

# Object Oriented Programming with Java

sandeepkulange@gmail.com

## Evaluation Strategy

- **Duration** : 104 Hours( 48 Theory + 48 Lab + 8 Revision Hours )
- **Evaluation** : Total 100 marks
- **Theory Exam** : 40%
- **Lab Exam** : 40%
- **Internals** : 20%
- **Reference Books** :
  1. Java, The Complete Reference – Herbert Schildt
  2. Core Java Volume I and II – Cay S. Horstmann
  3. Java 8 Programming Black Book – D.T. Editorial Services
  4. Object Oriented Analysis and Design with Applications – Grady Booch
- **Reference links** :
  1. <https://docs.oracle.com/javase/tutorial/>
  2. <https://dev.java/>
  3. <https://docs.oracle.com/javase/8/docs/api/>

## Agenda

---

- Language, Technology and Platform
- History of Java programming language
- Java version history
- Java Software Development Kit
- Simple "Hello World!!" application
- Exploring bytecode using javap
- Java comments
- Java entry point
- Primitive & non primitive types,
- Wrapper classes
- Command line arguments

## Classification of Programming Languages

---

- **Machine level programming language**
  - Binary language
- **Low level programming language**
  - Assembly programming language
- **High level programming language**
  - Procedure oriented programming languages
    - Algol, Fortran, COBOL, Pascal, C etc.
  - Object oriented programming languages
    - Simula, Smalltalk, C++, Java, C#, Go, Python etc.
  - Object based programming languages
    - Ada, Java Script, Action Script, VB Script, Visual Basic etc.
  - Functional programming languages
    - Haskell, Lisp, F#, Scala, Clojure, Ocaml, Python, C#, C++, Java

## “Hello World” Application using Machine Language

- Here is the “Hello World!!” program in binary:

```
01001000 01100101 01101100 01101100  
01101111 00100000 01010111 01101111  
01110010 01101100 01100100 00100001  
00100001
```

- Advantages of machine-level language:

1. Very fast execution times
2. Complete control over the computer's hardware
3. No need for a compiler or interpreter

- Limitations of machine-level language

1. Very difficult to read and write
2. Specific to a particular computer architecture, making it non-portable
3. Requires a deep understanding of the underlying hardware

## “Hello World” Application using Low Level Language

- Here is the “Hello World!!” program in x86 assembly language:

```
section .data  
msg db 'Hello World!!',0  
  
section .text  
global _start  
  
_start:  
    ; Write the string to stdout  
    mov eax, 4      ; System call for "write"  
    mov ebx, 1      ; File descriptor for stdout  
    mov ecx, msg    ; Pointer to the message  
    mov edx, 13     ; Length of the message  
    int 0x80        ; Call the kernel  
  
    ; Exit the program with a status of 0  
    mov eax, 1      ; System call for "exit"  
    xor ebx, ebx    ; Status code (0 = success)  
    int 0x80        ; Call the kernel
```

## Advantages and Limitations of Low Level Language

---

- Advantages of low-level languages include:
  1. Faster execution times than high-level languages
  2. Direct access to the computer's hardware
  3. More portable than machine-level language
- Limitations of low-level languages include:
  1. Still difficult to read and write compared to high-level languages
  2. Can be error-prone due to the need for manual memory management
  3. Not as user-friendly as high-level languages

## “Hello World” Application using High Level Language

---

- Here is the “Hello World!!” program in Python:

```
print("Hello, world!")
```
- Advantages of high-level language:
  1. Easy to read and write
  2. Less error-prone due to built-in error checking and automatic memory management
  3. More portable than machine-level and low-level languages
- Limitations of high-level language
  1. Slower execution times than low-level languages
  2. Less control over the computer's hardware than low-level languages
  3. More memory-intensive than low-level languages

## What is Language?

- Language is a system of communication that allows humans to convey meaning to each other through the use of symbols, such as words, gestures, and facial expressions.
- In the context of computer science, programming languages are used to write software programs that can be executed by computers.
- In General programming languages contains:
  - Syntax and Semantics
  - Tokens(Identifier, Keyword, Constant, operator, Separator)
  - Data Types
  - Comments
  - Built in unique features
- Even though languages are used to implement business logic, they can be used to create applications too:
  - Console User Interface(CUI) / Command Line Interface(CLI) Application
  - Graphical User Interface(GUI) application
  - Library Application(.lib/.dll/.jar)
- Example of Languages:
  - C, C++, Java, C#, Python, GO etc.

## What is Technology?

- Technology refers to the tools and techniques used to create and manipulate information. In Simple words, we can develop application using technology.
- Example:
  - Java
  - Artificial Intelligence (AI)
  - Blockchain
  - Augmented Reality (AR)
  - Internet of Things (IoT)
  - Cloud Computing
- Note: Every language can be considered as technology but every technology can not be considered as language.

## What is Platform?

- A *platform* is the hardware or software environment in which a program runs.
- Most platforms can be described as a combination of the operating system and underlying hardware:
  - Microsoft Windows
  - Linux
  - Mac OS
  - Android
- Software-only platforms are the applications that runs on top of other hardware-based platforms:
  - Java
  - Microsoft .NET

## History of Java Programming Language

- Java is high-level, statically type checked, Object-oriented as well as Functional programming language.
- Java language is both technology as well as platform.
- Code name of the Java was "Green" Project.
- Java was developed by ("Green Team") James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, which was later acquired by Oracle Corporation(2010).
- The name "Green" was chosen because the team wanted to emphasize that their goal was to create an environmentally-friendly language that would allow programs to run efficiently on a variety of devices, including small handheld devices and large servers.
- Green Team started working on "Green Project" in 1991 and first public version released in 1996.
- In fact, the original name for Java was "Oak", but it was changed to "Java" because another company had already registered the name "Oak" for their product.
- Java's slogan is "**Write Once, Run Anywhere**" (sometimes abbreviated as "WORA").
- The standardization of the Java language is overseen by the **Java Community Process (JCP)**.
- "Star" is the first project delivered by Green Team.
  - An extremely intelligent TV remote control.
- Reference to read a short history of Java: Core Java Volume I

Java Version History					
(https://en.wikipedia.org/wiki/Java_version_history)					
Version	class file format version[8]	Release date	End of Free Public Updates[9][10][11][12][13][14][15]	Extended Support Until	
JDK 1.0	45	23rd January 1996	May 1996	—	
JDK 1.1	45.3	2nd February 1997	October 2002	?	
J2SE 1.2	46	4th December 1998	September 2003	?	
J2SE 1.3	47	8th May 2000	October 2010	?	
J2SE 1.4	48	13th February 2002	October 2008	February 2013	
Java SE 5	49	29th September 2004	November 2009	April 2015	
Java SE 6	50	11th December 2006	April 2013	December 2026 for Oracle[9] December 2026 for Azul[12]	
Java SE 7	51	28th July 2011	September 2022 for OpenJDK Maintained by Oracle until May 2015[16], Red Hat until August 2020[17] and Azul until September 2022[18]	July 2022 for Oracle[9] June 2020 for Red Hat[13] December 2027 for Azul[12]	
Java SE 8 (LTS)	52	18th March 2014	(OpenJDK currently maintained by Red Hat)[19] March 2022 for Oracle (commercial) December 2030 for Oracle (non-commercial) December 2030 for Azul[12] May 2026 for IBM Semeru[14] At least May 2026 for Eclipse Adoptium[10] At least May 2026 for Amazon Corretto[11]	December 2030 for Oracle[9] November 2026 for Red Hat[13]	
Java SE 9	53	21st September 2017	March 2018 for OpenJDK	—	
Java SE 10	54	20th March 2018	September 2018 for OpenJDK	—	
Java SE 11 (LTS)	55	25th September 2018	(OpenJDK currently maintained by Red Hat)[20] September 2026 for Azul[12] October 2024 for IBM Semeru[14] At least October 2024 for Eclipse Adoptium[10] At least September 2027 for Amazon Corretto[11] At least October 2024 for Microsoft[21][15]	September 2026 for Oracle[9] September 2026 for Azul[12] October 2024 for Red Hat[13]	
Java SE 12	56	19th March 2019	September 2019 for OpenJDK	—	
Java SE 13	57	17th September 2019	(OpenJDK currently maintained by Azul)[22] March 2023 for Azul[12]	—	
Java SE 14	58	17th March 2020	September 2020 for OpenJDK	—	
Java SE 15	59	16th September 2020	(OpenJDK currently maintained by Azul)[23] March 2023 for Azul[12]	—	
Java SE 16	60	16th March 2021	September 2021 for OpenJDK	—	
Java SE 17 (LTS)	61	14th September 2021	(OpenJDK currently maintained by SAP)[24] September 2029 for Azul[12] October 2027 for IBM Semeru[14] At least September 2027 for Microsoft[15] At least September 2027 for Eclipse Adoptium[16]	September 2029 or later for Oracle[9] September 2029 for Azul[12] October 2027 for Red Hat[13]	
Java SE 18	62	22nd March 2022	September 2022 for OpenJDK and Adoptium	—	
Java SE 19	63	20th September 2022	March 2023 for OpenJDK	—	
Java SE 20	64	21st March 2023	September 2023 for OpenJDK	—	
Java SE 21 (LTS)	65	September 2023	September 2028	September 2031 for Oracle[9]	

Legend:  Old version  Older version, still maintained  Latest version  Future release

Java Code Names					
<ul style="list-style-type: none"> <li>Code names are used by developers and companies to refer to different versions of Java during the development process.</li> </ul>					
<ol style="list-style-type: none"> <li>1. JDK 1.1: Sparkler</li> <li>2. J2SE 1.2: Playground</li> <li>3. J2SE 1.3: Kestrel</li> <li>4. J2SE 1.4: Merlin</li> <li>5. J2SE 5.0: Tiger</li> <li>6. Java SE 6: Mustang</li> <li>7. Java SE 7: Dolphin</li> <li>8. Java SE 8: Spider</li> <li>9. Java SE 9: Project Jigsaw</li> <li>10. Java SE 10: Project Valhalla</li> <li>11. Java SE 11: Project Amber</li> <li>12. Java SE 12: Project Skara</li> <li>13. Java SE 13: Project ZGC</li> <li>14. Java SE 14: Project Panama</li> <li>15. Java SE 15: Project Loom</li> <li>16. Java SE 16: Project Metropolis</li> <li>17. Java SE 17: Project Panama 2.0</li> </ol>					

## JLS and JSR

---

- JLS stands for Java Language Specification.
- It is a document that defines the syntax, semantics, and behavior of the Java programming language.
- Reference: <https://docs.oracle.com/javase/specs/>
- JSR stands for Java Specification Request.
- It is a formal proposal for adding new features or making changes to the Java platform.
- JSRs are submitted by individuals, organizations, or companies to the Java Community Process (JCP) for review and approval.
- Reference: <https://jcp.org/en/jsr/detail?id=336>

## Editions of Java Platform

---

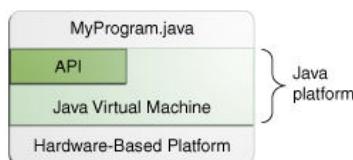
- 1. Java SE**
  - Java Standard Edition( also called as Core Java)
  - It is designed for developing and running desktop, server, and standalone applications.
  - It includes the core Java API and provides a foundation for building Java applications.
- 2. Java EE**
  - Java Enterprise Edition( also called as Web / Advanced / Enterprise Java )
  - It is designed for building web applications and web services.
  - It includes the Java Servlet API, Java Server Pages (JSP), and Enterprise JavaBeans (EJB).
- 3. Java ME**
  - Java Micro Edition.
  - It is designed for developing applications for resource-constrained devices such as mobile phones, PDAs, and other embedded systems.
  - It includes a subset of the Java SE APIs and provides a smaller footprint than Java SE.
- 4. Java Card**
  - It is designed for building smart card applications.
  - It includes a subset of the Java ME APIs and provides security and cryptographic features for smart card applications.

## Software Development Kit

- **SDK = Development Tools + Documentation + Supporting Libraries + Execution Environment.**
- **JDK = Java Development Tools + Java Docs + rt.jar + Java Virtual Machine.**
  - Java Development Tools: javac, java, javap, javadoc, etc.
  - Java Docs: HTML pages, which contains help of Core Java API
  - rt.jar: It contains core Java API
  - Java Virtual Machine: It is abstract computer which manages execution of bytecode.
- JDK is a platform dependent software that we can download from oracle site.
- Developer must install JDK to develop Java application.
- **JRE = rt.jar + Java Virtual Machine.**
- JRE is platform dependent software that we can download from oracle site.
- To run Java application on Client's machine we must install/deploy JRE.
- **JVM = Java Virtual Machine**
- The JVM is responsible for interpreting the bytecode and translating it into machine code that can be executed by the underlying operating system. It also provides memory management, security, and other runtime services necessary for executing Java applications.
- Reference: <https://www.oracle.com/in/java/technologies/downloads/>

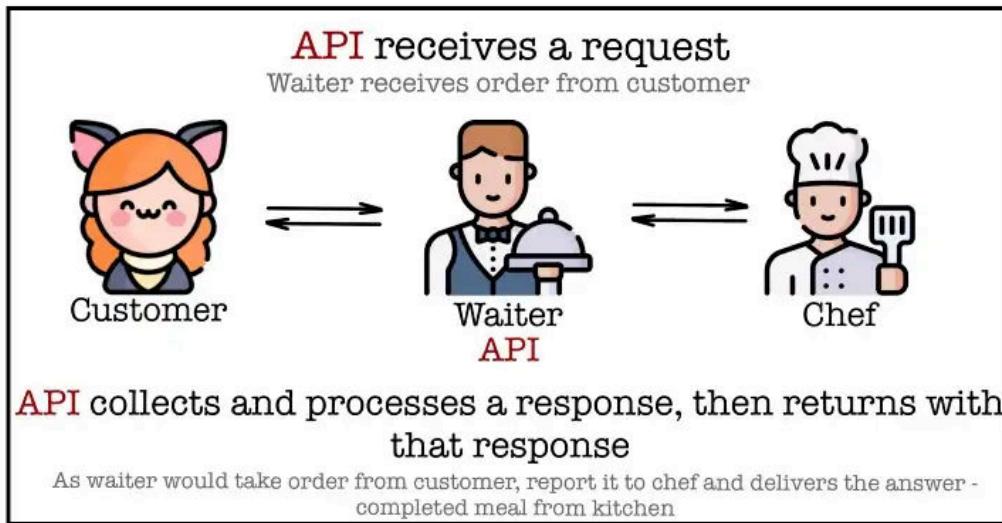
## Components of Java Platform

- The Java platform has two components:
  1. The *Java Virtual Machine*
  2. The *Java Application Programming Interface (API)*



- Reference: <https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>

## **Application Programming Interface**



# Day 2

## Java Documentation

The screenshot shows the Java API documentation for the `Boolean` class. The URL is `docs.oracle.com/javase/8/docs/api/`. The page title is "java.lang.Boolean". The top navigation bar includes links for Gmail, YouTube, Translate, Sign in with App P..., Jonas' Resources..., GitHub - srjainapu..., How To Implement..., Build Responsive..., New Tab, Google Meet, and Overview (Java Pl...). The top right corner shows "Java™ Platform Standard Ed. 8". The left sidebar lists packages under "List of Packages": java.awt.event, java.awt.font, java.awt.geom, java.awt.im, java.awt.im.spi, java.awt.image, java.awt.image.renderable, java.awt.print, java.beans, java.beans.beancontext, java.io, java.lang, java.lang.annotation, java.lang.instrument, java.lang.invoke, java.lang.management, java.lang.ref, java.lang.reflect, java.math, and java.net. A specific section for "java.lang" is highlighted. Under "Interfaces", there is a "List Of Types" section with items: Appendable, AutoCloseable, CharSequence, Comparable, Iterable, Readable, Runnable, and Thread.UncaughtExceptionHandler. Under "Classes", there is a list including Boolean, Byte, Character, Character.Subset, Character.UnicodeBlock, Class, ClassLoader, ClassValue, Compiler, and Double. The main content area shows the `Boolean` class definition, its inheritance from `Object`, and its implementation of `Serializable` and `Comparable<Boolean>`. It describes the class as wrapping a primitive boolean value. The "Field Summary" section is visible at the bottom.

## Contents of package

- Sub package
- Interface
  - Nested Type
  - Constant Fields
  - Abstract Methods
  - Default Methods
  - Static Interface Methods
- Class
  - Nested Types( Interface / Class / Enum )
  - Fields
  - Constructor
  - Method
- Enum
- Exception

- Error
- Annotation Type

## Modifiers in Java

- PUBLIC
- PRIVATE
- PROTECTED
- STATIC
- FINAL
- SYNCHRONIZED
- VOLATILE
- TRANSIENT
- NATIVE
- INTERFACE
- ABSTRACT
- STRICT
- Reference: <https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Modifier.html>

## Access Modifiers

- Modifiers which are used to control visibility of members of the class is called access modifier.
- Access modifiers
  - private
  - Package level private( also called as default )
  - protected
  - public

## More about main method

- In java, main method is considered as entry point method.
- With the help of main thread, JVM invoke main method.
- Syntax:
  - public static void main(String args[] )
  - public static void main(String[] args )
  - public static void main(String... args )
- We can define main method inside class as well as interface.

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

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

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

- Reference: <https://docs.oracle.com/javase/tutorial/getStarted/application/index.html>

## Java Comments

- To maintain documentation of source code we should use comments:
- Types of comments:
  - Implementation Comments
    - Block comments / multiline comments

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

- Single line comments

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

- Documentation comment / doc comments

```
/**  
 * Author is Sandeep Kulange  
 */  
class Program{  
    /**  
     * Entry point method of class Program  
     */  
    public static void main(String[] args ) {  
        System.out.println("Hello World!!!");  
    }  
}
```

- Reference: <https://www.oracle.com/java/technologies/javase/codeconventions-comments.html>

## Java OOPS concepts

- Consider following code:

```
class Test{  
    //Fields  
    private int num1;          //Instance variable  
    private static int num2;   //Class level variable  
    //Methods  
    public void setNum1( int num1 ){ //Instance method  
        this.num1 = num1;  
    }  
    public static void setNum2( int num2 ){ //Class level method  
        this.num2 = num2;  
    }  
}
```

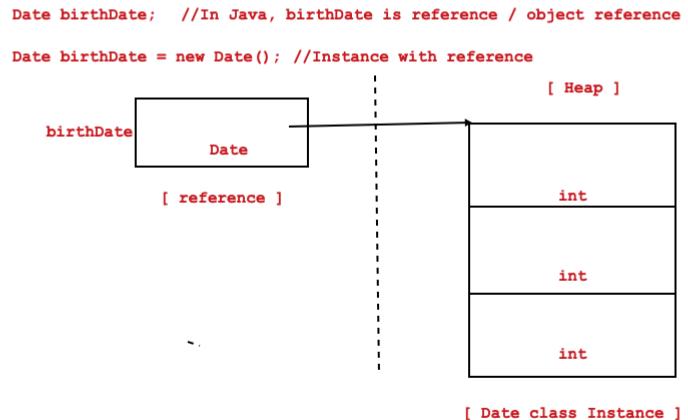
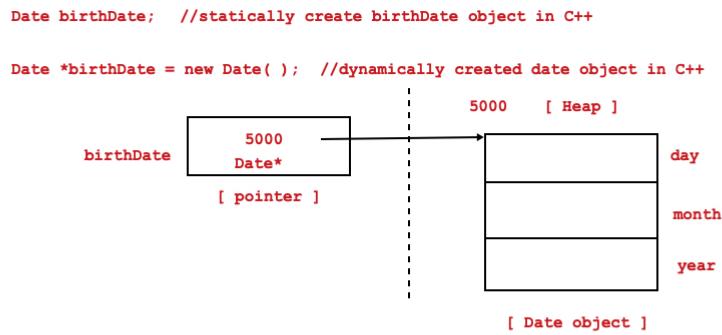
- In Java, to access static members ( class level members ) of the class, we must use class name and dot operator.

```
public static void main(String[] args) {  
    Test.setNum2( 123 );  
}
```

- To access non static members( instance members ) of the class, we must use instance of the class.

```
Test t; //Object reference / reference  
new Test; //Not OK  
new Test(); //OK: Instance  
Test t = new Test( ); //Instantiation
```

```
public static void main(String[] args) {  
    //Test.setNum1( 100 ); //Not OK  
    Test t = new Test( );  
    t.setNum1( 100 ); //OK  
}
```



- A class from which we can not create instance( In C/C++ it is called as object ) is called abstract class.
- In simple words, we can not instantiate abstract class.
- Example:
  - `java.lang.Number`
  - `java.lang.Enum`
- A class from which we can create instance is called concrete class.
- In simple words, we can instantiate concrete class.
- Example:
  - `java.lang.Runtime`
  - `java.lang.Thread`
- A method of class which is having body is called as concrete method.
- main method is concrete method.
- Example:

```

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

```

- A method of class which do not have body is called as abstract method.

- Example:

```
abstract class Shape{
    public abstract void calculateArea( );
}
```

- A class from which we can not create child / sub class is called as final class.
- We can instantiate final class.
- Example:
  - java.lang.System
  - java.lang.String
- If method in parent /super class is final then we can not redefine / override that method in child / sub class.

What is System.out.println?

```
package java.lang;
public final class System {
    //Fields
    public final static InputStream in = null;

    public final static PrintStream out = null;

    public final static PrintStream err = null;
```

- System is a final class declared in java.lang package.
- out is a reference of java.io.PrintStream class. It is declared as public static final field inside System class.

```
//How will you access in
System.in //It represents keyboard
//How will you access out
System.out //It represents Monitor
//How will you access err
System.err //It represents Monitor for errors
```

- PrintStream is a class declared in java.io package.

```
package java.io;
public class PrintStream extends FilterOutputStream implements Appendable, Closeable{

    public void print(boolean);
    public void print(char);
```

```

public void print(int);
public void print(long);
public void print(float);
public void print(double);
public void print(char[]);
public void print(String);
public void print(Object);

public void println();
public void println(boolean);
public void println(char);
public void println(int);
public void println(long);
public void println(float);
public void println(double);
public void println(char[]);
public void println(String);
public void println(Object);

public PrintStream printf(String, Object...);
public PrintStream printf(Locale, String, Object...);
}

```

- `println` is a non static method of `java.io.PrintStream` class.

### `print`, `println` and `printf`

- `System.out.print` method print output on console but it keeps cursor on same line.

```

class Program{
    public static void main1(String[] args ) {
        System.out.print("Hello ");
        System.out.print(" World");
        System.out.print("!!!");
    }
}

```

- `System.out.println` method print output on console but it moves cursor to the next line.

```

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

```

- To print formatted output on console / terminal, we should use `printf` method.

```

public static void main(String[] args) {
    String name1 = "Sandeep Kulange";
    int empid1 = 10003778;
    float salary1 = 45000.50f;

    String name2 = "Mahesh Koli";
    int empid2 = 3779;
    float salary2 = 125000.50f;

    System.out.printf("%-20s%-10d%-10.2f\n", name1, empid1,
salary1);
    System.out.printf("%-20s%-10d%-10.2f\n", name2, empid2,
salary2);
}

```

## Data Types

- Data type of any variable describe following things:
  - ( Memory ) How much memory is required to store the data.
  - ( Nature ) Which type of data is allowed to store inside allocated memory.
  - ( Range ) How much data is allowed to store inside allocated memory.
  - ( Operation ) Which operations are allowed to perform / execute on the data stored inside memory.
- Types of data type
  - Primitive data type
    - Variable of primitive data type contains value. Hence primitive type is also called as value type.
    - There are 8 primitive / value types in Java:
      - boolean
      - byte
      - char
      - short
      - int
      - float
      - double
      - long
  - Non Primitive data type
    - Variable of non primitive data type contains reference of the instance. Hence non primitive type is also called as reference type.
    - There are 4 non primitive / reference types in Java
      - Interface
      - Class
      - Enum
      - Array

- If we declare variable inside method then it is called as method local variable.

- In Java, we can not declare method local variable static.

```
class Program {  
    public static void main(String[] args) {  
        int number; //Non Static Method Local Variable //OK  
    }  
}
```

```
class Program {  
    public static void main(String[] args) {  
        int number; //Non Static Method Local Variable  
        System.out.println( number ); //error: variable number might not  
        have been initialized  
    }  
}
```

- If we want to use any method local variable( primitive / non primitive ) then we must store some value inside it.

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

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

Sr.No.	Primitive Type	Size	Default for field	Wrapper Class
1	<code>boolean</code>	Not Specified	<code>FALSE</code>	<code>java.lang.Boolean</code>
2	<code>byte</code>	1 byte	0	<code>java.lang.Byte</code>
3	<code>char</code>	2 Bytes	\u0000	<code>java.lang.Character</code>
4	<code>short</code>	2 Bytes	0	<code>java.lang.Short</code>
5	<code>int</code>	4 Bytes	0	<code>java.lang.Integer</code>
6	<code>float</code>	4 Bytes	0.0f	<code>java.lang.Float</code>
7	<code>double</code>	8 Bytes	0.0d	<code>java.lang.Double</code>
8	<code>long</code>	8 Bytes	0L	<code>java.lang.Long</code>

- Reference: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

## Wrapper classes

- In Java, primitive types are not classes. But Sun/Oracle developers has written classes for it. Such classes are called as Wrapper classes.
- All the Wrapper classes are final and declared in `java.lang` package.
- Class Hierarchy
  - `java.lang.Object`
    - `java.lang.Boolean`
    - `java.lang.Character`
    - `java.lang.Number`
      - `java.lang.Byte`
      - `java.lang.Short`
      - `java.lang.Integer`
      - `java.lang.Float`
      - `java.lang.Double`
      - `java.lang.Long`

## Initialization

- Process of storing value inside variable during its declaration is called initialization.

```
int number = 10; //Initialization
```

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

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

```
//int num2 = 20; //error: variable num2 is already defined in method
main(String[])
```

- We can initialize variable only once.
- Process of storing value inside variable after its declaration is called as assignment.

```
public static void main(String[] args) {
    int num1 = 10;      //Initialization
    int num2 = num1;   //Initialization
    num2 = 20;        //Assignment : OK
    num2 = 30;        //Assignment : OK
}
```

- We can do assignment multiple times.

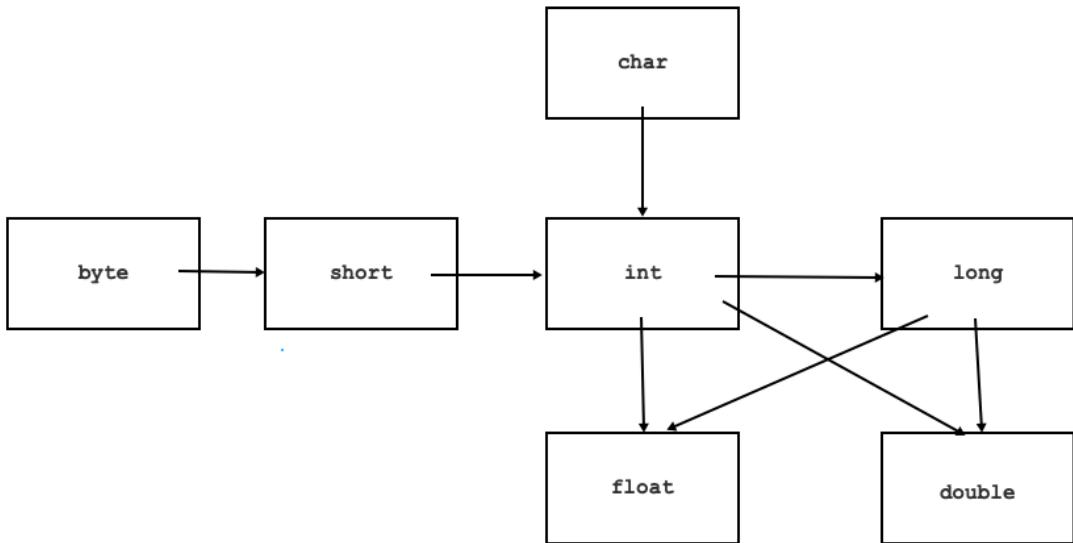
## Narrowing & Widening

- Process of converting, value of variable of narrower type into wider type is called as widening.

```
public static void main(String[] args) {
    int num1 = 10;
    double num2 = ( double )num1; //Widening
    System.out.println("Num2 : "+num2); //OK: Num2 : 10.0
}
```

- In case of widening conversion, explicit typecasting is optional.

```
public static void main(String[] args) {
    int num1 = 10;
    //double num2 = ( double )num1; //Widening
    double num2 = num1; //Widening
    System.out.println("Num2 : "+num2); //OK: Num2 : 10.0
}
```



- Process of converting value of variable of wider type into narrower type is called as narrowing.

```

public static void main(String[] args) {
    double num1 = 10.5d;
    int num2 = (int)num1; //Narrowing
    System.out.println("Num2 : "+num2);
}

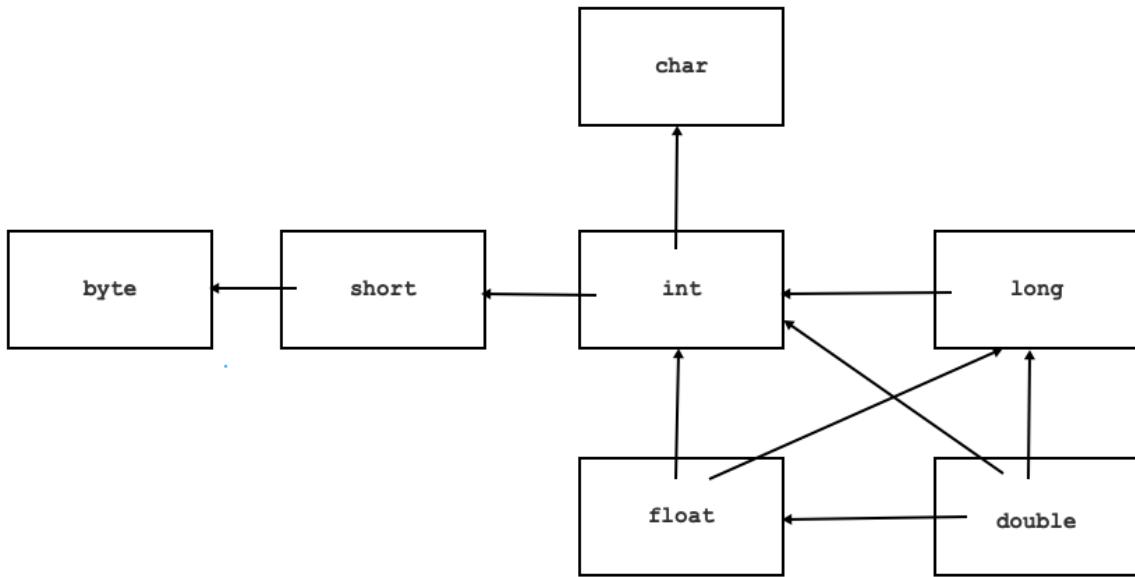
```

- In case of narrowing, explicit type casting is mandatory.

```

public static void main(String[] args) {
    double num1 = 10.5d;
    int num2 = (int)num1; //Narrowing
    //int num2 = num1; //error: incompatible types: possible lossy
    conversion from double to int
    System.out.println("Num2 : "+num2);
}

```



## Boxing & Unboxing

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

```

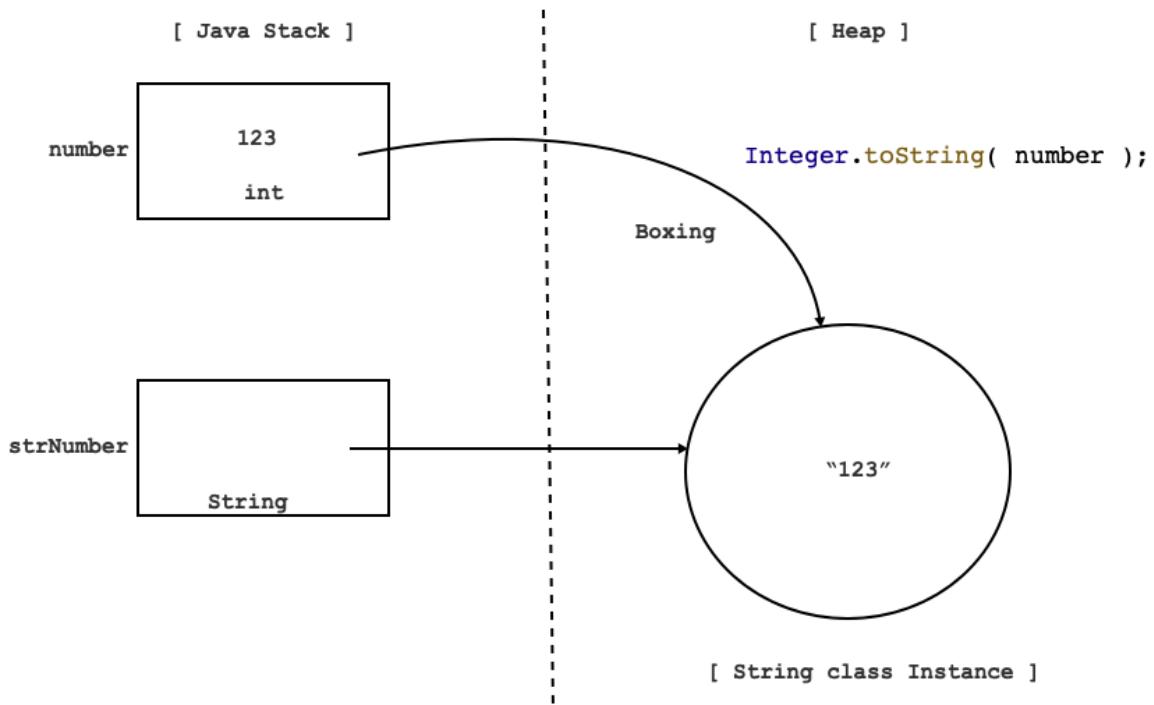
public static void main(String[] args) {
    int number = 123;
    //String str = ( String)number; //error: incompatible types: int
    //cannot be converted to String
    String strNumber = Integer.toString( number ); //Boxing
    System.out.println("Number : "+strNumber);
}

```

```

public static void main(String[] args) {
    int number = 123;
    String strNumber = String.valueOf( number ); //Boxing
    System.out.println("Number : "+strNumber);
}

```

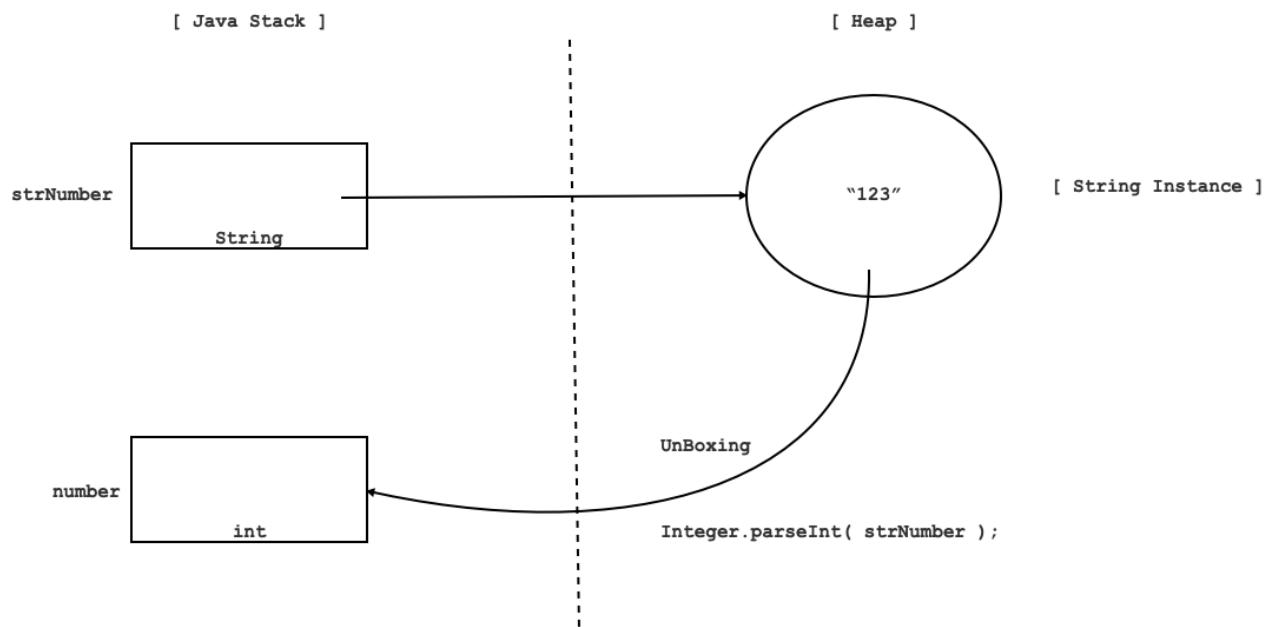


- Process of converting value of variable of non primitive type into primitive type is called as unboxing.

```

public static void main(String[] args) {
    String strNumber = new String( "123" );
    //int number = ( int ) strNumber; //error: incompatible types:
    String cannot be converted to int
    int number = Integer.parseInt(strNumber); //UnBoxing
    System.out.println("Number : "+number);
}

```



## NumberFormatException

```
public static void main(String[] args) {
    String strNumber = new String( "1A2B3C" );
    int number = Integer.parseInt(strNumber);
//java.lang.NumberFormatException
    System.out.println("Number : "+number);
}
```

- If the string does not contain a parsable numeric value then parseXXX() method throws NumberFormatException. Reference:  
<https://docs.oracle.com/javase/8/docs/api/java/lang/NumberFormatException.html>

## Commandline arguments

- Consider code in C language

```
int sum( int num1, int num2 ){ //num1, num2 <= Function parameters
    int result = num1 + num2;
    return result;
}
int main( void ){
    int result = sum( 10, 20 ); //10, 20 <= Function arguments
    return 0;
}
```

- How to pass argument from command line?

```
sandeep@sandeeps-MacBook-Air Day_2.6 % java Program Sandeep
```

```
class Program {
    //args is a method parameter
    public static void main(String[] args) {
        System.out.println("Hello,"+args[ 0 ]);
    }
}
```

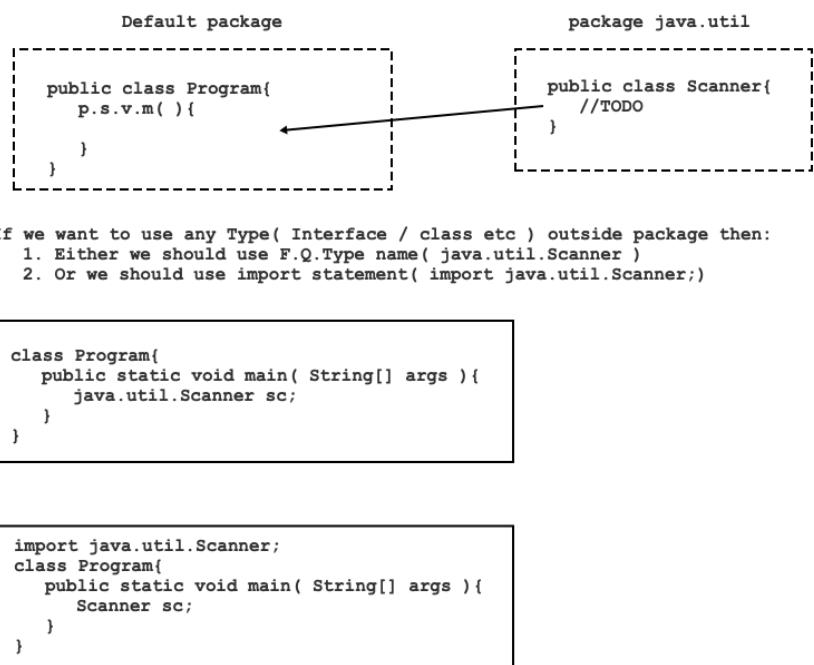
```
sandeep@sandeeps-MacBook-Air Day_2.6 % java Program "Sandeep Kulange"
```

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

```
int num1 = Integer.parseInt(args[ 0 ]);  
float num2 = Float.parseFloat(args[ 1 ]) ;  
double num3 = Double.parseDouble(args[ 2 ]) ;  
double result = num1 + num2 + num3;  
System.out.println("Result : "+result);  
}  
public static void main1(String[] args) {  
    int num1 = Integer.parseInt(args[ 0 ]);  
    int num2 = Integer.parseInt(args[ 1 ]) ;  
    int result = num1 + num2;  
    System.out.println("Result : "+result);  
}  
}
```

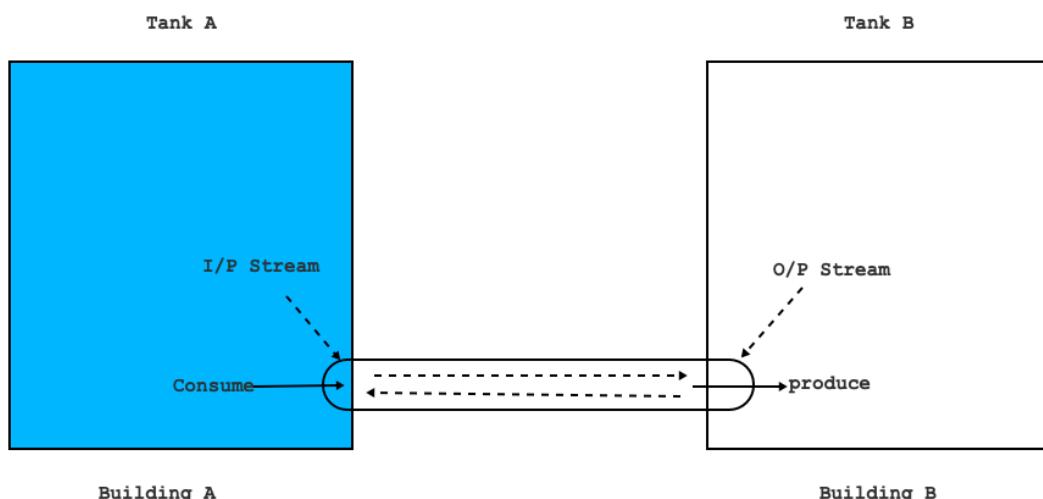
# Day 3

```
Core java Library : rt.jar
- Manifest File( metadata file )
  - Information about author
  - Information about JVM
  - Classpath information
- Resources
  - property files( Log4j2.properties )
  - Font files
  - Images( .gif /.jpg )
- Packages
  - Sub Package
  - Interface
  - Class
  - Enum
  - Exception
  - Error
- Annotation Types
```



## Stream

- It is an abstraction( object /instance ) which is used to produce( write ) or consume( read ) information from source to destination.
- Console = Keyboard + Monitor
- Keyboard => Console Input
- Monitor => Console Output



## Standard stream objects of Java associated with Console

- System.in( java.io.InputStream )
  - It represents keyboard

- System.out( java.io.PrintStream )
  - It represents monitor
  - System.out.println("Good Morning");
- System.err( java.io.PrintStream )
  - It represents monitor but generally designed to display errors on screen
  - System.err.println("Divide by zero exception");

## User input using Scanner class

- We can access user input from terminal using:
  - Console class
  - BufferedReader
  - Scanner class
  - JOptionPane class
- Scanner is a final class which is declared in java.util package.
- We can use it to read data from console as well as file.
- Instantiation:

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

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

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

- Methods of java.util.Scanner class
  - public boolean nextBoolean()

- public byte nextByte()
- public short nextShort()
- public int nextInt()
- public long nextLong()
- public float nextFloat()
- public double nextDouble()
- public String nextLine()

```

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

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

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

```

- Reference: <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>

## Java language Features / buzzwords / ( Sun Microsystems Marketing Keywords )

- Simple
- Object oriented
- Architecture neutral
- Portable
- Robust
- Multithreaded
- Dynamic
- Secure
- High Performance
- Distributed

## What is the meaning of Deprecated?

- In Java, the term "Deprecated" is used to indicate that a particular method, class, or interface should no longer be used because it is either obsolete, flawed, or has been replaced by a better alternative.

## Instance creation in Java

- In Java, we can create instance using 3 ways:
  - Using new operator

- Using factory method
- Using reflection

Manipulating system date and time using Date, Calendar, LocalDate, LocalTime, LocalDateTime

- System Date using java.util.Date class

```
import java.util.Date;
class Program {
    public static void main(String[] args) {
        Date date = new Date();
        int day = date.getDate();
        int month = date.getMonth() + 1;
        int year = date.getYear() + 1900;
        System.out.println(day+" / "+month+" / "+year);
    }
}
```

- The Calendar class should be used to convert between dates and time fields and the DateFormat class should be used to format and parse date strings. The corresponding methods in Date are deprecated.
- Calendar is abstract class declared in java.util package.

```
Calendar c = new Calendar(); //Not OK; Because Calendar is abstract class
```

- To create instance of Calendar class we should use factory method( getInstance()).

```
Calendar c = Calendar.getInstance(); //OK
```

- Fields of Calendar class

- public static final int DAY\_OF\_MONTH
- public static final int DATE
- public static final int MONTH
- public static final int YEAR

- "public int get(int field)" is non static / instance method of Calendar class.

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

```

import java.util.Calendar;
class Program {
    public static void main(String[] args) {
        Calendar c = Calendar.getInstance();
        //int day = c.get( Calendar.DAY_OF_MONTH );
        //or
        int day = c.get( Calendar.DATE );
        int month = c.get( Calendar.MONTH ) + 1;
        int year = c.get( Calendar.YEAR );

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

```

- System Date using `java.time.LocalDate` class.
- "`public static LocalDate now()`" is a factory method of `LocalDate` class

```
LocalDate ld = LocalDate.now();
```

- Methods of `LocalDate` class:
  - `public int getDayOfMonth()`
  - `public int getMonthValue()`
  - `public int getYear()`

```

import java.time.LocalDate;
class Program {
    public static void main(String[] args) {
        LocalDate ld = LocalDate.now();
        int day = ld.getDayOfMonth();
        int month = ld.getMonthValue();
        int year = ld.getYear();
        System.out.println(day+ " / "+month+ " / "+year);
    }
}

```

- System Time using `java.time.LocalTime` class.
- "`public static LocalTime now()`" is a factory method of `LocalTime` class.

```
LocalTime lt = LocalTime.now();
```

- Methods of `LocalTime` class:

- public int getHour()
- public int getMinute()
- public int getSecond()
- public int getNano()

## Formatting date and time using SimpleDateFormat

- If we want to parse or format date the we should use `java.text.SimpleDateFormat` class.

```
import java.util.Date;
import java.text.SimpleDateFormat;

class Program {
    public static void main(String[] args) throws Exception {
        String pattern = "dd/MM/yyyy";
        SimpleDateFormat sdf = new SimpleDateFormat( pattern );
        String source = "23/07/1983";
        Date date = sdf.parse( source );

        int day = date.getDate();
        int month = date.getMonth() + 1;
        int year = date.getYear() + 1900;
        System.out.println(day+ " / "+month+ " / "+year);
    }
}
```

```
import java.util.Date;
import java.text.SimpleDateFormat;

class Program {
    public static void main(String[] args) {
        Date date = new Date( );
        //SimpleDateFormat sdf = new SimpleDateFormat( "dd-MM-yyyy" );
        //SimpleDateFormat sdf = new SimpleDateFormat( "dd/MMM/yyyy" );
        SimpleDateFormat sdf = new SimpleDateFormat( "dd/MMMM/yyyy" );
        String strDate = sdf.format(date);
        System.out.println( strDate );
    }
}
```

- Reference: <https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html>

## Phases of software development life cycle

- Requirement
- Analysis
- Design
- Coding / implementation
- Testing
- Deployment / installation
- maintenance

## Procedure oriented programming language

- Write a program to accept, store and print information of employeee
  - name : char[ 30 ]
  - empid : int
  - salary : float

```
//File Name: Main.c
#include<stdio.h>
struct Employee{
    char name[ 30 ];
    int empid;
    float salary;
};
void accept_record( struct Employee *ptr ){
    printf("Name : ");
    scanf("%s", ptr->name);
    printf("Empid : ");
    scanf("%d", &ptr->empid);
    printf("Salary : ");
    scanf("%f", &ptr->salary);
}
void print_record( struct Employee *ptr ){
    printf("%-30s%-5d%-10.2f\n", ptr->name, ptr->empid, ptr->salary);
}
int main( void ){
    struct Employee emp;
    accept_record( &emp );
    print_record( &emp );
}
```

- A programming methodology in which we can solve real world problems using structure and global function.

## Object oriented programming in C++

```
//File Name: Main.cpp
#include<stdio.h>
class Employee{
private:
    char name[ 30 ];
```

```

int empid;
float salary;
public:
    void accept_record( /* struct Employee *this */ ){
        printf("Name : ");
        scanf("%s", this->name);
        printf("Empid : ");
        scanf("%d", &this->empid);
        printf("Salary : ");
        scanf("%f", &this->salary);
    }
    void print_record( /* struct Employee *this */ ){
        printf("%-30s%-5d%-10.2f\n", this->name, this->empid, this->salary);
    }
};

int main( void ){
    Employee emp;
    emp.accept_record( ); //emp.accept_record( &emp );
    emp.print_record( ); //emp.print_record( &emp );
    return 0;
}

```

- A programming methodology, in which we can solve real world problem using class and object is called object oriented programming.

## Object oriented programming in Java

- Write a program to store and print date information.
  - day : int
  - month : int
  - year : int
- Analyze problem statement and define the class.

```

class Date{
    //TODO
}

```

- Analyze the problem statement, decide and declare field inside class.
  - Variable declare inside class / class scope is called as field.
  - Variable declare inside method / method scope is called as method local variable.

```

class Date{
    //Fields
    int day;
    int month;
    int year;
}

```

- To store values, create instance of the class.
  - Process of creating instance of a class is called as instantiation.
  - A class from which we can create instance is called as concrete class. In other words, we can instantiate concrete class.
  - In Java, to create instance, we should use new operator.
  - If we use new operator to allocate memory then it gets space of heap memory area.

```
new Date(); //Anonymous instance of Date class
```

- If we create instance without reference then it is called as anonymous instance.
- Following entities do not get space inside instance:
  - Method parameter
  - Method local variable
  - Method
  - Nested Type / class
  - Static field
- Only non static field / instance variable get space inside instance.
- Non static field get space once per instance according to their order of declaration inside class.

```
new Date();
new Date();
new Date();
```

- After instantiation, if we want to perform operations on instance then we require object reference / reference of the class.

```
Date birthDate; //reference / object reference
Date joinDate; //reference / object reference
Date leaveDate; //reference / object reference
```

```
Date dt; //reference
dt = new Date(); //instance with reference
```

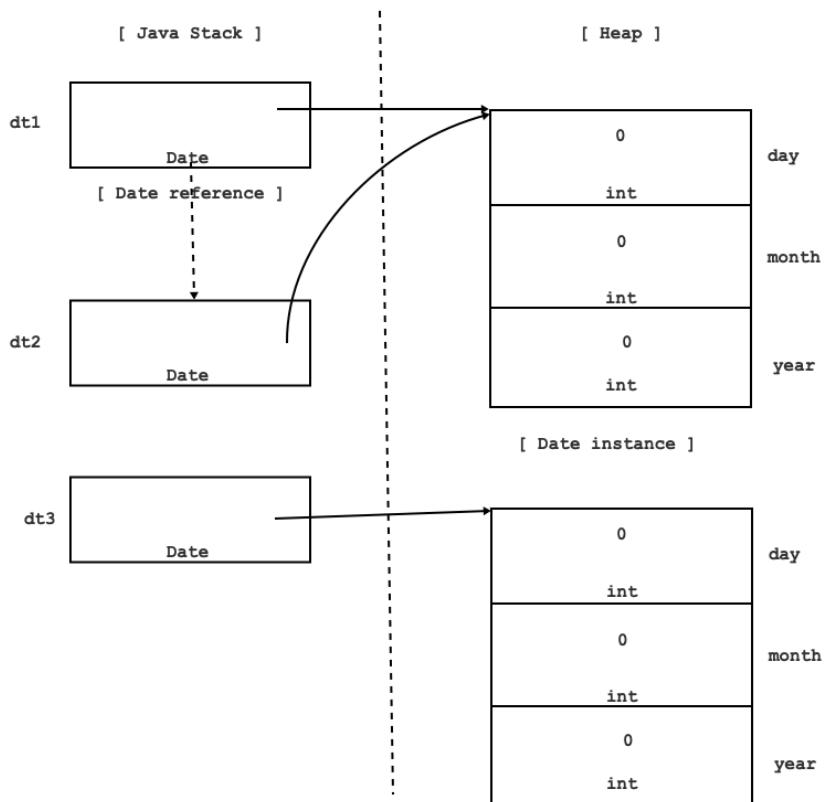
```
Date dt = new Date(); //instance with reference
```

- Test your understanding

```

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

```



- After instantiation, if we want to process( accept/print/get/set) state of the instance then we should call method on instance.

```

Date dt = new Date();
dt.acceptRecord(); //Message Passing

```

- Here, acceptRecord() method is called on dt instance.
  - function defined inside class / class scope is called as method.
  - method is also called as operation/behavior/message
- Process of calling method on instance is called as message passing.

# Day 4

How to find size of instance in Java?

- Reference: <https://www.baeldung.com/java-size-of-object>

Object oriented programming in Java

- Write a program to store and print date information.
  - day : int
  - month : int
  - year : int
- Analyze problem statement and define the class.

```
class Date{
    //TODO
}
```

- Analyze the problem statement, decide and declare field inside class.
  - Variable declare inside class / class scope is called as field.
  - Variable declare inside method / method scope is called as method local variable.

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

- To store values, create instance of the class.
  - Process of creating instance of a class is called as instantiation.
  - A class from which we can create instance is called as concrete class. In other words, we can instantiate concrete class.
  - In Java, to create instance, we should use new operator.
  - If we use new operator to allocate memory then it gets space of heap memory area.

```
new Date(); //Anonymous instance of Date class
```

- If we create instance without reference then it is called as anonymous instance.
- Following entities do not get space inside instance:
  - Method parameter
  - Method local variable
  - Method

- Nested Type / class
- Static field
- Only non static field / instance variable get space inside instance.
- Non static field get space once per instance according to their order of declaration inside class.

```
new Date();
new Date();
new Date();
```

- After instantiation, if we want to perform operations on instance then we require object reference / reference of the class.

```
Date birthDate; //In C++, It is object
Date birthDate; //In Java, It is reference / object reference
```

```
Date dt; //reference
dt = new Date(); //instance with reference
```

```
Date dt = new Date(); //instance with reference
```

- After instantiation, if we want to process( accept/print/get/set) state of the instance then we should call method on instance.

```
Date dt = new Date();
dt.acceptRecord(); //Message Passing
```

- Here, acceptRecord() method is called on dt instance.
  - function defined inside class / class scope is called as method.
  - method is also called as operation/behavior/message
- Process of calling method on instance is called as message passing.

## Short introduction of String

- Non primitive types / reference types
  - Interface
  - Class
  - Enum
  - Array

- String is a final class declared in java.lang package.

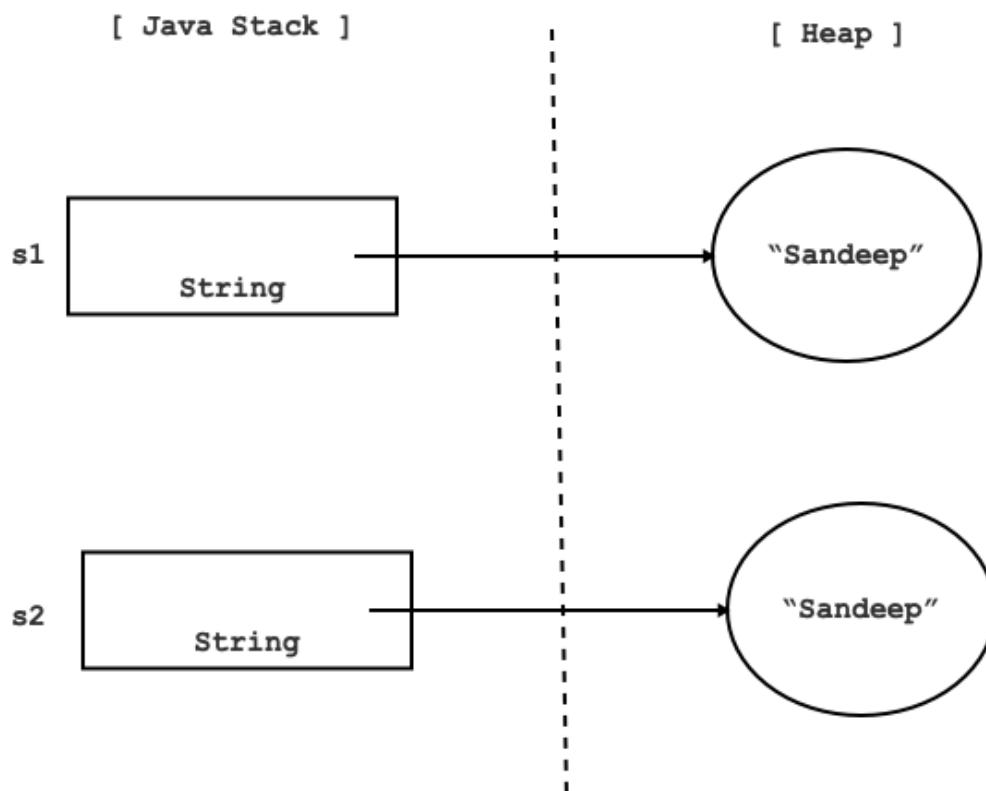
- Instantiation with new operator:

```
String s1 = new String();
String s2 = new String("Sandeep Kulange");
//new String("Sandeep Kulange"); => String instance
```

- String instance get space on heap( Runtime data/memory area of JVM ).

```
String s1 = new String("Sandeep");
```

```
String s2 = new String("Sandeep");
```



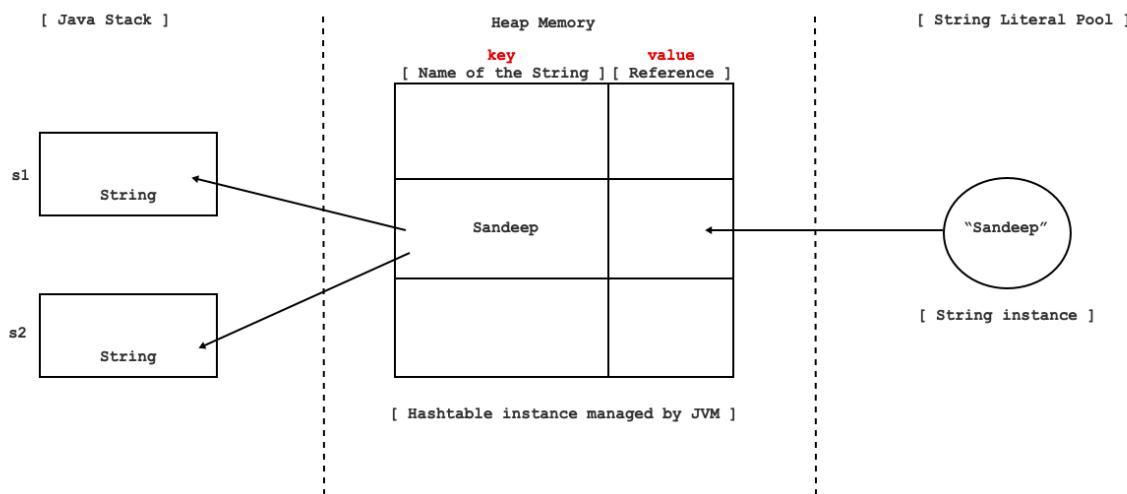
- Instantiation with out new operator:

```
String s1 = "";
String s2 = "Sandeep Kulange";
// "Sandeep Kulange" => String literal / literal

/* char[] data = new char[]{ 'S','a','n','d','e','e','p',' ',
  'K','u','l','a','n','g','e'};
String s1 = new String( data ); */
```

- String literals get space on String literal pool( Shared memory area for String instances on heap ).

```
String s1 = "Sandeep";
String s2 = "Sandeep";
```



What is this reference?

- this is a keyword in Java.
- this is pointer in C++ and reference in Java.
- this reference is considered as implicit parameter to the method.
- Method parameter do not get space inside instance. Since this is a method parameter, it doesn't get space inside instance.
- In Java, method local variable, method parameter and method call get space into Stack frame.
- Stack frame get space into Java Stack and Java stack get space per thread.
- this reference get space once per method call on java stack.
- If we call non static method ( is also called as instance method ) on instance then compiler implicitly pass reference of calling/current instance as argument to the method. To accept/catch reference of instance, compiler implicitly declare one reference variable as a method parameter. Such method parameter is called as this reference.
- Using this reference, non static fields and non static methods can communicate with each other. Hence this reference is considered as a link/connection between them.

```
void acceptRecord( /*Employee this*/ ){
    Scanner sc = new Scanner(System.in);
    System.out.print("Name : ");
    this.name = sc.nextLine();
    System.out.print("Empid : ");
    this.empid = sc.nextInt();
    System.out.print("Salary : ");
    this.salary = sc.nextFloat();
}
```

- To access instance members of the class inside method, use of this keyword is optional.

```

void acceptRecord( /*Employee this*/ ){
    Scanner sc = new Scanner(System.in);
    System.out.print("Name : ");
    name = sc.nextLine(); //OK
    System.out.print("Empid : ");
    empid = sc.nextInt(); //Ok
    System.out.print("Salary : ");
    salary = sc.nextFloat(); //OK
}

```

- Definition:
  - this reference is implicit reference variable which is available in every non static method / instance method of class which is used to store reference of current / calling instance.
- Consider code in C

```

int num1 = 10;
int main( void ){
    int num1 = 20;
    printf("Num1 : %d\n", num1);
    return 0;
}

```

- In C/C++, if name of local & global variable is same then preference is always given to the local variable.
- Consider code in java:

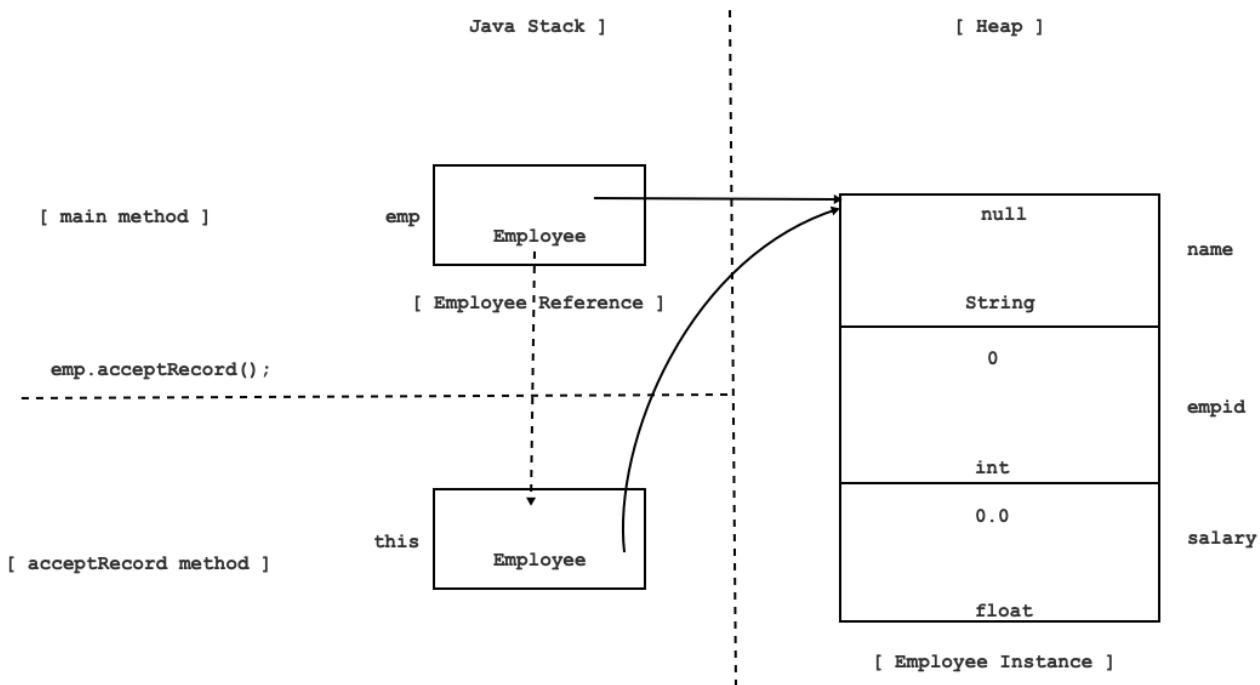
```

class Date{
    int day, month, year;
    void setDay( /* Date this, */int day ){
        //day = day; //Assignment to self
        this.day = day;
    }
}
class Program{
    public static void main(String[] args) {
        Date date = new Date();
        date.setDay( 23 ); //date.setDay( date, 23 );
    }
}

```

- If name of method local variable / method parameter and name of field is same then preference is always given to the local variable. In this case, If we want to give preference to field then we should use this reference before field. For the reference consider above code.

- Static method do not get this reference.



## Coding conventions in java:

- Reference: <https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>
- Pascal case naming / coding convention
  - Consider examples:
    - String
    - StringBuffer
    - NumberFormatException
    - IndexOutOfBoundsException
- In this coding convention, including first word, first character of each word must be in upper case.
- We should use this convention for:
  - For Type names( Interface/Class/Enum etc )
  - For file name
- Camel case naming / coding convention
- Consider examples:
  - main
  - parseInt
  - showInputDialog
  - waitForPendingFinalizer
- In this coding convention, excluding first word, first character of each word must be in upper case.

- We should use this convention for:
  - Field
  - Method
  - Method local variable and method parameter
- Coding convention for package name
- Consider examples:
  - java.lang
  - java.lang.reflect
  - cdac.in
  - org.example.test
- package name should be in lower case.
- Coding convention for enum constant
- Consider example:

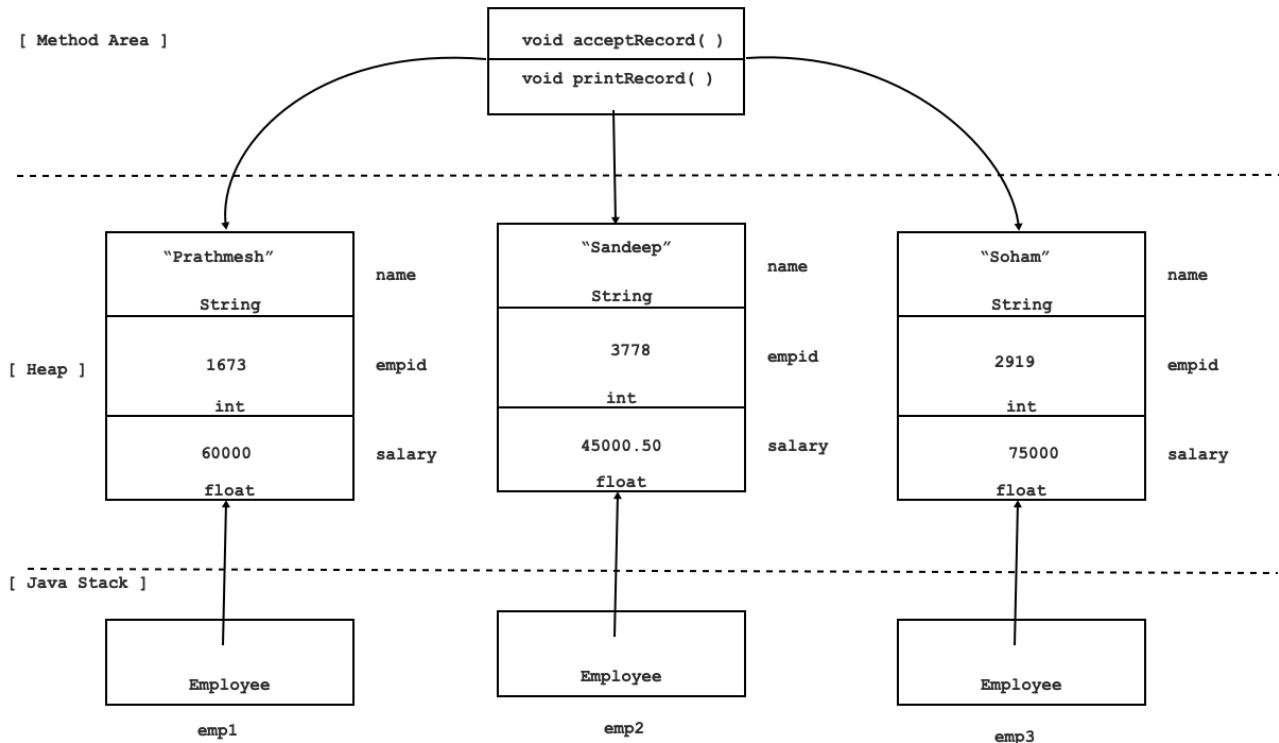
```
enum ShapeType{
    LINE, RECTANGLE, CIRCLE, TRIANGLE
}
```

- Name of enum constants should be in uppercase.
- Naming convention for final variable:
  - public static final int SIZE
  - public static final int BYTES
  - public static final int MIN\_VALUE
  - public static final int MAX\_VALUE
- Name of final field should be in upper case.

## OOPS concepts related to field and methods

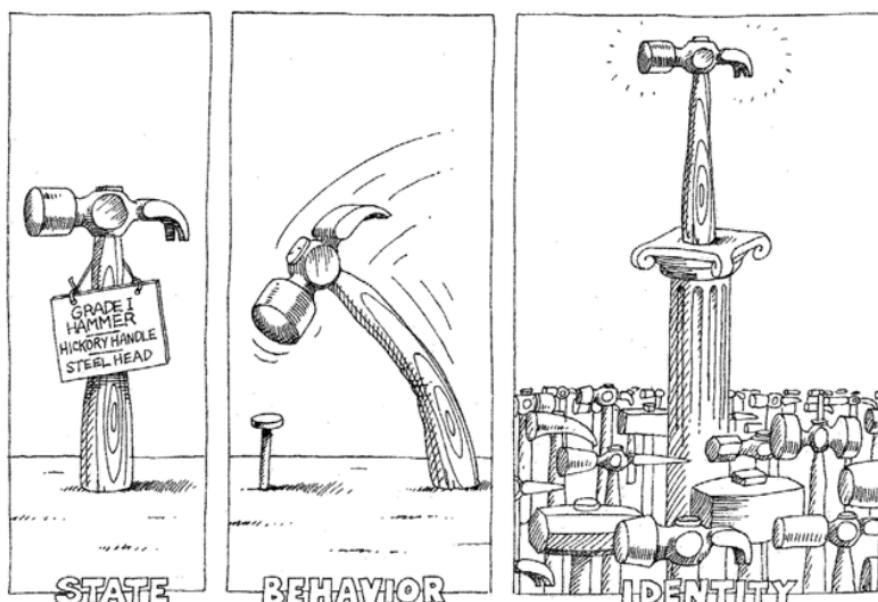
- If we declare fields( specifically instance variable ) inside class then it gets space once per instance according to order of their declaration inside class.
- Methods do not get space inside instance. Rather all the instances of "same class" share single copy of method defined inside class.
- By passing reference of current instance to the method, instances share single copy of the method.
- Structure of instances of same class is always same/common.

- All the instances of same class share single copy of method.



## Characteristics of instance

- State
  - Value stored inside instance is called as state of that instance.
  - Value stored inside field represents state of the instance.
- Behavior
  - Number / set of operations( behaviors/methods) which are allowed to perform/call on instance represents behavior of the instance.
  - Methods defined inside class represents behavior of the instance.
- Identity
  - Value of any field which is used to identify instance uniquely is called its identity.



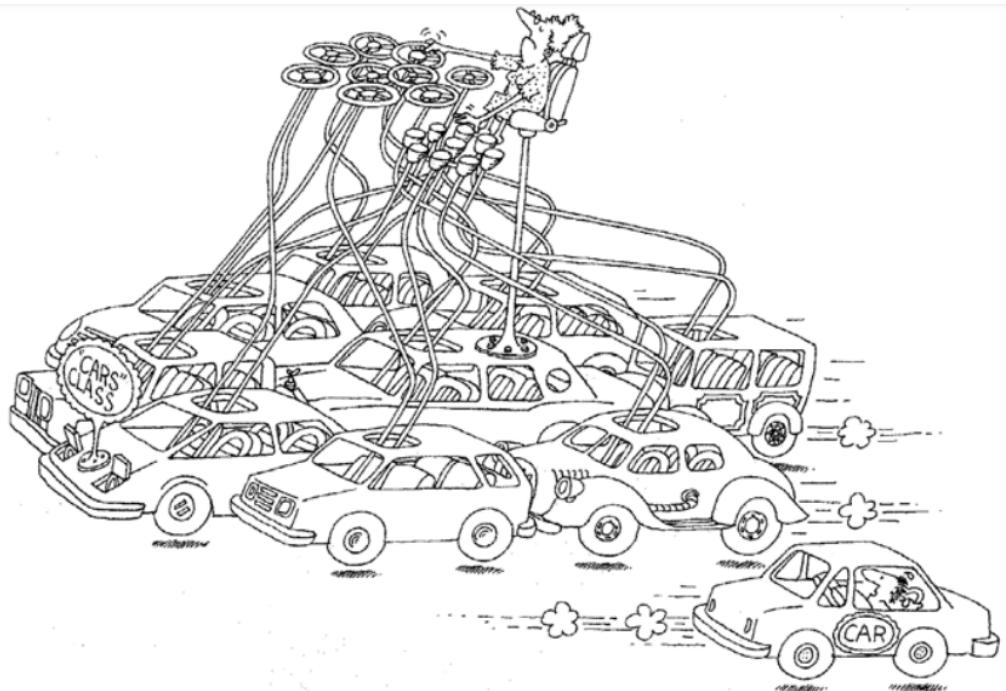
An object has state, exhibits some well-defined behavior, and has a unique identity.

- Reference:  
<https://zjnu2017.github.io/OOAD/reading/Object.Oriented.Analysis.and.Design.with.Applications.3rd.Edition.by.Booch.pdf>

What is class and instance?

## Class

- Definition:
  - Class is a collection of fields and methods.
  - Structure and behavior of the instance depends on class hence class is considered as a template model or blueprint for an instance.
  - Class represents set of instances which is having common structure and common behavior.
- Example: Car, Book, Laptop, Mobile Phone etc.
- Class is imaginary / logical entity.
- Class implementation represents encapsulation.
- Reference:  
<https://zjnu2017.github.io/OOAD/reading/Object.Oriented.Analysis.and.Design.with.Applications.3rd.Edition.by.Booch.pdf>



A class represents a set of objects that share a common structure and a common behavior.

## Instance

- Definition:
  - An entity which is having physical existence is called as instance.
  - In C++, physical entity is called as object and in Java it is called instance.
  - An entity which is having state, behavior and identity is called as instance.

- Example: Tata Nano, "Java Certification-Khalid Mughal", "MacBook Air", "iPhone-14"
- Process of creating instance is called as instantiation.
- By creating instance we are achieving abstraction.

null literal and NullPointerException

```
#define NULL ((void*)0) //C
#define NULL 0 //C++
```

- NULL is macro which used to initialize pointer.

```
int *ptr = NULL;
```

- Consider literals in Java:
  - true / false: boolean literals
  - 0,1,9: integer literals
  - 3.14: double literal
  - "DAC": String literal
  - null
- null is literal in Java, which is used to initialize reference.

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

- If reference contains, null value then reference variable is called as null reference variable / null object.
- Consider following code:

```
class Program{
    public static void main(String[] args) {
        Employee emp;
        emp.acceptRecord(); // error: variable emp might not have been
        initialized
        emp.printRecord();
    }
}
```

- Solution:

```
class Program{
    public static void main(String[] args) {
        Employee emp = null; //emp is null object
```

```
    emp.acceptRecord(); //java.lang.NullPointerException  
    emp.printRecord();  
}  
}
```

- Using null object, if we try to access any instance member of the class then JVM throws NullPointerException.
- Reference: <https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html>

## Difference between value type and reference types

- Reference: <https://docs.oracle.com/javase/tutorial/reflect/class/index.html>

### Value type

- Primitive type is also called as value type.
- There are 8 value/primitive types in java.
  - boolean, byte, char, short, int, float, double, long
- Variable of value type contains value.

```
int num1 = 10;
```

- In case of copy operation, value gets copied.

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

- In case of fields, variable of value type by default contains 0 value.

```
class Date{  
    int day; //0  
    int month; //0  
    int year; //0  
}
```

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

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

- 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( by considering it as method local variable ).

## Reference type

- Not primitive type is also called as reference type.
- There are 4 reference/non primitive types in java.
  - interface, class, enum, array
- Variable of reference type contains reference.

```
Integer i = new Integer(10);
```

- In case of copy operation, reference gets copied.

```
Integer i1 = new Integer( 10 );
Integer i2 = i1;
```

- In case of fields, variable of reference type by default contains null value.

```
class Employee{
    String name; //null
    Date joinDate; //null
}
```

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

```
Integer number = null; //OK
```

- To create instance of reference type, we must use new operator.

```
Integer i = new Integer( 10 );
Employee emp = new Employee( );
```

- Instance of reference type get space on Heap section.

## Memory allocation for reference variable

```

class Date{
    int day;
    int month;
    int year;
    //TODO
}

class Employee{
    static Scanner sc; //Static field
    //sc is reference variable and it will get space on method area.
    int empid; //Field
    Date joinDate; //Non Static Field
    //joinDate will get space inside Employee instance; i.e on Heap section.
}

class Program {
    public static void main(String[] args) {
        Employee emp = new Employee();
        //Employee emp=> It is object reference. Since it is method local
        varibale it will get space on Java Stack.
        //new Employee( ); => It is Date instance. It will get space on
        Heap.
    }
    public static void main1(String[] args) {
        Date joinDate = new Date( );
        //Date joinDate; => It is object reference. Since it is method local
        varibale it will get space on Java Stack.
        //new Date( ); => It is Date instance. It will get space on Heap.
    }
}

```

- Method local reference variable get space on Java stack.
- Non static reference field get space inside instance on heap.
- Static reference field get space on method area.

## Method Overloading

- Consider code in C programming language:

```

void sum( int num1, int num2 ){
    int result = num1 + num2;
    printf("Result : ",result);
}

void add( double num1, double num2 ){
    double result = num1 + num2;
    printf("Result : ",result);
}

int main( void ){
    sum( 10, 20 );
    add( 10.1, 20.2 );
    return 0;
}

```

- According to oops, If implementation of any method is logically same/equivalent then we should give same name to the method.
- If we want to give same name to the method then we should follow some rules:
  - If we want to give same name to the methods and if type of all the parameters are same then number of parameters passed to the method must be different.

```

class Program {
    static void sum( int num1, int num2 ){ // 2 method parameters
        int result = num1 + num2;
        System.out.println("Result : "+result);
    }
    static void sum( int num1, int num2, int num3 ){ // 3 method
parameters
        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 );
    }
}

```

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

```

class Program {
    static void sum( int num1, int num2 ){ // 2 method parameters
        int result = num1 + num2;
        System.out.println("Result : "+result);
    }
    static void sum( int num1, double num2 ){ // 2 method
parameters
        double result = num1 + num2;
        System.out.println("Result : "+result);
    }
    public static void main(String[] args) {
        Program.sum( 10, 20 );
        Program.sum( 10, 20.5 );
    }
}

```

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

```

class Program {
    static void sum( int num1, float num2 ){ // 2 method

```

```

parameters
    float result = num1 + num2;
    System.out.println("Result : "+result);
}
static void sum( float num1, int num2 ){ // 2 method
parameters
    float result = num1 + num2;
    System.out.println("Result : "+result);
}
public static void main(String[] args) {
    Program.sum( 10, 20.2f );
    Program.sum( 10.1f, 20 );
}
}

```

- Only on the basis of different return type, we can not give same name to methods.

```

class Program {
    static void sum( int num1, int num2 ){ // 2 method
parameters
    int result = num1 + num2;
    System.out.println("Result : "+result);
}
static int sum( int num1, int num2 ){ // 2 method parameters
    int result = num1 + num2;
    return result;
}
public static void main(String[] args) {
    Program.sum( 10, 20 );
    Program.sum( 10, 20 );
}
}

```

- When we define multiple methods with same name using above rules then it is called as method overloading.
- In other words, process of defining methods with same name and different signature is called as method overloading.
- Methods, which are taking part in method overloading are called as overloaded methods.
- If implementation of methods are functionally equivalent / same then we should overload method i.e. we should give same name to the methods.
- Return type is not considered in method overloading.
  - return value from method as well as catching value from method is optional hence return type is not considered in method overloading.

## Constructor

### Constructor chaining

### Instance initialization block

Need of getter and setter methods

# Day 5

## Coding conventions:

- Camel Case
  - Camel case is a naming convention in which the first letter of each word in a compound word is capitalized, except for the first word.
  - Example: showInputDialog
- Pascal Case
  - Pascal case is a naming convention in which the first letter of each word in a compound word is capitalized.
  - Example: ArrayIndexOutOfBoundsException
- Snake Case
  - Snake case combines words by replacing each space with an underscore (\_).
  - Example: MAX\_VALUE
- Kebab Case
  - Kebab case is the way to write compound words separated by hyphens (-) instead of using space. Generally, everything is written in lowercase.
  - Example: "what-is-kebab-case"

## Declaration and Definition

- According to Dennis Ritchie, declaration refers to the place where nature of the variable is stated but no storage is allocated.

```
int main( void ){
    int num1; //Declaration as well as definition
    extern int num2; //Declaration
    int num3 = 30; //Declaration as well as definition
    printf("%p\n", &num1); //OK
    printf("%p\n", &num2); //Linker Error
    return 0;
}
```

- Definition refers to the place where storage is allocated / assigned.

## Scope in C

- Block scope
- Function scope
- Prototype scope
- File scope
- Program scope

```

int num6 = 60; //Program Scope
static int num5 = 30; //File Scope
int main( void ){
    //Function Declaration
    int sum( int num1, int num2 ); //Prototype scope

    int num3 = 10; //Function Scope
    //Start new block here
    {
        int num4 = 20; //Block Scope
    }
    sum( 10, 20 ); //Function Call
    return 0
}
int sum( int num1, int num2 ){ //Function Definition
    return num1 + num2;
}

```

## Initialization( Revision )

- Process of providing value to the variable during its declaration is called as initialization.

```
int number = 10; //Initialization
```

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

- We can initialize any variable only

## Constructor

- A block of code( which looks like method but it is not considered as method ) which is used to initialize instance, is called as constructor of the class.
- Constructor is considered as special syntax of the class due to following reasons:
  - Its name is same as class name.
  - It doesn't have any return type.
  - It gets called implicitly.
  - In the lifetime of instance, constructor gets called only once.

```

class Date{
    int day;
    int month;
    int year;
    Date( ){ //Constructor
        LocalDate ld = LocalDate.now();
    }
}

```

```

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

```

- For the reference, JVM do not call constructor.
- JVM give call to the constructor once per instance.
- We can not call constructor on instance explicitly. It is designed to call implicitly.

```

import java.time.LocalDate;
import java.util.Scanner;
class Date{
    int day;
    int month;
    int year;
    Date( ){ //Constructor
        System.out.println("Inside constructor");
        LocalDate ld = LocalDate.now();
        this.day = ld.getDayOfMonth();
        this.month = ld.getMonthValue();
        this.year = ld.getYear();
    }
}
class Program{
    public static void main(String[] args) {
        Date dt1 = new Date(); //Here constructor will call implicitly
        //dt1.Date( ); //Not allowed
    }
    public static void main1(String[] args) {
        //Date dt1 = null; //Here constructor will not be called
        //new Date( ); //Here constructor will call on anonymous
        instance
        Date dt2 = new Date(); //Here constructor will call on only once
    }
}

```

- Reference: <https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html>

## Types of constructor

- There are 3 types of constructor in Java:
  - Parameterless constructor
  - Parameterized constructor
  - Default constructor

### Parameterless constructor

- It is also called as zero argument constructor / programmer defined default constructor.

- Consider example:

```
Date( ){ //Parameterless Constructor
    this.day = 0;
    this.month = 0;
    this.year = 0;
}
```

- A constructor which do not take any parameter is called as Parameterless constructor.
- Consider following code snippet:

```
Date dt = new Date();
```

- If we create instance without passing arguments( i.e. we are passing 0 arguments ) then Parameterless constructor gets called.

### Parameterized constructor

- A constructor which take parameter is called as Parameterized constructor.
- Consider example:

```
Date( int day, int month, int year ){ //Parameterized Constructor
    System.out.println("Inside constructor");
    this.day = day;
    this.month = month;
    this.year = year;
}
```

- Consider following code snippet:

```
Date dt1 = new Date( 23, 7, 1983 );
```

- If we create instance by passing arguments then parameterized constructor gets called.
- We can define multiple constructors inside class. It is called as constructor overloading.

```
class Date{
    int day;
    int month;
    int year;
    Date(){
        System.out.println("Inside parameterless constructor");
```

```

        this.day = 0;
        this.month = 0;
        this.year = 0;
    }
    Date( int day, int month, int year ){ //Constructor
        System.out.println("Inside parameterized constructor");
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
class Program{
    public static void main(String[] args) {
        Date dt1 = new Date( 23, 7, 1983 );

        Date dt2 = new Date( );
    }
}
/*
sandeep@Sandeeps-MacBook-Air Day_5.5 % javac Program.java
sandeep@Sandeeps-MacBook-Air Day_5.5 % java Program
Inside parameterized constructor
Inside parameterless constructor
*/

```

- Constructor calling sequence depends on order of instance declaration.

### Default constructor

- If we do not define any constructor( no parameterless / no parameterized ) inside class then compiler generates once constructor for the class by default. It is called as default constructor.
- Default constructor is zero argument constructor.
- Consider following code:

```

class Date{
    int day;
    int month;
    int year;
}
class Program{
    public static void main(String[] args) {
        Date dt1 = new Date( ); //OK
        //Here on instance, default constructor will call
    }
}

```

- Consider following code:

```

class Date{
    int day;
    int month;
    int year;
}
class Program{
    public static void main(String[] args) {
        Date dt1 = new Date(); //OK
        //Compiler can generate only default parameterless constructor

        Date dt2 = new Date( 1,4,2023); //NOT OK
        //Compiler never generate default parameterized constructor
    }
}

```

- Consider following code:

```

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

    Date( int day, int month, int year ){ //parameterized constructor
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
class Program{
    public static void main(String[] args) {
        Date dt1 = new Date(); //Error
        Date dt2 = new Date( 1,4,2023);
    }
}

```

- Consider constructors of java.lang.Integer class

- public Integer(int value)

```

//Integer i1 = new Integer(); //Not OK
int value = 123;
Integer i1 = new Integer( value ); //OK
Integer i2 = new Integer( 123 ); //OK

```

- public Integer(String s) throws NumberFormatException

```
String s = "123";
Integer i1 = new Integer( s ); //OK
Integer i2 = new Integer( "123" ); //OK
Integer i3 = new Integer( "a1b2c3" ); //NumberFormatException
```

- Consider constructors of java.lang.String class

- public String()

```
String s1 = new String( );
```

- public String(String original)

```
String str = "Hello";
String s2 = new String( str );
String s3 = new String( "Hello" );

String s1 = new String( "Hi" );
String s2 = new String( s1 );
```

## Explicit constructor invocation

- To reuse body of existing constructor we can call constructor from another constructor. It is called as constructor chaining.
- For constructor chaining we should use this statement.

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

    Date( ){
        //Explicit call to the constructor
        this( 1, 4, 2023 ); //Constructor chaining
    }
    Date( int day, int month, int year ){
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
```

- this statement must be first statement inside constructor.

- Consider example of Stack:

```
class Stack{
    int top;
    int[] arr;
    Stack( ){
        this( 5 ); //Constructor chaining
    }
    Stack( int size ){
        this.top = -1;
        this.arr = new int[ size ];
    }
}
```

- Reference: <https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html>

## Stack

- Data structure describes two things:
  - How to organize data in ram.
  - Which operations should be used to organise data inside RAM.
- Types of data structure:
  - Linear / sequential data structure
    - Array, Stack, Queue, LinkedList etc.
  - Non Linear data structure
    - Tree, Graph, Hashtable etc.
- Stack is a linear data structure in which we can manage elements in Last In First Out order( LIFO ).
- We can perform following operations on Stack:
  - boolean empty( );
  - void push( int element );
  - int peek( );
  - boolean full( );
  - void pop();
- Value stored inside data structure is called element.
- Consider syntax of array:

```
int[] arr = new int[ 3 ]; //Single dimensional array
```

## Instance initialization block

- Consider following code:

```
class Stack{
    int top;
    int[] arr;
```

```

    //Instance initialization block
    this.top = -1;
}
Stack( ){
    //this.top = -1;
    this.arr = new int[ 5 ];
}
Stack( int size ){
    //this.top = -1;
    this.arr = new int[ size ];
}
}

```

- Reference: <https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html>

## Access Modifier

- In Java, parent class is called as super class and child class is called as sub class.
- The modifiers, which are used to control visibility of the members of the class is called access modifiers.
- There are 4 access modifiers in Java:
  - private
  - package level private ( no modifier / default means --> package level private )
  - protected
  - public

Sr.No.	Access Modifier	Inside same package			Inside different Package	
		Same Class	Sub Class	Non Sub Class	Sub Class	Non Sub Class
1	<b>private</b>	A	NA	NA	NA	NA
2	<b>package level private</b>	A	A	A	NA	NA
3	<b>protected</b>	A	A	A	A	NA
4	<b>public</b>	A	A	A	A	A

- Reference: <https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

## Need of Getter and Setter method

- Process of declaring fields of the class private is called as data hiding.

```

class Student{
    //Data hiding
    private String name;
    private int rollNumber;
    private float marks;
}

```

- Data hiding is also called as data encapsulation.

- Process of giving controlled access to the data is called as data security.

```

class Student{
    private String name;
    private int rollNumber;
    private float marks;
    public void setMarks( float marks ){
        if( marks > 100 )
            throw new IllegalArgumentException("Invalid marks.");
        this.marks = marks;
    }
    //acceptRecord
}
class Program{
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setMarks( 101 );
    }
}

```

## Final modifier

## Absolute path and relative path

- Reference: <https://www.redhat.com/sysadmin/linux-path-absolute-relative>

## Path and classpath

- Reference: <https://docs.oracle.com/javase/tutorial/essential/environment/paths.html>

## java.lang.Object class

- Reference: <https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>
- Use below command on terminal to get more details:

```
javap java.lang.Object
```

## toString method

## Day 6

```
class Employee{  
    //TODO : Fields  
    //TODO : Ctor  
    //TODO : Getters and Setters  
}  
class EmployeeTest{  
    /* Employee emp;  
    public EmployeeTest(){  
        emp = new Employee();  
    } */  
    private Employee emp = new Employee( );  
    //TODO: acceptRecord()  
    //TODO: printRecord()  
}  
class Program{  
    public static void main(String[] args) {  
        EmployeeTest test = new EmployeeTest( );  
        //TODO: accept and print  
    }  
}
```

### Final Modifier

#### Final method local variable

```
class Program{  
    public static void main(String[] args) {  
        int number = 10; //Initialization  
        number = number + 5;  
        System.out.println("Number : "+number); //15  
    }  
}
```

- After storing value inside variable, if we don't want to modify its value then we should use final modifier.
- In java, we can declare local variable final.

```
class Program{  
    public static void main(String[] args) {  
        final int number = 10; //Initialization  
        number = number + 5; //error: cannot assign a value to final  
        variable number  
        System.out.println("Number : "+number);
```

```
}
```

```
}
```

```
class Program{
    public static void main(String[] args) {
        final int number;
        number = 10; //Assignment
        //number = number + 5; //error: variable number might already
        have been assigned
        System.out.println("Number : "+number); //10
    }
}
```

```
public static void main(String[] args) {
    final int number = 10; // //Initialization
    //number = 20; //error: cannot assign a value to final variable
    number
    System.out.println("Number : "+number); //10
}
```

```
class Program{
    private static Scanner sc = new Scanner(System.in);
    public static void main(String[] args) {
        System.out.print("Number : ");
        final int number = sc.nextInt(); // //Initialization
        //number = 20; //error: cannot assign a value to final variable
        number
        System.out.println("Number : "+number); //10
    }
}
```

- We can provide value to final variable before compilation as well as after compilation.

## Final field

- We can declare field final.
- After initialization, if we don't want to modify value of any field inside any method of the class including constructor body then we should declare field final.
- We must provide value to the final field either inside constructor body or at the time of declaration.

```
class Test{
    //Field
    private final int number; //NOT OK
}
```

```
class Test{
    //Field
    private final int number = 10; //OK
}
```

```
class Test{
    //Field
    private final int number;
    public Test( ){
        this.number = 10; //OK
    }
}
```

```
class Test{
    //Field
    private final int number = 10; //OK
    public Test( ){
        this.number = 20; //Not OK
    }
}
```

- Generally we should use snake case convention for final field.

```
class Test{
    private final int NUMBER = 10; //Field
    public void showRecord( ){
        //this.NUMBER = this.NUMBER + 1; //Not OK
        System.out.println("Number : "+this.NUMBER);
    }
    public void printRecord( ){
        //this.NUMBER = this.NUMBER + 1; //Not OK
        System.out.println("Number : "+this.NUMBER);
    }
}
class Program{
    public static void main(String[] args) {
        Test t = new Test();
        t.showRecord(); //10
        t.printRecord(); //10
        t.showRecord(); //10
    }
}
```

- We will discuss final method and final class in inheritance.
  - We can declare method final.
    - We can not redefine/override final method inside sub class.
  - We can declare class final.
    - We can not create sub class of final class.

**What is the meaning of "final Complex c1 = new Complex( 10, 20);" statement?**

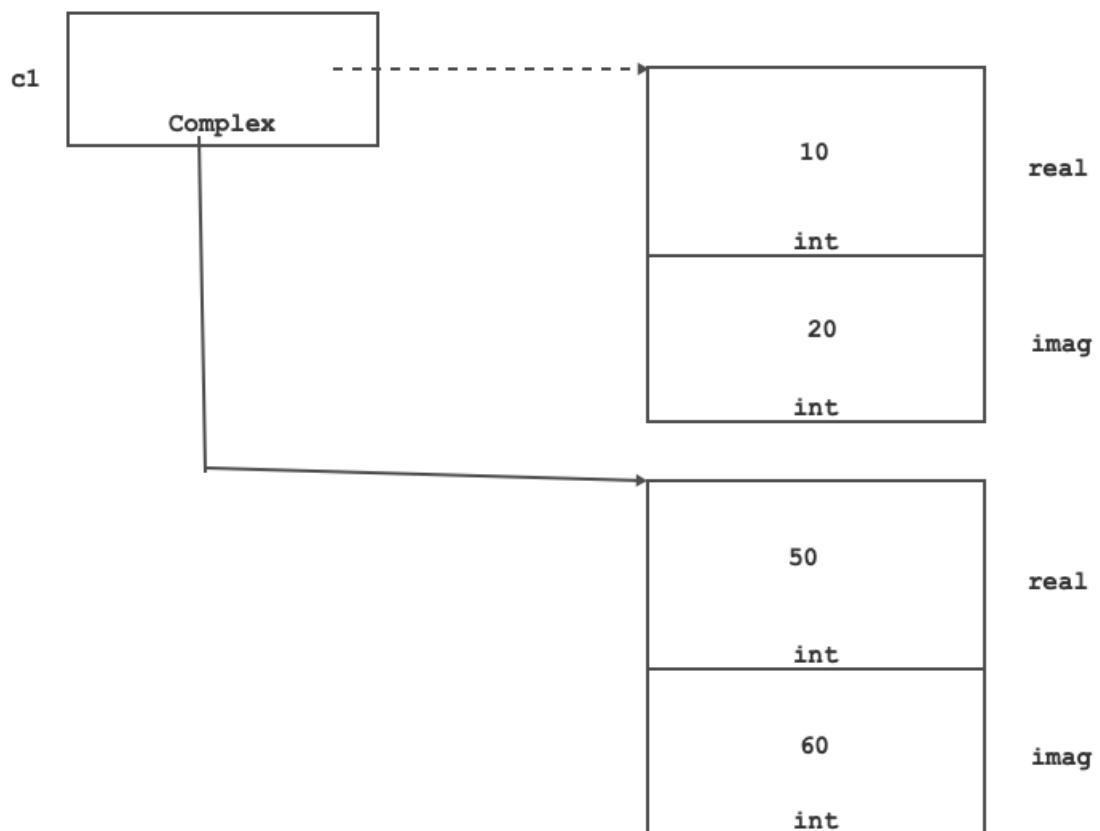
```

class Complex{
    private int real;
    private int imag;
    public Complex( ){
        this( 0, 0);
    }
    public Complex( int real, int imag ){
        this.real = real;
        this.imag = imag;
    }
    public int getReal() {
        return this.real;
    }
    public void setReal(int real) {
        this.real = real;
    }
    public int getImag() {
        return this.imag;
    }
    public void setImag(int imag) {
        this.imag = imag;
    }
}
class Program{
    public static void main(String[] args) {
        final Complex c1 = new Complex( 10, 20);
        c1.setReal(100); //OK
        c1.setImag(200); //OK
        System.out.println("Real Number : "+c1.getReal()); //100
        System.out.println("Imag Number : "+c1.getImag()); //200

        c1 = new Complex( 50, 60); //error: cannot assign a value to final
variable c1
    }
}

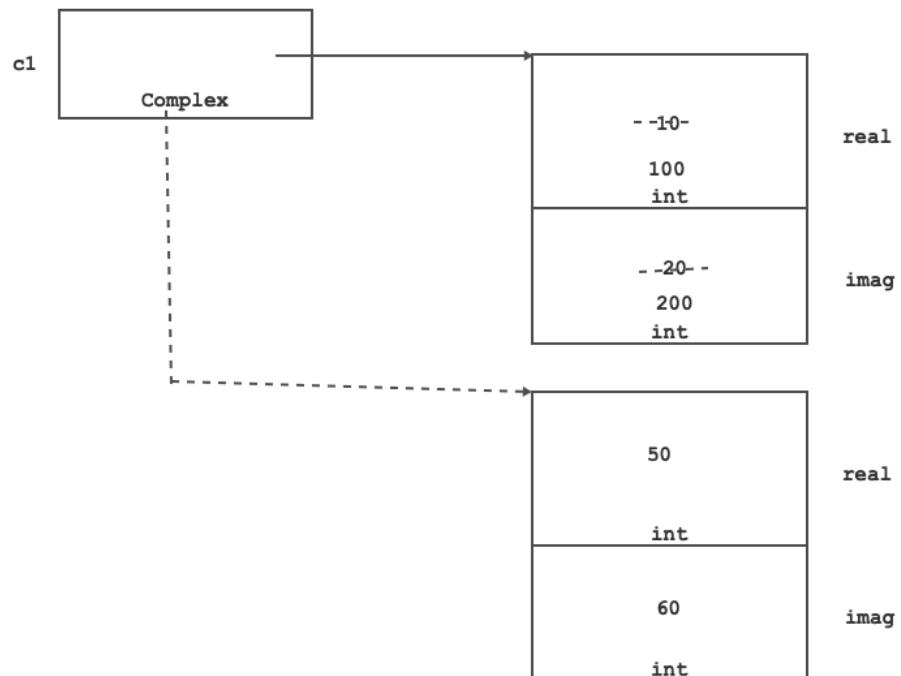
```

```
Complex c1 = new Complex( 10, 20); //Initialization of c1  
c1 = new Complex( 50, 60); //Assignment of c1
```



- In Java, we can declare reference final but we can not declare instance final.

```
final Complex c1 = new Complex( 10, 20); //Initialization of c1
c1.setReal( 100 );
c1.setImag( 200 );
//c1 = new Complex( 50, 60); //Not OK
```



## Absolute path versus Relative Path

### Absolute path

- Example: D:\CDAC\Sandeep\Java\Day6\Day\_6.1\src\Program.java.
- A path of a file / directory from root directory is called as absolute path.

### Relative Path

- .. represents parent directory
- . represents current directory
- Change directory: cd D:\CDAC\Sandeep\Java\Day6\Day\_6.1
- Example : .\src\Program.java
- Path of a file / directory from current directory is called as relative path.

## Path versus Classpath

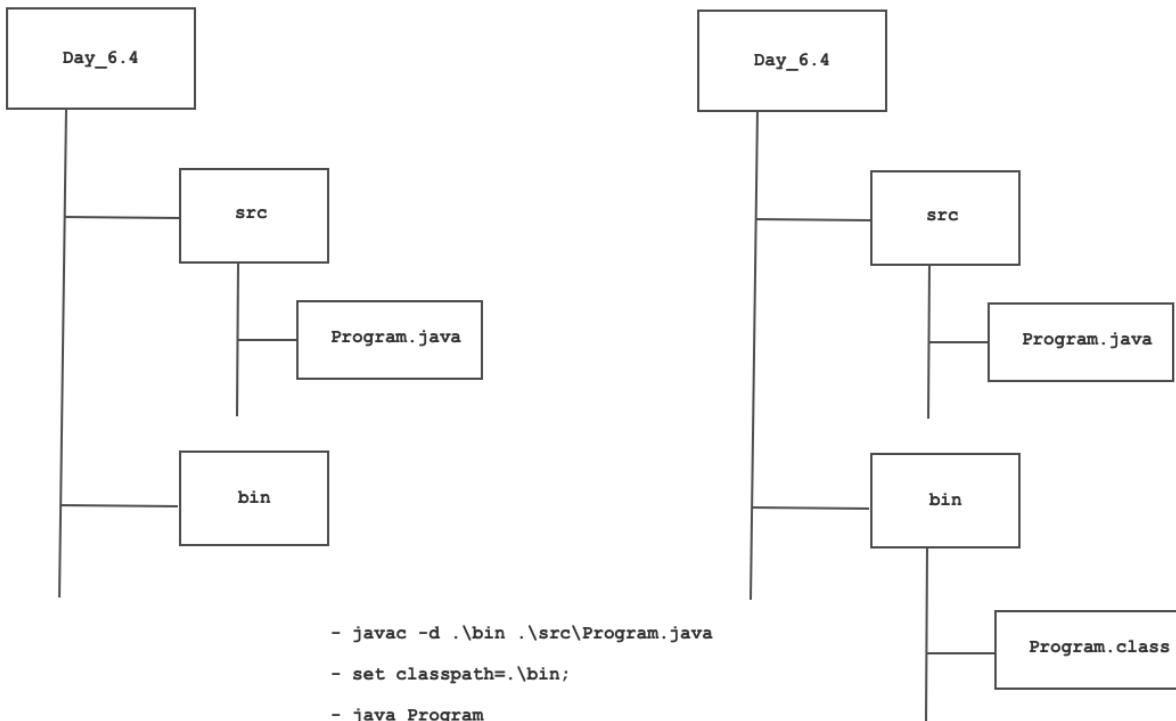
### Path

- Path is OS platforms environment variable which is used to locate java development tools.
- How to set path?
  - set path="C:\Program Files\Java\jdk\_1.8.0\_361\bin"; //In MS Windows
  - export PATH=/usr/bin/

### Classpath

- Classpath is Java platforms environment variable which is used to locate .class / .jar files
- How to set classpath?
  - set classpath=.\bin;
  - export CLASSPATH=./bin;

[ Compilation and execution should be done from Day\_6.4 ]



## Introduction of Object class

```
abstract class Number{ //Parent class / Super class
    //TODO
}
final class Integer extends Number{ //Child class / Sub class
    //TODO
}
```

- In Java, parent class is called as super class and child class is called as sub class.
- Object is a concrete class declared in java.lang package.
- java.lang.Object do not extend any class or do not implement any interface. In other words, it is super class of all the classes( not interfaces ) in core Java.
- It is also called as ultimate base class / super cosmic base class / root of java class hierarchy.

```
abstract class Number extends Object{
    //TODO
}
final class Integer extends Number{
    //TODO
}
```

```

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

```

- Any class can contain:
  - Nested types( interface / class / enum )
  - Field
  - Constructor
  - Methods
  - Blocks( instance / static )
- java.lang.Object class do not contain any nested type and field.
- java.lang.Object class contains only parameterless constructor.

```

Object o1 = new Object("Sandeep"); //Not OK
Object o2 = new Object(); //OK

```

- There are 11 methods( 5 non final( 2 native + 3 non native ) + 6 final methods( 4 native + 2 non native methods)) in java.lang.Object class.
- Use following command to check signature of methods:

```
javap java.lang.Object
```

- 5 Non final methods of java.lang.Object class( we can override these methods inside sub class )
  - public String toString( );
  - public boolean equals( Object obj );
  - public native int hashCode( );
  - protected native Object clone( )throws CloneNotSupportedException
  - protected void finalize( )throws Throwable
- 6 Final methods of java.lang.Object class
  - public final native Class<?> getClass()
  - public final void wait( ) throws InterruptedException
  - public final native void wait( long timeout ) throws InterruptedException
  - public final void wait( long timeout, int nanos ) throws InterruptedException
  - public final native void notify();
  - public final native void notifyAll();

## Overriding toString method

- `toString` is a non final method of `java.lang.Object` class

```
public String toString();
```

- If we want to return state of the instance in String form then we should use `toString` method.
- If we do not define `toString()` method inside class then super class's `toString` method will call. If any super class do not contain `toString()` method then `Object` class's `toString()` method will call.
- Consider definition of `toString` method from `java.lang.Object` class:

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

- `toString()` method of `java.lang.Object` class returns String in following form:

F.Q.Class name@Hexadecimal HashCode

- Hashcode is not an address / reference of the instance. It is a logical integer number which can be generated by processing state of the instance.
- According to client's requirement, If implementation of super class method is partially complete then we should redefine/override method inside sub class.
- If we dont want hashCode then we should override `toString()` method inside sub class.

```
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.

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

```
public String toString(){  
    return String.format("%-25s%-10d%-10.2f", this.name, this.empid,  
    this.salary);  
}
```

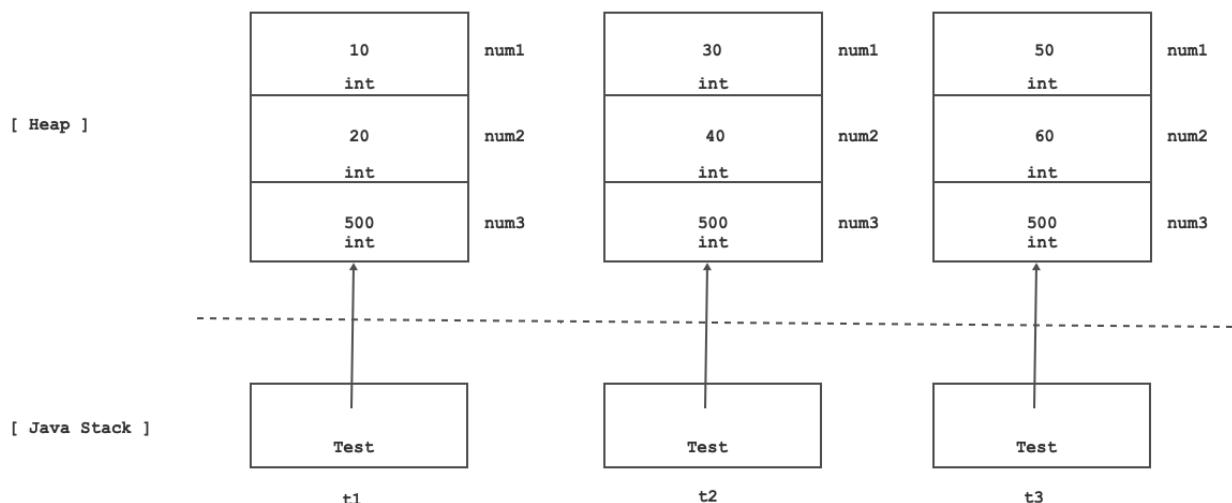
## Static Field

- If we create instance of the class then all the fields get space inside instance.

```

class Test{
    private int num1;
    private int num2;
    private int num3;
    public Test( int num1, int num2 ){
        this.num1 = num1;
        this.num2 = num2;
        this.num3 = 500;
    }
}
class Program {
    public static void main(String[] args) {
        Test t1 = new Test( 10, 20 ); //10,20,500
        Test t2 = new Test( 30, 40 ); //30,40,500
        Test t3 = new Test( 50, 60 ); //50,60,500
    }
}

```



- If we want to share value of any field inside all the instances of same class then we should declare field static.
- A field of the class, which get space inside instance is called as instance variable. In other words, only non static field get space inside instance. Hence non static field is also called as instance variable.
- Instance variable get space once per instance on heap memory.
- To use instance variable we must use object reference.
- A field of the class which do not get space inside instance is called as class level variable. In other words, static fields do not get space inside instance. Hence static field is also called as class level variable.

- Class level variable get space during class loading once per class on method area.
- To access class level variable we should use class name and dot operator.

```
class X{
    private int num1;
    private int num2;
    private static int count = 500;
}
X x1 = new X(10,20); //10,20, 500
X x2 = new X(30,40); //30,40,500
X x3 = new X(50,60); //50,60,500
```

```
class Y{
    private int num3;
    private int num4;
    private static int count = 600;
}
Y y1 = new Y(11,12); //11,12,600
Y y2 = new Y(13,14); //13,14,600
Y y3 = new Y(15,16); //15,16,600
```

```
class Z{
    private int num5;
    private int num6;
    private static int count = 700;
}
Z z1 = new Z(21,22); //21,22,700
Z z2 = new Z(23,24); //23,24,700
Z z3 = new Z(25,26); //25,26,700
```

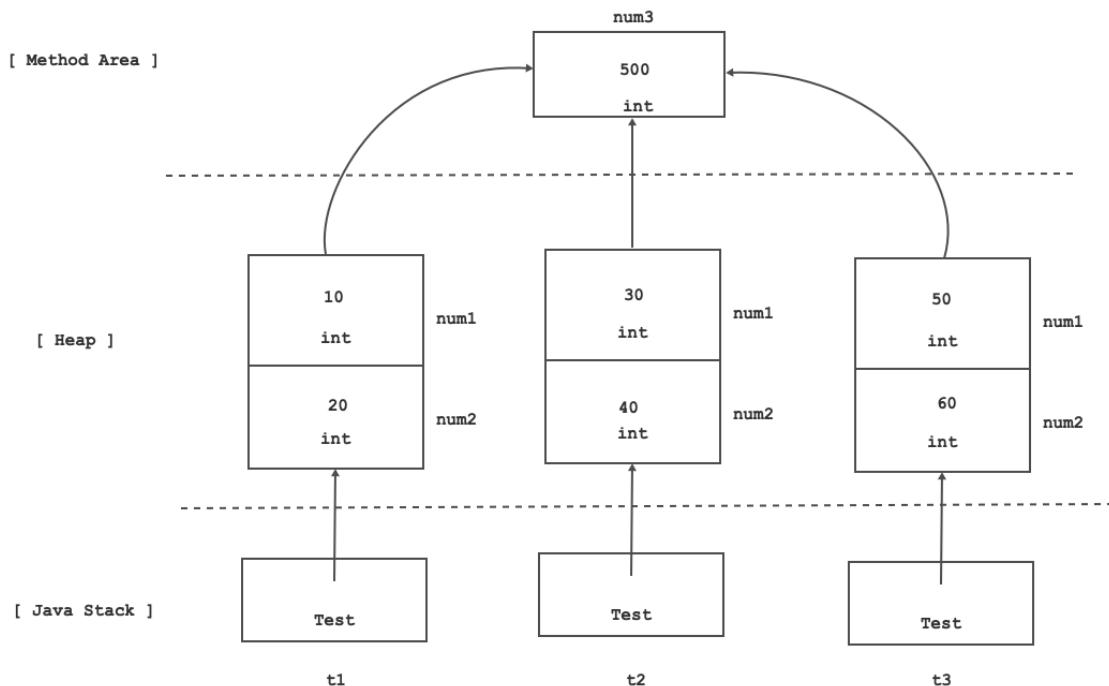
```
class Test{
    private int num1;
    private int num2;
    private static int num3;

    //Static initialization block
    static {
        Test.num3 = 500;
    }
    public Test( int num1, int num2 ){
        this.num1 = num1;
        this.num2 = num2;
    }
    public void printRecord( ){
        System.out.println("Num1 : "+this.num1);
    }
}
```

```

        System.out.println("Num2 : "+this.num2);
        System.out.println("Num3 : "+Test.num3);
    }
}
class Program {
    public static void main(String[] args) {
        Test t1 = new Test( 10, 20 ); //10,20,500
        t1.printRecord();
    }
}

```



## Static Initializer block

- To initialize non static fields we should use constructor and to initialize static field we should use static initialization block.

```

class Test{
    private int num1;
    private int num2;
    private static int num3;

    //Static initialization block
    static {
        Test.num3 = 500;
    }
    public Test( int num1, int num2 ){
        this.num1 = num1;
        this.num2 = num2
    }
}

```

- We can define multiple static blocks inside class. In this case JVM execute it sequentially.

## Static Method

- To access non static members of the class we should define non static method inside class and to access static members of the class we should define static method inside class.
- Non static method is designed to call on instance hence it is called as instance method and static method is designed to call on class name hence it is called as class level method.
- To access instance method we should use instance and to access class level method we should use class name and dot operator.
- According to Java concept, this reference is considered as a link/connection between instance variable and instance method.
- Static method do not get this reference.

### Why static method do not get this reference?

- If we call non static method on instance then non static method get this reference.
- Static method is designed to call on class name.
- Since static method is not designed to call on instance, it doesn't get this reference.
- Since static method do not get this reference, we can not access non static members directly inside static method.
- In other words, static method can access only static members of the class.

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

- Using instance, we can access non static members inside static method.

## Count instances of the class

### Singleton class

## Day 7

### static method

- Inside method, if there is need to use this reference then method should be non static otherwise method should be static.

```
class Program{
/* public double power( double base, int index ){
    double result = 1;
    for( int count = 1; count <= index; ++ count )
        result = result * base;
    return result;
} */
public static double power( double base, int index ){
    double result = 1;
    for( int count = 1; count <= index; ++ count )
        result = result * base;
    return result;
}
public static void main(String[] args) {
    //Program p = new Program();
    //double result = p.power(2, 3);

    double result = Program.power(2, 3); //OK
    //double result = power(2, 3); //OK
    System.out.println("Result : "+result);
}
}
```

- If constructor is public then we can create instance of the class inside method of same class as well as method of different class.

```
class Complex{
    private int real;
    private int imag;
    public Complex( ){
        System.out.println("Inside constructor.");
        this.real = 10;
        this.imag = 20;
    }
    public static void test( ){
        Complex c2 = new Complex(); //OK
    }
}
class Program{
    public static void main(String[] args) {
        Complex c1 = new Complex(); //OK
        Complex.test();
    }
}
```

```
    }
}
```

- We can declare constructor private.
- If constructor is private then we can create instance of the class inside method of same class only.

```
class Complex{
    private int real;
    private int imag;
    private Complex( ){
        System.out.println("Inside constructor.");
        this.real = 10;
        this.imag = 20;
    }
    public static void test( ){
        Complex c2 = new Complex(); //OK
    }
}
class Program{
    public static void main(String[] args) {
        //Complex c1 = new Complex(); //NOT OK
        Complex.test();
    }
}
```

## Instance Counter

```
class InstanceCounter{
    private static int count;
    public InstanceCounter( ){
        InstanceCounter.count = InstanceCounter.count + 1;
    }
    public static int getCount() {
        return InstanceCounter.count;
    }
}
class Program{
    public static void main(String[] args) {
        InstanceCounter c1 = new InstanceCounter();
        InstanceCounter c2 = new InstanceCounter();
        System.out.println("Instance Count : "+InstanceCounter.getCount());
    }
}
```

- Method local variable get space once per method call.

```

class Program{
    public static void printRecord( ){
        int count = 0; //Non Static Method Local Variable
        count = count + 1;
        System.out.println("Count : "+count);
    }
    public static void main(String[] args) {
        Program.printRecord(); //1
        Program.printRecord(); //1
        Program.printRecord(); //1
    }
}

```

- In Java, we can declare local variable final but we can not declare it static.

```

public static void printRecord( ){
    static int count = 0; //error: illegal start of expression
    count = count + 1;
    System.out.println("Count : "+count);
}

```

- Static variable is also called as class level variable. According to oops, class level variable should be declared inside class scope. Hence we can not declare local variable static. But we can declare field static.

```

class Program{
    private static int count = 0;
    public static void printRecord( ){
        count = count + 1;
        System.out.println("Count : "+count);
    }
    public static void main(String[] args) {
        Program.printRecord(); //1
        Program.printRecord(); //2
        Program.printRecord(); //3
    }
}

```

## Design pattern

- Reusable solution which is used to solve common problems during development is called as design pattern.
  - Creational
    - Abstract factory
    - Builder
    - Factory method

- Prototype
- Singleton
- Structural
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral
  - Chain of responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template method
  - Visitor

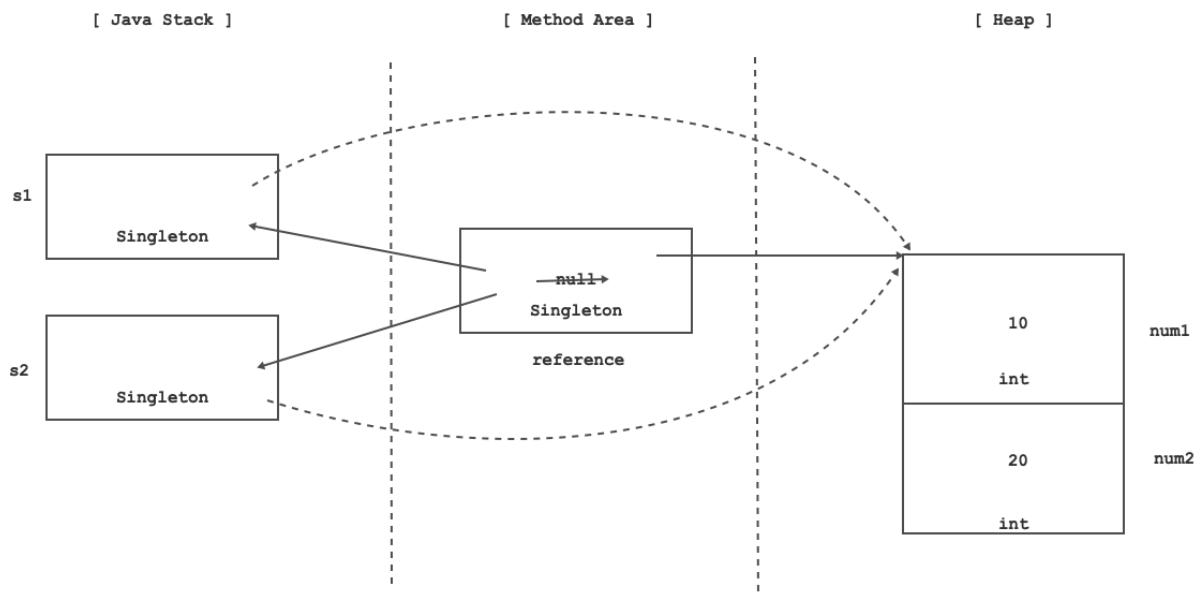
## Singleton class

- A class from which we can create only one instance is called as singleton class.
- Method 1:

```

class Singleton{
    private Singleton( ){
        //TODO
    }
    private static Singleton reference;
    public static Singleton getReference( ){
        if( reference == null )
            reference = new Singleton( );
        return reference;
    }
}
class Program{
    public static void main(String[] args) {
        Singleton s1 = Singleton.getReference();
        Singleton s2 = Singleton.getReference();
    }
}

```



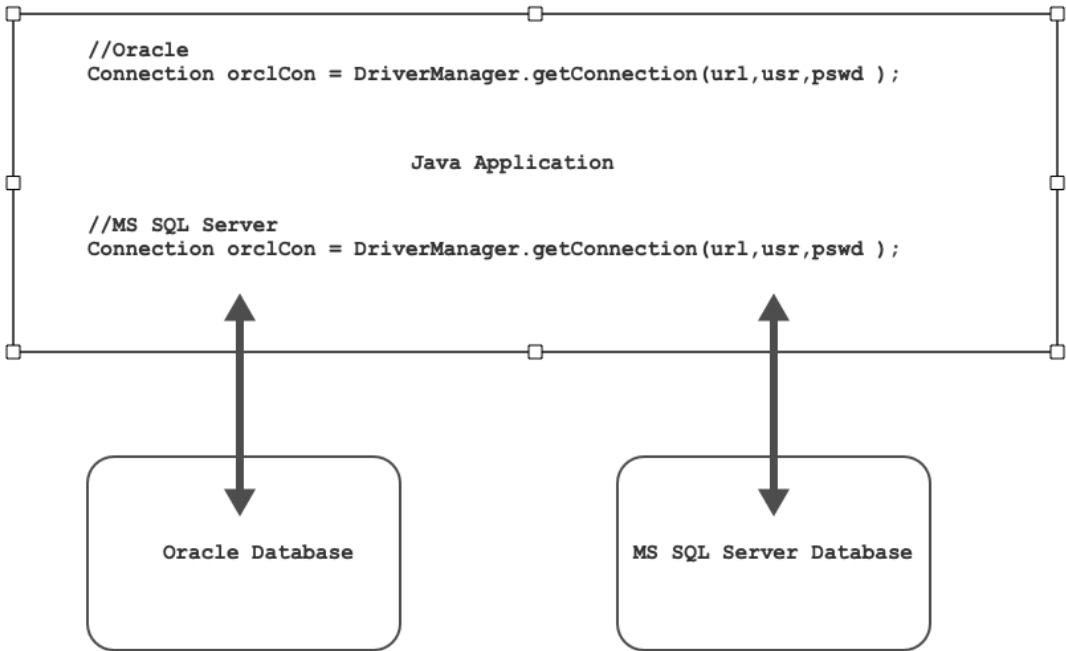
- Method 2

```

class Singleton{
    private int num1;
    private int num2;
    private static Singleton reference;
    static{
        reference = new Singleton();
    }
    public static Singleton getReference() {
        return reference;
    }
    private Singleton( ){
        this.num1 = 10;
        this.num2 = 20;
    }
}
class Program{
    public static void main(String[] args) {
        Singleton s1 = Singleton.getReference();
        Singleton s2 = Singleton.getReference();
    }
}

```

Package



- It is a Java language feature which is used:
  - To avoid name clashing/collision/ambiguity/conflict

```
java.util.Date
java.sql.Date
```

- To group functionally equivalent / related types together.
- Package is a logical concept. In other words we can not create instance of package.
- In Java, Package can contain:
  - Sub Package
  - Interface
  - Class
  - Enum
  - Exception
  - Error
  - Annotation Types
- Consider following example:

```
java.lang.Object
//java      : main package
//lang      : sub package
//Object   : Type name( interface, class....etc)
```

- package is a keyword in Java.
- How to define class inside package?
  - In C++

```
namespace std{
    class Complex{
        //TODO: Member declaration
    };
}
```

- In Java

```
package p1; //package declaration statement
class Complex{ //packaged class
    //TODO: Member declaration
};
```

- If we define any class inside package then it is called as packaged class.
- Package declaration statement must be first statement in .java file.

```
class Complex{
    //TODO: Member declaration
};
package p1; //Not OK
```

```
package p1; //OK
package p2; //Not OK
class Complex{
    //TODO: Member declaration
};
```

```
package p1; //OK
import java.util.Scanner;
import java.util.Date;
class Complex{
    //TODO: Member declaration
};
```

- If we do not define any class inside package then it is considered as a member of default package.

```
class Complex{
    //TODO: Member declaration
};
```

- Java compiler maps package name with the folder.
- If we want to use any type(class/interface) in different package then
  - Either we should use import statement
  - Or fully qualified type name
- Consider following code:

```
//File Name: Complex.java
package p1;
??? class TComplex { //Here ??? means package level private
    public String toString() {
        return "TComplex.toString( )";
    }
}
```

- If we do not specify access modifier then default modifier of class is always package level private.
- If access modifier of type is package level private then we can not access it outside the package.
- If we want to use any type outside the package then type must be public.
- According to Java Language Specification(JLS), name of the public class and name of the .java file must be same.

```
//File Name: Complex.java
package p1;
public class Complex {
    public String toString() {
        return "Complex.toString( )";
    }
}
```

- Since name of the public class and name of the .java file must be same, we can not define multiple public classes in single .java file.
- We can not declare class private or protected. In other words, access modifier of a class can be either package level private or public only.

## Demo 1

- Consider code in Complex.java

```
//File Name: Complex.java
package p1;
public class Complex {
    public String toString() {
        return "Complex.toString( )";
    }
}
```

- Compile Complex.java

```
javac -d .\bin .\src\Complex.java
```

- Consider code in Program.java

```
import p1.Complex;
class Program{
    public static void main(String[] args) {
        //p1.Complex c1 = new p1.Complex();
        Complex c1 = new Complex();
        System.out.println( c1.toString());
    }
}
```

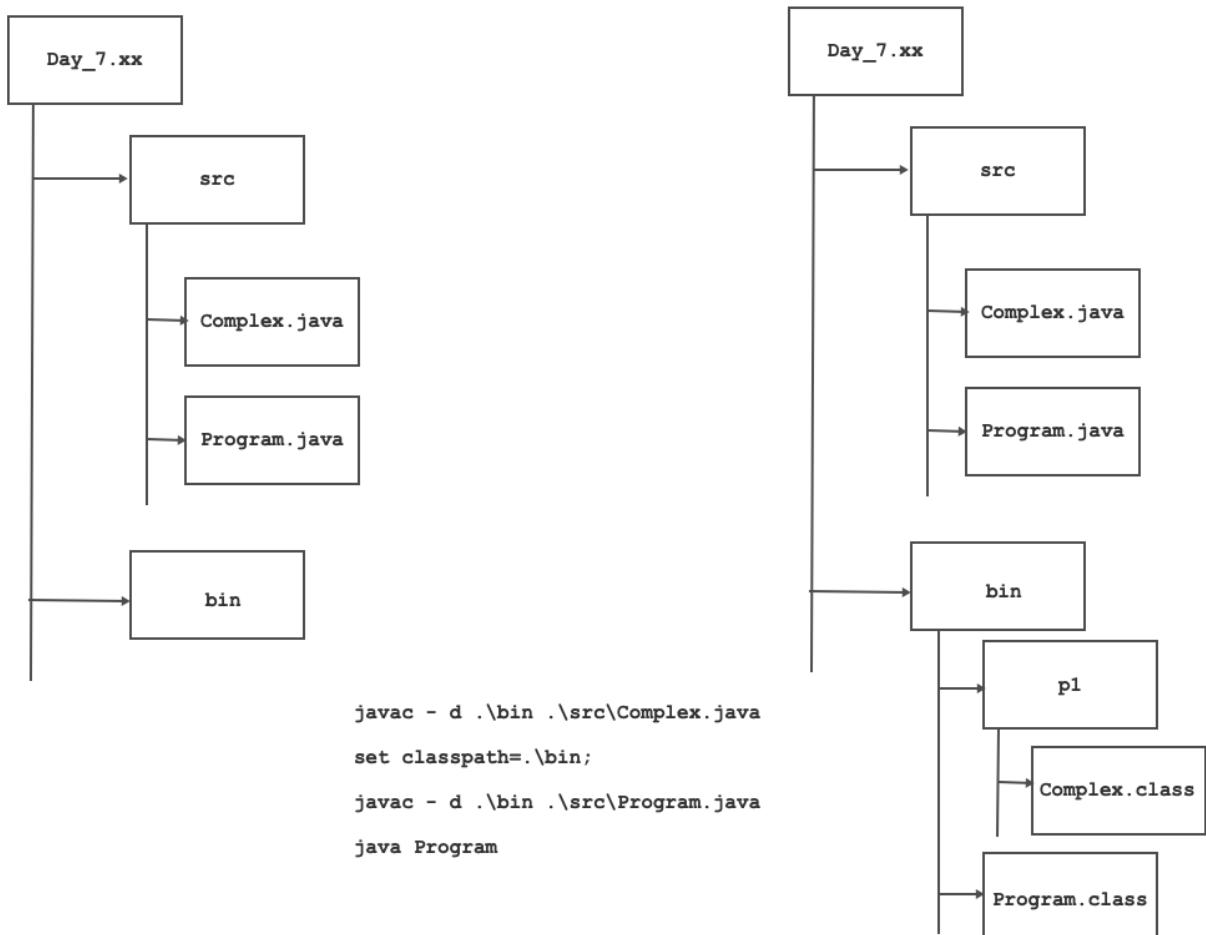
- Compile Program.java

```
set classpath=.\bin;
javac -d .\bin .\src\Program.java
```

- Execute Java application

```
java Program
```

- Conclusion: we can access packaged class inside unpackaged class.



## Demo 2

- Consider `Complex.java`

```

//File Name: Complex.java
public class Complex {
    public String toString() {
        return "Complex.toString( )";
    }
}
    
```

- Compile `Complex.java`

```

javac -d .\bin .\src\Complex.java
    
```

- Consider `Program.java`

```

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

```
        System.out.println( c1.toString());
    }
}
```

- Compile Program.java

```
set classpath=.\bin;
javac -d .\bin .\src\Program.java //error: cannot find symbol Complex
```

- If we define any class without package then it is considered as a member of default package.
- Conclusion: Since we can not import default package, It is impossible to use unpackged class inside packaged class.

### Demo 3

- Consider Complex.java

```
package p1;
public class Complex {
    public String toString() {
        return "Complex.toString( )";
    }
}
```

- Compile Complex.java

```
javac -d .\bin .\src\Complex.java
```

- Consider Program.java

```
package p2;
import p1.Complex;
class Program{
    public static void main(String[] args) {
        Complex c1 = new Complex();
        System.out.println( c1.toString());
    }
}
```

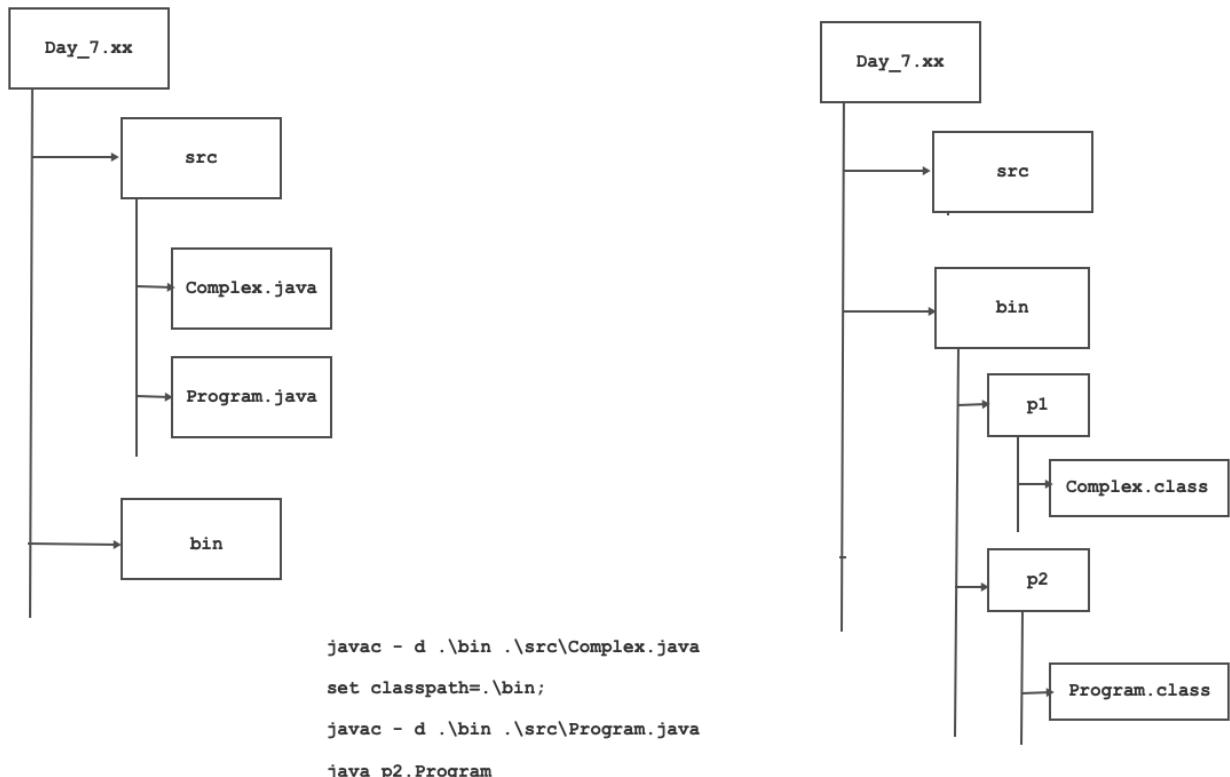
- Compile Program.java

```
set classpath=.\bin;
javac -d .\bin .\src\Program.java
```

- Execute Java application

```
java Program //Error: Could not find or load main class Program
java p2.Program //OK
```

- Conclusion: We can define every class in separate package.



## Demo 4

- Consider Complex.java

```
package p1;
public class Complex {
    public String toString() {
        return "Complex.toString( )";
    }
}
```

- Compile Complex.java

```
javac -d .\bin .\src\Complex.java
```

- Consider Program.java

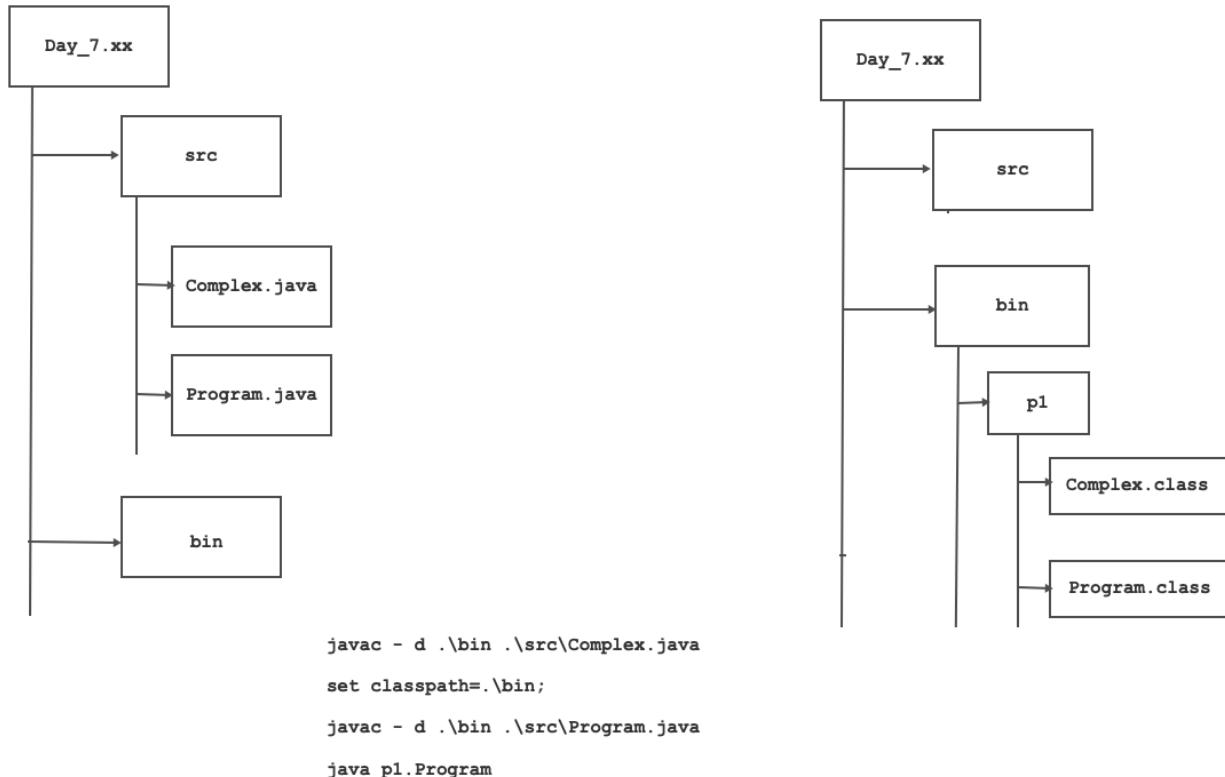
```
package p1;
//import p1.Complex; //Optional
class Program{
    public static void main(String[] args) {
        Complex c1 = new Complex();
        System.out.println( c1.toString());
    }
}
```

- Compile Program.java

```
set classpath=.\bin;
javac -d .\bin .\src\Program.java
```

- Execute Java application

```
java p1.Program
```



## Demo 5

- Consider Complex.java

```
package p1.p2;
public class Complex {
    public String toString() {
        return "Complex.toString( )";
    }
}
```

- Compile Complex.java

```
javac -d .\bin .\src\Complex.java
```

- Consider Program.java

```
package p1.p3;
import p1.p2.Complex;
class Program{
    public static void main(String[] args) {
        Complex c1 = new Complex();
        System.out.println( c1.toString());
    }
}
```

- Compile Program.java

```
set classpath=.\bin;
javac -d .\bin .\src\Program.java
```

- Execute Java application

```
java p1.p3.Program
```

- Conclusion: We can define class inside sub packages.
- java.lang package contains all the fundamental classes of core Java. Java compiler by default import java.lang package in every .java file. Hence to use types declared in java.lang package use of import statement is optional.

## import and static import

- Find area of circle:

```

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

```

- Is it possible to access static members of different class without class name.

```

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

```

- Without class name, if we want to access static members of the class then we should use static import.

```

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

```

- If we want to use any type( interface, class, enum, error, exception, annotation) outside package then we should use import statement.
- If we want to use static members of the class, without class outside class name, then we should use static import.

## naming conventions for package

- org.example.main
- in.cdac.accts.utils
- java.lang.reflect
- oracle.jdbc.driver
- com.mysql.jdbc
- Reference: <https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html>

## Eclipse introduction

# Day 8

Which are non primitive types in Java?

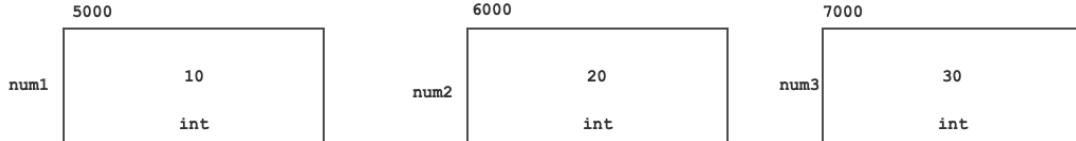
- Interface
- Class
- Enum
- Array

## Array

- How will you store 3 integer values in Java program?
  - First approach

```
int num1 = 10;
int num1 = 20;
int num1 = 30;
```

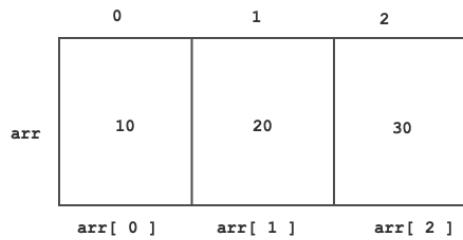
```
int num1 = 10;
int num2 = 20;
int num3 = 30;
```



- Second approach

```
int arr[ 3 ] = { 10, 20, 30 }; //In C/C++
```

```
int arr[ 3 ] = { 10, 20, 30 };
```



- arr is name of the array
- Array size is 3
- 10, 20, 30 are elements.
- Index of 10 is 0
- Index of 20 is 1
- Index of 30 is 2

- Definition
  - Array is a linear / sequential data structure in which, we can store multiple elements of same type in continuous memory location.
  - In Java, array is non primitive/reference type. In other words, to create array instance we must use new operator.
  - If we want to process elements of array then we should use method declared in:
    - java.util.Arrays
    - java.lang.reflect.Array( Reflection )
    - org.apache.commons.lang3.ArrayUtils
      - Download jar from  
"https://mvnrepository.com/artifact/org.apache.commons/commons-lang3/3.12.0"  
location
- Types of array
  - Single dimensional array
  - Multi dimensional array
  - Ragged array
- Types of loop
  - do-while loop
  - while loop
  - for loop
  - foreach loop

## Single dimensional array

- Array reference declaration:

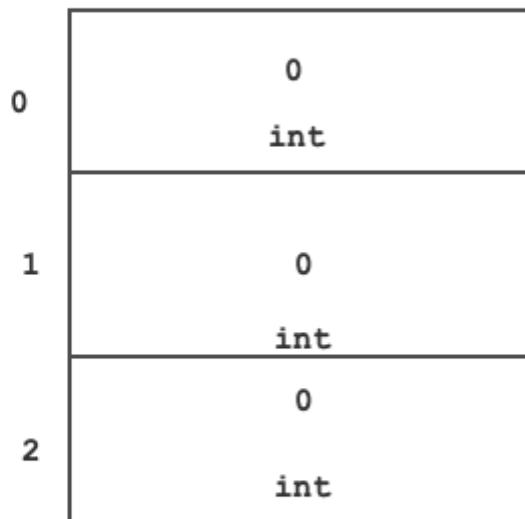
```
int arr[ ]; //OK
int [ arr ]; //Not OK
```

```
int[ ] arr; //OK: Recommended  
int[ ] arr = null; //OK
```

- Instantiation of array

```
new int[ 3 ]; //Array instance with default value: 0  
new boolean[ 3 ]; //Array instance with default value: false
```

### [ Heap ]

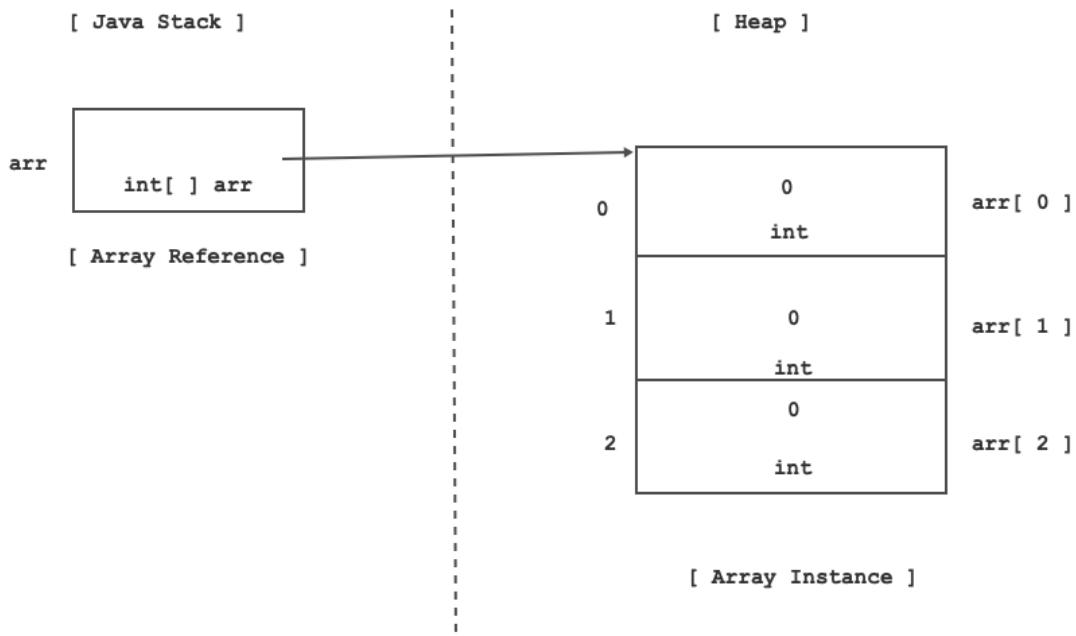


### [ Array Instance ]

- Array instance with reference:

```
int[] arr = null;  
arr = new int[ 3 ];
```

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



```
int[] arr = new int[ -3 ]; //NegativeArraySizeException
```

```
Scanner sc = new Scanner( System.in );
System.out.print("Enter array size : ");
int size = sc.nextInt();
int[] arr = new int[ size ];
```

- Consider example of do-while loop

```
public static void main(String[] args) {
    int[] arr = new int[ 3 ]; //OK
    int index = 0;
    do {
        System.out.println("arr[ "+index+" ] : "+arr[ index ]);
        index = index + 1;
    }while( index < 3 ); //Exit controlled loop
}
```

- Consider example of while loop

```
public static void main(String[] args) {
    int[] arr = new int[ 3 ]; //OK
    int index = 0;
    while( index < 3 ) { //Entry controlled loop
        System.out.println("arr[ "+index+" ] : "+arr[ index ]);
        index = index + 1;
    }
}
```

```
    }  
}
```

- Consider example of for loop

```
public static void main(String[] args) {  
    int[] arr = new int[ 3 ]; //OK  
    for( int index = 0; index < 3; ++ index )  
        System.out.println("arr[ "+index+" ] : "+arr[ index ]);  
}
```

- Consider example of foreach loop

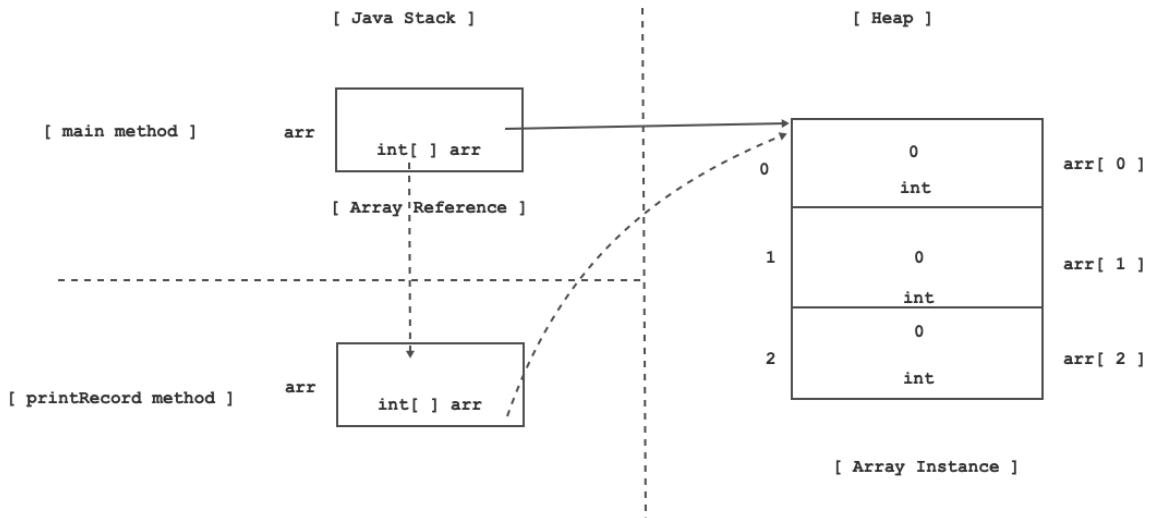
```
public static void main(String[] args) {  
    int[] arr = new int[ 3 ]; //OK  
    //ForEach loop; It is also called as iterator  
    for(int element : arr ) //( Forward only & read only loop)  
        System.out.println(element);  
}
```

- How will you initialize array in Java?

```
int[] arr = new int[ ]{ 10, 20, 30 }; //OK  
  
int[] arr = { 10, 20, 30 }; //OK  
  
int[] arr = new int[ 3 ]{ 10, 20, 30 }; //NOT OK
```

- length is a property of array data structure which returns size/length of the array.

```
private static void printRecord(int[] arr) {  
    for( int index = 0; index < arr.length; ++ index )  
        System.out.print( arr[ index ]+" ");  
    System.out.println();  
}  
public static void main(String[] args) {  
    int[] arr1 = new int[ ]{ 10, 20, 30, 40, 50 }; //OK  
    Program.printRecord( arr1 );  
  
    int[] arr2 = new int[ ]{ 10, 20, 30 }; //OK  
    Program.printRecord( arr2 );  
  
    int[] arr3 = new int[ ]{ 10, 20, 30, 40, 50, 60, 70 }; //OK  
    Program.printRecord( arr3 );  
}
```



- How will accept and print record of array

```

public class Program {
    static Scanner sc = new Scanner(System.in);
    private static void acceptRecord(int[] arr) {
        if( arr != null ) {
            for( int index = 0; index < arr.length; ++ index ) {
                System.out.print("Enter element :   ");
                arr[ index ] = sc.nextInt();
            }
        }
    }
    private static void printRecord(int[] arr) {
        if( arr != null ) {
            for( int index = 0; index < arr.length; ++ index )
                System.out.print( arr[ index ]+"   ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        //int[] arr = null;
        int[] arr = new int[ 3 ];

        Program.acceptRecord( arr );

        Program.printRecord(arr);
    }
}

```

- How to use for each loop on array

```

public class Program {
    private static void printRecord(int[] arr) {

```

```

    if (arr != null) {
        for( int element : arr )
            System.out.print(element+" ");
    }
    System.out.println();
}

public static void main(String[] args) {
    int[] arr = new int[] { 10, 20, 30 };
    Program.printRecord(arr);
}
}

```

- Consider following code:

```

public class Program {
    public static void main(String[] args) {
        boolean[] arr1 = new boolean[3];
        System.out.println( arr1.toString()); // [Z@6d06d69c

        byte[] arr2 = new byte[3];
        System.out.println( arr2.toString()); // [B@7852e922

        char[] arr3 = new char[3];
        System.out.println( arr3.toString()); // [C@4e25154f

        short[] arr4 = new short[3];
        System.out.println( arr4.toString()); // [S@70dea4e

        int[] arr5 = new int[3];
        System.out.println( arr5.toString()); // [I@5c647e05

        float[] arr6 = new float[3];
        System.out.println( arr6.toString()); // [F@33909752

        double[] arr7 = new double[3];
        System.out.println( arr7.toString()); // [D@55f96302

        long[] arr8 = new long[3];
        System.out.println( arr8.toString()); // [J@3d4eac69

        String[] arr9 = new String[3];
        System.out.println( arr9.toString());
// [Ljava.lang.String;@42a57993

        int[][] arr10 = new int[3][3];
        System.out.println(arr10.toString()); // [[I@75b84c92

        int[][][] arr11 = new int[3][3][3];
        System.out.println(arr11.toString()); // [[[I@6bc7c054
    }
}

```

- Reference: Please check documentation of java.lang.Class class: public String getName()
- If we want to get elements of array using `toString()` method then we should use `toString` method of `java.util.Arrays` class.

```
public static void main(String[] args) {
    int[] arr = new int[ ] { 10, 20, 30 };
    System.out.println( Arrays.toString(arr) );
}
```

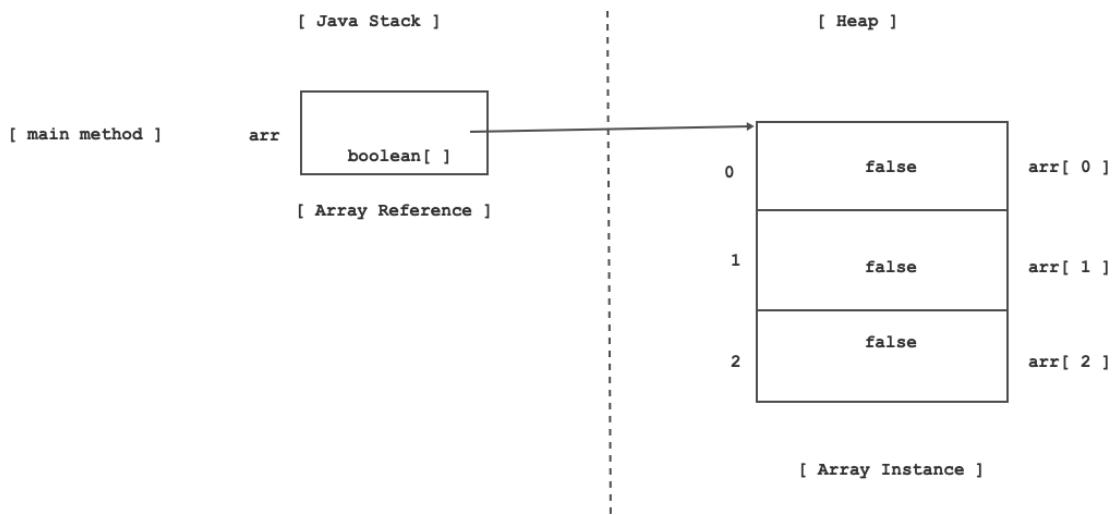
- Using illegal index, if we try to access elements of array then JVM throws `ArrayIndexOutOfBoundsException`.

```
public class Program {
    public static void main(String[] args) {
        int[] arr = new int[ ] { 10, 20, 30 };
        int element = arr[ arr.length ];
        //ArrayIndexOutOfBoundsException
        System.out.println( element );
    }
    public static void main2(String[] args) {
        int[] arr = new int[ ] { 10, 20, 30 };
        int element = arr[ 3 ]; //ArrayIndexOutOfBoundsException
        System.out.println( element );
    }
    public static void main1(String[] args) {
        int[] arr = new int[ ] { 10, 20, 30 };
        int element = arr[ -1 ]; //ArrayIndexOutOfBoundsException
        System.out.println( element );
    }
}
```

## How will you create array of primitive values

- Consider following

```
boolean[] arr = new boolean[ ]{ true, false, true };
int[] arr = { 10, 20, 30 };
double[] arr = new double[ 3 ];
arr[ 0 ] = 1.1;
arr[ 1 ] = 1.2;
arr[ 2 ] = 1.3;
```



```
boolean[] arr = new boolean[ 3 ];
```

- If we create array of primitive type then it contains values. If we do not specify value then default value is depends of default value of data type.

### How will you create array of references

- If we create of references then array by default contains null value.
- Consider following code:

```
package org.example.main;

import java.time.LocalDate;
import java.util.Arrays;
class Date{
    private int day;
    private int month;
    private int year;

    public Date() {
        LocalDate ld = LocalDate.now();
        this.day = ld.getDayOfMonth();
        this.month = ld.getMonthValue();
        this.year = ld.getYear();
    }

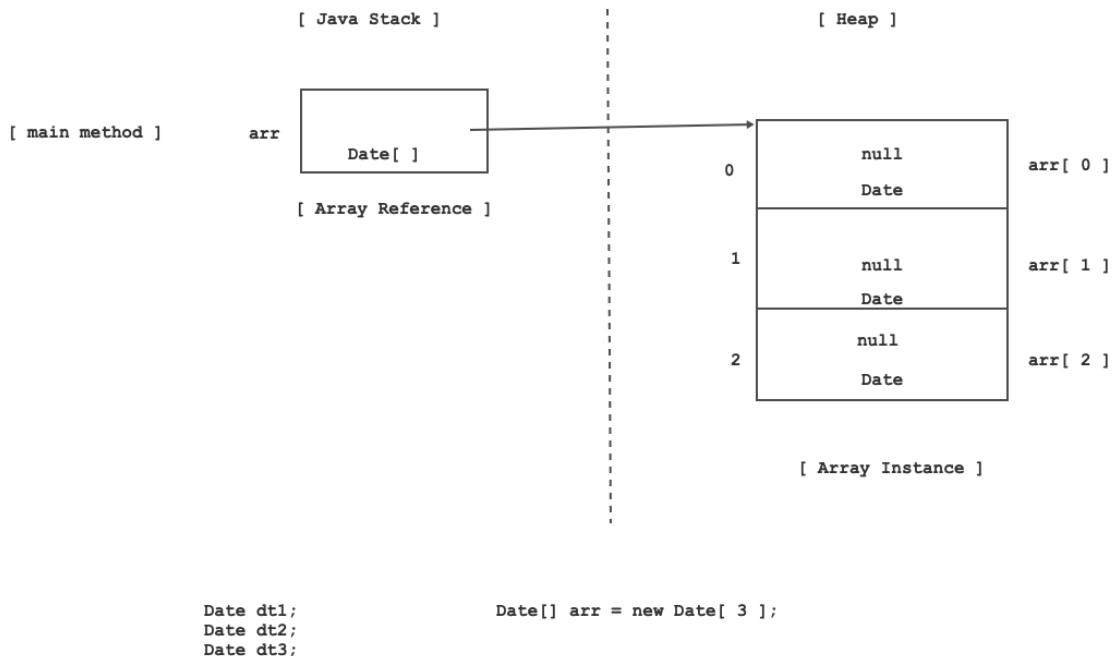
    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    @Override
    public String toString() {
        return this.day+" / "+this.month+" / "+this.year;
    }
}
```

```

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

```



## How will you create array of instances

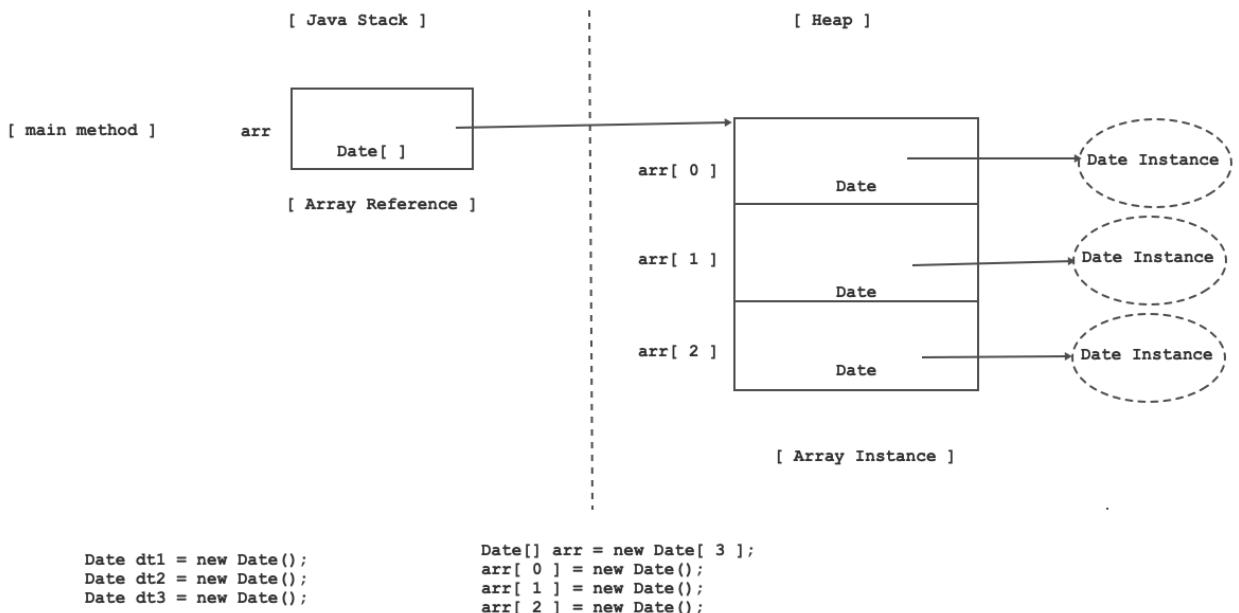
- Consider following code

```

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

    for( int index = 0; index < arr.length; ++ index )
        System.out.println( arr[ index ].toString()); //OK
}

```



How can we pass argument to the function in C++?

- We can pass argument to the function by value.

```
//int a = x;
//int b = y;
void swap( int a, int b ){
    int temp = a;
    a = b;
    b = temp;
}
int main( void ){
    int x = 10;
    int y = 20;

    swap( x, y ); //Function call by value

    cout<<"X : "<< x << endl; //10
    cout<<"Y : "<< y << endl; //20
    return 0;
}
```

- We can pass argument to the function by address.

```
//int *a = &x;
//int *b = &y;
void swap( int *a, int *b ){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```

int main( void ){
    int x = 10;
    int y = 20;

    swap( &x, &y ); //Function call by address

    cout<<"X : "<< x << endl; //20
    cout<<"Y : "<< x << endl; //10
    return 0;
}

```

- We can pass argument to the function by reference.

```

```java
//int &a = x;
//int &b = y;
void swap( int &a, int &b ){
    int temp = a;
    a = b;
    b = temp;
}
int main( void ){
    int x = 10;
    int y = 20;

    swap( x, y ); //Function call by reference

    cout<<"X : "<< x << endl; //20
    cout<<"Y : "<< x << endl; //10
    return 0;
}

```

- In Java, we can pass any argument( primitive / non primitive ) to the method by value only.
- Consider code:

```

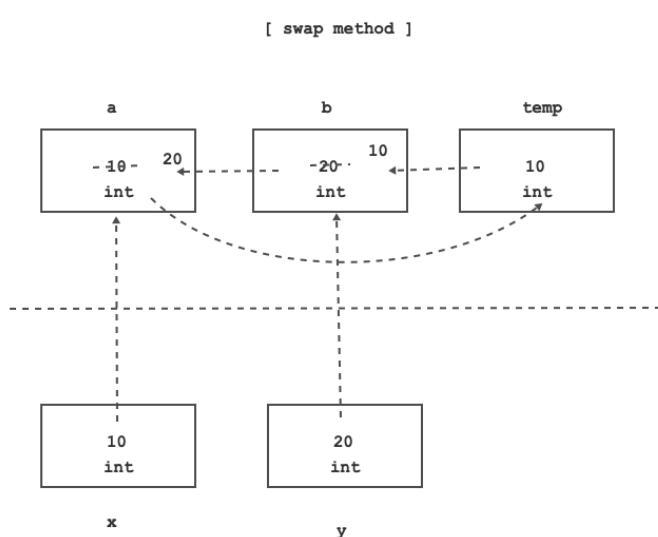
package org.example.main;
public class Program {

    private static void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
    }
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        Program.swap( x, y );
        System.out.println("X : "+x); //10
        System.out.println("Y : "+y); //20
    }
}

```

```
}
```

Method call by value



```
package org.example.main;
public class Program {

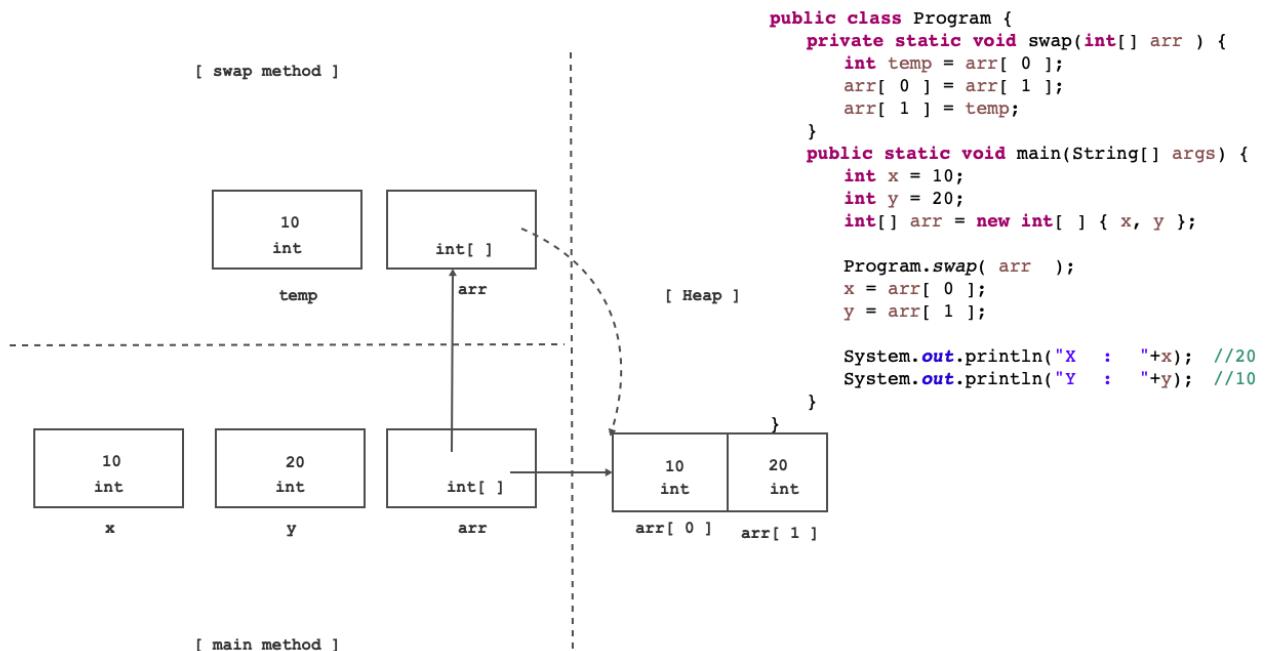
    private static void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
    }
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        Program.swap( x, y );
        System.out.println("X : "+x); //10
        System.out.println("Y : "+y); //20
    }
}
```

- If we want to simulate pass by reference mechanism then we should use array in Java.
- Consider following code:

```
public class Program {
    private static void swap(int[] arr ) {
        int temp = arr[ 0 ];
        arr[ 0 ] = arr[ 1 ];
        arr[ 1 ] = temp;
    }
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int[] arr = new int[ ] { x, y };

        Program.swap( arr );
        x = arr[ 0 ];
        y = arr[ 1 ];

        System.out.println("X : "+x); //20
        System.out.println("Y : "+y); //10
    }
}
```



- How to accept record for single variable inside method?

```

import java.util.Scanner;
public class Program {
    private static Scanner sc = new Scanner(System.in);
    public static void acceptRecord( int[] number ) {
        System.out.print("Enter number : ");
        number[0] = sc.nextInt();
    }
    private static void printRecord(int[] number) {
        System.out.println("Number : "+number[0]);
    }
    public static void main(String[] args) {
        int[] number = new int[1];
        Program.acceptRecord(number);
        Program.printRecord(number);
    }
}

```

## Multi dimensional array

- Array of array where, size of every array is same/constant.
- Reference declaration:

```

public static void main(String[] args) {
    int arr1[][] = null; //OK

    int[] arr2[] = null; //OK
}

```

```

        int[ ][ ] arr3 = null; //OK
    }
}

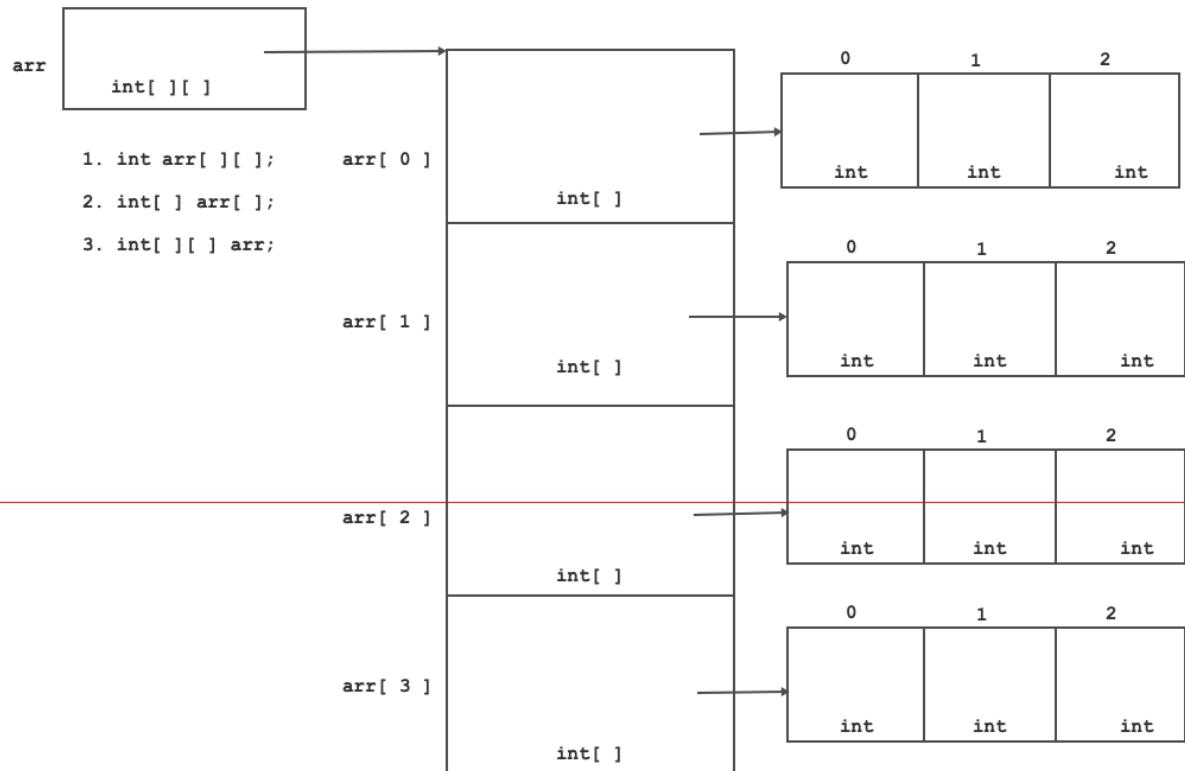
```

- Instantiation:

```

public static void main(String[] args) {
    int[ ][ ] arr = new int[ 4 ][ 3 ];
}

```



- How to accept and print record

```

import java.util.Scanner;

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

    private static void acceptRecord(int[][] arr) {
        for (int row = 0; row < arr.length; ++row) {
            for (int col = 0; col < arr[row].length; ++col) {
                System.out.print("Enter number : ");
                arr[row][col] = sc.nextInt();
            }
        }
    }

    private static void printRecord(int[][] arr) {
        for (int row = 0; row < arr.length; ++row) {

```

```

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

public static void main(String[] args) {
    int[][] arr = new int[4][3];
    Program.acceptRecord(arr);
    Program.printRecord(arr);
}

```

- How will you initialize multidimensional array

```

public static void main(String[] args) {
    int[][] arr1 = new int[][] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9
}, { 10, 11, 12 } }; //OK

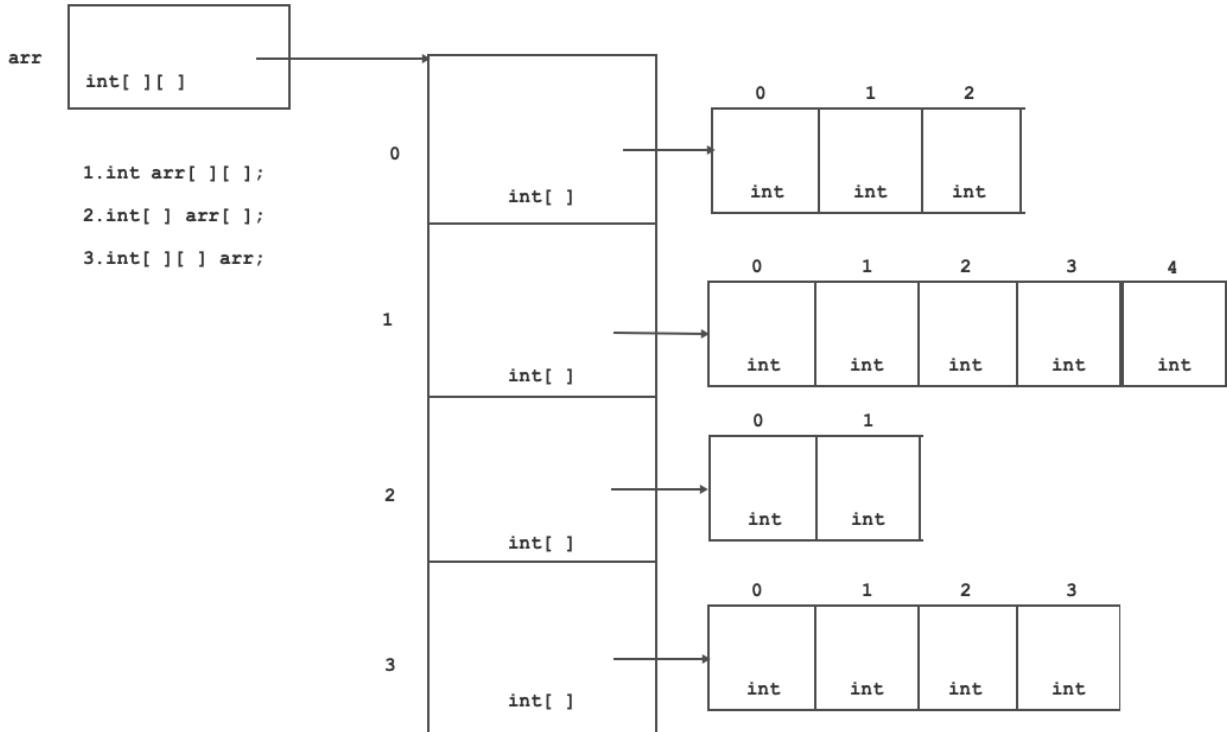
    int[][] arr2 = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11,
12 } }; //OK
}

```

# Day 9

## Ragged Array

- Array of array where column size of every array is different.



- Reference declaration:

```
public static void main(String[] args) {
    int arr1[ ][ ] = null;

    int[ ] arr2[ ] = null;

    int[ ][ ] arr3 = null;
}
```

- Instantiation

```
public static void main(String[] args) {
    int[ ][ ] arr = new int[ 4 ][ ];
    arr[ 0 ] = new int[ 3 ];
    arr[ 1 ] = new int[ 5 ];
    arr[ 2 ] = new int[ 2 ];
    arr[ 3 ] = new int[ 4 ];
}
```

- Initialization

```
public static void main(String[] args) {
    int[][] arr = new int[4][];
    arr[0] = new int[] { 1, 2, 3 };
    arr[1] = new int[] { 4, 5, 6, 7, 8 };
    arr[2] = new int[] { 9, 10 };
    arr[3] = new int[] { 11, 12, 13, 14 };
}
```

- How to use `toString()` method?

```
public static void main(String[] args) {
    int[][] arr = new int[4][];
    arr[0] = new int[] { 1, 2, 3 };
    arr[1] = new int[] { 4, 5, 6, 7, 8 };
    arr[2] = new int[] { 9, 10 };
    arr[3] = new int[] { 11, 12, 13, 14 };

    //System.out.println( Arrays.deepToString(arr));    //[[1, 2, 3],
    [4, 5, 6, 7, 8], [9, 10], [11, 12, 13, 14]]

    //System.out.println( arr.toString());   //[[I@6d06d69c

    /* System.out.println( Arrays.toString( arr[ 0 ] ) );
    System.out.println( Arrays.toString( arr[ 1 ] ) );
    System.out.println( Arrays.toString( arr[ 2 ] ) );
    System.out.println( Arrays.toString( arr[ 3 ] ) ); */

    for( int index = 0; index < 4; ++ index )
        System.out.println( Arrays.toString( arr[ index ] ) );
}
```

- How to use `foreach` loop

```
public static void main(String[] args) {
    int[][] arr = new int[4][];
    arr[0] = new int[] { 1, 2, 3 };
    arr[1] = new int[] { 4, 5, 6, 7, 8 };
    arr[2] = new int[] { 9, 10 };
    arr[3] = new int[] { 11, 12, 13, 14 };

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

- How to accept and print record?

```

import java.util.Scanner;

public class Program {
    private static Scanner sc = new Scanner(System.in);
    private static int[][] getArray() {
        int[][] arr = new int[4][];
        arr[0] = new int[3];
        arr[1] = new int[5];
        arr[2] = new int[2];
        arr[3] = new int[4];
        return arr;
    }
    private static void acceptRecord(int[][] arr) {
        if( arr != null ) {
            for( int row = 0; row < arr.length; ++ row ) {
                for( int col = 0; col < arr[ row ].length; ++ col ) {
                    System.out.print("arr[ "+row+" ][ "+col+" ] : ");
                    arr[ row ][ col ] = sc.nextInt();
                }
            }
        }
    }
    private static void printRecord(int[][] arr) {
        if( arr != null ) {
            for( int row = 0; row < arr.length; ++ row ) {
                for( int col = 0; col < arr[ row ].length; ++ col ) {
                    System.out.print(arr[ row ][ col ]+ " ");
                }
                System.out.println();
            }
        }
    }
    public static void main(String[] args) {
        int[][] arr = Program.getArray();
        Program.acceptRecord( arr );
        Program.printRecord( arr );
    }
}

```

- How will you write hello world w/o giving semicolon

```

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

```

## Variable argument / arity method

- Consider examples of variable argument method from Java API:
  - public PrintStream printf(String format, Object... args) -- java.io.PrintStream
  - public static String format(String format, Object... args) -- java.lang.String
  - public Object invoke(Object obj, Object... args) -- java.lang.reflect.Method

```
public class Program {  
    private 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 );  
  
        Program.sum( 10, 20, 30, 40, 50 );  
  
        Program.sum( 10, 20, 30, 40, 50, 60, 70 );  
  
        Program.sum( 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 );  
    }  
}
```

## Enum

- If we want to improve readability of the source code then we should use enum in C/C++/java.
- Consider code in C programming language:

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

```

    }

    void print_record( int result ){
        printf("Result : %d\n",result);
    }

    enum ArithmeticOperation{
        EXIT, SUM, SUB,MULTIPLICATION, DIVISION // Enum Constants
        //EXIT = 0, SUM = 1, SUB = 2,MULTIPLICATION = 3, DIVISION = 4 //
    Enum Constants
    };

    enum ArithmeticOperation menu_list( ){
        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;
            }
            print_record( result );
        }
        return 0;
    }
}

```

- enum is keyword in java
- enum is non primitive type in Java.
- Consider example of enum in Java:

```

enum Color{
    RED, GREEN, BLUE
    //RED = 0, GREEN = 1, BLUE = 2
}
//RED, GREEN, BLUE : Name of enum constants
//0,1,2 : Ordinals

```

- We can not change ordinal value of enum constant.
- How to give name to the literals in C programming language

```

enum Day{
    MON = 1, TUES = 2, WED = 3, THURS = 4, FRI = 5, SAT = 6, SUN = 7
}

```

- How to give name to the literals in Java programming language

```

enum Day{
    MON( 1 ), TUES( 2 ), WED( 3 ), THURS ( 4 ), FRI( 5 ), SAT( 6 ),
    SUN( 7 );
    // 1,2,3,4,5,6,7 : Literals
    //MON, TUES, WED, THURS, FRI, SAT, SUN : Name of enum constants
    //0,1,2,3,4,5,6 : Ordinals
}

```

- Consider another example of enum in Java:

```

enum Color{
    RED( 255,0,0 ), GREEN( 0,255,0 ), BLUE( 0,0,255 )
    //RED = 0, GREEN = 1, BLUE = 2
}
//(255,0,0),(0,255,0),(0,0,255)
//RED, GREEN, BLUE : Name of enum constants
//0,1,2 : Ordinals

```

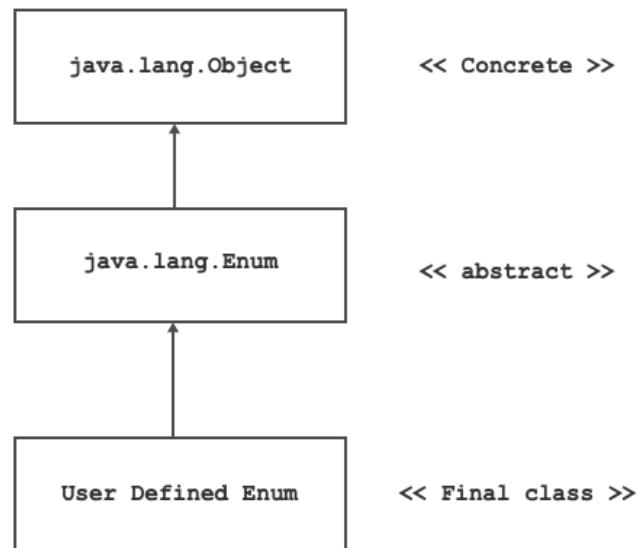
- Consider another example of enum in Java:

```

enum Day{
    MON( 1, "Monday" ), SUN( 7, "SunDay" );
}
//( 1, "Monday" ), ( 7, "SunDay" ); //Literals
//MON, SUN : Name of enum constants
//0, 1 : Ordinal values

```

- Using enum, we can give name to the single literal as well group of literals of same type / different type.
- java.lang.Enum is considered as super class of all the enums in Core Java language.



- Methods of `java.lang.Object` class

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

- Method of `java.lang.Enum` class

- `public final Class getDeclaringClass()`
- `public final String name()`
- `public final int ordinal()`
- `public static <T extends Enum> T valueOf(Class enumType, String name)`

- Methods of user defined enum

```
enum Color{
```

```
}
```

- public static Color[] values(); - It returns array of enum constant.
- public static Color valueOf(String name); - It converts string enum constant into Enum

- Consider enum code in Java

```
enum Color{ //Color.class
    RED, GREEN, BLUE
}
```

- Let us observe compiler generated code

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

    public static Color[] values();    //Color[] colors = Color.values()
    public static Color valueOf(String); //Color color =
    Color.valueOf("GREEN");
}
```

- Since enum is implicitly considered as final class, we can not extend enum. In other words we can not create sub type of enum.
- Consider code to demonstrate name()/ordinal() and values() method.

```
enum Color{
    RED, GREEN, BLUE
}
public class Program {
    public static void main(String[] args) {
        Color[] colors = Color.values();
        for( Color color : colors )
            System.out.println( color.name()+" "+color.ordinal());
    }
    public static void main4(String[] args) {
        Color color = Color.RED;
        System.out.println( color.name()+" "+color.ordinal());

        color = Color.GREEN;
        System.out.println( color.name()+" "+color.ordinal());

        color = Color.BLUE;
        System.out.println( color.name()+" "+color.ordinal());
    }
}
```

```

public static void main3(String[] args) {
    System.out.println( Color.BLUE.name());           //BLUE
    System.out.println( Color.BLUE.ordinal());        //2
}
public static void main2(String[] args) {
    System.out.println( Color.GREEN.name());          //GREEN
    System.out.println( Color.GREEN.ordinal());        //1
}
public static void main1(String[] args) {
    System.out.println( Color.RED.name());            //RED
    System.out.println( Color.RED.ordinal());          //0
}
}

```

- To assign name to the literals, it is mandatory to define constructor inside enum.
- Access modifier of enum constructor can be either package level private or private only.

```

package org.example.main;
enum Day {
    MON("Monday"), TUES("TuesDay"), WED("WednesDay");

    private String dayName;

    private Day(String dayName) {
        this.dayName = dayName;
    }
    public String getDayName() {
        return dayName;
    }
}

public class Program {
    public static void main(String[] args) {
        Day day = Day.MON;
        System.out.println(day.name());      //MON
        System.out.println(day.ordinal());   //0
        System.out.println(day.getDayName()); //Monday
    }
}

```

```

enum Day {
    MON(1), TUES(2), WED(3);

    private int dayNumber;

    private Day(int dayNumber) {
        this.dayNumber = dayNumber;
    }
}

```

```

        public int getDayNumber() {
            return dayNumber;
        }
    }

    public class Program {
        public static void main(String[] args) {
            Day day = Day.MON;
            System.out.println(day.name()); //MON
            System.out.println(day.ordinal()); //0
            System.out.println(day.getDayNumber()); //1
        }
    }
}

```

```

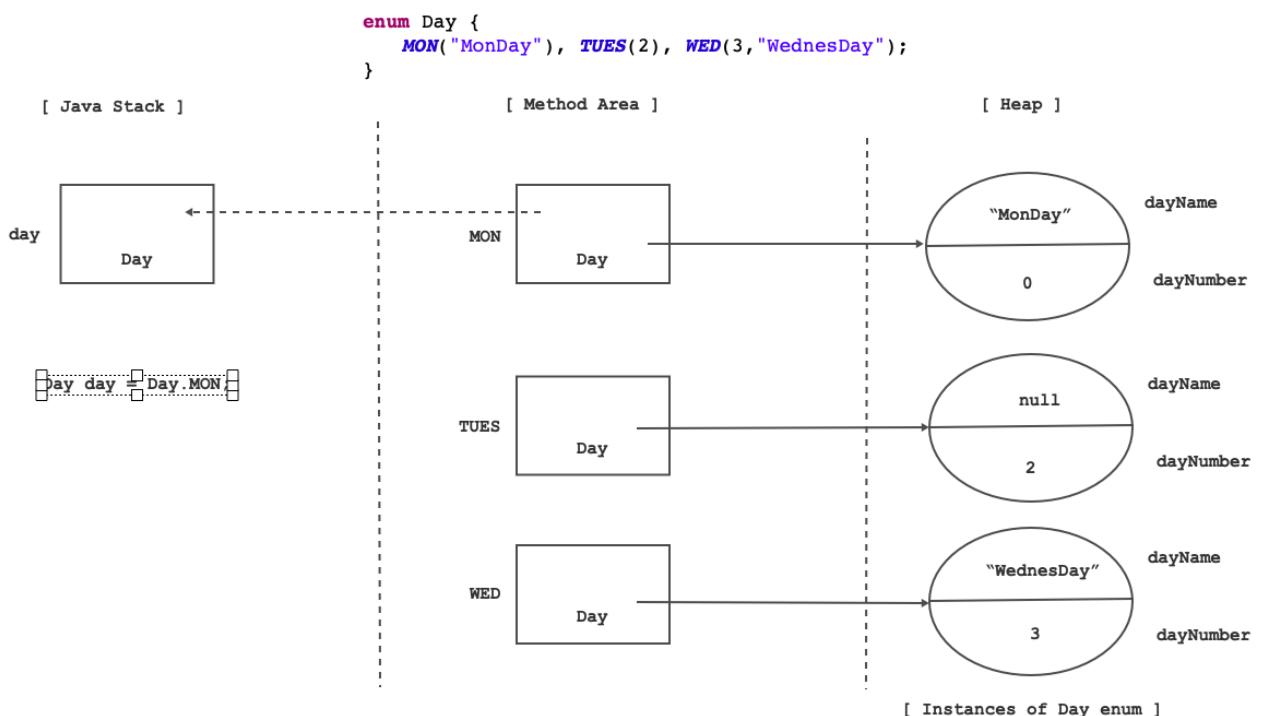
enum Day {
    MON(1, "MonDay"), TUES(2, "TuesDay"), WED(3, "WednesDay");

    private int dayNumber;
    private String dayName;
    private Day(int dayNumber, String dayName) {
        this.dayNumber = dayNumber;
        this.dayName = dayName;
    }
    public int getDayNumber() {
        return dayNumber;
    }
    public String getDayName() {
        return dayName;
    }
    public String toString() {
        return String.format("%-10s%-5d%-10s%-5d", this.name(),
this.ordinal(), this.dayName, this.dayNumber);
    }
}

public class Program {
    public static void main(String[] args) {
        Day[] days = Day.values();
        for( Day day : days )
            System.out.println( day.toString() );
    }
    public static void main1(String[] args) {
        Day day = Day.MON;
        System.out.println(day.name()); //MON
        System.out.println(day.ordinal()); //0

        System.out.println(day.getDayNumber()); //1
        System.out.println(day.getDayName()); //MonDay
    }
}

```



- Menu driven code using enum

```

package org.example.main;

import java.util.Arrays;
import java.util.Scanner;

enum ArithmeticOperation{
    EXIT, SUM, SUB, MULTIPLICATION, DIVISION
}
class ArithmeticOperationTest{
    public static int sum( int num1, int num2 ) {
        return num1 + num2;
    }
    public static int sub( int num1, int num2 ) {
        return num1 - num2;
    }
    public static int multiplication( int num1, int num2 ) {
        return num1 * num2;
    }
    public static int division( int num1, int num2 ) {
        return num1 / num2;
    }
    public static void printRecord( int result ) {
        System.out.println("Result : "+result);
    }
    private static Scanner sc = new Scanner(System.in);
    public static ArithmeticOperation menuList( ) {
        System.out.println("0.Exit");
        System.out.println("1.Sum");
        System.out.println("2.Sub");
    }
}

```

```

        System.out.println("3.Multiplication");
        System.out.println("4.Division");
        System.out.print("Enter choice : ");
        int choice = sc.nextInt();
        return ArithmeticOperation.values()[ choice ];

    /* ArithmeticOperation[] arr = ArithmeticOperation.values();
    // [EXIT, SUM, SUB, MULTIPLICATION, DIVISION]
    return arr[ choice ]; */

}

public class Program {
    public static void main(String[] args) {
        ArithmeticOperation choice;
        while( ( choice = ArithmeticOperationTest.menuList( ) ) != ArithmeticOperation.EXIT ) {
            int result = 0;
            switch( choice ) {
                case SUM:
                    result = ArithmeticOperationTest.sum(100, 20);
                    break;
                case SUB:
                    result = ArithmeticOperationTest.sub(100, 20);
                    break;
                case MULTIPLICATION:
                    result = ArithmeticOperationTest.multiplication(100, 20);
                    break;
                case DIVISION:
                    result = ArithmeticOperationTest.division(100, 20);
                    break;
            }
            ArithmeticOperationTest.printRecord(result);
        }
    }
}

```

## OOPS

- Object oriented programming structure/system
- OOPS is not a syntax. It is a programming methodology which is used to solve real world problems using classes and objects.
- Dr Alan Kay is inventor of OOPS.
- Grady Booch is inventor of UML.
  - Author of "Object Oriented Analysis and Design with Applications"
- According to Grady Booch there should be some major and minor features in the oops.

## 4 Major Elements / Features / Parts /Pillars

- By major, we mean that a language without any one of these elements is not object oriented.
  - Abstraction - To achieve simplicity

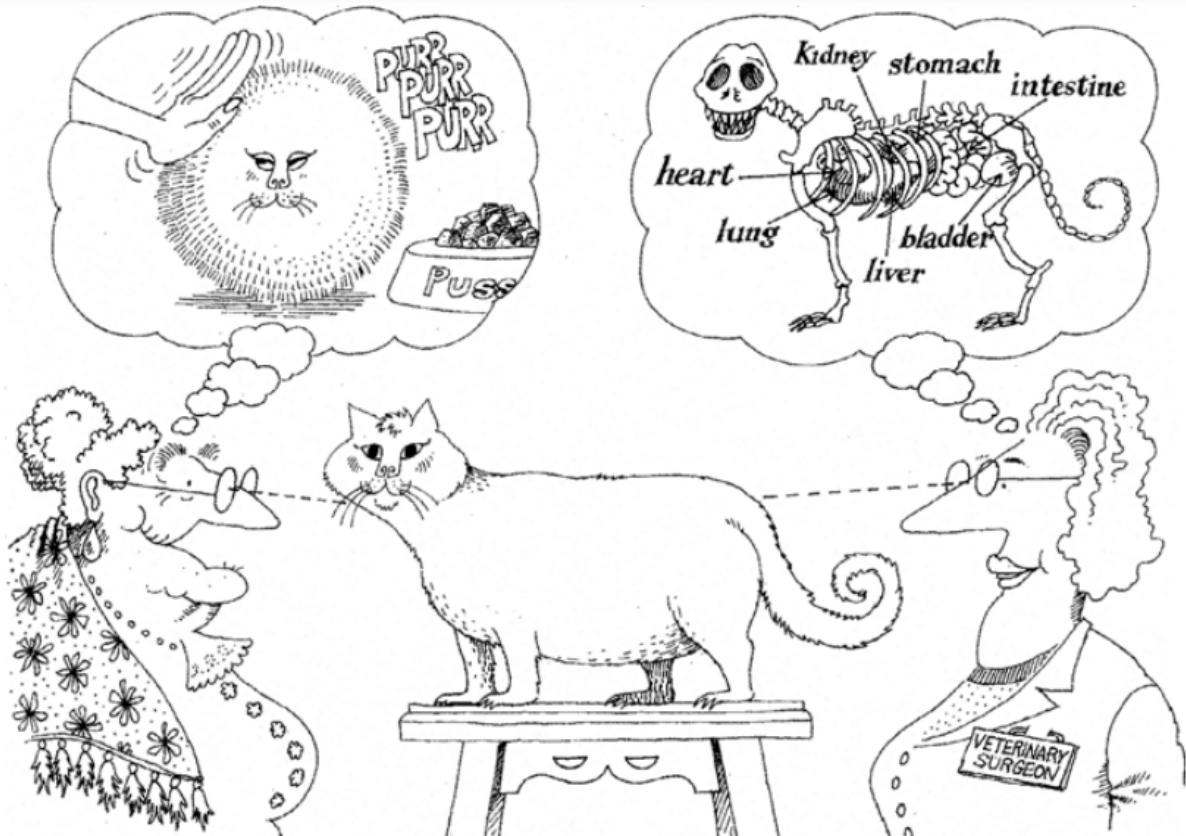
- Encapsulation - To hide data and to achieve security
- Modularity - To reduce module dependency
- Hierarchy - To achieve reusability
  - Types of hierarchy
    - Has-a - represents Association( Specialized form => Composition/Aggregation)
    - Is-a - represents generalization( also called as inheritance )
    - Use-a - represents dependency
    - Creates-a - represents instantiation

### 3 Minor Elements / Features / Parts /Pillars

- By minor, we mean that each of these elements is a useful, but not essential.
  - Typing / Polymorphism - To reduce maintenance of the system
  - Concurrency - To utilize h/w resources( Memory/CPU ) efficiently
  - Persistance - To maintain state of the instance of secondary storage.

### Abstraction

- Abstraction is a major element / pillar of oops.
- Process of getting essentials things from object-instance is called as abstraction.
- Abstraction focuses on outer behavior of an instance.



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

- Abstraction in java:

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

```

        Complex c1 = new Complex( );
        c1.acceptRecord();
        c1.PrintRecord();
    }
}

```

```

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

```

```

interface Printtable{
    void print();
}

```

- Creating instance and calling/invoking methods on it represents abstraction in Java.

## Encapsulation

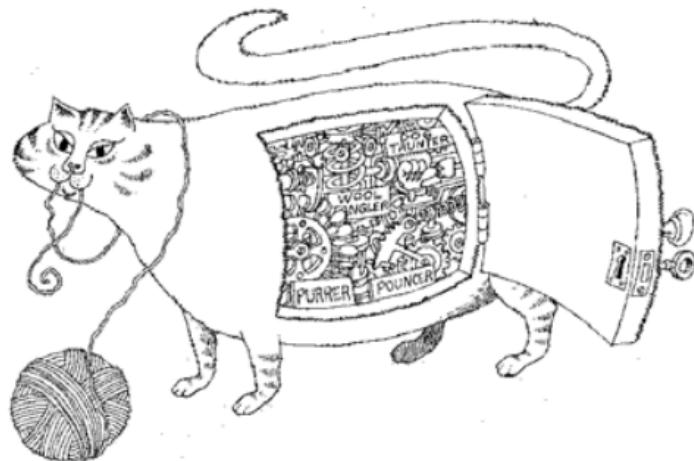
- Definition
  - Implementation of abstraction is called as encapsulation.
  - Binding of data and code together is called as encapsulation

```

class Complex{
    private int real;
    private int imag;
    public Complex( ){
        this.real = 0;
        this.imag = 0;
    }
    public void acceptRecord( ){
        Scanner sc = new Scanner( System.in );
        System.out.print("Real Number : ");
        this.real = sc.nextInt();
        System.out.print("Imag Number : ");
        this.imag = sc.nextInt();
    }
    public void printRecord( ){
        System.out.println("Real Number : "+this.real);
        System.out.println("Imag Number : "+this.imag);
    }
}

```

- Class definition represents encapsulation.



**Encapsulation hides the details of the implementation of an object.**

- 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.

# Day 10

Major and minor elements / pillars /parts of oops

## Abstraction

- It is a major pillar of oops.
- Process of getting essential things from instance / system is called as abstraction.
- Abstraction focuses on external behavior of the instance.
- Abstraction changes from entity to entity.
- Abstraction helps to achieve simplicity.
- In Java, if we want to achieve abstraction then we should create instance and invoke method on it.
- For Example:

```
class Program{
    public static void main(String[] args) {
        Complex c1 = new Complex();
        c1.acceptRecord();
        c1.printRecord();
    }
}
```

## Encapsulation

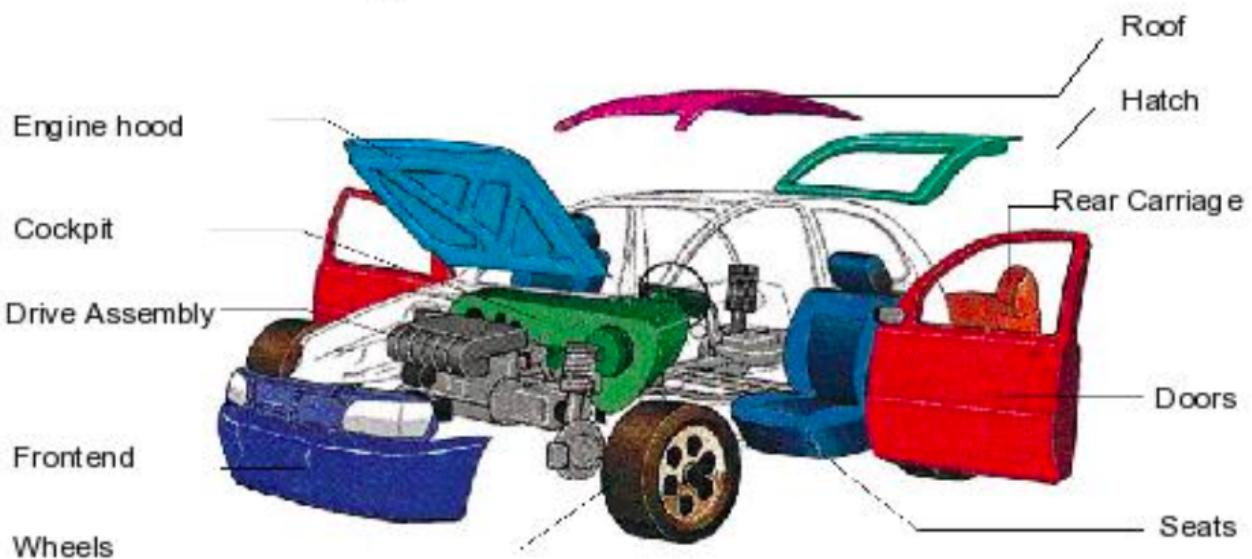
- Definition:
  - Binding of data and code together is called as encapsulation.
  - Implementation of abstraction is called as encapsulation.
- Abstraction and encapsulation are complementary concepts. In other words abstraction focuses on external behavior whereas encapsulation focuses internal behavior.
- Using encapsulation:
  - We can achieve abstraction
  - We can hide data from user:
    - Process of declaring fields private is called as data hiding. Data hiding is also called as data encapsulation.
    - Data hiding helps us to achieve data security.
- To achieve encapsulation we should define class. Hence class is considered as a basic unit of encapsulation.
- For Example:

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

```
public void acceptRecord( ){
    //TODO
}
public void printRecord( ){
    //TODO
}
```

## Modularity

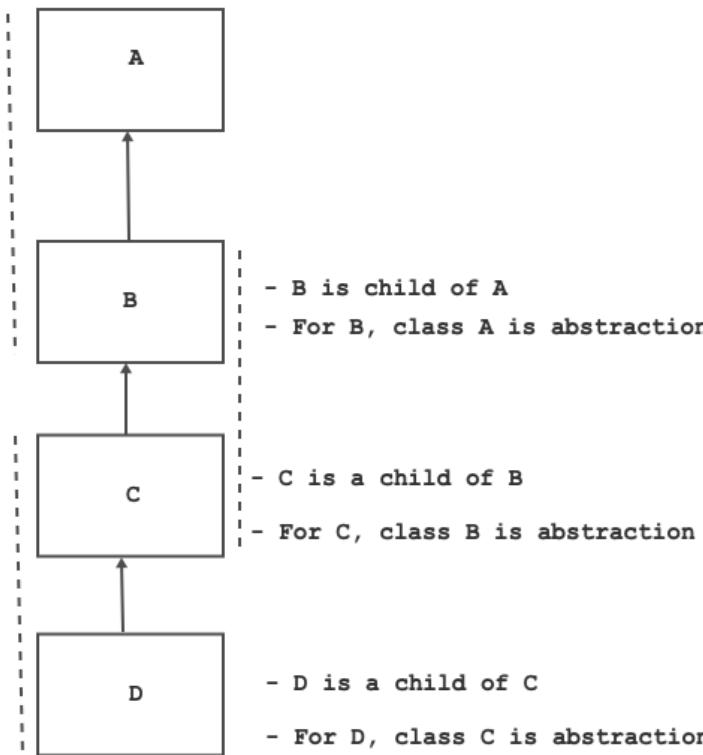
- It is a major pillar of oops.
- It is the process of developing complex system using small parts.
- Modularity helps us to minimize module dependency.
- In Java, we can achieve Modularity using packages and .jar file.



## Hierarchy

- It is a major pillar of oops.

- Level / order / ranking of abstraction is called as hierarchy.



- Using hierarchy, we can achieve code reusability.
  - Development time will be reduce
  - Development cost will be reduce
  - Developers effoort will reduce
- Types of hierarchy
  - Has-a hierarchy
    - Also called as part-of hierarchy.
    - It represents Association.
    - There are 2 specialized form of association:
      - Compositon
      - Aggregation
  - Is-a hierarchy
    - It is also called as Kind-of hierarchy.
    - It represents generalization.
    - Generalization is also called as inheritance.
    - Types of Inheritance:
      - Implementation inheritance
        - Single level inheritance : ( Allowed in Java )
        - Multiple inheritance : ( Not Allowed in Java )
        - Hierarchical level inheritance : ( Allowed in Java )
        - Multilevel inheritance : ( Allowed in Java )
      - Interface inheritance
        - Single level inheritance : ( Allowed in Java )
        - Multiple inheritance : ( Allowed in Java )
        - Hierarchical level inheritance : ( Allowed in Java )
        - Multilevel inheritance : ( Allowed in Java )

- Use-a hierarchy
  - It represents dependency.
- Creates-a hierarchy
  - It represents instantiation.

## Typing

- It is a minor pillar of oops.
- Typing is also called as Polymorphism.
- It comes from the greek roots "poly" (many) and "morphe" (form).
- An ability of an instance to take multiple forms is called as Polymorphism.
- If we want to reduce maintainence of the system then we should use Polymorphism.
- Types of Polymorphism:
  - Compile time Polymorphism
    - We can achieve it using method overloading
  - Runtime Polymorphism
    - In Java, it is also called as dynamic method dispatch.
    - We can achieve it using method overriding.

## Concurrency

- It is a minor pillar of oops.
- It is the process of executing multiple task simultaneously.
- Main purpose of concurrency is to utilize h/w resources efficiently.
- In Java we achieve concurrency using multithreading.

## Persistence

- It is a minor pillar of oops.
- It is process of maintaining state of instance of secondary storage( HDD / DB ).
- In Java, we can achieve Persistence using serialization , JDBC etc.
- Persistence helps to achieve reliability

## Association

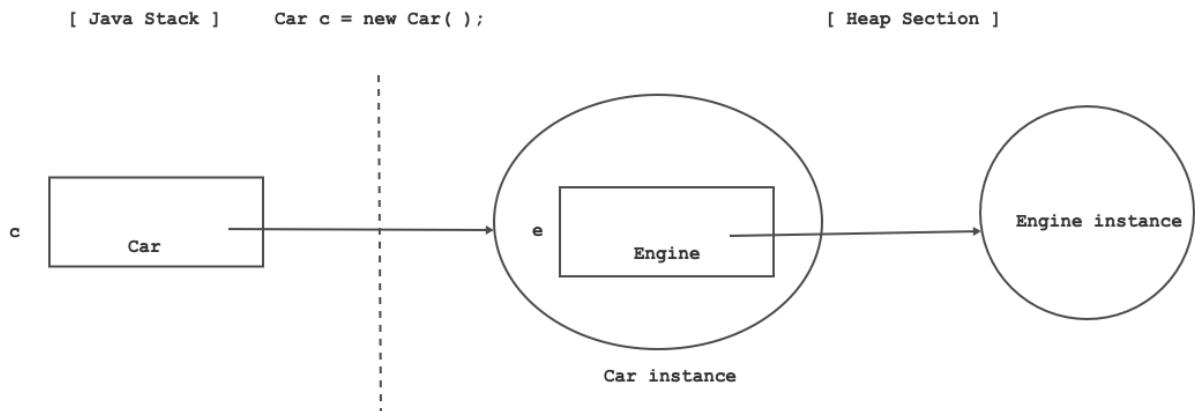
- If has-a relationship is exist between the type then we should use association.
- Consider following examples:
  - Car has-a engine / wheel.
  - Car has-a music system.
  - Room has-a wall/window
  - Room has-a chair
  - System unit has-a motherboard
  - System unit has-a modem
- Has-a hierarchy is also called as part-of hierarchy.
- Consider example of Car and Engine
  - Car has-a engine
  - Engine is part of car
- When object-instance is part/component of another object-instance then it is called as association.

- Consider following example:

```

class Engine{
    //Fields
    //Constructors
    //Methods
}
class Car{
    //Car has a engine
    private Engine e = new Engine(); //Association
}

```

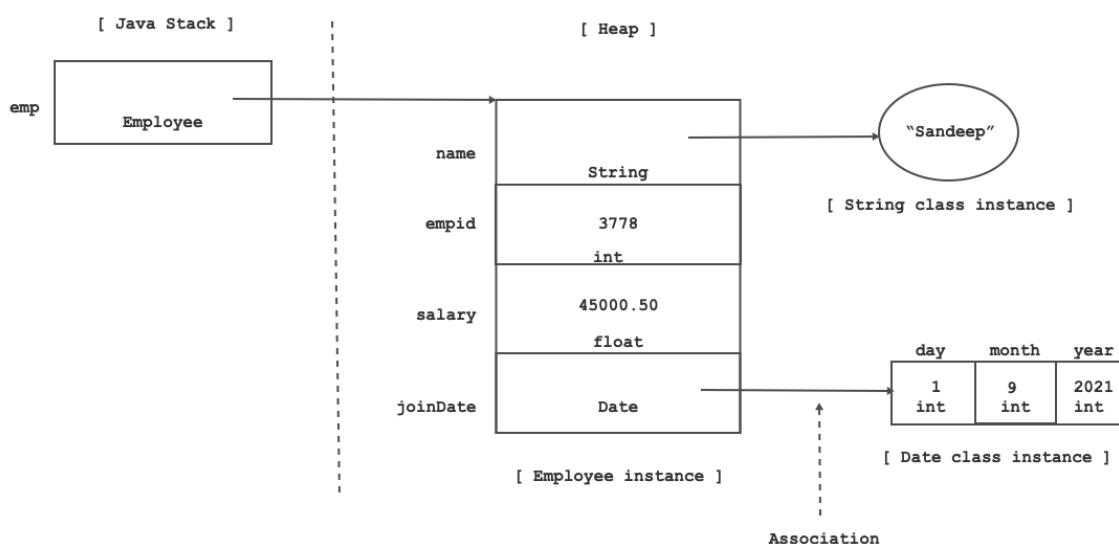


- Car Instance is dependent instance
- Engine instance dependency instance
- If we declare instance of a class as a field inside another class then it is considered as association.
- In Java, association do not represent physical containment. In other words, instance can not be part of another instance directly. Association is maintained through reference variable.

```

class Date{
    //TODO: Member declaration
}
class Employee{
    String name;
    int empid;
    float salary;
    Date joinDate;
}
class Program{
    public static void main( String[] args ){
        Employee emp = new Employee("Sandeep",3778,45000.50f,new Date(1,9,2021));
    }
}

```



## Composition

- Consider example of car and wheel & engine
  - Car has a wheel

```
class Wheel{
    //TODO
}
class Engine{

}
class Car{
    private Wheel[] wheels; //Association --> Composition
    private Engine e; //Association --> Composition
}
```

- Car instance is dependent instance
  - Wheel & Engine instance is dependency instance
- In case of association, if dependency instance can not exist without dependent instance then it represents composition.
  - Composition represents tight coupling.

## Aggregation

- Consider example of University and Student

```
class Student{
    //TODO
}
class University{
    private Student[] students; //Association --> Aggregation
    //TODO
}
```

- University instance is dependent instance.
  - Student instance is dependency instance.
- In case of association, if dependency instance exist without dependent instance then it represents aggregation.
  - Aggregation represents loose coupling.

## Inheritance

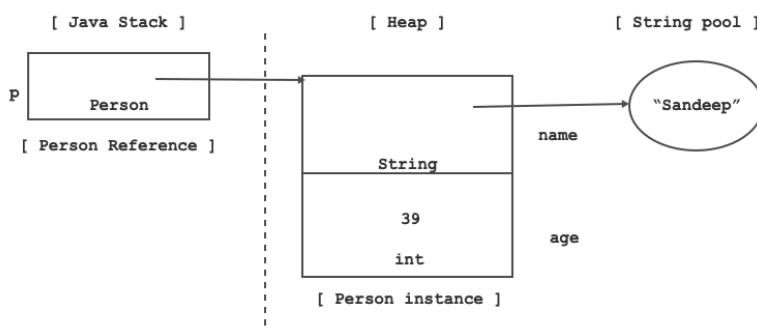
- Consider Person class

```

class Person{
    private String name;
    private int age;
    public Person( ) {
        this( "", 0 ); //Constructor chaining
    }
    public Person( String name, int age ) {
        this.name = name;
        this.age = age;
    }
    public void showRecord( ) {
        System.out.println("Name : "+this.name);
        System.out.println("Age : "+this.age);
    }
}

public class Program {
    public static void main(String[] args) {
        //Person p = new Person( );
        Person p = new Person( "Sandeep", 39 );
        p.showRecord( );
    }
}

```



- Consider Employee class

```

class Employee{
    private String name;
    private int age;
    private int empid;
    private float salary;

    public Employee() {
        this("",0,0,0.0f);
    }

    public Employee(String name, int age, int empid, float salary) {
        this.name = name;
        this.age = age;
        this.empid = empid;
    }
}

```

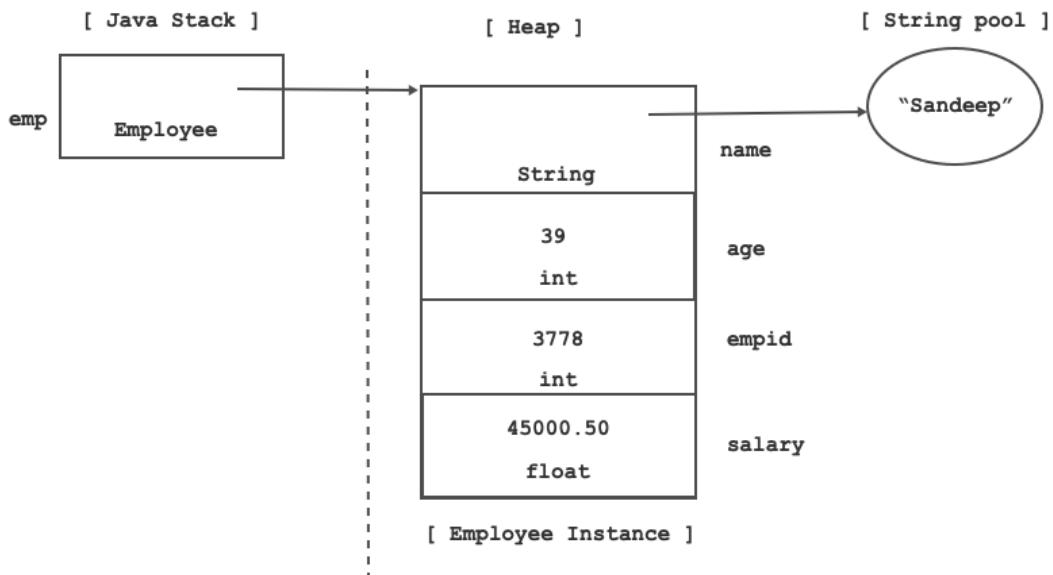
```

        this.salary = salary;
    }

    public void displayRecord( ) {
        System.out.println("Name : "+this.name);
        System.out.println("Age : "+this.age);
        System.out.println("Empid : "+this.empid);
        System.out.println("Salary : "+this.salary);
    }
}

public class Program {
    public static void main(String[] args) {
        //Employee emp = new Employee();
        Employee emp = new Employee("Sandeep", 39, 3778, 45000.50f);
        emp.displayRecord();
    }
}

```



- If a relationship exists between the types then we should use inheritance.

```

class Person{ //Parent class / Super class

}

class Employee extends Person{ //Child class / Sub class
    //Here Employee class can reuse fields/constructors/methods and
    nested types of Person class
}

```

- `extends` is a keyword which is used to create child class.
- In Java, parent class is called as super class and child is called as sub class.

- `java.lang.Object` is super class of all the classes in Java language.

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

- Now class `Person` is direct super class of class `Employee` and class `Object` is indirect super class of class `Employee`.
- In Java, any class can extend only one class.

```
class A{ } //OK
class B{ } //OK
class C extends B{ } //OK
class D extends A, B{ } //Not OK
```

- During inheritance, members of sub class do not inherit into super class. Hence using super class instance, we can access members of super class only.

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

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

class Employee extends Person{ //Sub class
    int empid;
    float salary;

    public void displayRecord( ) {
        System.out.println("Empid : "+this.empid);
        System.out.println("Salary : "+this.salary);
    }
}

public class Program {
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "Sandeep"; //OK
        p.age = 39; //OK
        //p.empid = 3778; //Not OK
```

```

    //p.salary = 45000.50f;    //Not OK
    p.showRecord();
    //p.displayRecord();   //Not OK

}
}

```

- 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.

```

class Person{ //Super class
    String name;
    int age;

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

class Employee extends Person{ //Sub class

    int empid;
    float salary;

    public void displayRecord( ) {
        System.out.println("Empid : "+this.empid);
        System.out.println("Salary : "+this.salary);
    }
}

public class Program {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.name = "Sandeep";
        emp.age = 39; //OK
        emp.empid = 3778; //OK
        emp.salary = 45000.45f; //OK
        emp.showRecord();
        emp.displayRecord();
    }
}

```

- If we create instance of sub class then all the non static fields declared in super class and sub 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/access static fields of super class. It means that static field of super class inherit into sub class.

- All( static and non static ) the fields of super class inherit into sub class but only non static fields get space inside instance of sub class.
- We can call/invoke, non static method of super class on instance of sub class. In other words, non static method of super class inherit into sub class.

```
public static void main(String[] args) {
    Employee emp = new Employee();
    emp.showRecord();
    emp.displayRecord();
}
```

- We can invoke, static method of super class on sub class name. In other words, static method of super class inherit into sub class.
- Except constructor, all(static and non static ) the methods of super class inherit into sub class.
- If we create instance of sub class then first super class constructor gets called and then sub class constructor gets called.
- From any constructor of sub class, by default, super class's parameterless constructor gets called.
- If we want to call any constructor of super class from constructor of sub class then we should use super statement.
- super is keyword in java.
- super statement must be first statement inside constructor body.

```
class Person{ //Super class
    String name;
    int age;
    public Person( ) {
        this.name = "";
        this.age = 0;
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
class Employee extends Person{ //Sub class
    int empid;
    float salary;
    public Employee( ) {
        this.empid = 0;
        this.salary = 0.0f;
    }
    public Employee( String name, int age, int empid, float salary ) {
        super( name, age ); //Super statement
    }
}
```

```

        this.empid = empid;
        this.salary = salary;
    }
}

```

- Except constructor, all the members of super class inherit into sub class.
- Definition of Inheritance
  - Journey from generalization to specialization is called inheritance.
  - Process of acquiring/accessing members of parent class into child class is called as inheritance.
- Interface inheritance:
  - During inheritance, if super type and sub types are interfaces then it is called as interface inheritance.

```

//Single Interface inheritance
interface A{
    //TODO
}
interface B extends A{ //Single Interface inheritance : OK
    //TODO
}

```

```

//Multiple Interface inheritance
interface A{
    //TODO
}
interface B{
    //TODO
}
interface C extends A, B{ //Multiple Interface inheritance :
OK
    //TODO
}

```

```

//Hierarchical interface inheritance
interface A{
    //TODO
}
interface B extends A{ //Single interface inheritance : OK
    //TODO
}
interface C extends A{ //Single Interface inheritance : OK
    //TODO
}

```

```
//TODO  
}
```

```
// Multilevel interface inheritance  
interface A{  
    //TODO  
}  
interface B extends A{ //Single interface inheritance : OK  
    //TODO  
}  
interface C extends B{ //Single Interface inheritance : OK  
    //TODO  
}
```

- Implementation inheritance

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

```
//Single implementation inheritance  
class A{  
    //TODO  
}  
class B extends A{ //Single implementation inheritance : OK  
    //TODO  
}
```

```
//Multiple implementation inheritance  
class A{  
    //TODO  
}  
class B{  
    //TODO  
}  
class C extends A, B{ //Multiple implementation inheritance :  
NOT OK  
    //TODO  
}
```

```
//Hierarchical implementation inheritance  
class A{  
    //TODO  
}  
class B extends A{ //Single Interface inheritance : OK  
    //TODO
```

```
}
```

```
class C extends A{ //Single Interface inheritance : OK
```

```
    //TODO
```

```
}
```

```
//Multilevel implementation inheritance
```

```
class A{
```

```
    //TODO
```

```
}
```

```
class B extends A{ //Single Interface inheritance : OK
```

```
    //TODO
```

```
}
```

```
class C extends B{ //Single Interface inheritance : OK
```

```
    //TODO
```

```
}
```

# Day 11

Is java support multiple inheritance?

- Java supports multiple interface inheritance.

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

- Java supports multiple interface implementation inheritance

```
interface A{ }
interface B{ }
class C implements A, B{ } //OK: Multiple interface implementation inheritance
```

- Java do not support multiple implementation inheritance

```
class A{ }
class B{ }
class C extends A, B{ } //Not OK: Multiple Implementation inheritance
```

- In C++, combination of two or more than two types of inheritance is called hybrid inheritance.
- If we combine hierarchical inheritance and multiple inheritance then it becomes diamond inheritance, which creates some problems.
- To avoid diamond problem, Java do not support multiple implementation inheritance / multiple class inheritance.
- Conclusion: In Java, class can extend only one class.

- Consider following code:

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

```

        public final void notifyAll();
    }
    class Person extends Object{
        //TODO
    }
    class Employee extends Person{
        //TODO
    }
}

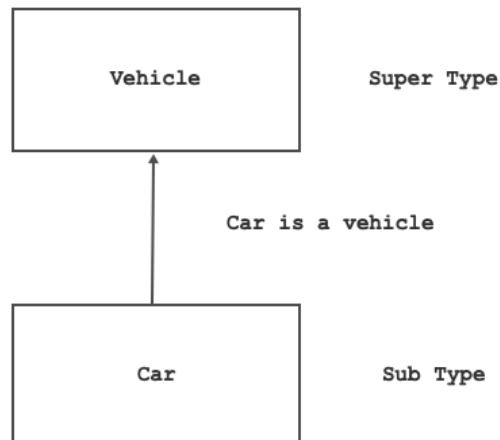
```

- In above code, class Person will extend java.lang.Object class and class Employee will extend only Person class( Not java.lang.Object class ).
- class Person is direct super class and class java.lang.Object is indirect super class of Employee class.

## Types of inheritance

### Single inheritance

- In case of inheritance, if single super type is having single sub type then it is called as single inheritance.
- For example: Car is a vehicle.



- Syntax:

```

class Vehicle{
    //TODO
}
class Car extends Vehicle{ //Single inheritance( implementation
inheritance )
    //TODO
}

```

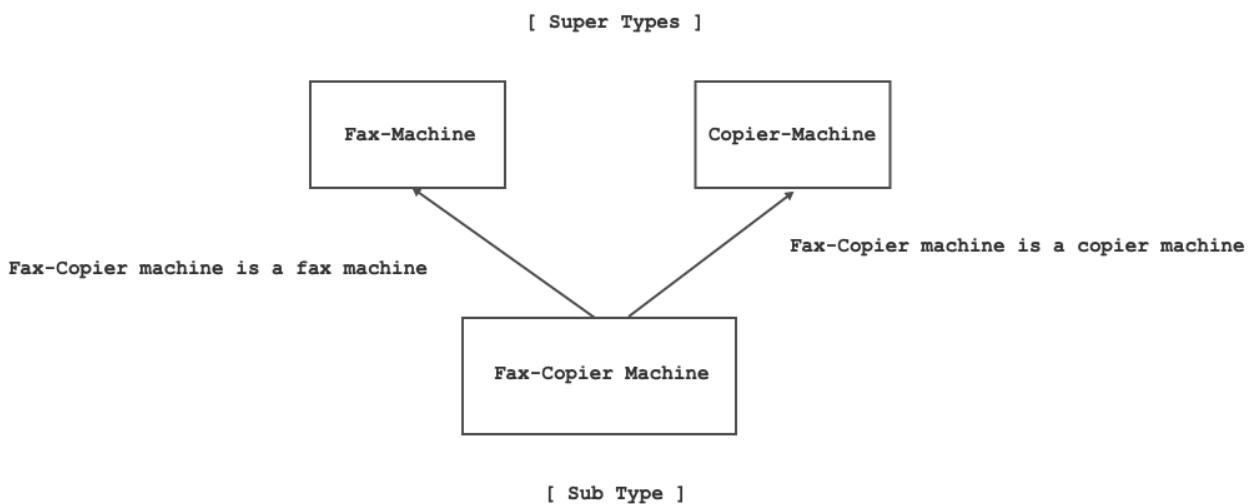
```

interface Iterable<T>{
    //TODO
}
interface Collection<E> extends Iterable<T>{ //Single inheritance(
interface inheritance )
    //TODO
}

```

## Multiple inheritance

- In case of inheritance, If multiple super types are having single sub type then it is called multiple inheritance.
- For Example: Fax-Copier machine is a fax machine and Fax-Copier machine is also Copier machine.



- Syntax:

```

class FaxMachine{} //OK
class CopierMachine{} //OK
class FaxCopierMachine extends FaxMachine, CopierMachine{} //Not
OK: Multiple Inheritance

```

```

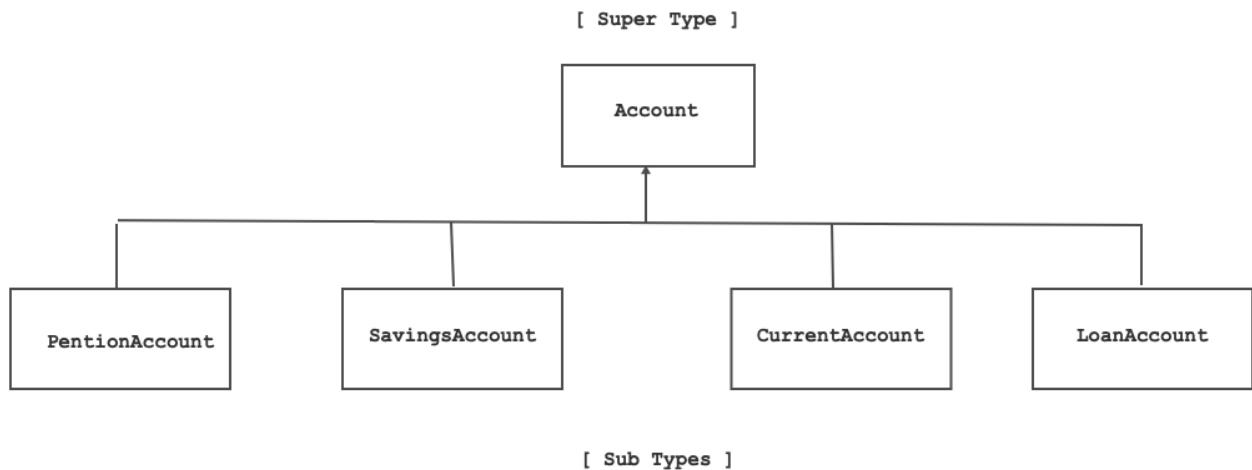
interface List<E>{}
interface Queue<E>{}
class LinkedList implements List<E>, Queue<E>{} //OK: Multiple
Inheritance

```

## Hierarchical inheritance

- In case of inheritance, If single super type is having multiple sub types then it is called as hierarchical inheritance.

- For Example: SavingsAccount is Account, CurrentAccount is a Account.



- Syntax:

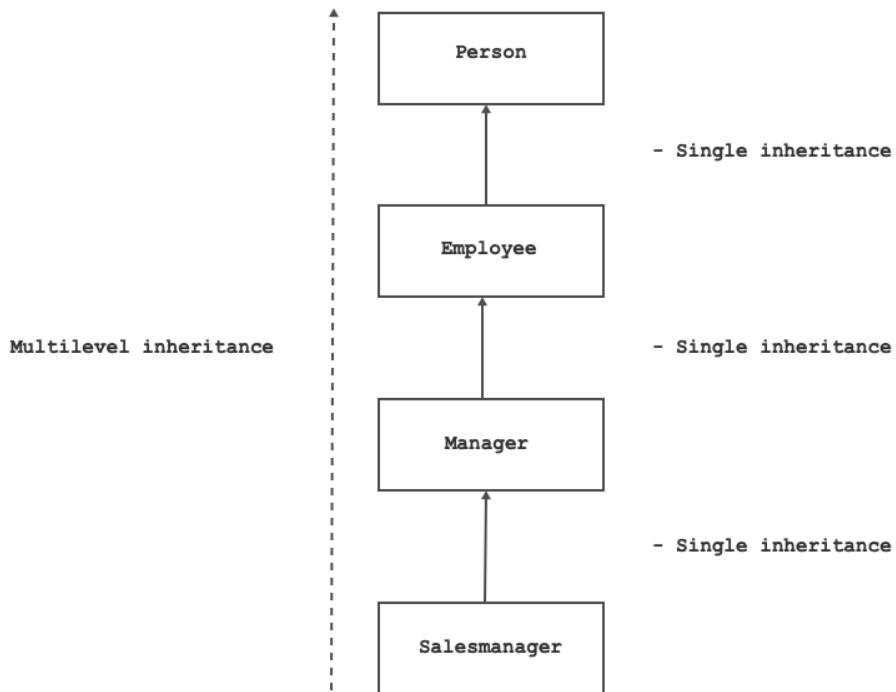
```
//Hierarchical inheritance
class Account{  }
class SavingsAccount extends Account{ } //Single inheritance
class CurrentAccount extends Account{ } //Single inheritance
```

```
interface Collection<E>{  }
interface List<Collection> extends <Collection<E>{  }
interface Queue<E> extends Collection<E>{  }
interface Set<E> extends Collection<E>{  }
```

## Multilevel inheritance

- In case of inheritance, if single inheritance is having multiple levels then it is called as multilevel inheritance.

- For Example: Employee is a Person; Manager is a Employee; Salesanager is a Manager;



- Syntax:

```
//Multilevel inheritance
class Person{ };
class Employee extends Person{ }           //Single inheritance
class Manager extends Employee{ }         //Single inheritance
class SalesManager extends Manager{ }     //Single inheritance
```

```
//Multilevel inheritance
interface Iterable<T>{ }
interface Collection<E> extends Iterable<T>{ }
interface Queue<E> extends Collection<E>{ }
interface Deque<E> extends Queue<E>{ }
```

## Syntax to use class and interface

- Interfaces: I1, I2, I3
- Classes: C1, C2, C3
  - I2 implements I1 //Not OK
  - I2 extends I1 //OK: Interface inheritance
  - I3 extends I1, I2 //OK: Multiple Interface inheritance
  - I1 extends C1; //Not OK
  - I1 implements C1; //Not OK
  - C1 extends I1; //Not OK
  - C1 implements I1; //OK: Interface implementation inheritance
  - C1 implements I1,I2; //OK: Multiple Interface implementation inheritance

- o C2 extends C1 implements I1,I2; //OK
- o C2 implements C1; //Not OK
- o C2 extends C1; // OK: Implementation inheritance
- o C3 extends C1,C2; // Not OK: Multiple Implementation inheritance

## Access modifiers revision

- Below table will describe the access modifiers in Java:

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

- Reference: <https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html>
- Private members( fields/methods/nested types ) inherit into sub class. If we want to access value of private field inside sub class then we should use getter and setter methods of super class.

```

class A{
    private int num1;
    public A( ) {
        this.num1 = 10;
    }
    public int getNum1() {
        return this.num1;
    }
}
class B extends A{

}
public class Program {
    public static void main(String[] args) {
        B b = new B();
        //System.out.println(b.num1); //The field A.num1 is not visible
        System.out.println( b.getNum1() );      //10
    }
    public static void main1(String[] args) {
        A a = new A();
        //System.out.println(a.num1); //The field A.num1 is not visible
        System.out.println( a.getNum1() );
    }
}

```

## Accessing members using super keyword

- According, clients requirement, if implementation of super class method is logically incomplete / partially complete then we should redefine method inside sub class. In other words, we should override method inside sub class.

```
public class Person {  
    private String name;  
    private int age;  
    public Person() {  
        this("", 0);  
    }  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public void printRecord() {  
        System.out.println("Name : "+this.name);  
        System.out.println("Age : "+this.age);  
    }  
}  
  
public class Employee extends Person{  
    private int empid;  
    private float salary;  
    public Employee() {  
        this("", 0, 0, 0.0f);  
    }  
    public Employee( String name, int age, int empid, float salary ) {  
        super( name, age );  
        this.empid = empid;  
        this.salary = salary;  
    }  
    public void printRecord() { //overriden method  
        System.out.println("Empid : "+this.empid);  
        System.out.println("Salary : "+this.salary);  
    }  
}  
public class Program {  
    public static void main(String[] args) {  
        Employee emp = new Employee("Sandeep", 39, 3778, 45000.50f);  
        emp.printRecord(); //Due to shadowing, Employee.printRecord()  
will call //3778, 45000.50f  
    }  
}
```

- If name of super class method and sub class method is same and if we try to invoke such method on instance of sub class then preference will be given to the sub class method. It is called as method shadowing.

- If we want to use any member of super class inside method of sub class then we should use super keyword.

```

public class Employee extends Person{
    private int empid;
    private float salary;
    public Employee() {
        this("",0,0,0.0f);
    }
    public Employee( String name, int age, int empid, float salary ) {
        super( name, age );
        this.empid = empid;
        this.salary = salary;
    }
    public void printRecord() { //overridden method
        super.printRecord();
        System.out.println("Empid : "+this.empid);
        System.out.println("Salary : "+this.salary);
    }
}

```

- According to client's requirement, if implementation of existing class is logically incomplete / partially complete then we should extend that class. In other words we should create sub class of that class i.e we should use inheritance.

## Shadowing

- In case of local variable

```

class Complex{
    private int real;
    private int imag;
    public Complex( ) {
        this.real = 10;
        this.imag = 20;
    }
    public void setReal(int real) {
        real = real; //Shadowing : Local variable is assigned to itself
    }
    public void setImag(int imag) {
        this.imag = imag;
    }
    @Override
    public String toString() {
        return this.real+" "+this.imag;
    }
}
public class Program {
    public static void main(String[] args) {
        Complex c1 = new Complex( ); //10,20
    }
}

```

```

        c1.setReal(50);
        System.out.println(c1.toString()); //10,20
    }
}

```

- In case of fields

```

class A{
    int num1 = 10;
    int num3 = 30;
}
class B extends A{
    int num2 = 20;
    int num3 = 40;
    public void printRecord( ) {
        System.out.println("Num1 : "+num1); //OK: 10
        System.out.println("Num1 : "+this.num1); //OK: 10
        System.out.println("Num1 : "+super.num1); //OK: 10

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

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

```

- In case method

```

class A{
    public void showRecord( ) {
        System.out.println("A.showRecord()");
    }
    public void printRecord( ) {
        System.out.println("A.printRecord()");
    }
}
class B extends A{
    public void displayRecord( ) {
        System.out.println("B.displayRecord()");
    }
}

```

```

    }
    public void printRecord( ) {
        System.out.println("B.printRecord()");
    }
}
public class Program {
    public static void main(String[] args) {
        B b = new B();
        b.showRecord();           //A.showRecord()
        b.displayRecord();       //B.displayRecord()
        b.printRecord();         //B.printRecord()  => Shadowing
    }
}

```

## Upcasting and downcasting

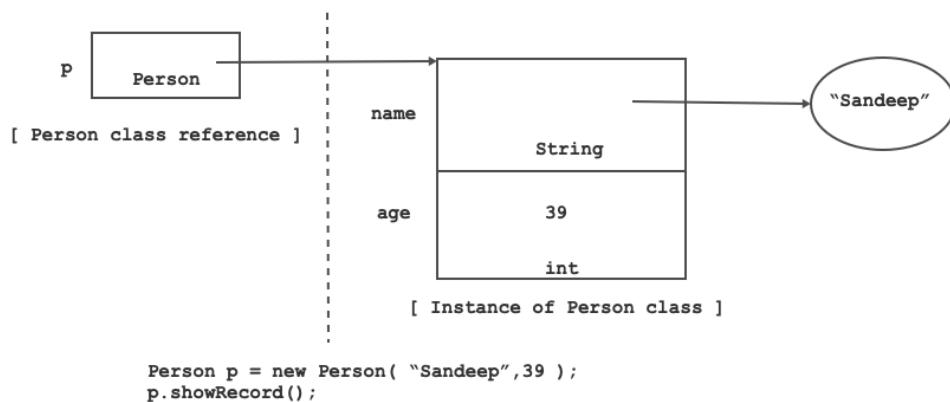
```

class Person{
    String name;
    int age;
    public Person() {
        this.name = "";
        this.age = 0;
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public void showRecord( ) {
        System.out.println("Name : "+this.name);
        System.out.println("Age : "+this.age);
    }
}
class Employee extends Person{
    int empid;
    float salary;
    public Employee() {
        super();
        this.empid = 0;
        this.salary = 0.0f;
    }
    public Employee( String name, int age, int empid, float salary ) {
        super( name, age );
        this.empid = empid;
        this.salary = salary;
    }
    public void displayRecord( ) {
        super.showRecord();
        System.out.println("Empid : "+this.empid);
        System.out.println("Salary : "+this.salary);
    }
}

```

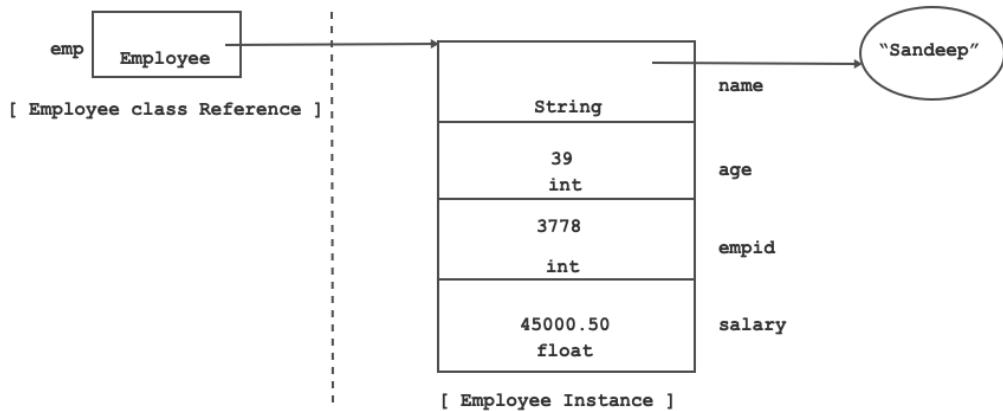
- Since members of sub class do not inherit into super class, using super class instance, we can access members of super class only.

```
public static void main1(String[] args) {
    Person p = new Person();
    p.name = "Sandeep";
    p.age = 39;
    //p.empid = 3778; //Not OK
    //p.salary = 45000.50f; //Not OK
    p.showRecord();
    //p.displayRecord(); //Not OK
}
```



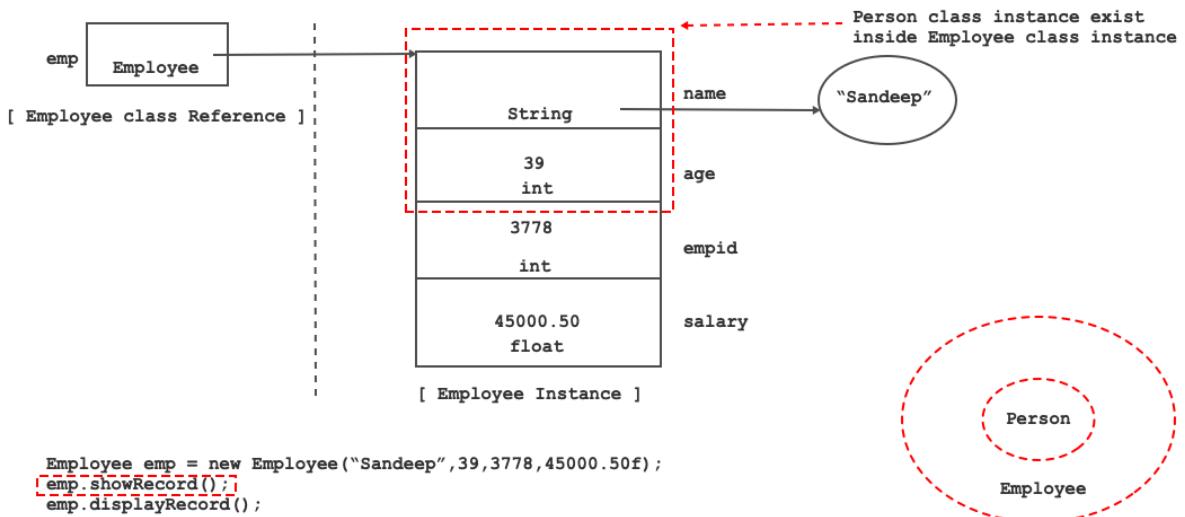
- Since members of super class inherit into sub class, using sub class instance we can access members of super class as well as sub class.

```
public static void main(String[] args) {
    Employee emp = new Employee();
    emp.name = "Sandeep";
    emp.age = 39;
    emp.empid = 3778;
    emp.salary = 45000.50f;
    emp.showRecord();
    emp.displayRecord();
}
```



```
Employee emp = new Employee("Sandeep", 39, 3778, 45000.50f);
emp.showRecord();
emp.displayRecord();
```

- Since members( fields, methods, nested type ) of super class inherit into sub class, we can consider sub class instane as super class instance.
- Since sub class instance can be considered as super class instance, we can use it in place of super class instance.



```
Person p1 = new Person(); //OK
Person p2 = p1; //OK
```

```
Employee e1 = new Employee(); //OK
Employee e2 = e1; //OK
```

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

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

- Since members of sub class do not inherit into super class, we can not consider super class instance as sub class instance.
- Since super class instance can not be considered as sub class instance, we can not use it in place of sub class instance.

```
Employee e1 = new Employee(); //OK  
Employee e2 = e1; //OK
```

```
Person p1 = new Person();  
Person p2 = p1;
```

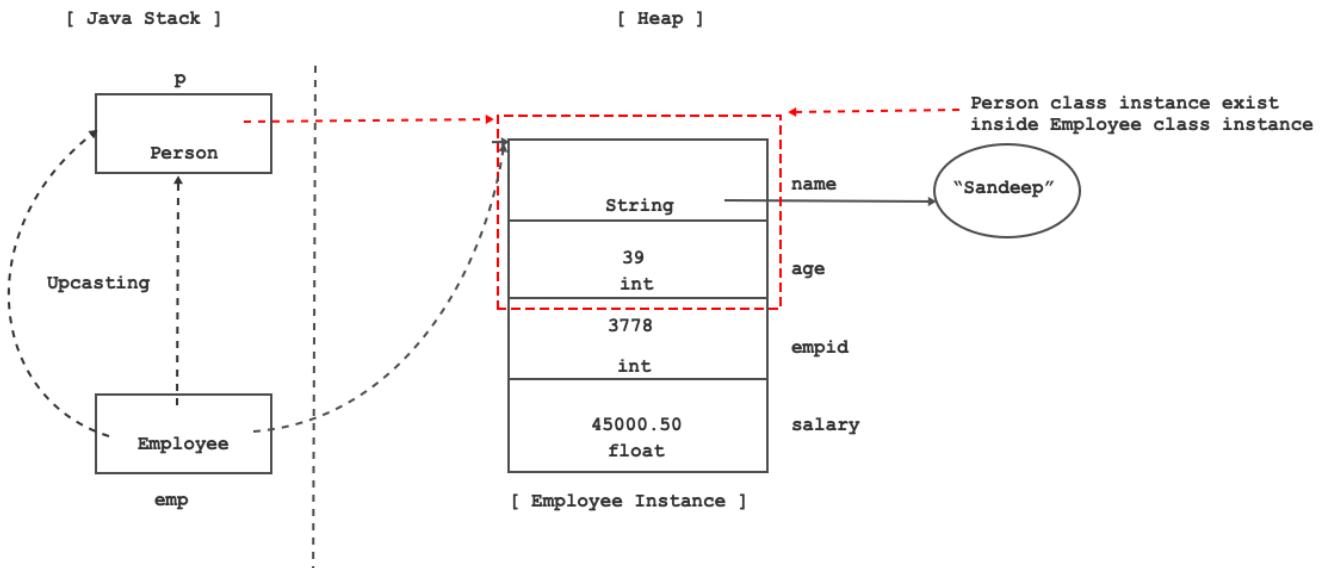
```
Person p = new Person();  
Employee emp = (Employee)p; //Not OK
```

```
Employee emp = new Person(); //Not OK
```

### Final conclusion:

- Person p = new Person(); //OK
- Person p = new Employee(); //OK
- Employee emp = new Employee(); //OK
- Employee emp = new Person(); //Not OK

### Upcasting



```
Employee emp = new Employee("Sandeep", 39, 3778, 45000.50);
//Person p = (Person)emp; //Upcasting: OK
Person p = emp; //Upcasting: OK
```

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

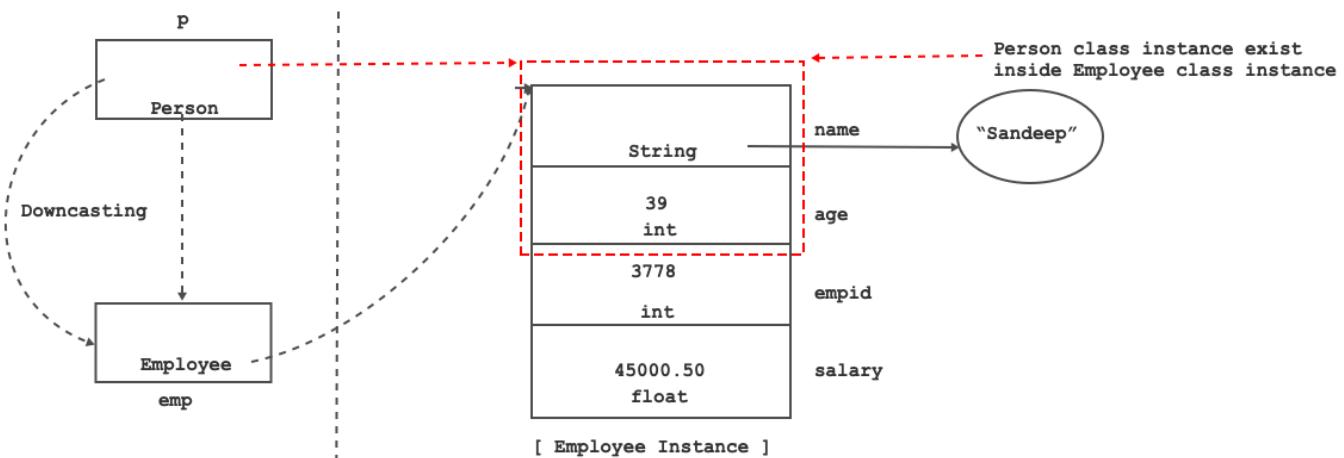
```
Employee emp = new Employee("Sandeep", 39, 3778, 45000.50);
//Person p = (Person)emp; //Upcasting: OK
Person p = emp; //Upcasting: OK
```

- Super class reference can contain reference of sub class instance. It is also called as upcasting.

```
Person p = new Employee("Sandeep", 39, 3778, 45000.50);
```

- If we want to minimize object/instance dependency in the code then we should use upcasting.

## Downcasting



```

Person p = new Employee();
p.name = "Sandeep";
p.age = 39;
Employee emp = (Employee)p; //Downcasting
emp.empid = 3778
emp.salary = 45000.50f;

```

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

```

Person p = new Employee(); //Upcasting
p.name = "Sandeep";
p.age = 39;
Employee emp = (Employee)p; //Downcasting
emp.empid = 3778
emp.salary = 45000.50f;

```

- In case of upcasting, explicit type casting is optional but in case of downcasting explicit typecasting is mandatory.

```

public static void main(String[] args) {
Person p = null;
Employee emp = (Employee) p; //Downcasting
System.out.println( p ); //null
System.out.println(emp); //null
}

```

```

public static void main(String[] args) {
Person p = new Employee(); //Upcasting
Employee emp = (Employee) p; //Downcasting: OK
}

```

```

public static void main(String[] args) {
    Person p = new Person();
    Employee emp = (Employee) p;      //Downcasting: ClassCastException
}

```

- If JVM fails to do downcasting then it throws ClassCastException.

## Method overriding

- If implementation of super class method is logically incomplete then we should redefine/override method inside sub class.
- Process of redefining, method of super class inside sub class is called as method overriding.
- In case of upcasting, process of calling method of sub class using reference of super class is called as dynamic method dispatch.

```

class Person{
    public void printRecord( ){
        System.out.println("Person.printRecord");
    }
}
class Employee extends Person{
    public void printRecord( ){
        System.out.println("Person.printRecord");
    }
}
class Program{
    public static void main(String[] args) {
        Person p = new Employee(); //Upcasting
        p.printRecord(); //Dynamic method dispatch
    }
}

```

- Note: In case of upcasting we can not access fields and non overridden methods of sub class. If we want to access it then we should do downcasting.

## Rules of method overriding

- Below are the rules of method overriding
  - Access modifier of sub class method should be same or it should be wider.
  - Return type in sub class method should be same or it should be sub type.
  - Method name, number of parameters and type of parameters in sub class method must be same.
  - Checked exception list in sub class method should be same or it should be sub set.
- Override is annotation declared in java.lang package. It helps developer to override method using above rules. If we make any mistake then it generates metadata for the compiler to generate error.

```
class Person{
    public void printRecord( ){
        System.out.println("Person.printRecord");
    }
}
class Employee extends Person{
    @Override
    public void printRecord( ){
        System.out.println("Person.printRecord");
    }
}
class Program{
    public static void main(String[] args) {
        Person p = new Employee(); //Upcasting
        p.printRecord(); //Dynamic method dispatch
    }
}
```

instanceof operator

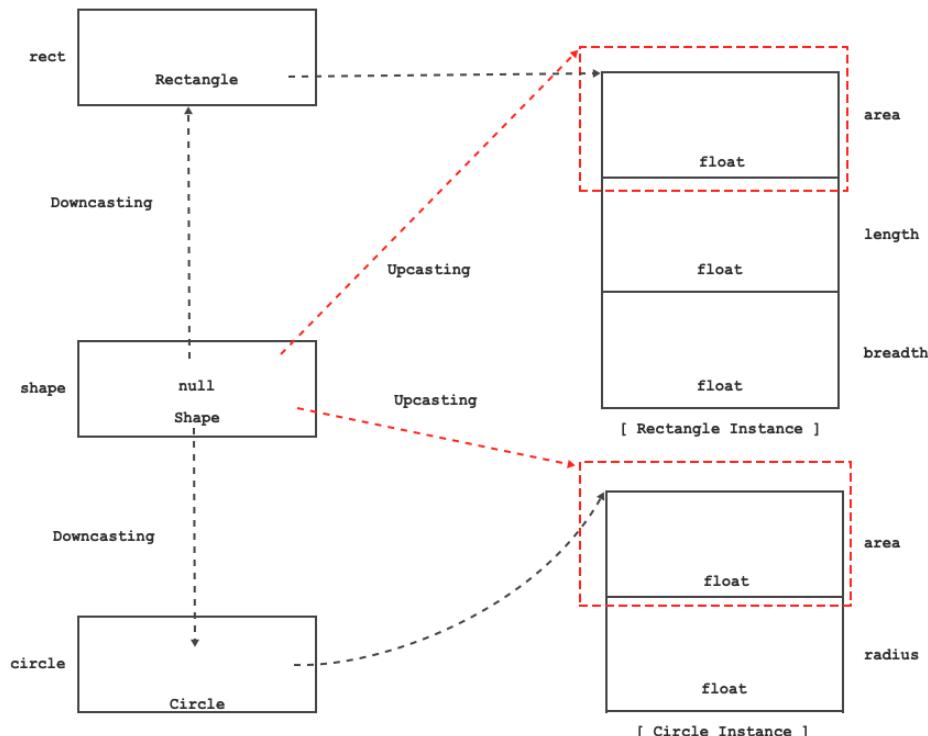
ArrayStoreException

operator== versus equals method

## Day 12

- If we want to reduce object/instance dependency in the code then we should use upcasting.
- In case of upcasting, using super class reference variable, we can access :
  1. Fields of super class( if it is non private )
  2. Methods of super class
  3. Overridden method of sub class
- In case of upcasting, if we want to access fields of sub class and non overridden method of sub class then we should do downcasting.

instanceof



- instanceof is operator in Java which returns boolean value( true / false )
- If we want to check type of instance at runtime then we should use instanceof operator. In other words, we want to check whether super class reference is containing reference of Rectangle instance of Circle instance then we should use instanceof operator.
- Consider example:

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

```

        //this.shape.calculateArea();
    }else {
        Circle circle = (Circle) this.shape;      //Downcasting
        System.out.print("Radius : ");
        circle.setRadius( sc.nextFloat());
        //this.shape.calculateArea();
    }
    this.shape.calculateArea(); //Dynamic method dispatch
}
}

```

- process of calling method of sub class on reference of super class is called as dynamic method dispatch.

### Boxing:

- Process of converting value of primitive type into non primitive type is called as boxing.
- For example:

```

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

```

```

int number = 10;
Integer i = Integer.valueOf( number ); //Boxing

```

- java.lang.Object is super class of all the classes in Java programming language.
- Consider following code:

```

Integer i1 = new Integer( 123 );
Object o1 = new Integer( 123 ); //Upcasting
Object o2 = Integer.valueOf( 123 ); //Upcasting

```

```

public static void main(String[] args) {
    int number = 123;
    Object o = number;
    // Integer.valueOf(number); //Auto-Boxing
    //Object o = Integer.valueOf(number); //Upcasting
    System.out.println( o ); //123
}

```

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

### ArrayStoreException:

- If we try to store incompatible value inside array then JVM throws ArrayStoreException
- Consider following code:

```

public static void main(String[] args) {
    /* Object o1 = new String("ABC");
    Object o2 = new String("PQR");
    Object o3 = new String("XYZ"); */

    Object[] arr = new String[ 3 ];
    arr[ 0 ] = new String("ABC");
    arr[ 1 ] = "PQR";
    arr[ 2 ] = new StringBuffer("XYZ"); //ArrayStoreException
    System.out.println(Arrays.toString(arr));
}

```

What is the difference between operator == and equals.

- We can not compare state of variable of primitive type using equals method.

```

public static void main(String[] args) {
    int num1 = 10;
    int num2 = 10;
    if( num1.equals( num2 ) )    //NOT OK: Cannot invoke equals(int) on
        the primitive type int
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
}

```

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

```

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

```

- We can use operator == with the variable of non primitive type.

```

public static void main(String[] args) {
    Employee emp1 = new Employee("Sandeep", 3778, 45000.50f );
}

```

```

Employee emp2 = new Employee("Sandeep", 3778, 45000.50f );
if( emp1 == emp2 ) //OK: Comparing state of references
    System.out.println("Equal");
else
    System.out.println("Not Equal");
//Not Equal
}

```

- If we want to compare state of references then we should use operator ==.
- If we want to compare state of instances then we should use equals method.
- equals is non final method of java.lang.Object class.
- Syntax:

```
public boolean equals( Object obj );
```

- If we do not define equals() method inside class then super class's equals method will call.
- Consider implementation of equals method from java.lang.Object class:

```

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

```

```

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", 3778, 45000.50f );
        Employee emp2 = new Employee("Sandeep", 3778, 45000.50f );
        if( emp1.equals(emp2) )
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Not Equal
    }
    public static void main1(String[] args) {
        Employee emp1 = new Employee("Sandeep", 3778, 45000.50f );

```

```

Employee emp2 = new Employee("Sandeep", 3778, 45000.50f );
if( emp1 == emp2 )      //OK: Comparing state of references
    System.out.println("Equal");
else
    System.out.println("Not Equal");
//Not Equal
}
}

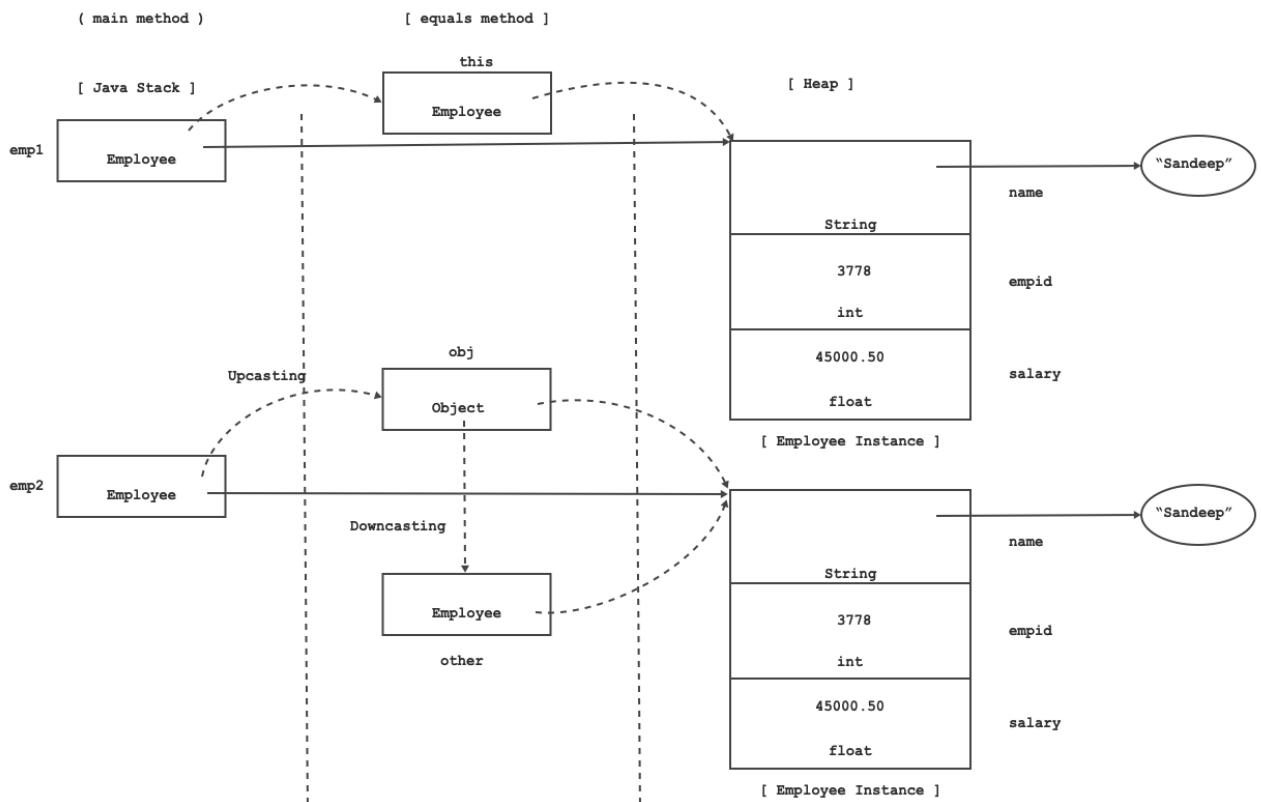
```

- According to clients requirement, If implementation of super class method is logically incomplete then we should redefine / override method in 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 = emp;    //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", 3778, 45000.50f );
        Employee emp2 = new Employee("Sandeep", 3778, 45000.50f );
        //Employee emp2 = null;
        if( emp1.equals(emp2) )
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
        //Equal
    }
}

```



# Day 13

## Project Lombok

- Reference <https://projectlombok.org/>

## External system resources

- Following are the operating system resources that we can use for the application development:
  - Memory
  - Processor
  - Input and Output devices
  - File
  - Socket
  - Network Connections
  - Database connections
  - Operating System API
- In the context of Java, all above resources are non Java resources. These are also called as unmanaged resources(except memory).
- In the context of Java, resource is any external system resource that we can use in the application.
- Since operating system resources are limited, we should use it carefully.

## Exception Concept

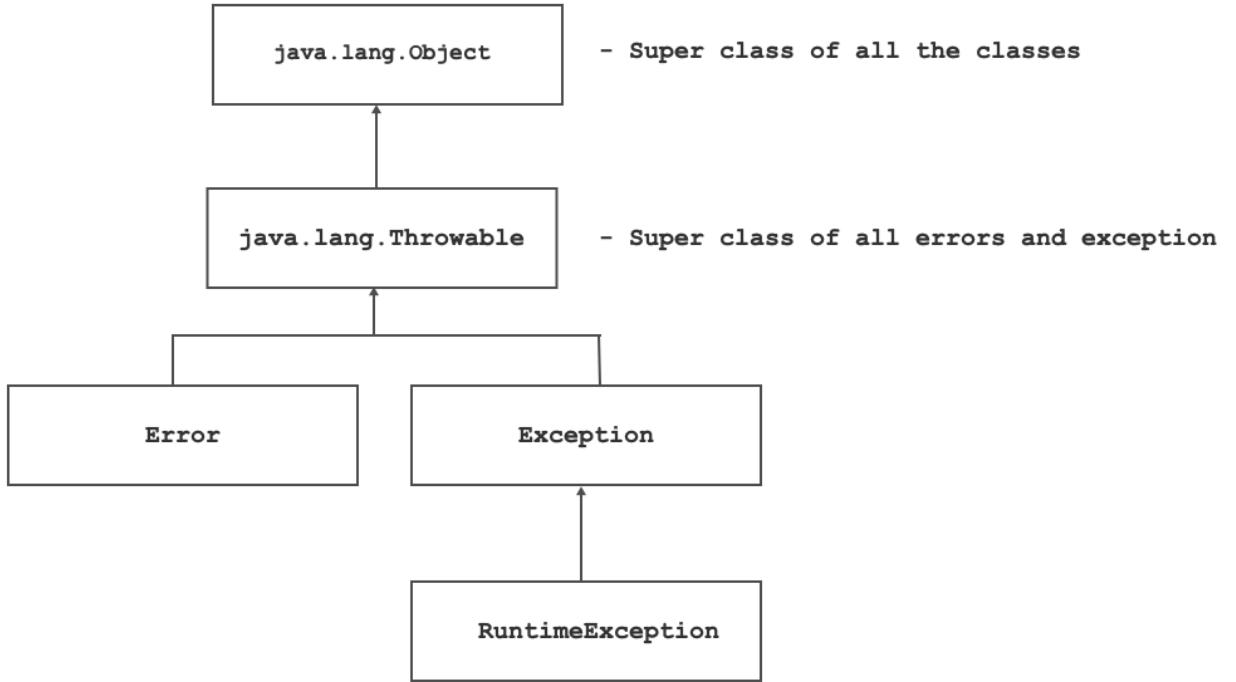
- Definition
  - Exception is an issue / unexpected event / instance which occurs during execution of application.
  - Exception is an instance which is used to acknowledge user of the system if any exception situation occurs in the code.
  - If we want to manage OS resources carefully then we should use exception handling in Java.

## Throwable class Hierarchy

- `java.lang.Object` is ultimate super class of all the classes in Java language.
- Methods of `java.lang.Object` class:
  - `public String toString( );`
  - `public boolean equals( Object obj );`
  - `public native int hashCode();`
  - `protected native Object clone( )throws CloneNotSupportedException`
  - `protected void finalize( )throws Throwable;`
  - `public final native Class<?> getClass();`
  - `public final void wait( )throws InterruptedException`
  - `public final native void wait( long timeOut )throws InterruptedException`
  - `public final void wait( long timeOut, int nanos)throws InterruptedException`
  - `public final native notify( );`
  - `public final native notifyAll( );`

### **`java.lang.Throwable:`**

*For students use only*



- 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 Java Virtual Machine or can be thrown by the Java throw statement.
- Consider code in C++

```

int main( void ){
    int num1;
    cout<<"Enter number : ";
    cin>>num1;

    int num2;
    cout<<"Enter number : ";
    cin>>num2;
    try{
        if( num2 == 0 ){
            //throw 0; //OK
            //throw ArithmeticException( "Divide by zero exception");
        //OK
            throw "Divide by zero exception"; //OK
        }else
            int result = num1 / num2;
            cout<<"Result : "<<result<<endl;
    }
    }catch( string &ex ){
        cout<<ex<<endl;
    }
    return 0;
}
    
```

- Consider code in Java

```

class MyException{
    private String message;
    public MyException(String message) {
        this.message = message;
    }
}
public class Program {
    public static void main(String[] args){
        int num1 = 10;
        int num2 = 0;
        try {
            if( num2 == 0 )
                //throw 0; //No exception of type int can be thrown; an
exception type must be a subclass of Throwable
                //throw "/ by 0"; //No exception of type String can be
thrown; an exception type must be a subclass of Throwable
                throw new MyException("/ by 0");//No exception of type
MyException can be thrown; an exception type must be a subclass of
Throwable
            int result = num1 / num2;
            System.out.println("Result : "+result);
        }catch( Exception ex ) {
            //TODO
        }
    }
}

```

- Consider following code:

```

public class Program {
    public static void main(String[] args){
        int num1 = 10;
        int num2 = 0;
        try {
            if( num2 == 0 ) {
                ArithmeticException ex = new ArithmeticException("Value of
denominator should not be zero");
                throw ex; //OK
            }
            int result = num1 / num2;
            System.out.println("Result : "+result);
        }catch( Exception ex ) {

        }

    }
}

```

- Similarly, only Throwable class or one of its subclasses can be the argument type in a catch clause.
- Consider following code:

```

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

        String str = null;
        str.charAt(10);
        int num1 = 10;
        int num2 = 0;
        try {
            int result = num1 / num2;
            System.out.println("Result : " + result);
        } catch (ArithmaticException ex) { //No exception of type
String can be thrown; an exception type must be a subclass of
Throwable

        }
    }
}

```

- Consider following code:

```

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

        String str = null;
        str.charAt(10);
        int num1 = 10;
        int num2 = 0;
        try {
            int result = num1 / num2;
            System.out.println("Result : " + result);
        } catch (ArithmaticException ex) { //
//TODO
        }
    }
}

```

- Constructor Summary
    - public Throwable()
- ```
Throwable tw = new Throwable();
```
- public Throwable(String message)

```
String message = "error message";
Throwable tw = new Throwable( message );
//or
Throwable tw = new Throwable( "error message" );
```

- public Throwable(Throwable cause)

```
Throwable cause = new Throwable( "error message" );
Throwable tw = new Throwable( cause );
//or
Throwable tw = new Throwable( new Throwable( "error message" ) );
;
```

- public Throwable(String message, Throwable cause)

```
Throwable cause = new Throwable( "error message" );
Throwable tw = new Throwable( "new error message", cause );
```

- protected Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)

```
Throwable cause = new Throwable( "error message" );
Throwable tw = new Throwable( "new error message", cause, false,
true );
```

- Method Summary

- public String getMessage()
- public Throwable initCause(Throwable cause)
- public Throwable getCause()
- public final void addSuppressed(Throwable exception)
- public void printStackTrace()
- public void printStackTrace(PrintStream s)
- public void printStackTrace(PrintWriter s)
- public StackTraceElement[] getStackTrace()

## Error versus Exception

- Error and Exception are direct sub classes of java.lang.Throwable class.

### Error

- An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch.
- Most such errors are abnormal conditions.
- Runtime error which gets generated due to environmental condition( hardware failure / OS failure / JVM failure etc ) is considered as error in java programming language.
- Consider following code:

```

public class Program {
    public static void main(String[] args) {
        try {
            int[] arr = new int[ Integer.MAX_VALUE ];
            System.out.println( Arrays.toString(arr));
        }catch( OutOfMemoryError error ) {
            System.out.println(error.getMessage()); //Requested array size
            exceeds VM limit
        }
    }
}

```

- We can write try-catch block to handle errors. But we can not recover from errors hence it is not recommended to use try catch block for the errors.
- Example:
  - OutOfMemoryError
  - StackOverflowError
  - VirtualMachineError

## Exception

- The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.
- Runtime error which gets generated due to application is considered as exception in java programming language.
- We can use try-catch block to handle exception.
- Example:
  - CloneNotSupportedException
  - InterruptedException
  - NumberFormatException
  - NullPointerException
  - NegativeArraySizeException
  - ArrayIndexOutOfBoundsException
  - ArrayStoreException
  - ClassCastException
  - ArithmeticException

Checked versus unchecked exception

- Checked exception and unchecked exception are types of exception in Java, which are designed for Java compiler( Not for JVM ).

## Unchecked Exception

- java.lang.RuntimeException is considered as super class of all the unchecked exception.
- java.lang.RuntimeException and all its sub classes are considered as unchecked exception.
- Examples of unchecked exception
  - RuntimeException
  - NumberFormatException
  - NullPointerException
  - NegativeArraySizeException
  - ArrayIndexOutOfBoundsException
  - ArrayStoreException
  - ClassCastException
  - ArithmeticException
- Compiler do not force developer to handle or to use try-catch block for unchecked exception.

## Checked Exception

- java.lang.Exception is considered as super class of all the checked exception.
- java.lang.Exception and all its sub classes except java.lang.RuntimeException(and its sub classes ) are considered as checked exceptions
- Examples of checked exception
  - java.lang.CloneNotSupportedException
  - java.lang.InterruptedIOException
  - java.io.IOException
  - java.sql.SQLException
- Compiler force developer to handle or to use try-catch block for checked exception.

## AutoCloseable and Closeable

- Closeable is an interface which is delcared in java.io package.
  - Method: void close() throws IOException
- It is introduced in JDK 1.5
- Consider following code:

```

import java.io.Closeable;
import java.io.IOException;
import java.util.Scanner;

class Test implements Closeable{
    private Scanner sc;
    public Test() {
        this.sc = new Scanner(System.in);
    }
}

```

```

@Override
public void close() throws IOException {
    this.sc.close();
}

}

public class Program {
    public static void main(String[] args) {
        try {
            Test t = new Test();

            t.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

- "void close() throws IOException" is a method of java.io.Closeable interface which is used to clean/release resources.
- If any class implements Closeable interface then that class has a ability to close resources using close method.
- AutoCloseable is an interface which is declared in java.lang package.
  - Method: void close()throws Exception
- It is introduced in JDK 1.7
- AutoCloseable is same as Closeable with gurantee of calling close method automatically.

```

//Class Test => Resource Type
class Test implements AutoCloseable{
    private sc;
    public Test() {
        this.sc = new Scanner(System.in);
    }

    @Override
    public void close() throws Exception {
        this.sc.close();
    }
}

public class Program {
    public static void main(String[] args) {
        try {
            Test t = new Test(); //new Test() => Resource
        }
    }
}

```

```

        t.close();
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

```

- Any class which implements AutoCloseable / Closeable interface is called as resource type and its instance is called resource.
- If we use try with resource then close() method gets called automatically.

### Exception handling using try catch throw throws and finally

- If we want to handle exception then we should use 5 keywords in java:
  - try
  - catch
  - throw
  - throws
  - finally
- While performing arithmetic operation, if we get any exception condition like "divide by zero" then JVM throws ArithmeticException.

#### try

- It is a keyword in Java.
- If we want to keep watch on single statement or group of statements for exception then we should use try block / handler.
- we can not define try block after catch/finally block.
- Try block must have at least one catch block or finally block or resource statement.
- Consider following syntax:

```

public static void main(String[] args) {
    try{
        //TODO
    }catch( Exception ex ){
        //TODO
    }
}

```

```

public static void main(String[] args) {
    try{
        //TODO
    }finally{
        //TODO
    }
}

```

```
public static void main(String[] args) {
    try(Scanner sc = new Scanner()){ //try-with-resource
        //TODO
    }
}
```

## throw

- It is a keyword in java.
- If we want to generate new exception then 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.

```
String ex = new String("Divide by zero exception");
throw ex; //No exception of type String can be thrown; an exception
type must be a subclass of Throwable
```

- throw statement is a jump statement.

## catch

- It is a keyword in Java.
- To handle exception, we should use catch block / handler.
- We can not define catch block before try and after finally block.
- Catch block can handle exception thrown from try block only.
- For single try block, we can provide multiple catch block. In this case, JVM can execute only one catch depending on the situation.
- In a single catch block, we can handle multiple specific exception.

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

- Do not Repeat Yourself( DRY ).
- If we want to handle multiple exceptions of super and sub types then first we must handle sub types exception.

```

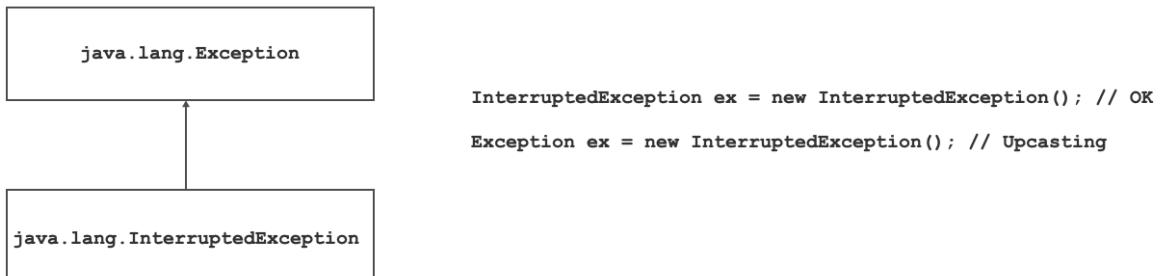
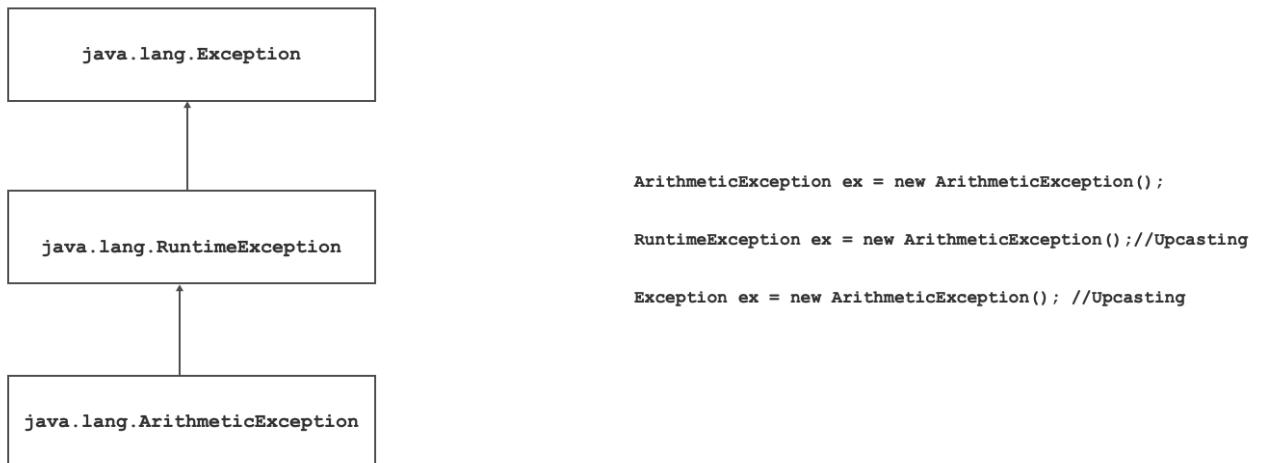
public static void main(String[] args) {
    Scanner sc = null;
    try {
        System.out.println("Opening resource");
        sc = new Scanner(System.in);

        System.out.print("Num1 : ");
        int num1 = sc.nextInt();

        System.out.print("Num2 : ");
        int num2 = sc.nextInt();

        if( num2 == 0 )
            throw new ArithmeticException("Divide by zero exception");
        int result = num1 / num2;
        System.out.println("Result : "+result);
    }catch ( ArithmeticException ex) {
        ex.printStackTrace();
    }catch ( RuntimeException ex) {
        ex.printStackTrace();
    }catch ( Exception ex) {
        ex.printStackTrace();
    }
}

```



- Using `java.lang.Exception` class we can define catch block which can handle any checked as well as unchecked exception.

```
try{
    //TODO
}catch( Exception ex ){ //Generic catch block
    ex.printStackTrace();
}
```

- Generally, generic catch block comes after all specific catch blocks.

## **finally**

- It is a keyword in Java.
- If we want to close or release local resources then we should use finally block.
- For given try block we can provide only one finally block.
- We can define block after all try and catch blocks.
- JVM always execute finally block.

## try with resource

```
public static void main(String[] args) {
    //try ( Program p = new Program()) { //Not Ok: The resource
    type Program does not implement java.lang.AutoCloseable
    try( Scanner sc = new Scanner(System.in)){
        System.out.print("Num1 : ");
        int num1 = sc.nextInt();

        System.out.print("Num2 : ");
        int num2 = sc.nextInt();

        if( num2 == 0 )
            throw new ArithmeticException("Divide by zero exception");
        int result = num1 / num2;
        System.out.println("Result : "+result);

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

## **throws**

- It is a keyword in Java
- If we want delegate exception from method to the caller method then we should use throws keyword/clause.

```

public class Program {
    public static void displayRecord( ) throws InterruptedException {
        for( int count = 1; count <= 10; ++ count ) {
            System.out.println("Count : "+count);
            Thread.sleep(500);
        }
    }
    public static void main(String[] args) {
        try {
            Program.displayRecord();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

## Custom exception and its need.

- JVM do not understand exceptional conditions in the business logic. To handle it we should define user defined / custom exception class.
- How to define custom unchecked exception class?

```

class StackOverflowException extends RuntimeException{
    //TODO
}

```

- How to define custom checked exception class?

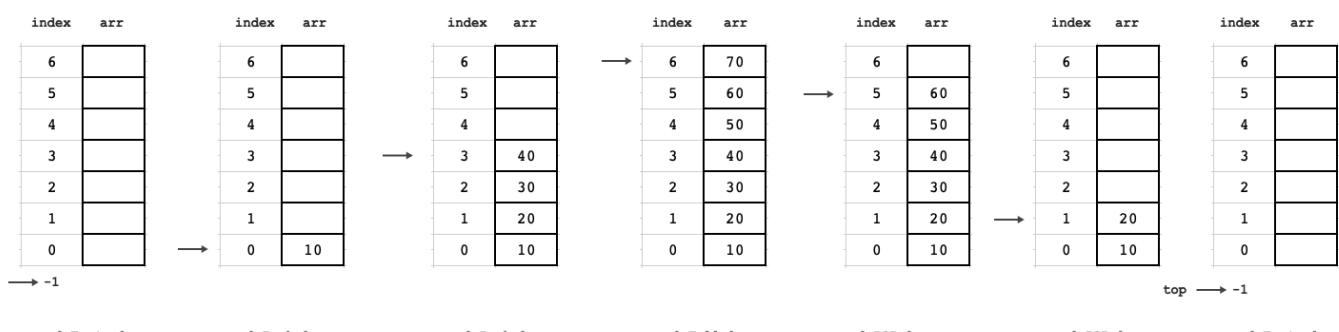
```

class StackOverflowException extends Exception{
    //TODO
}

```

## Exception chaining

Stack: Last In First Out (LIFO) operations



- Process of handling exception by throwing new type of exception is called as exception chaining.
- Consider following code:

```

package org.example;
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(250);
            }
        } catch (InterruptedException cause) {
            throw new RuntimeException(cause); //Exception Chaining
        }
    }
}
public class Program {
    public static void main(String[] args) {
        try {
            A a = new B();
            a.print(); //Dynamic method dispatch
        } catch (RuntimeException e) {
            //e.printStackTrace();
            Throwable cause = e.getCause();
            System.out.println(cause);
        }
    }
}

```

## Day 14

```
class A extends Exception{      }
class B extends Exception{      }
class C extends Exception{      }
public class Program {
    //public static void print( int number ) throws A, B, C {
    public static void print( int number ) throws Exception {
        if( number == 10 )
            throw new A();
        if( number == 20 )
            throw new B();
        if( number == 30 )
            throw new C();
        System.out.println("Number      :      "+number);
    }
    public static void main(String[] args) {
        try {
            Program.print(50);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- We can not override static method inside sub class:

```
class Super{
    public static void showRecord( ) {
        System.out.println("Super.showRecord()");
    }
}
class Sub extends Super{
    @Override
    public static void showRecord( ) {    //Compiler Error
        System.out.println("Sub.showRecord()");
    }
}
```

- Non static methods are by default virtual in Java.
- In Java virtual methods are designed to call on object reference.
- Static methods are designed to call on class name. Since static methods are not designed to call on object reference it is not virtual by default. Hence we can not override static method inside sub class.

```

class Super{
    public static void showRecord( ) {
        System.out.println("Super.showRecord()");
    }
}
class Sub extends Super{
    public static void showRecord( ) {
        System.out.println("Sub.showRecord()");
    }
}
public class Program {
    public static void main(String[] args) {
        Super.showRecord(); //Super.showRecord()
        Sub.showRecord(); //Sub.showRecord() //Due to method shadowing
    / method hiding
    }
}

```

```

class Super{
    public void printRecord( ) {
        System.out.println("Super.printRecord()");
    }
}
class Sub extends Super{
    @Override
    public void printRecord( ) { //Overrided method
        System.out.println("Sub.printRecord()");
    }
}
public class Program {
    public static void main(String[] args) {
        //Super s1 = new Super();
        //s1.printRecord();

        Sub s2 = new Sub();
        s2.printRecord(); //s2.printRecord() //Due to method shadowing
    / method hiding
    }
}

```

## Rules of method overriding

- Access modifier in sub class method should be same or it should be wider.
- Return type of sub class method should covariant. It means that, return type in sub class method should be same or it should be sub type.
- Method name, number of parameters and type of parameters inside sub class method must be same.
- Checked exception list in sub class method should be same or it should be sub set.

## Following methods we can not override / redefine inside sub class

- private method
- static method
- constructor
- final method

## Final method final class

- According business requirement, if implementation of super class method is logically incomplete then we should override method inside sub class.
- According business requirement, if implementation of super class method is logically 100% complete then we should declare super class method final.
- final is modifier in Java.
- We can not redefine final method inside sub class. In other words, we can not override final method inside sub class.
- Final method inherit into sub class. Hence we can use it inside sub class.
- Examples of final method:
  - public final int ordinal();
  - public final String name();
  - public final Class<?> getClass();
  - public final void wait( )throws InterruptedException
  - public final native void wait( long timeOut )throws InterruptedException
  - public final void wait( long timeOut, int nanos )throws InterruptedException
  - public final void notify();
  - public final void notifyAll();
- We can declare overriden method final.

```
class A{
    public void f2( ) {
        System.out.println("A.f2");
    }
    public final void f3() {
        System.out.println("A.f3");
    }
}
class B extends A{
    @Override
    public final void f2() {
        System.out.println("B.f2");
    }
}
public class Program {
    public static void main(String[] args) {
        B b = new B();
        b.f2();
    }
}
```

- According business requirement, if implementation of class is logically 100% complete then we should declare such class final.
- We can not extend final class. In other words, we can not create sub class of final class.
- Examples of final class:
  - java.lang.System
  - java.lang.String
  - java.lang.StringBuffer
  - java.lang.StringBuilder
  - All the wrapper classes
  - java.lang.Math
  - java.util.Scanner

## Abstract method an abstract class

- According business requirement, if implementation of super class method is logically 100% incomplete then we should declare super class method abstract.
- abstract is modifier in java.
- We can not provide body to the abstract method.
- If we declare method abstract then we must declare class abstract.
- We can not instantiate abstract class. In other words, we can not create instance of abstract class but we can create reference of abstract class.
- Abstract class may/may not contain abstract method.
- If super class contains abstract method then sub class must override it otherwise sub class will be considered as abstract.
- Consider following code:

```
abstract class A{
    public abstract void f1( );
}
class B extends A{
    @Override
    public void f1() {
        //TODO
    }
}
```

- Consider following code:

```
abstract class A{
    public abstract void f1( );
}
abstract class B extends A{
```

```
}
```

- Examples of abstract class:

- java.lang.Enum
- java.lang.Number
- java.util.Calendar
- java.util.Dictionary

- Examples of abstract methods:

- public abstract int intValue();
- public abstract float floatValue()
- public abstract double doubleValue()
- public abstract long longValue()

- Without declarraig method abstract, we can can declare class abstract.

```
abstract class A{
    public abstract void f1( );
    public abstract void f2( );
    public abstract void f3( );
}

abstract class B extends A{
    @Override
    public void f1() { }
    @Override
    public void f2() { }
    @Override
    public void f3() { }
}

class C extends B {
    @Override
    public void f1() {
        System.out.println("C.f1");
    }
}

class D extends B {
    @Override
    public void f2() {
        System.out.println("D.f2");
    }
}

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

public class Program {
```

```

public static void main(String[] args) {
    A a = null;

    a = new C();
    a.f1();

    a = new D();
    a.f2();

    a = new E();
    a.f3();
}
}

```

## Sole constructor

- A constructor of super class which is designed to call from constructor of only 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;
    }
    public void printRecord( ) {
        System.out.println("Num1 : "+this.num1);
        System.out.println("Num2 : "+this.num2);
    }
}
class B extends A{
    private int num3;
    public B( int num1, int num2, int num3 ) {
        super( num1, num2 );
        this.num3 = num3;
    }
    @Override
    public void printRecord() {
        super.printRecord();
        System.out.println("Num3 : "+this.num3);
    }
}
public class Program {
    public static void main(String[] args) {

        B b = new B( 10, 20, 30 );
        b.printRecord();
    }
}

```

Wrapper class hierarchy

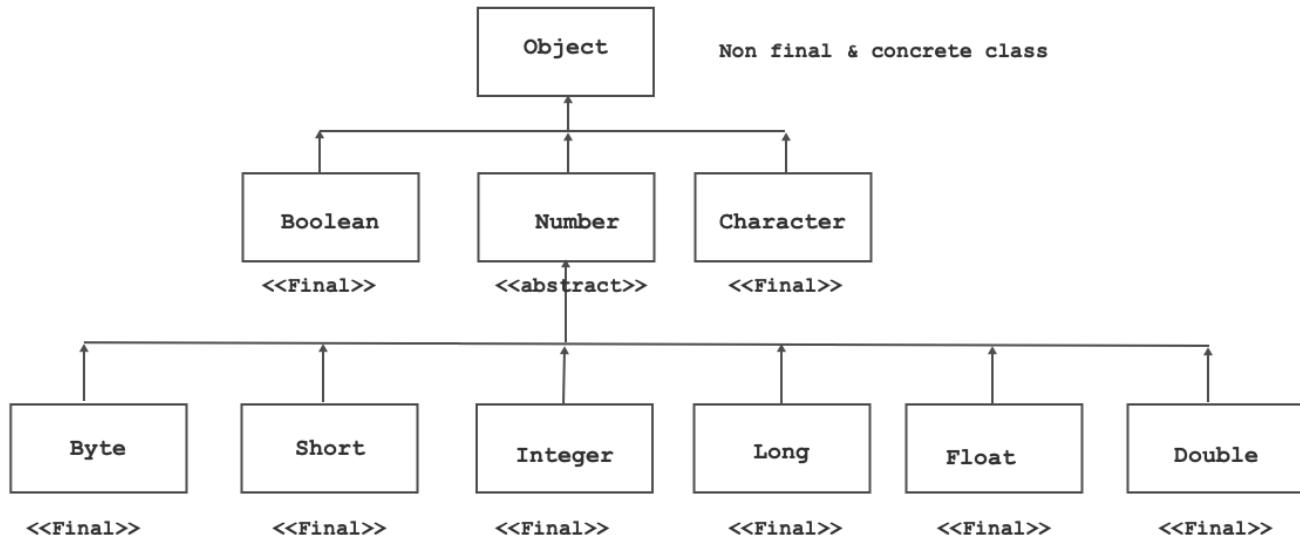
Boxing & auto-boxing

Unboxing & auto-unboxing

Generic programming

# Day 15

## Wrapper class



- Primitive Types in Java
  - boolean, byte, char, short, int, float, double, long.
- In Java, primitive types are not classes. If we want to process values of primitive type then we can use corresponding class. It is called wrapper class.
- All the wrapper classes are final and declared in `java.lang` package.
- All the wrapper classes implements `java.lang.Comparable` and `java.io.Serializable` interface.
- Any Wrapper class do not contain parameterless constructor
- Super class reference variable can contain reference of instance of sub class. It is called as upcasting.

```
Object o1 = new Boolean( true ); //OK: Upcasting
Object o2 = new Character( 'A' ); //OK: Upcasting
Object o3 = new Byte( 123 ); //OK: Upcasting
Number n1 = new Byte( 123 ); //OK: Upcasting
Byte b1 = new Byte( 123 ); //OK
Object o4 = new Short( 9999 ); //OK: Upcasting
Number n2 = new Short(9999 ); //OK: Upcasting
Object o5 = new Integer( 123456 ); //OK: Upcasting
Number n3 = new Integer( 123456 ); //OK: Upcasting
Object o6 = new Long(123456789 ); //OK: Upcasting
Number n4 = new Long(123456789); //OK: Upcasting
Object o7 = new Float(3.14f); //OK: Upcasting
Number n5 = new Float(3.14f); //OK: Upcasting
Object o8 = new Double(3.142d); //OK: Upcasting
Number n6 = new Double(3.142d); //OK: Upcasting
```

- Consider example of integer

```

int n1 = new int( 10 ); //Not OK
Integer n1 = new Integer( 10 ); //OK
Number n2 = new Integer( 10 ); //OK: Upcasting
Object n3 = new Integer( 10 ); //OK: Upcasting

```

## Boxing and Auto-Boxing

- Consider following example:

```

int number = 123;
Integer i1 = new Integer( number ); //Boxing

```

```

int number = 123;
Integer i2 = Integer.valueOf( number ); //Boxing

```

```

int number = 123;
String s1 = Integer.toString( number ); //Boxing

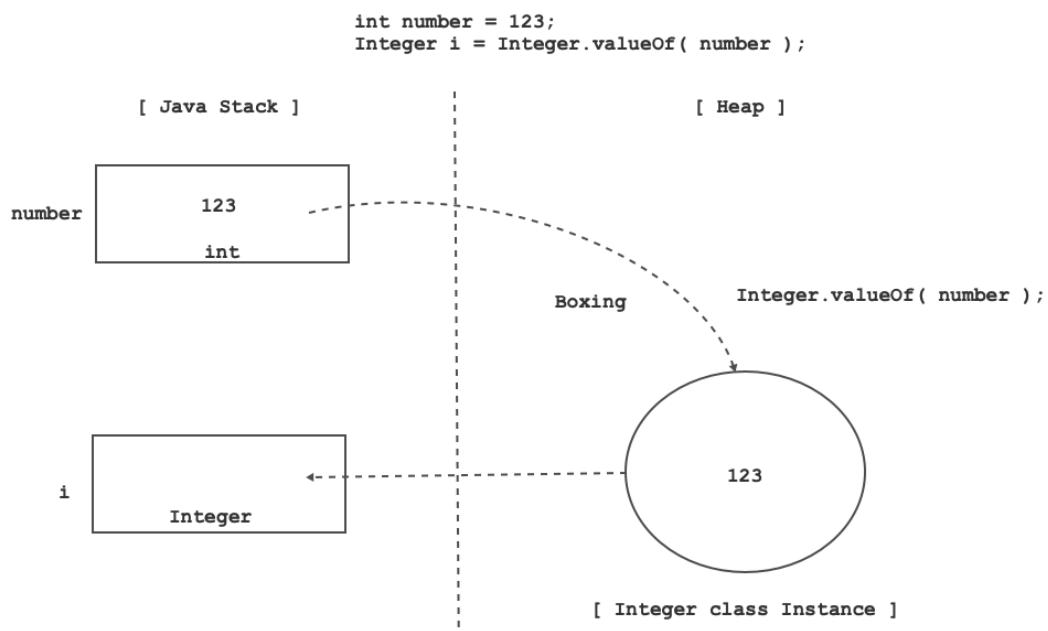
```

```

int number = 123;
String s2 = String.valueOf( number );

```

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



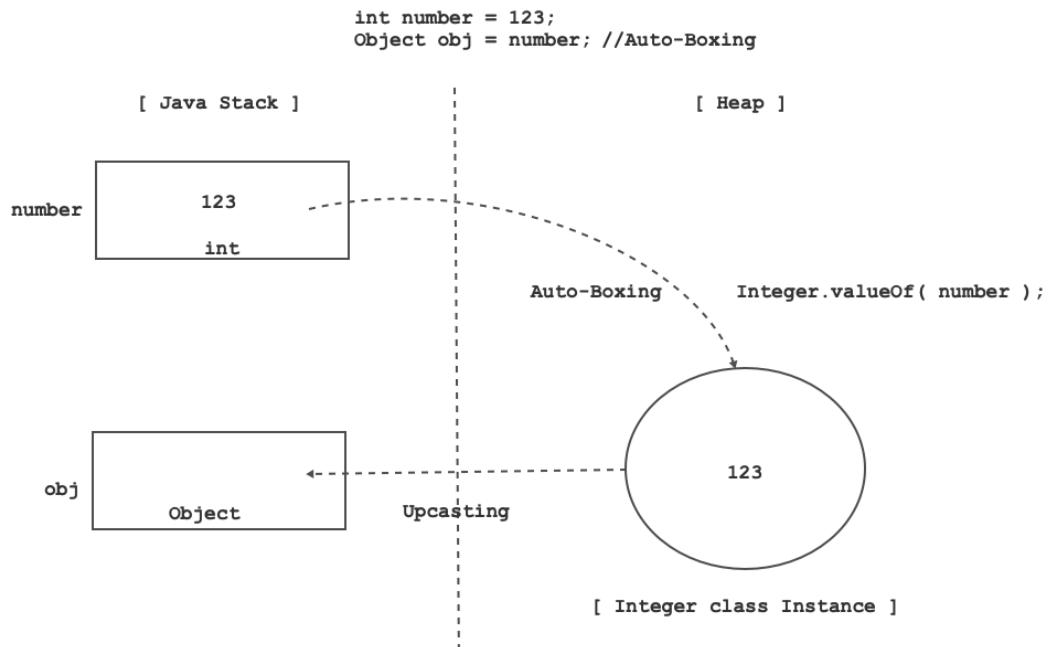
- Consider following example:

```

public static void main(String[] args) {
    int number = 123;
    Object obj = number;      //Auto-Boxing
    //Integer i = Integer.valueOf(number)    //Boxing
    //Object o = i; //Upcasting
    System.out.println( obj );
}

```

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



## UnBoxing and Auto-Unboxing

- Consider following example:

```

//Integer i = new Integer( "123" );
Integer i = new Integer( 123 );
int number = i.intValue(); //UnBoxing

```

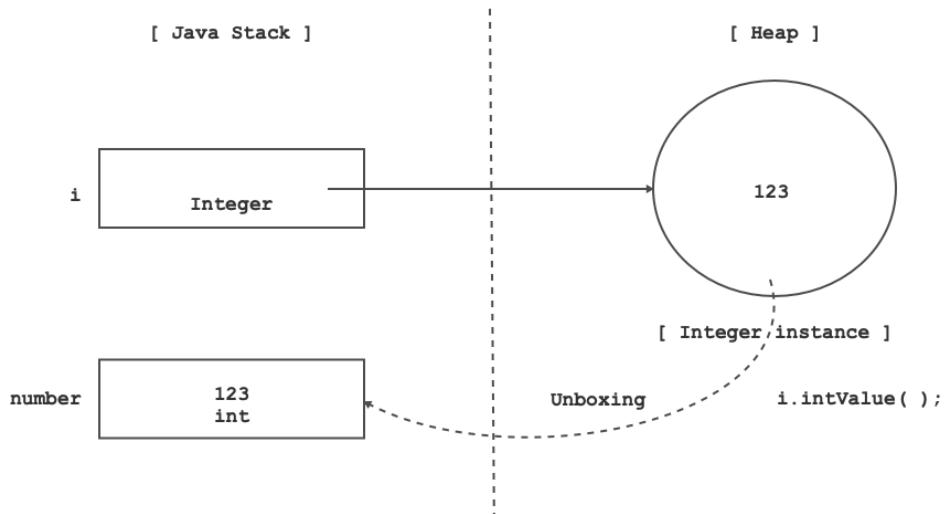
```

String str = "456";
int number = Integer.parseInt( str ); //UnBoxing

```

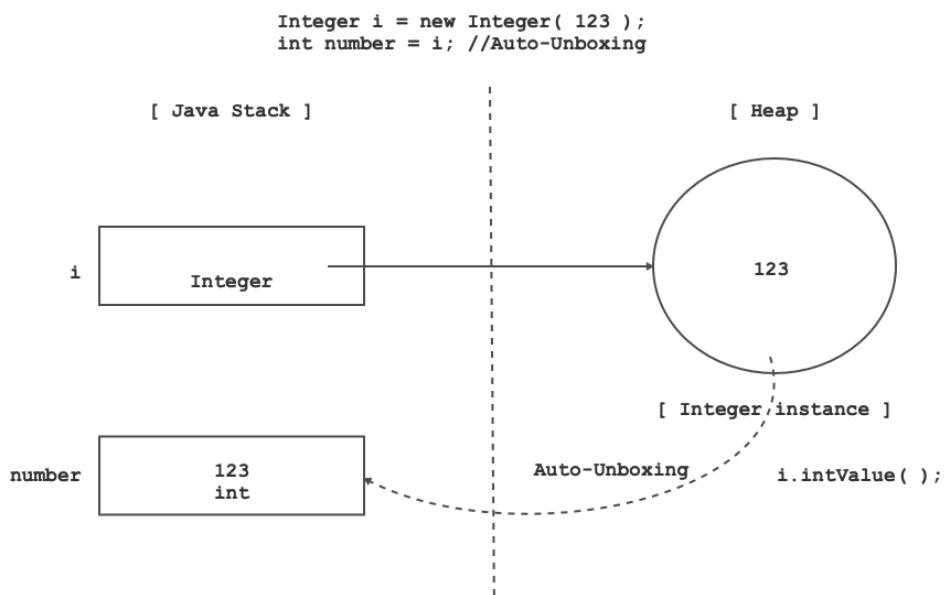
- Process of converting state of non primitive type into primitive type is called as unboxing.

```
Integer i = new Integer( 123 );
int number = i.intValue(); //Unboxing
```



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

```
public static void main(String[] args) {
    Integer i = new Integer(123);
    int number = i; //Auto-Unboxing
    //int number = i.intValue();
    System.out.println("Number : " + number);
}
```



## Upcasting and downcasting

- Upcasting definition

- We can convert reference of sub class into reference of super class. It is called as upcasting.

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

- We can directly store reference of sub class instance into super class reference variable. It is called as upcasting.

```
Person p = new Employee(); //Upcasting
```

- If we want to reduce object/instance dependency in the code then we should use upcasting.
- In case of upcasting, using super class reference variable we can access:
  - non private fields of super class.
  - non private methods of super class.
  - overridden methods of sub class.
- Downcasting definition
  - We can convert reference super class into reference of sub class. It is called as downcasting.

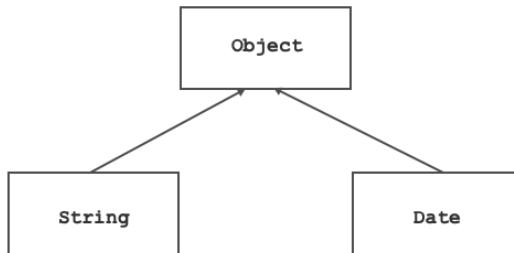
```
Person p = new Employee(); //Upcasting
Employee emp = (Employee)p; //Downcasting
```

- If JVM failed to do downcasting then it throws ClassCastException.

```
Employee e = new Person(); //Compile time checking => Compiler
Error

Person p = new Person();
Employee emp = (Employee)p; //Downcasting: ClassCastException
```

- In case of upcasting, using super class reference variable, we can not access non private fields and non overridden methods of sub class. If we want to access it then we should do downcasting.



```
String str = new String("CDAC");//OK
```

```
Object o1 = new String("CDAC"); //OK: Upcasting
String s1 = (String)o1;           //OK: Downcasting
```

```
Object o2 = null;
String s2 = (String)o2; //OK: Downcasting
```

```
Date date = new Date();//OK
```

```
Object o1 = new Date(); //OK: Upcasting
Date dt1 = (Date)o1;   //Ok: Downcasting
```

```
Object o2 = null;
Date dt2 = (Date)o2; //OK: Downcasting
```

```
Object o3 = new Date(); //Ok: Upcasting
Date dt3 = (Date)o3;   //OK: Downcasting
String s3 = (String)o3; //ClassCastException
```

- Consider following code:

```
class A{
    public void print( ){
        System.out.println("A.print");
    }
    public void display( ){
        System.out.println("A.display");
    }
}
class B extends A{
    @Override
    public void print( ){
        System.out.println("B.print");
    }
    public void show( ){
        System.out.println("B.show");
    }
}
```

```
class Program{
    public static void main( String[] args ){
        A a1 = new A(); //OK
        a1.print(); //A.print

        B b1 = new B(); //OK
        b1.print(); //B.print => Due to shadowing preference will be given
        to B.print

        A a2 = new B(); //OK: Upcasting
        a2.print(); //B.print
```

```

    a2.display( );//A.display
    a2.show( ); //Compiler error

    B b2 = new A(); //Compiler error
    b2.print( );
}
}

```

## Generic Programming

- Let us define stack to store boolean elements

```

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

```

- Let us define stack to store integer elements

```

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

```

- Let us define stack to store double elements

```

class Stack{
    private int top = -1;
    private double[] arr;
    public Stack( ) {
        this( 5 );
    }
    public Stack( int size ) {
        this.arr = new double[ size ];
    }
    public boolean empty( ) {
        return this.top == -1;
    }
    public boolean full( ) {
        return this.top == this.arr.length - 1;
    }
    public void push( double element ) throws StackOverflowException {

```

```

        if( this.full() )
            throw new StackOverflowException("Stack is full");
        this.top = this.top + 1;
        this.arr[ this.top ] = element;
    }
    public double peek( ) throws StackUnderflowException {
        if( this.empty() )
            throw new StackUnderflowException("Stack is empty");
        return this.arr[ this.top ];
    }
    public void pop( ) throws StackUnderflowException {
        if( this.empty() )
            throw new StackUnderflowException("Stack is empty");
        this.top = this.top - 1;
    }
}

```

- If we want to reduce developers effort then we should do generic programming.
- In Java, we can write generic code using:
  1. java.lang.Object class
  2. Generics

### Generic programming using java.lang.Object class

- We can use java.lang.Object class to store value of primitive as well as non-primitive. Hence to write generic code we use java.lang.Object class.

```

int number = 123;
Object obj = number; //OK
/*
Integer i = Integer.valueOf( number ); //Boxing
Object obj = i; //Upcasting
*/

```

```
Object obj = new Date(); //OK
```

- Consider following code:

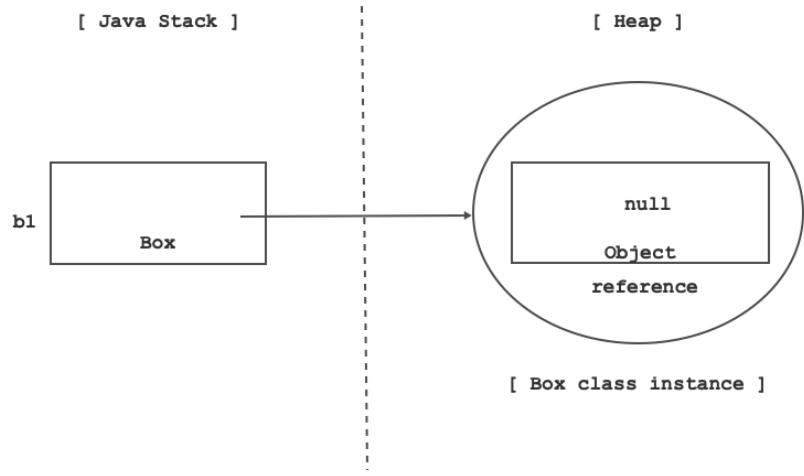
```

class Box{
    private Object reference;
    public Object getReference() {
        return reference;
    }
    public void setReference(Object reference) {
        this.reference = reference;
    }
}

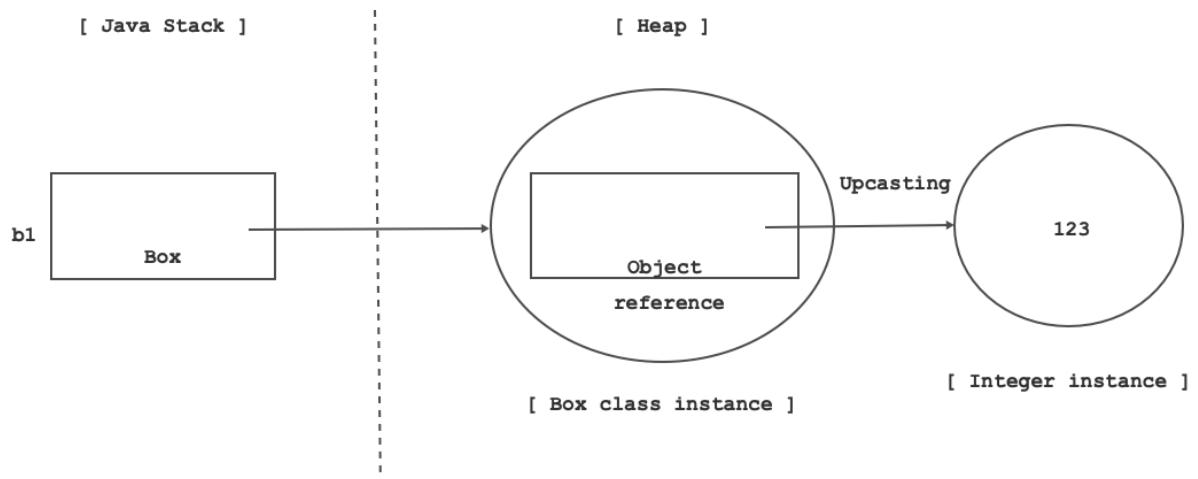
```

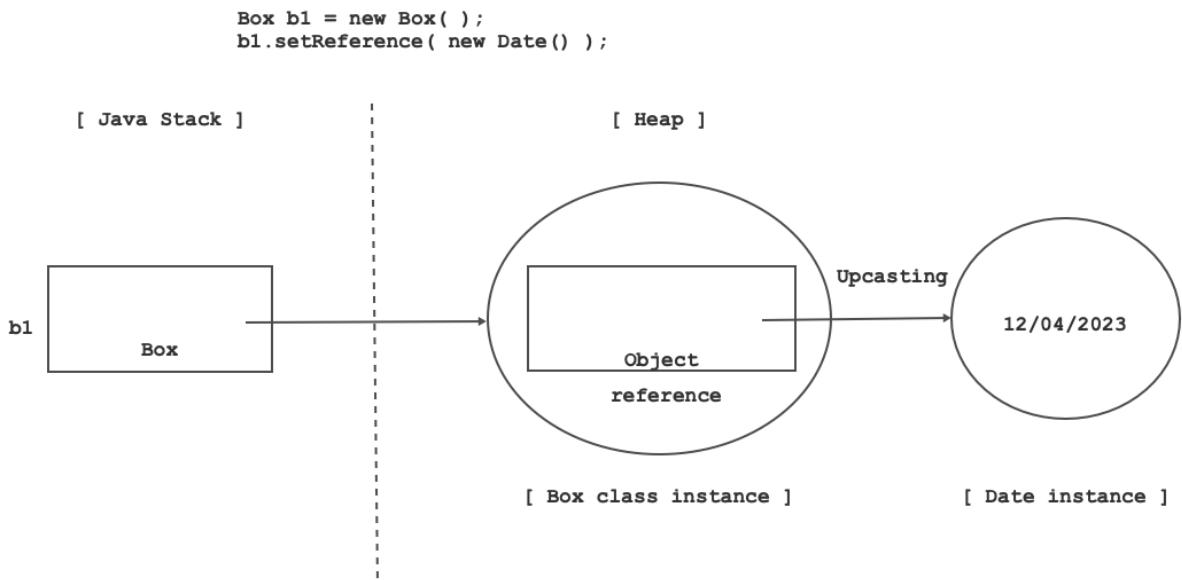
```
}
```

```
Box b1 = new Box( );
```



```
Box b1 = new Box();
b1.setReference( 123 );
```





- Using `java.lang.Object` class, we can write generic code but we can not write type-safe generic code.

```

public static void main(String[] args) {
    Box b1 = new Box();
    b1.setReference(new Date());
    //Date date = (Date) b1.getReference();      //Downcasting
    String str = (String) b1.getReference();     //Downcasting:
    ClassCastException
}

```

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

## Generics

- Generics is a Java language feature which helps developer to write generic code by passing data type as a argument.
- Consider generic code using Generics:

```

//Now class Box is called as parameterized type
class Box<T>{ // Here T is Type parameter
    private T reference;
    public T getReference() {
        return reference;
    }
    public void setReference(T reference) {
        this.reference = reference;
    }
}
public class Program {
    public static void main(String[] args) {

        Box<Date> b1 = new Box<Date>(); //Here Date is called as Type
    }
}

```

```

    argument

        b1.setReference(new Date());

        Date date = b1.getReference();

        System.out.println(date);
    }
}

```

## Why Generics

- Generics gives us stronger typechecking at compile time. In other words, using generics we can write type-safe generic code.
- It completely eliminates need of explicit typecasting.
- It helps developer to define generic algorithms and data structures.

### Generics Syntax:

- We can specify type argument during declaration of reference as well as instantiation.

```
Box<Date> b1 = new Box<Date>(); //OK
```

- If we specify type argument during reference declaration then specifying type argument during instantiation is optional. It is called as type inference.

```
Box<Date> b1 = new Box<>(); //OK
```

- We must specify type argument during declaration of reference.

```
Box<> b1 = new Box<Date>(); //Not OK
```

- We can not use inheritance for type argument.

```
Box<Object> b1 = new Box< Date>(); //Not OK
List<Date> list = new ArrayList<Date>(); //OK
```

```
Box<Date> b1 = new Box< Object>(); //Not OK
```

- If we use parameterized type without type argument then it is called as raw type.

```
Box b = new Box();      //OK: Here Box is called as raw type
//Box<Object> b1 = new Box< Object>();
```

- During instantiation of parameterized type, type argument must be non primitive type.

```
Box<int> b1 = new Box<>();    //Not OK: type argument int is not
allowed
```

- If we want to store primitive values inside instance of parameterized type then type argument must be Wrapper class.

```
Box<Integer> b1 = new Box<>();    //OK
```

## Commonly used type parameter names in Java

- T : Type
- N : Number
- K : Key
- V : Value
- E : Element
- S, U, R : Second type parameter names

## We can specify multiple type arguments

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

class HashMap<K,V> implements Map<K,V>{
    private K key;
    private V value;
    public HashMap( K key, V value ) {
        this.key = key;
        this.value = value;
    }
    @Override
    public K getKey() {
        return this.key;
    }
    @Override
    public V getValue() {
        return this.value;
    }
}
```

```

public class Program {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>(1, "DAC");
        Integer key = map.getKey();
        String value = map.getValue();
        System.out.println(key + " " + value);
    }
}

```

## Bounded Type Parameter

- If we want to put restriction on data type which is used as type argument then we should specify bounded type parameter.

```

class Box<T extends Number>{ //Now T is bounded Type parameter
    private T reference;
    public T getReference() {
        return reference;
    }
    public void setReference(T reference) {
        this.reference = reference;
    }
}
public class Program {
    public static void main(String[] args) {
        //Box<Boolean> b1 = new Box<>(); //Not OK
        //Box<Character> b2 = new Box<>(); //Not OK
        Box<Number> b3 = new Box<>(); //OK
        Box<Integer> b4 = new Box<>(); //OK
        Box<Double> b5 = new Box<>(); //OK
        //Box<String> b6 = new Box<>(); //Not OK
        //Box<Date> b7 = new Box<>(); //Not OK
    }
}

```

## ArrayList demo

```

import java.util.ArrayList;

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

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

```

```
    }  
}
```

## Wild Card

- In Generics ? is called as wild card which represents unknown type.
- Types of wild card
  - Unbounded wild card
  - Upper Bounded wild card
  - Lower Bounded wild card.
- Consider following code

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

### Unbounded wild card

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

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

```

public static void main(String[] args) {
    ArrayList<Integer> integerList = Program.getIntegerArrayList();
    Program.printRecord( integerList );

    ArrayList<Double> doubleList = Program.getDoubleArrayList();
    Program.printRecord(doubleList);

    ArrayList<String> stringList = Program.getStringArrayList();
    Program.printRecord(stringList );
}

```

#### Upper Bounded wild card

```

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

```

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

```

public static void main(String[] args) {
    ArrayList<Integer> integerList = Program.getIntegerArrayList();
    Program.printRecord( integerList ); //OK

    ArrayList<Double> doubleList = Program.getDoubleArrayList();
    Program.printRecord(doubleList); //OK

    ArrayList<String> stringList = Program.getStringArrayList();
    Program.printRecord(stringList ); //Not OK
}

```

#### Lower Bounded wild card

```

private static void printRecord(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 elements.

```
public static void main(String[] args) {  
    ArrayList<Integer> integerList = Program.getIntegerArrayList();  
    Program.printRecord( integerList ); //OK  
  
    ArrayList<Double> doubleList = Program.getDoubleArrayList();  
    //Program.printRecord(doubleList); //NOT OK  
  
    ArrayList<String> stringList = Program.getStringArrayList();  
    //Program.printRecord(stringList); //Not OK  
}
```

## Restrictions on Generics

- Reference: <https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>

# Day 16

## Synthetic Constructs in Java

- Reference: <https://www.baeldung.com/java-synthetic>

### Generic Method

- We can define generic method using java.lang.Object class.

```
public class Program {  
    public static void print( Object object ) {  
        System.out.println( object );  
    }  
    public static void main(String[] args) {  
        Program.print( true );  
        Program.print( 123 );  
        Program.print( 'A' );  
        Program.print( 1234567 );  
        Program.print( 3.142f );  
        Program.print( 123.4567d );  
        Program.print( "Good Morning!!" );  
        Program.print( new Date() );  
    }  
}
```

- Generic method using generics

```
public class Program {  
    public static <T> void print( T value ) {  
        System.out.println( value );  
    }  
    public static void main(String[] args) {  
        Program.print( true );  
        Program.print( 123 );  
        Program.print( 'A' );  
        Program.print( 1234567 );  
        Program.print( 3.142f );  
        Program.print( 123.4567d );  
        Program.print( "Good Morning!!" );  
        Program.print( new Date() );  
    }  
}
```

- We can specify bounded type parameter for method:

```

public class Program {
    public static <T extends Number> void print( T value ) {
        System.out.println( value );
    }
    public static void main(String[] args) {
        //Program.print( true );      //Not OK
        Program.print( 123 );
        //Program.print( 'A' );//Not OK
        Program.print( 1234567 );
        Program.print( 3.142f );
        Program.print( 123.4567d );
        //Program.print( "Good Morning!!" );//Not OK
        //Program.print( new Date() );//Not OK
    }
}

```

## Type Erasure

- Consider generic type without upper bound.

```

class Box<T>{
    private T data;

    public Box() {
    }
    public Box(T data) {
        this.data = data;
    }
    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
    @Override
    public String toString() {
        return this.data.toString();
    }
}

```

- Below is the code with type erasure.

```

class Box{
    private Object data;

    public Box() {
    }
    public Box(Object data) {

```

```

        this.data = data;
    }
    public Object getData() {
        return data;
    }
    public void setData(Object data) {
        this.data = data;
    }
    @Override
    public String toString() {
        return this.data.toString();
    }
}

```

- Consider generic type with upper bound( Number).

```

class Box<T extends Number>{
    private T data;

    public Box() {
    }
    public Box(T data) {
        this.data = data;
    }
    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
    @Override
    public String toString() {
        return this.data.toString();
    }
}

```

- Below is the code with type erasure.

```

class Box{
    private Number data;

    public Box() {
    }
    public Box(Number data) {
        this.data = data;
    }
    public Number getData() {
        return data;
    }
}

```

```

public void setData(Number data) {
    this.data = data;
}
@Override
public String toString() {
    return this.data.toString();
}
}

```

## Bridge method

- Consider following code:

```

class Box<T>{
    private T data;

    public Box() {
    }
    public Box(T data) {
        this.data = data;
    }
    public void setData(T data) {
        this.data = data;
    }
    @Override
    public String toString() {
        return this.data.toString();
    }
}
class Sample extends Box<Integer>{

    public Sample() {
        super();
    }
    public Sample(Integer data) {
        super(data);
    }
    /*
     //Method added by compiler to achieve dynamic method dispatch
    public void setData(Object data) {      //Bridge method
        super.setData((Integer)data);
    } */
    @Override
    public void setData(Integer data) {
        super.setData(data);
    }
}

```

```

public static void main3(String[] args) {
    Sample s = new Sample();
    Box b = s; //Upcasting
    b.setData(123); //OK
    System.out.println(b.toString()); //123
}

```

```

public static void main(String[] args) {
    Sample s = new Sample();
    Box b = s; //Upcasting
    b.setData( "Hello" ); //ClassCastException
    System.out.println(b.toString());
}

```

## Restrictions on Generics

- Cannot Instantiate Generic Types with Primitive Types

```

Stack<int> s1 = new Stack<>(); //Not OK
Stack<Integer> s1 = new Stack<>(); //OK

```

- Cannot Declare Static Fields Whose Types are Type Parameters

```

class Box<T>{
    private static T data; //Not OK
    //TODO: Getter and Setter
}

```

- Cannot Use Casts or instanceof with Parameterized Types

```

public static void printRecord( List<String> list ){
    if( list instanceof ArrayList<String> ){ //Not OK
        ArrayList<String> arrayList = ( ArrayList<String> ) list; //OK
    }
}

public static void main( String[] args ){
    ArrayList<String> arrayList = new ArrayList<>();
    list.add( "DAC" );
    list.add( "DMC" );
    list.add( "DESD" );
    Program.print( arrayList );

    LinkedList<String> linkedList = new LinkedList<>();
    list.add( "RED" );
}

```

```
    ist.add( "GREEN" );
    list.add( "BLUE" );
    Program.print( linkedList );
}
```

- Cannot Create Arrays of Parameterized Types

```
Box<Integer> box = new Box<>(); //OK
Box<Integer>[ ] arr = new Box<>[ 5 ]; //Not OK
```

- Cannot Create, Catch, or Throw Objects of Parameterized Types

```
class QueueFullException<T> extends Throwable // compile-time error
{ /* ... */ }
```

```
class MathException<T> extends Exception // compile-time error
{ /* ... */ }
```

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) { // compile-time error
        // ...
    }
}
```

- A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

Why main method is static?

- JVM is responsible for calling main method.
- Consider following scenarios if main method is non static:
  - If class is abstract.

```
abstract class Program{
    public void main( String[] args ){
        //TODO
    }
}
```

- If class is concrete but constructor is private:

```
class Program{
    private Program( ){
        //TODO
    }
    public void main( String[] args ){
        //TODO
    }
}
```

- If class is concrete and class contains only public parameterized constructor

```
class Program{
    public Program( String s1, int i1, float f1, double d1 ){
        //TODO
    }
    public void main( String[] args ){
        //TODO
    }
}
```

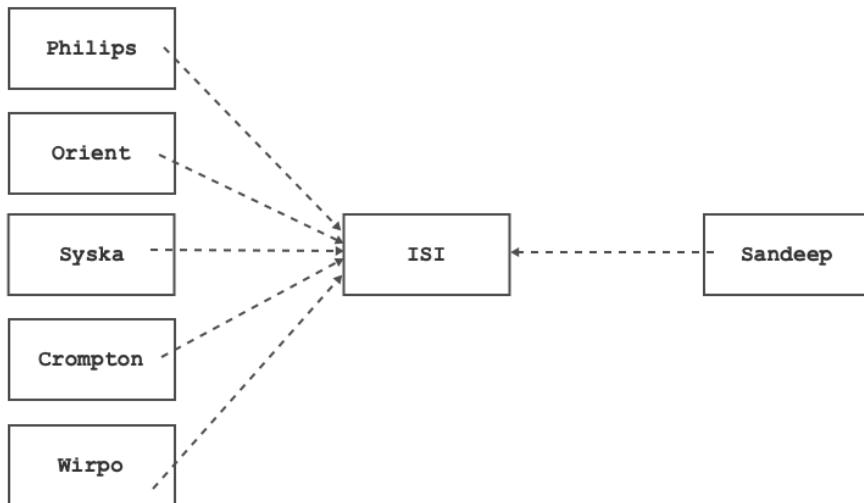
- To overcome above problems, main method is declared as static.

### Fragile Base class problem

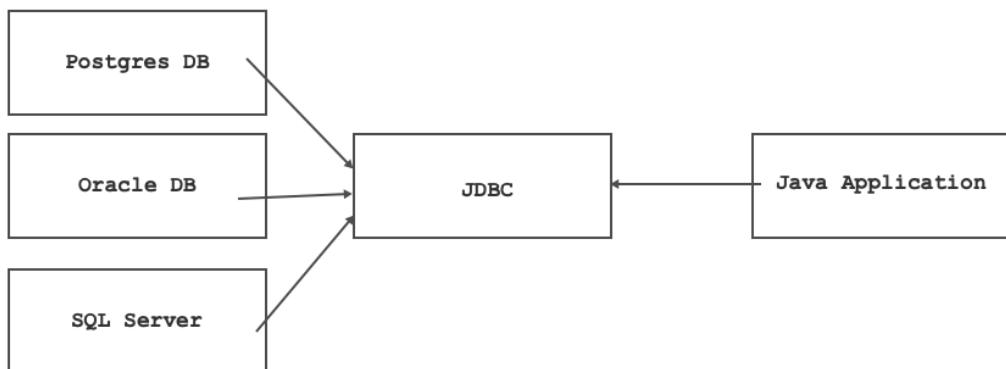
- If we make changes in the body super class then we must recompile super class as well as all its sub classes. This problem is called as fragile base class problem.
- We can solve fragile base class problem by defining super type as interface.

### Abstraction using interface

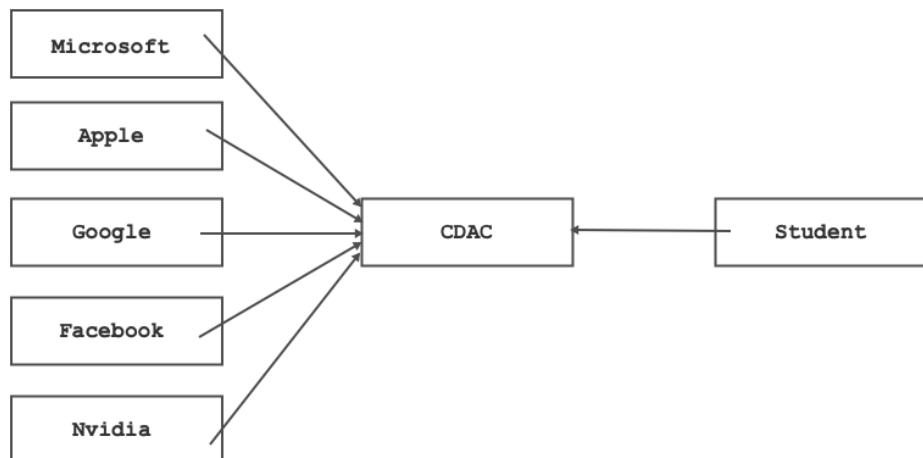
[ Service Provider ] [ Contract ] [ Service Consumer ]



[ Service Provider ] [ Contract ] [ Service Consumer ]



[ Service Consumer ] [ Contract ] [ Service Provider ]



- Set of rules are called as specification/standard.
- If we want to define specifications for sub classes in java then we should define interface.
- Interface is contract between service provider and service consumer
- Advantage of interface:
  - To build trust between service provider and service consumer.
  - It helps to achieve abstraction
  - It helps to minimize service provider dependency
- Non primitive types in Java:
  - interface
  - class
  - enum
  - array
- interface is a keyword in java
- In Java, interface can contain:
  - Nested type
  - Field
  - Abstract method
  - Default method
  - Static interface method
- interface fields are implicitly considered as public static and final

```
interface Printable{
    //int value; //Error: The blank final field value may not have been
    //initialized
    int value = 123;
    //public static final int value = 123;
}
```

- interface methods are implicitly considered as public and abstract.

```
interface Printable{
    //void print( ) { } //Error: Abstract methods do not specify a
    //body
    void print( );
    //public abstract void print( );
}
```

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

```

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

```

- We can not define constructor inside interface
- Consider following code:

```

interface Printable{      //Contract
    int value = 123;
    //public static final int value = 123;

    void print( );
    //public abstract void print( );
}

class Test implements Printable{      //Service Provider
    @Override
    public void print() {
        System.out.println("Value : "+Printable.value);
    }
}

public class Program { //Service Consumer
    public static void main(String[] args) {
        Printable p = null; //OK
        p = new Test(); //Upcasting
        p.print(); //Dynamic method dispatch
    }
}

```

## Syntax to use interface

- Interfaces: I1, I2, I3
- Classes: C1, C2, C3
  - I2 implements C1 //Not OK
    - Super type of interface must be interface
  - I2 implements I1 //Not OK
    - Interface can extend another interface
  - I2 extends I1 //OK
  - I3 extends I1, I2 //OK
    - Interface can extend more than one interface. It is called multiple interface inheritance
  - C1 extends I1;
    - Class can implement interface
  - C1 implements I1 //OK
  - C1 implements I1, I2 //OK
    - Class can implement more than one interfaces. It is called as multiple interface implementation inheritance.

- C2 implements C1 //Not OK
  - Class can extend another class
- C2 extends C1 //OK
- C3 extends C1, C2 //Not OK
  - Class can not extend more than once class. In short class do not support multiple implementation inheritance.
- C2 implements I1 extends C1 //Not OK
  - Class should first extend class and then implement interface
- C2 extends C1 implements I1 //OK
- C2 extends C1 implements I1, I2 //OK

## Interface fields syntax

```

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{ //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); //70
        System.out.println("B.num5 : "+B.num5); //80
        //System.out.println("C.num5 : "+C.num5); //Error: The field
C.num5 is ambiguous
    }
    public static void main2(String[] args) {
        System.out.println("A.Num4 : "+A.num4); //40
        System.out.println("B.Num4 : "+B.num4); //50
        System.out.println("C.Num4 : "+C.num4); //60
    }
    public static void main1(String[] args) {
        System.out.println("Num1 : "+A.num1);
        System.out.println("Num1 : "+C.num1);

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

        System.out.println("Num3 : "+C.num3);
    }
}

```

## Interface method syntax:

```
interface A{
    void f1();
}
interface B{
    void f2();
}
interface C extends A, B{
    void f3();
}
class D implements C{
    @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) {
        D d = new D();
        d.f1(); //OK

        A a = new D();
        a.f1(); //OK

        B b = new D();
        b.f2(); //OK

        C c = new D();
        c.f1(); //OK
        c.f2(); //OK
        c.f3(); //OK
    }
}
```

```
interface A{
    void f1();
    void f3();
}
interface B{
    void f2();
}
```

```

        void f3();
    }
    class C implements A, B{
        @Override
        public void f1() {
            System.out.println("C.f1");
        }
        @Override
        public void f2() {
            System.out.println("C.f2");
        }
        @Override
        public void f3() {
            System.out.println("C.f3");
        }
    }
    public class Program {
        public static void main(String[] args) {
            A a = new C();
            a.f1();
            a.f3();

            B b = new C();
            b.f2();
            b.f3();
        }
    }
}

```

## How to override some of the methods of interface

```

interface Printable{
    void f1();
    void f2();
    void f3();
    void f4();
}

abstract class AbstractPrintable implements Printable{
    @Override public void f1() {    }
    @Override public void f2() {    }
    @Override public void f3() {    }
}

class A extends AbstractPrintable{
    @Override
    public void f1() {
        System.out.println("A.f1");
    }
}

class B extends AbstractPrintable{
    @Override
    public void f2() {
        System.out.println("B.f2");
    }
}

```

```

    }
}

class C extends AbstractPrintable{
    @Override
    public void f3() {
        System.out.println("C.f3");
    }
}

public class Program {
    public static void main(String[] args) {
        Printable p = null;

        p = new A();
        p.f1(); //A.f1

        p = new B();
        p.f2(); //B.f2

        p = new C();
        p.f3(); //C.f3
    }
}

```

## Types of inheritance

- Interface inheritance
  - Single inheritance( Allowed in java )
  - Multiple inheritance( Allowed in java )
  - Hierarchical inheritance( Allowed in java )
  - Multilevel inheritance( Allowed in java )
- implementation inheritace
  - Single inheritance( Allowed in java )
  - Multiple inheritance( Not Allowed in java )
  - Hierarchical inheritance( Allowed in java )
  - Multilevel inheritance( Allowed in java )

## Default interface method

- If we want to make changes in the interface at runtime then we should use default method.
- We can not provide body to the abstract method but it is mandatory to provide body to the default method.
- It is mandatory to override abstract method but it is optional to override default method.

```

interface A{
    void f1( );
    default void f2( ){
        //TODO
    }
}

```

```
}
```

```
class B implements A{
```

```
    @override
```

```
    public void f1( ){
```

```
        //TODO
```

```
    }
```

```
}
```

```
interface A{
```

```
    void f1( );
```

```
    default void f2( ){
```

```
        //TODO
```

```
    }
```

```
}
```

```
interface B{
```

```
    void f1( );
```

```
    default void f3( ){
```

```
        //TODO
```

```
    }
```

```
}
```

```
class C implements A, B{
```

```
    @override
```

```
    public void f1( ){
```

```
        //TODO
```

```
    }
```

```
}
```

```
interface A{
```

```
    void f1( );
```

```
    default void f2( ){
```

```
        //TODO
```

```
    }
```

```
}
```

```
interface B{
```

```
    void f1( );
```

```
    default void f2( ){
```

```
        //TODO
```

```
    }
```

```
}
```

```
class C implements A, B{
```

```
    @override
```

```
    public void f1( ){
```

```
        //TODO
```

```
    }
```

```
    @Override
```

```
    public void f2( ){ //mandatory to override
```

```
        //TODO
```

```
    }
```

```
}
```

- Consider following code:

```

interface Collection {
    void acceptRecord();

    int[] toArray();

    void printRecord();

    static void swap( int[] arr ) {
        int temp = arr[ 0 ];
        arr[ 0 ] = arr[ 1 ];
        arr[ 1 ] = temp;
    }
    default void sort() {
        int[] arr = this.toArray();
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {

                    int[] temp = new int[ ] { arr[j], arr[ j + 1 ] };
                    Collection.swap(temp);
                    arr[j] = temp[ 0 ];
                    arr[ j + 1 ] = temp[ 1 ];
                }
            }
        }
    }
}

class Array implements Collection {
    private int[] arr;

    public Array() {
        this(5);
    }

    public Array(int size) {
        this.arr = new int[size];
    }

    @Override
    public void acceptRecord() {
        try (Scanner sc = new Scanner(System.in)) {
            for (int index = 0; index < this.arr.length; ++index) {
                System.out.print("Enter element :   ");
                this.arr[index] = sc.nextInt();
            }
        }
    }
}

```

```

@Override
public int[] toArray() {
    return this.arr;
}

@Override
public void sort() {
    for( int i = 0; i < this.arr.length - 1; ++ i ) {
        for( int j = i + 1; j < this.arr.length; ++ j ) {
            if( this.arr[ i ] > this.arr[ j ] ) {
                int[] temp = new int[ ] { arr[ i ], arr[ j ] };
                Collection.swap(temp);
                arr[ i ] = temp[ 0 ];
                arr[ j ] = temp[ 1 ];
            }
        }
    }
}

@Override
public void printRecord() {
    System.out.println(Arrays.toString(this.arr));
}
}

public class Program {
    public static void main(String[] args) {
        Collection c = new Array();
        c.acceptRecord();
        c.sort();
        c.printRecord();
    }
}

```

- Static interface methods are helper methods that we can use inside default method as well as inside sub class. But we can not override it inside sub class.

## Functional interface

- An interface which can contain Single Abstract Method (SAM) is called as Functional interface / SAM interface.
- Example:
  - java.lang.Runnable
  - java.util.Comparator
  - java.util.function.Predicate
  - java.util.function.Consumer
  - java.util.function.Supplier
  - java.util.function.Function
- Consider following code:

```
@FunctionalInterface  
interface A{  
    void f1();  
}
```

```
@FunctionalInterface  
interface A{  
    void f1();  
    default void f2( ){  
    }  
}
```

```
@FunctionalInterface  
interface A{  
    void f1();  
    default void f2( ){  
    }  
    static void f3( ){  
    }  
}
```

```
@FunctionalInterface  
interface A{  
    void f1();  
    default void f2( ){  
    }  
    default void f3( ){  
    }  
    static void f4( ){  
    }  
    static void f5( ){  
    }  
}
```

## Shallow copy, Deep copy

- Process of copying contents from variable into another variable as it is, is called shallow copy.

```
int num1 = 10;  
int num2 = num1; //Shallow Copy
```

```
Date dt1 = new Date( 13,4,2023);
Date dt2 = dt1; //Shallow Copy of references
```

- If we want to create new instance from existing instance then we should use clone method.
- clone is non final and native method of java.lang.Object class:
- Syntax:
  - protected native Object clone() throws CloneNotSupportedException
- Inside clone method, if we want to create shallow copy of instance then we should use super.clone();
- Without implementing Cloneable interface, if we try to create clone() of the instance then clone method throws CloneNotSupportedException.
- Marker interface:
  - An interface which do not contain any member is called as marker / tagging interface.
  - Marker interface are used to generate metadata. It helps JVM to perform some operations e.g to do clone, serializing state of java instance etc.
  - Example:
    - java.lang.Cloneable
    - java.util.EventListener
    - java.util.RandomAccess
    - java.rmi.Remote

```
class Date implements Cloneable{
    private int day;
    private int month;
    private int year;

    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public void setDay(int day) {
        this.day = day;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public void setYear(int year) {
        this.year = year;
    }
}
```

```

@Override
public Date clone( ) {
    try {
        Date other = (Date) super.clone();
        return other;
    } catch (CloneNotSupportedException e) {
        throw new InternalError(e);
    }
}

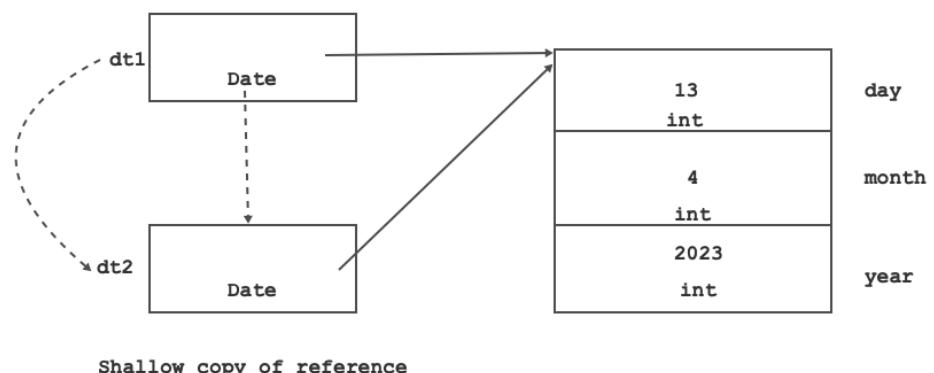
@Override
public String toString() {
    return this.day+" / "+this.month+" / "+this.year;
}
}

```

```

public class Program {
    public static void main(String[] args) {
        Date dt1 = new Date(13, 4, 2023);
        Date dt2 = dt1; //Shallow copy of references
        //System.out.println( dt1 == dt2 ); //true
    }
}

```



```

public static void main(String[] args) {
    try {
        Date dt1 = new Date(13, 4, 2023);
        Date dt2 = dt1.clone();
        dt2.setDay(23);
        dt2.setMonth(7);
        dt2.setYear(1983);

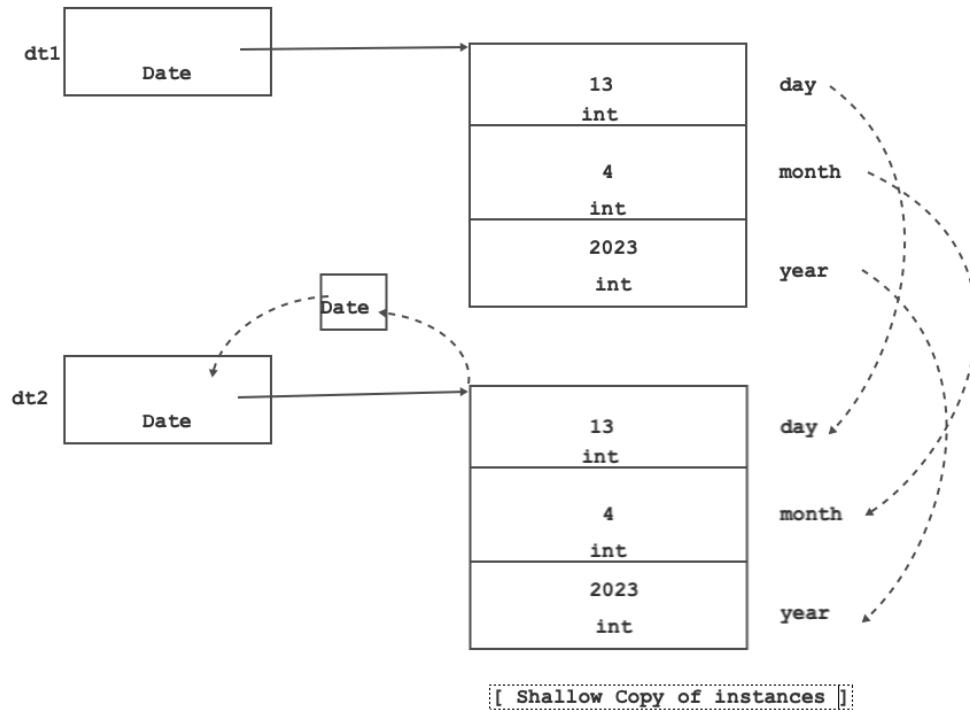
        System.out.println(dt1);
        System.out.println(dt2);
    }
}

```

```

} catch (CloneNotSupportedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}

```



[ Shallow Copy of instances ]

- Consider example of ArrayList:
- public class ArrayList extends AbstractList implements List, RandomAccess, Cloneable, Serializable

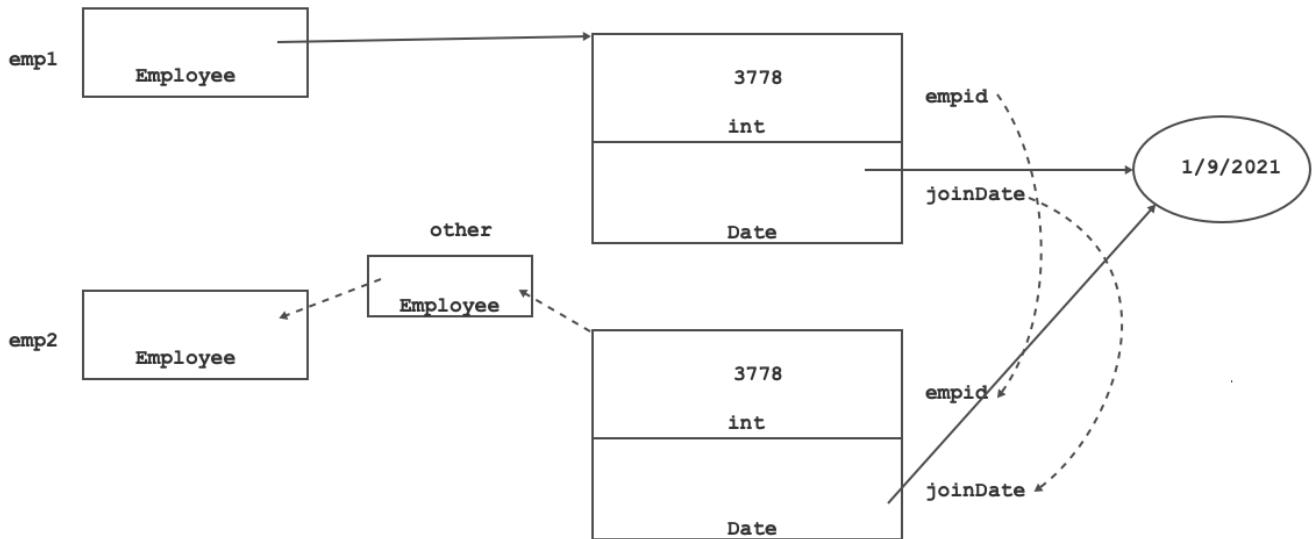
```

public static void main(String[] args) {
    ArrayList<Integer> list1 = new ArrayList<>();
    list1.add(10);
    list1.add(20);
    list1.add(30);

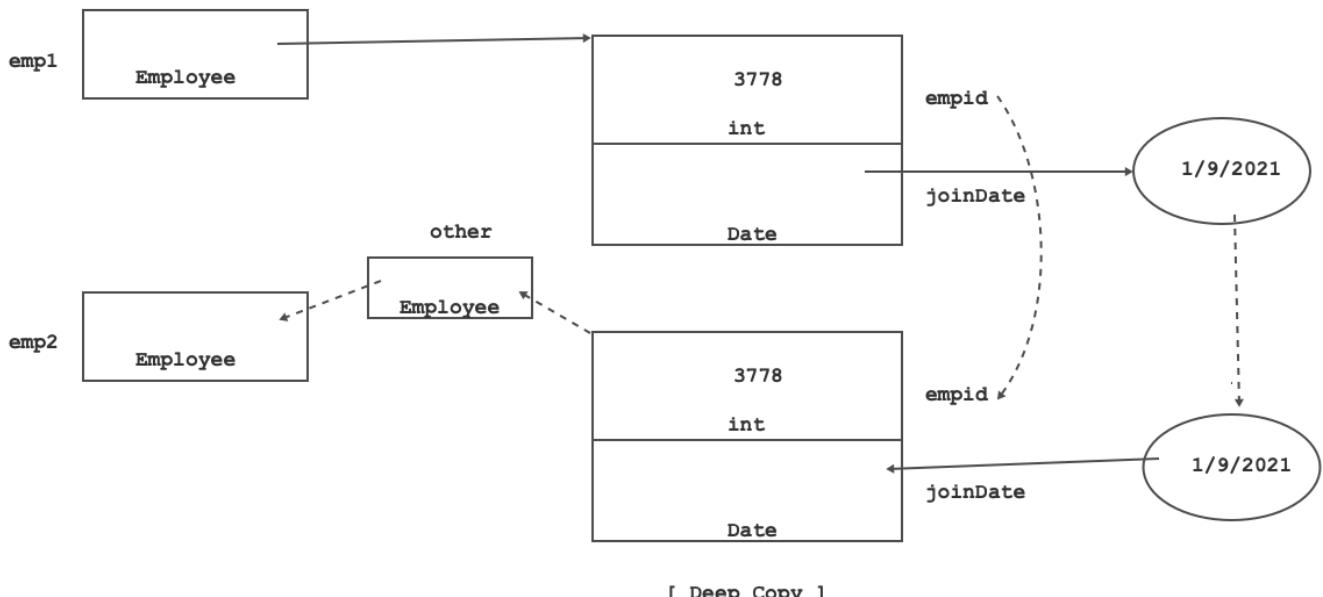
    ArrayList<Integer> list2 = (ArrayList<Integer>)list1.clone();
    list1.clear();
    System.out.println(list1);
    System.out.println(list2);
}

```

## Shallow Copy



## Deep Copy

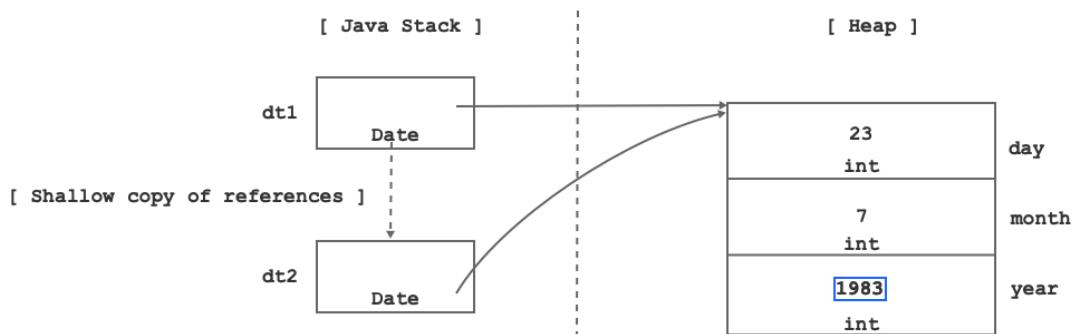


Write a program to generate Linear singly linked list in java

- Operations:
  - `public boolean empty()`
  - `public void addLast( int element )`
  - `public void removeFirst( )`
  - `public void printList( )`

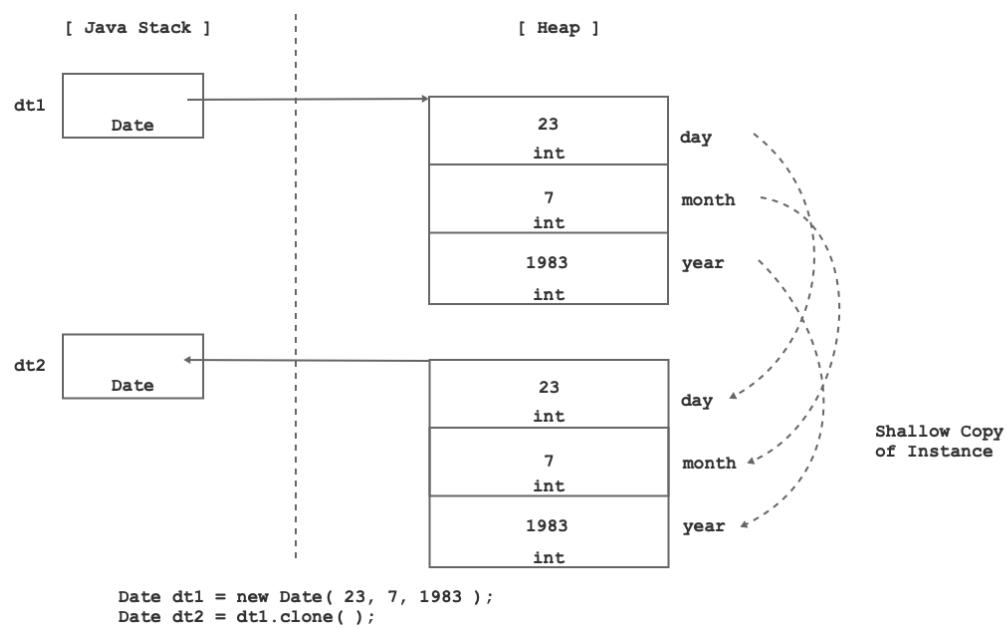
# Day 17

- Shallow Copy of Date reference:



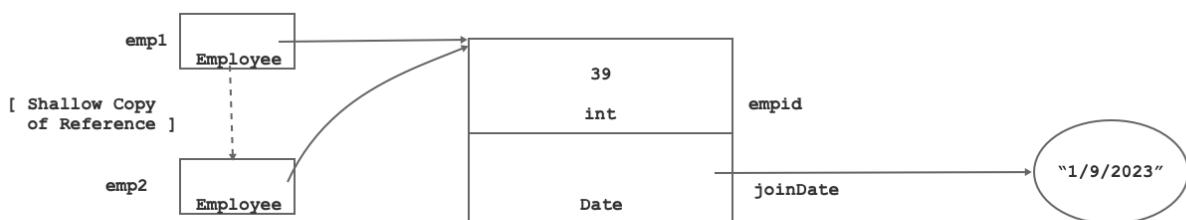
```
Date dt1 = new Date( 23, 7, 1983 );
Date dt2 = dt1; //Shallow Copy of references
```

- Shallow copy of Date instance:



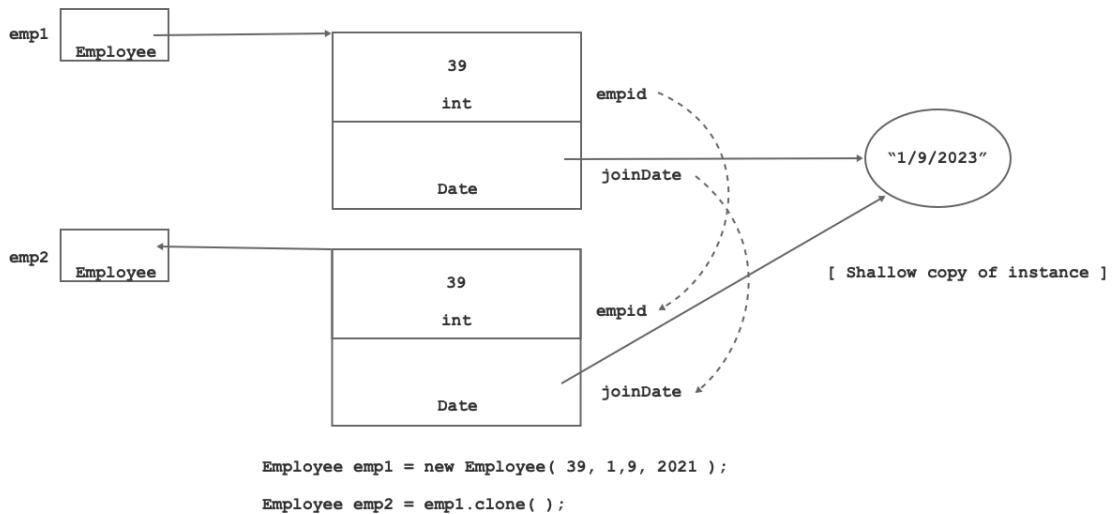
```
Date dt1 = new Date( 23, 7, 1983 );
Date dt2 = dt1.clone();
```

- Shallow Copy of Employee reference:

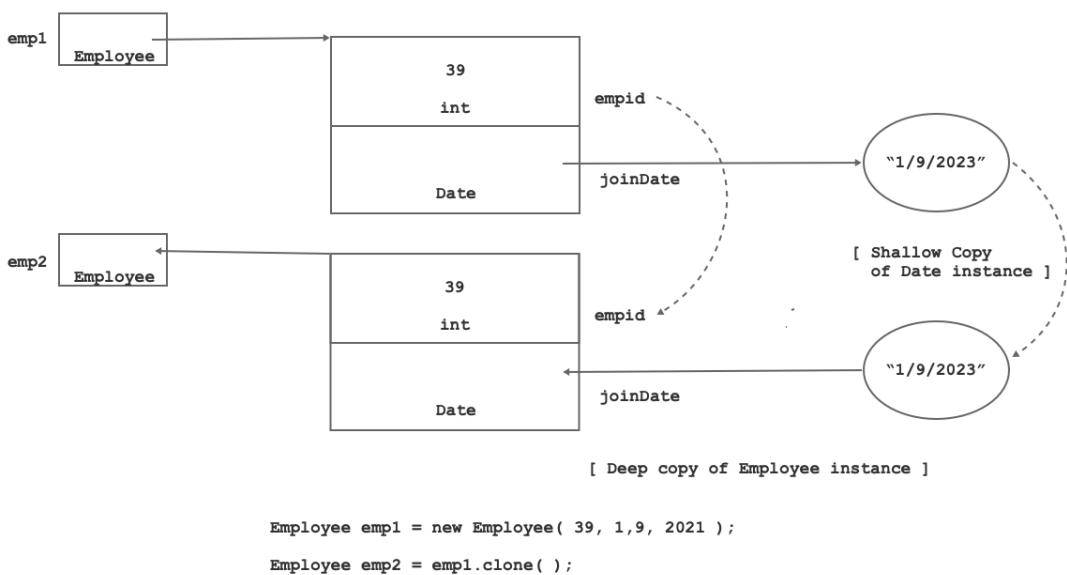


```
Employee emp1 = new Employee( 39, 1, 9, 2021 );
Employee emp2 = emp1; //Shallow copy of references
```

- Shallow Copy of Employee instance:



- Deep Copy of Employee instance:



## Fundamental interfaces of Core Java:

- java.lang.AutoCloseable
- java.io.Closeable
- java.lang.Cloneable
- java.lang.Comparable
- java.util.Comparator
- java.lang.Iterable
- java.util.Iterator

Comparable interface implementation

- Consider data for sorting

| EMPNO | ENAME  | JOB       | MGR  | HIREDATE   | SAL     | COMM    | DEPTNO |
|-------|--------|-----------|------|------------|---------|---------|--------|
| 7369  | SMITH  | CLERK     | 7902 | 1980-12-17 | 800.00  | -       | 20     |
| 7499  | ALLEN  | SALESMAN  | 7698 | 1981-02-20 | 1600.00 | 300.00  | 30     |
| 7521  | WARD   | SALESMAN  | 7698 | 1981-02-22 | 1250.00 | 500.00  | 30     |
| 7566  | JONES  | MANAGER   | 7839 | 1981-04-02 | 2975.00 | -       | 20     |
| 7654  | MARTIN | SALESMAN  | 7698 | 1981-09-28 | 1250.00 | 1400.00 | 30     |
| 7698  | BLAKE  | MANAGER   | 7839 | 1981-05-01 | 2850.00 | -       | 30     |
| 7782  | CLARK  | MANAGER   | 7839 | 1981-06-09 | 2450.00 | -       | 10     |
| 7788  | SCOTT  | ANALYST   | 7566 | 1982-12-09 | 3000.00 | -       | 20     |
| 7839  | KING   | PRESIDENT | -    | 1981-11-17 | 5000.00 | -       | 10     |
| 7844  | TURNER | SALESMAN  | 7698 | 1981-09-08 | 1500.00 | 0.00    | 30     |
| 7876  | ADAMS  | CLERK     | 7788 | 1983-01-12 | 1100.00 | -       | 20     |
| 7900  | JAMES  | CLERK     | 7698 | 1981-12-03 | 950.00  | -       | 30     |
| 7902  | FORD   | ANALYST   | 7566 | 1981-12-03 | 3000.00 | -       | 20     |
| 7934  | MILLER | CLERK     | 7782 | 1982-01-23 | 1300.00 | -       | 10     |

- If we want to sort array/collection of non primitive type using Arrays.sort() method then non primitive type must implement Comparable interface.
- Comparable is interface declared in java.lang package.
  - T -> the type of objects that this object may be compared to
- Method:
  - int compareTo(T other)
    - Returns a negative integer(-1) : current/calling object is less than the specified object.
    - Returns zero(0): current/calling object is equal to the specified object.
    - Returns a positive integer(1) : current/calling object is greater than the specified object.
- Comparable interface provides natural ordering( default ordering ) of elements inside same class.
- This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

```

@NoArgsConstructor
@AllArgsConstructor
@Getter @Setter
//@ToString
  
```

```

public class Employee implements Comparable<Employee>{
    private int empNumber;
    private String empName;
    private String job;
    private int manager;
    private LocalDate hireDate;
    private float salary;
    private float commision;
    private int deptNumber;

    @Override
    public int compareTo(Employee other) {
        return this.empNumber - other.empNumber;
    }
    @Override
    public String toString() {
        return String.format("%-5d%-10s%-10s%-5d%-15s%-10.2f%-10.2f%-5d",
this.empNumber, this.empName, this.job, this.manager, this.hireDate,
this.salary, this.commision, this.deptNumber);
    }
}

```

## Comparator interface implementation

- If we wan to sort array/collection of non primitive type using different criteria then we should implement Comparator interface.
- Comparator is functional interface declared in java.util package.
  - T -> the type of objects that may be compared by this comparator
- Method:
  - int compare(T o1, T o2)
    - Returns a negative integer(-1) : If first argument is less than the second argument.
    - Returns zero(0): If first argument is equal to the second argument.
    - Returns a positive integer(1) : If first argument is greater than the second argument.
- Comparator interface provides custom ordering of elements in different classes.

```

import java.util.Comparator;

public class DeptNumberComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.getDeptNumber() - e2.getDeptNumber();
    }
}

```

```

import java.util.Comparator;

public class HireDateComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.getHireDate().compareTo(e2.getHireDate());
    }
}

```

```

import java.util.Comparator;
public class IdComparator implements Comparator<Person>{
    @Override
    public int compare(Person p1, Person p2) {
        if( p1 instanceof Student && p2 instanceof Student ) {
            Student s1 = (Student) p1;
            Student s2 = (Student) p2;
            return s1.getRollNumber() - s2.getRollNumber();
        }else if( p1 instanceof Employee && p2 instanceof Employee ) {
            Employee e1 = (Employee) p1;
            Employee e2 = (Employee) p2;
            return e1.getEmpid() - e2.getEmpid();
        }else if( p1 instanceof Student && p2 instanceof Employee ) {
            Student s1 = (Student) p1;
            Employee e2 = (Employee) p2;
            return s1.getRollNumber() - e2.getEmpid();
        }else {
            Employee e1 = (Employee) p1;
            Student s2 = (Student) p2;
            return e1.getEmpid() - s2.getRollNumber();
        }
    }
}

```

## Iterable & Iterator

- Traversing using iterator

```

import java.util.Iterator;
import java.util.LinkedList;

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

        Integer element = null;
    }
}

```

```

Iterator<Integer> itr = list.iterator();
while( itr.hasNext() ) {
    element = itr.next();
    System.out.print(element+"    ");
}
System.out.println();
}
}

```

- Traversing using foreach loop

```

public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<>();
    list.addLast(10);
    list.addLast(20);
    list.addLast(30);

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

```

- We can use foreach loop on Array and any instance which implements java.lang.Iterable interface.
- Iterable is interface declared in java.lang package.
  - T -> the type of elements returned by the iterator
- Implementing this interface allows an object to be the target of the "for-each loop" statement.
- It is introduced in JDK 1.5
- Methods:
  - java.util.Iterator iterator()
  - default Spliterator spliterator()
  - default void forEach(Consumer<? super T> action)
- Iterator is interface declared in java.util package.
  - E -> the type of elements returned by this iterator
- It is introduced in JDK 1.2
- Methods:
  - boolean hasNext()
  - E next()
  - default void remove()
  - default void forEachRemaining(Consumer<? super E> action)

```
import java.util.Iterator;
class Node{
    int data;
    Node next = null;
    public Node( int data ) {
        this.data = data;
    }
}
```

```
class LinkedList implements Iterable<Integer>{
    private Node head = null;
    private Node tail = null;

    public boolean empty( ) {
        return this.head == null;
    }
    public void addLast( int element ) {
        Node newNode = new Node( element );
        if( this.empty() )
            this.head = newNode;
        else
            this.tail.next = newNode;
        this.tail = newNode;
    }
    @Override
    public Iterator<Integer> iterator( ) {
        Iterator<Integer> itr = new LinkedListIterator( this.head );
        //Upcasting
        return itr;
    }
}
```

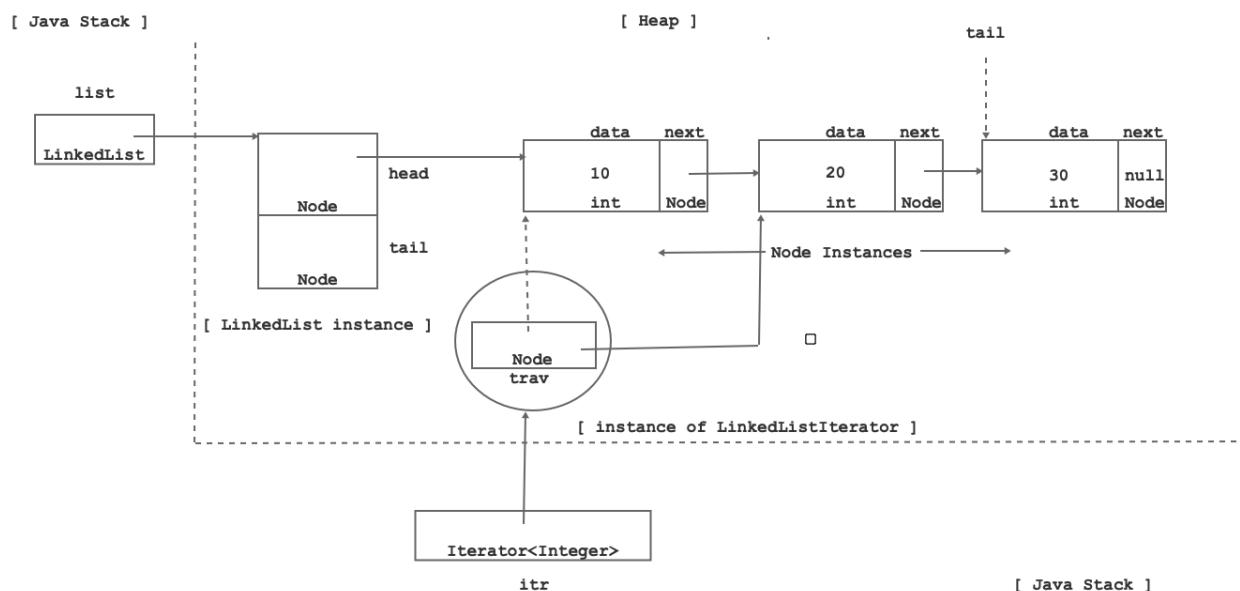
```
class LinkedListIterator implements Iterator<Integer>{
    private Node trav;

    public LinkedListIterator(Node head) {
        this.trav = head;
    }
    @Override
    public boolean hasNext( ) {
        return this.trav != null;
    }
    @Override
    public Integer next( ) {
        int data = trav.data;
        trav = trav.next;
        return data;
    }
}
```

```
}
```

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

        for( int element : list )
            System.out.println( element );
    }
}
```



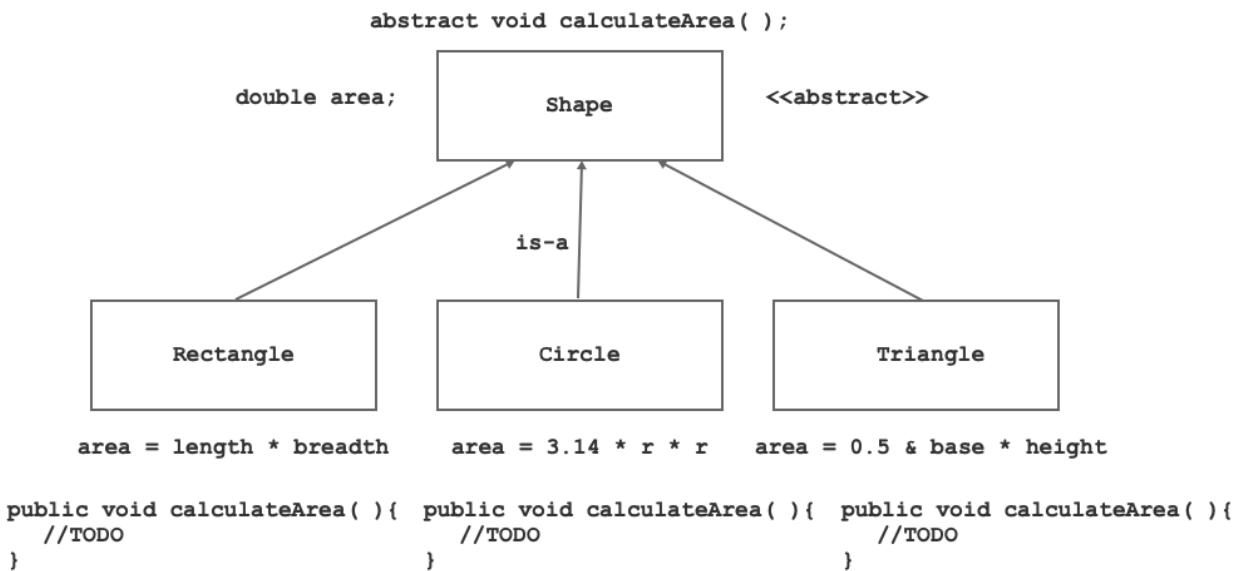
- Process of visiting elements in a collection is called as traversing.
- In C++ if we use any object as a pointer then such object is called as smart pointer.
- Iterator is a smart pointer which is used to traverse collection.
- Any class which implements Iterable interface is considered as traversable using foreach loop.

## Abstract class versus Interface

### Abstract class

- If "is-a" relationship exists between super type & sub type and if we want to use same method design/signature in all the sub classes then super type should be abstract class.

- Consider following diagram



- Using abstract class we can group elements/instances of related types together.

```

Shape[] arr = new Shape[ 3 ];
arr[ 0 ] = new Rectangle(); //Upcasting
arr[ 1 ] = new Circle();  //Upcasting
arr[ 2 ] = new Triangle(); //Upcasting

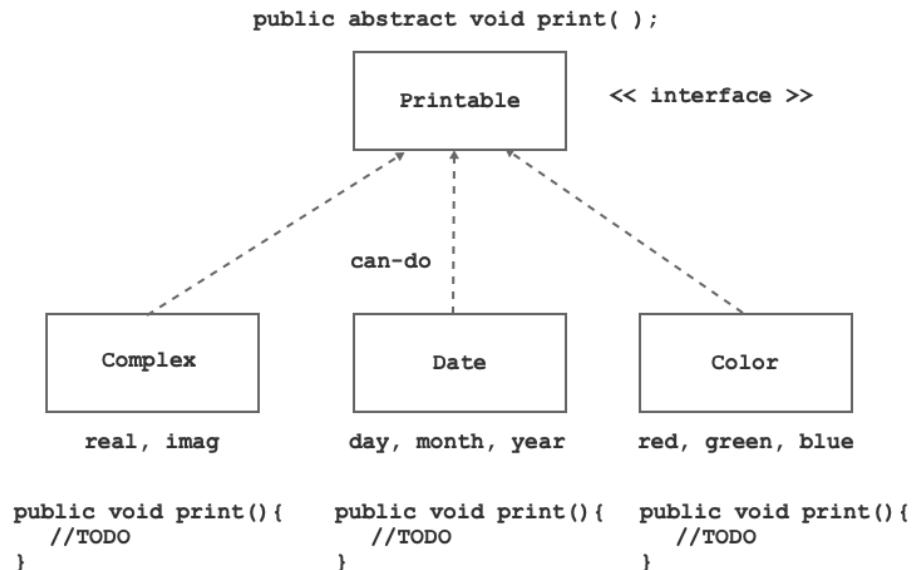
```

- Super class of abstract class can be either concrete class / abstract class. Abstract class can extend only one abstract class / concrete class.
- We can define constructor inside abstract class.
- Abstract may/may not contain abstract method.
- When state is involved in super type then it should be abstract class

## Interface

- If "is-a" relationship is not exist between super type & sub type( "can-do" relationship is exist ) and if we want to use same method design in all the sub classes then super type should be interface.

- Consider following diagram



- Using interface, we can group elements/instances of unrelated type together.

```

Printable[] arr = new Printable[ 3 ];
arr[ 0 ] = new Complex( ); //Upcasting
arr[ 1 ] = new Date( ); //Upcasting
arr[ 2 ] = new Color( ); //Upcasting

```

- Super type of interface must be interface. Interface can extend multiple interfaces.
- We can not define constructor inside interface.
- Interface methods are by default public and abstract.
- When state is not involved in super type then it should be interface.

## Nested class

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

```

//Top level class
class Outer{ //Outer.class
    //Nested class
    class Inner{ //Outer$Inner.class
        //TODO: Implementation part
    }
}

```

- Access modifier of top level class can be either package level private or public only but we can use any access modifier on nested class.
- Types of nested class
  - Non static nested class / Inner class

- Static nested class

## Inner class

- In Java, non static nested class is called as inner class.
- If implementation of nested class depends on top level class then we should declare nested class non static.

```
class LinkedList implements Iterable<Integer>{

    private Node head = null;
    private Node tail = null;

    //TODO: other implementation

    @Override
    public Iterator<Integer> iterator() {
        Iterator<Integer> itr = new LinkedListIterator( this.head );
        //Upcasting
        return itr;
    }

    class LinkedListIterator implements Iterator<Integer>{
        private Node trav;
        public LinkedListIterator(Node head) {
            this.trav = head;
        }
        //TODO: Other implementation
    }
}
```

- Note: For the simplicity, consider non static nested class a non static method of the class.
- Consider following code:

```
class Outer{
    class Inner{
        //TODO
    }
}
```

- Instantiation of top level class:

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

- Instantiation of inner class:

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

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

- Inside non static nested class i.e inner class, we can not define static members( fields & methods ).  
But if we want to declare any static field then it must be final.
- Using instance, we can access members of non static nested class inside method of top level class.
- Consider following example:

```
class Outer{
    private int num1 = 10;      //OK
    private static int num2 = 20; //OK

    class Inner{
        private int num3 = 30;  //OK
        //private static int num4 = 40; //Not OK
        private final static int num4 = 40; //OK
    }

    public void print( ) {
        System.out.println("Num1      : "+this.num1);
        System.out.println("Num2      : "+Outer.num2);
        Inner in = new Inner();
        System.out.println("Num3      : "+in.num3);
        System.out.println("Num4      : "+Inner.num4);
    }
}

public class Program {
    public static void main(String[] args) {
        Outer out = new Outer();
        out.print();
    }
}
```

- Without instance, we can access all the members of top level class inside method of non static nested class i.e inner class.
- Consider following code:

```
class Outer{
    private int num1 = 10;
    private static int num2 = 20;
```

```

class Inner{
    private int num3 = 30;
    private final static int num4 = 40;
    public void print( ) {
        System.out.println("Num1 : "+num1); //OK
        System.out.println("Num2 : "+num2); //OK
        System.out.println("Num3 : "+this.num3);
        System.out.println("Num4 : "+Inner.num4);
    }
}
}

public class Program {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.print();
    }
}

```

- Consider following code snippet

```

class Outer{
    private int num1 = 10;
    class Inner{
        private int num1 = 20;
        public void print( ) {
            int num1 = 30;
            System.out.println("Num1 : "+Outer.this.num1); //10
            System.out.println("Num1 : "+this.num1); //20
            System.out.println("Num1 : "+num1); //30
        }
    }
}

public class Program {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.print();
    }
}

```

## Static Nested Class

- If we declare nested class as a static then it is called as static nested class.
- If implementation of nested class do not depend on top level class then we should declare nested class static.
- Consider following code:

```

class LinkedList implements Iterable<Integer>{
    static class Node{
        int data;
        Node next = null;
        public Node( int data ) {
            this.data = data;
        }
    }

    private Node head = null;
    private Node tail = null;
}

```

- In Java, we can not declare top level class static but we can declare nested class static.

```

static class Outer{ //Not OK
    //TODO
}

```

```

//Top Level class
class Outer{ //Outer.class
    //Static Nested class
    static class Inner{ //OK: Outer$Inner.class
        //TODO
    }
}

```

- Note: For simplicity, consider static nested class as a static method of the class.
- Instantiation of Top level class:

```

Outer out = new Outer();

```

- Instantiation of static nested class:

```

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

```

- We can declare/define static members inside static nested class.
- Using instance, we can access all the members of top level class inside method of top level class.

```

class Outer{
    private int num1 = 10;
    private static int num2 = 20;

    static class Inner{
        private int num3 = 30; //OK
        private static int num4 = 40; //OK
    }
    public void print( ) {
        System.out.println("Num1 : "+this.num1);
        System.out.println("Num2 : "+Outer.num2);
        Inner in = new Inner();
        System.out.println("Num3 : "+in.num3);
        System.out.println("Num4 : "+Inner.num4);
    }
}
public class Program {
    public static void main(String[] args) {
        Outer out = new Outer();
        out.print();
    }
}

```

- We can access static members of the top level class inside method of static nested class directly. But to access non static member of the class we must use instance of the class.

```

class Outer{
    private int num1 = 10;
    private static int num2 = 20;

    static class Inner{
        private int num3 = 30; //OK
        private static int num4 = 40; //OK

        public void print( ) {
            //System.out.println("Num1 : "+num1); //Not OK
            Outer out = new Outer();
            System.out.println("Num1 : "+out.num1); //OK
            System.out.println("Num2 : "+num2); //OK
            System.out.println("Num3 : "+this.num3);
            System.out.println("Num4 : "+Inner.num4);
        }
    }
}
public class Program {
    public static void main(String[] args) {
        Outer.Inner in = new Outer.Inner();
        in.print();
    }
}

```

## Local Class

- We can define class inside scope of another method. It is called as local class.
- Local class is also called as method local class.
- we can not use reference / instance of local class outside method.
- Types of local class:
  - Method local inner class.
  - Method local anonymous inner class.

### Method local inner class.

- In Java, we can not declare local variable as well as local class static.
- Since method local class is non static, it is also called as method local inner class.

```
public class Program {  
    public static void main(String[] args) {  
        //Method local inner class  
        class Complex{ //Program$1Complex.class  
            private int real = 10;  
            private int imag = 20;  
            public Complex() {  
                this(0,0);  
            }  
            public Complex(int real, int imag) {  
                this.real = real;  
                this.imag = imag;  
            }  
            @Override  
            public String toString() {  
                return this.real+" "+this.imag;  
            }  
        }  
        Complex c1 = new Complex();  
        System.out.println(c1.toString());  
    }  
}
```

### Method local anonymous inner class

- In Java, we can define class without name. It is called as anonymous class.
- We can define anonymous class inside method only. Hence it is called as anonymous inner class.
- If we want to define anonymous inner class then we must use new operator.

```
public class Program {  
    public static void main(String[] args) {  
        //Object obj; //obj is reference of java.lang.Object class
```

```

//new Object( );      //Anonymous instance of java.lang.Object class
//Object obj = new Object();    //Instance with object reference

Object obj = new Object( ) {      //Program$1.class
    private String message = "Hello World";
    @Override
    public String toString() {
        return this.message;
    }
};

String str = obj.toString();
System.out.println( str );
}
}

```

```

abstract class Shape{
    protected double area;
    public double getArea() {
        return area;
    }
    public abstract void calculateArea();
}

public class Program {
    public static void main(String[] args) {
        Shape sh = new Shape() {
            private double radius = 10.5;
            @Override
            public void calculateArea() {
                this.area = Math.PI * Math.pow(radius, 2);
            }
        };
        sh.calculateArea();
        System.out.println("Area : "+sh.getArea());
    }
}

```

```

interface Printable{
    void print( );
}

public class Program {
    public static void main(String[] args) {
        Printable p = new Printable() {
            @Override
            public void print() {
                System.out.println("Hello World!!");
            }
        };
    }
}

```

```
    p.print();  
}  
}
```

Reflection

Annotation

# Day 18

## What is Metadata?

- Data about data is called as metadata. In other words, metadata refers to data that provides information about other data.
- For example, metadata for a photo might include:
  - The date and time it was taken
  - The location where it was taken
  - The camera used to take it
  - The resolution of the image.
- In the case of digital media, metadata can include information about:
  - the artist
  - title
  - genre and album of a particular song or video.
- In a file management application, metadata such as:
  - file name
  - size
  - creation date and modification date are commonly used to organize and search for files.

## Metadata of Interface

- What is the name of interface?
- In which package it is declared?
- Which is the access modifier of interface?
- Which annotations are used on interface?
- Which are the super interfaces of interface?
- Which are the members declared inside interface?

## Metadata of Class

- What is the name of class?
- In which package it is declared?
- Which are the modifiers used on class?
- Which are the annotations used on class?
- Which is the super class of class?
- Which are the super interfaces of the class?
- Which are the members declared inside class?

## Metadata of the Field

- What is the name of the field?
- Which is the type of field?
- Which are the modifiers of field?
- Which annotations has been used on the field?
- Whether field is declared or inherited field?

## Metadata of the method

- What is the name of method?
- Which are the modifiers used with method?
- Which is the return type of method?
- Which are the parameters of the methods?
- Which exceptions method throws?
- Which annotations has been used on the method.
- Whether method is declared or inherited method?

## Application of Metadata

- Metadata removes the need for native C/C++ header and library files when compiling.
- Integrated Development Environment(IDE) uses metadata to help us write code. Its IntelliSense feature parses metadata to tell us what fields and methods a type offers and in the case of a method, what parameters the method expects.
- Metadata allows an object's fields to be serialized into a memory block, sent to another machine, and then deserialized, re-creating the object's state on the remote machine.
- The Garbage Collector uses metadata to keep track of each object's lifecycle, from creation to destruction.

## Reflection

- Reflection in Java is a feature that allows a program to examine or modify the behavior of a class, method, or object at runtime.
- It is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language.
- reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.
- using reflection, we can:
  - Obtain information about the class at runtime, such as its name, superclass, implemented interfaces, constructors, methods, and fields.
  - Create new objects of a class dynamically, without knowing the class name at compile time.
  - Access and modify the values of fields in an object, even if they are declared as private.
  - Invoke methods on an object dynamically, without knowing the method names at compile time.

## How to use reflection in Java?

- Consider following code:

```
class Date{
    //TODO: Member definition
}
class Address{
    //TODO: Member definition
}
class Person{
    //TODO: Member definition
}
```

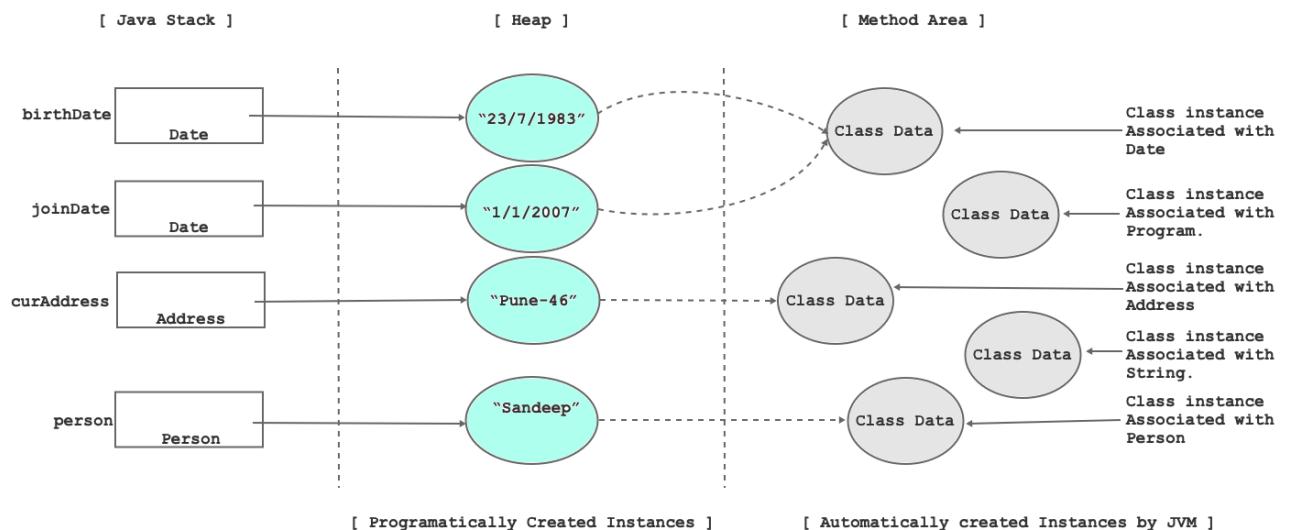
```

}

class Program{
    public static void main(String[] args) {
        Date birthDate = new Date( 23, 7, 1983 );
        Date joinDate = new Date( 1, 1, 2007 );
        Address curAddress = new Address( "Pune-46" );
        Person person = new Person("Sandeep", currentAddress, birthDate );
    }
}

```

- When class loader loads Program, String, Date, Address, Person class for execution then it creates instances of java.lang.Class per loaded type on Method area. Instance contains metadata of the loaded type.



## java.lang.Class class

- Class class is a final class declared in `java.lang` package.
- The entry point for all reflection operations is `java.lang.Class`.
- Instances of the class `java.lang.Class` represent classes and interfaces in a running Java application.
- Class has no public constructor. Instead Class objects are constructed automatically by the Java Virtual Machine.
- Methods of `java.lang.Class`:
  - `public static Class<?> forName(String className) throws ClassNotFoundException`
  - `public Annotation[] getAnnotations()`
  - `public Annotation[] getDeclaredAnnotations()`
  - `public ClassLoader getClassLoader()`
  - `public Constructor<?>[] getConstructors() throws SecurityException`
  - `public Constructor getConstructor(Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException`
  - `public Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException`
  - `public Field[] getDeclaredFields() throws SecurityException`
  - `public Field getField(String name) throws NoSuchFieldException, SecurityException`
  - `public Field[] getFields() throws SecurityException`
  - `public Class<?>[] getInterfaces()`
  - `public Method[] getMethods() throws SecurityException`

- public Method[] getDeclaredMethods() throws SecurityException
- public String getName()
- public String getSimpleName()
- public Package getPackage()
- public InputStream getResourceAsStream(String name)
- public String getTypeName()
- public T newInstance() throws InstantiationException, IllegalAccessException

## Retrieving Class Objects

- Using getClass() method:

```
Integer i = new Integer( 123 );
Class<?> c = i.getClass( ); //OK
```

- getClass is final method of java.lang.Object class:

```
public final native Class<?> getClass();
```

- Using .class Syntax

```
int i = 123;
Class<?> c = i.getClass( ); //Not OK
Class<?> c = i.class; //OK
Class<?> c = Number.class; //OK
```

- It is convenient to use .class syntax with primitive type and abstract class.

- Using Class.forName() method

- If the fully-qualified name of a class is available, it is possible to get the corresponding Class using the static method Class.forName().

```
Class<?> c = Class.forName("org.example.main.Program"); //OK

Class<?> c = Class.forName("[D"); //OK: same as double[].class

Class<?> c = Class.forName("[Ljava.lang.String"); //OK: same as
String[][] .class
```

- Using TYPE Field for Primitive Type

- Each of the primitive types and void has a wrapper class in java.lang that is used for boxing of primitive types to reference types. Each wrapper class contains a field named TYPE which is

equal to the Class for the primitive type being wrapped.

- The .class syntax is a more convenient and the preferred way to obtain the Class for a primitive type; however there is another way to acquire the Class.

```
Class<?> c = Double.TYPE;
Class<?> c = Void.TYPE; //OK
Class<?> c = Void.class; //OK
```

### Examine Type Metadata

```
import java.lang.reflect.Modifier;

public class Program {
    public static void main(String[] args) {
        Class<?> c = Integer.class;

        String typeName = c.getName(); //Returns the name of the entity
        represented by this Class object.

        String simpleTypeName = c.getSimpleName(); //Returns the simple
        name of the underlying class.

        Package pkg = c.getPackage();
        String packageName = pkg.getName(); //Return the name of this
        package.

        int mod= c.getModifiers();
        String modifiers = Modifier.toString(mod); //Returns string
        representation of the set of modifiers.

        Class<?> sc = c.getSuperclass();
        String superClassname = sc.getName(); //Returns the name of the
        super class

        Class<?>[] si = c.getInterfaces();
        StringBuffer sb = new StringBuffer();
        for (Class<?> i : si) {
            sb.append(i.getName()); //Returns the name of the super
        interfaces
        }
    }
}
```

### Examine Field Metadata

```
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
```

```

public class Program {
    public static void main(String[] args) {
        Class<?> c = Integer.class;
        Field[] fields = c.getFields(); //Returns an array of public
fields that are declared in the class or its superclasses
        Field[] declaredFields = c.getDeclaredFields(); //Returns an array
of all the fields declared in the class
        for (Field field : declaredFields) {
            String modifiers = Modifier.toString(field.getModifiers());
            String typeName = field.getType().getSimpleName();
            String fieldName = field.getName();
            System.out.println( modifiers+" "+typeName+" "+fieldName);
        }
    }
}

```

- `getFields()` returns an array of public fields that are declared in the class or its superclasses. This includes fields inherited from the superclass or any interface that the class implements. This method does not return any private or protected fields, regardless of whether they are inherited or declared in the class.
- `getDeclaredFields()` returns an array of all the fields declared in the class. This includes public, private, and protected fields. It does not include any fields inherited from a superclass or an interface.

#### Examine Method Metadata

```

import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Parameter;

public class Program {
    public static void main(String[] args) {
        Class<?> c = Integer.class;
        Method[] methods = c.getMethods();
        Method[] declaredMethods = c.getDeclaredMethods();
        for (Method method : declaredMethods) {
            String modifiers = Modifier.toString( method.getModifiers());
            String returnType = method.getReturnType().getSimpleName();
            String methodName = method.getName();

            StringBuffer paramList = new StringBuffer("(");
            Parameter[] parameters = method.getParameters();
            for( Parameter parameter : parameters ) {
                String type = parameter.getType().getSimpleName();
                paramList.append(type);
                paramList.append(" ");
                String name = parameter.getName();
                paramList.append(name);
                paramList.append(", ");
            }
        }
    }
}

```

```

        if( parameters.length > 0 )
            paramList.deleteCharAt( paramList.length() - 2 );
        paramList.append(" )");

        StringBuffer exceptionList = new StringBuffer( );
        Class<?>[] exceptionTypes= method.getExceptionTypes();
        if( exceptionTypes.length > 0 )
            exceptionList.append("throws ");
        for( Class<?> exceptionType : exceptionTypes) {
            exceptionList.append( exceptionType.getSimpleName() );
            exceptionList.append( ", " );
        }
        if( exceptionTypes.length > 0 )
            exceptionList.deleteCharAt( exceptionList.length() - 2 );

        System.out.println( modifiers+" "+ returnType+" "+methodName
+"""+paramList+"""+exceptionList);
    }
}
}

```

- `getMethods()`: This method returns an array of `Method` objects that represent all the public methods of the class (including inherited methods) and the public methods declared in any interfaces implemented by the class.
- `getDeclaredMethods()`: This method returns an array of `Method` objects that represent all the methods declared explicitly by the class, including both public and non-public methods. It does not include any inherited methods or the methods declared in any interfaces implemented by the class.

#### Accessing private fields using Reflection

```

import java.lang.reflect.Field;

class Complex{
    private int real;
    private int imag;
    public Complex() {
        this.real = 10;
        this.imag = 20;
    }
    public int getReal() {
        return this.real;
    }
    public int getImag() {
        return this.imag;
    }
}
public class Program {
    public static void main(String[] args) {
        try {

```

```

Complex complex = new Complex();
System.out.println("Real Number : "+complex.getReal());
System.out.println("Imag Number : "+complex.getImag());

Class<?> c = complex.getClass();
Field field = null;

field = c.getDeclaredField("real");
field.setAccessible(true);
field.setInt(complex, 50);

field = c.getDeclaredField("imag");
field.setAccessible(true);
field.setInt(complex, 60);

System.out.println("Real Number : "+complex.getReal());
System.out.println("Imag Number : "+complex.getImag());
} catch (NoSuchFieldException | SecurityException |
IllegalArgumentException | IllegalAccessException e) {
    e.printStackTrace();
}
}
}
}

```

### Reflection to access and invoke a private constructor of a class.

```

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

class Complex {
    private int real;
    private int imag;

    private Complex(int real, int imag) {
        this.real = real;
        this.imag = imag;
    }

    public int getReal() {
        return this.real;
    }

    public int getImag() {
        return this.imag;
    }
}

public class Program {
    public static void main(String[] args) {
        try {
            Class<?> c = Complex.class;

```

```

        Constructor<?> constructor =
c.getDeclaredConstructor(int.class, int.class);
constructor.setAccessible(true);
Complex complex = (Complex) constructor.newInstance(50, 60);
System.out.println("Real Number : " + complex.getReal());
System.out.println("Imag Number : " + complex.getImag());
} catch (NoSuchMethodException | SecurityException |
InstantiationException | IllegalAccessException
| InvocationTargetException e)
{
    e.printStackTrace();
}
}
}

```

## Middleware Application

```

//Calculator.java
public class Calculator {
    public double sum( int num1, float num2, double num3) {
        return num1 + num2 + num3;
    }
    public int sub( int num1, int num2 ) {
        return num1 - num2;
    }
}

```

```

//Convert.java
class Convert{
    public static Object changeType( String type, String value ) {
        switch( type ) {
        case "int":
            return Integer.parseInt(value);
        case "float":
            return Float.parseFloat(value);
        case "double":
            return Double.parseDouble(value);
        }
        return null;
    }
}

```

```

//Program.java
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
import java.util.Scanner;

```

```

public class Program {
    public static void main(String[] args) {
        try( Scanner sc = new Scanner(System.in)){
            System.out.print("Enter F.Q. Class Name : ");
            String className = sc.nextLine();
        //org.example.domain.Calculator
            Class<?> c = Class.forName(className);

            System.out.print("Enter Method Name : ");
            String methodName = sc.nextLine(); //sum

            Method[] methods = c.getMethods();
            for (Method method : methods) {
                if( method.getName().equals( methodName ) ) {
                    Parameter[] parameters = method.getParameters();
                    Object[] arguments = new Object[ method.getParameterCount() ];
                    for( int index = 0; index < method.getParameterCount(); ++
index ) {
                        String parameterType = parameters[ index
].getType().getName();
                        System.out.print("Enter value for the argument
"+parameterType+" Type : ");
                        Object argument = Convert.changeType(parameterType,
sc.nextLine());
                        if( argument != null )
                            arguments[ index ] = argument;
                    }

                    Object result = method.invoke(c.newInstance(), arguments );
                    System.out.println("Result : "+result);
                    break;
                }
            }
        }catch( Exception ex ) {
            ex.printStackTrace();
        }
    }
}

```

### **Advantages of Reflection:**

- Dynamic class loading: Reflection allows classes to be loaded and instantiated dynamically, which can be useful when the class to be used is not known at compile time.
- Introspection: Reflection allows the properties and methods of a class to be inspected at runtime. This can be useful for building tools like debuggers or IDEs, where we need to be able to examine the structure of a program while it is running.
- Frameworks and libraries: Many Java frameworks and libraries make use of reflection to provide powerful features like dependency injection, ORM, and serialization.
- Testing: Reflection can be useful for testing by allowing access to private methods and fields, and for mocking objects.

## Drawbacks of Reflection:

- Performance overhead: Reflection operations are slower than direct method calls because they involve additional runtime checks and lookups.
- Security risks: Reflection can be used to bypass access controls and security measures, allowing malicious code to access private fields and methods or modify the behavior of a program in unexpected ways.
- Complexity and readability: Reflection code can be harder to read and understand, especially for developers who are not familiar with the language's reflection APIs.

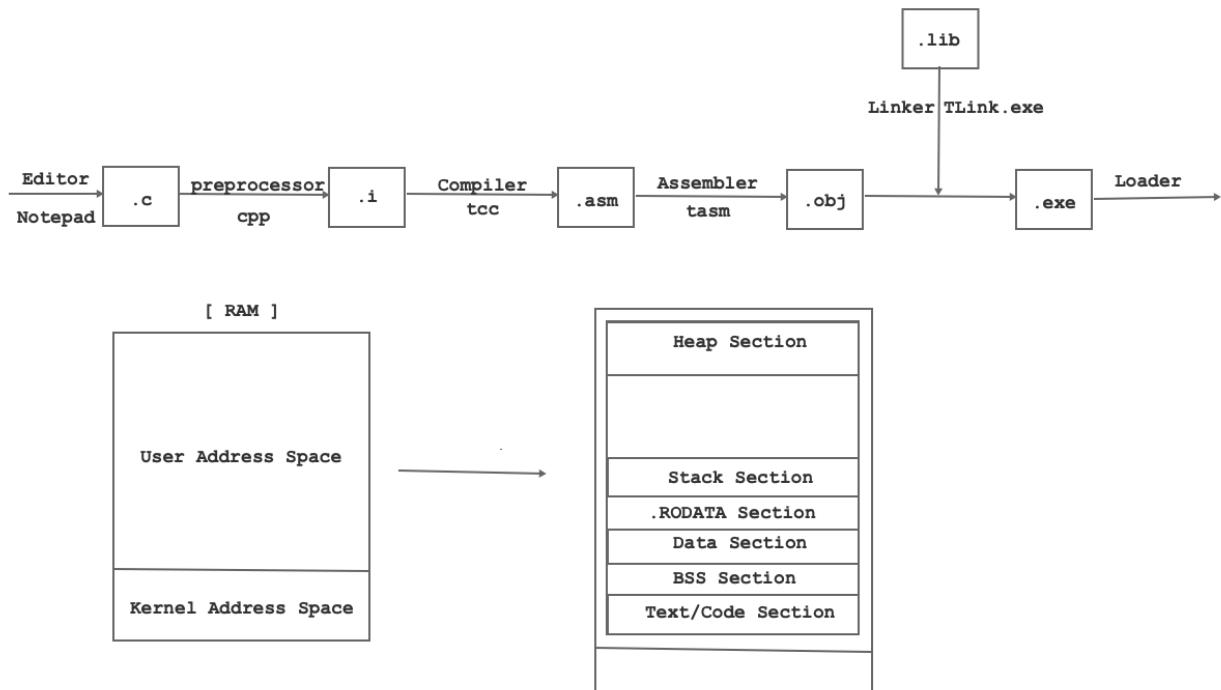
## Proxy Design Pattern( Assignment )

- The official Java documentation on the Proxy pattern:
  - <https://blogs.oracle.com/javamagazine/post/the-proxy-pattern>
- Reference: The Gang of Four (GoF) book: The GoF book "Design Patterns: Elements of Reusable Object-Oriented Software".

## Multithreading

### Singletasking versus multitasking

- Process Definition:
  - Program in execution is called as process.
  - Running instance of a program is called as process.
- Process is also called as task.

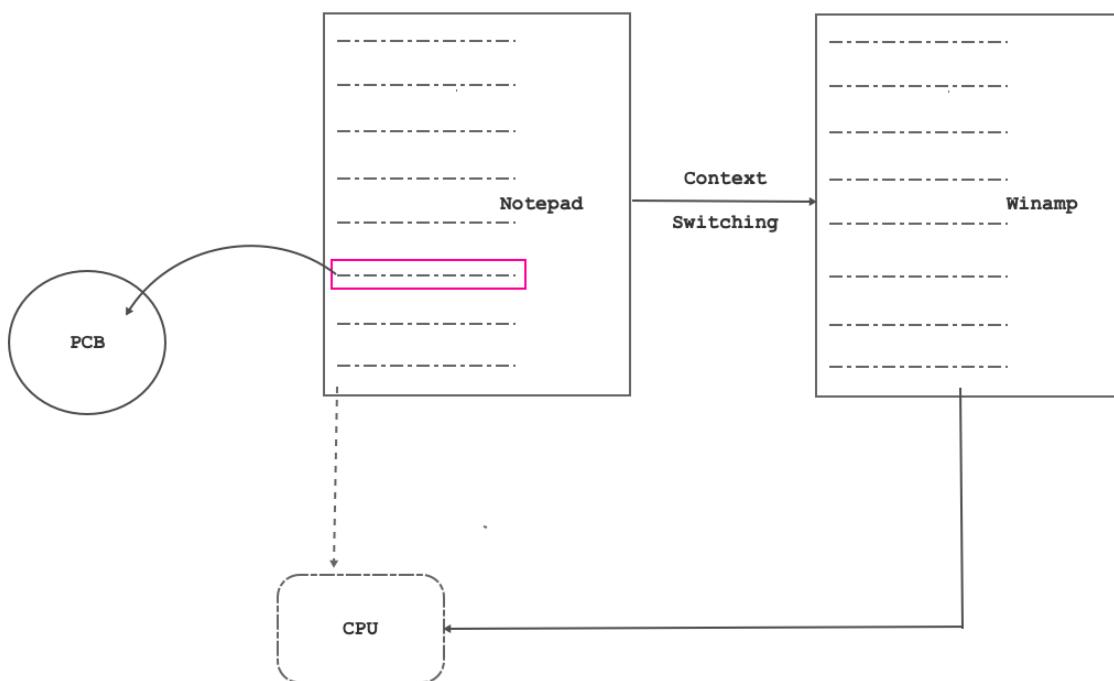


- Term Singletasking and multitasking is always used in the context of Operating System
- An ability of operating system to execute single task at a time is called as Singletasking.
- Example: MS DOS is singletasking operating system.
- An ability of operating system to execute multiple task at a time is called as Multitasking.
- Example: MS Windows, Linux, Mac OS etc.

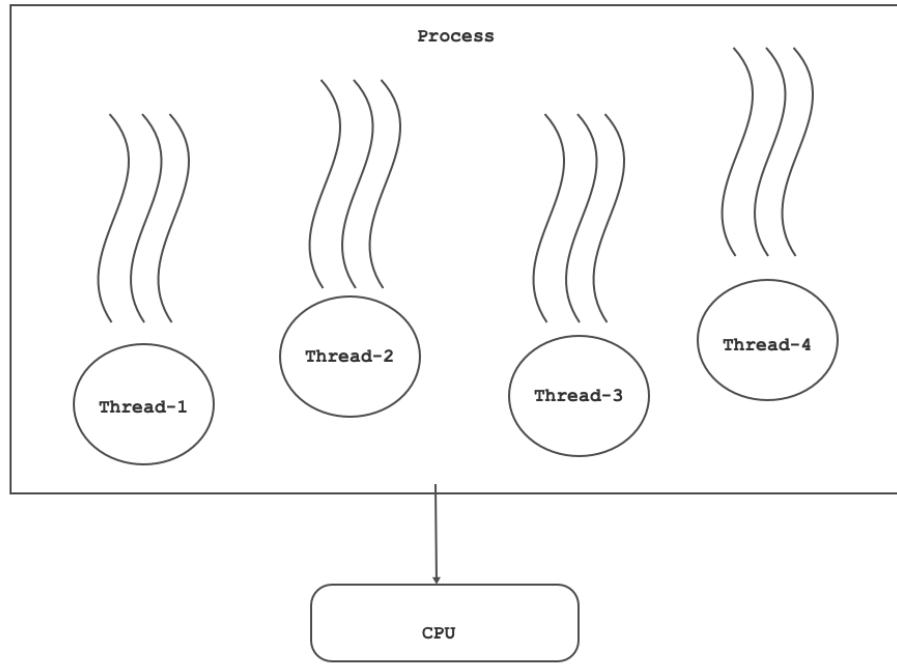
## Thread concept

- Thread Definition
  - Lightweight process is called as thread.
  - According to Java, thread is a separate path of execution which runs independently
- Thread always resides in process.
- If we want to utilize H/W resources( memory, cpu) efficiently then we should use thread.
- If any application takes help of single thread for the execution then such application is called as single threaded application.
- If any application takes help of multiple threads for the execution then such application is called as multi threaded application.
- Thread is Non Java resource. It is also called unmanaged resource.

## process based versus thread based multitasking



- Before transferring control of CPU from one process to another, scheduler must save state of the process into process control block. Then another process gets access of CPU. It is called context switching.
- Since context switching is a heavy task, process based multitasking is called as heavy weight multitasking.



- Threads always reside into process. Hence to access the resources thread do not require context switching. So thread based multitasking is called as lightweight multitasking

### **Java is multithreaded**

- When we start execution of Java application, JVM starts execution of main thread and Garbage collector. Due to these threads, every Java application is multithreaded.
- Main Thread
  - It is called as user thread / Non daemon thread.
  - Main thread is responsible for invoking main method.
  - In Java, priority of main thread is 5( Thread.NORM\_PRIORITY ).
- Garbage Collector
  - It is called as daemon thread / background thread.
  - Garbage collector is responsible for invoking finalize method and deallocating / releasing memory of unused objects.
  - Garbage collector is also called as finalizer.
  - In Java, priority of garbage collector is 8( Thread.NORM\_PRIORITY + 3).

### **Multithreading in Java**

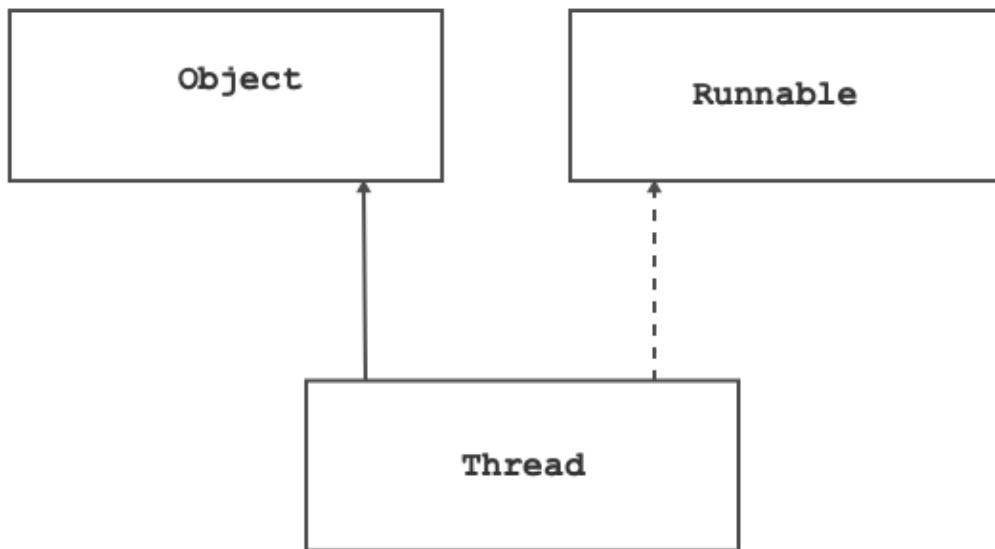
- If we want to use threads in Java then we should use Types declared in java.lang package.
- Interface(s)
  - Runnable
- Class(es)
  - Thread
  - ThreadGroup
  - ThreadLocal
- Enum:
  - Thread.State
- Exception
  - IllegalThreadStateException

- IllegalMonitorStateException
- InterruptedException

## Runnable

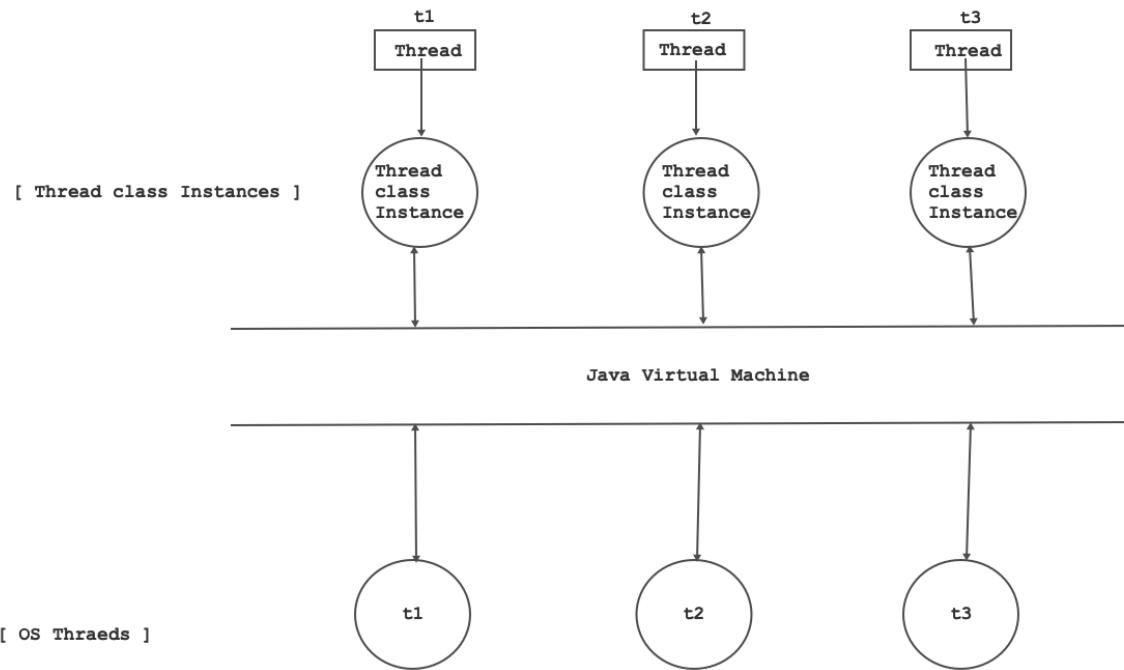
- It is functional interface declared in java.lang package.
- Method:
  - void run() //Business logic method of thread
- To create thread, we can use Runnable interface.

## Thread



- Thread is sub class of java.lang.Object class and it implements java.lang.Runnable interface.
- Instance of Thread class is not a OS thread. Rather it represents OS thread.

- JVM is responsible for mapping Thread instance with OS thread.



- Nested Type

- `Thread.State` is enum declared inside `Thread` class.

```

import java.lang.Thread.State;

public class Program {
    public static void main(String[] args) {
        State[] states = State.values();
        for (State state : states) {
            System.out.printf("%-15s%-5d\n", state.name(),
state.ordinal());
        }
    }
}

```

|               |   |
|---------------|---|
| NEW           | 0 |
| RUNNABLE      | 1 |
| BLOCKED       | 2 |
| WAITING       | 3 |
| TIMED_WAITING | 4 |
| TERMINATED    | 5 |

- Fields:

- `public static final int MIN_PRIORITY //1`
- `public static final int NORM_PRIORITY //5`
- `public static final int MAX_PRIORITY //10`

- Constructors
  - public Thread()
  - public Thread(String name)
  - public Thread(Runnable target)
  - public Thread(Runnable target, String name)
  - public Thread(ThreadGroup group, Runnable target, String name)
- Methods
  - public static Thread currentThread()
  - public final String getName()
  - public final void setName(String name)
  - public final int getPriority()
  - public final void setPriority(int newPriority)
  - public Thread.State getState()
  - public final boolean isAlive()
  - public final boolean isDaemon()
  - public final void join() throws InterruptedException
  - public final void setDaemon(boolean on)
  - public static void sleep(long millis) throws InterruptedException
  - public void start()
  - public static void yield()

## **Thread creation using java.lang.Thread class and Runnable interface**

### **User Thread versus Daemon Thread**

### **Thread termination**

### **Blocking calls in Thread**

### **Race condition and synchronized keyword**

### **Inter thread communication using wait,notify/notifyAll**

### **Synchronization using consumer/producer**

### **Thread life cycle.**

# Day 19

How to analyze thread dumps:

- <https://fastthread.io/>

## Why Java is multi-threaded

- When we start execution of Java application then JVM starts execution of main thread and garbage collector( GC ). Due to these two threads every java application is multithreaded.
  - Main Thread
    - It is user thread / non daemon thread.
    - It is responsible for calling/invoking main method.
    - Its default priority in 5( In Java, Thread.NORM\_PRIORITY ).
  - Garbage Collector / Finalizer
    - It is daemon thread / background thread.
    - It is responsible for releasing / reclaiming / deallocating memory of unused( whose reference count is 0 ) instances.
    - Before releasing memory of unused instance, GC invoke finalize( similar to destructor in C++) method on instance.
    - Its default priority in 8( In Java, Thread.NORM\_PRIORITY + 3 ).
- If we want to use non java resource into Java then we must use Java Native Interface framework.
- Thread is non Java resource and to access it we require JNI. But Java application developer need not to worry about it. Because SUN/ORACLE developers has already written logic to access OS thread in Java. In other words, Java has built-in support to thread. Hence java is considered as multithreaded programming language.

## Runnable

- It is a functional interface declared in java.lang package.
- "void run( )" is a method of java.lang.Runnable interface. In the context of multi-threading, run() method is called as business logic method.
- If we want to create thread in Java then we should use Runnable interface.

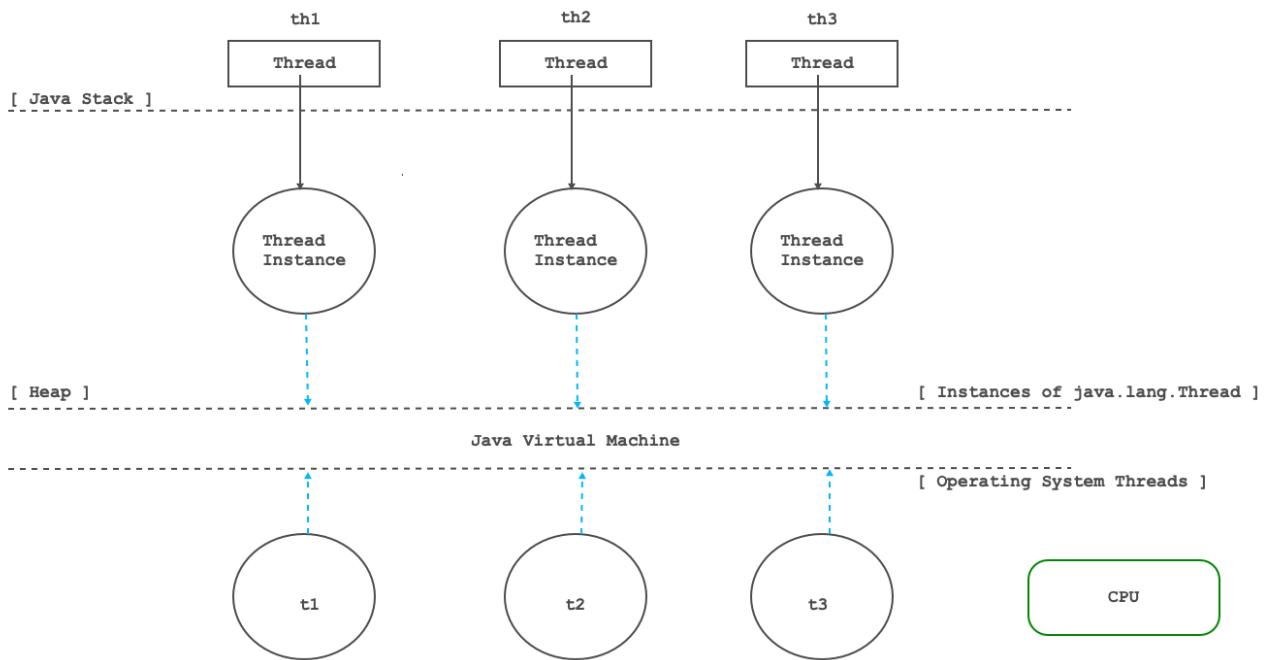
```
class Task implements Runnable{
    @Override
    public void run() {
        System.out.println("Hello from run method()");
    }
}
public class Program {
    public static void main(String[] args) {
        Runnable target = new Task(); //Upcasting
        Thread th = new Thread(target);
        th.start();
    }
}
```

## Thread

- java.lang.Thread class is a sub class of java.lang.Object class and it implements java.lang.Runnable interface.

```
public class Thread  
    extends Object  
    implements Runnable
```

- Thread is non Java resource. In other words, it is unmanaged resource. Hence developer must take care of its creation as well as termination / dispose.
- Instance of java.lang.Thread is not a operating system thread. Rather it represents operating system thread.



- If we want to utilize hardware resources efficiently then we should use thread.
- Method(s) of java.lang.Object class:
  - public String toString();
  - public boolean equals( Object obj );
  - public native int hashCode( );
  - protected native Object clone( )throws CloneNotSupportedException
  - protected void finalize()throws Throwable
  - public final native Class<?> getClass();
  - public final void wait( )throws InterruptedException
  - public final native void wait( long timeOut )throws InterruptedException
  - public final void wait( long timeOut, int nanos )throws InterruptedException
  - public final void notify();
  - public final void notifyAll();
- Method(s) of java.lang.Runnable interface

- void run();
- Nested type of java.lang.Thread class
  - java.lang.Thread.State is nested enum defined inside java.lang.Thread class.

```
package java.lang;
public class Thread extends Object implements Runnable{
    public static enum State{
        NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED;
    }
}
```

- java.lang.Thread.UncaughtExceptionHandler is nested functional interface defined inside java.lang.Thread class.

```
package java.lang;
public class Thread extends Object implements Runnable{
    @FunctionalInterface
    public static interface UncaughtExceptionHandler {
        void uncaughtException(Thread t, Throwable e);
    }
}
```

- Fields of java.lang.Thread class
  - public static final int MIN\_PRIORITY = 1; //Thread.MIN\_PRIORITY
  - public static final int NORM\_PRIORITY = 5; //Thread.NORM\_PRIORITY
  - public static final int MAX\_PRIORITY = 10; //Thread.MAX\_PRIORITY
- Constructor Summary of java.lang.Thread class:
  - public Thread();

```
Thread th = new Thread( );
```

- public Thread(String name);

```
Thread th = new Thread( "User Thread-1" );
```

- public Thread(Runnable target);

```
Runnable target = new Task();
Thread th = new Thread( target );
```

- public Thread(Runnable target, String name);

```
Runnable target = new Task();
Thread th = new Thread( target, "User Thread-1" );
```

- public Thread(ThreadGroup group, Runnable target);

```
ThreadGroup group = new ThreadGroup("CDAC-Thread-Group");
Runnable target = new Task();
Thread th = new Thread( group, target );
```

- public Thread(ThreadGroup, String name );

```
ThreadGroup group = new ThreadGroup("CDAC-Thread-Group");
Thread th = new Thread( group, "User Thread-1");
```

- public Thread(ThreadGroup group, Runnable target, String name);

```
ThreadGroup group = new ThreadGroup("CDAC-Thread-Group");
Runnable target = new Task();
Thread th = new Thread( group, target, "User Thread-1");
```

- public Thread(ThreadGroup group, Runnable target, String name , long stackSize) ;

```
ThreadGroup group = new ThreadGroup("CDAC-Thread-Group");
Runnable target = new Task();
Thread th = new Thread( group, target, "User Thread-1", 0 );
```

- In Java, we can create thread using java.lang.Thread class.

```
class Task extends Thread{
    @Override
    public void run() {
        System.out.println("Hello from run method");
    }
}
public class Program {
    public static void main(String[] args) {
        Thread th = new Task(); //Upcasting
        th.start();
    }
}
```

```
    }  
}
```

- Method Summary of java.lang.Thread class

- public static Thread currentThread()
- public long getId()
- public final String getName()
- public final void setName(String name)
- public final int getPriority()
- public final void setPriority(int newPriority)
- public Thread.State getState()
- public final ThreadGroup getThreadGroup()
- public Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()
- public static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
- public void interrupt()
- public static boolean interrupted()
- public boolean isInterrupted()
- public final boolean isAlive()
- public final boolean isDaemon()
- public final void join() throws InterruptedException
- public final void setDaemon(boolean on)
- public static void sleep(long millis) throws InterruptedException
- public static void yield()

- Getting information of main thread

```
public static void main(String[] args) {  
    Thread thread = Thread.currentThread();  
    System.out.println( thread.toString() ); //Thread[main,5,main]  
    //main : thread's name  
    //5 : priority  
    //main : thread group.  
}
```

```
public static void main(String[] args) {  
    Thread thread = Thread.currentThread();  
  
    String name = thread.getName();  
    System.out.println("Thread Name      : "+name);  
  
    int priority= thread.getPriority();  
    System.out.println("Thread Priority   : "+priority);  
  
    ThreadGroup group = thread.getThreadGroup();  
    System.out.println("Thread Group     : "+group.getName());
```

```

        State state = thread.getState();
        System.out.println("Thread State      : "+state.name());

        boolean type = thread.isDaemon();
        System.out.println("Thread Type     : "+( type ? "Daemon Thread"
: "User Thread") );

        boolean status = thread.isAlive();
        System.out.println("Thread Status   : "+( status ? "Alive" :
"Dead") );
    }
}

```

- Output is:

```

Thread Name      : main
Thread Priority  : 5
Thread Group     : main
Thread State     : RUNNABLE
Thread Type      : User Thread
Thread Status    : Alive

```

- Getting information of garbage collector

```

class Sample{
    public Sample( ) {
        System.out.println("Inside constructor of
"+this.getClass().getSimpleName());
    }
    public void print( ) {
        System.out.println("Inside print method of
"+this.getClass().getSimpleName());
    }
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Inside finalize method of
"+this.getClass().getSimpleName());
    }

    Thread thread = Thread.currentThread();

    String name = thread.getName();
    System.out.println("Thread Name      : "+name);

    int priority= thread.getPriority();
    System.out.println("Thread Priority   : "+priority);

    ThreadGroup group = thread.getThreadGroup();
    System.out.println("Thread Group     : "+group.getName());

    State state = thread.getState();
}

```

```

        System.out.println("Thread State      : "+state.name());
        boolean type = thread.isDaemon();
        System.out.println("Thread Type      : "+( type ? "Daemon Thread" : "User Thread" ) );
        boolean status = thread.isAlive();
        System.out.println("Thread Status     : "+( status ? "Alive" : "Dead" ) );
    }
}

```

```

public static void main1(String[] args) {
    Sample sample = null;
    sample = new Sample();
    sample.print();
    sample = null;
    //System.gc(); //or
    Runtime.getRuntime().gc();
}

```

- Output is

```

Thread Name      : Finalizer
Thread Priority   : 8
Thread Group     : system
Thread State     : RUNNABLE
Thread Type      : Daemon Thread
Thread Status     : Alive

```

## What do you know about final/finally/finalize?

- final
  - It is keyword, we can use with local variable, field, method and class.
    - If we use final modifier with method local variable and field then it is considered as constant.
    - If we use final modifier with method then we can not override/redefine method inside sub class
    - If we use final modifier with class then we can not extend it.
- finally
  - It is a block that we can use with try/catch.
    - If we want to close local resources then we should use finally block.

```

public static void main(String[] args) {
    Scanner sc = null; //Method local variable
    try{

```

```

sc = new Scanner( System.in );
//TODO
}catch( Exception ex ){
    ex.printStackTrace();
}finally{
    sc.close(); //Releasing local resource
}
}

```

- finalize
  - It is non final method of java.lang.Object class.
  - If we want to release class level( which is declared as field ) resources then we should use finalize method.

```

class Sample{
    private Scanner sc; //Field
    public Sample( ) {
        this.sc = new Scanner( System.in );
    }
    //TODO
    @Override
    protected void finalize() throws Throwable {
        this.sc.close();
    }
}
public class Program {
    public static void main(String[] args) {
        Sample sample = null;
        sample = new Sample();
        sample.print();
        sample = null;
        System.gc
    }
}

```

## Thread Creation in Java

- In Java, we can create thread using 2 ways:
  - By implementing Runnable interface
  - By extending Thread class.

### Thread Creation in by implementing Runnable interface

- If we create Thread instance but do not call start() method on it then is considered in NEW state.

```

Runnable target = new Task(); //Upcasting
Thread th = new Thread(target); //NEW

```

- To start execution of thread we should start() method.
- If we call start() method on thread instance then thread is considered in RUNNABLE state.

```
Runnable target = new Task(); //Upcasting
Thread th = new Thread(target); //NEW
th.start(); //RUNNABLE
```

- If we call start() method on already started thread then start() method throws IllegalThreadStateException

```
Runnable target = new Task(); //Upcasting
Thread th = new Thread(target); //NEW
th.start(); //RUNNABLE
th.start(); //IllegalThreadStateException
```

- When control come out of run method then thread gets terminated. In this case thread is considered in TERMINATED state.
- In following cases, control can come out of run method and thread can terminate:
  - Successful completion of run method.
  - Getting exception during execution of run method.
  - Execution of jump statement( return statement ) inside run method.
- If we want to suspend execution of running thread then we should use sleep() method. It is a static method of java.lang.Thread class.

```
public static void sleep(long millis) throws InterruptedException
```

- millis - the length of time to sleep in milliseconds
  - 1000 millisecond = 1 second
- Usage:

```
Thread.sleep( 250 )
```

- Following methods calls are considered as blocking calls:
  - sleep()
  - suspend()
  - join()
  - wait()
  - Input operations in run method

- Generally we should use blocking calls very carefully.
- If we want to group functionally realated threads together then we should use ThreadGroup.

```

public static void main(String[] args) {
    Runnable target = new Task();

    ThreadGroup threadGroup = new ThreadGroup("acts");

    Thread th1 = new Thread(threadGroup, target);
    th1.setName("User Thread#1");
    th1.start();

    Thread th2 = new Thread(threadGroup, target);
    th2.setName("User Thread#2");
    th2.start();
}

```

- Thread creation using Runnable

```

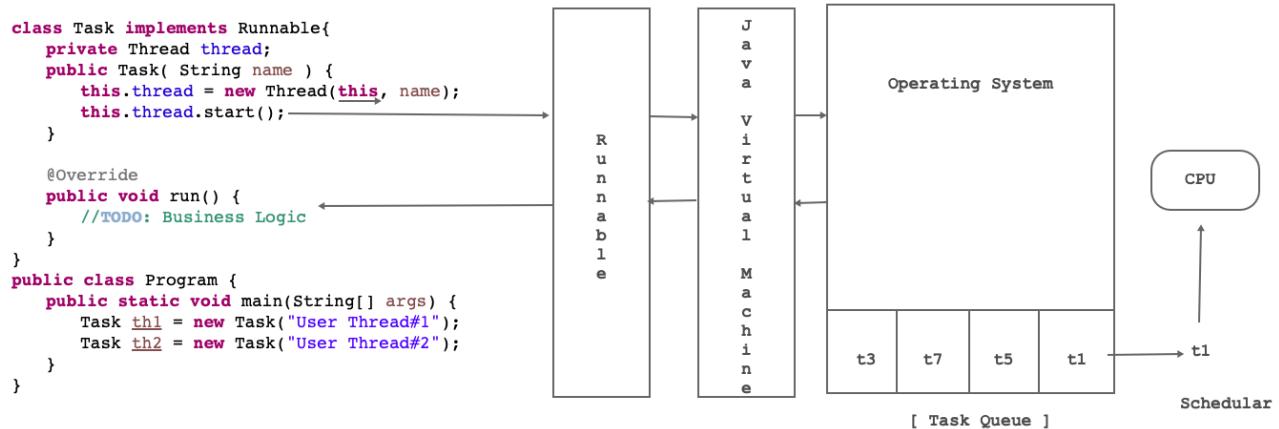
class Task implements Runnable{
    private Thread thread;
    public Task( String name ) {
        this.thread = new Thread(this, name);
        this.thread.start();
    }

    @Override
    public void run() {
        //TODO: Business Logic
    }
}

public class Program {
    public static void main(String[] args) {
        Task th1 = new Task("User Thread#1");
        Task th2 = new Task("User Thread#2");
    }
}

```

- what is the relation between start and run method?



- start() method do not call run() method.
- If we call start() method on java thread instance then it is considered as request to the JVM to get thread from operating system and to map it with java thread instance.
- Scheduler is responsible for picking thread from queue and assigning CPU to it. When scheduler assign CPU to the OS thread then JVM invoke run method on instance who has implemented java.lang.Runnable interface.

### Thread Creation in by extending Thread class

```

class Task extends Thread{
    public Task( String name ) {
        super(name);
        this.start();
    }
    @Override
    public void run() {
        //TODO : Business logic
    }
}
public class Program {
    public static void main(String[] args) {
        Task th1 = new Task("User Thread#1");
    }
}

```

## Day 20

**What will happen if we call run method instead of start method on thread instance**

```

class Task extends Thread{
    public Task( String name ) {
        super( name );
    }
    @Override

```

```

public void run() {
    System.out.println("Inside run method : "
+Thread.currentThread().getName());
}
}

public class Program {
    public static void main(String[] args) {
        Thread thread = new Task( "Thread#1");
        //thread.start(); //Inside run method : Thread#1
        thread.run(); //Inside run method : main
    }
}

```

- If we call start() method on thread instance then JVM starts execution of new Thread. But if we call run() method on thread instance then JVM do not start execution of new Thread. In above code, main thread is calling main method and main method is calling run() method. In directly main thread is calling run() method.

## What is the difference between creating thread using Runnable and Thread class

### Runnable

- Consider Thread creation using Runnable interface:

```

class A implements Runnable{
    private Thread thread;
    public A( ){
        this.thread = new Thread( this);
        this.thread.start();
    }
    @Override
    public void run( ){
        //TODO: Write business logic here
    }
}

```

- Here class can still extend another class.
- Let us try to create sub class of class A

```

class B extends A{
    public B( ){
    }
    @Override
    public void run( ){
        //TODO: Write business logic here
    }
}

```

```

class C extends B{
    public C( ){
    }
    @Override
    public void run( ){
        //TODO: Write business logic here
    }
}

```

```

class D extends C{
    public D( ){
    }
    @Override
    public void run( ){
        //TODO: Write business logic here
    }
}

```

- If create instance of class B then first A class and then B class constructor will call. In class A, Thread registration process will start and when OS thread will get CPU then B class run method will call. Same is the case of class C and D. In simple words, If class implement Runnable interface then that class and all its sub classes must participate in Threading behavior.

## Thread

- Consider Thread creation by extending Thread class:

```

class A extends Thread{
    public A( ){
        this.start();
    }
    @Override
    public void run( ){
        //TODO: Write business logic here
    }
}

```

- In Java, class can extend more than one class. So here we can not extend another class / if class already sub class of another class then we can not extend Thread.
- Let us try to create sub class of class A

```

class B extends A{
    public B( ){
    }
}

```

```
@Override  
public void run( ){  
    //TODO: Write business logic here  
}  
}
```

```
class C extends B{  
    public C( ){  
    }  
    @Override  
    public void run( ){  
        //TODO: Write business logic here  
    }  
}
```

```
class D extends C{  
    public D( ){  
    }  
    @Override  
    public synchronized void start( ) {  
        //Keep Empty  
    }  
}
```

- In above code class A, B and C must participate into Threading behavior. But by overriding start( ) method class D can come out of threading behavior.

## Types of Thread in java:

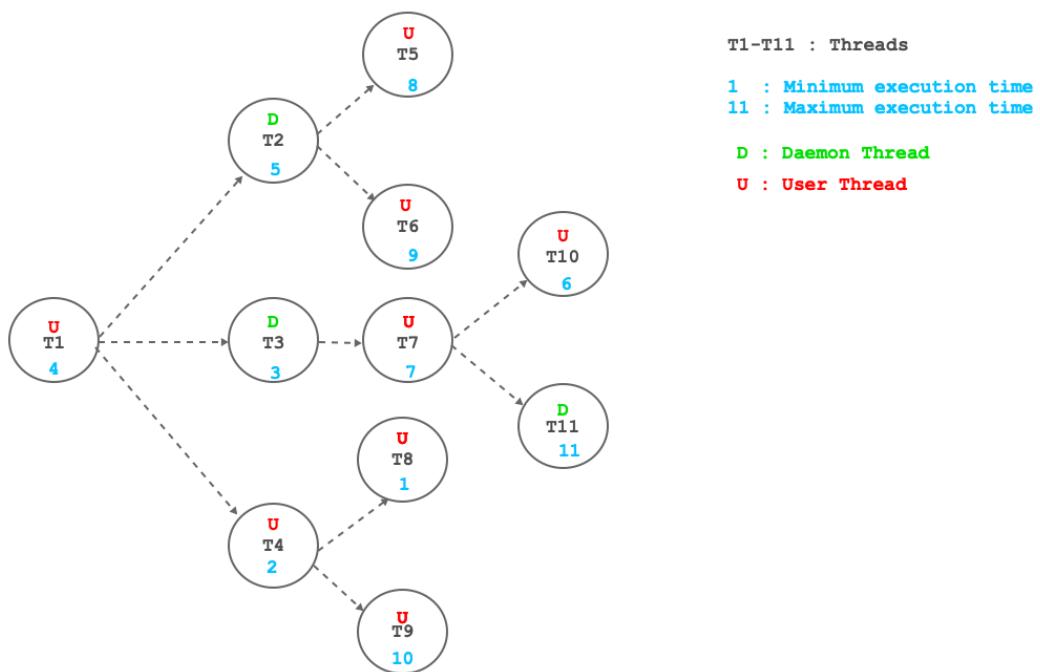
- User Thread
  - It is also called as non daemon thread.
  - If we create any thread from main method / another user thread then it is by default considered as user thread.
  - Default Properties of main thread:
    - Type: User Thread
    - Priority: 5
    - Thread group: main
- Daemon Thread
  - It is also called as background thread.
  - If we create any thread from finalize method / another deamon thread then it is by default considered as daemon thread.
  - Default Properties of GC thread:
    - Type: Daemon Thread
    - Priority: 8
    - Thread group: system

- Using "public final boolean isDaemon()" method we can check type of thread.
- Using "public final void setDaemon(boolean on)" method we can marks thread as either a daemon thread or a user thread.

```
Thread th = new Thread( target, name );
th.setDaemon( true ); //Not thread will be considered as Daemon
th.start();
```

## What is the difference between User Thread and Daemon Thread

- User threads have higher priority than daemon threads, which means that the JVM will keep them running as long as the application is running. Daemon threads, on the other hand, have lower priority, and the JVM will stop them as soon as all user threads have terminated.



## interrupt() / interrupted() / isInterrupted()

- An interrupt() is a notification sent to a thread to request that it should stop what it's doing and do something else.

```
public void interrupt()
```

```
class Task implements Runnable {
    @Override
    public void run() {
        for (int count = 1; count <= 100; ++count) {
            System.out.println("Count : " + count);
            if( count == 50 )
```

```

        Thread.currentThread().interrupt();
    }
}
}

```

- When the interrupt signal is sent to the thread, it will set an interrupted flag on the thread. We can check that flag using isInterrupted() method.

```
public boolean isInterrupted();
```

```

for (int count = 1; count <= 100; ++count) {
    if( !Thread.currentThread().isInterrupted()) {
        System.out.println("Count : " + count);
        if( count == 50 ) {
            Thread.currentThread().interrupt();
        }
    }
}

```

- isInterrupted() method only check interrupted status but do not reset the status.
- Using interrupted() method we can check interrupted status but it reset interrupted status. In other words, if this method were to be called twice in succession, the second call would return false

```
public static boolean interrupted()
```

```

class Task implements Runnable {
    @Override
    public void run() {
        for (int count = 1; count <= 100; ++count) {
            if (!Thread.currentThread().isInterrupted()) {
                System.out.println("Count : " + count);
                if (count == 50 ) {
                    Thread.currentThread().interrupt();
                }
            }
        }

        while (!Thread.interrupted()) {
            for (int count = 100; count <= 120; ++count)
                System.out.println("Count : " + count);
        }
    }
}

```

```

public class Program {
    public static void main(String[] args) throws Exception {
        Thread thread = new Thread(new Task());
        thread.start();
    }
}

```

## Thread Priority

- OS scheduler is responsible for assigning CPU to the thread.
- On the basis of thread priority scheduler assign CPU to the thread.
- Thread priorities of Java and OS are different.
- Thread priorities in Java:
  - Thread.MIN\_PRIORITY = 1;
  - Thread.NORM\_PRIORITY = 5;
  - Thread.MAX\_PRIORITY = 10;
- How will you read thread priority in Java?

```

public static void main(String[] args) {
    Thread thread = Thread.currentThread();
    int priority = thread.getPriority();
    System.out.println("Priority of "+thread.getName()+" is
"+priority);
}

```

- How will you set thread priority in Java?

```

public static void main(String[] args) {
    Thread thread = Thread.currentThread();
    thread.setPriority(thread.getPriority() + 2);
    System.out.println("Priority of "+thread.getName()+" is
"+thread.getPriority());
    //Priority of main is 7
}

```

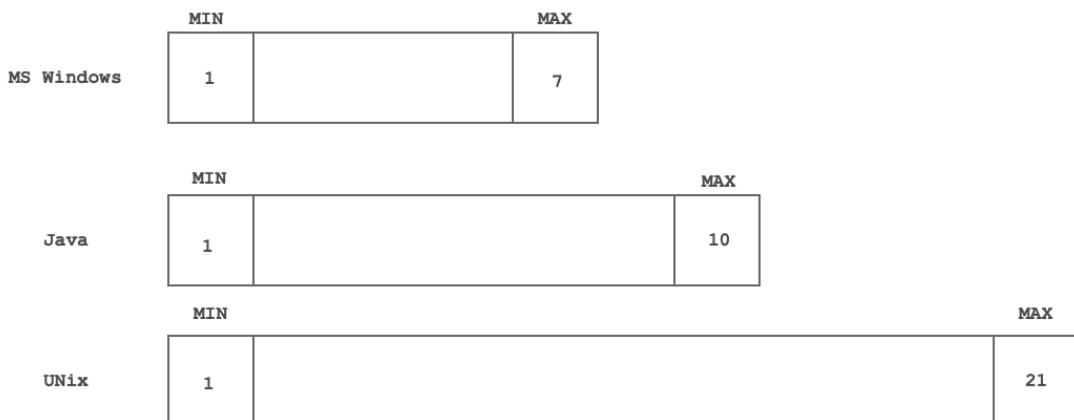
```

public static void main(String[] args) {
    Thread thread = Thread.currentThread();
    thread.setPriority(thread.getPriority() + 6);
    //IllegalArgumentException
    System.out.println("Priority of "+thread.getName()+" is
"+thread.getPriority());
}

```

- Default priority of any thread depends on priority of its parent thread.
- Consider following code:

```
class Task implements Runnable{
    private Thread thread;
    public Task(){
        this.thread = new Thread( this );
        this.thread.setPriority( Thread.MAX_PRIORITY); //OK
        this.thread.start();
    }
    public void run( ){
        //TODO
    }
}
```



| Threads | Java | Unix | Windows |
|---------|------|------|---------|
| t1      | 8    | 16   | 7       |
| t2      | 7    | 14   | 7       |
| t3      | 3    | 6    | 3       |

- If we set priority of a thread in Java then OS map it differently on different system. Hence Same Java application produces different behavior on different OS.
- Which features of Java makes Java application platform dependent:
  - Abstract Window Toolkit(AWT) Components
    - AWT components implicitly use peer classes and these classes have been written in C++ which are OS specific.
  - Thread priorities

## Thread Join

- If we call join method on thread instance then it blocks execution of all other threads

- Consider following code:

```

class Task extends Thread{
    public Task( String name ) {
        super( name );
        this.start();
    }
    @Override
    public void run() {
        System.out.println("Start of run method");
        try {
            for( int count = 1; count <= 10; ++ count ) {
                System.out.println(this.getName()+" : Count : "+count);
                Thread.sleep(250);
            }
        } catch (InterruptedException cause) {
            throw new RuntimeException( cause );
        }
        System.out.println("End of run method");
    }
}
public class Program {
    public static void main(String[] args) throws InterruptedException
    {
        Thread th1 = new Task("Thread#01");
        th1.join();

        Thread th2 = new Task("Thread#02");
        th2.join();

        Thread th3 = new Task("Thread#03");
        th3.join();

        Thread th4 = new Task("Thread#04");
        th4.join();

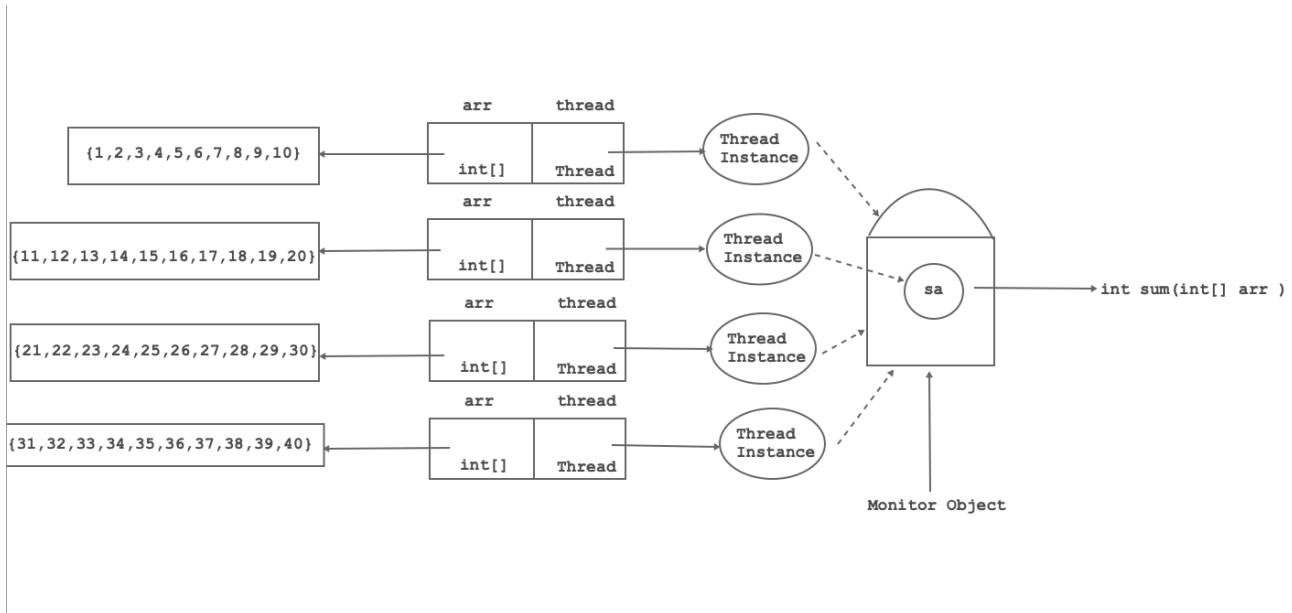
        Thread th5 = new Task("Thread#05");
        th5.join();
    }
}

```

## Race Condition

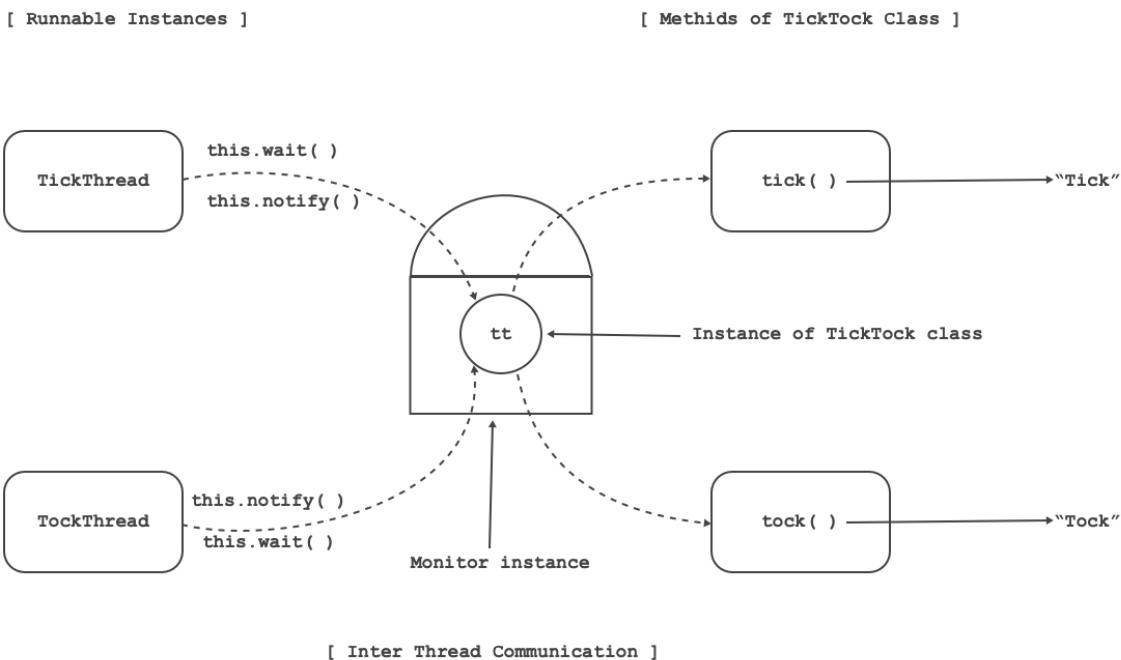
- A race condition is a situation that occurs in a concurrent system when two or more threads or processes access a shared resource or variable in an uncontrolled order, resulting in unpredictable and often incorrect behavior.
- To prevent race conditions, concurrency control mechanisms such as locks, semaphores, and monitors can be used to ensure that only one thread at a time can access and modify a shared resource.
- In Java, we can use synchronized keyword with block/method to avoid race condition.

- If threads are waiting to get monitor object associated with shared resource then it is consider in BLOCKED state.



## Inter Thread Communication

- Synchronization is the process of controlling the access of multiple threads to a shared resource.
- Inter-thread communication is a mechanism that allows threads in a multi-threaded application to communicate with each other and coordinate their actions.



- In Java, inter-thread communication is achieved using the `wait()`, `notify()` and `notifyAll()` methods, which are defined in the `Object` class.
  - wait()** : This method causes the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method for the same object. The thread will release the lock it holds on the object and wait until it's notified by another thread.
  - notify()** : This method wakes up a single thread that is waiting on the object. If there are multiple threads waiting, only one thread will be awakened. The awakened thread will not be

- able to proceed until it regains the lock on the object.
- **notifyAll()** : This method wakes up all the threads that are waiting on the object. All the threads will then compete for the lock on the object.
  - The JVM throws the IllegalMonitorStateException when a thread attempts to call the wait(), notify(), or notifyAll() methods on an object without holding the object's monitor.

```
class TickTock{
    public void tick() throws InterruptedException {
        synchronized( this ) {
            System.out.print("Tick  ");
            this.notify();
            this.wait( 1000 ); //To avoid deadlock pass time
        }
    }
    public void tock() throws InterruptedException {
        synchronized( this ) {
            System.out.println("      Tock");
            this.notify();
            this.wait( 1000 ); //To avoid deadlock pass time
        }
    }
}
```

```
class Task implements Runnable{
    Thread thread;
    public Task( String name) {
        this.thread = new Thread(this, name);
        this.thread.start();
    }
    private static TickTock tt = new TickTock();
    @Override
    public void run() {
        try {
            if( Thread.currentThread().getName().equals("TickThread") ) {
                for( int count = 1; count <= 5; ++ count ) {
                    tt.tick();
                    Thread.sleep(250);
                }
            }else {
                for( int count = 1; count <= 5; ++ count ) {
                    tt.tock();
                    Thread.sleep(250);
                }
            }
        } catch (InterruptedException cause) {
            throw new RuntimeException( cause );
        }
    }
}
```

```

public class Program {
    public static void main(String[] args) throws Exception{
        Task t1 = new Task("TickThread");

        Task t2 = new Task("TockThread");
    }
}

```

### Why wait/notify/notifyAll methods are declared in java.lang.Object class?

- The wait(), notify(), and notifyAll() methods are designed to work with monitors, which are associated with objects rather than threads.
- When a thread calls wait() on an object, it releases the monitor associated with that object and enters a waiting state until another thread notifies it by calling notify() or notifyAll() on the same object.
- Therefore, it makes sense for these methods to be declared in the java.lang.Object class.

```

public class Program {
    public void print( String str ) throws InterruptedException {
        synchronized( this ) {
            System.out.println(str);
            //str.wait(1000); //IllegalMonitorStateException
            this.wait( 1000 );
        }
    }
    public static void main(String[] args) throws Exception{
        Program p = new Program();
        p.print("Hello");
    }
}

```

### The Producer-Consumer problem

- It is a classic synchronization problem in computer science where there are two threads - a producer and a consumer - sharing a common resource.
- Consider a problem can be seen in a fast-food restaurant's kitchen:
  - the kitchen is the common resource
  - the chef is the producer who produces the food
  - waiter is the consumer who serves the food to the customers.
- We will try to solve this problem using synchronization mechanisms like wait(), notify(), and notifyAll().
- Consider code for Kitchen class:

```

import java.util.LinkedList;
public class Kitchen {
    private LinkedList<String> orders = new LinkedList<>();
    private int maxOrders = 10;

    public synchronized void addOrder(String order) throws
InterruptedException {
        while (orders.size() == maxOrders) {
            wait();
        }
        orders.add(order);
        System.out.println("Order added: " + order);
        notifyAll();
    }

    public synchronized String getOrder() throws InterruptedException {
        while (orders.size() == 0) {
            wait();
        }
        String order = orders.removeFirst();
        System.out.println("Order removed: " + order);
        notifyAll();
        return order;
    }
}

```

- Consider code for Chef class:

```

public class Chef implements Runnable {
    private Kitchen kitchen;

    public Chef(Kitchen kitchen) {
        this.kitchen = kitchen;
    }

    @Override
    public void run() {
        try {
            while (true) {
                String order = "Burger"; // produce food
                kitchen.addOrder(order);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

- Consider code for Waiter class:

```

public class Waiter implements Runnable {
    private Kitchen kitchen;
    public Waiter(Kitchen kitchen) {
        this.kitchen = kitchen;
    }

    @Override
    public void run() {
        try {
            while (true) {
                String order = kitchen.getOrder();
                serve(order); // consume food
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    private void serve(String order) {
        System.out.println("Serving " + order);
    }
}

```

- Consider code for Waiter class:

```

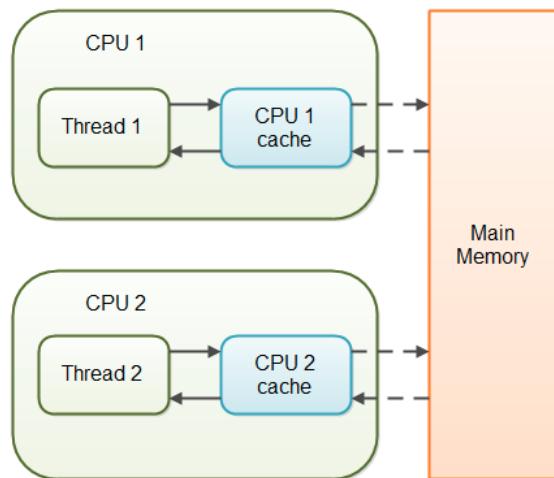
public class Restaurant {
    public static void main(String[] args) {
        Kitchen kitchen = new Kitchen();
        Thread chefThread = new Thread(new Chef(kitchen));
        Thread waiterThread = new Thread(new Waiter(kitchen));
        chefThread.start();
        waiterThread.start();
    }
}

```

## Volatile Fields:

- Computers with multiple processors can temporarily hold memory values in registers or local memory caches. As a consequence, threads running in different processors may see different values for the

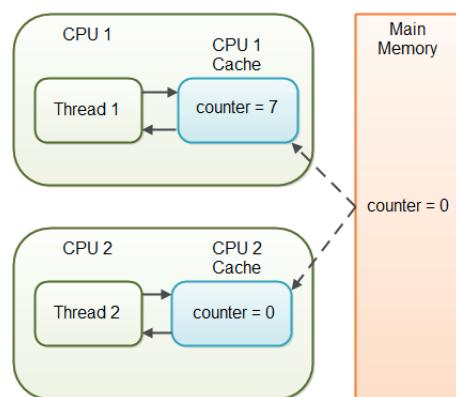
same memory location.



- Consider following code:

```
class SharedInstance{  
    int counter = 0;  
}
```

- Here we assume that there are two threads are working on SharedInstance.
- If these two threads run on different processors then each thread will have its own local copy of counter.
- If we modifies value of one thread, then its value might not reflect in the original one in the main memory instantly. It is totally depends on the write policy of cache.
- Now the other thread is not aware of the modified value which leads to data inconsistency.



- If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory.
- volatile is a keyword in Java which is applicable only for fields.

```
Access_Modifier volatile Data_Type variableName;
```

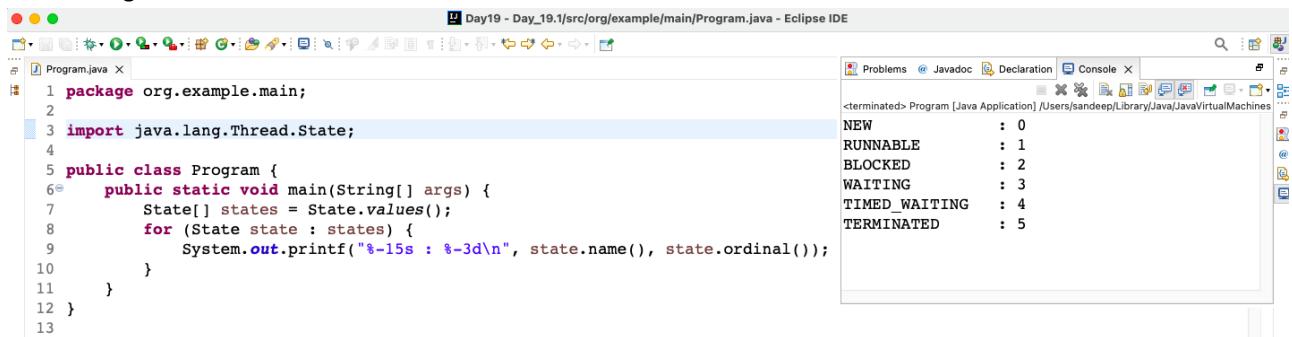
- volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.
- The volatile keyword tells the computer that a variable may be accessed by multiple threads at the same time. This means that any changes made to the variable are immediately visible to all other threads. It works like a special type of variable that is shared among all threads and stored in main memory, so every time a thread wants to access it, it reads the updated value from main memory and writes any changes directly back to main memory. Any change to a volatile variable is visible to all other threads immediately after it happens.

## Thread life cycle:

- Thread.State is nested enum declared in java.lang.Thread class.

```
public enum State {
    NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED;
}
```

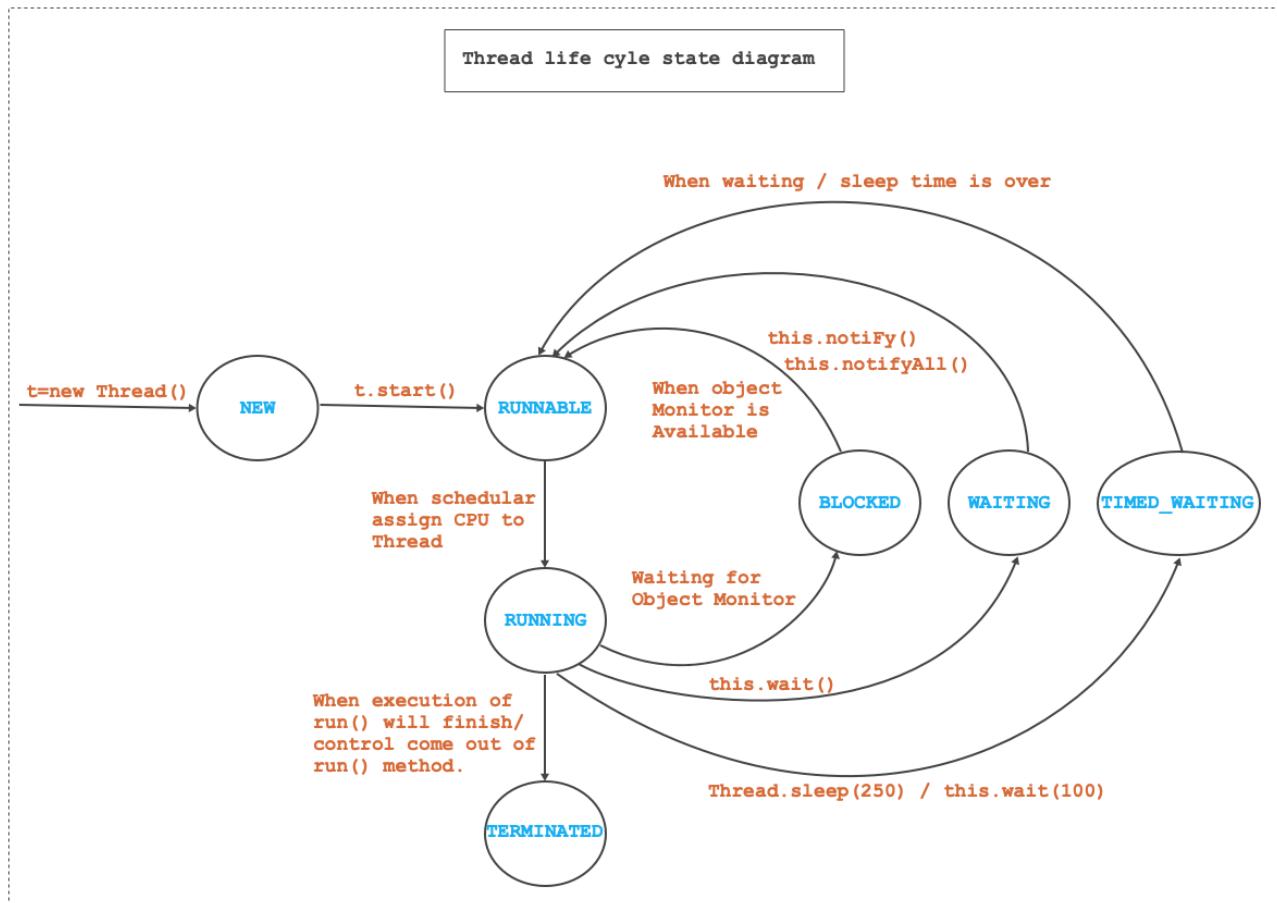
- Accessing enum in Java



The screenshot shows the Eclipse IDE interface with a Java file named 'Program.java' open. The code prints the names and ordinal values of the Thread.State enum. The output window shows the results:

| State         | Ordinal |
|---------------|---------|
| NEW           | 0       |
| RUNNABLE      | 1       |
| BLOCKED       | 2       |
| WAITING       | 3       |
| TIMED_WAITING | 4       |
| TERMINATED    | 5       |

- The life cycle of a thread in Java consists of several states, which are as follows:



- A thread can be in only one state at a given point in time.
- These states are virtual machine states which do not reflect any operating system thread states.
  - New:** The thread is in the new state when it is created but has not yet started executing. The thread remains in this state until the start() method is called.
  - Runnable:** The thread is in the runnable state when it is ready to run, but the thread scheduler has not selected it to run yet. When the thread scheduler selects the thread, it moves to the running state.
  - Running:** The thread is in the running state when its run() method is being executed.
  - Blocked:** The thread is in the blocked state when it is waiting for a monitor lock to be released, so it can enter a synchronized block or method.
  - Waiting:** The thread is in the waiting state when it is waiting for some other thread to perform a particular action before it can continue executing.
  - Timed Waiting:** The thread is in the timed waiting state when it is waiting for a specific amount of time for some other thread to perform a particular action before it can continue executing.
  - Terminated:** The thread is in the terminated state when its run() method has completed execution.

## String Handling

- Consider String in C/C++ language:

```
"DAC";
//{'D','A','C','\0'}
```

- In Java, String is collection of character instances which do not end with '\0' character.
- If we want to manipulate String in Java then we can use following classes:
  - java.lang.String
  - java.lang.StringBuffer
  - java.lang.StringBuilder
  - java.util.StringTokenizer
  - java.text.Pattern
  - java.text.Matcher
  - org.apache.commons.lang3.StringUtils

### **java.lang.String**

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

- String is final class declared in java.lang package.
- It is sub class of java.lang.Object class and it implements Serializable, Comparable, CharSequence
- In Java, String is not primitive data type. Since it is a class, it is considered as non primitive / reference type.
- Even though String is non primitive type, we can create its instance with and without new operator.

```
String s1 = new String("CDAC");
//s1 : String reference
//new String("CDAC"); : String instance
```

- If we create String instance using new operator then it gets space on heap section.

```
String s1 = "CDAC";
//s1 : String reference
//"CDAC" : String literal
```

- If we create String without new operator the it gets space on String literal on method area.

```

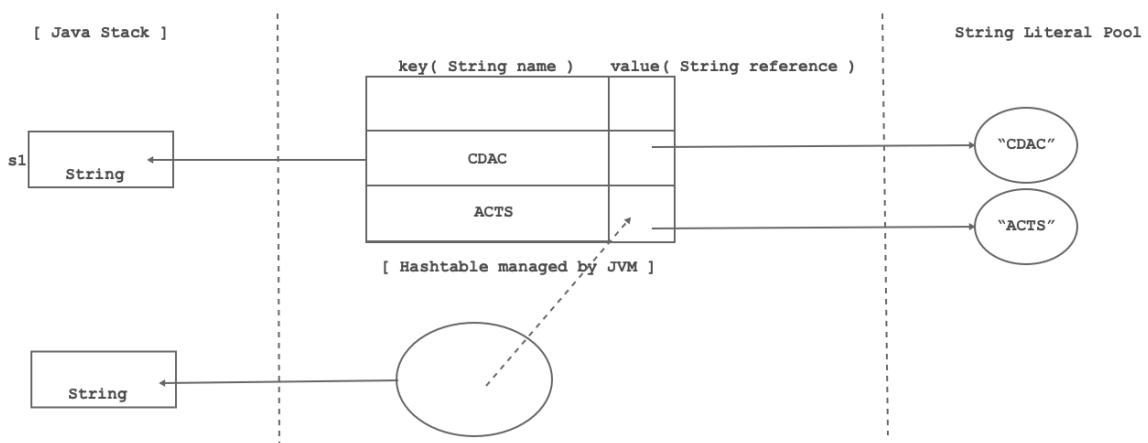
String s1 = "CDAC";
//char[] data = {'C','D','A','C'};
//String s1 = new String( data );

```

```

String s1 = "CDAC";
String s2 = new String("ACTS");

```



- In Java, `String` do not ends with '\0' character. Using illegal index, if we try to access character from `String` then `String` methods throws `StringIndexOutOfBoundsException`.

```

public static void main(String[] args){
    String str = "CDAC";
    char ch = str.charAt(4); //StringIndexOutOfBoundsException
}

```

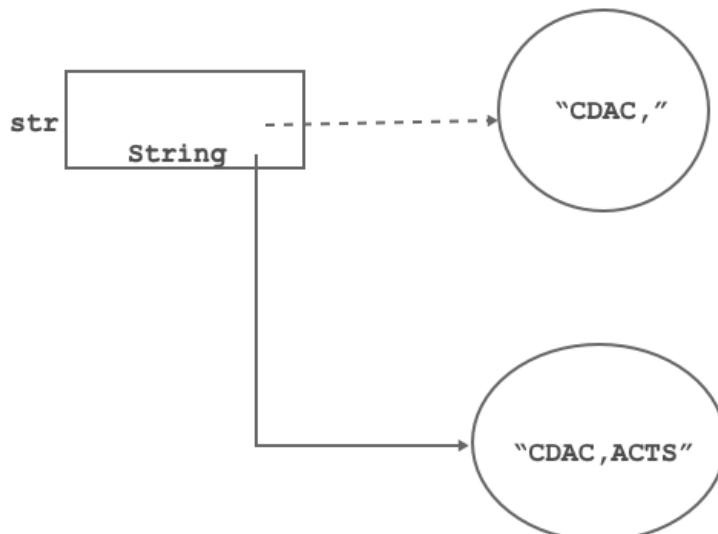
- Strings are constant; their values cannot be changed after they are created. In other words, `String` instances are immutable.
- Consider following code:

```

String str = "CDAC,";
str = str + "ACTS";
System.out.println( str ); //CDAC,ACTS

```

```
String str = "CDAC,";  
str = str + "ACTS";
```



[ String objects are immutable objects ]

```
public static void main(String[] args) {  
    String s1 = "CDAC,";  
    s1.concat("ACTS");  
    System.out.println(s1); //CDAC,  
  
    System.out.println(s1.concat("ACTS")); //CDAC,ACTS  
  
    String s2 = s1.concat("ACTS");  
    System.out.println(s2); //CDAC,ACTS  
}
```

- The Java language provides special support for the string concatenation operator ( + ), and for conversion of other objects to strings.
  - Using + operator, we can contact any(primitive/non primitive) value to the String but using concat method we can concat only String.

```
int number = 123;  
String s1 = "Number : "+number;  
String s2 = "Date : "+new Date();
```

```
int number = 123;  
String s1 = "Number : ".concat( number ); //Compiler error  
String s1 = "Number : ".concat( String.valueOf(number) ); //OK
```

- Constructor Summary of String class

- public String()

```
String s1 = new String();
```

- public String(byte[] bytes)

```
byte[] bs = new byte[ ]{ 65, 66, 67 };  
String s1 = new String( bs ); //ABC
```

- public String(char[] value)

```
char[] data = new char[ ]{ 'A','B','C'};  
String s1 = new String( data ); //ABC
```

- public String(String original)

```
String s1 = new String("ABC");
```

- public String(StringBuffer buffer)

```
StringBuffer buffer = new StringBuffer("CDAC");  
String s1 = new String( buffer );
```

- public String(StringBuilder builder)

```
StringBuilder builder = new StringBuilder("ACTS");  
String s1 = new String( builder );
```

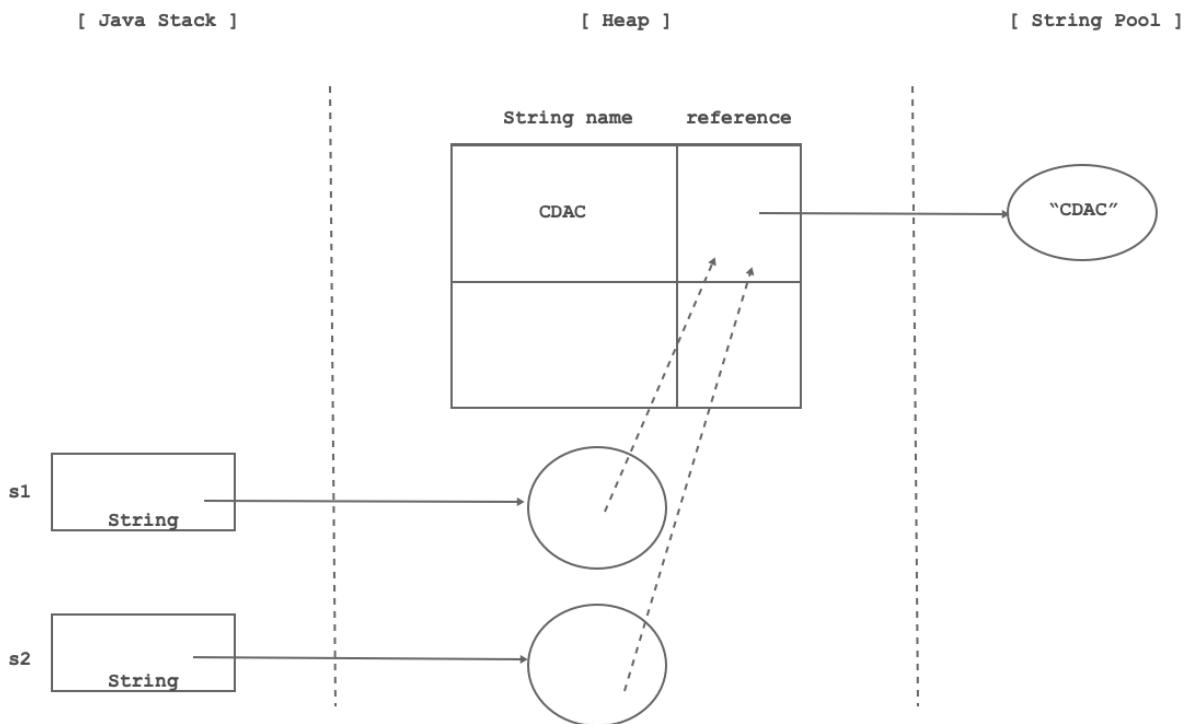
- Method Summary of String class:

- public char charAt(int index)
  - public String concat(String str)
  - public boolean contains(CharSequence s)
  - public boolean equalsIgnoreCase(String anotherString)
  - public static String format(String format, Object... args)
  - public byte[] getBytes()
  - public int indexOf(int ch)
  - public int lastIndexOf(int ch)

- public int indexOf(String str)
- public int lastIndexOf(String str)
- public String intern()
- public int length()
- public boolean matches(String regex)
- public String[] split(String regex)
- public boolean startsWith(String prefix)
- public boolean endsWith(String suffix)
- public char[] toCharArray()
- public String toUpperCase()
- public String toLowerCase()
- public String trim()
- public static String valueOf(Object obj)

### String twisters

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



```

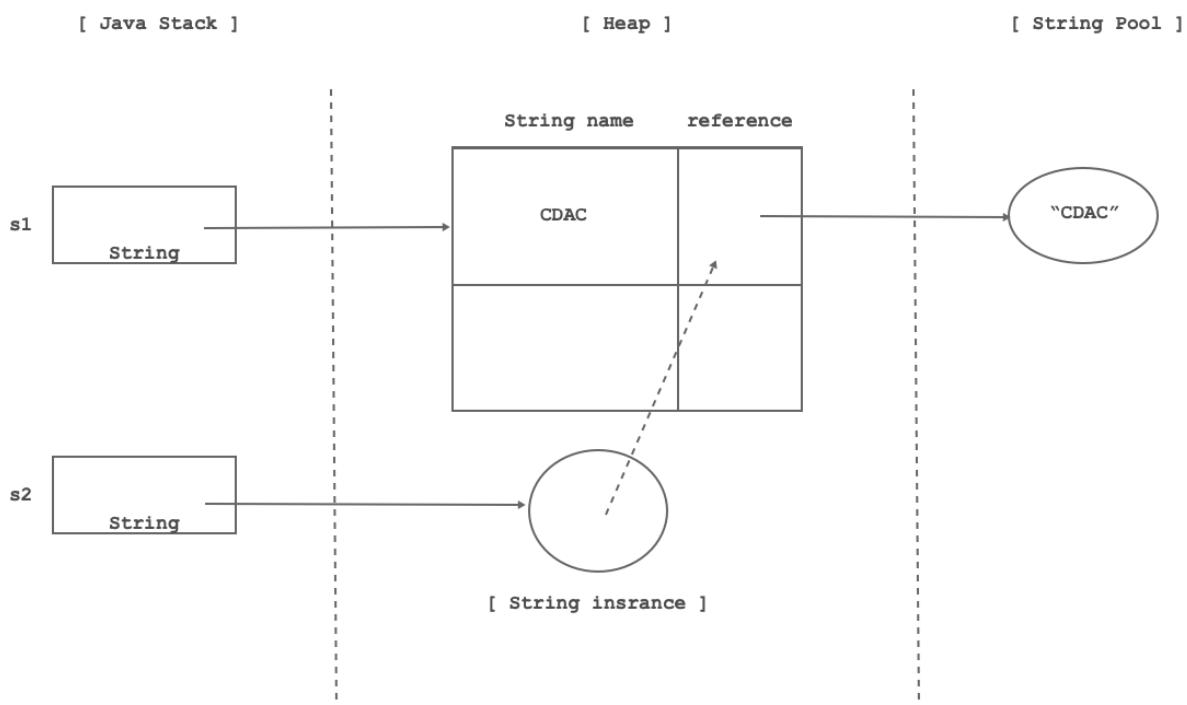
public static void main(String[] args){
    String s1 = new String("CDAC");
    String s2 = new String("CDAC");
    if( s1.equals(s2) )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
    //Output: Equal
}

```

```

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

```



```

public static void main(String[] args){
    String s1 = "CDAC";
    String s2 = new String("CDAC");
    if( s1.equals(s2))
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
}

```

```

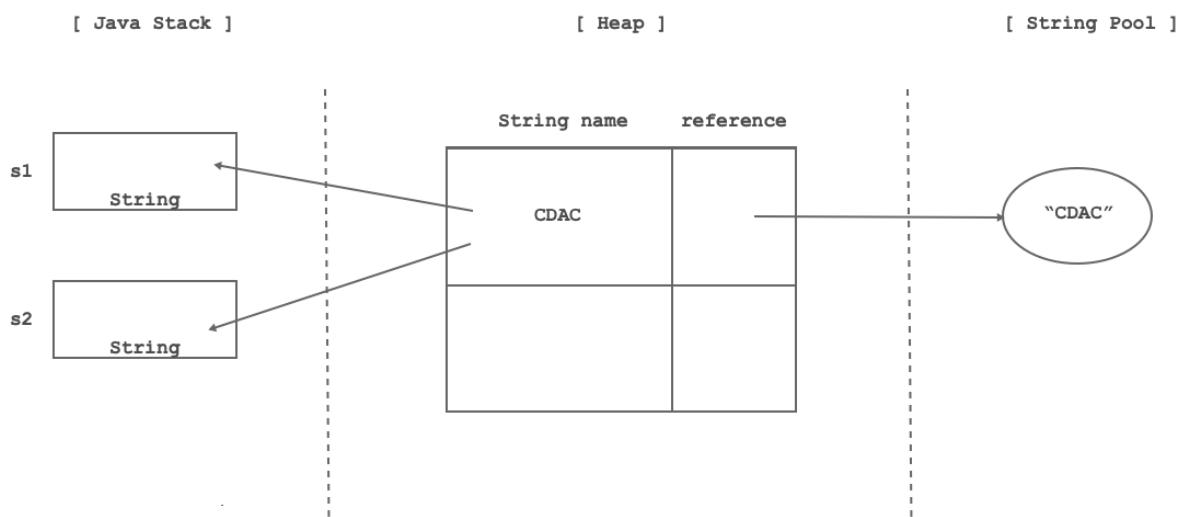
        //Output: Equal
    }
}

```

```

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

```



```

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

```

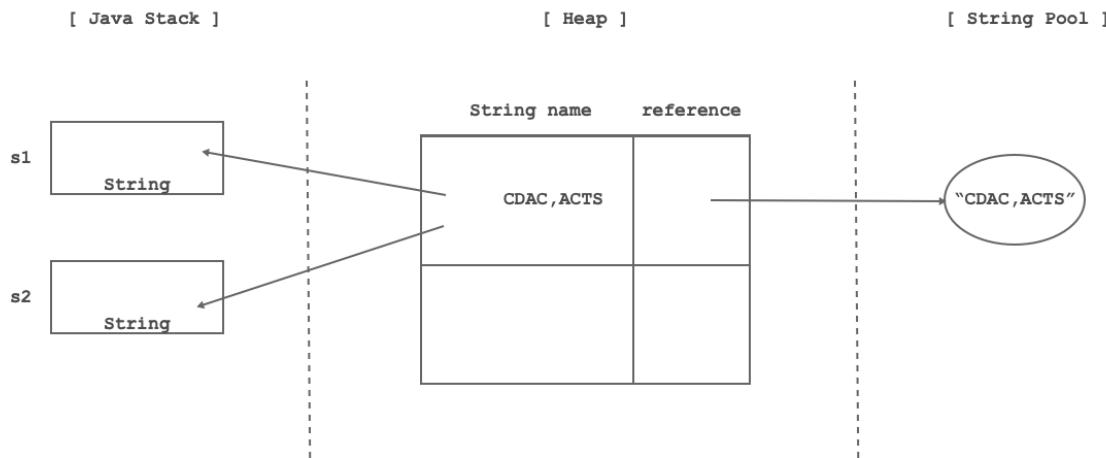
```

public static void main(String[] args) {
    String s1 = "CDAC,"+"ACTS";
    //String s1 = "CDAC,ACTS";
    String s2 = "CDAC,ACTS";
    if( s1 == s2 )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
}

```

```
//Output: Equal
}
```

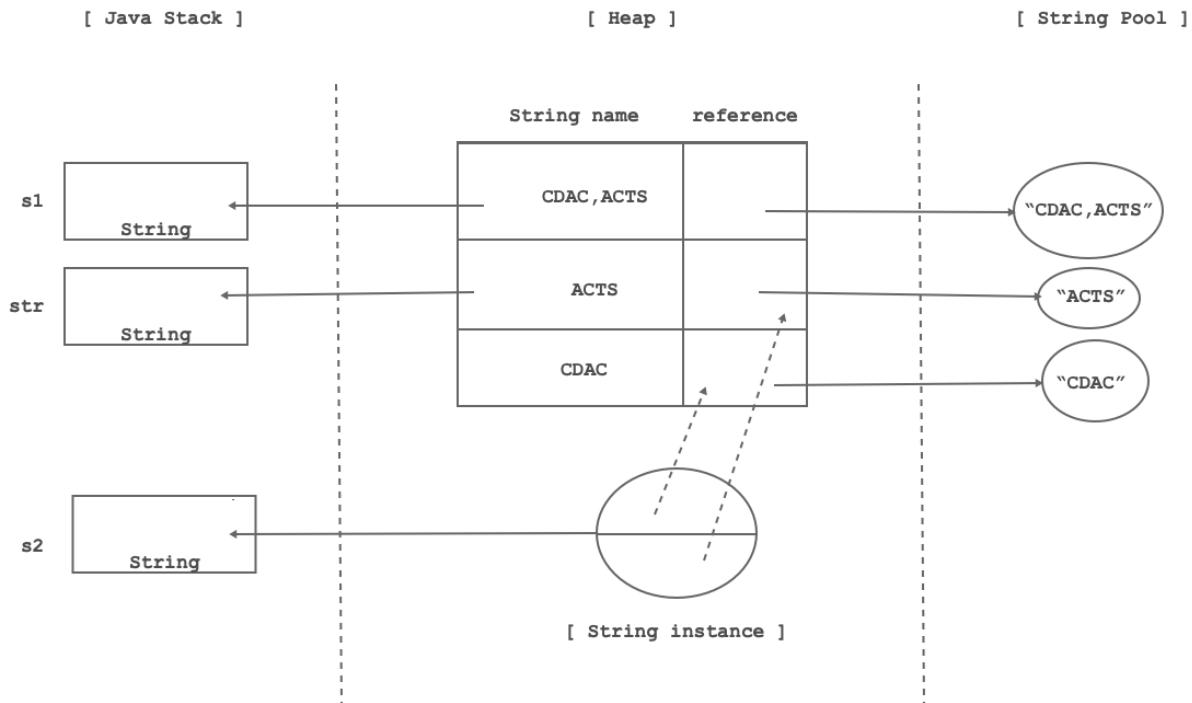
- Constant expressions always gets evaluated at compile time. Hence "CDAC,"+"ACTS" will be considered as "CDAC,ACTS" at compile time.



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

```
public static void main(String[] args) {
    String s1 = "CDAC,ACTS";
    String str = "ACTS";
    String s2 = "CDAC,"+str;

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



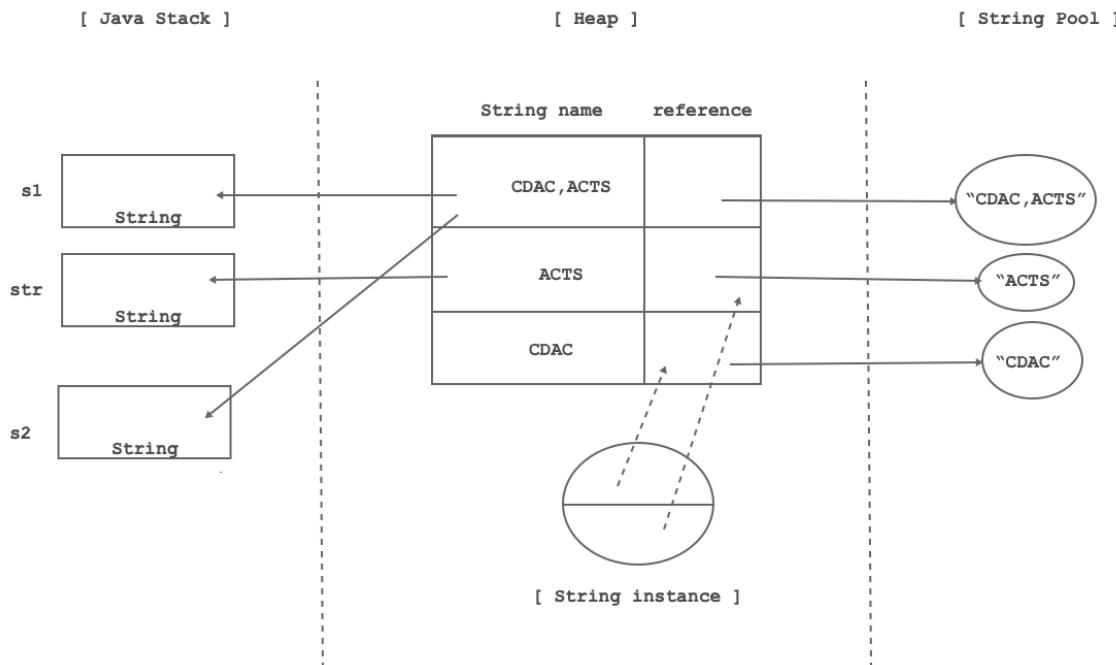
```
public static void main(String[] args) {
    String s1 = "CDAC,ACTS";
    String str = "ACTS";
    String s2 = "CDAC,"+str;

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

```
public static void main(String[] args) {
    String s1 = "CDAC,ACTS";
    String str = "ACTS";
    String s2 = ( "CDAC,"+str ).intern();

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

- `intern()` method returns reference of the String from String pool.



```

public static void main(String[] args) {
    String s1 = "CDAC,ACTS";
    String str = "ACTS";
    String s2 = ( "CDAC,"+str ).intern();

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

```

## A Strategy for Defining Immutable Objects

- Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
- Make all fields final and private.
- Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final.
- If the instance fields include references to mutable objects, don't allow those objects to be changed
- Reference: <https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>

## StringBuffer versus StringBuilder

- StringBuffer and StringBuilder are final classes declared in `java.lang` packages.
- StringBuffer and StringBuilder are used to create mutable String instances.
- StringBuffer and StringBuilder class do not override `equals` and `hashCode()` method.
- To create instance of StringBuffer and StringBuilder, we must use new operator.
- StringBuffer is thread-safe/synchronized but StringBuilder is non thread-safe / unsynchronized.

- Since StringBuffer is synchronized, it is slower in performance. Since StringBuilder is unsynchronized, it is faster in performance.
- Instantiation

```
StringBuffer sb = new StringBuffer("CDAC"); //OK
StringBuilder sb = new StringBuilder("CDAC"); //OK
```

```
public static void main(String[] args) {
    StringBuffer sb1 = new StringBuffer("CDAC");
    StringBuffer sb2 = new StringBuffer("CDAC");
    if( sb1 == sb2 )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
    //Output: Not Equal
}
```

```
public static void main(String[] args) {
    StringBuffer sb1 = new StringBuffer("CDAC");
    StringBuffer sb2 = new StringBuffer("CDAC");
    if( sb1.equals(sb2) )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
    //Output: Not Equal
}
```

```
public static void main(String[] args) {
    String s1 = new String("CDAC");
    StringBuffer sb2 = new StringBuffer("CDAC");
    if( s1.equals(sb2) )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
    //Output: Not Equal
}
```

- Reference: [https://learn.microsoft.com/en-us/previous-versions/windows/desktop/ms762271\(v=vs.85\)](https://learn.microsoft.com/en-us/previous-versions/windows/desktop/ms762271(v=vs.85))

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

```
StringBuffer result = new StringBuffer();
Scanner sc = new Scanner(new File( "books.xml"));
while( sc.hasNextLine() ) {
    String line = sc.nextLine();
    result.append( line );
}
System.out.println( result );
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
}
```

```
public static void main(String[] args) {
    StringBuilder sb1 = new StringBuilder("CDAC");
    StringBuilder sb2 = new StringBuilder("CDAC");
    if( sb1 == sb2 )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
//Output: Not Equal
}
```

```
public static void main(String[] args) {
    StringBuilder sb1 = new StringBuilder("CDAC");
    StringBuilder sb2 = new StringBuilder("CDAC");
    if( sb1.equals(sb2) )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
//Output: Not Equal
}
```

```
public static void main(String[] args) {
    String s1 = new String("CDAC");
    StringBuilder sb2 = new StringBuilder("CDAC");
    if( s1.equals(sb2) )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
//Output: Not Equal
}
```

## StringTokenizer

```
public class StringTokenizer  
extends Object  
implements Enumeration<Object>
```

- Enumeration is interface declared in java.util package
  - boolean hasMoreElements()
  - E nextElement()
- StringTokenizer is a class declared in java.util package.
- The string tokenizer class allows an application to break a string into tokens.

```
www.cdac.in  
www cdac in //Tokens  
. //Delimiter
```

- Constructor Summary
  - public StringTokenizer(String str)
  - public StringTokenizer(String str, String delim)
  - public StringTokenizer(String str, String delim, boolean returnDelims)
- Method Summary
  - public int countTokens()
  - public boolean hasMoreTokens()
  - public String nextToken()
  - public String nextToken()

```
public static void main(String[] args) {  
    String str = "Have a nice day";  
    StringTokenizer stk = new StringTokenizer(str);  
    System.out.println("Count Tokens : "+stk.countTokens());  
//4  
}
```

```
public static void main(String[] args) {  
    String str = "Have a nice day";  
    StringTokenizer stk = new StringTokenizer(str);  
    while( stk.hasMoreElements() ) {  
        String token = (String) stk.nextElement();  
        System.out.println(token);  
    }  
}
```

```
public static void main(String[] args) {  
    String str = "Have a nice day";
```

```

StringTokenizer stk = new StringTokenizer(str);
while( stk.hasMoreTokens() ) {
    String token = stk.nextToken();
    System.out.println(token);
}
}

```

```

public static void main(String[] args) {
    String str = "www.crif.highmark.com";
    String delim = ".";
    StringTokenizer stk = new StringTokenizer(str, delim);
    while( stk.hasMoreTokens() ) {
        String token = stk.nextToken();
        System.out.println(token);
    }
}

```

```

public static void main(String[] args) {
    String str = "https://docs.oracle.com/javase/8/docs/api/";
    String delim =(":/.";
    StringTokenizer stk = new StringTokenizer(str, delim);
    while( stk.hasMoreTokens() ) {
        String token = stk.nextToken();
        System.out.println(token);
    }
}

```

```

public static void main(String[] args) {
    String str = "https://docs.oracle.com/javase/8/docs/api/";
    String delim =(":/.";
    StringTokenizer stk = new StringTokenizer(str, delim, true);
    while( stk.hasMoreTokens() ) {
        String token = stk.nextToken();
        System.out.println(token);
    }
}

```

## Pattern and Matcher

- If we want to validate string then we should use regular expression
- Pattern and Matcher are final classes declared in `java.util.regex` package.
- An instance of the Pattern class represents a regular expression and instances of the Matcher class are used to match character sequences against a given pattern.

```

import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Program {
    public static void main(String[] args) {
        try( Scanner sc = new Scanner(System.in)){
            System.out.print("Enter email : ");
            String email = sc.nextLine();

            Pattern pattern = Pattern.compile("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$");
            Matcher matcher = pattern.matcher( email );
            if( matcher.matches() )
                System.out.println("Email : "+email);
            else
                System.out.println("Invalid Email");
        }
    }
}

```

```

public static final String EMAIL_PATTERN = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$";
public static void main(String[] args) {
    try( Scanner sc = new Scanner(System.in)){
        System.out.print("Enter email : ");
        String email = sc.nextLine();
        if( Pattern.matches(EMAIL_PATTERN, email) )
            System.out.println("Email : "+email);
        else
            System.out.println("Invalid Email");
    }
}

```

```

public static final String EMAIL_PATTERN = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$";
public static void main(String[] args) {
    try( Scanner sc = new Scanner(System.in)){
        System.out.print("Enter email : ");
        String email = sc.nextLine();
        if( email.matches(EMAIL_PATTERN) )
            System.out.println("Email : "+email);
        else
            System.out.println("Invalid Email");
    }
}

```

```

class Validator{
    public static final String NAME_PATTERN= "^[A-Za-z]+(\s[A-Za-z]+)*$";
    public static boolean validateName( String name ) {
        return name.matches(NAME_PATTERN);
    }
    public static final String EMAIL_PATTERN= "^[a-zA-Z0-9._%+-]+@[a-zA-
Z0-9.-]+\.[a-zA-Z]{2,}$";
    public static boolean validateEmail( String email ) {
        return email.matches(EMAIL_PATTERN);
    }
    public static final String PHONE_NUMBER_PATTERN= "^\\d{10}$";
    public static boolean validatePhoneNumber( String phoneNumber ) {
        return phoneNumber.matches(PHONE_NUMBER_PATTERN);
    }
}
public class Program {
    public static void main(String[] args) {
        try( Scanner sc = new Scanner(System.in)){
            System.out.print("Enter email : ");
            String email = sc.nextLine();
            if( Validator.validateEmail(email))
                System.out.println("Email : "+email);
            else
                System.out.println("Invalid email");
        }
    }
}

```

## Regular Expression

- regular expression to validate a 10-digit mobile number:

`^\d{10}$`

- A regular expression to validate a full name

`^[A-Za-z]+(\s[A-Za-z]+)*$`

- A regular expression to validate email

`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`

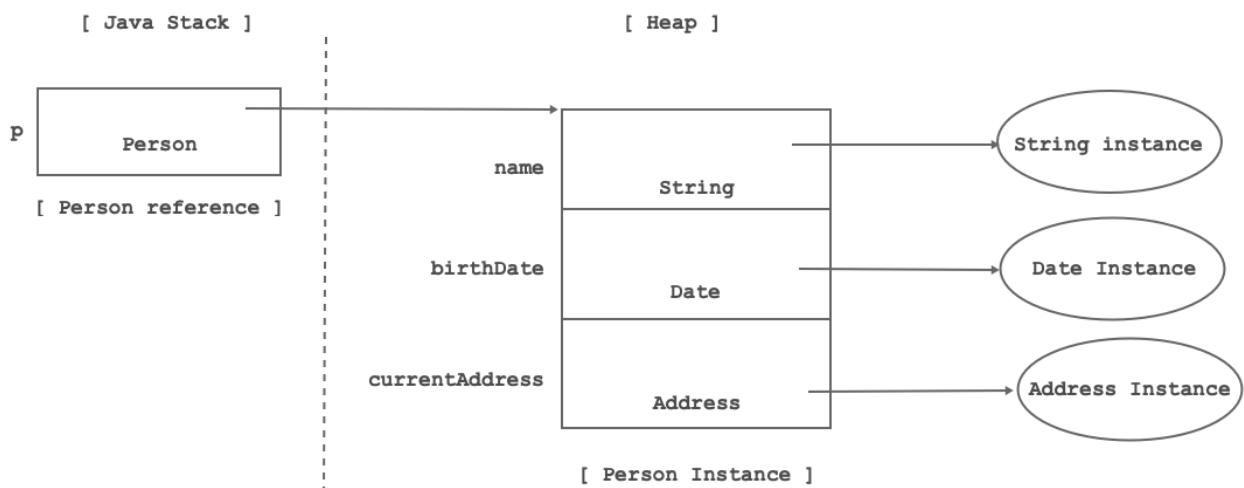
# Day 21

## Collection Framework

- Consider following example:

```
class Date{ }
class Address{ }
class Person{
    private String name = new String();
    private Date birthDate = new Date();
    private Address currentAddress = new Address();
}
class Program{
    public static void main(String[] args) {
        Person p = new Person();
    }
}
```

- In Java, instance do not get space inside another instance. Rather instance contains reference of another instance.



## Library

- In Java, .jar file is a library file.
- It can contain, menifest file, resources, packages.
- Package can contain sub package, interface, class, enum, exception, error, annotation types
- Example: rt.jar

## Framework

- framework = collection of libraries + tools + rules/guidelines
- It is a development platform which contain reusable partial code on the top of it we can develop application.
- Examples:

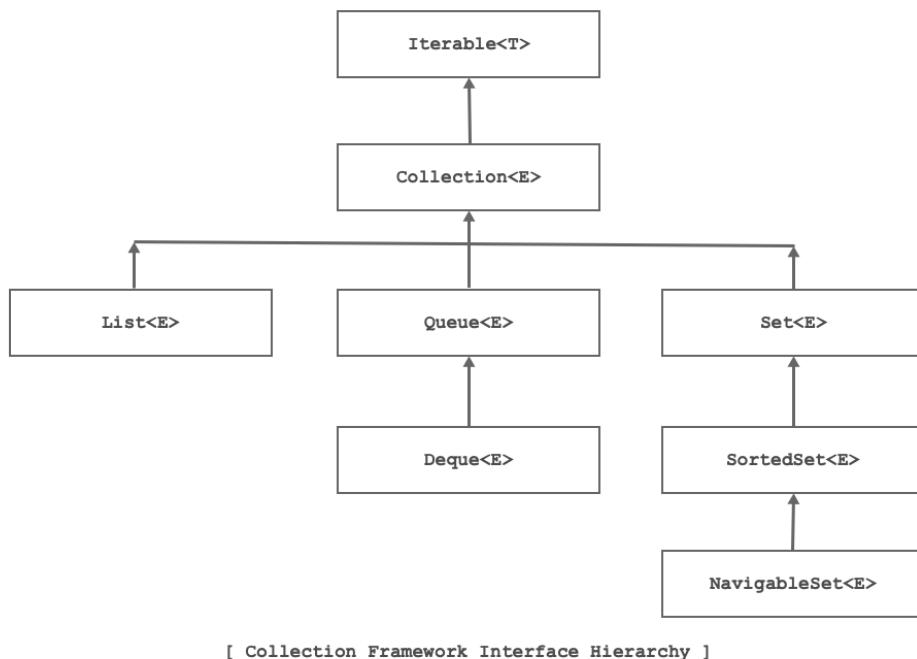
- o JUnit: Unit testing framework which is used to write test case.
- o Apache Log4j2: Logging framework which is used to record activities.
- o AWT/Swing/Java-FX: GUI framework.
- o JNI: Framework to access native code
- o Struts: Readymade MVC based web application framework.
- o Hibernate: ORM based automatic persistence framework
- o Spring: Enterprise framework

## Collection

- Any instance which contains multiple elements is called as collection.
- In java, data structure is also called as collection.

## Collection Framework

- Collection framework is a library of data structure classes on the top of it we can develop Java application.
- In Java, collection framework talk about use not about implementation.
- In Java, when we use collection to store instance then it doesn't contain instance rather it contains reference of the instance.
- To use collection framework, we should import `java.util` package.



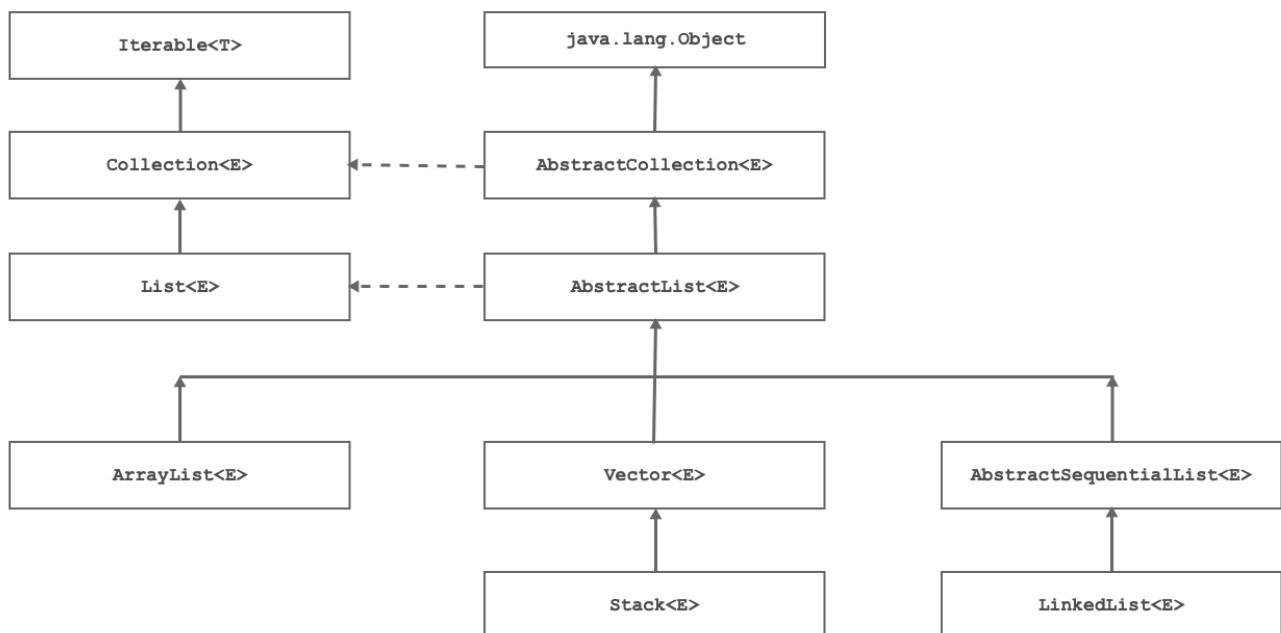
## Iterable

- It is interface declared in `java.lang` package.
- It is introduced in jDK 1.5.
- Implementing this interface allows an object to be the target of the "for-each loop" statement.
- Methods:
  - o `Iterator iterator()`
  - o `default Spliterator spliterator()`
  - o `default void forEach(Consumer<? super T> action)`

## Collection

- Reference: <https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>
- Value stored inside any collection( Array, Stack, Queue, LinkedList etc.) is called as element.
- It is interface declared in java.util package.
- It is root interface in the collection framework interface hierarchy.
- The JDK does not provide any direct implementations of Collection interface.
- Direct implementation classes of Collection interface are AbstractList, AbstractQueue, AbstractSet.
- List, Queue, Set are sub interfaces of java.util.Collection interface.
- Abstract methods of java.util.Collection interface:
  - boolean add(E e)
  - boolean addAll(Collection<? extends E> c)
  - void clear()
  - boolean contains(Object o)
  - boolean containsAll(Collection<?> c)
  - boolean isEmpty()
  - boolean remove(Object o)
  - boolean removeAll(Collection<?> c)
  - boolean retainAll(Collection<?> c)
  - int size()
  - Object[] toArray()
  - T[] toArray(T[] a)
- Default methods of java.util.Collection interface:
  - default Stream stream()
  - default Stream parallelStream()
  - default boolean removeIf(Predicate<? super E> filter)

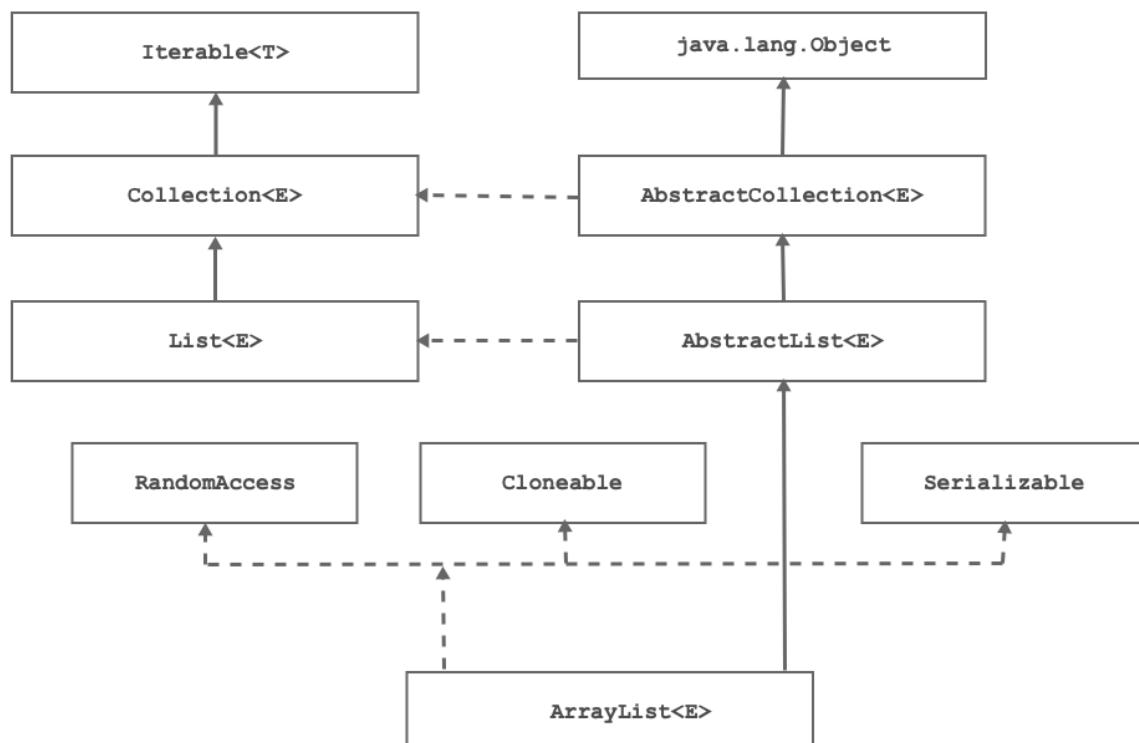
## List



- This interface is a member of the Java Collections Framework and introduced in JDK 1.2

- It is sub interface of Collection interface. It means that all the methods of Collection interface will be inherited into List interface.
- Direct implementation classes of List interfaces are AbstractList, ArrayList, Vector, Stack, LinkedList. These collection classes are called as List collections.
- Inside List collection we can store data in sequential fashion.
- We can store duplicate elements inside any List collection.
- We can store multiple null values inside List collection.
- With the help of integer index, we can access elements from List collection.
- We can traverse elements of any List collection using Iterator as well as ListIterator.
- This interface is a member of the Java Collections Framework.
- Abstract methods of java.util.List interface:
  - void add(int index, E element)
  - boolean addAll(int index, Collection<? extends E> c)
  - E remove(int index)
  - E get(int index)
  - E set(int index, E element)
  - int indexOf(Object o)
  - int lastIndexOf(Object o)
  - ListIterator listIterator()
  - ListIterator listIterator(int index)
  - List subList(int fromIndex, int toIndex)
- Default methods of java.util.List interface:
  - default void sort(Comparator<? super E> c)
  - default void replaceAll(UnaryOperator operator)
- Note: If we want to manage elements of non final type inside any List collection then we should override at least equals methods inside non final type.

## ArrayList



- Array is collection of fixed elements. ArrayList is resizable array.
- Implementation of ArrayList is based of array.
- ArrayList is List collection.
- Since ArrayList is List collection we can store elements sequentially.
- Since ArrayList is List collection, we can store duplicate elements as well as null elements inside ArrayList.
- Since ArrayList is List collection, we can access its elements using integer index.
- Since ArrayList is List collection, we can traverse its elements using Iterator as well as ListIterator.
- ArrayList implementation is unsynchronized. Using Collections.synchronizedList() method we can make it synchronized.
- If ArrayList is full then its capacity gets increased by half of existing capacity.
- This class is a member of the Java Collections Framework and introduced in JDK 1.2.
- Constructor Summary of ArrayList class:
  - public ArrayList()

```
ArrayList<Intger> list = new ArrayList();
```

- public ArrayList(int initialCapacity)

```
ArrayList<Intger> list = new ArrayList( 15 );
```

- public ArrayList(Collection<? extends E> c)

```
Collection<Integer> c = new ArrayList<>();
c.add( 10 );
c.add( 20 );
c.add( 30 );
```

```
ArrayList<Integer> list = new ArrayList<>( c );
```

- Method Summary of ArrayList class:
  - public void ensureCapacity(int minCapacity)
  - protected void removeRange(int fromIndex, int toIndex)
  - public void trimToSize()
- Instantiation:

```

public static void main(String[] args){
    Collection<Integer> collection = new ArrayList<>(); //OK:
Upcasting
    List<Integer> list = new ArrayList<>(); //OK: Upcasting
    ArrayList<Integer> arrayList = new ArrayList<>(); //OK
}

```

- How to add single element inside ArrayList?

```

public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(40);
    list.add(50);
    list.add(2, 30);
    System.out.println( list.toString());
}

```

```

public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
}
public static void main(String[] args) {
    List<Integer> list = Program.getList();
    System.out.println( list.toString()); // [10, 20, 30, 40, 50]
}

```

```

public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
}
public static void main(String[] args) {
    List<Integer> list = Program.getList();
    Integer element = null;
    for( int index = 0; index < list.size(); ++ index ) {

```

```

        element = list.get( index );
        System.out.println(element);
    }
}

```

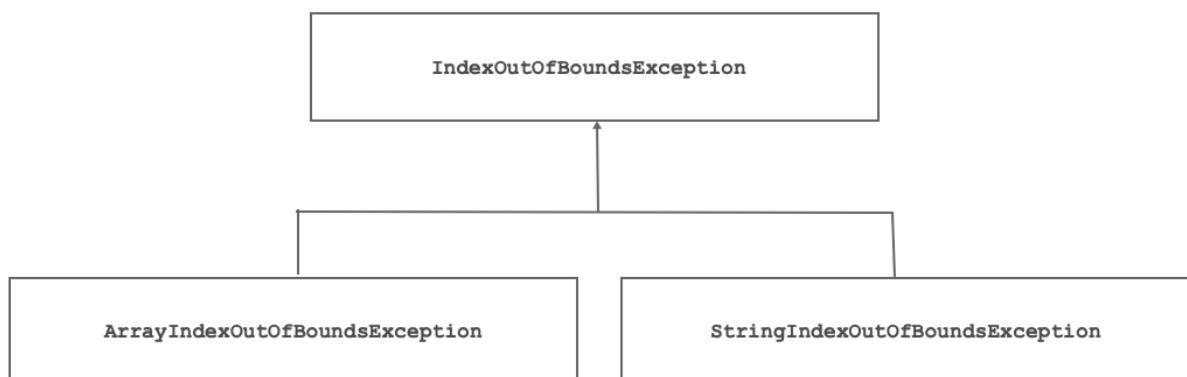
```

public static List<Integer> getList( ){
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
list.add(40);
list.add(50);
return list;
}
public static void main(String[] args) {
    int[] arr = new int[ ] { 10, 20, 30 };
    //int element = arr[ arr.length ];
//ArrayIndexOutOfBoundsException

String str = "CDAC";
//char ch = str.charAt(str.length());
//StringIndexOutOfBoundsException

List<Integer> list = Program.getList();
Integer element = list.get( list.size() );
//IndexOutOfBoundsException
}

```



```

public static List<Integer> getList( ){
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
list.add(40);

```

```
        list.add(50);
        return list;
    }
    public static void main(String[] args) {
        List<Integer> list = Program.getList();
        Integer element = null;
        Iterator<Integer> itr = list.iterator();
        while( itr.hasNext() ) {
            element = itr.next();
            System.out.println(element);
        }
    }
}
```

```
public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
}
public static void main(String[] args) {
    List<Integer> list = Program.getList();
    for( Integer element : list )
        System.out.println( element );
}
```

```
public static void main(String[] args) {
    List<Integer> list = Program.getList();
    /* Consumer<Integer> action = System.out::println;
    list.forEach(action); */
    list.forEach( System.out::println );
}
```

```
public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
}
public static void main(String[] args) {
    List<Integer> list = Program.getList();
    ListIterator<Integer> itr = list.listIterator();
```

```

Integer element = null;
while( itr.hasNext() ) {
    element = itr.next();
    System.out.print( element+" " );
}
System.out.println();
while( itr.hasPrevious() ) {
    element = itr.previous();
    System.out.print( element+" " );
}
}

```

```

//Object[] elementData;
private static int capacity(List<Integer> list) throws Exception{
    Class<?> c = list.getClass();
    Field field = c.getDeclaredField("elementData");
    field.setAccessible(true);
    Object[] elementData = (Object[]) field.get(list);
    return elementData.length;
}
public static void main(String[] args) {
    try {
        List<Integer> list = Program.getList();
        System.out.println("Size : "+list.size()); //5

        int capacity = Program.capacity( list );
        System.out.println("Capacity : "+capacity);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

- How to add multiple elements inside ArrayList?

```

public static void main(String[] args){
    Collection<Integer> collection = new ArrayList<>();
    collection.add(30);
    collection.add(40);
    collection.add(50);

    //List<Integer> list = new ArrayList<>( collection ); //OK
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.addAll(collection);
    System.out.println(list);
}

```

```

public static void main(String[] args) {
    Collection<Integer> collection = new ArrayList<>();
    collection.add(30);
    collection.add(40);
    collection.add(50);

    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(60);
    list.add(70);
    list.addAll(2, collection);
    System.out.println(list);
}

```

- How will you search single element inside ArrayList?

```

public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    for( int count = 1; count <= 10; ++ count )
        list.add( count * 10 );
    return list;
}

public static void main(String[] args) {
    List<Integer> list = Program.getList(); // [10, 20, 30, 40, 50, 60,
70, 80, 90, 100]
    Integer key = new Integer(500);
    if( list.contains(key) ) {
        int index = list.indexOf(key);
        System.out.println( key+" found at index : "+index );
    }else
        System.out.println(key+" not found.");
}

```

- How will you search and remove multiple elements

```

public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    for( int count = 1; count <= 10; ++ count )
        list.add( count * 10 );
    return list;
}

public static void main(String[] args) {
    List<Integer> list = Program.getList(); // [10, 20, 30, 40, 50, 60,
70, 80, 90, 100]
    Collection<Integer> keys = new ArrayList<>();
    keys.add(30);
    keys.add(50);
    keys.add(70);
}

```

```

if( list.containsAll(keys)) {
    list.removeAll(keys); // [10, 20, 40, 60, 80, 90, 100]
    //list.addAll(keys); // [30, 50, 70]
    System.out.println( list );
}else
    System.out.println(keys+" not found.");
}

```

- How will you search and remove single element from ArrayList?

```

public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    for( int count = 1; count <= 10; ++ count )
        list.add( count * 10 );
    return list;
}
public static void main(String[] args) {
    List<Integer> list = Program.getList(); // [10, 20, 30, 40, 50, 60,
70, 80, 90, 100]
    Integer key = new Integer(50);
    if( list.contains(key)) {
        //list.remove(key);
        int index = list.indexOf(key);
        list.remove(index);
        System.out.println( list ); // [10, 20, 30, 40, 60, 70, 80, 90,
100]
    }else
        System.out.println(key+" not found.");
}

```

- How will you sort ArrayList?

```

public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(50);
    list.add(10);
    list.add(30);
    list.add(20);
    list.add(40);

    System.out.println(list); // [50, 10, 30, 20, 40]
    //Collections.sort( list );
    list.sort(null);
    System.out.println(list); // [10, 20, 30, 40, 50]
}

```

- How will you convert ArrayList into array?

```

public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(50);
    list.add(10);
    list.add(30);
    list.add(20);
    list.add(40);

    //Object[] arr = list.toArray();

    Integer[] arr = new Integer[ list.size() ];
    list.toArray(arr);

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

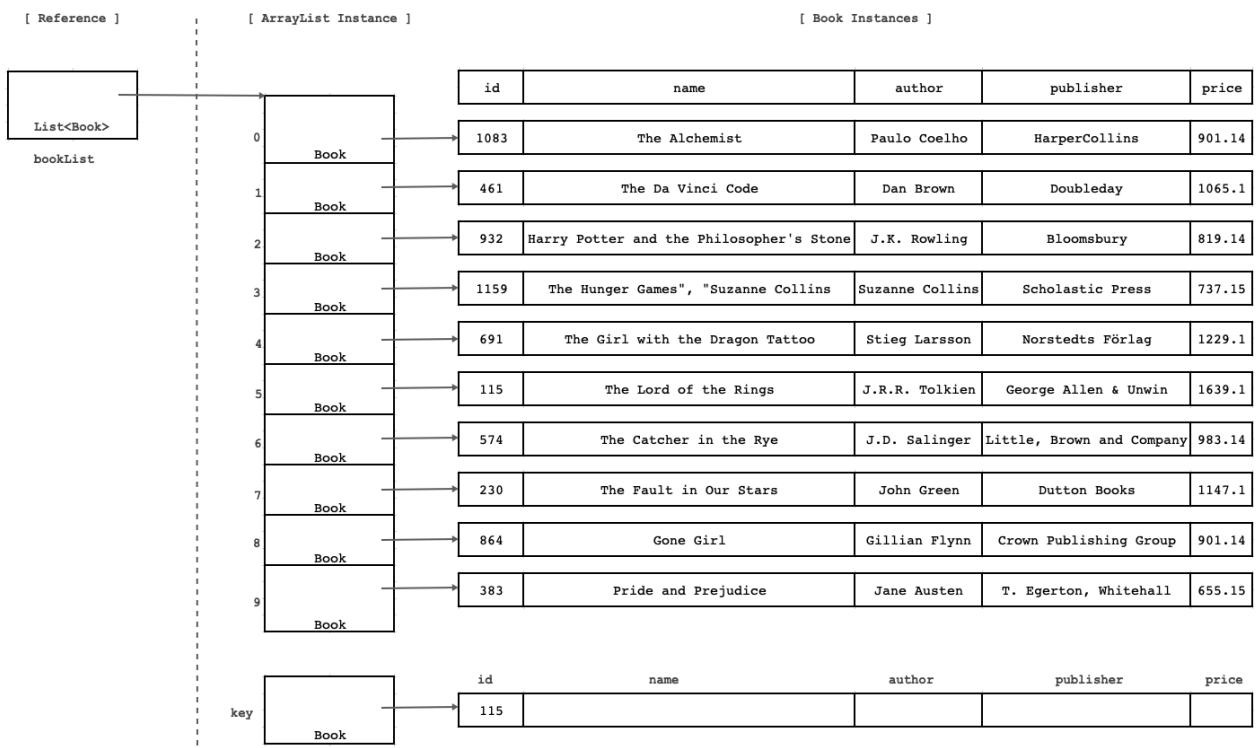
```

- Using Arrays.asList() method

```

public static void main(String[] args) {
    List<Integer> list = Arrays.asList(10, 20, 30, 40, 50 );
    System.out.println( list.getClass().getName());
    //java.util.Arrays$ArrayList
    System.out.println( list ); // [10, 20, 30, 40, 50]
}

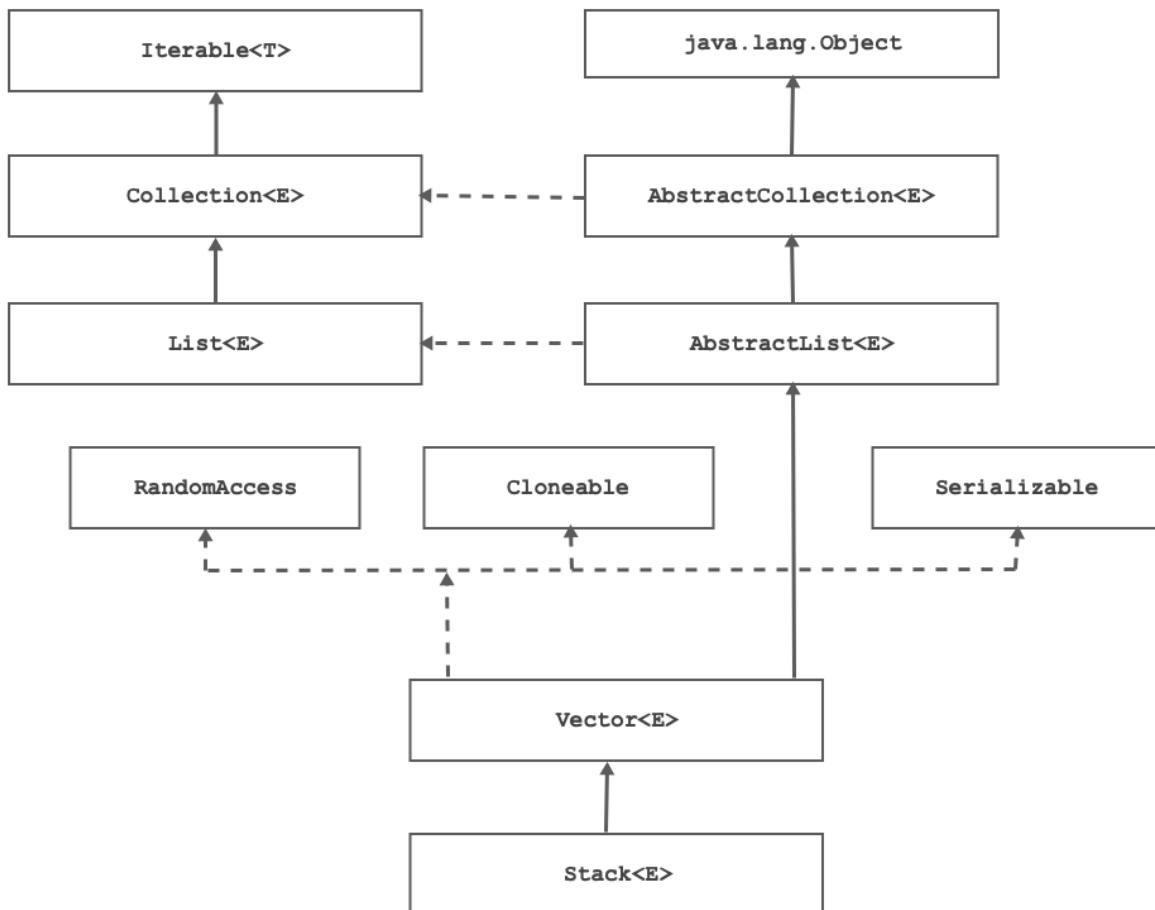
```



Which collection classes are by default synchronized in Java?

- `java.util.Vector`
- `java.util.Stack`
- `java.util.Hashtable`
- `java.util.Properties`

### Vector



- `Vector` is a class declared in `java.util` package.
- `Vector` is `List` collection whose implementation is based on array.
- Since `Vector` is `List` collection, it is ordered/sequential collection.
- Since `Vector` is `List` collection, it can contain duplicate elements as well as null elements
- Since `Vector` is `List` collection, we can traverse its elements using `Iterator` as well as `ListIterator`.
- We can traverse elements of `Vector` using `java.util.Enumeration`, `java.util.Iterator` as well as `ListIterator`.
- `Vector` is Synchronized collection.
- Default capacity is 10 elements. Once `Vector` is full it gets double capacity.
- It was introduced in JDK 1.0. Hence it is also called as legacy class.

### Traversing using Enumeration

- `Enumeration` is interface declared in `java.util` package.
- It was introduced in JDK 1.0.
- Methods of `Enumeration` I/F:

- o boolean hasMoreElements()
- o E nextElement()
- Using Enumeration we can traverse limited collections. For Example: Vector, Hashtable etc.
- Using Enumeration, we can traverse collection only forward direction. During traversing we can not add, set or remove elements from underlying collection.

```

public static void main(String[] args){
    Vector<Integer> v = new Vector<>();
    for( int count = 1; count <= 10; ++ count )
        v.add(count);

    Integer element = null;
    Enumeration<Integer> e = v.elements() ;
    while( e.hasMoreElements() ) {
        element = e.nextElement();
        System.out.println(element);
    }
}

```

### Traversing using Iterator

- Iterator is interface declared in java.util package.
- This interface is a member of the Java Collections Framework.
- Methods of Iterator interface:
  - o boolean hasNext()
  - o boolean hasNext()
  - o default void remove()
  - o default void forEachRemaining(Consumer<? super E> action)
- Iterator takes the place of Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:
  - o Iterators allow the caller to remove elements from the underlying collection during the iteration.
  - o Method names have been improved.

```

public static void main(String[] args){
    Vector<Integer> v = new Vector<>();
    for( int count = 1; count <= 10; ++ count )
        v.add(count);

    Integer element = null;
    Iterator<Integer> itr = v.iterator();
    while( itr.hasNext() ) {
        element = itr.next();
        System.out.println(element);
    }
}

```

```
    }  
}
```

## Traversing using ListIterator

- It is subinterface of Iterator interface which is declared in java.util package.
- We can use it to traverse only List collections( ArrayList, Vector, Stack, LinkedList etc.)
- We can use ListIterator to traverse collection in bidirection. During traversing, using iterator we can add/set/remove element from collection.
- Method Summary
  - void add(E e)
  - void set(E e)
  - void remove()
  - boolean hasNext()
  - E next()
  - boolean hasPrevious()
  - E previous()
  - int nextIndex()
  - int previousIndex()
- This interface is a member of the Java Collections Framework.
- It is introduced in JDK 1.2

```
public static void main(String[] args){  
    Vector<Integer> v = new Vector<>();  
    for( int count = 1; count <= 10; ++ count )  
        v.add(count);  
  
    Integer element = null;  
    ListIterator<Integer> itr = v.listIterator();  
    //ListIterator<Integer> itr = v.listIterator( 4 );  
    //ListIterator<Integer> itr = v.listIterator( v.size() );  
    while( itr.hasNext() ) {  
        element = itr.next();  
        System.out.print(element+ " ");  
    }  
    System.out.println();  
    while( itr.hasPrevious() ) {  
        element = itr.previous();  
        System.out.print(element+ " ");  
    }  
}
```

## What is the difference between Enumeration and Iterator

- Using Enumeration we can traverse collection only in forward direction. During traversing, using Enumeration, we can not add/set/remove element from underlying Collection. Using Iterator we can traverse collection only in forward direction. During traversing, using Iterator, we can not add/set element but we can remove element from underlying Collection.
- We can use Enumeration for few Collections only but we can use Iterator for any collection that implements Iterable interface.
- Enumeration method names are long but Iterator methods names are short.
- Enumeration was introduced in JDK 1.0 whereas Iterator was introduced in JDK1.2.

#### **What is the difference between Iterator and ListIterator**

- Using Iterator we can traverse any Collection which implements Iterable interface but Using ListIterator we can traverse any List collection.
- Using Iterator we can traverse collection only in forward direction whereas using ListIterator we can traverse collection in bidirection.
- During traversing, using iterator, we can not add/set element from underlying collection but we can remove element. During traversing, using ListIterator, we can add/set/remove element from underlying collection.

#### **What do you know about fail-fast and not fail-fast( i.e. fail-safe) iterator**

##### **or What do you know about ConcurrentModificationException?**

- During traversing, without iterator, if we try to make changes in underlying collection and if we get ConcurrentModificationException then such iterator is called as fail-fast Iterator.

```

public static void main(String[] args){
    Vector<Integer> v = new Vector<>();
    for( int count = 1; count <= 10; ++ count )
        v.add(count);

    Integer element = null;
    Iterator<Integer> itr = v.iterator();
    while( itr.hasNext() ) {
        element = itr.next();
        System.out.println(element);
        if( element == 10 )
            v.add(11); //ConcurrentModificationException
    }
}

```

- During traversing, without iterator, if we try to make changes in underlying collection and if we do not get ConcurrentModificationException then such iterator is called as fail-safe Iterator. Such iterators works by creating copy of the Collection.

```

public static void main1(String[] args){
    Vector<Integer> v = new Vector<>();

```

```

        for( int count = 1; count <= 10; ++ count )
            v.add(count);

        Integer element = null;
        Enumeration<Integer> e = v.elements() ;
        while( e.hasMoreElements() ) {
            element = e.nextElement();
            System.out.println(element);
            if( element == 10 )
                v.add(11); //OK
        }
        System.out.println(v);
    }
}

```

### What is the difference between ArrayList and Vector?

- Synchronization: ArrayList collection is unsynchronized whereas Vector is collection synchronized.
- Capacity: In case of arrayList, capacity gets increased by half of existing capacity. In case of vector, capacity get increased by existing capacity.
- Traversing: We can traverse elements of ArrayList using Iterator and ListIterator whereas we can traverse elements of vector using Enumeration, Iterator and ListIterator.
- Legacy: ArrayList collection is introduced in JDK 1.2 whereas Vector collection is introduced in JDK 1.0.

### Stack

- It is sub class of java.util.Vector class.
- IN Java, Stack is synchronized collection.
- If we want to perform operations in Last In First Out(LIFO) order/manner then we should use Stack.
- Method Summary Stack:
  - public boolean empty()
  - public E push(E item)
  - public E peek()
  - public E peek()
  - public int search(Object o)

```

public static void main(String[] args) {
    Stack<Integer> stk = new Stack<>();
    stk.push(10);
    stk.push(20);
    stk.push(30);
    stk.push(40);
    stk.push(50);

    Integer element = null;
    while( !stk.empty() ) {
        element = stk.peek();
        System.out.println("Removed element is : "+element);
    }
}

```

```

        stk.pop();
    }
}

```

- If we want, unsynchronized implementation of Stack then we should use Deque implementation

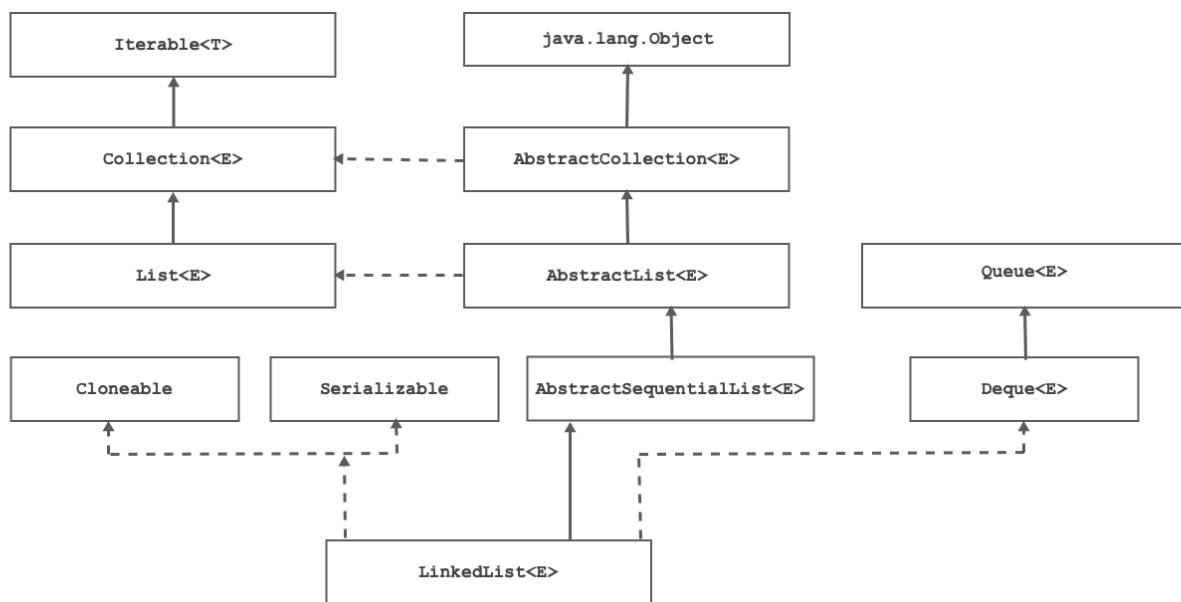
```

public static void main(String[] args) {
    Deque<Integer> stk = new ArrayDeque<>();
    stk.push(10);
    stk.push(20);
    stk.push(30);
    stk.push(40);
    stk.push(50);

    Integer element = null;
    while( !stk.isEmpty() ) {
        element = stk.peek();
        System.out.println("Removed element is : " +element);
        stk.pop();
    }
}

```

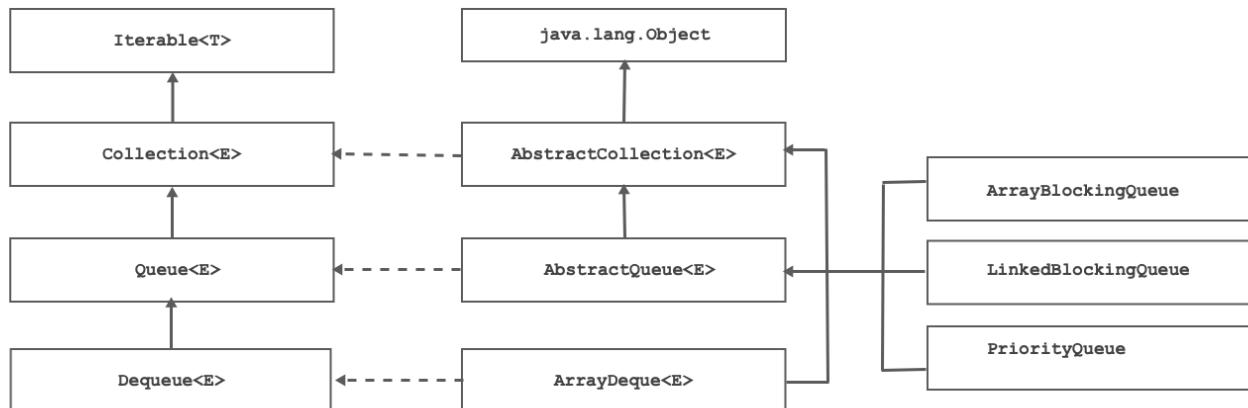
## LinkedList



- It is a class declared in `java.util` package. Its implementation is based on Doubly Linked List.
- `LinkedList` class implements `List` as well as `Deque` interface.
- Since it is `List` collection, It stored elements in sequential manner.
- Since it is `List` collection, It can contain duplicate elements as well as null elements
- Since it is `List` collection, We can access its elements using integer index.
- Since it is `List` collection, We can traverse its elements using `Iterator` and `ListIterator`
- `LinkedList` collection is unsynchronized. Using `Collections.synchronizedList()` method we can make it synchronized.

- This class is a member of the Java Collections Framework. It is introduced in JDK 1.2

## Queue



- It is sub interface of Collection interface.
- If we want to perform operations in First In First Out order then we should use Queue implementation.
- This interface is a member of the Java Collections Framework.
- It is introduced in JDK 1.5
- Method Summary of Queue interface:
  - boolean add(E e)
  - boolean offer(E e)
  - E remove()
  - E poll()
  - E element()
  - E peek()
- Consider following code:

```

public static void main(String[] args) {
    Queue<Integer> que = new ArrayDeque<>();
    que.add(10);
    que.add(20);
    que.add(30);
    que.add(40);
    que.add(50);
    //que.add(null);    //Not Allowed

    Integer element = null;
    while( !que.isEmpty() ) {
        element = que.element();
        System.out.println("Removed element is : "+element);
        que.remove();
    }
}
  
```

- Consider following code:

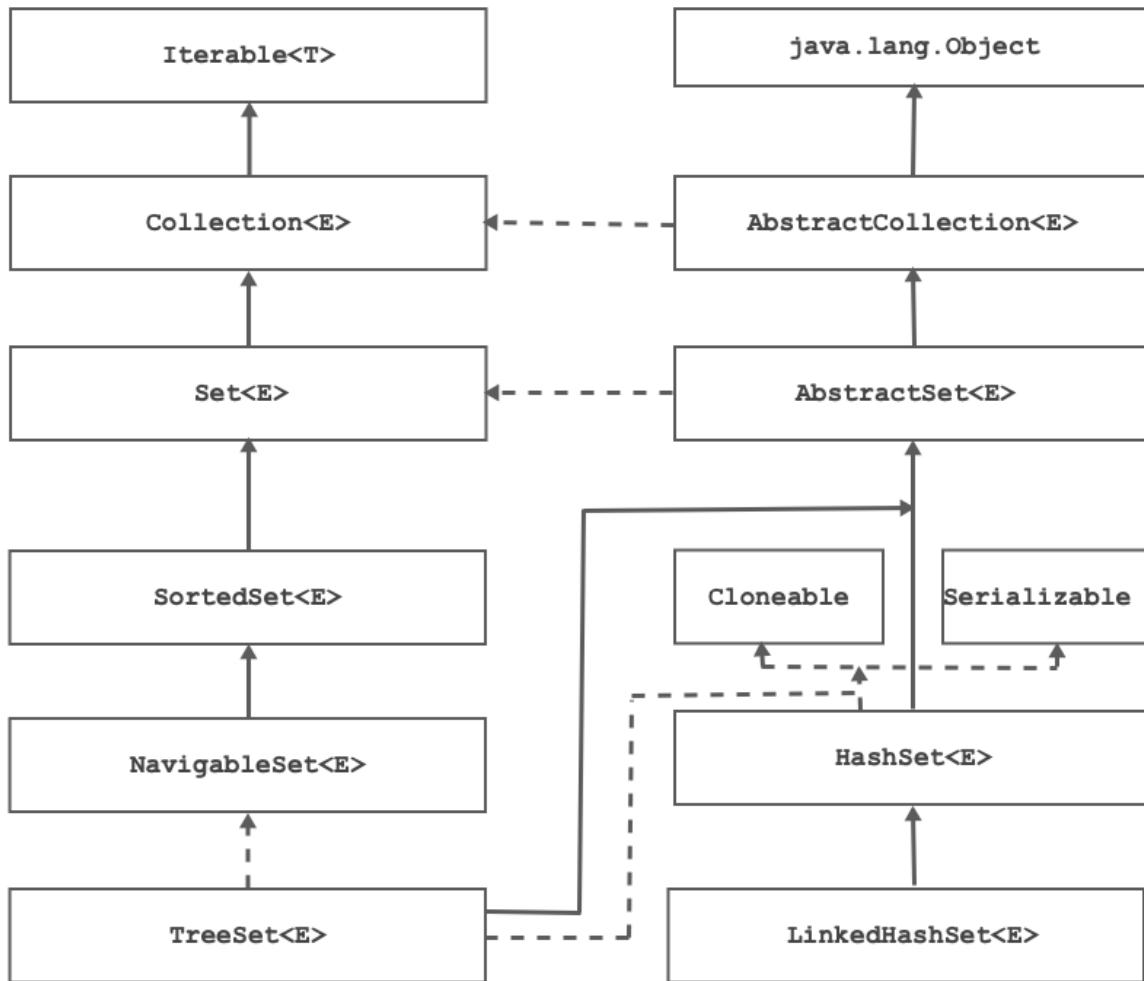
```
public static void main(String[] args) {  
    Queue<Integer> que = new ArrayDeque<>();  
    que.offer(10);  
    que.offer(20);  
    que.offer(30);  
    que.offer(40);  
    que.offer(50);  
    //que.offer(null); //Not Allowed  
  
    Integer element = null;  
    while( !que.isEmpty() ) {  
        element = que.peek();  
        System.out.println("Removed element is : "+element);  
        que.poll();  
    }  
}
```

## Deque

- It is sub interface of Queue interface.
- The name deque is short for "double ended queue" and is usually pronounced "deck".
- This interface is a member of the Java Collections Framework.
- It is introduced in JDK 1.6

# Day 22

## Set



- It is sub interface of `java.util.Collection` interface.
- `HashSet`, `LinkedHashSet`, `TreeSet`, `EnumSet` are Set collections.
- Set Collections do not contain duplicate elements.
- This interface is a member of the Java Collections Framework.
- It is introduced in JDK 1.2
- Method names of `Collection` and `Set` interface are same. No new method is added in `Set` interface.

## What is the difference between List and Set?

- `ArrayList`, `Vector`, `LinkedList` are List Collections and `HashSet`, `LinkedHashSet`, `TreeSet` are Set collections
- List collections can contain duplicate elements but Set collections do not contain duplicate elements.
- List collections can contain null elements but not all Set collections contain null elements.
- We can traverse elements of List collection using `ListIterator` as well as `Iterator` but we can we can traverse elements of Set collection using `Iterator` only.
- All List collections are sequential collections but we can not give guarantee of order of elements in Set collection.

## TreeSet

- It is a Set collection.
- It can not contain duplicate elements.

- It can not contain null elements.
- It contains data in sorted order.
- TreeSet implementation is based on TreeMap<K,V>.
- It is unsynchronized collection. Using Collections.synchronizedSortedSet() method we can make it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.2
- Note: If we want to use TreeSet to store elements of non final type then non final type should implement Comparable interface.
- How to create instance of TreeSet

```

public static void main(String[] args) {
    TreeSet<Integer> treeSet = new TreeSet<>();

    Set<Integer> set = new TreeSet<>(); //Upcasting

    Collection<Integer> collection = new TreeSet<>(); //Upcasting
}

```

- How will you add elements inside TreeSet?

```

public static void main(String[] args) {
    Set<Integer> set = new TreeSet<>(); //Upcasting
    set.add(50);
    set.add(10);
    set.add(30);
    set.add(20);
    set.add(40);
    System.out.println( set ); // [10, 20, 30, 40, 50]
}

```

- Can we add duplicate elements inside TreeSet

```

public static void main(String[] args) {
    Set<Integer> set = new TreeSet<>(); //Upcasting
    set.add(50);
    set.add(10);
    set.add(30);
    set.add(20);
    set.add(40);

    set.add(50);

```

```

        set.add(10);
        set.add(30);
        set.add(20);
        set.add(40);

        System.out.println( set ); // [10, 20, 30, 40, 50]
    }
}

```

- Can we add null element inside TreeSet?

```

public static void main(String[] args) {
    Set<Integer> set = new TreeSet<>(); //Upcasting
    set.add(50);
    set.add(10);
    set.add(30);
    set.add(20);
    set.add(40);
    set.add(null); //NullPointerException

    System.out.println( set );
}

```

- Can we convert TreeSet into ArrayList?

```

Set<Integer> set = new TreeSet<>(); //Upcasting
set.add( 10 );
set.add( 20 );
set.add( 30 );
List<Integer> list = new ArrayList( set );

```

## Algorithm

- In Computer science algorithm and data structure are two different branches.
- Data structure describes 2 things:
  - How to organize data inside RAM?
  - Which operations should be used to organize data inside RAM.
- Data structure can be linear / non linear.
- Well defined set of statements that we can use to solve real world common problems is called as algorithms.
- In the context of data structure algorithms can be searching / sorting algorithms.

## Searching

- Searching refers to the process of finding location( index / reference ) of an element in Collection.
- The collection of data may be in array, list, database, or any other data structure.
- There are several searching algorithms, each with its own strengths and weaknesses:

- Linear search
- Binary search
- Hashing
- Interpolation search
- Exponential search

## Linear Search

- Linear search is a simple searching algorithm that sequentially searches each element in a collection until the desired item is found.
- Consider following code:

```
public static int linearSearch(int[] arr, int key) {
    if( arr != null ){
        for (int index = 0; index < arr.length; index++) {
            if (arr[ index ] == key)
                return index;
        }
    }
    return -1;
}
```

- Pros of linear search:
  - Simplicity: Linear search is easy to understand and implement, making it a good choice for small collections.
  - Flexibility: Linear search can be used with any type of collection, including arrays and linked lists.
  - No requirement for sorted data: Unlike some other search algorithms, linear search does not require the data to be sorted.
- Cons of linear search:
  - Time complexity: The worst-case time complexity of linear search is  $O(n)$ , where  $n$  is the number of elements in the collection.
  - Inefficiency with large collections: Linear search is inefficient for large collections, as it requires checking every element in the collection
  - Not suitable for complex data structures: For complex data structures, such as trees or graphs

## Binary search

- Binary search is a searching algorithm used to search for a specific element in a sorted array.
- The basic idea of binary search is to divide the array into halves repeatedly until the desired element is found.
- Consider following example:

```

public static int binarySearch(int[] arr, int key) {
    int left = 0;
    int right = arr.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}

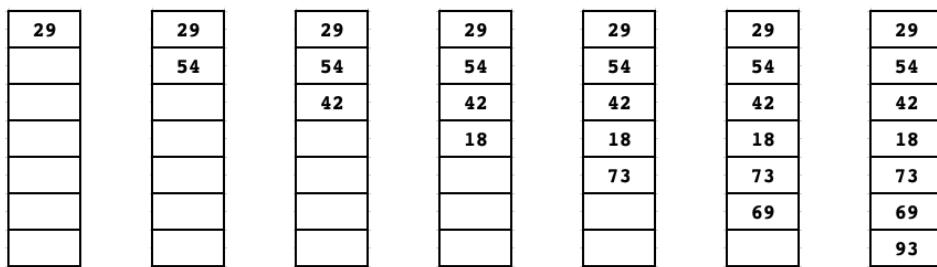
```

- Pros of Binary Search:
  - Efficient: Binary search is an efficient algorithm, with a time complexity of  $O(\log n)$  for a sorted array of size  $n$ . This is much faster than linear search, which has a time complexity of  $O(n)$ .
  - If the array is already sorted, it can be a very efficient way to search for an element.
- Cons of Binary Search:
  - Requires Sorted Array: As mentioned, binary search requires the array to be sorted in advance. If the array is unsorted or frequently updated, then binary search may not be the best choice.
  - Requires Random Access: Binary search requires random access to the array elements, which is not available in some data structures such as linked lists.
  - Limited Applicability: Binary search is limited to searching for an element in a one-dimensional sorted array. It cannot be used to search for an element in a multidimensional array or a database table.

## Hashing

- Consider numbers, 29,54,42,18,73,69,93 and insert it into array.

arr → 29, 54, 42, 18, 73, 69, 93



|    |          |  |
|----|----------|--|
| 29 | arr[ 0 ] | if key = 29 then it requires 1 comparison  |
| 54 | arr[ 1 ] | if key = 54 then it requires 2 comparisons |
| 42 | arr[ 2 ] | if key = 42 then it requires 3 comparisons |
| 18 | arr[ 3 ] | if key = 18 then it requires 4 comparisons |
| 73 | arr[ 4 ] | if key = 73 then it requires 5 comparisons |
| 69 | arr[ 5 ] | if key = 69 then it requires 6 comparisons |
| 93 | arr[ 6 ] | if key = 93 then it requires 7 comparisons |

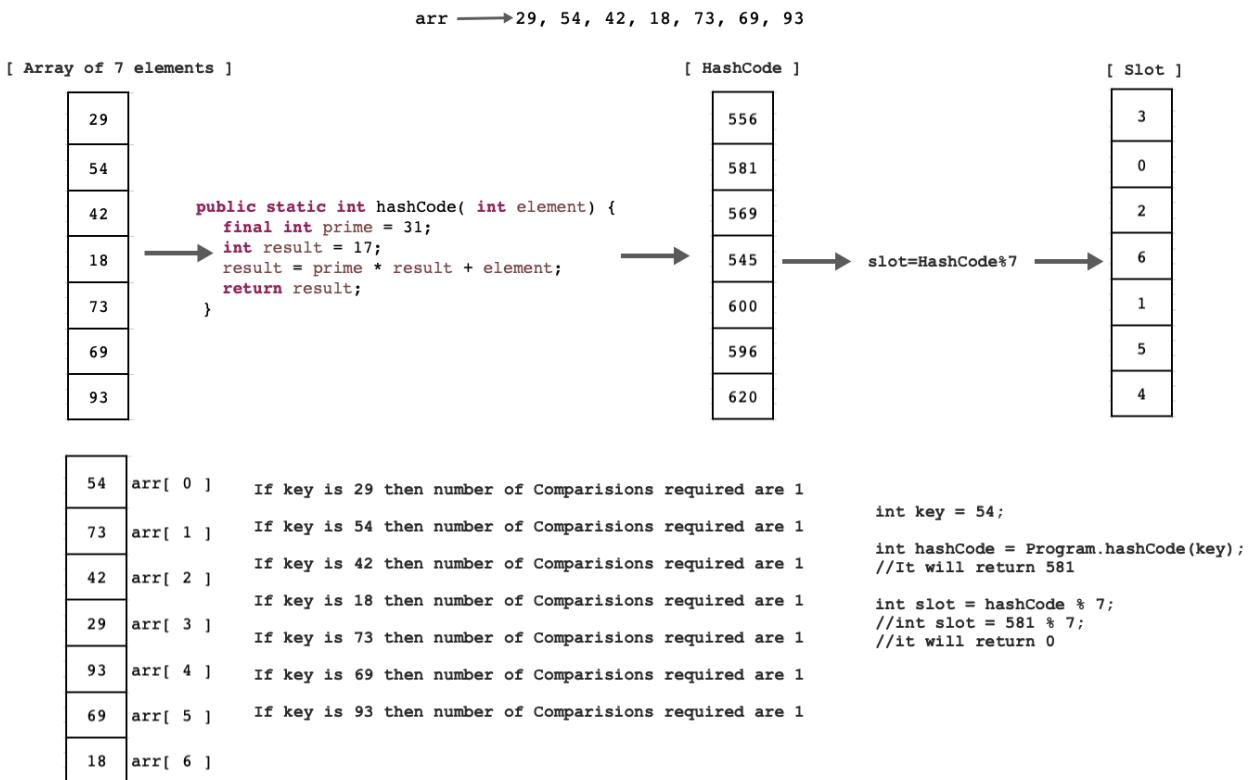
Note: Since we are adding elements sequentially, time required to search every element is different

- Hashing is a technique used to store and retrieve data in a fast and efficient manner.
- Hashing searching is based on hash code.
- HashCode is not an index / address / reference. It is a logical integer number that can be generated by processing state of the instance.
- To generate hash code we should use hash function. Consider following example

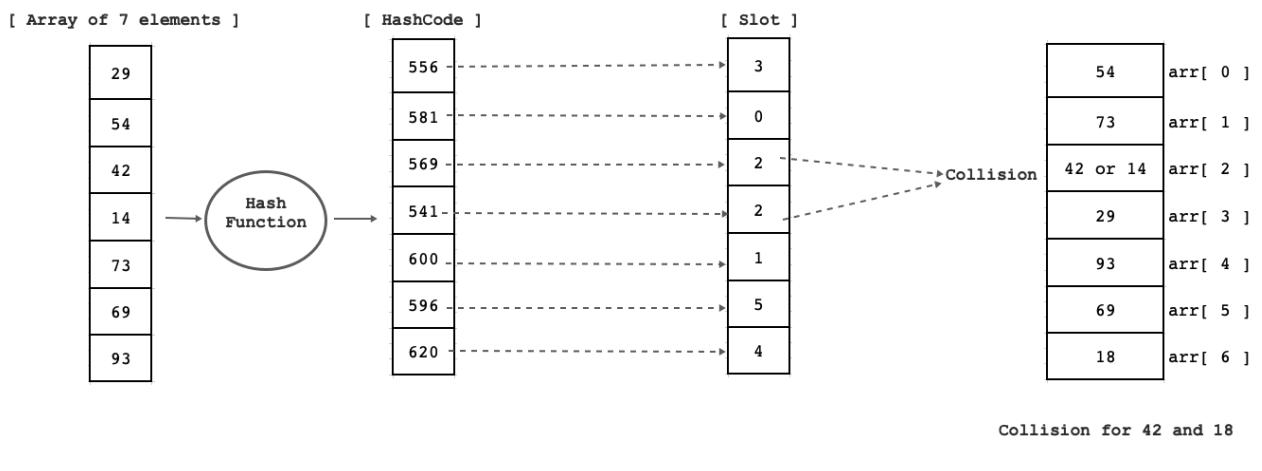
```
public int hashCode( int element ) {
    final int prime = 31;
    int result = 17;
    result = prime * result + element;
    return result;
}
```

- In the context of hashing, a slot refers to a location in the hash table where a key-value pair can be stored.

- In other words, hash code is required to generate index which is called as slot in hashing.



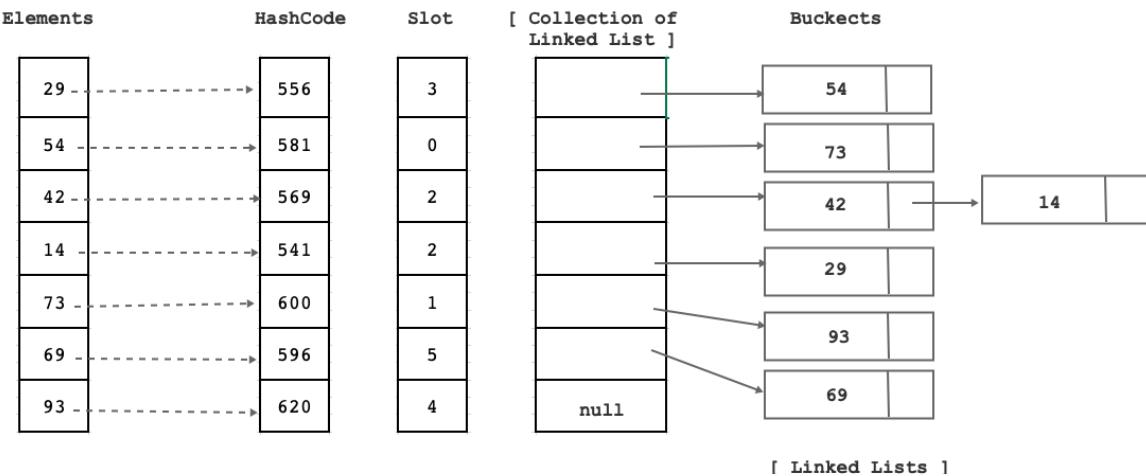
- By Processing state of the instance, if we get same slot then it is called as collision.



- To avoid collision, we should use collision resolution techniques:

- Separate Chaining / Open Hashing
- Open Addressing / Close Hashing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

- Separate Chaining



- To avoid collision, we can maintain one collection(LinkedList/BST) per slot. It is called as bucket.
- Note: If we want to store any element of non primitive type inside Hashcode based collection then we should override equals and hashCode method inside non primitive type.
- Pros of Hashing
  - Fast retrieval: Hashing allows for constant-time retrieval of data, regardless of the size of the data set. This makes it a very efficient searching algorithm.
  - Flexibility: Hashing can be used to search for any type of data, including strings, numbers, and custom objects.
  - Easy to implement
  - Supports dynamic resizing: Hash tables can be resized dynamically to accommodate more data, without having to recreate the entire data structure.
- Cons of Hashing:
  - Hash collisions: Hash collisions occur when two different keys are mapped to the same hash value. This can result in slower lookup times and may require additional processing to resolve the collision.
  - Memory usage: Hash tables can consume a lot of memory, particularly if the data set is large. This can lead to performance issues if the available memory is limited.
  - Hash functions: The efficiency of hashing depends on the quality of the hash function used. Poor hash functions can lead to more collisions, slower retrieval times, and other issues.
- equals and hashCode are non final methods of java.lang.Object class.
- If we do not override equals method then super class's equals method will call. equals method of Object class do not compare state of instances. It compares state of references.
- Consider equals method definition of Object class:

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

```
}
```

- hashCode is non final but native method of java.lang.Object class.

- Syntax:

```
public native int hashCode();
```

- hashCode() method of java.lang.Object class convert memory address of instance into integer value. Hence even though state of the instances are same we get different hashCode. If we want hashCode based on state of the instance then we should override hashCode method inside class.

- Consider implementation:

```
@Override  
public int hashCode() {  
    int prime = 31;  
    int result = 1;  
    result = result * prime + this.empid;  
    return result;  
}
```

## HashSet

- It is hashCode based collection whose implementation is based on Hashtable.
- In HashSet elements get space according its hash code hence It doesn't give any guarantee about order of the elements.
- Since it is Set collection, it doesn't contain duplicate elements.
- HashSet can contain null element.
- It is unsynchronized collection. Using Collections.synchronizedSet() method we can consider it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.2
- If we want to use any element of non final type inside HashSet then non final type should override equals() and hashCode() method.
- Instantiation of HashSet:

```
public static void main(String[] args) {  
    //HashSet<Integer> set = new HashSet<>();
```

```
    Set<Integer> set = new HashSet<>();  
}
```

- How will you add element inside HashSet?

```
public static void main(String[] args) {  
    Set<Integer> set = new HashSet<>();  
    set.add(101);  
    set.add(125);  
    set.add(13);  
    set.add(314);  
    set.add(215);  
    System.out.println(set);      // [101, 215, 314, 125, 13]  
}
```

- Is it possible to store duplicates and null elements?

```
public static void main(String[] args) {  
    Set<Integer> set = new HashSet<>();  
    set.add(101);  
    set.add(125);  
    set.add(13);  
    set.add(314);  
    set.add(215);  
    set.add(null);  
  
    set.add(101);  
    set.add(125);  
    set.add(13);  
    set.add(314);  
    set.add(215);  
    set.add(null);  
    System.out.println(set);      // [null, 101, 215, 314, 125, 13]  
}
```

## LinkedHashSet

- It is hashCode based collection whose implementation is based on Hashtable and LinkedList.
- During traversing it maintains order of elements.
- Since it is Set collection, it doesn't contain duplicate elements.
- HashSet can contain null element.
- It is unsynchronized collection. Using Collections.synchronizedSet() method we can consider it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.4
- If we want to use any element of non final type inside HashSet then non final type should override equals() and hashCode() method.

## Dictionary<K,V>

- Dictionary<K,V> is abstract class declared in java.util package.
- We can use it store elements in key value pair format.
- Hashtable is sub class of Dictionary<K,V> class.
- It was introduced in JDK 1.0
- Method Summary
  - public abstract V put(K key, V value)
  - public abstract int size()
  - public abstract V get(Object key)
  - public abstract V remove(Object key)
  - public abstract boolean isEmpty()
  - public abstract Enumeration keys()
  - public abstract Enumeration elements()
- Consider example of Dictionary class:

```
import java.util.Dictionary;
import java.util.Enumeration;
import java.util.Hashtable;
public class Program {
    public static Dictionary<Integer, String> getDictionary( ){
        Dictionary<Integer, String> d = new Hashtable<>(); //Upcasting
        d.put(1,"PreDAC");
        d.put(2,"DAC");
        d.put(3,"DMC");
        d.put(4,"DIVESD");
        d.put(5,"DESD");
        d.put(6,"DBDA");
        return d;
    }
    private static void countAndPrintEntries(Dictionary<Integer, String>
d) {
        System.out.println("Count of entries : "+d.size());
    }
    private static void printKeys(Dictionary<Integer, String> d) {
        Enumeration<Integer> keys = d.keys();
        Integer key = null;
        while( keys.hasMoreElements() ) {
            key = keys.nextElement();
            System.out.println( key );
        }
    }
    private static void printValues(Dictionary<Integer, String> d) {
        Enumeration<String> values = d.elements();
        String value = null;
        while( values.hasMoreElements() ) {
            value = values.nextElement();
            System.out.println(value);
        }
    }
    private static void printValue(Dictionary<Integer, String> d, int
```

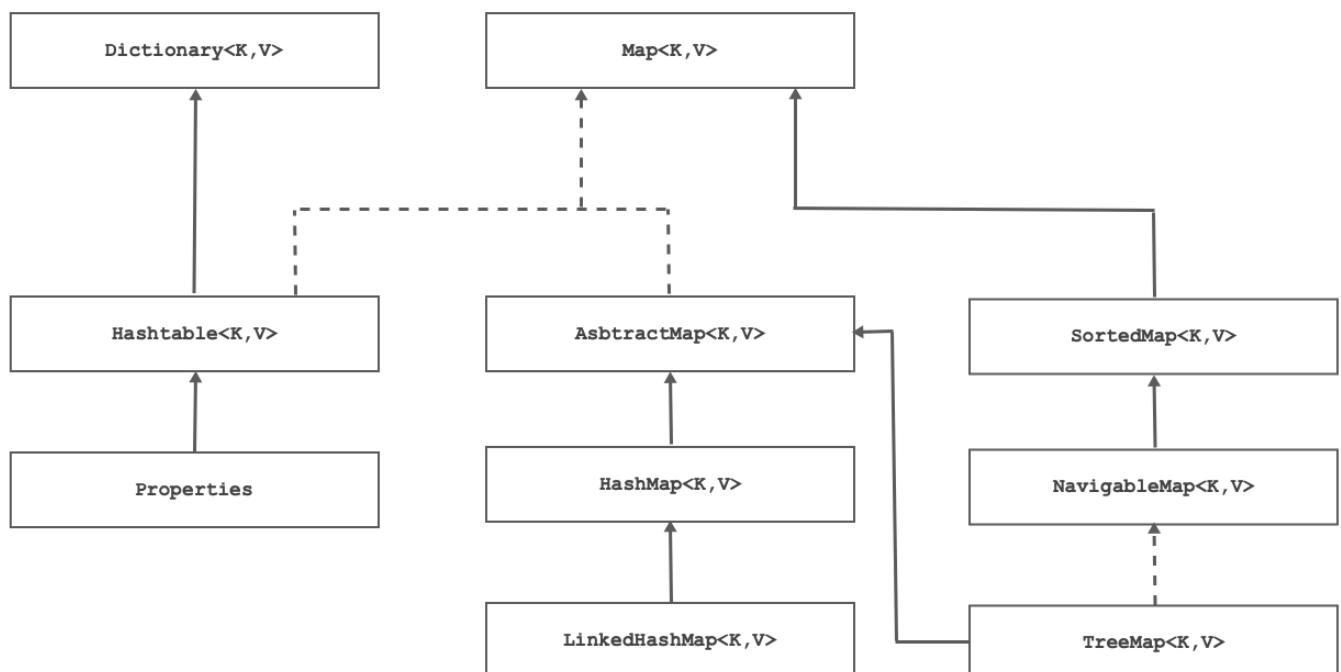
```

courseId) {
    Integer key = new Integer( courseId );
    String value = d.get(key);
    if( value != null )
        System.out.println(key+" "+value);
    else
        System.out.println("Invalid key");
}
private static void removeEntry(Dictionary<Integer, String> d, int
courseId) {
    Integer key = new Integer( courseId );
    String value = d.remove(key);
    if( value != null )
        System.out.println(key+" "+value+" is removed");
    else
        System.out.println("Invalid key");
}
public static void main(String[] args) {
    Dictionary<Integer, String> d = Program.getDictionary();
    //Program.countAndPrintEntries( d );
    //Program.printKeys( d );
    //Program.printValues( d );
    //Program.printValue( d, 2 );
    Program.removeEntry( d, 200);
}
}

```

- This class is obsolete. New implementations should implement the Map interface, rather than extending this class.

## Map<K,V>



- It is Key/Value pair interface which is declared in java.util package.

- It is member of Collection framework but it doesn't extend Collection interface.
- This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
- Hashtable<K,V>, HashMap<K,V>, LinkedHashMap<K,V>, TreeMap<K,V> etc. are map collections.
- If we want to insert any element/value inside map collection then we should provide key for it.
- In map collection we can insert duplicate value but we can not insert duplicate key.
- This interface is a member of the Java Collections Framework.
- It was introduced in JDK 1.2
- Map.Entry<K,V> is nested interface of Map interface.
- (key-value pair) instance is also called as entry.
- Methods of Map.Entry<K,V> interface
  - K getKey()
  - V getValue()
  - V setValue(V value)
- Method Summary of Map<K,V> interface
  - boolean isEmpty()
  - V put(K key, V value)
  - void putAll(Map<? extends K,? extends V> m)
  - int size()
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)
  - V get(Object key)
  - V remove(Object key)
  - void clear()
  - Set keySet()
  - Collection values()
  - Set<Map.Entry<K,V>> entrySet()

## Hashtable<K,V>

- It is a sub class of Dictionary<K,V> class and it implements Map<K,V>
- Since it is Map collection, we can not insert duplicate keys but we can insert duplicate value.
- In Hashtable key and value can not be null.
- It is synchronized collection.
- It is introduced in JDK 1.0.
- If we want to use any element of non primitive type inside Hashtable then non primitive type should override equals and hashCode method.

```

import java.util.ArrayList;
import java.util.Collection;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

```

```

import java.util.Set;

public class Program {
    private static Map<Integer, String> getMap() {
        Map<Integer, String> map = new Hashtable<>();
        map.put(1, "DAC");
        map.put(2, "DMC");
        map.put(3, "DESD");
        map.put(4, "DBDA");
        map.put(5, "DIVESD");
        return map;
    }
    private static void printAndCountEntries(Map<Integer, String> map) {
        System.out.println("Entries Count : "+map.size());
    }
    private static void printKeys(Map<Integer, String> map) {
        Set<Integer> keys = map.keySet();
        for (Integer key : keys) {
            System.out.print(key+" ");
        }
        System.out.println();
    }
    private static void printValues(Map<Integer, String> map) {
        Collection<String> values = map.values();
        for (String value : values) {
            System.out.print(value+" ");
        }
        System.out.println();
    }
    private static void printEntries(Map<Integer, String> map) {
        Set<Entry<Integer, String>> entries = map.entrySet();
        for (Entry<Integer, String> entry : entries) {
            System.out.println(entry.getKey()+" "+entry.getValue());
        }
    }
    private static void printEntry(Map<Integer, String> map, int id) {
        Integer key = new Integer(id);
        if( map.containsKey(key)) {
            String value = map.get(key);
            System.out.println( key+" "+value);
        }else
            System.out.println(key+" not found");
    }
    private static void removeEntry(Map<Integer, String> map, int id) {
        Integer key = new Integer(id);
        if( map.containsKey(key)) {
            String value = map.remove(key);
            System.out.println( key+" "+value+" is removed");
        }else
            System.out.println(key+" not found");
    }
    public static void main(String[] args) {
        Map<Integer, String> map = Program.getMap();
        //Program.printAndCountEntries( map );

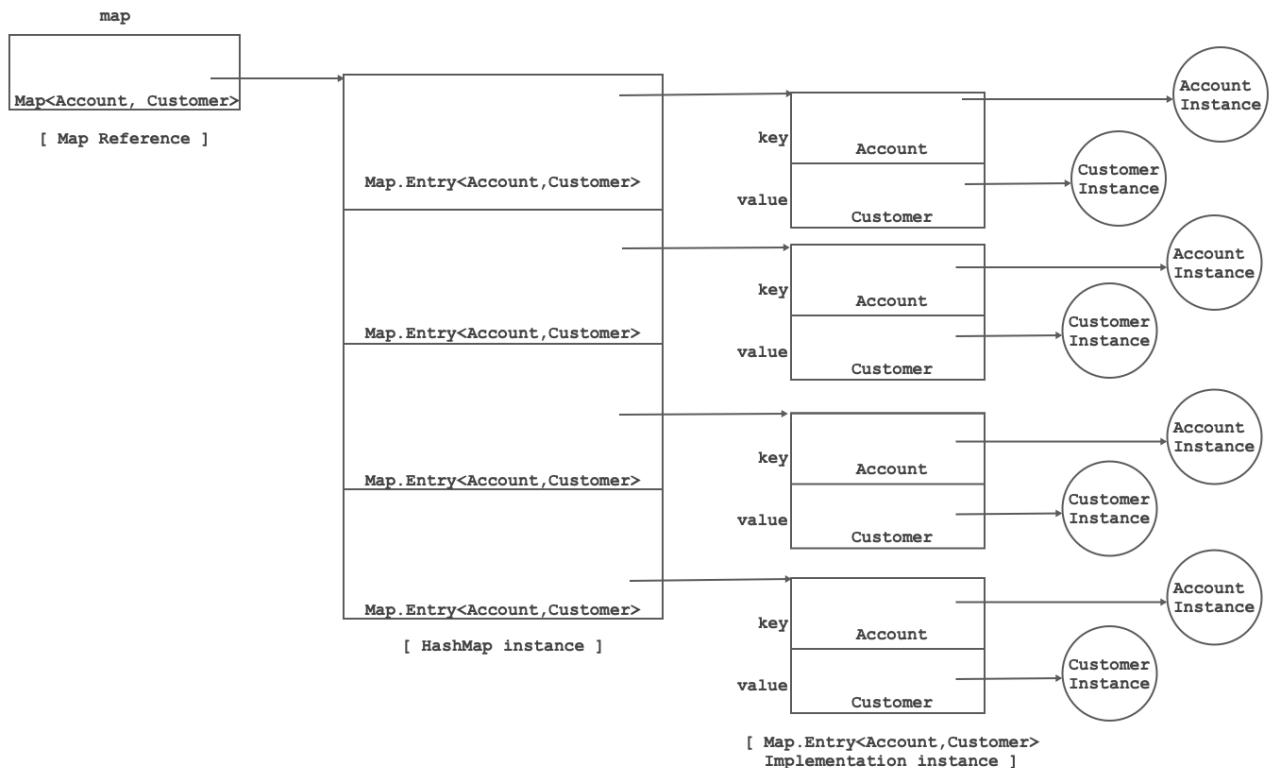
```

```

//Program.printKeys( map );
//Program.printValues( map );
//Program.printEntries( map );
//Program.printEntry( map, 3 );
//Program.removeEntry( map, 3 );

List<String> list = new ArrayList<>(map.values());
System.out.println(list);
}
}

```



# Day 23

## Hashtable

- Map is collection of entries where each entry contains key/value pair.
- In Map, we can use any type of instance as a key and value.
- Example:

```
Map<Integer, String> map = new Hashtable<>();  
Map<Account, Customer> map = new Hashtable<>();
```

- Consider example:

```
public class Account {  
    private int number;  
    private String type;  
    private float balance;  
    public Account() {  
        // TODO Auto-generated constructor stub  
    }  
    public Account(int number, String type, float balance) {  
        this.number = number;  
        this.type = type;  
        this.balance = balance;  
    }  
    public int getNumber() {  
        return number;  
    }  
    public void setNumber(int number) {  
        this.number = number;  
    }  
    public String getType() {  
        return type;  
    }  
    public void setType(String type) {  
        this.type = type;  
    }  
    public float getBalance() {  
        return balance;  
    }  
    public void setBalance(float balance) {  
        this.balance = balance;  
    }  
  
    @Override  
    public int hashCode() {  
        final int prime = 31;  
        int result = 1;
```

```

        result = prime * result + number;
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Account other = (Account) obj;
        if (number != other.number)
            return false;
        return true;
    }
    @Override
    public String toString() {
        return String.format("%-10d%-15s%8.2f", this.number, this.type,
this.balance);
    }
}

```

```

public class Customer {
    private String name;
    private String email;
    private String contactNumber;
    public Customer() {
        // TODO Auto-generated constructor stub
    }
    public Customer(String name, String email, String contactNumber) {
        this.name = name;
        this.email = email;
        this.contactNumber = contactNumber;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getContactNumber() {
        return contactNumber;
    }
    public void setContactNumber(String contactNumber) {

```

```

        this.contactNumber = contactNumber;
    }
    @Override
    public String toString() {
        return String.format("%-15s%-20s%-15s", this.name, this.email,
this.contactNumber);
    }
}

```

```

import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

import org.example.domain.Account;
import org.example.domain.Customer;

public class MapTest {
    private Map<Account, Customer> map;
    public void setMap(Map<Account, Customer> map) {
        this.map = map;
    }
    public void addEntries(Account[] keys, Customer[] values) {
        if( this.map != null ) {
            for( int index = 0; index < keys.length; ++ index ) {
                Account key = keys[ index ];
                Customer value = values[ index ];
                this.map.put(key, value);
            }
        }
    }
    public Customer findCustomer(int number) {
        if( this.map != null ) {
            Account key = new Account();
            key.setNumber(number);
            if( this.map.containsKey(key)) {
                Customer value = this.map.get(key);
                return value;
            }
        }
        return null;
    }
    public boolean removeEntry(int number) {
        if( this.map != null ) {
            Account key = new Account();
            key.setNumber(number);
            if( this.map.containsKey(key)) {
                this.map.remove(key);
                return true;
            }
        }
        return false;
    }
}

```

```

    }
    public void printEntries() {
        if( this.map != null ) {
            Set<Entry<Account, Customer>> entries = this.map.entrySet();
            for (Entry<Account, Customer> entry : entries) {
                Account key = entry.getKey();
                Customer value = entry.getValue();
                System.out.println(key+" "+value);
            }
        }
    }
}

```

```

import java.util.Hashtable;
import java.util.Scanner;

import org.example.domain.Account;
import org.example.domain.Customer;
import org.example.test.MapTest;

public class Program {
    private static Scanner sc = new Scanner(System.in);
    private static Account[] getKeys( ) {
        Account[] arr = new Account[ 5 ];
        arr[ 0 ] = new Account( 4627,"Saving", 10000.50f );
        arr[ 1 ] = new Account( 1930,"Current",20000.50f );
        arr[ 2 ] = new Account( 5629,"Loan",30000.50f );
        arr[ 3 ] = new Account( 9356,"Pention",40000.50f );
        arr[ 4 ] = new Account( 7485,"PPF",50000.50f );
        return arr;
    }
    private static Customer[] getValues( ) {
        Customer[] arr = new Customer[ 5 ];
        arr[ 0 ] = new Customer("Chetan","chetan@gmail.com","9527325202");
        arr[ 1 ] = new Customer("Ganesh","ganesh@gmail.com","9527325202");
        arr[ 2 ] = new
Customer("Saishwar","saishwar@gmail.com","9527325202");
        arr[ 3 ] = new Customer("Anup","anup@gmail.com","9527325202");
        arr[ 4 ] = new Customer("Mahesh","mahesh@gmail.com","9527325202");
        return arr;
    }
    public static void acceptRecord( int[] accountNumber ) {
        if( accountNumber != null ) {
            System.out.print("Enter account number      :   ");
            accountNumber[ 0 ] = sc.nextInt();
        }
    }
    private static void printRecord(Customer value) {
        if( value != null )
            System.out.println( value );
        else

```

```

        System.out.println("Account not found");
    }
    private static void printRecord(boolean removedStatus) {
        if( removedStatus )
            System.out.println("Entry is removed.");
        else
            System.out.println("Account not found");
    }
    private static int menuList( ) {
        System.out.println("0.Exit.");
        System.out.println("1.Add Entry.");
        System.out.println("2.Find Customer.");
        System.out.println("3.Remove Entry.");
        System.out.println("4.Print Entries.");
        System.out.print("Enter choice : ");
        return sc.nextInt();
    }
    public static void main(String[] args) {
        int choice;
        int[] accountNumber = new int[ 1 ];
        MapTest test = new MapTest();
        test.setMap( new Hashtable<>() );
        while( ( choice = Program.menuList( ) ) != 0 ) {
            switch( choice ) {
            case 1:
                Account[] keys = Program.getKeys();
                Customer[] values = Program.getValues();
                test.addEntries( keys, values );
                break;
            case 2:
                Program.acceptRecord(accountNumber);
                Customer value = test.findCustomer( accountNumber[ 0 ] );
                Program.printRecord( value );
                break;
            case 3:
                Program.acceptRecord(accountNumber);
                boolean removedStatus = test.removeEntry( accountNumber[ 0 ] );
                Program.printRecord( removedStatus );
                break;
            case 4:
                test.printEntries( );
                break;
            }
        }
    }
}

```

## HashMap<K,V>

- It is a sub class of AbstractMap<K,V> class and it implements Map<K,V>
- Its implementation is based on Hash table.

- Since it is Map collection, we can not insert duplicate keys but we can insert duplicate value.
- In HashMap<K,V> key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method we can consider it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.2.
- If we want to use any element of non final type inside Hashtable then non final type should override equals and hashCode method.
- Instantiation:

```
HashMap<Integer, String> map = new HashMap<>();
Map<Integer, String> map = new HashMap<>();
```

## LinkedHashMap<K,V>

- It is a sub class of HashMap<K,V>.
- Its implementation is based on Hash table and linked list.
- During traversing it maintains order of elements.
- Since it is Map collection, we can not insert duplicate keys but we can insert duplicate value.
- In HashMap<K,V> key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method we can consider it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.2.
- If we want to use any element of non final type inside Hashtable then non final type should override equals and hashCode method.
- Instantiation:

```
LinkedHashMap<Integer, String> map = new LinkedHashMap<>(); //OK
HashMap<Integer, String> map = new LinkedHashMap<>(); //OK
Map<Integer, String> map = new LinkedHashMap<>(); //OK
```

## TreeMap<K,V>

- It is a sub class of AbstractMap<K,V> class and it implements NavigableMap<K,V>
- Its implementation is based on Red Black Tree.
- TreeMap store entries in sorted order.
- Since it is Map collection, we can not insert duplicate keys but we can insert duplicate value.
- In TreeMap<K,V> key can not be null but value can be null.
- It is unsynchronized collection. Using Collections.Collections.synchronizedSortedMap() method we can consider it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.2.

- If we want to use any element of non final type inside TreeMap then non final type should implement Comparable interface.
- Instantiation:

```
TreeMap<Integer, String> map = new TreeMap<>(); //OK
Map<Integer, String> map = new TreeMap<>(); //OK
```

What is the difference between Collection and Collections?

What is the difference between Collection and Map?

What is the difference between Hashtable and HashMap?

What is the difference between HashMap and LinkedHashMap?

What is the difference between HashMap and TreeMap?

What is the difference between HashSet and Hashtable?

How will you convert HashMap keys and Values into ArrayList?

```
Map<Integer, String> map = new HashMap<>();
map.put( 1, "DAC");
map.put( 2, "DMC");
map.put( 3, "DESD");

Set<Integer> keys = map.keySet();
List<Integer> list = new ArrayList( keys );

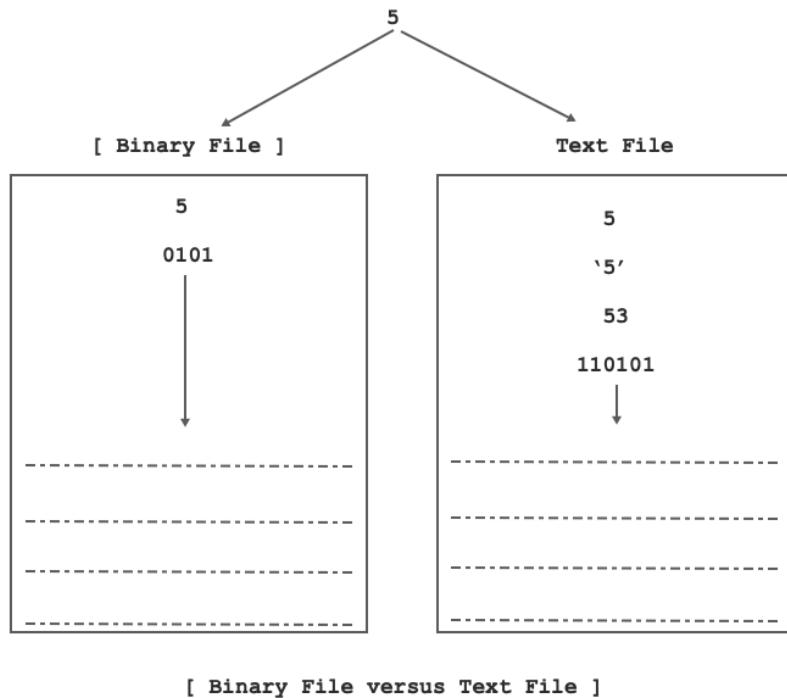
Collection<String> values = map.values();
List<String> list = new ArrayList( values );
```

## File I/O

- Variable
  - It is temporary container which is used to store record in primary memory( RAM ).
- File
  - It is permanent container which is used to store record on HDD.

- Types of files:

( Let us write 5 in binary & text file )



[ Binary File versus Text File ]

#### ■ Text File

- Examples: .c, .cpp, .java, .cs, .html, css, .js, .txt, .doc, .docs, .xml, .json etc.
- We can read text file using any text editor.
- It requires more processing than binary file hence it is slower in performance.
- If we want to save data in human readable format then we should create text file.

#### ■ Binary File

- Examples: .jpg, .jpeg, .bmp, .gif, .mp3, .mp4, .obj, .class etc.
- To read binary file, we must use specific program.
- It requires less processing than text file hence it is faster in performance.
- If we dont want to save data in human readable format then we should create binary file.

### • Stream

- It is an abstraction( instance ) which either consume( read) or produce( write ) information from source to destination.
- Stream is always associated with resource.
- Standard stream instances of Java programming languages which are associated with Console( Keyboard / Monitor ):
  - System.in
  - System.out
  - System.err

- If we want to save data in file the we should use types declared in java.io package.

- java.io.Console class represents Console.

```

import java.io.Console;

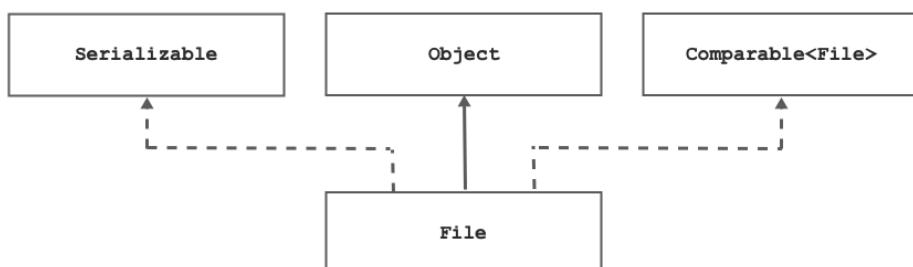
public class Program {
    public static void main(String[] args) {
        System.out.print("Enter name : ");
        Console console = System.console();
        String name = console.readLine();
        //System.out.println("Name : "+name);
        console.printf("Name : %s\n", name);
    }
}

```

- java.io.File class represents Physical file on HDD.

## File

- It is a class declared in java.io package.



- We can use java.io.File class:
  - To create new file or to remove existing file.
  - To create new directory or to remove existing directory.
  - To read metadata of file, directory or drive.
- Constructor:
  - public File(String pathname)

```

String pathname = "Sample.txt";
File file = new File( pathname );

```

- Create new file:

```

public static void main(String[] args) {
try {
    String pathName = "Sample.txt";
    File file = new File(pathName);
    if( file.exists() )
        System.out.println("File already exist.");
    else {
        boolean status = file.createNewFile();
        System.out.println("File creation is successful.");
    }
}

```

```

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

```

- Remove existing file:

```

public static void main(String[] args) {
try {
    String pathName = "Sample.txt";
    File file = new File(pathName);
    if( !file.exists())
        System.out.println("File does not exist.");
    else {
        boolean status = file.delete();
        System.out.println("File deletion is successful.");
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

```

- Create new directory

```

public static void main(String[] args) {
try {
    String pathName = "Sample";
    File file = new File(pathName);
    if( file.exists())
        System.out.println("Directory already exist.");
    else {
        boolean status = file.mkdir();
        System.out.println("Directory creation is successful.");
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

```

- Remove existing directory

```

public static void main(String[] args) {
try {
    String pathName = "Sample";
    File file = new File(pathName);
    if( !file.exists())

```

```

        System.out.println("Directory does not exist.");
    else {
        boolean status = file.delete();
        System.out.println("Directory deletion is successful.");
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

```

- Read Metadata of files:

```

public static void main(String[] args) {
try {
//String pathName = "D:\\\\Users\\\\sandeep\\\\CDAC\\\\Quiz.txt";
//Windows
String pathName = "/Users/sandeep/Desktop/CDAC/Quiz.txt";
//Linux
File file = new File(pathName);
if( file.exists() ) {
    System.out.println( "File Name : "+file.getName());
//Quiz.txt
    System.out.println("Parent Directory: "+file.getParent());
    System.out.println("Length : "+file.length());
    System.out.println("Last Modified : "+new
SimpleDateFormat("dd MMM,yyyy hh:mm:ss").format(new
Date(file.lastModified())));
}
} catch (Exception e) {
    e.printStackTrace();
}
}

```

- Read Metadata of directory:

```

public static void main(String[] args) {
try {
String pathName = "/Users/sandeep/Desktop/CDAC/";
File file = new File(pathName);
if( file.exists() && file.isDirectory() ) {
//String[] nameList = file.list();
File[] files = file.listFiles();
for (File f : files) {
    if( !f.isHidden())
        System.out.println(f.getName());
}
} catch (Exception e) {
    e.printStackTrace();
}
}

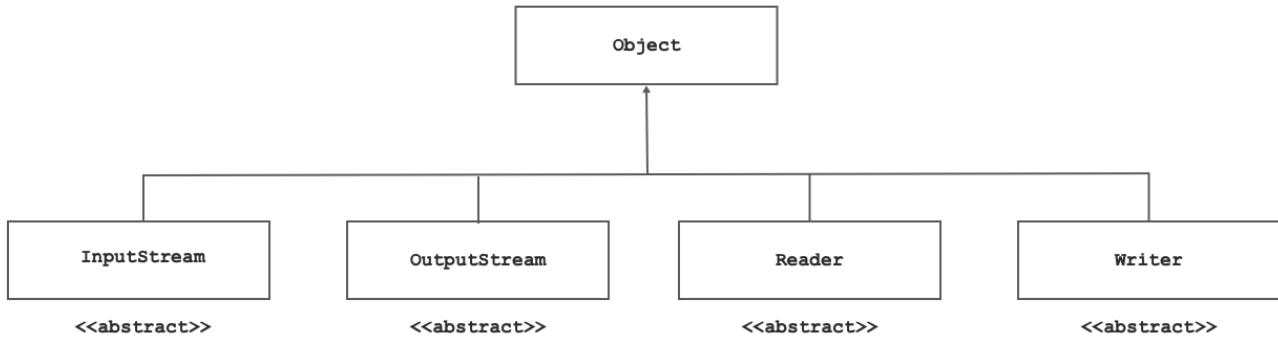
```

```
}
```

Assignment: Write a code in java to simulate windows command prompt?

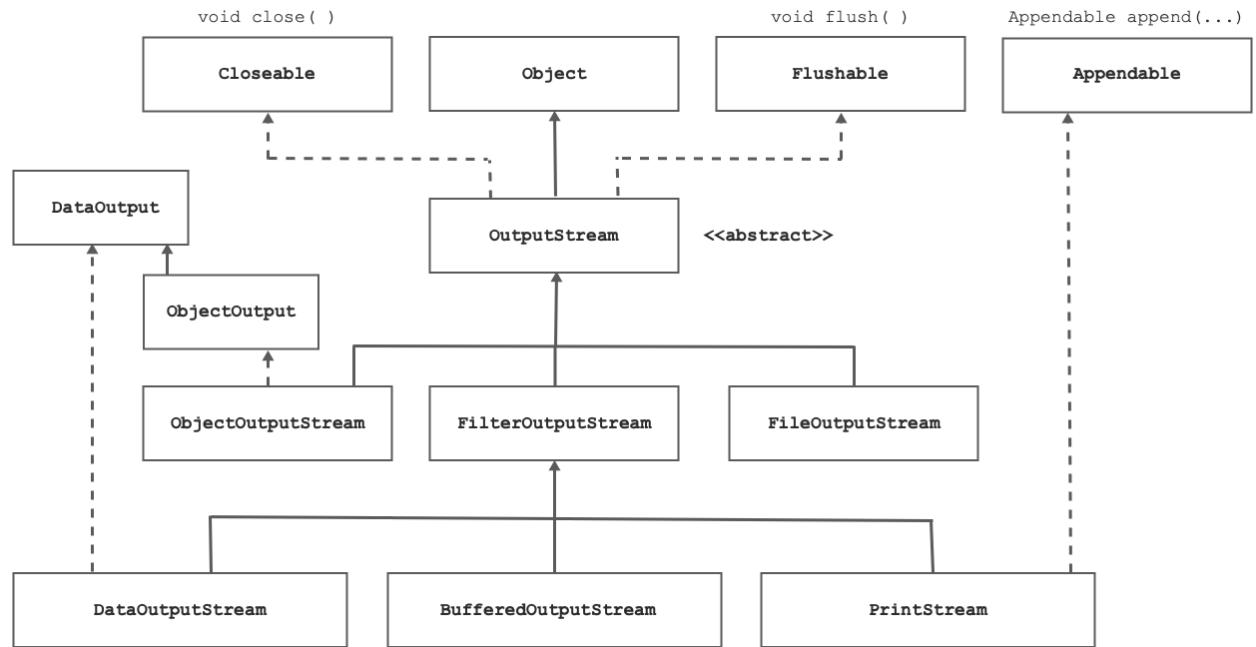
```
Enter path:>
If it is file then display File metadata
If it is Directory then display File, Directory and Drive metadata
```

## File data manipulation

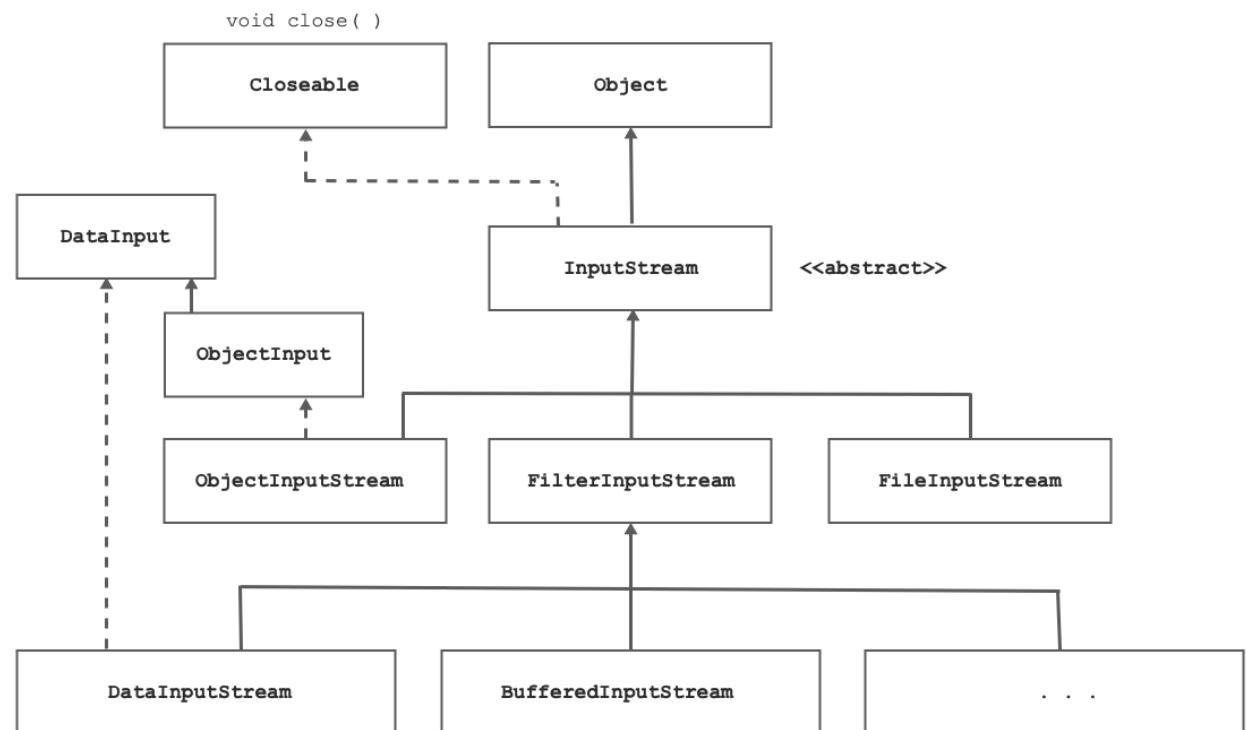


- Interfaces:
  - `FileFilter`
  - `FilenameFilter`
  - `Closeable`
  - `Flushable`
  - `DataInput`
  - `DataOutput`
  - `ObjectInput`
  - `ObjectOutput`
  - `Serializable`
  - `Externalizable`
- If we want to read/write data into binary file then we should use `InputStream`, `OutputStream` and their sub classes.

- Consider OutputStream hierarchy:

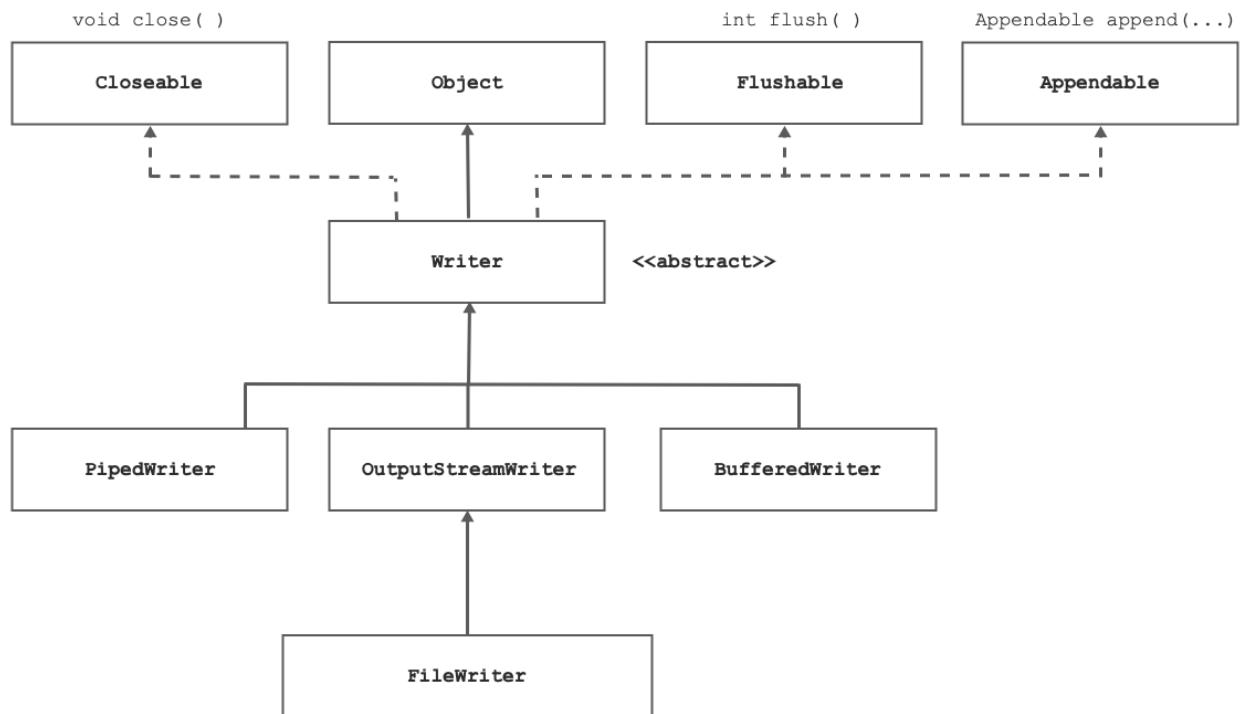


- Consider InputStream hierarchy:

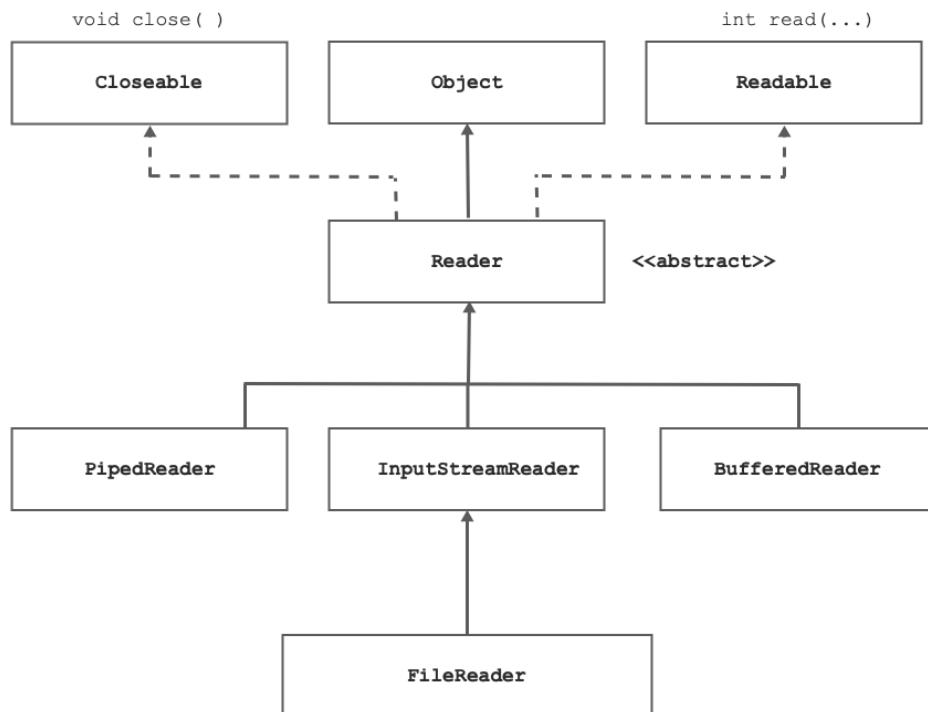


- If we want to read/write data into text file then we should use Reader, Writer and their sub classes.

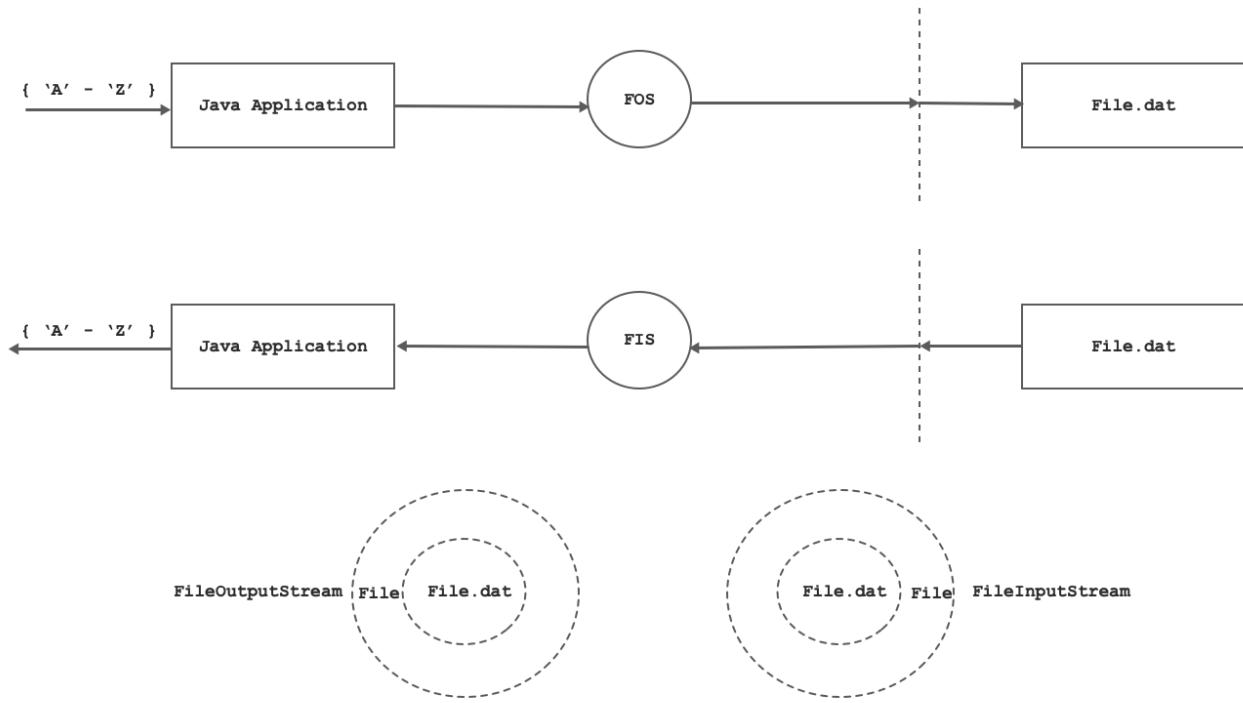
- Consider Writer hierarchy:



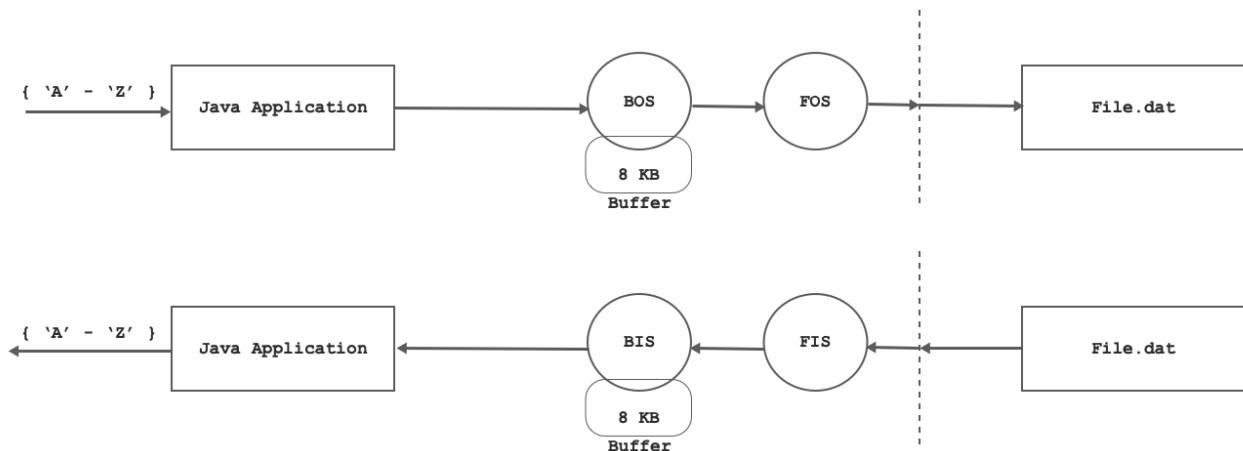
- Consider Reader hierarchy:



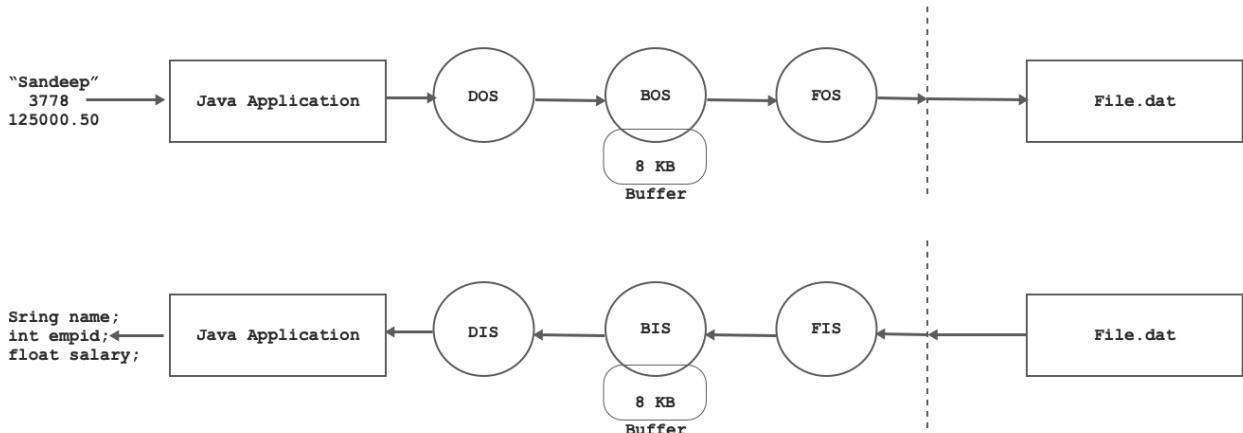
- How to read and write single byte data into file?



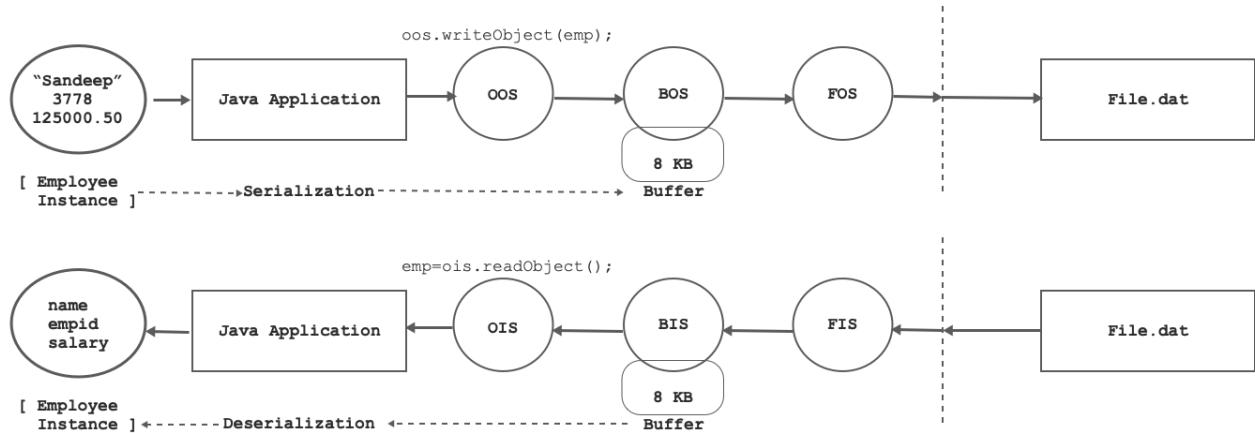
- How to improve performance of read/write operations?



- How to read/write primitive values and Strings to and from files?



- How to serialize / deserialize state of java Instance?



- Process of converting state of Java instance into binary data is called as serialization.
- To serialize Java instance type of instance must implement Serializable interface otherwise we get NotSerializableException.
- If class contains fields of non primitive type then its type must implement Serializable interface. Consider following example:

```

class Date implements Serializable{
}
class Address implements Serializable{
}
class Person implements Serializable{
    private String name; //Final class. Already implemented
    Serializable
    private Address address;
    private Date birthDate;
}

```

- transient is modifier in Java. If we declare any field transient then JVM do not serialize its state.
- Process of converting binary data into Java instance is called as deserialization.

# Day 24

## SerialVersionUID

- In Java, the **serialVersionUID** is a special static variable that is used to control the serialization and deserialization process of objects. It is a version number associated with a serialized class, and it serves as a unique identifier for the class.
- When an object is serialized, its state is converted into a byte stream. The **serialVersionUID** is included in this byte stream. During deserialization, the JVM checks if the **serialVersionUID** of the serialized object matches the **serialVersionUID** of the corresponding class in the local environment. If the **serialVersionUID** values match, the deserialization process proceeds successfully. However, if the **serialVersionUID** values don't match, a **InvalidClassException** is thrown, indicating a version mismatch between the serialized object and the class definition.
- The **serialVersionUID** is used for versioning purposes to ensure that the serialized object and the class definition are compatible. It helps to maintain compatibility between different versions of a class when objects are serialized and deserialized.
- Here's an example that demonstrates the usage of **serialVersionUID**:

```
import java.io.*;
class Sample implements Serializable {
    private static final long serialVersionUID = 1L;
    private int data;

    public Sample(int data) {
        this.data = data;
    }

    public int getData() {
        return this.data;
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        Sample sample = new Sample(42);

        // Serialize the object to a file
        try ( ObjectOutputStream outputStream = new
ObjectOutputStream(new FileOutputStream("object.ser"))) {
            outputStream.writeObject(sample);
            System.out.println("Object serialized successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

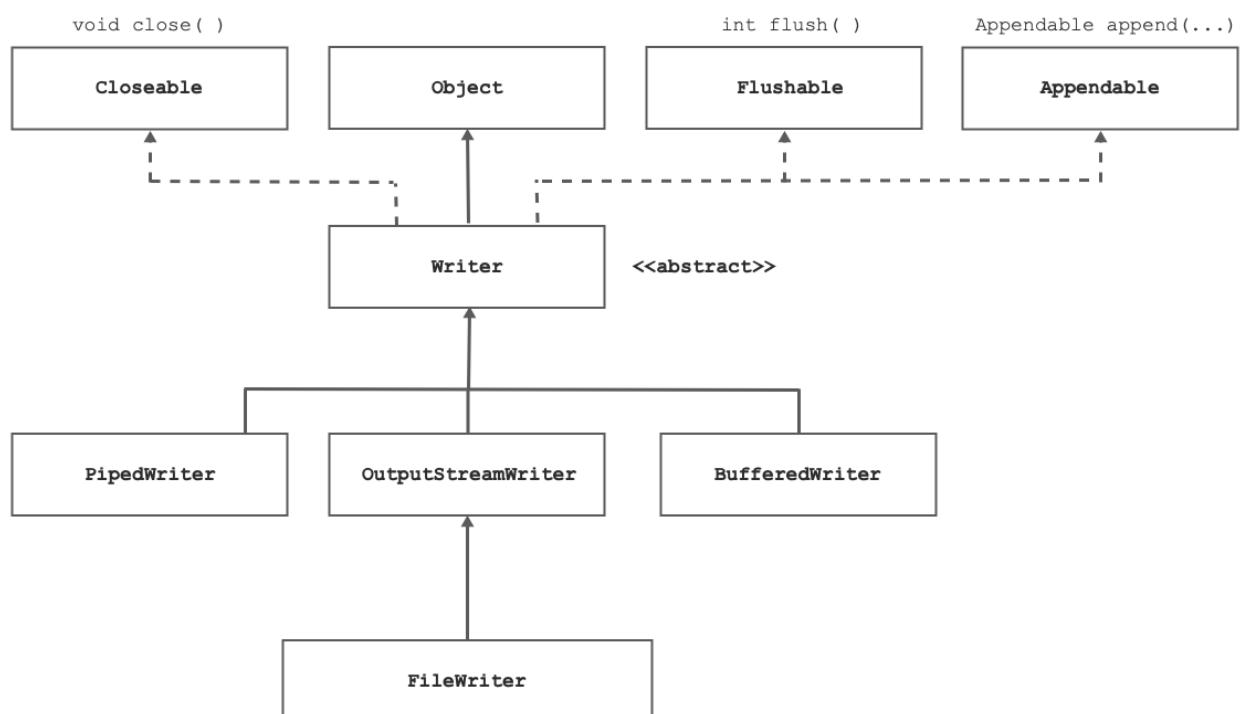
// Deserialize the object from the file
try {
    ObjectInputStream inputStream = new ObjectInputStream(new
FileInputStream("object.ser")){
        Sample sample = (Sample) inputStream.readObject();
        System.out.println("Deserialized object data: " +
sample.getData());
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}

```

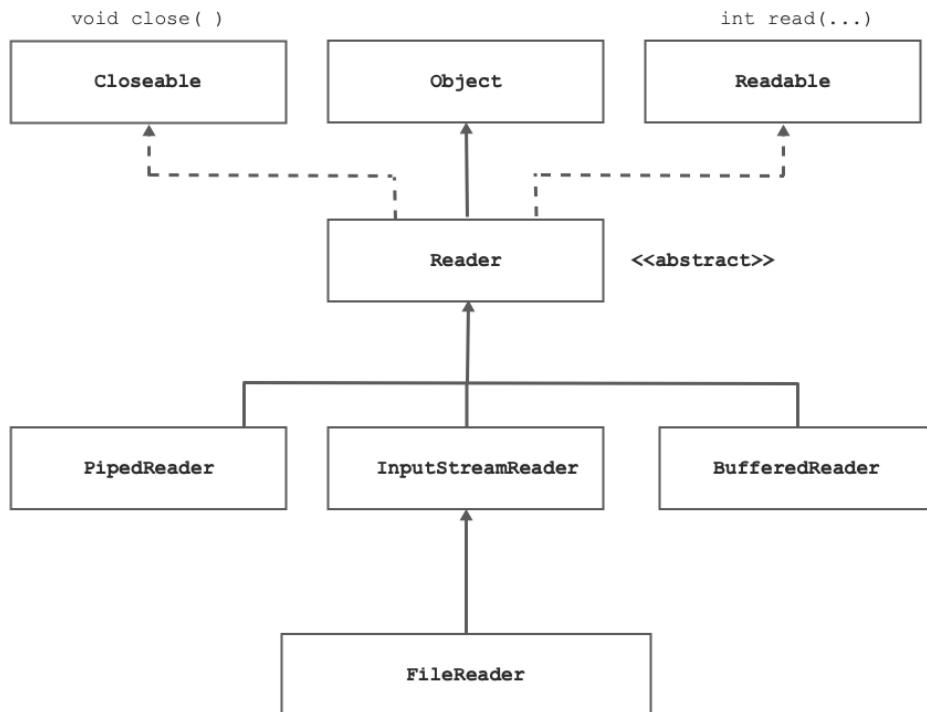
- In the above example, the Sample implements the Serializable interface, indicating that its objects can be serialized. The class also defines a serialVersionUID as 1L. When an object of Sample is serialized, the serialVersionUID is included in the serialized byte stream. During deserialization, the serialVersionUID is checked to ensure compatibility between the serialized object and the class definition.
- It's important to note that if you make any changes to a serialized class, such as adding or removing fields or changing their types, you should update the serialVersionUID accordingly to maintain compatibility between different versions of the class.

## Text file manipulation

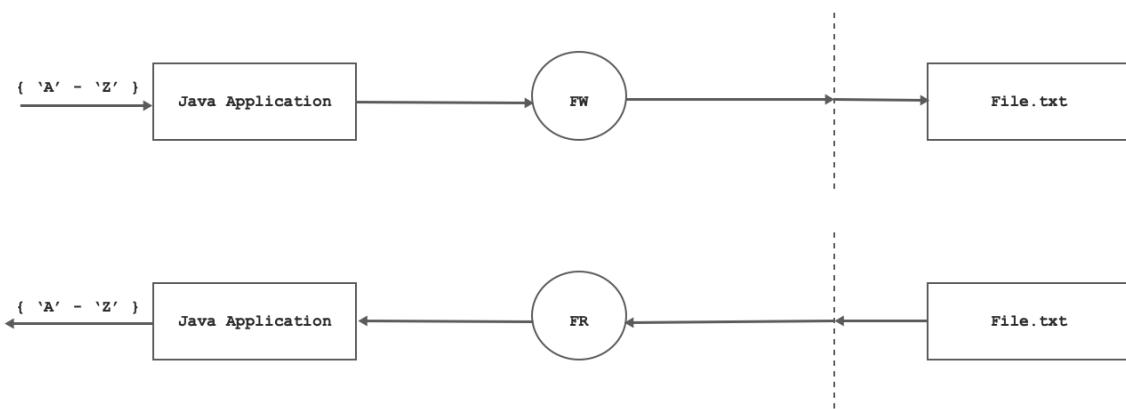
- If we/you want to manipulate text file then we should use Reader/writer classes and their sub classes.
- Consider Writer hierarchy:



- Consider Reader hierarchy:

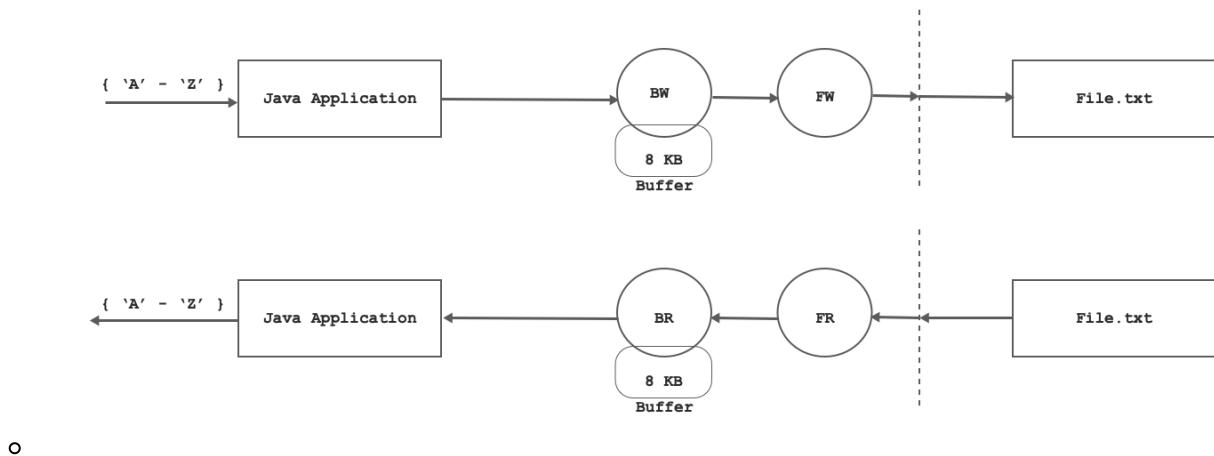


- The **FileWriter** class is used to write characters to a file. It extends the Writer class and provides methods to write characters or character arrays to a file. It handles the underlying low-level operations required for writing characters to a file, such as opening the file, writing the data, and closing the file.
- The **FileReader** class is used to read characters from a file. It extends the Reader class and provides methods to read characters into a buffer from a file. It handles the low-level operations required for reading characters from a file, such as opening the file, reading the data, and closing the file.



o

- The **BufferedWriter** class is used to write characters to a character stream with buffering capabilities. It wraps an existing Writer and improves the performance of writing characters by buffering them in memory before writing them to the underlying stream.
- The **BufferedReader** class is used to read characters from a character stream with buffering capabilities. It wraps an existing Reader and improves the performance of reading characters by buffering them in memory before accessing the underlying stream.



- In Java, InputStreamReader and OutputStreamWriter are classes that provide a bridge between byte streams and character streams. They are used to convert bytes to characters (for input) and characters to bytes (for output) while reading from or writing to streams.
- The InputStreamReader class is used to read bytes from an InputStream and decode them into characters using a specified character encoding. It converts a stream of bytes into a stream of characters.
- Consider following code:

```

import java.io.*;
public class Program {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        // Reading from a file using InputStreamReader
        try (FileInputStream fileInputStream = new
FileInputStream(inputFile);
        InputStreamReader inputStreamReader = new
InputStreamReader(fileInputStream)) {
            char[] buffer = new char[1024];
            int length;
            while ((length = inputStreamReader.read(buffer)) != -1)
{
                System.out.println(new String(buffer, 0, length));
}
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
    
```

- The OutputStreamWriter class is used to write characters to an output stream of bytes. It wraps an existing OutputStream and provides methods to write characters to the output stream using a specified character encoding.
- Consider following example:

```

import java.io.*;
public class Program {
    public static void main(String[] args) {
        String outputFile = "output.txt";
        // Writing to a file using OutputStreamWriter
        try (FileOutputStream fileOutputStream = new
FileOutputStream(outputFile);
            OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(fileOutputStream)) {
            String data = "Hello, World!";
            outputStreamWriter.write(data);
            System.out.println("Data written to the file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

- How to serialize and deserialize state of Java instance into text file?

```

class Employee implements Serializable {
    private static final long serialVersionUID = 1L;

    private String name;
    private int age;
    private String department;

    public Employee(String name, int age, String department) {
        this.name = name;
        this.age = age;
        this.department = department;
    }
    public String getName() {
        return this.name;
    }
    public int getAge() {
        return this.age;
    }
    public String getDepartment() {
        return this.department;
    }
    public String toString() {
        return "Name: " + this.name + ", Age: " + this.age + ", "
Department: " + this.department;
    }
}

```

```

public class Program {
    public static void writeRecord( String pathname ) throws
Exception{
        try ( BufferedWriter bufferedWriter = new BufferedWriter(new
FileWriter(pathname))) {
            Employee employee = new Employee("Sandeep Kulange", 39,
"Engineering");
            String str = new StringBuilder()
                .append(employee.getName()).append(",")
                .append(employee.getAge()).append(",")
                .append(employee.getDepartment())
                .toString();
            bufferedWriter.write(str);
            System.out.println("Employee object serialized and written
to the "+pathname+" file.");
        }
    }
    private static Employee deserializeEmployee(String str) {
        String[] parts = str.split(",");
        String name = parts[0].trim();
        int age = Integer.parseInt(parts[1].trim());
        String department = parts[2].trim();
        return new Employee(name, age, department);
    }
    public static void readRecord( String pathname ) throws Exception{
        try (BufferedReader bufferedReader = new BufferedReader(new
FileReader(pathname))) {
            String str = bufferedReader.readLine();
            Employee emp = deserializeEmployee(str);
            System.out.println(emp.toString());
        }
    }
    public static void main(String[] args) {
        try {
            String fileName = "employees.txt";
            Program.writeRecord(fileName);
            Program.readRecord(fileName);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

## Socket Programming

- A URL stands for Uniform Resource Locator.
- It is a standardized address used to locate resources on the internet.
  - Web Page.
  - Image files.

- Audio/video files.
  - Document files etc.
- A URL has the following components:
    - Protocol: It specifies the protocol used to access the resource, such as HTTP, HTTPS, FTP, etc.
    - Host: It specifies the domain name or IP address of the server hosting the resource.
    - Port: (Optional) It specifies the port number on the server to connect to. If not specified, it defaults to the default port for the specified protocol (e.g., 80 for HTTP, 443 for HTTPS).
      - Ports are identified by numbers ranging from 0 to 65535.
      - In networking, certain port numbers are reserved for specific services and protocols.

These reserved ports are standardized and commonly used for well-known services.

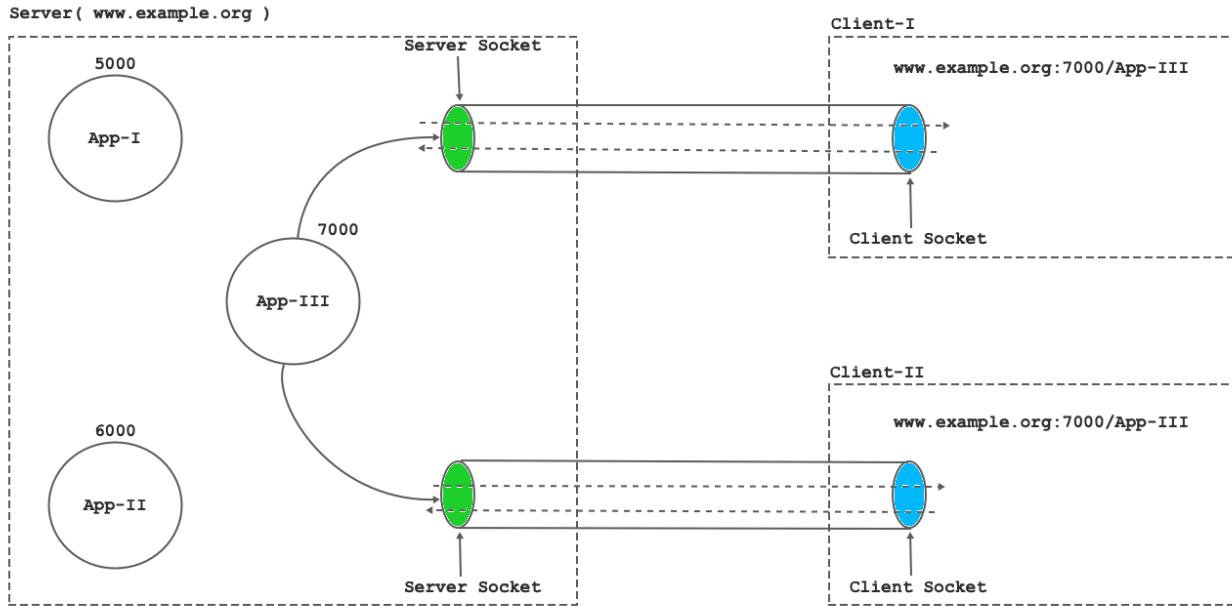
Here are some examples of reserved ports:

      - Well-Known Ports (0-1023):
        - Port 80: Hypertext Transfer Protocol (HTTP)
        - Port 443: Hypertext Transfer Protocol Secure (HTTPS)
        - Port 21: File Transfer Protocol (FTP)
        - Port 22: Secure Shell (SSH)
        - Port 25: Simple Mail Transfer Protocol (SMTP)
        - Port 110: Post Office Protocol (POP3)
        - Port 143: Internet Message Access Protocol (IMAP)
        - Port 3389: Remote Desktop Protocol (RDP)
        - Port 53: Domain Name System (DNS)
      - Registered Ports (1024-49151):
        - Port 3306: MySQL Database System
        - Port 5432: PostgreSQL Database System
        - Port 8080: Hypertext Transfer Protocol (HTTP) alternative port
        - Port 8443: Hypertext Transfer Protocol Secure (HTTPS) alternative port
      - Dynamic and/or Private Ports (49152-65535):
        - These ports are typically used for dynamic or private purposes and are not assigned to specific services.
      - When developing network applications, it's recommended to use port numbers above 1023 for custom services to avoid conflicts with reserved ports.
    - Path: It specifies the path to the resource on the server. It can include directories and subdirectories leading to the specific file or resource.
    - Parameters: (Optional) It specifies additional parameters passed to the server as part of the URL. Parameters are typically key-value pairs used to provide information or customize the request.
    - Fragment: (Optional) It specifies a specific location within the resource, such as an anchor or section identifier within an HTML page.
  - An example of a URL:

```
https://www.example.com:8080/path/to/resource?
key1=value1&key2=value2#section1
```

- In this example:
  - Protocol: HTTPS
  - Host: www.example.com
  - Port: 8080
  - Path: /path/to/resource
  - Parameters: key1=value1, key2=value2
  - Fragment: section1

- Let us understand concept of socket. Consider following diagram:



- Port number is logical integer number which is used to identify process running on server.
- A physical memory which is allocated client and server through which client/server can send/receive data is called as socket.

- In java, If we want to do socket programming then we should import `java.net` package.
- In Java, the `InetAddress` class is used to represent IP addresses and perform various operations related to network communication. Consider following code:

```

import java.net.InetAddress;
import java.net.UnknownHostException;
public class Program {
    public static void main(String[] args) {
        try {
            // Get the local host's IP address
            InetAddress localHost = InetAddress.getLocalHost();
            System.out.println("Local Host IP Address: " +
localHost.getHostAddress());

            // Resolve the IP address for a given hostname
            InetAddress googleAddress =
InetAddress.getByName("www.google.com");
            System.out.println("Google IP Address: " +

```

```

        googleAddress.getHostAddress());
        System.out.println("Google Hostname: " +
googleAddress.getHostName());

        // Checking reachability
        boolean isReachable = googleAddress.isReachable(5000); // 5
seconds timeout
        System.out.println("Is Google Reachable: " + isReachable);
    } catch (UnknownHostException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

- In Java, Inet4Address and Inet6Address are subclasses of the InetAddress class and represent IP addresses in the IPv4 and IPv6 formats, respectively. Consider following code:

```

import java.net.InetAddress;
import java.net.Inet4Address;
import java.net.Inet6Address;
import java.net.UnknownHostException;
public class Program {
    public static void main(String[] args) {
        try {
            // Retrieve an IPv4 address
            InetAddress ipv4Address =
Inet4Address.getByName("192.168.0.1");
            System.out.println("IPv4 Address: " +
ipv4Address.getHostAddress());

            // Retrieve an IPv6 address
            InetAddress ipv6Address =
Inet6Address.getByName("2001:0db8:85a3:0000:0000:8a2e:0370:7334");
            System.out.println("IPv6 Address: " +
ipv6Address.getHostAddress());

            // Resolve hostname for an IPv6 address
            InetAddress ipv6Host =
Inet6Address.getByName("2001:4860:4860::8888");
            System.out.println("IPv6 Hostname: " +
ipv6Host.getHostName());
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

- In Java, the ServerSocket class is a mechanism for creating server applications that listen for and accept incoming client connections.
- The ServerSocket class provides the following functionalities:
  - Listening for Incoming Connections
  - Accepting Client Connections
  - Handling Multiple Client Connections
  - Closing the ServerSocket
- A simple example that demonstrates the usage of ServerSocket:

```

package org.example.server;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Program {
    public static void main(String[] args) {
        int portNumber = 12345; // Port number to listen on
        try {
            // Create a ServerSocket and bind it to the specified port
            ServerSocket serverSocket = new ServerSocket(portNumber);

            System.out.println("Server listening on port " + portNumber);

            while (true) {
                // Accept client connections
                Socket clientSocket = serverSocket.accept();
                System.out.println("Client connected: " +
clientSocket.getInetAddress().getHostAddress());

                // Perform further processing on the clientSocket
                // For example, start a new thread to handle the client
                connection
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

- In Java, the Socket class is a fundamental component of network communication and is used to establish a connection between a client and a server.
- The Socket class provides the following functionalities:
  - Creating a Socket
  - Communicating with the Server
  - Sending and Receiving Data
  - Closing the Socket

- A simple example that demonstrates the usage of Socket in a client-server scenario:

```

package org.example.client;
import java.io.*;
import java.net.*;

public class Program {
    public static void main(String[] args) {
        String serverHostname = "127.0.0.1"; // Server IP address or
hostname
        int serverPort = 12345; // Server port

        try {
            // Create a socket and connect to the server
            Socket socket = new Socket(serverHostname, serverPort);

            //TODO: Send/recieve data

            // Close the socket
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

- Consider code for half duplex chating application:
- Server

```

package org.example.server;

import java.io.BufferedReader;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class Program {
    public static final String HOST = "localhost";
    public static final int PORT = 23423;
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        DataInputStream inputStream = null;
        DataOutputStream outputStream = null;
        Scanner sc = null;
        try {

```

```

serverSocket = new ServerSocket(PORT);
System.out.print("Listening for incoming connection....");
Socket socket = serverSocket.accept();
System.out.println("Connected.");
inputStream = new DataInputStream(new BufferedInputStream(
socket.getInputStream()));
outputStream = new DataOutputStream(new
BufferedOutputStream(socket.getOutputStream()));
sc = new Scanner(System.in);

String message;
do {
    System.out.print("S:Server : ");
    message = sc.nextLine();
    outputStream.writeUTF(message);
    outputStream.flush();

    message = inputStream.readUTF();
    System.out.println("S:Client : "+message);
}while( !message.equalsIgnoreCase("end"));

}catch( Exception ex ) {
    ex.printStackTrace();
}finally {
    try {
        serverSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}
}

```

- Client

```

package org.example.client;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.util.Scanner;

public class Program {
    public static final String HOST = "localhost";
    public static final int PORT = 23423;
    public static void main(String[] args) {
        Socket socket = null;
        DataInputStream inputStream = null;

```

```

DataOutputStream outputStream = null;
Scanner sc = null;
try {
    socket = new Socket(HOST, PORT);
    inputStream = new DataInputStream(new BufferedInputStream(
socket.getInputStream()));
    outputStream = new DataOutputStream(new
BufferedOutputStream(socket.getOutputStream()));
    sc = new Scanner(System.in);
    String message = null;
    do {
        message = inputStream.readUTF();
        System.out.println("C:Server : "+message);

        System.out.print("C:Client : ");
        message = sc.nextLine();
        outputStream.writeUTF(message);
        outputStream.flush();
    }while( !message.equalsIgnoreCase("end"));

}catch( Exception ex ) {
    ex.printStackTrace();
}finally {
    try {
        socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

## Functional Programming in Java 8

- Functional programming is a programming paradigm that focuses on using functions to perform computations.
- In functional programming, functions are treated as first-class citizens, which means they can be passed as arguments, returned as values, and stored in variables.
- In Java, functional programming can be practiced using several features and concepts introduced in the language.
  - Functional Interfaces
  - Lambda Expressions
  - Method References
  - Stream API
  - Optional class

## Functional interface

- In Java, a functional interface is an interface that contains exactly one abstract method. It is also known as a Single Abstract Method (SAM) interface.

- The key characteristics of functional interfaces are:
  - Single Abstract Method: Functional interfaces have only one abstract method. This method represents the behavior that can be implemented by lambda expressions or method references.
  - Default and Static Methods: Functional interfaces can have default and static methods in addition to the single abstract method.
  - Annotation: It is recommended to annotate functional interfaces with the `@FunctionalInterface` annotation. Although this annotation is not required for an interface to be considered functional, it helps prevent accidental addition of extra abstract methods.
- Java 8 introduced several functional interfaces in the `java.util.function` package to support functional programming concepts.
- These functional interfaces can be used with lambda expressions and method references to express behavior concisely. Here are some commonly used functional interfaces in Java 8:
  - `Predicate`
  - `Consumer`
  - `Supplier`
  - `Function<T, R>`
  - `UnaryOperator`
  - `BinaryOperator`

## Lambda expression

- An expression which contain lambda operator( `->` ) is called as lambda expression.
- Lambda expressions were introduced in Java 8 as a concise way to represent anonymous functions. They provide a simplified syntax for writing functional-style code and enable the use of functional interfaces to pass behavior as arguments or return values.
- Syntax of Lambda Expressions:

```
(parameters) -> expression/body
```

- A lambda expression consists of three main components: parameter list, arrow token, and body.
- Here's a breakdown of the components:
  - Parameters: The input parameters to the function. They can be omitted if the function takes no arguments or inferred if there is only one parameter.
  - Arrow Token: The arrow token `->` separates the parameter list from the body of the lambda expression.
  - Expression/Body: The implementation of the lambda function, which can be a single expression or a block of code enclosed in curly braces `{}`.
- Lambda Expression with No Parameters

```
() -> System.out.println("Hello, world!");
```

- Lambda Expression with One Parameter

```
name -> System.out.println("Hello, " + name);
```

- Lambda Expression with Multiple Parameters

```
(int a, int b) -> {
    int sum = a + b;
    System.out.println("The sum is: " + sum);
}
```

- consider the Runnable functional interface

```
Runnable runnable = () -> {
    System.out.println("Business Logic..."); 
    // Additional code here
};
```

## Method Reference

- Method references in Java provide a way to refer to methods without executing them. They can be seen as a shorthand notation for writing lambda expressions that invoke a single method.
- There are four different types of method references in Java:
  - Reference to a Static Method:

```
FunctionalInterface func = ClassName::staticMethodName;
```

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

public class Main {
    public static void main(String[] args) {
        MathOperation operation = MathUtils::add;
        int result = operation.operate(3, 4);
        System.out.println(result); // Output: 7
    }
}
```

- Reference to an Instance Method of a Particular Object

```
FunctionalInterface func = objectReference::instanceMethodName;
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
numbers.forEach(System.out::println);
```

- Reference to an Instance Method of an Arbitrary Object of a Particular Type

```
FunctionalInterface func = ClassName::instanceMethodName;
```

```
List<String> names = Arrays.asList("Dac", "Dmc", "Dbda");  
names.forEach(String::toUpperCase);
```

- Reference to a Constructor

```
FunctionalInterface func = ClassName::new;
```

```
Supplier<List<String>> listSupplier = ArrayList::new;  
List<String> names = listSupplier.get();
```