Java

```
public class Program {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        final int num1;
        System.out.print("Num1  :   ");
        num1 = sc.nextInt();
        //num1 = 200;
        System.out.println(num1);
    }
    public static void main3(String[] args) {
        final int num1;
        num1 = 10;  //OK
        //num1 = 20;    //NOT OK
        System.out.println(num1);
    }
    public static void main2(String[] args) {
        final int num1 = 10;    //OK
        //num1 = 20;    //Not OK
        System.out.println(num1);
    }
    public static void main1(String[] args) {
        final int count = 10;
        //count = count + 1;    //Not OK
        System.out.println("Count   :   "+count);
    }
}
```

- If we dont want to modify value of method local variable then we should declare it final.
- Once we store value inside final variable then we can not modify it.
- We can provide value to the final variable at compile time as well as runtime.

Final Field

- If we dont want to modify state/value of any field inside any method of the class then we should declare such field final.
- If we want to declare any field final then we should declare it static too.

```
class Test{
    public static final int number = 10;
    public void showRecord( ) {
        //this.number  = this.number + 1;   //Not OK
        System.out.println("Number  :   "+Test.number);
    }
    public void displayRecord( ) {
        //this.number  = this.number + 1;   //Not OK
        System.out.println("Number  :   "+Test.number);
```

```
    }
}
public class Program {
    public static void main(String[] args) {
        Test t = new Test( );
        t.showRecord();
        t.displayRecord();
    }
}
```

- We can declare reference final but we can not declare instance final.

```
final Complex c1 = new Complex(10, 20);
c1.setReal(11);
c1.setImag(22);
c1.printRecord();

c1 = new Complex(50, 60);   //Not OK
```

Day 7

Coding / Naming Conventions

```
1. Pascal Case Coding Convention
2. Camel Case Coding Convention
```

Pascal Case Coding Convention

- Example
    1. System
    2. StringBuilder
    3. NullPointerException
    4. IndexOutOfBoundsException
- In this case, Including first word, first character of each word must be in upper case.
- In Java, we should use this convention for:
    1. Type Name(Interface, Class, Enum, Annotation)
    2. File Name.

```
File Name : Program.java
class Program{
}
```

Camel Case Coding Convention

- Example
    1. main( )
    2. parseInt( )
    3. showInputDialog( )
    4. addNumberOfDays( )
- In this case, excluding first word, first character of each word must be in upper case.
- We should use this convention for:
    1. Method local variable
    2. Method parameter
    3. Field
    4. Method
    5. Object referece / reference

```
File Name : Complex.java
class Complex{
    private int real;
    private int imag;
```

```
    public int getReal( );
    public void setReal( int real );
    public int getImag( );
    public void setImag( int imag );
}
```

Coding convention for final variable and enum constant

- Final Variable

```
class Math{
    public static final double PI = 3.142;
}
```

- Name of the final variable / field must be in upper case.
- enum

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

- Name of the enum constant must be un upper case.

Naming convention for package

- Example
    1. java
    2. java.lang
    3. java.lang.reflect
    4. java.util
    5. java.io
    6. com.mysql.cj.jdbc
    7. org.cdac.sunbeam.dac
- Name of the package should be in lower case.

Path

- Example: C:\CDAC\Sunbeam\dac\java\Day6\Day_6.1\src\Program.java
- Path contains:
    1. Root directory
    2. Path seperator character.
    3. Sub directories
    4. File Name

Absolute path

- A path of a file from root directory is called absolute path.
- Example: C:/CDAC/Sunbeam/dac/java/Day6/Day_6.1/src/Program.java

Relative path

- A path of a file from current directory is called relative path. cd c: cd CDAC cd SunBeam cd dac cd java cd Day6 cd Day_6.1
- How to refere Program.java
    1. C:/CDAC/Sunbeam/dac/java/Day6/Day_6.1/src/Program.java
    2. .\src\Program.java

Package

- It is java language feature that we can use to group functionally equivalent types together and to avoid name ambiguity.
- Package can contain sub package, interface, class, enum, error, exception and type annotation.
- package is a keyword in Java.
- How to add type inside package?
    - File Name : Complex.java

```
class TComplex{

}
//Output : TComplex.class
```

```
- File Name : Complex.java
```

```
package p1; //OK
class TComplex{ //OK
}
```

- If we define any type/class inside package then it is called packaged type/class.
    - File Name : Complex.java

```
class TComplex{ //OK
}
package p1; //NOT OK
```

```
- File Name : Complex.java
```

```
import java.util.Scanner; //OK
package p1; //NOT OK
class TComplex{ //OK
}
```

```
- File Name : Complex.java
```

```
package p1; //OK
package p2; //NOT OK
class TComplex{ //OK
}
```

- Package declaration statement must be first statement inside .java file.
    - File Name : Complex.java

```
package p1,p2; //NOT OK
class TComplex{ //OK
}
```

- We can define class inside one package only.
    - File Name : Complex.java

```
package p1; //OK
class TComplex{ //OK
}
```

- Steps:
    1. set path(optional : open jdk)
    2. Compile Complex.java
        - javac -d ./bin ./src/Complex.java
        - Output : p1/TComplex.class
- Package name is pysically mapped to the folder.
    - File Name : Complex.java

```
package p1; //OK
class TComplex{ //OK
}
```

```
- File Name : Program.java
```

```
import p1.TComplex;
class Program{
    public static void main(String[] args) {
        //Step No 1 : Use F.Q. Class/Type Name
        //p1.TComplex c1 = new p1.TComplex( );

        //Step No 2 : Use import statement
        TComplex c1 = new TComplex( );
    }
}
```

- If we want to use packaged type inside different package then
    1. Either we should use F.Q. Type name
    2. Or we should use import statement.
- Default access modifier of any type is package level private/default.

```
package p1; //OK
??? class TComplex{
}
```

- Here "???" means package level private / default.
- If access modifier of any type is package level private the we can not use it in different package.
- If we want to use any type outside package then access modifier of type must be public.
    - File Name : Complex.java

```
package p1; //OK
public class TComplex{  //OK
}
```

```
- File Name : Complex.java
```

```
package p1; //OK
private class TComplex{  //NOT OK
}
```

```
- File Name : Complex.java
```

```
package p1; //OK
protected class TComplex{  //NOT OK
}
```

- Access modifier of type can be either package level private(default) or public only.
- According to Java Language Specification, name of the public type and name of the .java file must be same.
    - File Name : Complex.java

```
package p1; //OK
public class Complex{ //Packaged Class
}
```

```
- File Name : Program.java
```

```
import p1.Complex;
class Program{  //Unpackaged Class
    public static void main(String[] args) {
        //Step No 1 : Use F.Q. Class/Type Name
        //p1.Complex c1 = new p1.Complex( );

        //Step No 2 : Use import statement
        Complex c1 = new Complex( );
    }
}
```

- Compilation Steps

```
    javac -d ./bin ./src/Complex.java   =>  Complex.class
    set classpath=./bin;
    javac -d ./bin ./src/Program.java   =>  Program.class
    java Program
```

- We can use packaged types inside unpackaged type.
- Consider unpackaged class
    - Complex.java

```
public class Complex{   //Unpackaged class=> part of default package
    public void print( ){
```

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

```
javac -d ./bin ./src/Complex.java
export CLASSPATH=./bin
```

- If we define any type without package then it is considered as a member of default package.
- We can not import default package.
- Since we can not import default package, it is not possible to use unpackaged type from packaged type.
- Consider packaged class

```
package p1;
public class Program {  //Packaged class
    public static void main(String[] args) {
        Complex c1 = new Complex( );
        c1.print( );
    }
}
```

```
javac -d ./bin ./src/Program.java --Error
```

- Consider packaged class
    - Complex.java

```
package p1;
public class Complex{   //packaged class
    public void print( ){
        System.out.println("Hello World");
    }
}
```

- Consider packaged class

```
package p2;
import p1.Complex;
public class Program {  //Packaged class
    public static void main(String[] args) {
        Complex c1 = new Complex( );
```

```
        c1.print( );
    }
}
```

- Steps for compilation
    1. javac -d ./bin ./src/Complex.java //p1/Complex.class
    2. export CLASSPATH=./bin
    3. javac -d ./bin ./src/Program.java //p2/Program.class
    4. java p2.Program
- Consider packaged class
    - Complex.java

```
package p1;
public class Complex{   //packaged class
    public void print( ){
        System.out.println("Hello World");
    }
}
```

- Consider packaged class

```
package p1;
//import p1.Complex;  //Optional
public class Program {  //Packaged class
    public static void main(String[] args) {
        Complex c1 = new Complex( );
        c1.print( );
    }
}
```

- Steps for compilation
    1. javac -d ./bin ./src/Complex.java //p1/Complex.class
    2. export CLASSPATH=./bin
    3. javac -d ./bin ./src/Program.java //p1/Program.class
    4. java p1.Program
- Consider packaged class
    - Complex.java

```
package p1.p2;
public class Complex{   //packaged class
    public void print( ){
```

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

- Consider packaged class

```
package p1.p3;
import p1.p2.Complex;  //Optional
public class Program {  //Packaged class
    public static void main(String[] args) {
        Complex c1 = new Complex( );
        c1.print( );
    }
}
```

- Steps for compilation
    1. javac -d ./bin ./src/Complex.java //p1.p2/Complex.class
    2. export CLASSPATH=./bin
    3. javac -d ./bin ./src/Program.java //p1.p3/Program.class
    4. java p1.p3.Program

Day 8

- java.lang package contains all the fundamentals types of core java.
- java.lang package is by default imported in every .java file. Hence to use any type declared in java.lang package, use of import statement is optional.

Static Import

- Consider following code:

```
class Test{
    public static void showRecord( ) {
        System.out.println("Test.showRecord()");
    }
}
public class Program {
    public static void displayRecord( ) {
        System.out.println("Program.displayRecord()");
    }
    public static void main(String[] args) {
        Program.displayRecord();    //OK
        displayRecord();            //OK

        Test.showRecord();          //OK
        showRecord();           //NOT OK
    }
}
```

- If static members belong to the same class then static method can access another static members directly(w/o class name).
- If static members belong to the different class then static method can not access another static members directly(w/o class name). In this case it is mandatory to use class name and dot operator.
- If we want to use types(Interface/Class/Enum/Annotation) declared in package inside different package then we should use import statement.

```
import static java.lang.System.out;
import static java.lang.Math.*;
public class Program {
    public static void main(String[] args) {
        double radius = 10.5d;
        double area = PI * pow(radius, 2);
        out.println("Area   :   "+area);
    }
}
```

- If we want to use static members of the class w/o class name then we should use static import statement.

Array

- Collection : A data structure which contains elements.
- Data Structures:
    - Linear/Sequential
        1. Array
            - Single dimensional
            - Multi dimensional
            - Ragged Array
        2. Stack
        3. Queue
            - Linear queue
            - Circular queue
            - Priority Queue
            - Double Ended Queue(Deque "Deck")
        4. LinkedList
            - Singly LinkedList
                - Linear Singly LinkedList
                - Circular Singly LinkedList
            - Doubly LinkedList
                - Linear Doubly LinkedList
                - Circular Doubly LinkedList
    - Non Linear
        - Tree
        - Graph
        - Hashtable
- element : A value stored inside data structure/collection is called element.
- Array is a linear data structure/collection which is used to store elements of same type in continous memory location.
- Array is a reference type. In other words to create instance of array it is mandatory to use new operator.
- Example : Create array of 3 integers

```
new int[ 3 ];
```

- If we want to access elements of array then we should use integer index. Array index always begings with 0.
- If we create array of 5 integers:
    - Min index : 0

    - Max Index : 4
- In Java, checking array bounds(min & max index) is a job of JVM.
- If we specify illegal index then JVM throws ArrayIndexOutOfBoundsException.
- "java.util.Arrays" class contains various methods for manipulating arrays (such as sorting and searching).
    - public static List asList(T... a)
    - public static int binarySearch(int[] a, int key)
    - public static int[] copyOf(int[] original, int newLength)
    - public static void sort(int[] a)
    - public static void sort(T[] a, Comparator<? super T> c)
    - public static String toString(int[] a)

Types of Array

```
1. Single dimensional Array
2. Multi dimensional Array
3. Ragged Array
```

Single dimensional Array

- Create instance of single dimensional array

```
new int[ 3 ];   //Get space on Heap
```

- Array w/o reference is called anonymous array.
- If we want to process elements of array then it is necessary to create reference of array.

```
int arr1[ ]; //OK : Array reference

int [ arr2 ]; //NOT OK

int[ ] arr3; //OK : Array reference => Recommended
```

- If we try to create array instance with negative size then JVM throws NegativeArraySizeException.

```
public static void main(String[] args) {
    int[] arr = new int[ -3 ]; //NegativeArraySizeException
}
```

- Types of loop
    1. do-while
    2. while loop

3. for loop
4. for each loop(also called as iterator)
    - It is read only and forward only loop.

Multi Dimensional Array

- Array of array(Array in which every element is array) in which column size of every array is same is called multi dimensional array.

Ragged Array

- Array of array in which column size of every array is different is called ragged.

Array of primitive type

```
boolean[] arr = new boolean[ 3 ];
for( boolean element : arr )
    System.out.println(element);    //false, false, false
```

```
int[] arr = new int[ 3 ];
for( int element : arr )
    System.out.println(element);    //0,0,0
```

```
double[] arr = new double[ 3 ];
for( double element : arr )
    System.out.println(element);    //0.0,0.0,0.0
```

- If we create array of primitive/value type then default value of elements of array depends on default value of the data type.

Array of references

```
public class Program {
    public static void main(String[] args) {
        Complex[] arr = new Complex[ 3 ];   //Array of references
        System.out.println(Arrays.toString(arr));
    }
}
```

- If we create array of references then by default elements of array contains null value.

Array of instances of non primitive type

```
public class Program {
    public static void main(String[] args) {
        Complex[] arr = new Complex[ 3 ];   //Array of references
        for( int index = 0; index < arr.length; ++ index )
            arr[ index ] = new Complex( );
    }
}
```

Parameter passing methods/Ways

- In C, We can pass argument to the function using:
    1. By Value
    2. By Address / by reference
- In Java, We can pass argument to the method using:
    1. By Value only.

Day 9

Variable Arity Method

- A method which can accept variable number of arguments is called variable arity method / variable argument method.

```
public class Program {
    //Variable argument  / variable arity method.
    public static void sum( int... arguments ) {
         int result = 0;
         for( int element : arguments )
             result = result + element;
         System.out.println("Result :   "+result);
    }
    public static void main(String[] args) {
        Program.sum( );
        Program.sum( 10, 20, 30 );
        Program.sum( 10, 20, 30, 50, 60 );
        Program.sum( 10, 20, 30, 50, 60, 70, 80, 90, 100 );
    }
}
```

- Example:

    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);

- Question : Write a program to print "Hello World" without giving semicolon.

```
public class Program {
    public static void main(String[] args) {
        if ( System.out.printf("Hello World") != null ) {
        }
    }
}
```

Java Comments

- If we want to maintain documentation of source code then we should use comment.
- Types of comments in Java
    1. //Single line comment
    2. /* Multiline comment */
    3. /** Java Documentation/Java Doc comment */
- Example

```
/**
     * Returns the {@code char} value at the
     * specified index. An index ranges from {@code 0} to
     * {@code length() - 1}. The first {@code char} value of the sequence
     * is at index {@code 0}, the next at index {@code 1},
     * and so on, as for array indexing.
     *
     * <p>If the {@code char} value specified by the index is a
     * <a href="Character.html#unicode">surrogate</a>, the surrogate
     * value is returned.
     *
     * @param      index   the index of the {@code char} value.
     * @return     the {@code char} value at the specified index of this
string.
     *             The first {@code char} value is at index {@code 0}.
     * @exception  IndexOutOfBoundsException  if the {@code index}
     *             argument is negative or not less than the length of
this
     *             string.
     */
public char charAt(int index) {
    if ((index < 0) || (index >= value.length)) {
        throw new StringIndexOutOfBoundsException(index);
    }
    return value[index];
}
```

Enum

- Consider code in C:

```
enum ArithmeticOperation{
    //Enum Constants/enumerator
    EXIT, SUM, SUB, MULTIPLICATION, DIVISION
};
enum ArithmeticOperation menu_list( void ){
    enum ArithmeticOperation 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;
}
int main( void ){
    enum ArithmeticOperation choice;
    while( ( choice = menu_list( ) ) != EXIT ){
        int result = 0;
```

```
        switch( choice ){
        case SUM:
            result = sum( 100, 20 );
            break;
        case SUB:
            result = sub( 100, 20 );
            break;
        case MULTIPLICATION:
            result = multiplication( 100, 20 );
            break;
        case DIVISION:
            result = division( 100, 20 );
            break;
        }
        printf("Result  :   %d\n", result);
    }
    return 0;
}
```

- If we want to improve readability of the source code then we should use enum.
- In Java, Using enum, we give name to any literal or group of literal.

```
enum Day{
    SUN("SunDay"), MON(2), TUES("TuesDay",3);
}
```

- Consider Example

```
enum Color{
    RED, GREEN, BLUE    //Name of enum constants
    //RED=0, GREEN=1, BLUE=2    //0,1,2 => Ordinal
}
```

- We can not change ordinal of enum constant.
- enum is a keyword in Java.
- enum is non primitive/reference type in Java.
- java.io.Serializable is empty interface. It is also called marker / tagging interface.
- java.lang.Comparable is a interface. "int compareTo(T other)" is a method of Comparable interface.
- java.lang.Object is a class. It is having 12 methods:
    1. toString()
    2. equals()
    3. hashCode()
    4. clone()
    5. finalize()
    6. getClass()
    7. 3 overloaded wait methods.

8. notify()
9. notifyAll( )

- java.lang.Enum is an abstract class which is considered as super class for enum.
- This class is introduced in JDK 1.5
- Methods of Enum class:
    1. public final Class getDeclaringClass()
    2. public final String name()
    3. public final int ordinal()
    4. public static <T extends Enum> T valueOf(Class enumType, String name);
- Consider enum in Java

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

- Compiler generate .class file per interface, class and enum.
- Compiler generated code for enum

```
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( String str );
}
```

- If we define enum in Java, then it is implicitly considered as final class. Hence we can not extends enum.
- Enum constants are references of same enum which is considered as public static final field of the class. Please consider above code.
- values() and valueOf() are methods of enum which gets added at compile time.
- In Java, if we want to assign name to the literals then it is mandatory to define constructor inside enum. And to get value of the literal it is nessary to define getter methods inside enum.

```
enum Day{
    SUN("SunDay"), MON(2), TUES("TuesDay", 3);//Must be first line
    private String dayName;
    private int dayNumber;
    private Day(String dayName) {
        this.dayName = dayName;
    }
    private Day(int dayNumber) {
        this.dayNumber = dayNumber;
    }
    private Day(String dayName, int dayNumber) {
```

```
        this.dayName = dayName;
        this.dayNumber = dayNumber;
    }
    public String getDayName() {
        return dayName;
    }
    public int getDayNumber() {
        return dayNumber;
    }
}
```

Object Oriented Programming Structure/System(OOPS)

```
- It is not a syntax.
- It is a process/methodology that we can implement using any OO
programming language.
```

Major Pillars Of OOps

```
1. Abstrasction
2. Encapsulation
3. Modularity
4. Hierarchy
```

Minor Pillars Of OOps

```
1. Typing / Polymorphism
2. Concurrency
3. Persistence
```

Abstraction

- It is a major pillar of OOPS.
- It is a process of getting essential things from system.
- Abstraction focuses on outer behavior of instance.
- Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.
- Using abstraction, we can achieve simplicity.
- Abstraction in Java

```
Complex c1 = new Complex( );
c1.acceptRecord( );
```

```
c1.printRecord( );
```

- Creating instance and calling method on it is abstrasction in Java.

```
import java.util.Scanner;
Scanner sc = new Scanner( System.in );
int number = sc.nextInt( );
```

Encapsulation

- It is a major pillar of oops.
- Defintion:
    1. Binding of data and code together is called encapsulation.
    2. To achive abstrasction, we need to implement some business logic. It is called encapsulation.
- Class implementation represents encapasulation.
- Encapsulation in Java

```
class Complex{
    //Fields of the class   => Data
    private int real;
    private int imag;
    //Methods of a class    =>  Code
    public Complex( ){
    }
    public void acceptRecord( ){
    }
    public void printRecord( ){
    }
}
```

- Encapsulation represents internal behavior of the instance.
- Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.
- Proces of declaraing field of the class private is called hiding. Hiding represents data encapsulation.

```
class Employee{
    private float salary;   //Hiding
    public void setSalary( float salary ){
        if( salary > 0 )    //Data Security
            this.salary = salary;
        else
            throw new IllegalArgumentException("Invalid Salary");
    }
}
```

- Data hiding helps to achieve data security.
- Process of giving controlled access to the data is called data security.

Modularity

- It is a major pillar of oops.
- Process of developing complex system using small parts/modules is called modularity.
- If we want to minimize module dependancy then we should use modularity.
- With the help .jar file, we can achive modularity.

Hierarchy

- It is a major pillar of oops.
- A level/order/ranking of abstraction is called hierarchy.
- Using hierarchy, we can achive reusability.
- Advnatages of reusability:
    1. We can reduce development time.
    2. We can reduce development cost.
    3. We can reduce develoeprs effort.
- Types of Hierarchy:
    1. Has-a/part-of => Association
    2. Is-a/kind-of => Inheritance/Generalization
    3. Use-a => Dependancy
    4. Creates-a => Instantiation

Typing / Polymorphism

- It is minor pillar of oops.
- An ability of instance to take multiple forms is called polymorphism.
- Polymorphism = Poly(many) + morphism(forms/behavior)
- Types of polymorphism:
    1. Compile time polymorphism
        - Method Overlaoding
    2. Runtime polymorphism
        - Method Overriding
- Using polymorphism, we can reduce maintenance of the system.

Concurrency

- It is minor pillar of oops.
- An ability of OS to execute multiple processes simultaneously is called Concurrency.
- In context of oops it is called concurrency.
- If we want to utilize H/W resources(CPU) efficiently then we should use concurrency.
- In Java, using thread, we can achive concurrency.

Persistance

- It is minor pillar of oops.
- Process of storing state of instanec on HDD is called Persistance.
- Using persistence, we can maintain state of instance on secondry storage.
- Using File IO and JDBC, we can implement persistence.

System Date

```
import java.time.LocalDate;

LocalDate ldt = LocalDate.now();
int day = ldt.getDayOfMonth();
int month = ldt.getMonthValue();
int year = ldt.getYear();
```

Day 10

- Deprecated means Type/Field/method is available to use in Java but not recommended to use because better alternative is available.
- System Date using java.util.Date class

```
Date date = new Date();
int day = date.getDate();
int month = date.getMonth() + 1;
int year = date.getYear() + 1900;
```

- System Date using java.util.Calendar class

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

- Using java.time.LocalDate

```
LocalDate ldt = LocalDate.now();
int day = ldt.getDayOfMonth();
int month = ldt.getMonthValue();
int year = ldt.getYear();
```

- Has-a, is-a, use-a and creates-a are class hierarchies.

Association

- Consider Examples:
    1. Room has-a Chair
    2. Room has-a Wall
    3. Car has-a engine
    4. Car has-a audio system
    5. Human has-a heart
    6. Department has-a faculty.
- If "has-a" realationship is exist between two types(class) then we should use association.
- How will you relate Car and Engine?
    1. Car has-a engine.
    2. Engine is a part of Car.
- Has-a hierarchy is also called as part-of hierarchy.

- Engine is part of car in other words engine instance(BS6) is a part of car instance(Maruti Suzuki - Alto).
- From object oriented point of view, if instance is a part of instance means it must get space inside another instance.

```
class Engine{
}
class Car{
    private Engine e;//Association
    public Car(){
        this.e = new Engine( );
    }
}
//Engine Instance   :   Dependancy Instance
//Car Instance      :   Dependant Instance
class Program{
    public static void main(String[] args) {
        Car c = new Car( );
    }
}
```

- If we declare instance(object) of a class as field inside another class then it is called association.
- In Java, association do not represent physical containment. In Java instance can be part of another instance using reference variable.
- Association => Instance is outside instance.
- How will you relate Person, date and address?
- Java Archive(jar) is a library file of Java that we can use to create reusable component.
- ".jar" files contains:
    1. META-INF - Manifest file
    2. Resources(images/audio files/fonts)
    3. Packages
- Create "associationlib.jar" file
- Project : AssociationLib

```
package org.sunbeam.dac.lib;
class Date{
    private int day, month, year;
}
class Address{
    private String cityName, stateName, pincode;
}
class Person{
    private String name;         //Association
    private Date birthDate;     //Association
    private Address address;    //Association
}
```

- Project : AssociationTest

1. include "associationlib.jar" into classpath/runtime classpath / build path.
2. import org.sunbeam.dac.lib package.

```
class Program{
    public static void main(String[] args) {
        Date date = new Date( );
        Address address = new Address( );
        Person person = new Person( );
    }
}
```

- Association has 2 special forms:
    1. Composition
    2. Aggregation

Composition

- Consider Example of Human and Heart
    - Human has-a heart. //Association

```
class Heart{
}
class Human{
    private Heart h = new Heart( ); //Association => Composition
}
//Heart Instance   :   Dependancy Instance
//Human Instance   :   Dependant Instance
```

- In case of association, if dependancy instance can not exist w/o Dependant instance then it is called composition.
- Composition represents tight coupling.

Aggregation

- Consider Example of Faculty and Department
    - Department has-a faculty.

```
class Faculty{
}
class Department{
    private Faculty f = new Faculty( ); //Association => Aggregation
}
//Faculty Instance   :   Dependancy Instance
//Department Instance   :   Dependant Instance
```

- In case of association, if dependancy instance can exist without Dependant instance then it is called as aggregation.
- Aggregation represents loose coupling.
- Consider Employee has-a joindate

```
class Date{
}
class Employee{
    private Date joinDate = new Date( ); //Association
}
```

Inheritance

- Consider Example
    1. Car is a Vehicle
    2. Book is a Product
    3. Manager is a Employee
    4. SavingAccount is a Account
- If "is-a" relationship is exist between two types then we should use inheritance.
- If we want to create child class / extend class then we should use extends keyword.

```
class Person{   //Parent class / Super class
}
class Employee extends Person{  //Child class / Sub class
}
```

- In Java, parent class is called super class. Child class is called sub class.
- If we create instance of super class then all the non static fields declared in super class get space inside it.
- If we create instance of sub class then all the non static fields declared in super class & sub class get space inside it.
- If we create instance of sub class then all(private/default/protected/public) the non static fields declared in super class get space inside it. In other words, non static fields of super class inherit into sub class.
- Using sub class name, we can use static field declared in super class. In other words, static fields of super class inherit into sub class.
- Fields of sub class do not inherit into super class. All the fields of super class inherit into sub class but only non static field get space inside instance sub class.
- We can call/invoke, non static method of super class on instance of sub class. In other words, non static methods of super class inherit into sub class.

- We can call, static method of super class on sub class. In other words, static methods of super class inherit into sub class.
- Points to remember:
    1. All the non static members(fields + methods) of super class inherit into sub class.
    2. All the static members(fields + methods) of super class inherit into sub class.
    3. Nested type of super class inherit into sub class.
    4. Constructor do not inherit into sub class.
- Every super class is abstraction for the sub class.
- If we create instance of super class then only super class's constructor gets called.
- If we create instance of sub class then first super class and then sub class constructor gets called.
- From any constructor of sub class, by default, super class's parameterless constructor gets call.
- Using super statement, we can call any constructor of super class from constructor of sub class.
- Super statement must be first statement inside constructor body.
- Inheritance is a process of acquiring/accessing/getting properties(fields) and behavior(methods) of super class inside sub class.
- Inheritance is also called as Generalization.
- According to client's requirement, if implementation of exisiting class is partialy complete/logically incomplete then we should extend that class i.e we should use inheritance.
- According to client's requirement, if implementation of super class method is incomplete then we should redefine that method inside sub class.
- If name of members of super class and sub class are same and if we try to access these members using instance of sub class then preference is given to the sub class members.
- If name of members of super class and sub class are same and if we try to access these members using instance of sub class then sub class members hides implementation of super class members. This process is called shadowing.
- Using super keyword, we can use any member of super class inside method of sub class.

Day 11

Modifiers In Java

1. PRIVATE
2. PROTECTED
3. PUBLIC
4. STATIC
5. FINAL
6. SYNCHRONIZED
7. VOLATILE
8. TRANSIENT
9. NATIVE
10. INTERFACE
11. ABSTRACT
12. STRICT

Access Modifiers

- If we control visibility of members of the class then we should use access modifiers.
- There are 4 acccess modifiers in Java:
    1. private(-)
    2. package level private / default(~)
    3. protected(#)
    4. public(+)
- Except constructor, all the members of super class inherit into sub class.

Types of Inheritance

```
1. Interface Inheritance.
2. Implementation Inheritance.
```

- Above types are further classified into 4 types:
    1. Single Inheritance
    2. Multiple Inheritance
    3. Hierarchical Inheritance
    4. Multilevel Inheritance

Interface Inheritance

- During inheritance, if super type and sub type is interface then it is called interface inheritance.

Single Inheritance

```
interface A
{    }
interface B extends A   //OK:Single Interface Inheritance
{    }
```

- During inheritance, if there is single super interface and single sub interface then it is called single interface inheritance.

Multiple Inheritance

```
interface A
{    }
interface B
{    }
interface C extends A, B   //OK:Multiple Interface Inheritance
{    }
```

- During inheritance, if there are multiple super interfaces and single sub interface then it is called multiple interface inheritance.

Hierarchical Inheritance

```
interface A
{    }
interface B extends A
{    }
interface C extends A
{    }
```

- During inheritance, if there is single super interface and multiple sub interfaces then it is called hierarchical interface inheritance.

Multi Level Inheritance

```
interface A
{    }
interface B extends A
{    }
interface C extends B
{    }
```

- During inheritance, if single interface inheritance is having multiple levels then it is called multilevel interface inheritance.

Implementation Inheritance

- During inheritance, if super type and sub type is class then it is called implementation inheritance.

Single Inheritance

```
class A
{    }
class B extends A   //OK:Single Implementation Inheritance
{    }
```

- During inheritance, if there is single super class and single sub class then it is called single implementation inheritance.

Multiple Inheritance

```
class A
{    }
class B
{    }
class C extends A, B   //NOT OK:Multiple Implementation Inheritance
{    }
```

- During inheritance, if there are multiple super classes and single sub class then it is called multiple implementation inheritance.
- Java do not support multiple implementation inheritance.

Hierarchical Inheritance

```
class A
{    }
class B extends A   //OK
{    }
class C extends A   //OK
{    }
```

- During inheritance, if there is single super class and multiple sub classes then it is called hierarchical implementation inheritance.

Multi Level Inheritance

```
class A
{    }
class B extends A
```

```
{    }
class C extends B
{    }
```

- During inheritance, if single implementation inheritance is having multiple levels then it is called multilevel implementation inheritance.
- In Java multi class inheritance is not allowed. In other words, class can extends only one super class(abstract/ concrete).
- During inheritance, members of super class inherit into sub class. Hence using sub class instance we can access members of super class as well as sub class.
- During inheritance, members of sub class do not inherit into super class hence using super class instance we can access members of super clas only.
- Members of super class inherit into sub class hence we can consider sub class instance as a super class instance.
- Example : Every employee is a person.
- Since we can consider sub class instance as as super class instance, we can use it in place of super class instance.
- Consider following code:

```
Employee emp1 = null;   //OK

Employee emp2 = new Employee(); //OK
```

```
Person p1 = null;//OK

Person p2 = new Person( );     //OK
```

```
Employee emp = new Employee( )//OK

Person p1 = emp;    //OK

Person p2 = new Employee( );    //OK
```

- Members of sub class do not inherit into super class hence We can not consider super class instance as a sub class instance.
- Example : Every person is not a employee.

- Since, super class instance, can not be considered as sub class instance, we can not use it in place of sub class instance.
- Consider following code:

```
Person p1 = null;   //OK

Person p2 = new Person( );  //OK
```

```
Employee emp1 = null;   //OK

Employee emp2 = new Employee( );    //OK
```

```
Person p = new Person( );

Employee emp1 = p;   //NOT OK

Employee emp2 = new Person( );   //NOT OK
```

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

```
public static void main(String[] args) {
    Derived derived = new Derived();
    derived.displayRecord();
    //Base base = ( Base )derived;  //Upcasting : OK
    Base base = derived;    //Upcasting : OK
    base.showRecord();
}
```

- In other words, super class reference can contain reference of sub class instance. It is called upcasting.

```
public static void main(String[] args) {
    Base base = new Derived();  //Upcasting : OK
}
```

- In case of upcasting, using super class reference variable, we can access fields of super class only.

```
public static void main(String[] args) {
    Base base = new Derived();  //Upcasting : OK
    System.out.println("Num1    :   "+base.num1);   //OK
```

```
    System.out.println("Num2    :   "+base.num2);   //OK
    System.out.println("Num3    :   "+base.num3);   //NOT OK
}
```

- In Case of Upcasting, using super class reference variable, we can not access sub class specific methods(which are not available in super class).

```
public static void main(String[] args) {
    Base base = new Derived();  //Upcasting : OK
    base.showRecord();//OK
    //base.displayRecord( );    //NOT OK
}
```

- In case of upcasting, from sub class instance, super class reference variable can access only member of super class. It is also called object slicing.
- Upcassting help us to reduce maintenance of the application.
- Process of converting reference of super class into reference of sub class is called downcasting.
- In Case of downcasting, explicit typecasting is mandatory.

```
public static void main(String[] args) {
    Base base = new Derived();  //Upcasting : OK
    base.setNum1(10);   //OK
    base.setNum2(20);   //OK

    Derived derived = (Derived) base;   //Downcasting
    derived.setNum3(30);    //OK

    System.out.println("Num1    :   "+base.getNum1());
    System.out.println("Num2    :   "+base.getNum2());
    System.out.println("Num3    :   "+derived.getNum3());
}
```

- process of redefining method of super class inside sub class is called method overriding.
- Process of calling method of sub class using reference of super class is called dynamic method dispatch.

```
public static void main(String[] args) {
    Base base = new Derived();  //Upcasting
    base.print();   //Runtime Polymorphism / Dynamic Method Dispatch
}
```

Day 12

- In case of upcasting, using super class reference variable, we can access:
    1. Fields of super class inherited into sub class.
    2. Method of super class inherited into sub class.
    3. Overriden method of sub class.
- In case of upcasting, using super class reference variable, we can not access:

1. Fields of sub class
2. Non overriden method of sub class.

- In this case, we need to do downcasting.
- instanceof is an operator. In case of upcasting, it is used to check, super class reference is storing reference which sub class instance.

```
private static void acceptRecord(Product product) {
    System.out.print("Title     :   ");
    product.setTitle(sc.nextLine());
    System.out.print("price     :   ");
    product.setPrice(sc.nextFloat());
    if( product instanceof Book ) {
        Book book = (Book) product; //Downcasting
        System.out.print("Page Count    :   ");
        book.setPageCount(sc.nextInt());
    }else {
        Tape tape = (Tape) product; //Downcasting
        System.out.println("Play Time   :   ");
        tape.setPlayTime(sc.nextInt());
    }
}
```

- If downcasting fails, then JVM throws ClassCastException.

```
public static void main(String[] args) {
    Object obj = new Date( );
    Date date = (Date) obj;     //Downcasting : It will work
    String str = (String) obj;  //Downcasting : ClassCastException
}
```

Rules of method overriding

- Sub class overrides method of super class.

1. Access modifier in sub class method should be same or it should be wider than super class method.
2. Return type in sub class method should be same or it should be sub type. In other words it shout be covariant.

3. Method name, number of parameters and type of parameters in sub class method must be same.
4. Checked exception list in sub class method should be same or it should be sub set.

- Using above rules, if we redifine method in sub class then it is called method overriding.

We can not override following methods in sub class:

```
1. private methods
2. final method
3. static method
```

Override Annotation / Annotation Type

- Override is Annotation declared in java.lang package.
- It generates metadata(data which describes other data) for the compiler.
- If we do not follow rules of method overriding and if we use Override Annotation then Java compipler generates error.

Final method

- According to client's requirement, if implementation of super class method is logically 100% complete then we should declare super class method final.
- We can not override, final method in sub class.
- Final method inherit into sub class hence we can use it with instance of sub class.
- Example:
    - name() and ordinal() methods of java.lang.Enum class
    - Six methods of java.lang.Object class
        1. public final Class<?> getClass( );
        2. public final void wait( ) throws IE;
        3. public final void wait(long timeout) throws IE;
        4. public final void wait(long timeout, int nanos) throws IE;
        5. public final void notify( );
        6. public final void notifyAll( );
- We can declare overriden method final.

abstract method

- According to client's requirement, if implementation of super class method is logically 100% incomplete then we should declare super class method abstract.
- abstract is keyword in Java.
- We can not provide body to the abstract method.
    - A method of a class which is having a body is called concrete method(static/ non static).
    - A method of a class which is not having a body is called abstract method.
- If we declare method abstract then we must declare class abstract.
- It is mandatory to override abstract method in sub class otherwise sub class can be considered as abstract.

```
abstract class A{
    public abstract void f3( );
}
```

- Option 1

```
class B extends A{
    @Override
    public final void f3() {
        System.out.println("B.f3");
    }
}
```

- Option 2

```
abstract class B extends A{
}
```

Abstract class

- a class may/may not contain abstract method.
- Since implementation of abstract class is logically incomplete, we can not instantiate it. But we can create reference it.
- Example:
    1. java.lang.Number
    2. java.lang.Enum
    3. java.util.Calendar
    4. java.util.Dictionary

Final Class

- If implementation of a class is logically 100% complete then we should delcare such class final.
- We can instantiate final class but we can not extend it. In other words, we can not create sub class of final class.
- Example:
    1. java.lang.System
    2. java.lang.Math
    3. All wrapper classes
    4. java.lang.String, StringBuffer, StringBuilder
    5. java.util.Scanner
- we can not use abstract and final keyword together.

Object Class

- Object is non final and concrete class which is declared in java.lang package.
- java.lang.Object is ultimate base class / super cosmic base class / root of Java class hierarchy. In other words, all the classes(not interfaces) are directly or indirectly extended from java.lang.Object class.
- Object class do not extend any class or do not implement any interface.
- It doesn't contain nested type(Interface/class/enum/annotation).
- It doesn't contain any field.
- It contains only parameterless constructor(actually default).

```
Object o1 = new Object( );  //OK
Object o2 = new Object( "SunBeam" );    //NOT OK
```

- It constains 11 methods(5 non final + 6 final methods)
- Following are non final methods of java.lang.Object class.

1. public String toString( );
2. public boolean equals(Object obj);
3. public native int hashCode( );
4. protected native Object clone( )throws CloneNotSupportedException
5. protected void finalize( )throws Throwable

- Following are 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 notity( );
11. public final native void notityAll( );

Day 13

Sole constructor

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

```
abstract class A{
    private int num1;
    private int num2;
    public A(int num1, int num2) {  //Sole Constructor
        this.num1 = num1;
        this.num2 = num2;
    }
}
class B extends A{
    private int num3;
    public B(int num1, int num2, int num3 ) {
        super( num1, num2 );    //Call to suprt class constructor
        this.num3 = num3;
    }
}
public class Program {
    public static void main(String[] args) {
        B b = new B( 10,20,30);
        b.printRecord();

    }
}
```

toString

- toString is non final / non native method of java.lang.Object class
- Syntax: "public String toString( )"
- Use

```
Employee emp = new Employee( ... );
String str = emp.toString( );
```

- If we want to return state of current instance in String format then we should use toString method.
- If we do not define toString method inside class then its super class toString method gets called.
- Consider source code of toString method from Object class:

```
public String toString() {
    return this.getClass().getName() + "@" +
```

```
Integer.toHexString(this.hashCode());
}
```

- The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object.
- In short, toString method of Object class returns String in following format

```
F.Q.ClassName@Hashcode
```

- According to clients requirement, if implementation of super class method is partially complete then we should override method in sub class.

```
@Override
public String toString() {
    return this.name+" "+this.empid+" "+this.department+"
"+this.designation+" "+this.salary;
}
```

- The result in toString method should be a concise but informative that is easy for a person to read.

```
@Override
    public String toString() {
        return String.format("%-20s%-4d%-10.2f", this.name, this.empid,
this.salary);
    }
```

- It is recommended that all subclasses override toString() method.

equals

- If we want to compare value/state of variable of value type then we should use operator ==.

```
public static void main1(String[] args) {
    int num1 = 10;
    int num2 = 10;
    if( num1 == num2 )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
    //Output : Equal
}
```

- In Java, primitive types are not classes. Hence we can not use equals method to compare state of variable of value type.

```
public static void main(String[] args) {
    int num1 = 10;
    int num2 = 10;
    if( num1.equals( num2 ) )   //Not OK
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
    //Output : Equal
}
```

- We can use operator == with varibale of reference type.
- If we use operator == with variable of reference type then it doesn't compare state of instances rather it compares state of reference variable.

```
class Employee{
    private String name;
    private int empid;
    private float salary;
    public Employee(String name, int empid, float salary) {
        this.name = name;
        this.empid = empid;
        this.salary = salary;
    }
}
public class Program {
    public static void main(String[] args) {
        Employee emp1 = new Employee("Sandeep",33, 45000);
        Employee emp2 = new Employee("Sandeep",33, 45000);
        if( emp1 == emp2 )//Comparing state of references
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output : Not Equal
    }
}
```

- If we want to compare state of instanes then we should use equals method.
- equals is non final and non native method of java.lang.Object class.
- Syntax: "public boolean equals(Object obj)"
- if we do not define equals method inside class then its super class's equals method gets called.
- Consider code of equals method of object class:

```
public boolean equals(Object obj) {
    return (this == obj);
```

```
}
```

- equals method of Object class do not compare state of instances. It compares state of references.

```
public class Program {
    public static void main(String[] args) {
        Employee emp1 = new Employee("Sandeep",33, 45000);
        Employee emp2 = new Employee("Sandeep",33, 45000);
        if( emp1.equals(emp2) ) //Calling Object class's equals method
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Output : Not Equal
    }
}
```

- Inside equals method, if we want to compare state of instances then we should override equals method inside sub class.

```
class Employee {
    private String name;
    private int empid;
    private float salary;
    public Employee(String name, int empid, float salary) {
        this.name = name;
        this.empid = empid;
        this.salary = salary;
    }
    //Employee this = emp1;
    //Object obj = emp2;    //Upcasting
    @Override
    public boolean equals(Object obj) {
        if( obj != null ) {
            Employee other = (Employee) obj;    //Downcasting
            if( this.empid == other.empid )
                return true;
        }
        return false;
    }

}
public class Program {
    public static void main(String[] args) {
        Employee emp1 = new Employee("Sandeep",33, 45000);
        Employee emp2 = new Employee("Sandeep",33, 45000);
        if( emp1.equals(emp2) ) //Calling Employee class's equals method
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
```

```
        //Output : Not Equal
    }
}
```

- In Case of collection framework, to compare state of instances, we should use equals method.

Exception Handling

- Consider code in C/C++

1. Compiler error

```
int main( void )
{
    return 0    //Compiler error – ; missing
}
```

- If we make syntactical mistake in a program then compiler generates error.

2. Linker Error

```
int main( void )
{
    void print( );  //Local Function Declaration;
    print( );   //Function Call     =>  Linker error
    return 0;
}
```

- Without definition if we try to access any member of the class then linker generates error.

3. Bug

```
int main( void )
{
    int number = 10;
    if( number % 2 == 0 )
        printf("Even\n");
    else;   //bug
        printf("Odd\n");
    return 0;
}
```

- Logical error / syntacticaly valid but logical invalid statement is called bug.

4. Exception

- Runtime error is called exception

- It gets generated due to wrong input.

Resource type and resource

- AutoCloseable is interface declared in java.lang package.
- "void close()throws Exception" is a method of AutoCloseable interface(I/F).
- Closeable is interface declared in java.io package.
- "void close()throws IOException" is a method of Closeable interface.
- Consider following code

```
//class Test is Resource Type
class Test implements AutoCloseable{
    public Test() {
    }
    @Override
    public void close() throws Exception {
    }
}
public class Program {
    public static void main(String[] args) {
        Test t = new Test( );
        //Test t;   => reference
        //new Test( );  => Instance : resource
    }
}
```

- In context of exception handling, a class which implements AutoCloseable or Closeable interface is called resource type and its instance is called resource.

Operating system resources for java application

```
1. Thread
2. File
3. Socket
4. Connection
5. IO devices
```

What is exception

- Runtime error is called exception.
- Exception is an instance that is used to send notification to the end user of the system if any exceptional situation/condition occurs in the program.
- Operating system resources are limited hence we should use it carefully. If we want to handle it carefully then we should use exception handling mechanism.

How to handle exception

- Using following keywords, we can handle exception.
    1. try
    2. catch
    3. throw
    4. throws
    5. finally
- Throwable is a class declared in java.lang package.
- 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 this class or one of its subclasses can be the argument type in a catch clause.

Constructors of Throwable class

1. public Throwable()

```
Throwable t1 = new Throwable( );
```

2. public Throwable(String message)

```
Throwable t1 = new Throwable("Message");
```

3. public Throwable(Throwable cause)

```
Throwable cause = new Throwable("Message");
Throwable t1 = new Throwable( cause );
```

4. public Throwable(String message, Throwable cause)

```
Throwable cause = new Throwable();
Throwable t1 = new Throwable( "Message", cause );
```

Methods of Throwable class

1. public String getMessage()
2. public Throwable getCause()
3. public void printStackTrace()

Error

- It is a sub class of java.lang.Throwable class which is declared in java.lang package.
- if runtime error gets generated due to environmental condition then it is called error

- Example:
    - InternalError
    - VirtualMachineError
    - Class OutOfMemoryError
    - StackOverflowError
- We can write catch block to handle error. We can not recover from error hence it is not recommended to write try catch block to handle error.

Exception

- It is a sub class of java.lang.Throwable class which is declared in java.lang package.
- if runtime error gets generated due to application then it is called exception
- Exception
    - NumberFormatException
    - ClassCastException
    - NullPointerException
    - ArrayIndexOutOfBoundsException
- We can write catch block to handle error. We can recover from exception hence it is recommended to write try catch block to handle exception.

Types of exception

```
1. Checked Exception
2. Unchecked Exception
```

- These types are designed for java compiler.

Unchecked Exception

- java.lang.RuntimeException and all its sub classes are considered as unchecked exception classes.
- It is not mandatory to handle unchecked exception.
- Example:
    1. NumberFormatException
    2. ClassCastException
    3. NullPointerException
    4. ArrayIndexOutOfBoundsException

Checked Exception

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

4. IOException

try

- It is a keyword in Java.
- If we want to inspect group statements for excetion then we should use try block/handler.
- Try block must have at least once catch block/finally block/reource.

catch

- It is a keyword in Java.
- If we want to handle exception them we should use catch block/catch handler.
- try block may have multiple catch block.
- Only Throwable class or one of its subclasses can be the argument type in a catch clause.
- Multi catch block can handle multiple specific exception inside single catch block
- Consider Following Code

```
NullPointerException ex1 = new NullPointerException( ); //OK
RuntimeException ex2 = new NullPointerException( ); //OK => Upcasting
Exception ex2 = new NullPointerException( ); //OK => Upcasting
```

- Consider Following Code

```
InterruptedException ex1 = new InterruptedException( ); //OK
Exception ex2 = new InterruptedException( ); //OK
```

- Exception class reference variable can contain reference of checked as well as unchecked exception. Hence to write generic catch block we should use java.lang.Exception.

```
try{
    //TODO
}catch( Exception ex ){ //generic catch blocks
    ex.printStackTrace( );
}
```

throw

- It is a keyword in Java.
- In Java application, JVM/programmer can generate exception. To generate exception, we should use throw keyword.
- 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.
- throw statement is a jump statement.

finally

- It is a keyword in Java.
- If we want to release local resources then we should use finally block.
- JVM always execute finally block.
- try block may have multiple catch block but it can contain only one finally block.

try with resource

```
try( Scanner sc = new Scanner(System.in); ) {   //try with resource
    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 ) {
    ex.printStackTrace();
}
```

throws

- It is a keyword in Java.
- If we want to forward/delegate exception from one method to another method then we should use throws keyword.
    - public static int parseInt(String s) throws NumberFormatException
- If method throws unchecked exception then handling it is optional.
    - public static void sleep(long millis)throws InterruptedException
- If method throws checked exception then handling it is mandatory.

```
class A extends Exception{  }
class B extends Exception{  }
class C extends Exception{  }
class Program{
    //public static void print( )throws A, B, C{    //or
        public static void print( )throws Exception{
        if( expr1 )
            throw new A( );
        else if( expr2 )
            throw new B( );
        else if( expr3 )
            throw new C( );
        else
            //TODO
    }
}
```

Topics For tomorrow

- Custom Exception
- Exception Chaining
- Exception Propagation
- Generics
- Interface

Day 14

Exception Handling

Custom/User Defined Exception

- JVM can not understand exceptional conditions of business logic. If we want to handle it then we should define user defined exception class.
- Custom unchecked exception class

```
class StackOverflowException extends RuntimeException{
    //TODO
}
```

- Custom checked exception class

```
class StackOverflowException extends Exception{
    //TODO
}
```

- Stack is a linear data structure/collection in which, we can store elements in Last In First Out manner(LIFO).
- We can perform following operations on Stack:
    1. boolean empty( ) //index == -1
    2. boolean full( ) //if index == size - 1
    3. void push(int element) //To insert element into stack
    4. int peek( ); //To read topmost value from stack //readonly
    5. void pop( ); //To remove element

Exception Chaining

```
package test;

abstract class A{
    public abstract void print( );
}
class B extends A{
    @Override
    public void print() throws RuntimeException{
        try {
            for( int count = 1; count <= 10; ++ count ) {
                System.out.println("Count   :   "+count);
                Thread.sleep(500);
                if(count  == 5 )
```

```
                    Thread.currentThread().interrupt();
                }
            } catch (InterruptedException cause) {
                throw new RuntimeException(cause);  //Exception Chaining
            }
        }
    }
    public class Program {
        public static void main(String[] args) {
            try {
                A a = new B( ); //Upcasting
                a.print();
            } catch (RuntimeException e) {
                Throwable cause = e.getCause();
                System.out.println(cause);
                //e.printStackTrace();
            }
        }
    }
```

- We can handle exception by throwing new type of exception. It is called as exception chaining.

In the context of exception handling:

1. Bug
    - If runtime errors gets generated due to developers mistake then it is considered as bug in java application.
    - Example:
        1. NullPointerException
        2. ArrayIndexOutOfBoundsException
        3. ClassCastException
    - We should not write try catch block to handle bug.
2. Exception
    - If runtime errors gets generated due to end users/clients mistake then it is considered as exception in java application.
    - Example
        1. ClassNotFoundException
        2. FileNotFoundException
        3. ArithmeticException
    - We should write try catch block to handle exception.
3. Error
    - If runtime errors gets generated due to environmental condition then it is considered as error in java application.
    - Example
        1. OutOfMemoryError
        2. StackOverflowError
        3. InternalError
        4. VirtualMachineError

- It is not recommended to write try-catch block to handle error.

Function Activation Record(FAR)

- Program in execution is called process/running instance of a program is called process.
- Process may contain one or more threads.
- JVM maintains runtime stack per thread. This stack contains stack frame.
- Stack Frame contains data of called function:
  1. Return address
  2. Method Parameters
  3. Method Local variable
  4. Other method calls
  5. temp space of computing.
- This information stored inside stack frame is called function activation record.

Exception Propagation

- During execution, if exception occurs then, first call stack gets destroyed and then control is retured to the calling method. This process is called exception propagation.

Assertion : Home Work

Boxing

- Consider example
  - int is primitive/value type
  - String is class hence it is non primitive/reference type

```
int number = 10;
String strNumber = String.valueOf( number );    //Boxing
```

- Process of converting, value of variable of primitive type into non primitive type is called boxing.

Auto-Boxing

- If boxing is done implicitly then it is called auto-boxing.

```
int number = 10;
Object obj = number;    //OK : Auto-Boxing

//Integer i = Integer.valueOf( number );    //Auto-Boxing
//Object obj = i;   //Upcasting
```

UnBoxing

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

- Process of converting, state of instance of non primitive type into primitive type is called unboxing.

Auto-UnBoxing

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

```
Integer i1 = new Integer(125);
//int i2 = i1.intValue();   //UnBoxing
int i2 = i1;    //Auto-UnBoxing
```

Generic Programming

- If we want to write generic code in java then we can use:
    1. java.lang.Object class
    2. Generics
- Generic code using java.lang.Object

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

- Storing primitive value inside Box instane.

```
public static void main2(String[] args) {
    Box box = new Box();    //OK
    box.setObject(125); //box.setObject(Integer.valueOf(125));
}
```

- Storing non primitive value inside Box instane.

```
public static void main3(String[] args) {
    Box box = new Box();    //OK
    box.setObject( new Date( ) );   //OK
}
```

- Using Object class we can write generic code but we can not write typesafe generic code.

```
public static void main(String[] args) {
    Box box = new Box();    //OK
    box.setObject( new Date( ) );   //OK
    String str =  (String) box.getObject(); //Downcasting :
ClassCastException
}
```

- If we want to write type-safe generic code then we should use generics.

Generics

- By passing data type as a argument, we can write generic code in java hence parameterized type is also called generics.

```
//Parameterized Class/Type
class Box<T>{   //T =>  Type Parameter
    private T object;
    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();
        System.out.println(date);
    }
}
```

- Why Generics?
    1. It gives us stronger type checking at compile time. In other words, we can write type safe code in Java.
    2. It completly eleminates need explict type casting.
    3. We can write generic algorithm and data structure which reduces develpers effort.
- Type Inference:

```
Box<Date> box = new Box<Date>( );   //OK
Box<Date> box = new Box<>( );   //Type Inference
//Type argument of instane will be infered from reference.
```

- An ability of Java compiler to decide type argument of instance by looking toward type argument of reference is called type inference.
- Raw Type:

```
Box box = new Box( );   //Box=> Raw Type
//Box<Object> box = new Box<>( );
```

- If we use parameterized type w/o passing type argument then it is called raw type. In this case java.lang.Object is considered as default type argument.
- During instantiation of parameterized type, type argument must be reference type.
    - int : primitive type / value type
    - Integer : non primitive type / reference type

```
//Box<int> box = new Box<>( );  //int => NOT OK
Box<Integer> box = new Box<>( );    //Integer => OK
```

- If we want to store primitive values inside instance of parameterized type then type argument must be Wrapper Class.
- Use of Wrapper:
    1. For parsing(Converting state of string into primitive values)
        - parseXXX( );
    2. To use as a type argument in parameterized type.
        - Box box = null;
- Wrapper Class Hierarcy:
    - java.lang.Object
        - java.lang.Boolean
        - java.lang.Character
        - java.lang.Number
            - java.lang.Byte
            - java.lang.Short
            - java.lang.Integer
            - java.lang.Long
            - java.lang.Float
            - java.lang.Double
- Commonly used type parameter names in generics:
    1. T : Type
    2. N : Type of Number
    3. E : Type of Element

4. K : Type of Key

5. V : Type Of Value

6. S,U, V : Second Type Parameters

- Specifying type parameter is a job of Type/class implementor and mentioning Type argument is a job of class user.
- It is possible to pass multiple type arguments:

```
interface Pair<K, V>{
    K getKey( );
    V getValue( );
}
class Dictionary<K, V> implements Pair<K,V>{
    private K key;
    private V value;
    public Dictionary(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return key;
    }
    @Override
    public V getValue() {
        return this.value;
    }
}
public class Program {
    public static void main(String[] args) {
        Pair<Integer, String> p = new Dictionary<>(1, "DAC");
        System.out.println("Key :   "+p.getKey());
        System.out.println("Value   :   "+p.getValue());
    }
}
```

Bounded Type Parameter

- If we want to put restriction on data type that can be used as type argument then we should specify bounded type parameter.
- Class implementor is responsible for mentioning bounded type parameter.

```
class Box<T extends Number >{   // T is bounded type parameter
    private T object;
    public T getObject() {
        return object;
    }
    public void setObject(T object) {
```

```
        this.object = object;
    }
}
public class Program {
    public static void main(String[] args) {
        Box<Number> b1 = new Box<>();   //OK
        Box<Integer> b2 = new Box<>();  //OK
        Box<Double> b3 = new Box<>();   //OK
        Box<Boolean> b4 = new Box<>();  //Not OK
        Box<String> b5 = new Box<>();   //Not OK
        Box<Date> b6 = new Box<>();     //Not OK
    }
}
```

- ArrayList is resizable array declared in java.util package

```
public class Program {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>( );
        list.add(10);
        list.add(20);
        list.add(30);
        for( Integer element : list )
            System.out.println(element);
    }
}
```

```
    public static ArrayList<Integer>getIntegerList( ){
        ArrayList<Integer> list = new ArrayList<>( );
        list.add(10);
        list.add(20);
        list.add(30);
        return list;
    }
    public static ArrayList<Double>getDoubleList( ){
        ArrayList<Double> list = new ArrayList<>( );
        list.add(10.1);
        list.add(20.2);
        list.add(30.3);
        return list;
    }
    public static ArrayList<String>getStringList( ){
        ArrayList<String> list = new ArrayList<>( );
        list.add("DAC");
        list.add("DMC");
        list.add("DESD");
        return list;
    }
```

Wild Card

- In generics, "?" is called wild card which represent 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) {
        for( Object element : list )
            System.out.println(element);
    }
```

- In above code, list will contain reference of ArrayList which can contain unknown/any type of element.

```
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printList( integerList );   //OK

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

    ArrayList<String> stringList = Program.getStringList();
    Program.printList(stringList);  //OK
```

Upper bounded Wild Card

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

- In above code, list will contain reference of ArrayList which can contain Number and its sub type of elements only.

```
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printList( integerList );   //OK

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

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

Lower bounded Wild Card

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

- In above code, list will contain reference of ArrayList which can contain Integer and its super type of element.

```
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printList( integerList );   //OK

    ArrayList<Double> doubleList = Program.getDoubleList();
    Program.printList( doubleList );    //NOT OK

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

Generic Method:

```
    public static <T> void print( T obj ) {
        System.out.println(obj);
    }
    public static void main(String[] args) {
        Program.print(true);    //OK
        Program.print('A');    //OK
        Program.print(123);    //OK
        Program.print(456.78);    //OK
        Program.print("Java");    //OK
        Program.print(new Date());    //OK
    }
```

```
    public static <T extends Number> void print( T obj ) {
        System.out.println(obj);
    }
    public static void main(String[] args) {
        Program.print(true);    //Not OK
        Program.print('A'); //Not OK
        Program.print(123); //OK
        Program.print(456.78);  //OK
        Program.print("Java");  //Not OK
```

```
        Program.print(new Date());  //Not OK
    }
```

Restrictions on Generics

Link : https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html

Type Erasure

Link : https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Assignment:

```
- Write a LinkedList in Java.
```

Day 15

Fragile Base Class Base Problem

- If we make changes in the super class then it necessary to compile super class as well as all its sub classes. It is called as fragile base class problem.

Interface

- Set of rules is called as specification/standard.
- If we want to define specification for the sub classes then we should use interface in Java.
- We should use interface because:
    1. It helps to build trust between service provider and service consumer.
    2. It helps to minimize vendor dependancy. In other words, we can achieve loose coupling.
- Reference types in Java
    1. Interface
    2. Class
    3. Enum
    4. Array
- Interface is non primitive/reference type in Java.
- interface is keyword in Java.
- Syntax

```
interface Printable{
    //TODO
}
```

- We can not instantiate interface but we can create reference it.

```
public static void main(String[] args) {
    printable p = null; //reference //OK
    p = new printable( );   //Not OK
}
```

- In Java, inside interface we can declare:
    1. Nested interface
    2. Field
    3. Abstract method
    4. Default method
    5. Static method
- We can define interface inside interface. It is called nested interface.

```
package java.util;
public interface Map<K,V>{
    public static interface Entry<K,V>{    //Nested interface
        K getKey( );
        V getValue( );
        V setValue( V value );
    }
}
```

- We can declare/define methods inside interface(I/F) but we can not declare/define constructor(ctor) inside interface.
- We can declare fields inside interface. It is by default "public static final".

```
interface A{
    int number = 10;
    //public static final int number = 10;
}
public class Program {
    public static void main(String[] args) {
        System.out.println(A.number);
    }
}
```

- Consider another example

```
javap java.sql.ResultSet    (press enter )
```

```
package java.sql;
interface ResultSet{
    public static final int HOLD_CURSORS_OVER_COMMIT = 1;
    public static final int CLOSE_CURSORS_AT_COMMIT = 2;
    public static final int FETCH_FORWARD = 1000;
    public static final int FETCH_REVERSE = 1001;
    public static final int FETCH_UNKNOWN = 1002;
    public static final int TYPE_FORWARD_ONLY = 1003;
    public static final int TYPE_SCROLL_INSENSITIVE = 1004;
    public static final int TYPE_SCROLL_SENSITIVE = 1005;
    public static final int CONCUR_READ_ONLY = 1007;
    public static final int CONCUR_UPDATABLE = 1008;
}
```

- We can write method inside interface. It is by default considered as public and abstrasct.

```

```

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

```
package java.util;
public interface Enumeration{
    boolean hasMoreElements();
    E nextElement();
}
```

- Using implements keyword, we can implement interface for the class.
- It is mandatory to implement all the abstract methods of interface in sub class otherwise sub class can be considered as abstract.
- Solution 1:

```
interface A{
    int number = 10;    //public static final int number = 10;
    void print( );      //public abstract void print( ) ;
}
abstract class B implements A{
}
```

- Solution 2:

```
interface A{    //ISI
    int number = 10;
    //public static final int number = 10;
    void print( ) ;
    //public abstract void print( ) ;
}
class B implements A{   //Service Provider
    @Override
    public void print() {
        System.out.println("Number  :   "+A.number);
    }
}
```

```
public class Program {  //Service Consumer
    public static void main(String[] args) {
        B b = new B( );
        b.print();  //OK : 10

        A a = new B( ); //Upcasting : Recommended
```

```
        a.print();  //OK : 10
    }
}
```

Choose correct syntax

- 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 ; //OK
13. C2 extends C1 implements I1, I2 ; //OK

- Conclusion:
    1. Interace extends interface.
    2. Interface can extend more than one interfaces. In other words, Java support to multiple interface inheritance.
    3. Super type of interface must be interface.
    4. Class do not extend interface rather it implements interface. It is called interface implementation inheritance.
    5. Class can implement more than one interfaces. In other words, Java support to multiple interface implementation inheritance.
    6. Class can extend another class. It is called implementation inheritance.
    7. Class can not extends more than one class. In other words, Java do not support to multiple implementation inheritance.

Interface fields

```
interface A{
    int num1 = 10;
    int num4 = 40;
    int num5 = 70;
}
interface B{
    int num2 = 20;
    int num4 = 50;
    int num5 = 80;
```

```
}
interface C extends A, B{   //OK : Multiple Interface Inheritance.
    int num3 = 30;
    int num4 = 60;
}
public class Program {
    public static void main(String[] args) {
        System.out.println("A.Num5  :   "+A.num5);  //OK    :   70
        System.out.println("B.Num5  :   "+B.num5);  //OK    :   80
        System.out.println("C.Num5  :   "+C.num5);  //NOT OK    :   The
field C.num5 is ambiguous
    }
    public static void main4(String[] args) {
        System.out.println("A.Num4  :   "+A.num4);  //OK    :   40
        System.out.println("B.Num4  :   "+B.num4);  //OK    :   50
        System.out.println("C.Num4  :   "+C.num4);  //OK    :   60
    }
    public static void main3(String[] args) {
        System.out.println("Num3    :   "+C.num3);  //OK    :   30
    }
    public static void main2(String[] args) {
        System.out.println("Num2    :   "+B.num2);  //OK    :   20
        System.out.println("Num2    :   "+C.num2);  //OK    :   20
    }
    public static void main1(String[] args) {
        System.out.println("Num1    :   "+A.num1);  //OK    :   10
        System.out.println("Num1    :   "+C.num1);  //OK    :   10
    }
}
```

Interface Abstract Methods

```
interface A{
    void f1( );
    void f3( );
}
interface B{
    void f2( );
    void f3( );
}
abstract class C{
    public abstract void f3( );
}
class D extends C implements A, B{
    @Override
    public void f1() {
        System.out.println("D.f1");
    }
    @Override
    public void f2() {
        System.out.println("D.f2");
```

```
    }
    @Override
    public void f3() {
        System.out.println("D.f3");
    }
}
public class Program {
    public static void main(String[] args) {
        A a = new D( );
        a.f1();
        a.f3();

        B b = new D( );
        b.f2();
        b.f3();

        C c = new D( );
        c.f3();
    }
}
```

Abstract Method versus default method.

- It is mandatory to override abstract method of I/F inside sub class.
- We can not provide body to the abstract method.
- It is optional to override default method of I/F inside sub class.
- It is madatory to provide body to the default method.
- Using "InterfaceName.super.methodName()", we can use default method of I/F inside sub class.
- Interface static methods are helper method which are not designed to override rather it is designed to help to default method and sub class methods.

```
interface A{
    default void f1( ) {
        System.out.println("A.f1");
    }
}
interface B{
    default void f1( ) {
        System.out.println("B.f1");
    }
}
class C implements A, B{
    @Override
    public void f1() {
        System.out.println("C.f1");
    }
}
public class Program {
    public static void main(String[] args) {
        A a = new C( );
```

```
        a.f1();

        B b = new C( );
        b.f1();
    }
}
```

- If name of default method of super I/F is same then overriding that method is mandatory.
- An interface which contains Single Abstract Method(SAM) is called Functional Interface.
- Functional Interface is also called SAM interface.
- It can contains:
    1. multiple default methods
    2. multiple static methods
    3. Single abstract method.

```
interface A{    //OK
    void f1( );
}
```

- If we want to ensure whether interface is Functional or not then we should use @FunctionalInterface annotation.

```
@FunctionalInterface
interface A{    //OK
    void f1( );//functional method or method descriptor.
}
```

- Abstract method defined inside functional interface is called functional method or method descriptor.
- java.util.function package contains all SUN/ORACLE supplied functional interface.
- Functional interfaces declared in java.util.function package are:
    1. Predicate
        - boolean test(T t)
    2. Consumer
        - void accept(T t)
    3. Supplier
        - T get()
    4. Function<T,R>
        - R apply(T t)
    5. UnaryOperator

Following are commonly used interfaces in Core Java:

```
1. java.lang.AutoCloseable
2. java.lang.Cloneable
```

```
3. java.lang.Iterable
4. java.util.Iterator
5. java.lang.Comparable
6. java.util.Comparator
7. java.io.Serializable
```

Comparable and Comparator Interface implementation

- if we want to sort array then we should use Arrays.sort() method.
- In case of primitive types, Arrays.sort( ) implicitly use Dual-Pivot Quicksort algorithm.

```
public class Program {
    private static void print(int[] arr) {
        if( arr != null )
            System.out.println(Arrays.toString(arr));
        System.out.println();
    }
    public static void main(String[] args) {
        int[] arr = new int[] { 5, 1, 4, 2, 3 };
        Program.print( arr );   //[5, 1, 4, 2, 3]
        Arrays.sort(arr);   //Dual-Pivot Quicksort
        Program.print( arr );   //[1, 2, 3, 4, 5]
    }
}
```

```
Employee[] arr = new Employee[ 14 ];

arr[ 0 ] = new Employee(7369,"SMITH","CLERK","1980-12-
17",800.00f,"RESEARCH");

arr[ 1 ] = new Employee(7499,"ALLEN","SALESMAN","1981-02-
20",1600.00f,"SALES");

arr[ 2 ] = new Employee(7521,"WARD","SALESMAN","1981-02-
22",1250.00f,"SALES");

arr[ 3 ] = new Employee(7566,"JONES","MANAGER","1981-04-
02",2975.00f,"RESEARCH");

arr[ 4 ] = new Employee(7654,"MARTIN","SALESMAN","1981-09-
28",1250.00f,"SALES");

arr[ 5 ] = new Employee(7698,"BLAKE","MANAGER","1981-05-
01",2850.00f,"SALES");

arr[ 6 ] = new Employee(7782,"CLARK","MANAGER","1981-06-
09",2450.00f,"ACCOUNTING");
```

```
arr[ 7 ] = new Employee(7788,"SCOTT","ANALYST","1982-12-
09",3000.00f,"RESEARCH");

arr[ 8 ] = new Employee(7839,"KING","PRESIDENT","1981-11-
17",5000.00f,"ACCOUNTING");

arr[ 9 ] = new Employee(7844,"TURNER","SALESMAN","1981-09-
08",1500.00f,"SALES");

arr[ 10 ] = new Employee(7876,"ADAMS","CLERK","1983-01-
12",1100.00f,"RESEARCH");

arr[ 11 ] = new Employee(7900,"JAMES","CLERK","1981-12-
03",950.00f,"SALES");

arr[ 12 ] = new Employee(7902,"FORD","ANALYST","1981-12-
03",3000.00f,"RESEARCH");

arr[ 13 ] = new Employee(7934,"MILLER","CLERK","1982-01-
23",1300.00f,"ACCOUNTING");
```

- Comparable is an interface declared in java.lang package.
- "int compareTo(T other)" is a method of java.lang.Comparable I/F.
- If we want to sort array of instances of same class then class must implementation java.lang.Comparable interface.
- return type of compareTo method is integer.
    1. If state of current instance is less than specified instance then we should return any -ve number(-1).
    2. If state of current instance is greater than specified instance then we should return any +ve number(+1).
    3. If state of current instance is equal to specified instance then we should return 0.
- Comparator is interface declared in java.util package.
- "int compare(T o1,T o2)" is a method of java.util.Comparator interface.
- If we want to sort array of instances of different class then we should implement java.util.Comparator interface.
- return type of compare method is integer.

Day 16

LinkedList

- LinkedList is collection of elements where each element is called as node. In short LinkedList is collection of nodes.
- Node is object/instance which may contain either 2 parts / 3 parts depending on type of LinkedList.
- Types of LinkedList:
    1. Singly LinkedList
        - In a LinkedList, if node contains 2 parts:
            1. data
            2. a reference variable which contains reference of next node instance is called Singly LinkedList.
    2. Doubly LinkedList
        - In a LinkedList, if node contains 3 parts:
            1. a reference variable which contains reference of prev node instance
            2. data
            3. a reference variable which contains reference of next node instance is called Doubly LinkedList.

Iterable

- In foreach loop source can be:
    1. Array
    2. Instance of a class which implements Iterable interface.
- Iterable is an interface declared in java.lang package.
- Implementing this interface allows an object to be the target of the "for-each loop" statement.
- It is introduced in JDK 1.5
- Methods of java.lang.Iterable
    1. Iterator iterator()
    2. default Spliterator spliterator()
    3. default void forEach(Consumer<? super T> action)
- Consumer is a function interface declared in java.util.function package.
- "void accept(T t)" is a functional method of Consumer.

Iterator

- It is interface declared in java.util package.
- It is introduced JDK 1.2
- Methods of Iterator interface:
    1. boolean hasNext()
    2. E next()
    3. default void remove()
    4. default void forEachRemaining(Consumer<? super E> action)
- Conclusion : If class implements Iterator interface means it is allowed to traverse in foreach loop.

Shallow Copy

- Process of copying state of variable into another variable as it is, is called shallow copy.
- Shallow copy is also called as bitwise/bit-by-bit copy.

```
int num1 = 10;
int num2 = num1;    //Shallow Copy
System.out.println(num1);   //10
System.out.println(num2);   //10
```

```
Date dt1 = new Date( 23, 7, 1983 );
Date dt2 = dt1; //Shallow Copy of references
```

Collection Framework

- If we want to manage data efficiently in RAM then we should data structure.
- In Java, data structure classes are called collection collection classes.
- Example: Stack, Queue, LinkedList etc. are collections.
- 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 core java application.
- In Java collection is not a collection of instances rather it is collection of references.
- To use collection framework, we should import java.util package.
- Link : https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html
- Collection Framework Interface Hierarchy
    - java.lang.Iterable
        - java.util.Collection
            - java.util.List
            - java.util.Queue
                - java.util.Deque
            - java.util.Set
                - java.util.SortedSet
                    - java.util.NavigableSet
    - java.util.Map<K,V>
        - java.util.SortedMap<K,V>
            - java.util.NavigableMap<K,V>

Day 17

Nested Class

- We can define class inside scope of another class. It is called as nested class.
- Consider Example:

```
class Node{
    //TODO : Write code here
}
class LinkedList{
    private Node head;
    private Node tail;
    public void addLast( int element ){
        Node newNode = new Node( element );
        //TODO : Write code here
    }
    public void print( ){
        Node trav = this.head;
        //TODO : Write code here
    }
}
```

```
class LinkedList{   //Top Level Class
    class Node{ //Nested class
        //TODO : Write code here
    }
    private Node head;
    private Node tail;
    public void addLast( int element ){
        Node newNode = new Node( element );
        //TODO : Write code here
    }
    public void print( ){
        Node trav = this.head;
        //TODO : Write code here
    }
}
```

- By defining nested class, we implement encapsulation.

```
//Top Level class
class Outer{    //Outer.class
    //Nested class
    class Inner{    //Outer$Inner.class
```

```
    }
}
```

- Access modifier of top level class can be package level private or public only.
- Access modifier of nested class can be private, package level private, protected or public.

Types of nested class:

```
1. Non static nested class / Inner class
2. Static nested class
```

```
class LinkedList implements Iterable<Integer>{
    //Static nested class
    class Node{
        int data;
        Node next;
    }

    private Node head;
    private Node tail;
    public Iterator<Integer> iterator( ){
        Iterator<Integer> itr = new LinkedListIterator( this.head );
        return itr;
    }
    //Non static nested class
    class LinkedListIterator implements Iterator<Integer>{
        private Node trav;
        public LinkedListIterator( Node head ){
            this.trav = head;
        }
    }
}
```

Non Static Nested class.

- Non Static Nested class is also called as inner class.
- If implementation of nested class depends on top level class then nested class should be non static.
- In above code, LinkedListIterator depends on LinkedListIterator hence it should be non static.

```
//Top Level class
class Outer{    //Outer.class
    //Nested class
    public class Inner{    //Outer$Inner.class

    }
}
```

- Hint : For simplicity, consider non static nested class as a non static method of a class.
- Instantiation of top level class:

```
Outer out = new Outer( );
```

- Instantiation of non static nested class:
- First way

```
Outer out = new Outer( );
Outer.Inner in = out.new Inner( );
```

- Second way

```
Outer.Inner in = new Outer( ).new Inner( );
```

Static Nested class.

- When we declared nested class static then it is simply called as static nested class.
- If implementation of nested class do not depends on top level class then nested class should be static.
- In above code, Node do not depends on LinkedListIterator hence it should be static.

```
//Top Level class
class Outer{    //Outer.class
    //Nested class
    public static class Inner{    //Outer$Inner.class

    }
}
```

- We can not declare top level class static but we can declare nested class static.
- Hint : For simplicity, consider static nested class as a static method of a class.
- Instantiation of top level class:

```
Outer out = new Outer( );
```

- Instantiation of static nested class:

```
Outer.Inner in = new Outer.Inner( );
```

Day 18

Local Class

- We can define class inside scope of another method. It is called local class/method local class.

```
class Program{  //Program.class
    public static void main(String[] args) {
        //Method Local Class
        class Complex{  //Program$1Complex.class

        }
    }
}
```

- We can not use reference/instance of method local class outside method.
- Types of Local class:
  1. Method Local Inner class
  2. Method Local Anonymous Inner class #### Method Local Inner class
- In Java, we can not declare local variable/class static.
- Non static, method local class is also called as method local inner class.

```
public class Program {
    public static void main(String[] args) {
        class Complex{
            private int real = 10, imag = 20;
            @Override
            public String toString() {
                return "Complex [real=" + real + ", imag=" + imag + "]";
            }
        }
        Complex c1 = null;
        c1 = new Complex();
        System.out.println(c1.toString());
    }
}
```

Method Local Anonymous Inner class

- In Java, we can define class without name. It is called annonymous class.
- In Java, we can create annonymous class inside method only hence it is called as method local annonymous class.
- In Java, we can not declare local class static. Hence annonymous class is also called as metod local annonymous inner class.
- Consider following syntax:

```
Object obj = null;  //object reference / reference
```

```
new Object( );  //Anonymous instance of class java.lang.Object

Object obj = new Object( );  //Instance with reference
```

- Anonymous inner class using concrete class

```
public class Program {  //Program.class
    public static void main(String[] args) {
        Object obj = new Object() { //Program$1.class
            private String message = "Hello";
            @Override
            public String toString() {
                return this.message;
            }
        };
        System.out.println(obj.toString());
    }
}
```

- Anonymous inner class using abstract class:

```
abstract class Shape{
    protected double area;
    public abstract void calcuateArea( );
    public double getArea() {
        return area;
    }
}
public class Program {  //Program.class
    public static void main(String[] args) {
        Shape sh = new Shape() {
            private double radius = 10;
            @Override
            public void calcuateArea() {
                this.area = Math.PI * Math.pow(this.radius, 2);
            }
        };

        sh.calcuateArea();
        System.out.println("Area    :   "+sh.getArea());
    }
}
```

- Anonymous inner class using interface:

```
interface Printable{
    void print( );
}
public class Program {  //Program.class
```

```
    public static void main(String[] args) {
        Printable p = new Printable() {
            private String message = "Good Morning";
            @Override
            public void print() {
                System.out.println(message);
            }
        };
        p.print();
    }
}
```

Functional Programming

- Book : Java 8 In Action.
- POP : It is programming methodology in which we can solve real world problems using global function and structure.
- OOP : It is programming methodology in which we can solve real world problems using class and object.
- Funtional Programming : It is programming methodology in which we can solve real world problems by chaining/linking functions(pipeline of function).
- Key features in Java 8 Functinoal Programming:
  1. Interface Default Method
  2. Static interface method
  3. Functional Interface
  4. Lambda expression
  5. Method reference
  6. Java 8 streams. ### Lambda Expression
- Expression is a statement which may contain:
  1. Literals / constants
  2. Variables
  3. Operators
- Example:

```
int num1 = 10, num2 = 20;
int result = num1 + num2;    //Arithmetic Expression
```

- "->" operator is called lambda operator. Its meaning is "goes to".
- If expression contains lambda operator then such expression is called lambda expression.
- Syntax:
  - FunctionalInterface ref = (Input params)-> { Lambda Body };
  - Input parameters goes to lambda body.
  - If lambda body contains single statement then use of { } is optional.
- Lambda expresion is also called as Anonymous method.
- Abstract method of Functinoal interface is called Functinoal method /

method descriptor.
- To implement functional interface we should define lambda expression
- If interface contains single abstrasct method then it is called functional interface.

```
@FunctionalInterface
interface Printable{
    void print( );  //Functional method / Method descriptor
}

Printable p = ( )-> System.out.println("Inside Lambda Expresion");
p.print();
```

- Consider another example:

```
@FunctionalInterface
interface Printable{
    void print( String message );   //Functional method / Method descriptor
}
public class Program {  //Program.class
    public static void main(String[] args) {
        //Printable p = ( String msg )-> System.out.print(msg);
        //Printable p = ( String message )-> System.out.print(message);
        //Printable p = ( message )-> System.out.print(message);
        Printable p = message -> System.out.print(message);
        p.print( "Have a nice day." );
    }
}
```

- Consider another example

```
@FunctionalInterface
interface IMath{
    int sum( int num1, int num2 );
}
public class Program {  //Program.class
    public static void main(String[] args) {
        IMath m = ( num1, num2 )-> num1 + num2;
        int result = m.sum(10, 20);
        System.out.println("Result  :   "+result);
    }
}
```

- Consider another example:

```
@FunctionalInterface
interface IMath{
    int factorial( int number );
}
public class Program {  //Program.class
```

```
    public static void main(String[] args) {
        IMath m = number ->{
            int fact = 1;
            for( int count = 1; count <= number; ++ count ) {
                fact = fact * count;
            }
            return fact;
        };

        int result = m.factorial(5);
        System.out.println("Result  :   "+result);
    }
}
```

- Consider example

```
public class Program {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>( );
        list.add(10);
        list.add(20);
        list.add(30);

        //Consumer<Integer> action = ( Integer number)-> System.out.println(number);
        //list.forEach(action);

        list.forEach( ( Integer number)-> System.out.println(number) );
    }
}
```

Method reference

```
@FunctionalInterface
interface Printable{
    void print( );
}
public class Program {
    public static void showRecord( ) {
        System.out.println("Inside showRecord() method");
    }
    public void displayRecord( ) {
        System.out.println("Inside displayRecord() method");
    }
    public static void main(String[] args) {
        Printable p1 = Program::showRecord;
        p1.print();
```

```
        Program prog = new Program();
        Printable p2 = prog::displayRecord;
        p2.print();

        Printable p3 = System.out::println;
        p3.print();

    }
}
```

- Consider Example:

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>( );
    list.add(10);
    list.add(20);
    list.add(30);

    list.forEach( System.out::println );
}
```

String Handling

- In Java, string is a collection of character object.
- If we want to manipulate String then we can use:
  1. java.lang.String
  2. java.lang.StringBuffer
  3. java.lang.StringBuilder
  4. java.util.StringTokenizer #### String
- It is a final class declared in java.lang package.
- It implements Serializable, Comparable, CharSequence interface.
- String is not buit-in / primitive type in Java. It is a class hence it is considered as reference type.
- We can create String instance with and without new operator.
- Consider example:

```
String s1 = new String("SunBeam");  //OK : String Instance => Heap

String s2 = "SunBeam";  //OK : String Literal => String Literal Pool
```

- String s2 = "SunBeam" is equivalent to

```
char[] data = {'S','u','n','B','e','a','m'};
String str = new String( data );
```

- String objects are constant. If we try to modify its state then new string instane gets created. In other words, String objects are immutable objects.
- If we want to concatenate strings then we should use "concat()" method.

- If we want to concatenate string and variable of primitive/non primitive type then we should use operator.

StringBuffer and StringBuilder

- Both are final classes declared in java.lang package.
- Both are used to create mutable String object.
- It is mandatory to use new operator to create their instance.
- Even though both are final, equals and hashCode() method is not overriden inside it.
- StringBuffer is thread-safe/synchronized whereas StringBuilder is unsynchronized.
- StringBuilder is introduced in JDK 1.0 and StringBuffer is introduced on JDK 1.

Reflection