



Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane



Day01 Agenda

- Documentation
- Java history
- Version history
- Java platforms , SDK, JDK, JRE, JVM
- Structure of Java class
- Java application flow of execution
- Comments and Entry point method
- Data types Wrapper class Introduction
- Narrowing and Widening
- Operators



Documentation

- **JDK download:**
 - <https://adoptopenjdk.net/>
- **Spring Tool Suite 3 download:**
 - <https://github.com/spring-projects/toolsuite-distribution/wiki/Spring-Tool-Suite-3>
- **Oracle Tutorial:**
 - <https://docs.oracle.com/javase/tutorial/>
- **Java 8 API and Java 11 API Documentation Documentation:**
 - <https://docs.oracle.com/javase/8/docs/api/>
 - <https://docs.oracle.com/en/java/javase/11/docs/api/>
- **MySQL Connector:**
 - <https://downloads.mysql.com/archives/c-j/>
- **Core Java Tutorials:**
 1. <http://tutorials.jenkov.com/java/index.html>
 2. <https://www.baeldung.com/java-tutorial>
 3. <https://www.journaldev.com/7153/core-java-tutorial>



Java Text Books

- o **Java, The complete reference, Herbert Schildt**
- o **Core and Advanced Java Black Book**
- o **Head First Java: A Brain-Friendly Guide , by Kathy Sierra**



Java Installation

1. JDK 11 download

<https://www.oracle.com/java/technologies/javase-jdk11-downloads.html>

Choose your platform , download the installer and install the JDK

2. Open command prompt or terminal

Type

`java -version`

It should show you : java version "11.0.7" 2020-04-14 LTS

Note : Basic version should be 11, update version may differ.

3. Java IDE download

<https://github.com/spring-projects/toolsuite-distribution/wiki/Spring-Tool-Suite-3>

Choose 3.9.+ version and download. Extract it. No installation is required.

4. Java API Documentation link

<https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

5. Offline version

<https://www.oracle.com/java/technologies/javase-jdk11-doc-downloads.html>



WHY Java ?

BUZZWORDS.

1. Simple
2. Secure
3. Portable
4. Object Oriented
5. Robust
6. Architecture Netural(Platform Independent)
7. Multithreading
8. Interpreted
9. High Performance
10. Distributed
11. Dynamic



Java History

- James Gosling, Mike Sheridan and Patrick Naughton initiated the Java language project in June 1991.
- Java was originally developed by James Gosling at Sun Microsystems and released in 1995.
- The language was initially called Oak after an oak tree that stood outside Gosling's office.
- Later the project went by the name *Green* and was finally renamed *Java*, from Java coffee, a type of coffee from Indonesia.
- Gosling and his team did a brainstorm session and after the session, they came up with several names such as **JAVA, DNA, SILK, RUBY, etc.**
- Sun Microsystems released the first public implementation as Java 1.0 in 1996.



Java History

- The Java programming language is a general-purpose, concurrent, class-based, object-oriented language.
- The Java programming language is a high-level language.
- The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included.
- **It is intended to be a production language, not a research language.**
- The Java programming language is statically typed.
- It promised **Write Once, Run Anywhere (WORA)** functionality.



Version History

Version	Date	
JDK Beta	1995	<ul style="list-style-type: none">- The first version was released on January 23, 1996.
JDK 1.0	January 23, 1996 ^[39]	<ul style="list-style-type: none">- The acquisition of Sun Microsystems by Oracle Corporation was completed on January 27, 2010
JDK 1.1	February 19, 1997	
J2SE 1.2	December 8, 1998	
J2SE 1.3	May 8, 2000	
J2SE 1.4	February 6, 2002	
J2SE 5.0	September 30, 2004	
Java SE 6	December 11, 2006	
Java SE 7	July 28, 2011	
Java SE 8	March 18, 2014	<ul style="list-style-type: none">- In September 2017, Mark Reinhold, chief Architect of the Java Platform, proposed to change the release train to "<u>one feature release every six months</u>".- OpenJDK (Open Java Development Kit) is a free and open source implementation of the (Java SE). It is the result of an effort Sun Microsystems began in 2006.
Java SE 9	September 21, 2017	
Java SE 10	March 20, 2018	
Java SE 11	September 25, 2018 ^[40]	
Java SE 12	March 19, 2019	
Java SE 13	September 17, 2019	
Java SE 14	March 17, 2020	
Java SE 15	September 15, 2020 ^[41]	
Java SE 16	March 16, 2021	



Java Platforms

1. Java SE

- Java Platform Standard Edition.
- It is also called as Core Java.
- For general purpose use on Desktop PC's, servers and similar devices.

2. Java EE

- Java Platform Enterprise Edition.
- It is also called as advanced Java / enterprise java / web java.
- Java SE plus various API's which are useful client-server applications.

3. Java ME

- Java Platform Micro Edition.
- Specifies several different sets of libraries for devices with limited storage, display, and power capacities.
- It is often used to develop applications for mobile devices, PDAs, TV set-top boxes and printers.

4. Java Card

- A technology that allows small Java-based applications (applets) to be run securely on smart cards and similar small-memory devices.

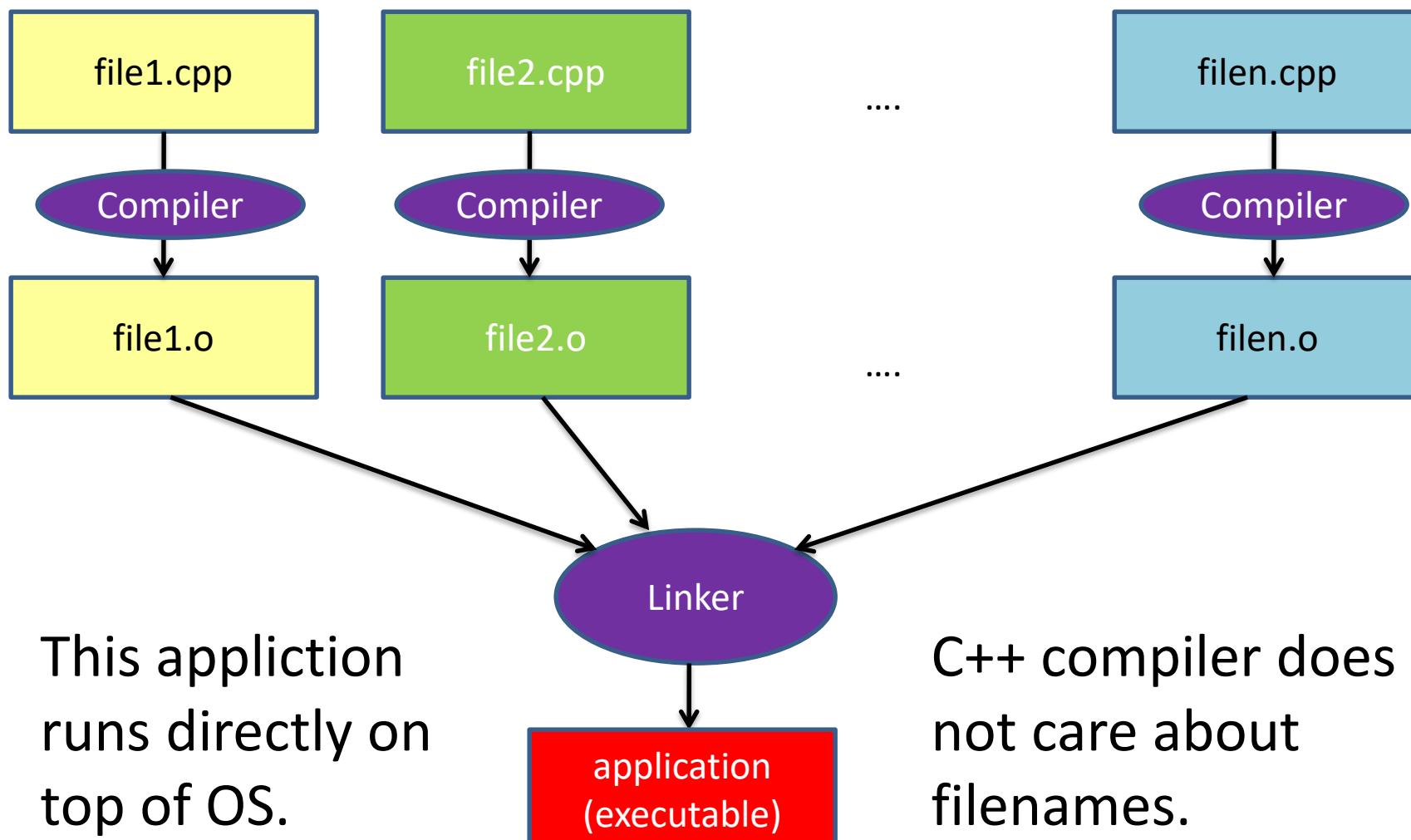


SDK, JDK, JRE, JVM

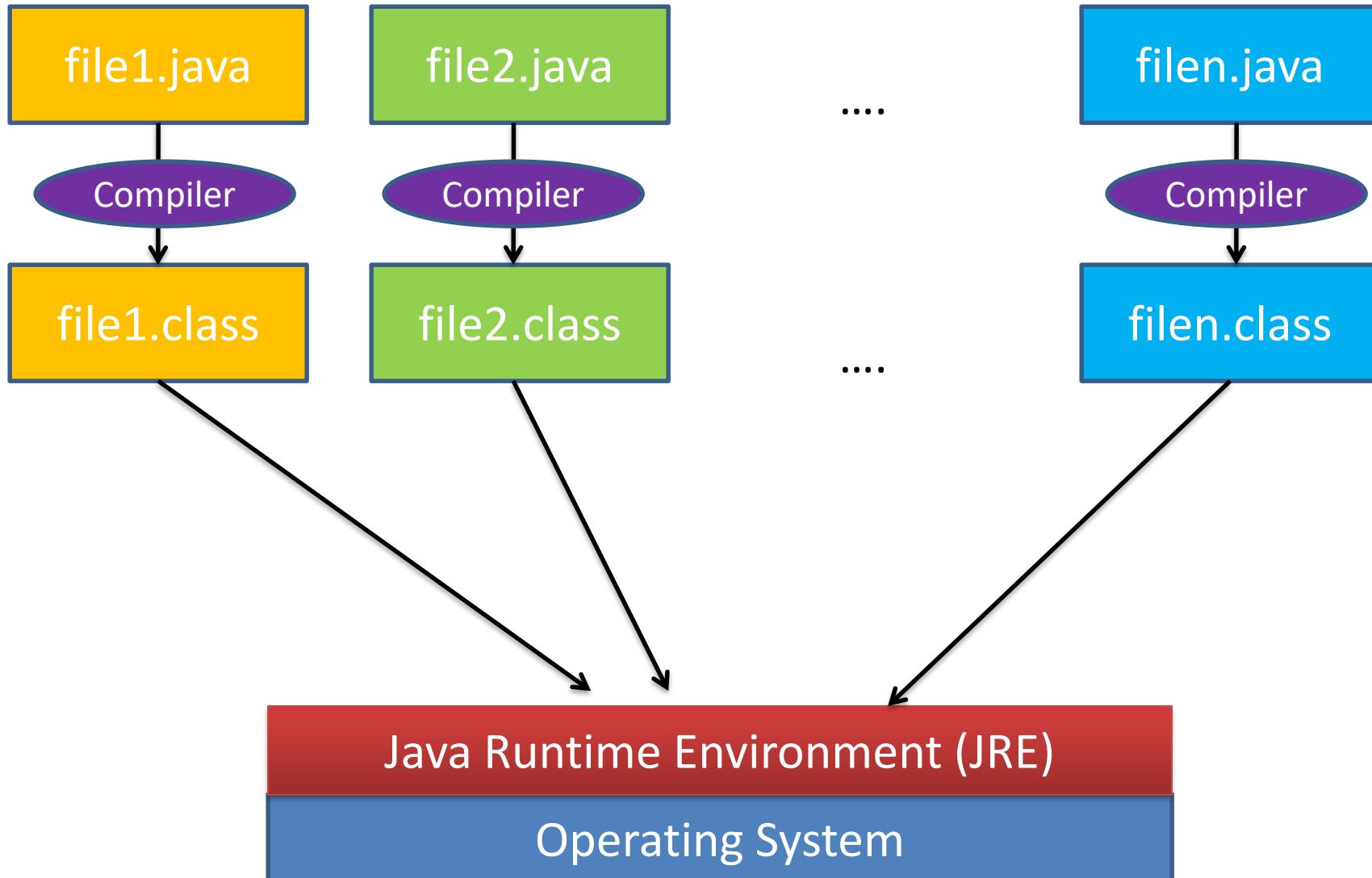
- **SDK** = Development Tools + Documentation + Libraries + Runtime Environment.
- **JDK** = Java Development Tools + Java Docs + **rt.jar** + **JVM**.
 - JDK : Java Development Kit.
 - It is a software, that need to be install on developers machine.
 - We can download it from oracle official website.
- **JDK** = Java Development Tools + Java Docs + **JRE[rt.jar + JVM]**.
 - JRE : Java Runtime Environment.
 - rt.jar and JVM are integrated part of JRE.
 - JRE is a software which comes with JDK. We can also download it separately.
 - To deploy application, we should install it on client's machine.
- **rt.jar** file contains core Java API in **compiled form**.
- **JVM** : An engine, which manages execution of Java application. (also called as Execution Engine)



C++ compiler & Linker usage



Java compiler usage



Language Basics

- Keywords
- Variables
- Conditional Statements
- Loops
- Data Types
- Operators
- Coding Conventions



Java Keywords

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
true	false	null			



Simple Hello Application

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

```
Compilation=> javac Program.java //Output : Program.class  
Execution=> java Program
```

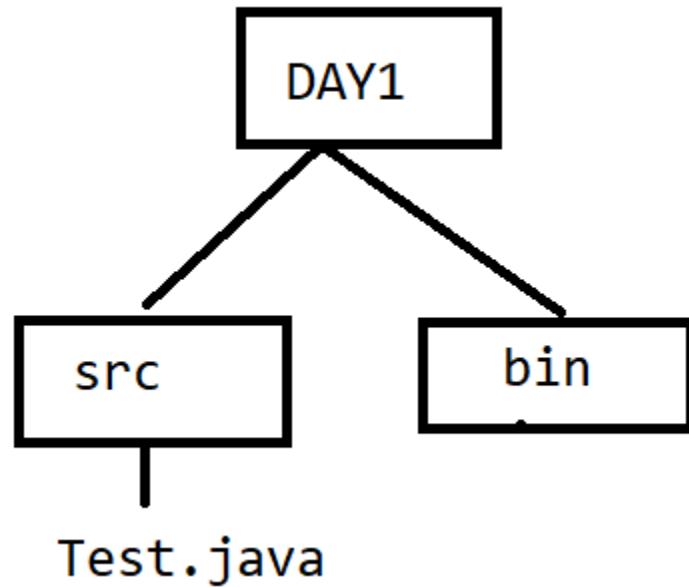
```
System      : Final class declared in java.lang package  
out         : public static final field of System class. Type of out is PrintStream  
println     : Non static method of java.io.PrintStream class
```

To view class File :

```
javap -c Program.class
```



How to compile the code from src and store .class inside bin



```
D:\DAY1> cd src  
D:\DAY1\src>javac -d ..\bin Test.java  
D:\DAY1\src>cd ..\bin  
D:\DAY1\bin>java Test
```



java.lang.System class

```
package java.lang;
import java.io.*;
public final class System{
    public static final InputStream in;
    public static final OutputStream out;
    public static final OutputStream err;
    public static Console console();
    public static void exit(int status);
    public static void gc();
}
```

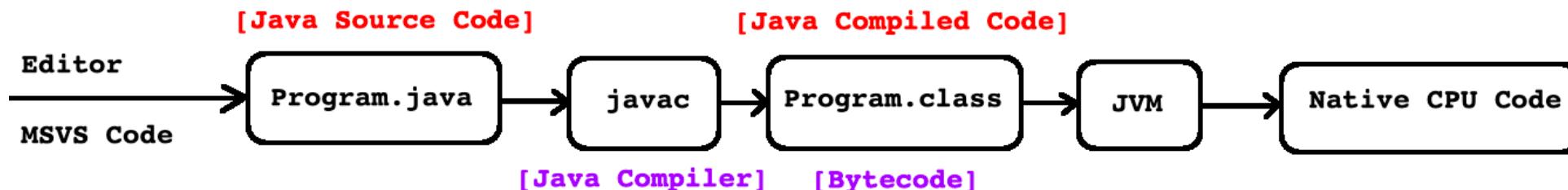


Stream

- Stream is an abstraction(object) which either produce(write)/consume(read) information from source to destination.
- Standard stream objects of Java which is associated with console:
 1. **System.in**
 - Ø It represents keyboard.
 2. **System.out**
 - Ø It represents Monitor.
 3. **System.err**
 - Ø Error stream which represents Monitor.



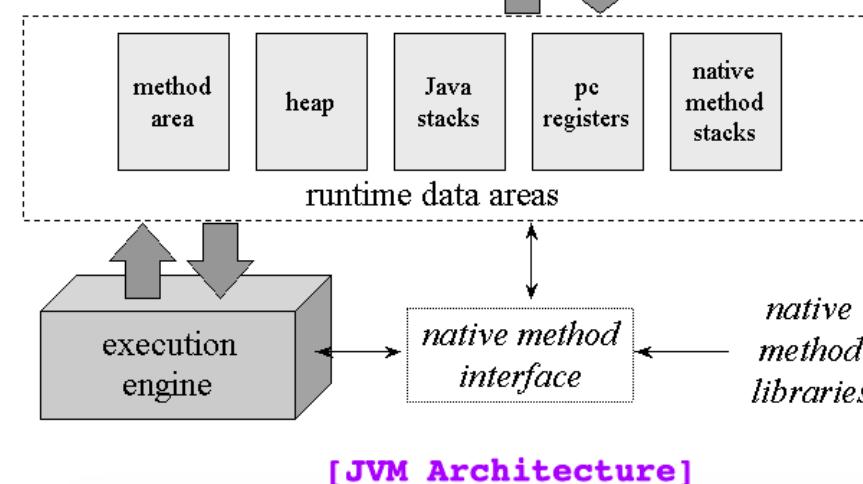
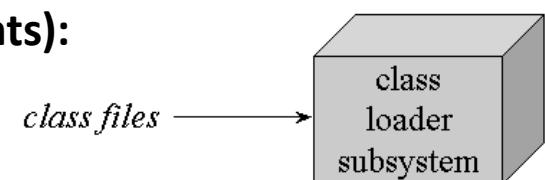
Java application flow of execution



Components of JVM (Major 3 Components):

1. Class Loader Sub System

- Bootstrap class loader
- Extension classLoader
- Application classLoader
- User Defined class Loader



2. Runtime Data Areas

3. Execution Engine

- Interpreter
- Compiler
- Garbage Collector

Method Area

- Metaspace: JDK 1.8 onwards Loaded Byte codes (Loaded class info) 1 Single copy static data members, constructors, methods

Heap

- Java object heap state of the object (non static data members) 1 single copy

Java Stack

- Stack is created one per thread , method local info(like args,local vars, ret vals) Individual stack : stack frames



Comments

- Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program.
- Types of Comments:
 - **Implementation Comment**
 - Implementation comments are those found in C++, which are delimited by `/*...*/`, and `//`.
 1. Single-Line Comment
 2. Block Comment(also called as multiline comment)
 - **Documentation Comment**
 - Documentation comments (known as "doc comments") are Java-only, and are delimited by `/**...*/`.
 - Doc comments can be extracted to HTML files using the javadoc tool.

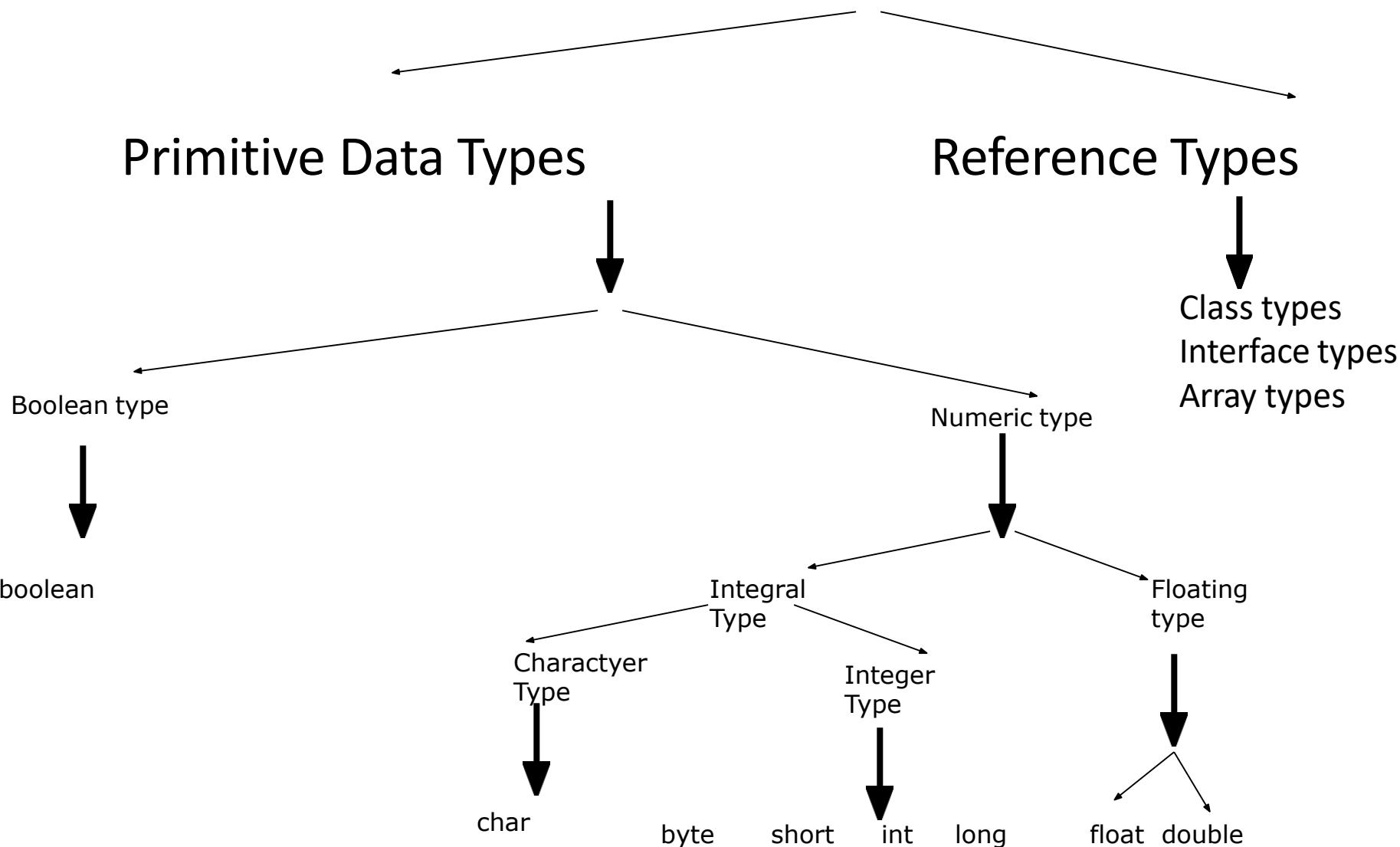


Entry point method

- Syntax:
 1. **public static void main(String[] args)**
 2. **public static void main(String... args)**
- Java compiler do not check/invoke main method. JVM invoke main method.
- When we start execution of Java application then JVM starts execution of two threads:
 1. Main thread : responsible for invoking main method.
 2. Garbage Collector : responsible for deallocated memory of unused object.
- We can overload main method in Java.
- We can define main method per class. But only one main method can be considered as entry point method.



Data types In Java



Data Types

- Integral Type

- byte 8 bits
- short 16 bits
- int 32 bits
- long 64 bits

- Textual Type

- char 16 bits, UNICODE Character
- String

- Floating Point Type

- float 32 bits
- double 64 bits

- Boolean Type 1 bit

- true
- false



Data Types

- Data type of any variable decide following things:
 1. **Memory:** How much memory is required to store the data.
 2. **Nature:** Which kind of data is allowed to store inside memory.
 3. **Operation:** Which operations are allowed to perform on the data stored in memory.
- The Java programming language is a statically typed language, which means that every variable and every expression has a type that is known at compile time.
 - Types of data type:
 1. **Primitive type(also called as value type)**
 - **boolean** type
 - Numeric type
 - 1. Integral types(**byte, char, short, int, long**)
 - 2. Floating point types(**float, double**)
 2. **Non primitive type(also called as reference type)**
 - **Interface, Class, Type variable, Array**



Variables

- A **variable** is a name given memory location. That memory is associated to a data type and can be assigned a value.
- int n; float f1; char ch; double d;

Rules of Variables

- All variable names must begin with a letter of the alphabet, an underscore (_), or a dollar sign (\$). Can't begin with a digit. The rest of the characters may be any of those previously mentioned plus the digits 0-9.
- The convention is to always use a (lower case) letter of the alphabet. The dollar sign and the underscore are discouraged.

```
1. int n1;  
2. n1 =21 ;           // assignment  
3. int val=50; //initialization  
5. double d = 21.8;   // initialization  
6. d = n1;           // assignment  
7. float f1 = 16.13F;
```



Data Types

Sr.No.	Primitive Type	Size[In Bytes]	Default Value[For Field]	Wrapper Class
1	boolean	Isn't specified	FALSE	Boolean
2	byte	1	0	Byte
3	char	2	\u0000	Character
4	short	2	0	Short
5	int	4	0	Integer
6	float	4	0.0f	Float
7	double	8	0.0d	Double
8	long	8	0L	Long

Note : Char datatype supports UNICODE character set, so 2 bytes .



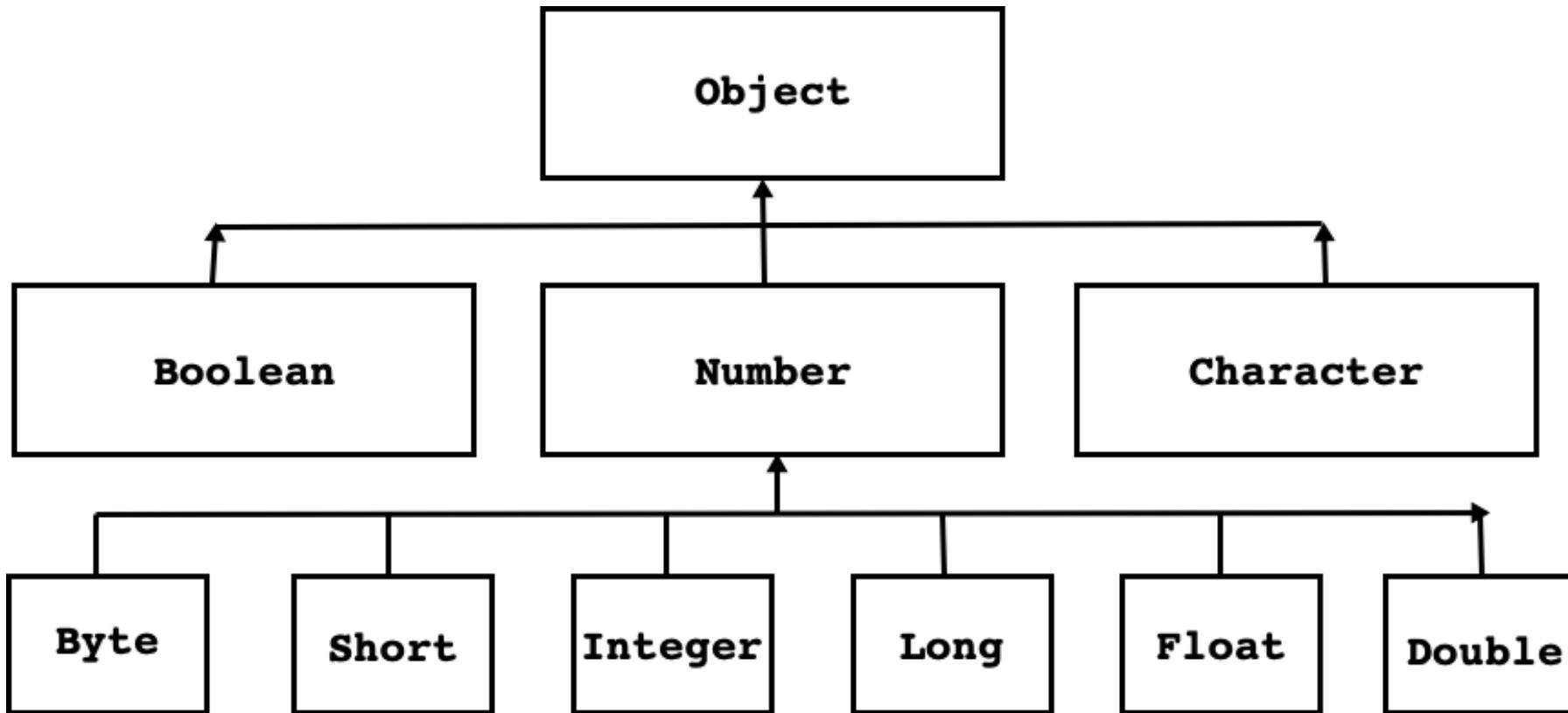
Wrapper class

- In Java, primitive types are not classes. But for every primitive type, Java has defined a class. It is called wrapper class.
- All wrapper classes are final.
- All wrapper classes are declared in **java.lang** package.
- Uses of Wrapper class
 1. To parse string(i.e. to convert state of string into numeric type).
example :

```
int num = Integer.parseInt("123")
float val = Float.parseFloat("125.34f");
double d = Double.parseDouble("42.3d");
```
 1. To store value of primitive type into instance of generic class, type argument must be wrapper class.
 - **Stack<int> stk = new Stack<int>(); //Not OK**
 - **Stack<Integer> stk = new Stack<Integer>(); //OK**



Wrapper class



Widening

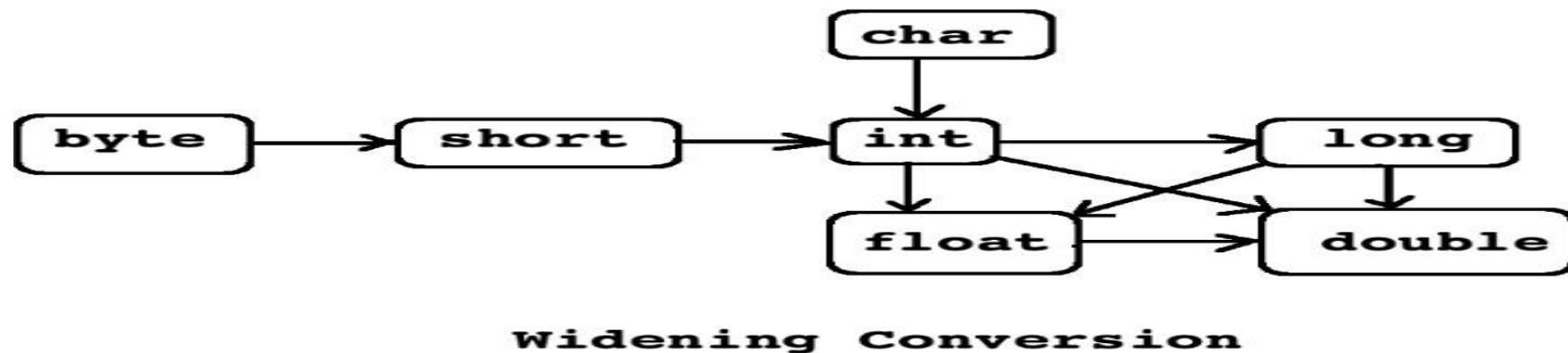
- Process of converting value of variable of narrower type into wider type is called widening.
- E.g. Converting int to double
-

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

- In case of widening, there is no loss of data
- So , explicit type casting is optional.



Widening



The range of values that can be represented by a float or double is much larger than the range that can be represented by a long. Although one might lose significant digits when converting from a long to a float, it is still a "widening" operation because the range is wider.

A widening conversion of an int or a long value to float, or of a long value to double, may result in loss of precision - that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode.

Note that a double can exactly represent every possible int value.

long --->float ---is considered automatic type of conversion(since float data type can hold larger range of values than long data type)



Rules

- src & dest - must be compatible, typically dest data type must be able to store larger magnitude of values than that of src data type.
- 1. Any arithmetic operation involving byte,short --- automatically promoted to --int
- 2. int & long ---> long
- 3. long & float ---> float
- 4. byte,short.....& float & double---> double



Narrowing (Forced Conversion)

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

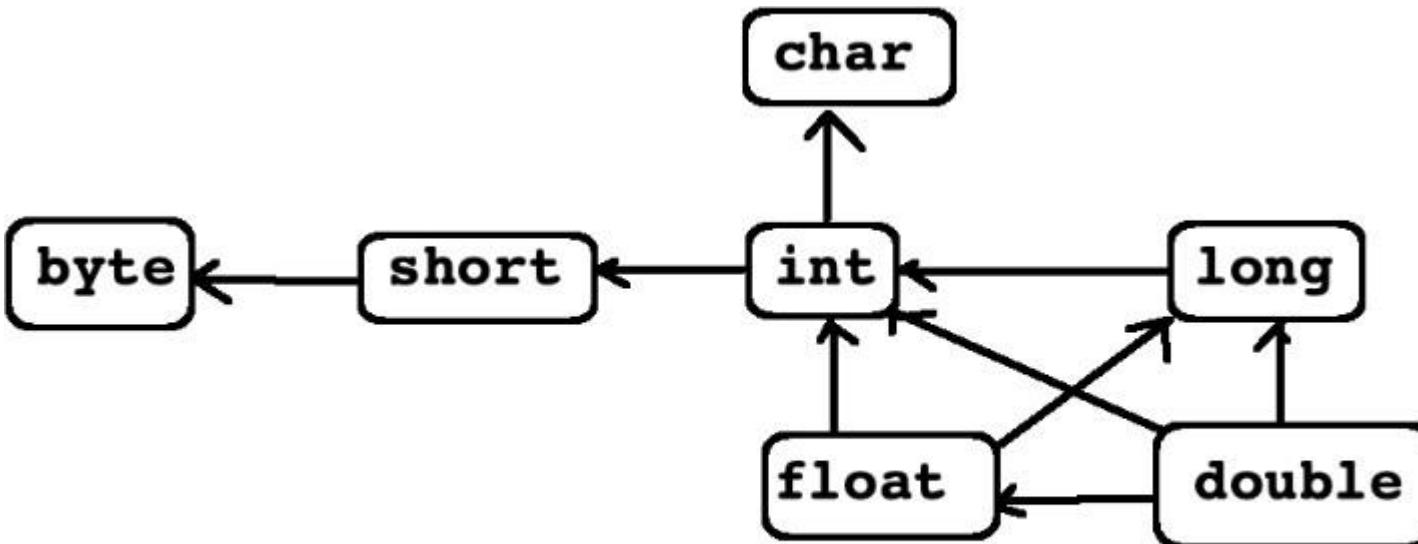
```
public static void main(String[] args) {  
    double num1 = 10.5;  
  
    int num2 = ( int )num1; //Narrowing : OK  
    //int num2 = num1; //Narrowing : NOT OK  
  
    System.out.println("Num2      :      "+num2);  
}
```

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

Note : In case of narrowing and widening both variables are of primitive



Narrowing



eg ---

double ---> int

float --> long

double ---> float

Narrowing Conversion.

Operators

- Arithmetic Operators
- Unary Operators
- Assignment Operator
- Relational Operators
- Logical Operators
- Ternary Operator
- Bitwise Operators
- Shift Operators



Operators Cont..

- **Arithmetic Operators**
 - They are used to perform simple arithmetic operations on primitive data types.
 - * : Multiplication
 - / : Division
 - % : Modulo
 - + : Addition
 - : Subtraction
- **Unary Operators**
 - Unary operators need only one operand. They are used to increment, decrement or negate a value.
 - Unary minus, used for negating the values.
 - eg : int a=20; int b=-a;
 - ++ : Increment operator, used for incrementing the value by 1. There are two varieties of increment operator.
 - Post-Increment : Value is first used for computing the result and then incremented.
 - Pre-Increment : Value is incremented first and then result is computed.
 - -- : Decrement operator, used for decrementing the value by 1. There are two varieties of decrement operator.
 - Post-decrement : Value is first used for computing the result and then decremented.
 - Pre-Decrement : Value is decremented first and then result is computed.
 - ! : Logical not operator, used for inverting a boolean value.
 - eg : boolean jobDone=true; boolean flag=!jobDone; System.out.println(flag);



Operators Cont..

- **Assignment Operators**
 - '=' Assignment operator is used to assign a value to any variable. It has a right to left associativity.
 - Eg. int val = 500;
 - assignment operator can be combined with other operators to build a shorter version of statement called Compound Statement.
 - +=, for adding left operand with right operand and then assigning it to variable on the left.
 - -=, for subtracting left operand with right operand and then assigning it to variable on the left.
 - *=, for multiplying left operand with right operand and then assigning it to variable on the left.
 - /=, for dividing left operand with right operand and then assigning it to variable on the left.
 - %=, for assigning modulo of left operand with right operand and then assigning it to variable on the left.
- **Relational Operators**
 - These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are used in looping statements and conditional if else statements.
 - ==, Equal to : returns true if left hand side is equal to right hand side.
 - !=, Not Equal to : returns true if left hand side is not equal to right hand side.
 - <, less than : returns true if left hand side is less than right hand side.
 - <=, less than or equal to : returns true if left hand side is less than or equal to right hand side.
 - >, Greater than : returns true if left hand side is greater than right hand side.
 - >=, Greater than or equal to: returns true if left hand side is greater than or equal to right hand side.



Operator Cont...

- **Logical Operators :**
 - These operators are used to perform “logical AND” and “logical OR” operation, i.e. the function similar to AND gate and OR gate in digital electronics.
 &&, Logical AND : returns true when both conditions are true.
 ||, Logical OR : returns true if at least one condition is true.
 - eg :

```
int data=100;
      int data2=50;
      if(data > 60 && data2 < 100)
          System.out.println("test performed...");
      else
          System.out.println("test not performed...");
```
- **Ternary operator :** Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary.
 - General format is :
`condition ? if true : if false`
The above statement means that if the condition evaluates to true, then execute the statements after the ‘?’
else execute the statements after the ‘:’.
eg :

```
int data=100;
      System.out.println(data>100?"Yes":"No");
```



Operators Cont..

- **Bitwise Operators :**
 - These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.
 - &, Bitwise AND operator: returns bit by bit AND of input values.
 - |, Bitwise OR operator: returns bit by bit OR of input values.
 - ^, Bitwise XOR operator: returns bit by bit XOR of input values.
 - ~, Bitwise Complement Operator: This is a unary operator which returns the one's compliment representation of the input value, i.e. with all bits inverted.
 - Eg : String binary[] = { "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
 "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111" };
 int a = 3; // 0 + 2 + 1 or 0011 in binary
 int b = 6; // 4 + 2 + 0 or 0110 in binary
 int c = a | b;
 int d = a & b;
 int e = a ^ b;
 System.out.println(" a = " + binary[a]);
 System.out.println(" b = " + binary[b]);
 System.out.println(" a|b = " + binary[c]);
 System.out.println(" a&b = " + binary[d]);
 System.out.println(" a^b = " + binary[e]); }



Operators Cont...

- **Shift Operators :**
 - These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two.
 - <<, Left shift operator: shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.

eg :

```
int a = 25;  
System.out.println(a<<4); //25 * 16 = 400  
a=-25;  
System.out.println(a<<4);//-25 * 16 = -400
```

- Signed right shift operator : The signed right shift operator '>>' uses the sign bit to fill the trailing positions. For example, if the number is positive then 0 will be used to fill the trailing positions and if the number is negative then 1 will be used to fill the trailing positions.



Example Shift Operations

- Assume if $a = 60$ and $b = -60$; now in binary format, they will be as follows –
- $a = 0000\ 0000\ 0000\ 0000\ 0000\ 0011\ 1100$
- $b = 1111\ 1111\ 1111\ 1111\ 1111\ 1100\ 0100$
- In Java, negative numbers are stored as 2's complement.
- Thus $a \gg 1 = 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 1110$
- And $b \gg 1 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0010$
- Unsigned right shift operator
- The unsigned right shift operator ' $>>$ ' do not use the sign bit to fill the trailing positions. It always fills the trailing positions by 0s.
- Thus $a \ggg 1 = 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 1110$
- And $b \ggg 1 = 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0010$





Thank you.

rohan.paramane@sunbeaminfo.com



Agenda

- Installation
- History
- Buzz Words
- Hello world
- Data types
- Operators
- Widening
- Narrowing

Installation

1. Install JDK
2. Install STS

Install both of them from the links provided in the pdf.

Java BuzzWords

1. Simple
2. Secure
3. Portable
4. Object Oriented
5. Robust
6. Architecture Neutral(Platform Independent)
7. Multithreading
8. Interpreted
9. High Performance
10. Distributed
11. Dynamic

History of Java

Java Platforms

- Java SE - Standard Edition
- Core java
- Java EE - Enterprise Edition
- Java ME - Micro Edition
- Java Card

SDK,JDK,JRE,JVM

- SDK - Software Development Kit - Development Tools + Documentation + Libraries + RunTime Environment
- JDK - Java Development Kit
 - Development Tools + Documentation + Libraries + RunTime Environment
- JRE - Java Runtime environment - rt.jar + JVM
- JVM - Java Virtual Machine An engine which helps to execute the java code. We can also call it as Execution Engine

Datatypes

- It defines 3 things
1. Memory
 - How much memory is required to store that data.
 2. Nature
 - What kind of data i can store
 3. Operations
 - What all operations we can perform on these data

- Datatypes are classified into 2 types
 1. Primitive (value Type) byte,short,int,long,float,double,char,boolean
 2. Non Primitive (Reference Type) class,Array,Interface

Wrapper classes

- For every primitive datatype java have defined a class.
- These classes are called as wrapper classes.
- All wrapper classes are final classes
- All wrapperclasses are declared in java.lang package

Widening

- a process of converting narrower type of data into wider type is called as widening
- No need of explicit Typecasting

Narrowing

- a process of convertinf wider type oda data into narrower type is called as narrowing

- It is a forced conversion.
- Explicit Typecasting is required.

Operators

1. Arithmetic Operators

+,-,*,/,%

2. Unary Operators

++,--

3. Relational Operators

==, !=, <, > <=, =>

4. Bitwise Operators

&, | , ~, ^

5. logical operators

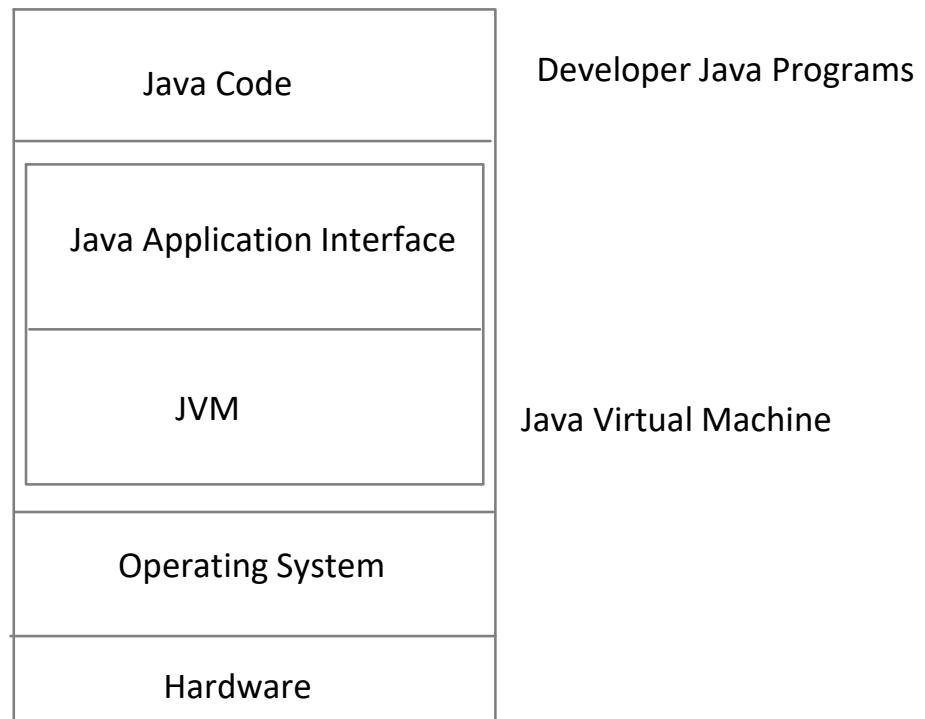
&&, ||, !

6. Assignment Operator

=, +=, -=, *=, /=, %=

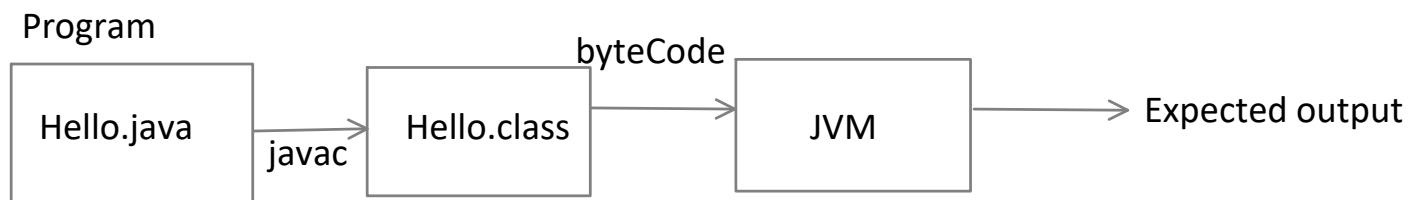
7. Shift operator

>>, <<



Flow of Execution

15 September 2022 10:40 AM





Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane



Agenda

- Boxing
- Unboxing
- Command Line Arguments
- Value type vs Reference type
- Scanner class
- Loops
- If.. else



Boxing

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

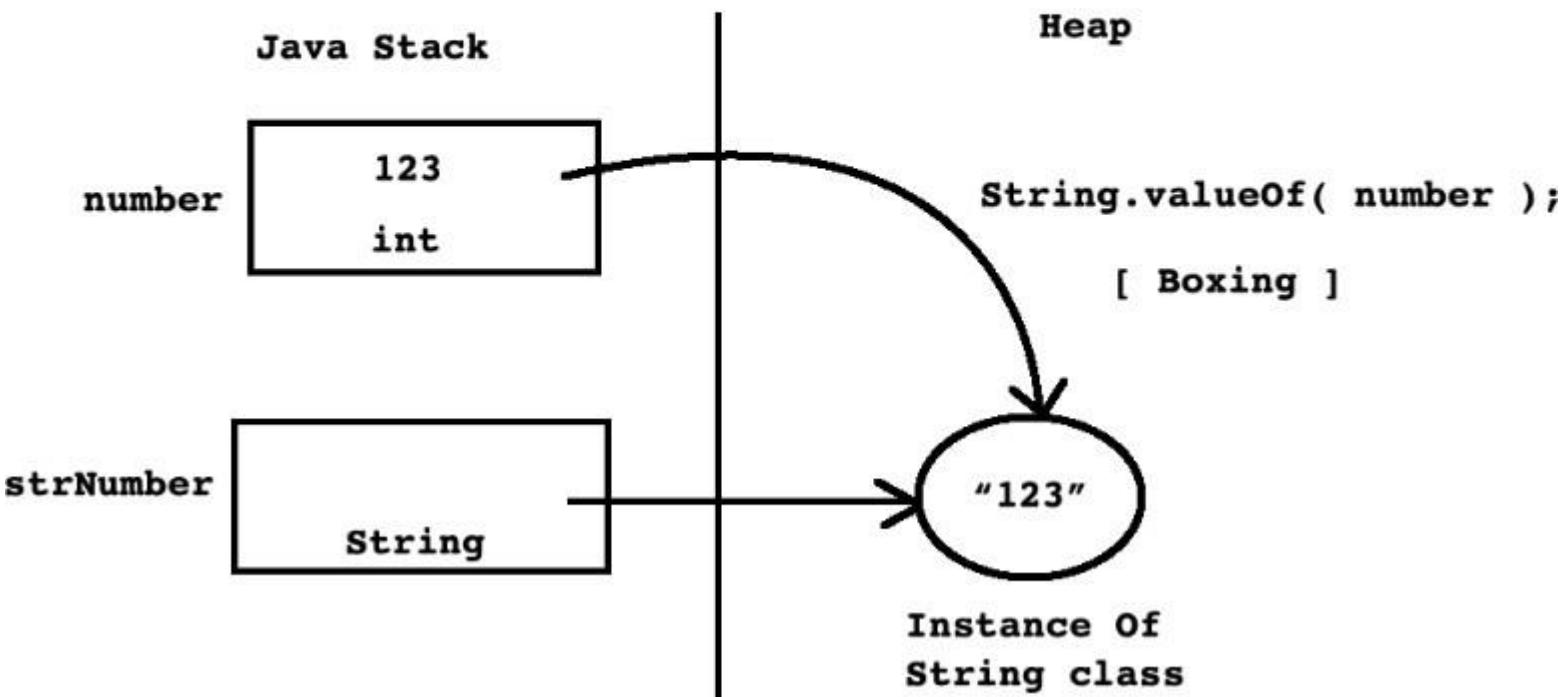
```
public static void main(String[] args) {  
    int number = 123;  
    //String str = Integer.toString( number );      //Boxing : OK  
    String str = String.valueOf(number);           //Boxing : OK  
    System.out.println("Str : " +str);  
}
```

- int n1=10; float f=3.5f; double d1=3.45
- String str1=String.valueOf(n1);
- String str2=String.valueOf(f);
- String str3=String.valueOf(d1);



Boxing

```
int number = 123;  
String strNumber = String.valueOf( number ); //Boxing
```



Unboxing

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

```
public static void main(String[] args) {  
    String str = "123";  
    int number = Integer.parseInt(str); //UnBoxing  
    System.out.println("Number : "+number);  
}
```

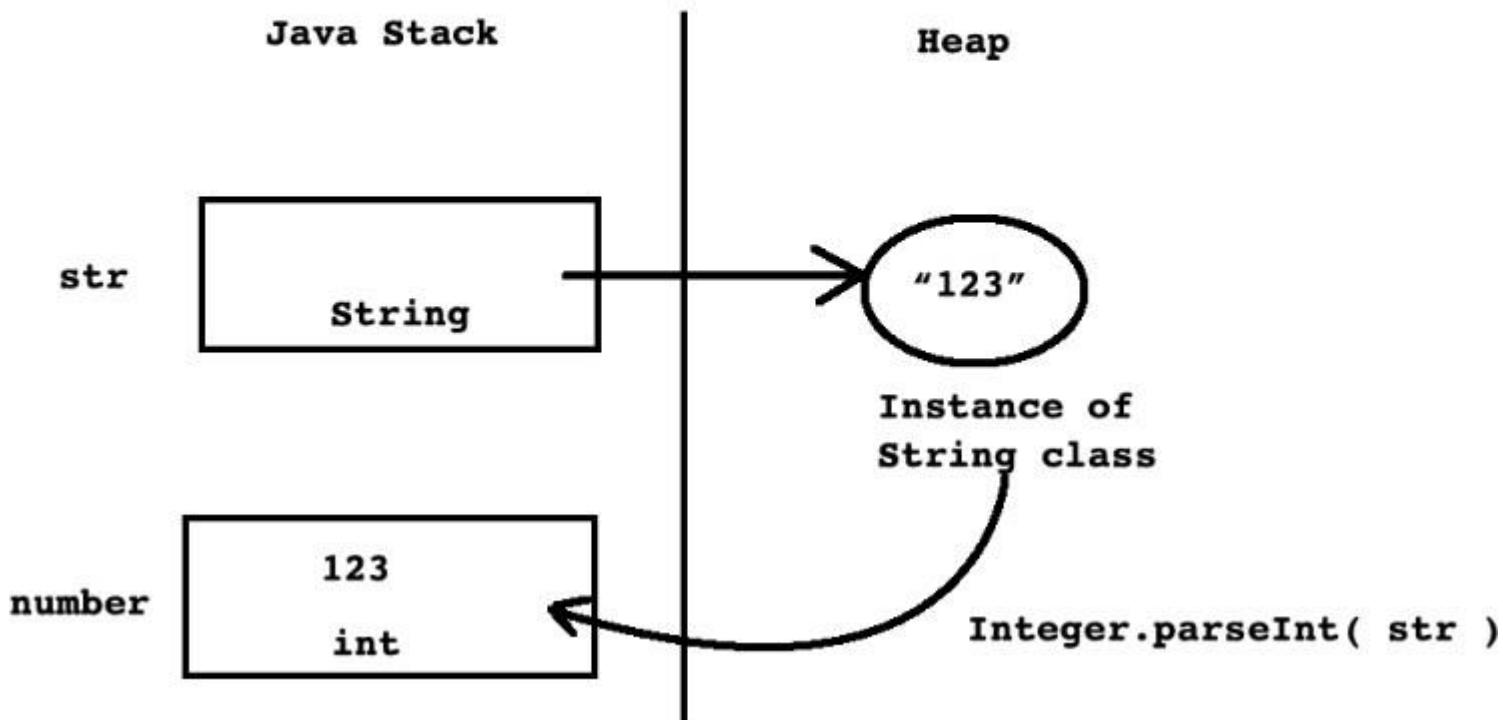
- If string does not contain parseable numeric value then **parseXXX()** method throws **NumberFormatException**.

```
String str = "12c";  
int number = Integer.parseInt(str); //UnBoxing : NumberFormatException
```



Unboxing

```
String str = "123";
int number = Integer.parseInt( str ); //UnBoxing
```



Note : In case of boxing and unboxing one variable is primitive and other Is not primitive



Command line argument

```
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);  
    }  
}
```

- + User input from terminal:
 - java Program 10 20.3f 35.2d (Press enter key)



null

If reference variable contains null value then it is called null reference variable / null object.

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

```
public static void main(String[] args) {  
  
    Employee emp = null; //null reference varibale / null object  
  
    emp.printRecord(); //NullPointerException  
}
```



Value type VS Reference Type

Sr. No.	Value Type	Reference Type
1	primitive type is also called as value type.	Non primitive type is also called as reference type.
2	boolean, byte, char, short, int, float, double, long are primitive/value type.	Interface, class, type variable and array are non primitive/reference type.
3	Variable of value type contains value.	Variable of reference type contains reference.
4	Variable of value type by default contains 0 value.	Variable of reference type by default contain null reference.
5	We can not create variable of value type using new operator.	It is mandatory to use new operator to create instance of reference type.
6	variable of value type get space on Java stack.	Instance of reference type get space on heap section.
7	We can not store null value inside variable of value type.	We can store null value inside variable reference type.
8	In case of copy, value gets copied.	In case of copy, reference gets copied.



if Statement – different syntax options

```
if  
(expression)  
statement;
```

→ A single statement.

```
if  
(expression)  
{  
    statements;  
}
```

→ A block of statements.

```
if  
(expression)  
statement;  
else  
statement;
```

→ Single statement in the if and a single statement in the else.

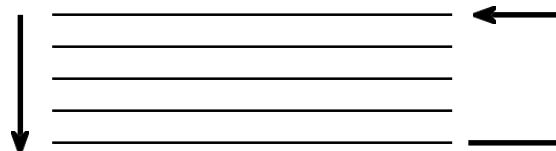
```
if  
(expression)  
    statement;  
else  
{  
    statements;  
}
```

→ A single statement in the if and a block of statements in the else.



Loops

- Loops break the serial execution of the program.
- A group of statements is executed a number of times.



There are three kinds of loops :

- **while**
- **for**
- **do ... while**



While – Loop

- Syntax:

```
while (expression)  
Statement;
```

or

```
while (expression)  
{Statements ; }
```

- The loop continues to iterate as long as the value of expression is true (expression differs from zero).
- Expression is evaluated each time before the loop body is executed.
- The braces { } are used to group declarations and statements together into a compound statement or block, so they are syntactically equivalent to a single statement.



for - Loop

- Syntax:

```
for (expr1 ; expr2 ; expr3)
    statement;
```

or

```
for (expr1 ; expr2 ; expr3)
{
    statements;
}
```

- Is equivalent to:

```
expr1;
while (expr2)
{
    {statements;}
    expr3;
}
```



What is Scanner ?

- A class (java.util.Scanner) that represents text based parser(has inherent small ~ 1K buffer)
- It can parse text data from any source --Console input,Text file , socket, string

e.g. Scanner input = new Scanner(System.in);

```
System.out.print("Enter your name: ");
```

```
String name = input.next();
```

```
System.out.println("Your name is " + name);
```

```
input.close();
```



User Input Using Scanner class.

- Scanner is a final class declared in java.util package.
- Methods of Scanner class:

1. public String nextLine()
2. public int nextInt()
3. public float nextFloat()
4. public double nextDouble()

- How to user Scanner?

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

String name = sc.nextLine();
int empid = sc.nextInt();
float salary = sc.nextFloat();
```





Thank you.

Rohan . paramane@sunbeaminfo . com



Agenda

- Wrapper Class
- Value Type and Reference Type
- Boxing and Unboxing
- Command Line arguments
- Take input from user(Console,Scanner class)
- Loops
- break

Wrapper Class

- package - java.lang
- Integer
- Float

Memory Sections

- Stack-Local Variables
- Data - Global and Static Variables
- Heap - Variables with Dynamically allocated memory

Boxing

Automatic Conversion of primitive type of data in to equivalent Wrapper type type is called as boxing

UnBoxing

Automatic Conversion of reference type of data in to equivalent primitive type is called as unboxing

1. Abstraction - knowing only essential things

- a. Driving a car
- b. ATM
- c. Whats app
- d. Learning Java
- e. Operating Electronics
- f. Using operating system

Modularity

Dividing the entire task into smaller modules(sub tasks)

Hirerachy

2. Encapsulation

OOP-Minor Pillars

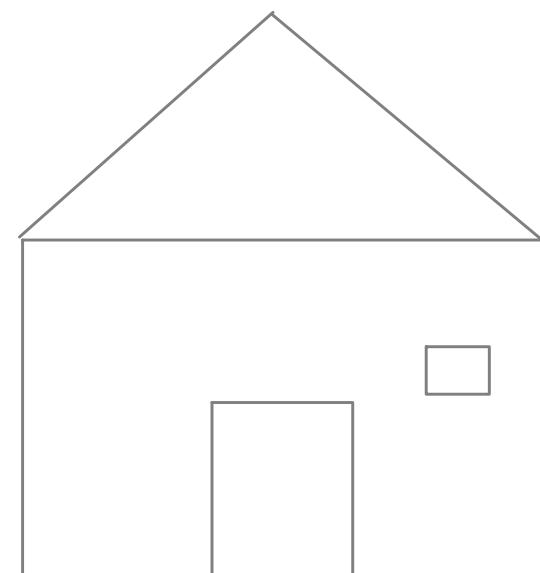
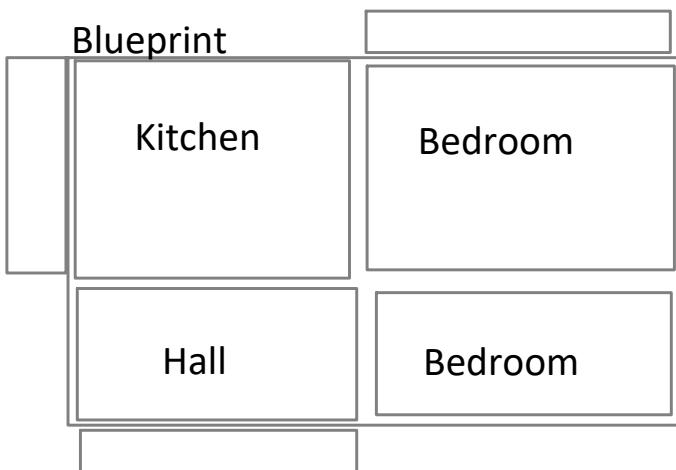
17 September 2022 10:39 AM

1. Typing/ Polymorphism

- a. Smart Watch
- b. Laptop
- c. Smart TV
- d. Omni
- e. Mask
- f. Bed

2. Persistance

3. Concurrency



House - Physical Existence



Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane



Agenda

- Switch Case
- do..while
- Break and continue
- OOA,OOD,OOP
- OOPs Concept
- Class
- Object
- Access modifier



switch Statement

- The nested if can become complicated and unreadable.
- The switch statement is an alternative to the nested if.
- Syntax:

```
switch(expression)
{
    case constant expr:
        statement(s);
        [break;]
    case constant expr:
        statement(s);
        [break;]
    case constant expr:
        statement(s);
        [break;]
    default :
        statement(s);
        [break;]
}
```

- Usually, but not always, the last statement of a case is break.
- default case is optional.



do while Loop

- Syntax:

```
do
{
    Statements;
}while (expression);
```

- The condition expression for looping is evaluated only after the loop body had executed.



break Statement

- We have seen how to use the break statement within the switch statement.
- A break statement causes an exit from the innermost containing while, do, for or switch statement.



continue Statement

- In some situations, you might want to skip to the next iteration of a loop without finishing the current iteration.
- The **continue** statement allows you to do that.
- When encountered, **continue** skips over the remaining statements of the loop, but **continues** to the next iteration of the loop.



Object-oriented software development (OOsd)

- In the past, the problems faced by software development were relatively simple, from task analysis to programming, and then to the debugging of the program, if its not too big it can be done by one person or a group.
- With the rapid increase of software scale, software personnel faces the problem that is very complicated, and there are many factors that need to be considered.
- The errors generated and hidden errors may reach an astonishing degree, this is not something that can be solved in the programming stage.
- Need to standardize the entire software development process and clarify the software
- The tasks of each stage in the development process, while ensuring the correctness of the work of the previous stage, proceed to the next stage work.
- This is the problem that software engineering needs to study and solve. Object-oriented software development and engineering include the following parts:



1.Object oriented analysis (OOA)

- The first step of Object-oriented software development is Object-Oriented Analysis (OOA)
- In the system analysis stage of software engineering, system analysts must integrate with users to make precise Accurate analysis and clear description summarize what the system should do (not how) from a macro perspective.
- Face right The analysis of the image should be based on object-oriented concepts and methods.
- In the analysis of the task, from the objective existence of things and things The relationship between the related objects (including the attributes and behaviors of the objects) and the relationship between the objects are summarized, and the Objects with the same attributes and behaviors are represented by a class.
- Establish a need to reflect the real work situation Seek a model. The model formed at this stage is relatively rough (rather than fine).



2.Object oriented design (OOD)

- The second step of Object-oriented software development is Object-Oriented Design (OOD),
- According to the demand model formed in the object-oriented analysis stage, each part is specifically designed.
- The design of the line class, the design of the class may contain multiple levels (using inheritance).
- Then these classes are based Put forward the ideas and methods of program design, including the design of algorithms.
- In the design stage, no specific plan is involved. Computer language, but a more general description tool (such as pseudo code or flowchart) to describe.



3.Object-oriented programming (OOP)

- The third step of Object-oriented software development is Object-oriented Programming (OOP), According to the results of object-oriented design, to write it into a program in a computer language, it is obvious that object-oriented Computer language (e.g. C++), Otherwise, the requirements of object-oriented design cannot be achieved.
- It is a programming methodology to organize complex program in to simple program in terms of classes and object such methodology is called oops.
- It is a programming methodology to organized complex program into simple program by using concept of abstraction , encapsulation , polymorphism and inheritance.
- Languages which support abstraction , encapsulation polymorphism and inheritance are called oop language.



Major pillars of oops

- **Abstraction**

- getting only essential things and hiding unnecessary details is called as abstraction.
- Abstraction always describe outer behavior of object.
- In console application when we give call to function in to the main function , it represents the abstraction

- **Encapsulation**

- binding of data and code together is called as encapsulation.
- Implementation of abstraction is called encapsulation.
- Encapsulation always describe inner behavior of object
- Function call is abstraction
- Function definition is encapsulation.
- Information hiding
 - Data : unprocessed raw material is called as data.
 - Process data is called as information.
 - Hiding information from user is called information hiding.
 - In c++ we used access Specifier to provide information hiding.

- **Modularity**

- Dividing programs into small modules for the purpose of simplicity is called modularity.

- **Hierarchy (Inheritance [is-a] , Composition [has-a] , Aggregation[has-a], Dependancy)**

- Hierarchy is ranking or ordering of abstractions.
- Main purpose of hierarchy is to achieve re-usability.



Minor pillars of oops

- **Polymorphism (Typing)**

- One interface having multiple forms is called as polymorphism.
- Polymorphism have two types

- 1. **Compile time polymorphism**

when the call to the function resolved at compile time it is called as compile time polymorphism. And it is achieved by using function overloading and operator overloading

- 2. **Runtime polymorphism.**

when the call to the function resolved at run time it is called as run time polymorphism. And it is achieved by using function overriding.

- Compile time / Static polymorphism / Static binding / Early binding / Weak typing / False Polymorphism
- Run time / Dynamic polymorphism / Dynamic binding / Late binding / Strong typing / True polymorphism

- Concurrency

- The concurrency problem arises when multiple threads simultaneously access same object.
 - You need to take care of object synchronization when concurrency is introduced in the system.

- Persistence

- It is property by which object maintains its state across time and space.
 - It talks about concept of serialization and also about transferring object across network.



- Consider following examples:
 1. day, month, year - related to - Date
 2. hour, minute, second - related to - Time
 3. red, green, blue - related to Color
 4. real, imag - related to - Complex
 5. xPosition, yPosition - related to Point
 6. number, type, balance - related to Account
 7. name, id, salary - related to Employee
- If we want to group related data elements together then we should use/ define class in Java.

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

```
class Employee{  
    String name;  //Field  
    int id;       //Field  
    float salary; //Field  
}
```

- class is a non primitive/reference type in Java.
- Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

A **class** is a user defined blueprint or prototype or template : from which objects are created.

It represents the set of properties(DATA) and methods(ACTIONs) that are common to all objects of one type.

Class declaration includes

1. Access specifiers : A class can be public or has default access
2. Class name: The name should begin with a capital letter & then follow camel case convention
3. Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. (Implicit super class of all java classes is java.lang.Object)
4. Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

eg : public class Emp extends Person implements Runnable,Comparable{...}

5. Body: The class body surrounded by braces, { }.
6. Constructors are used for initializing state of the new object/s.
7. Fields are variables that provides the state of the class and its objects
8. Methods are used to implement the behavior of the class and its objects.

eg : Student,Employee,Flight,PurchaseOrder, Shape ,BankAccount.....



- It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which interact by invoking methods.
- An object consists of :
 - State : It is represented by attributes of an object. (properties of an object) / instance variables(non static)
 - Behavior : It is represented by methods of an object (actions upon data)
 - Identity : It gives a unique identity to an object and enables one object to interact with other objects. eg : Emp id / Student PRN / Invoice No
- Creating an object
 - The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

- **Field**
 - Ø A variable declared inside class / class scope is called a field.
 - Ø Field is also called as attribute or property.
- **Method**
 - Ø A function implemented inside class/class scope is called as method.
 - Ø Method is also called as operation, behavior or message.
- **Class**
 - Ø Class is a collection of fields and methods.
 - Ø Class can contain
 1. Nested Type
 2. Field
 3. Constructor
 4. Method
- **Instance** : In Java, Object is also called as instance.

Note: All stand-alone C++ programs require a function named main and can have numerous other functions. Java does not have stand alone functions, all functions (called methods) are members of a class. All classes in Java ultimately inherit from the Object class, while it is possible to create inheritance trees that are completely unrelated to one another in C++. In this sense , Java is a pure Object oriented language, while C++ is a mixture of Object oriented and structure language.

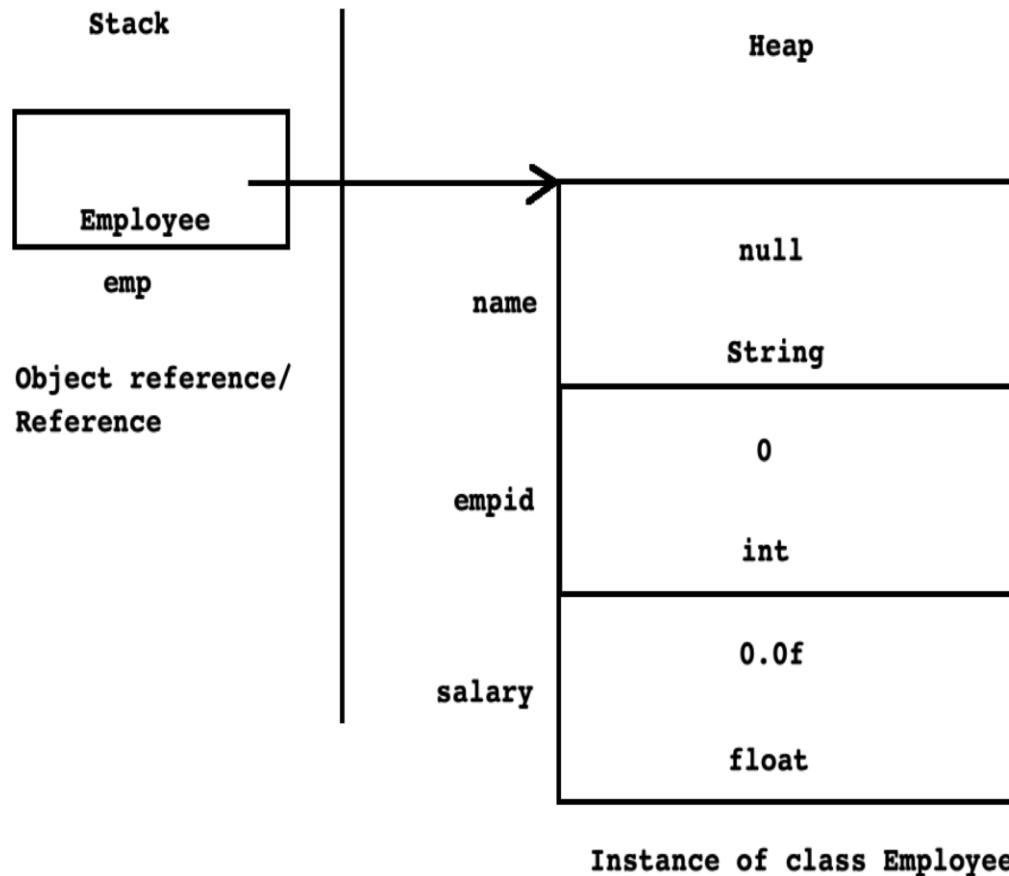


Instantiation

- Process of creating instance/object from a class is called as instantiation.
- In C programming language
 - Ø Syntax : struct StructureName identifier_name; struct Employee emp;
- In C++ programming language
 - Ø Syntax : [class] ClassName identifier_name;
 - Ø Employee emp;
- In Java programming language
 - Ø Syntax : ClassName identifier_name = new ClassName();
 - Ø Employee emp = new Employee();
- **Every instance on heap section is anonymous.**

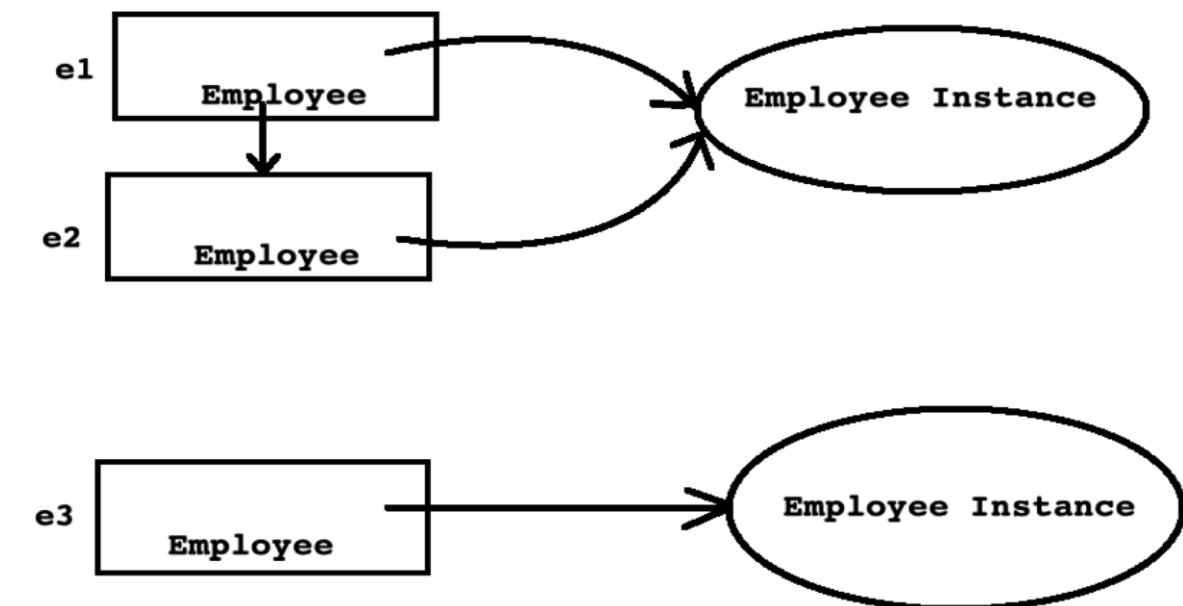


Instantiation



For eg :

1. `Employee e1 = new Employee();`
2. `Employee e2 = e1;`
3. `Employee e3 = new Employee();`



Coding Convention

- **Pascal Case Coding/Naming Convention:**

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



Coding Convention

- **Camel Case Coding/Naming Convention:**

- Example
 - 1. main
 - 2. parseInt
 - 3. showInputDialog
 - 4. addNumberOfDays
- In this case, excluding first word, first character of each word must in upper case.
- We should use this convention for:
 - 1. Method Parameter and Local variable
 - 2. Field
 - 3. Method
 - 4. Reference



Coding Convention

- **Naming Convention for package:**

- We can specify name of the package in uppercase as well as lower-case. But generally it is mentioned in lower case.
- Example
 - 1. java.lang
 - 2. java.lang.reflect
 - 3. java.util
 - 4. java.io
 - 5. java.net
 - 6. java.sql



S u **Coding Convention**

- **Naming Convention for constant variable and enum constant:**
 - Example
 - 1. public static final int SIZE;
 - 2. enum Color{ RED, GREEN, BLUE }
 - 3. Name of the final variable and name of the enum constant should be in upper case.



Coding Convention

- **Pascal Case Coding/Naming Convention:**

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



Coding Convention

- **Camel Case Coding/Naming Convention:**

- Example
 - 1. main
 - 2. parseInt
 - 3. showInputDialog
 - 4. addNumberOfDays
- In this case, excluding first word, first character of each word must in upper case.
- We should use this convention for:
 - 1. Method Parameter and Local variable
 - 2. Field
 - 3. Method
 - 4. Reference



Coding Convention

- **Naming Convention for package:**

- We can specify name of the package in uppercase as well as lower-case. But generally it is mentioned in lower case.
- Example
 - 1. java.lang
 - 2. java.lang.reflect
 - 3. java.util
 - 4. java.io
 - 5. java.net
 - 6. java.sql



Coding Convention

- **Naming Convention for constant variable and enum constant:**
 - Example
 - 1. public static final int SIZE;
 - 2. enum Color{ RED, GREEN, BLUE }
 - 3. Name of the final variable and name of the enum constant should be in upper case.



Access Modifier

- If we want to control visibility of members of class then we should use access modifier.
- There are 4 access modifiers in Java:
 1. private
 2. package-level private / default
 3. protected
 4. public

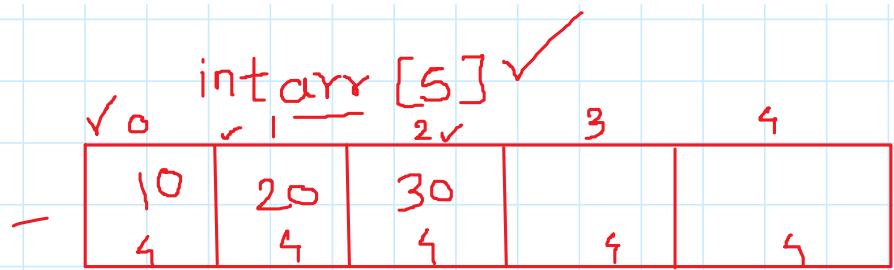
Access Modifiers	Same Package			Different Package	
	Same class	Sub class	Non sub cass	Sub class	Non Sub class
private	A	NA	NA	NA	NA
package level private/Default	A	A	A	NA	NA
protected	A	A	A	A	NA
public	A	A	A	A	A



Thank you.

Rohan . paramane@sunbeaminfo . com

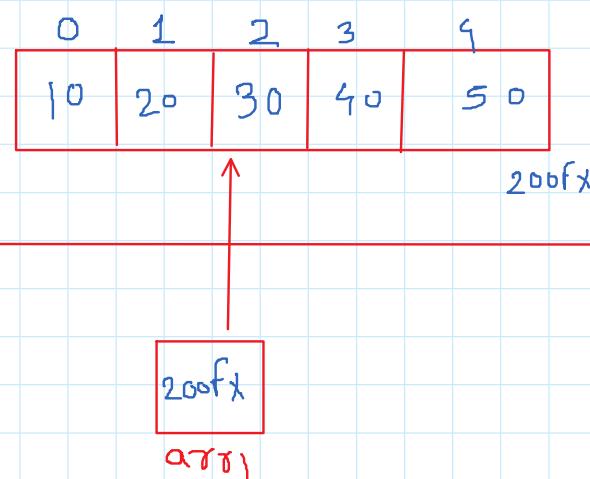




20 bytes

✓ arr[0] = 10

arr[1] = 20



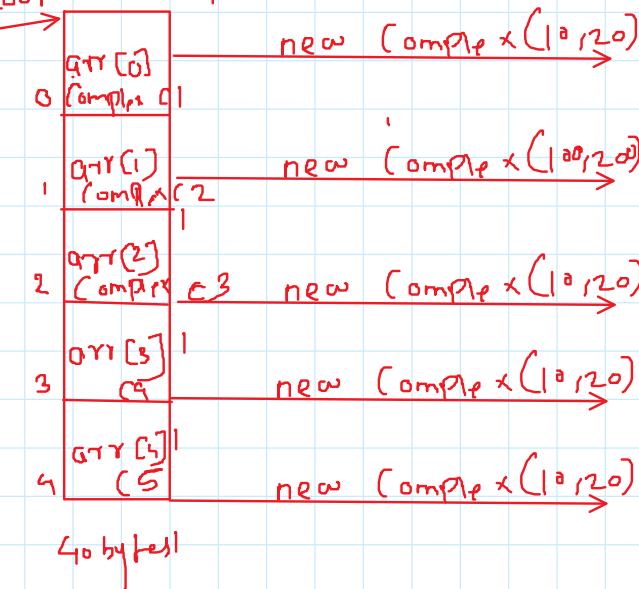
Stack

`2dofx`

`ATR`

heap

`2dofx Complex`

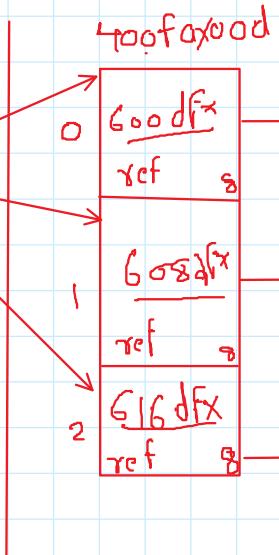


real	imag
1.0	2.0
1.00	2.00
1.0	2.0
1.0	2.0
1.0	2.0

$\text{Complex arr[3] = new Complex[3];}$

Stack

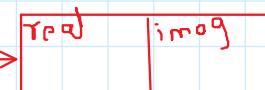
GoodFX
200FX



heap

`new Complex(10,20);`

GoodFX



`new Complex(10,20);`

GoodFX

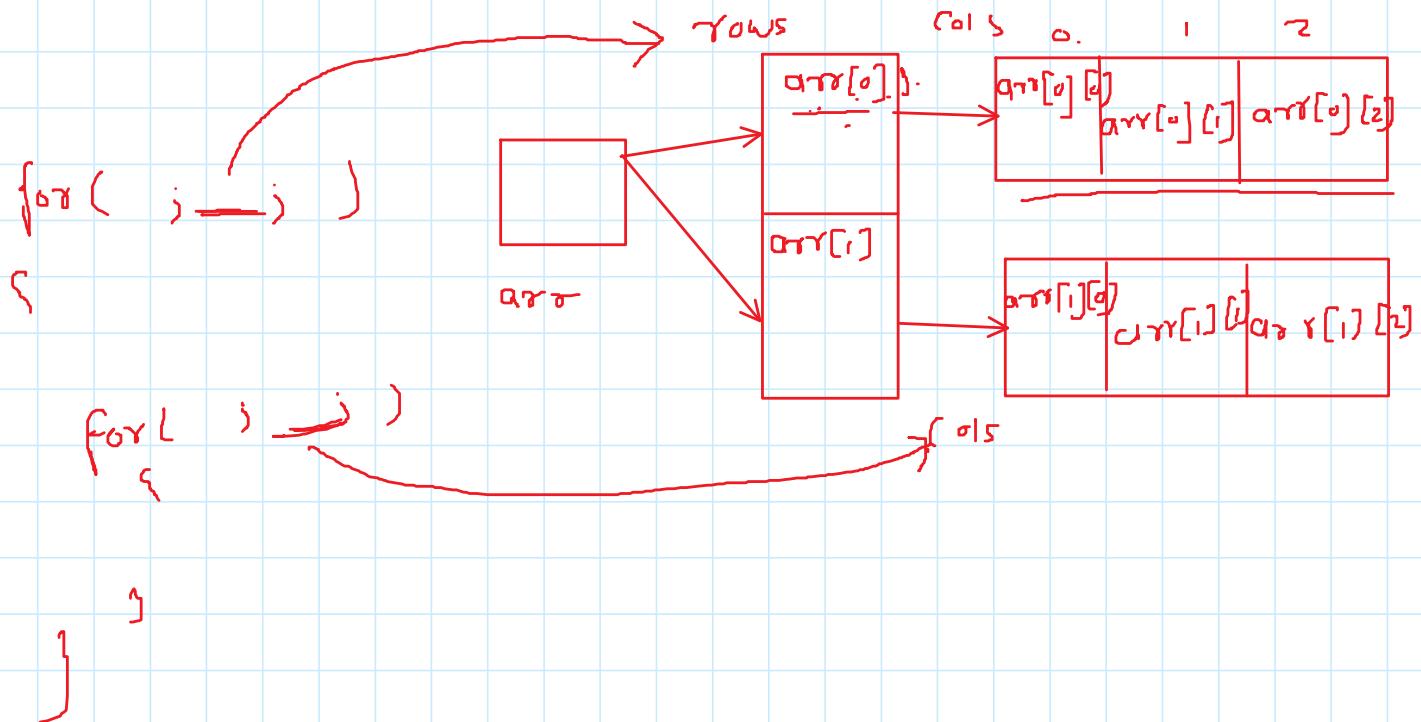


`new Complex(10,20);`

GoodFX



`int arr[2][3].j`



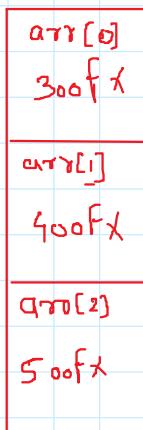
Stack

200fx

arr[3][]

200fx

heap



new int[2]

200fx

700 704

0

1

10

20

new int[3]

200fx 0

1

2

30

40

50

new int[4]

0

1

2

3

60

70

80

90

500fx



Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane



Agenda

- Package
- Import
- Access Modifiers
- this reference
- Ctor
- Methods
- Final field/ variable
- object class
- static



this current object reference

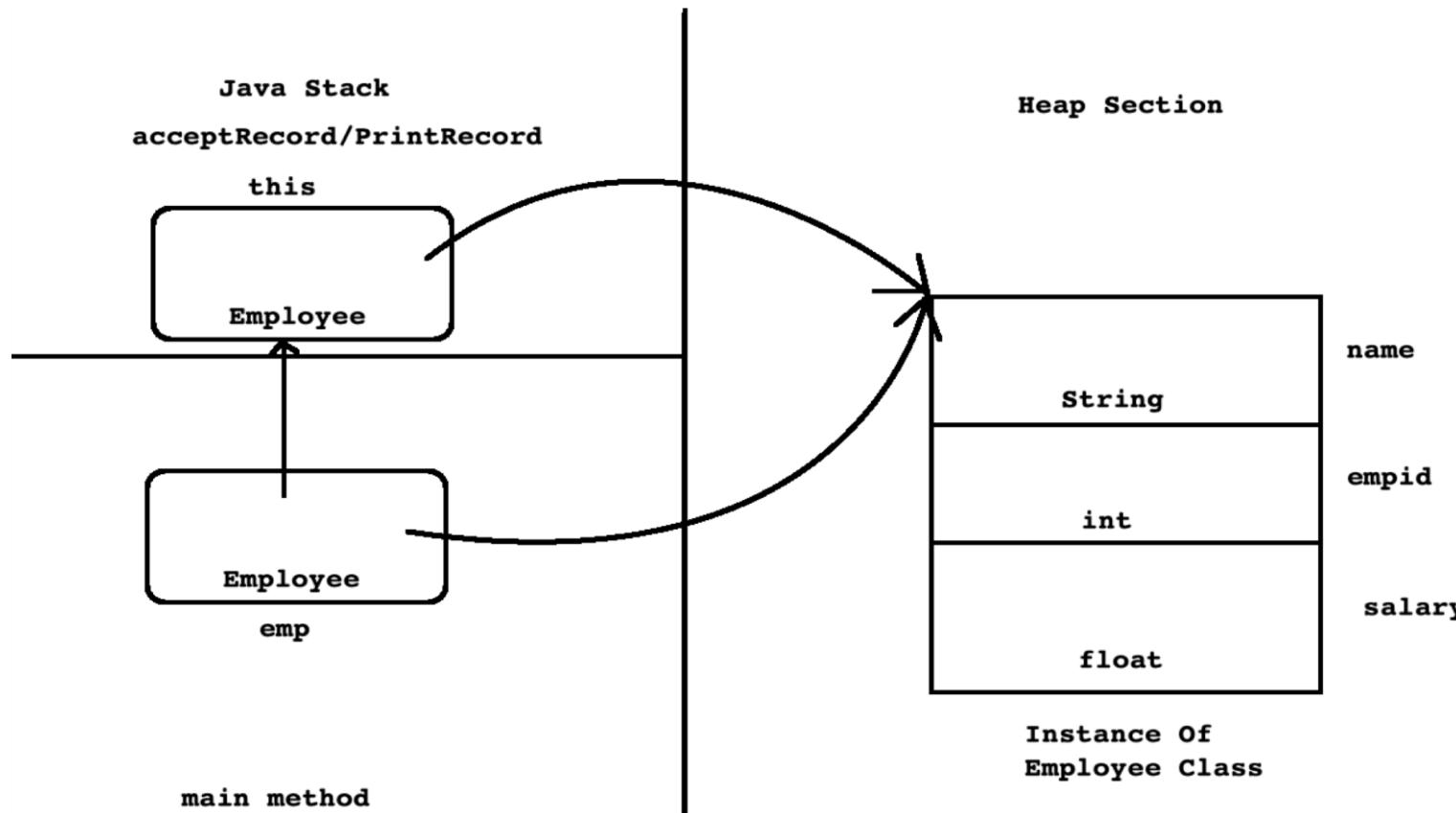
- If we call non static method on instance(actually object reference) then compiler implicitly pass, reference of current/calling instance as a argument to the method implicitly. To store reference of current/calling instance, compiler implicitly declare one reference as a parameter inside method. It is called this reference.
- **Using this reference, non static fields and non static methods are communicating with each other. Hence this reference is considered as a link/connection between them.**
- Definition
 - ∅ “this” is implicit reference variable that is available in every non static method of class which is used to store reference of current/calling instance.
- Inside method, to access members of same class, use this keyword is optional

Uses of this keyword :

1. To unhide , instance variables from method local variables.(to resolve the conflict)
eg : this.name=name;
2. To invoke the constructor , from another overloaded constructor in the same class.(constructor chaining , to avoid duplication)



this reference



this reference

- If name of local variable/parameter and name of field is same then preference is always given to the local variable.

```
class Employee{  
    private String name;  
    private int empid;  
    private float salary;  
    public void initEmployee(String name, int empid, float salary ){  
        this.name = name;  
        this.empid = empid;  
        this.salary = salary;  
    }  
}
```



- If we want to initialize instance then we should define constructor inside class.
- Constructor look like method but it is not considered as method.
- It is special because:
 - Its name is same as class name.
 - It doesn't have any return type.
 - It is designed to be called implicitly
 - It is called once per instance.
- We can not call constructor on instance explicitly
 - **Employee emp = new Employee();**
 - **emp.Employee();//Not Ok**
- **Types of constructor:**
 1. Parameterless constructor
 2. Parameterized constructor
 3. Default constructor .

Parameterless Constructor

- If we define constructor without parameter then it is called as parameterless constructor.
- It is also called as zero argument / user defined default constructor.
- If we create instance without passing argument then parameterless constructor gets called.

```
public Employee( ) {  
    //TODO  
}
```

```
Employee emp = new Employee( ); //Here on instance parameterless ctor will call.
```

Parameterized Constructor

- If we define constructor with parameter then it is called as parameterized constructor.
- If we create instance by passing argument then parameterized constructor gets called.

```
public Employee( String name, int empid, float salary ) {  
    //TODO  
}
```

```
Employee emp = new Employee( "ABC",123, 8000 ); //Here on instance parameterized ctor will call.
```



- If we do not define any constructor inside class then compiler generate one constructor for the class by default. It is called default constructor.
- Compiler generated default constructor is parameterless.
- Compiler never generate default parameterized constructor. In other words, if we want to create instance by passing arguments then we must define parameterized constructor inside class.

Constructor Chaining

- We can call constructor from another constructor. It is called constructor chaining.
- For constructor chaining, we should use this statement.
- this statement must be first statement inside constructor body.
- Using constructor chaining, we can reduce developers effort.

```
class Employee{  
    //TODO : Field declaration  
    public Employee( ){  
        this( "None" , 0 , 8500 );      //Constructor Chaining  
    }  
    public Employee( String name, int empid, float salary ){  
        this.name = name;  
        this.empid = empid;  
        this.salary = salary;  
    }  
}
```



Object class

- It is a non final and concrete class declared in java.lang package.
- In java all the classes(not interfaces)are directly or indirectly extended from java.lang.Object class.
- In other words, java.lang.Object class is ultimate base class/super cosmic base class/root of Java class hierarchy.
- Object class do not extend any class or implement any interface.
- It doesn't contain nested type as well as field.
- It contains default constructor.
 - `Object o = new Object("Hello");` //Not OK
 - `Object o = new Object();` //OK
- Object class contains 11 methods.



Object class

- Consider the following code:

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

- In above code, `java.lang.Object` is direct super class of class `Person`.
- In case class `Employee`, class `Person` is direct super class and class `Object` is indirect super class.



Methods Of Object class

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

6. public **final native** Class<?> getClass();
7. public **final** void wait()throws InterruptedException
8. public **final native** void wait(long timeout)throws InterruptedException
9. public **final** void wait(long timeout, int nanos)throws InterruptedException
10. public **final native** void notify();
11. public **final native** void notifyAll();



S u **toString() method**

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

- Syntax:

➤ **public String toString();**

- If we want to return state of Java instance in String form then we should use `toString()` method.
- Consider definition of `toString` inside Object class:

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



S u n b e a m i n f o t e c h toString() method

- If we do not define `toString()` method inside class then super class's `toString()` method gets call.
- If we do not define `toString()` method inside any class then object class's `toString()` method gets call.
- It return String in following form:
 - **F.Q.ClassName@HashCode**
 - **Example : test.Employee@6d06d69c**
- If we want state of instance then we should override `toString()` method inside class.
- The result in `toString` method should be a concise but informative that is easy for a person to read.
- It is recommended that all subclasses override this method.



Final variable

- In java we do not get const keyword. But we can use final keyword.
- After storing value, if we don't want to modify it then we should declare variable final.

```
public static void main(String[] args) {  
    final int number = 10; //Initialization  
    //number = number + 5; //Not OK  
    System.out.println("Number : "+number );  
}
```

```
public static void main(String[] args) {  
    final int number;  
    number = 10; //Assignment  
    //number = number + 5; //Not OK  
    System.out.println("Number : "+number );  
}
```



Final variable

- We can provide value to the final variable either at compile time or run time.

```
public static void main(String[] args) {  
    Scanner sc = new Scanner( System.in );  
    System.out.print("Number      :      ");  
    final int number = sc.nextInt();      //OK  
    //number = number + 5;      //Not OK  
    System.out.println("Number      :      "+number );  
}
```



Final field

- once initialized, if we don't want to modify state of any field inside any method of the class(including constructor body) then we should declare field final.

```
class Circle{  
    private float area;  
    private float radius = 10;  
    public static final float PI = 3.142f;  
    public void calculateArea( ){  
        this.area = PI * this.radius * this.radius;  
    }  
    public void printRecord( ){  
        System.out.println("Area : "+this.area);  
    }  
}
```

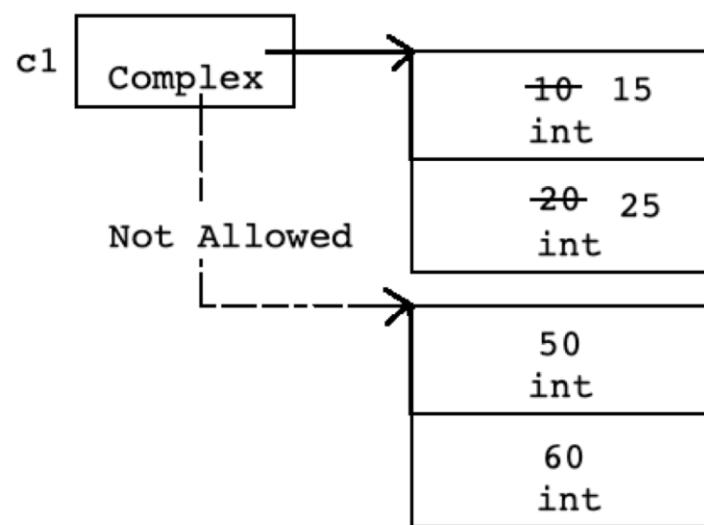
- If we want to declare any field final then we should declare it static also.



Final Reference Variable

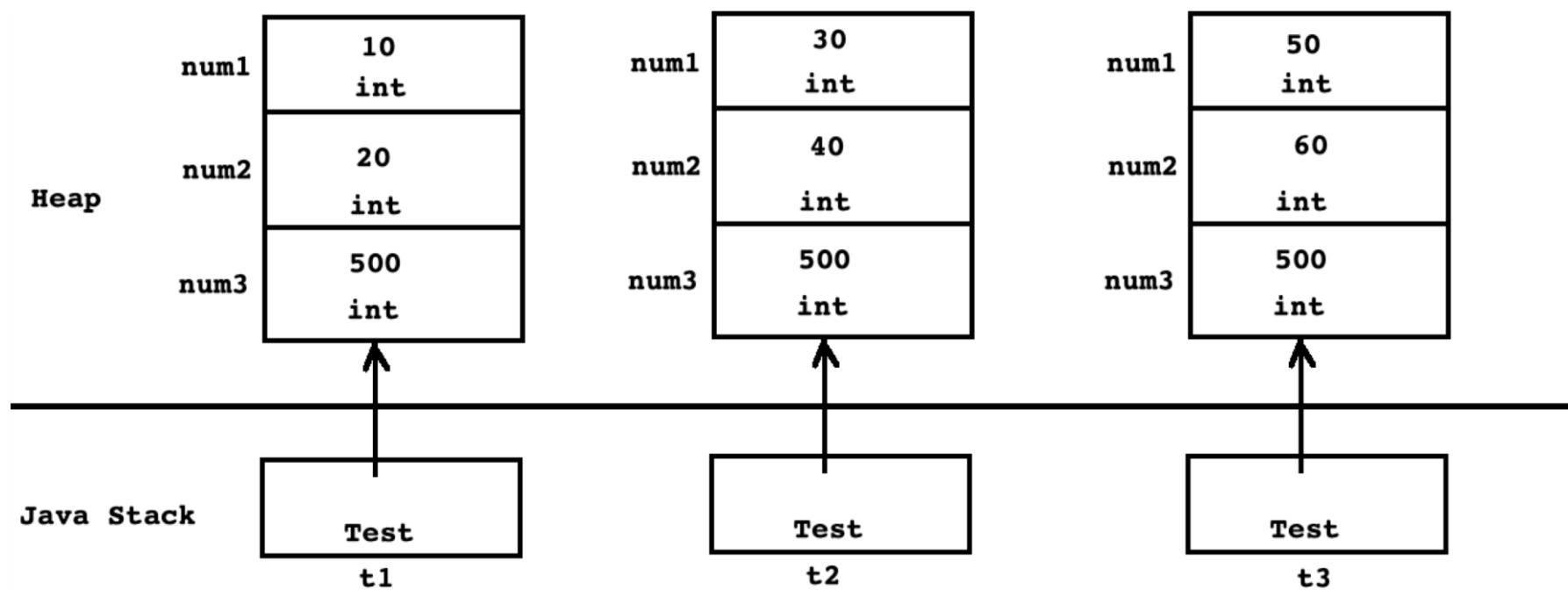
- In Java, we can declare reference final but we can not declare instance final.

```
public static void main(String[] args) {  
    final Complex c1 = new Complex( 10, 20 );  
    c1.setReal(15);  
    c1.setImag(25);  
    //c1 = new Complex(50,60); //Not OK  
    c1.printRecord( );      //15, 25  
}
```

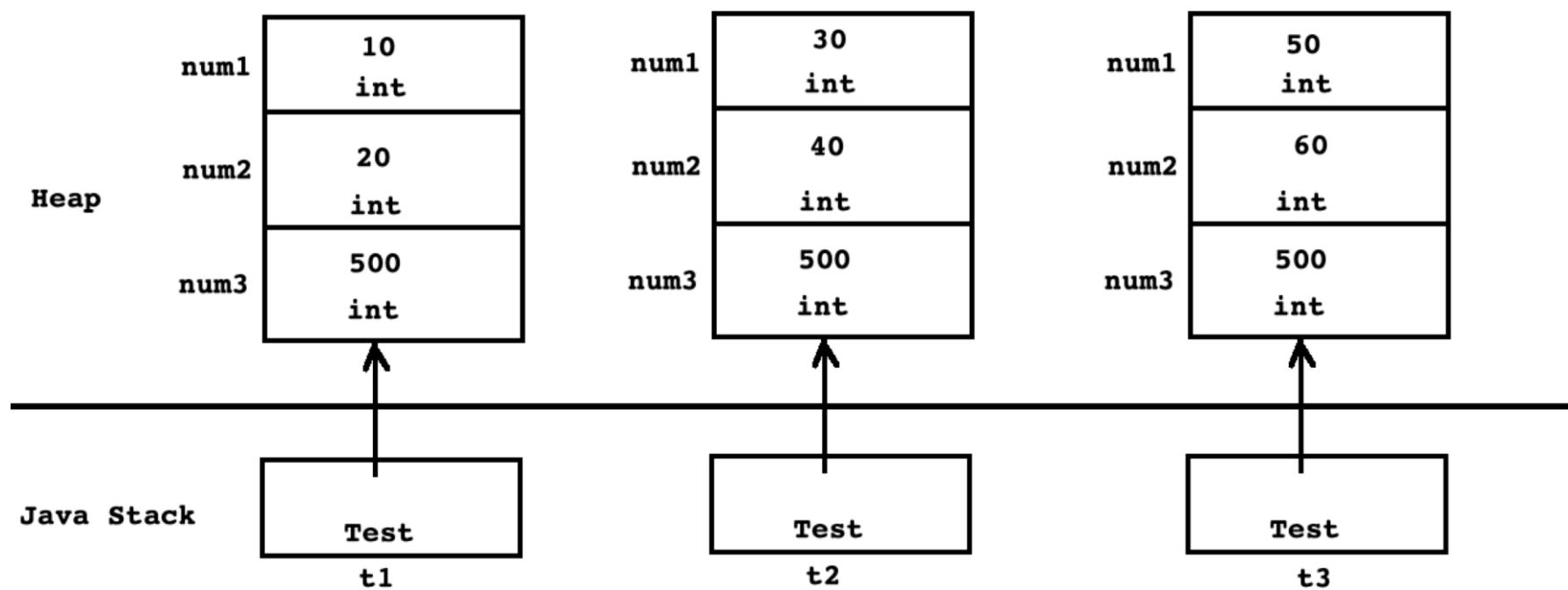


- We can declare method final. It is not allowed to override final method in sub class.
- We can declare class final. It is not allowed to extend final class.

Static Field

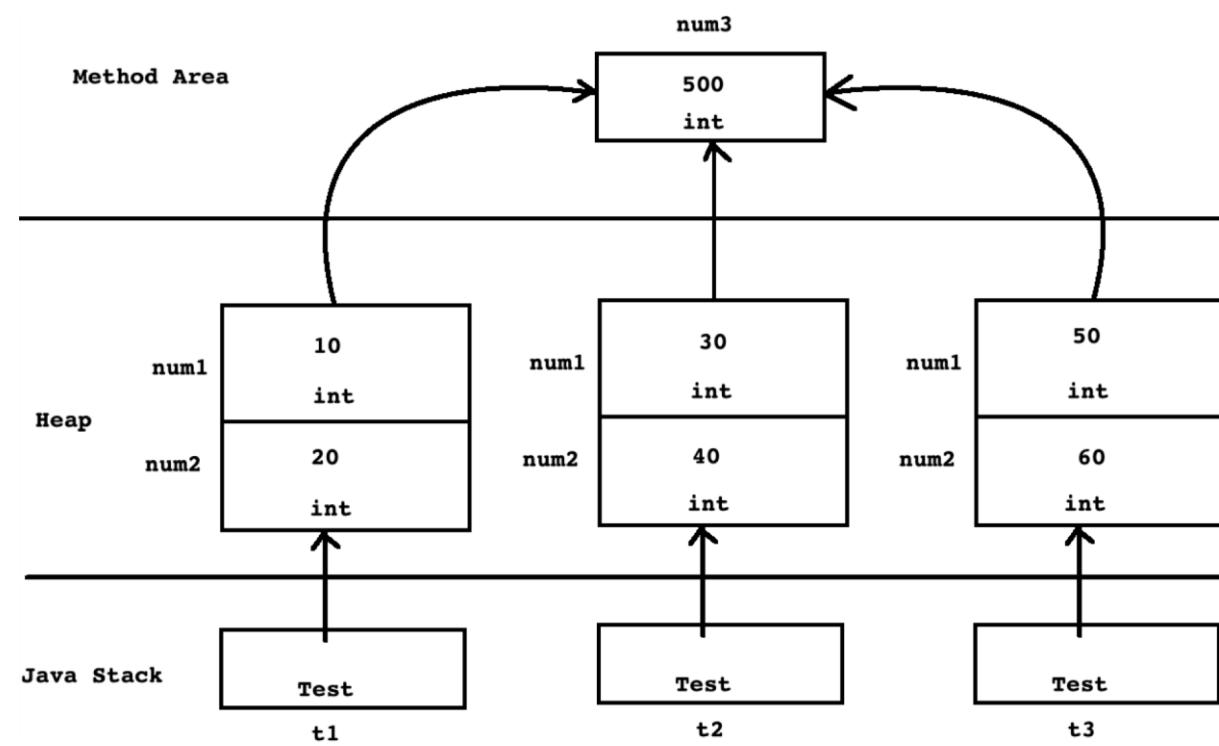


Static Field



Static Field

- If we want to share value of any field inside all the instances of same class then we should declare that field static.



Static Field

- Static field do not get space inside instance rather all the instances of same class share single copy of it.
- Non static Field is also called as instance variable. It gets space once per instance.
- Static Field is also called as class variable. It gets space once per class.
- Static Field gets space once per class during class loading on method area.
- Instance variables are designed to access using object reference.
- Class level variable can be accessed using object reference but it is designed to access using class name and dot operator.



Static Initialization Block

- A *static initialization block* is a normal block of code enclosed in braces, {}, and preceded by the static keyword. Here is an example:

```
static {  
    // whatever code is needed for initialization goes here  
}
```

- A class can have any number of static initialization blocks, and they can appear anywhere in the class body.
- The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.
- There is an alternative to static blocks – you can write a private static method:



Static Method

- To access non static members of the class, we should define non static method inside class.
- Non static method/instance method is designed to call on instance.
- To access static members of the class, we should define static method inside class.
- static method/class level method is designed to call on class name.
- static method do not get this reference:
 1. If we call, non static method on instance then method get this reference.
 2. Static method is designed to call on class name.
 3. Since static method is not designed to call on instance, it doesn't get this reference.



Static Method

- this reference is a link/connection between non static field and non static method.
- Since static method do not get this reference, we can not access non static members inside static method directly. In other words, static method can access static members of the class only.
- Using instance, we can use non static members inside static method.

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



Static Method

- Inside method, If we are going to use this reference then method should be non static otherwise it should be static.

```
class Math{  
    public static int power( int base, int index ){  
        int result = 1;  
        for( int count = 1; count <= index; ++ count ){  
            result = result * base;  
        }  
        return result;  
    }  
}
```

```
class Program{  
    public static void main(String[] args) {  
        int result = Math.power(10, 2);  
        System.out.println("Result : "+result);  
    }  
}
```



Static Import

- If static members belonging to the same class then use of type name and dot operator is optional.

```
package p1;

public class Program{

    private static int number = 10;

    public static void main(String[] args) {
        System.out.println("Number : "+Program.number); //OK      : 10
        System.out.println("Number : "+number); //OK      : 10
    }
}
```



Static Import

- If static members belonging to the different class then use of type name and dot operator is mandatory.
- PI and pow are static members of `java.lang.Math` class. To use `Math` class import statement is not required.
- Consider Following code:

```
package p1;

public class Program{

    public static void main(String[] args) {

        float radius = 10.5f;

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

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



Static Import

- There are situations where you need frequent access to static final fields (constants) and static methods from one or two classes. Prefixing the name of these classes over and over can result in cluttered code. The static import statement gives you a way to import the constants and static methods that you want to use so that you do not need to prefix the name of their class.

- Consider code:

```
package p1;

import static java.lang.System.out;
import static java.lang.Math.*;

public class Program{

    public static void main(String[] args) {

        float radius = 10.5f;

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

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

    }

}
```



Package

- Not necessarily but as shown below, package can contain some or types.
 1. Sub package
 2. Interface
 3. Class
 4. Enum
 5. Exception
 6. Error
 7. Annotation Type



Package Creation

- package is a keyword in Java.
- To define type inside package, it is mandatory write package declaration statement inside .java file.
- Package declaration statement must be first statement inside .
- If we define any type inside package then it is called as packaged type otherwise it will be unpackaged type.
- Any type can be member of single package only.

```
package p1; //OK
class Program{
    //TODO
}
```

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

```
package p1; //OK
package p2; //NOT OK
class Program{
    //TODO
}
package p3; //Not OK
```



Un-named Package

- If we define any type without package then it is considered as member of unnamed/default package.
- Unnamed packages are provided by the Java SE platform principally for convenience when developing small or temporary applications or when just beginning development.
- An unnamed package cannot have sub packages.
- In following code, class Program is a part of unnamed package.

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



Naming Convention

- For small programs and casual development, a package can be unnamed or have a simple name, but if code is to be widely distributed, unique package names should be chosen using qualified names.
- Generally Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
- Companies use their reserved internet domain name to begin their package names. For example : com.example.mypackage
- Following examples will help you in deciding name of package:
 1. java.lang.reflect.Proxy
 2. oracle.jdbc.driver.OracleDriver
 3. com.mysql.jdbc.cj.Driver
 4. org.cdac.sunbeam.dac.utils.Date



How to use package members in different package?

- If we want to use types declared inside package anywhere outside the package then
 1. Either we should use fully qualified type name or
 2. import statement.
- If we are going to use any type infrequently then we should use fully qualified name.
- Let us see how to use type using package name.

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



How to use package members in different package?

- If we are going to use any type frequently then we should use import statement.
- Let us see how to import Scanner.

```
import java.util.Scanner;

class Program{

    public static void main(String[] args) {

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

}
```



How to use package members in different package?

- There can be any number of import statements after package declaration statement
- With the help of(*) we can import entire package.

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



How to use package members in different package?

- Another, less common form of import allows us to import the public nested classes of an enclosing class. Consider following code.

```
import java.lang.Thread.State;

class Program{

    public static void main(String[] args) {

        Thread thread = Thread.currentThread( );
        State state = thread.getState( );
    }
}
```

- Note : java.lang package contains fundamental types of core java. This package is by default imported in every .java file hence to use type declared in java.lang package, import statement is optional.





Thank you.

Rohan . paramane@sunbeaminfo . com





Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane



Agenda

- Array
- Enum



Array Introduction

- Array, stack, queue, LinkedList are data structures.
- In Java, data structure is called collection and value stored inside collection is called element.
- Array is a sequential/linear container/collection which is used to store elements of same type in continuous memory location.

In C/C++

Static Memory allocation for array

```
int arr1[ 3 ];    //OK  
  
int size = 3;  
int arr2[ size ];    //OK
```

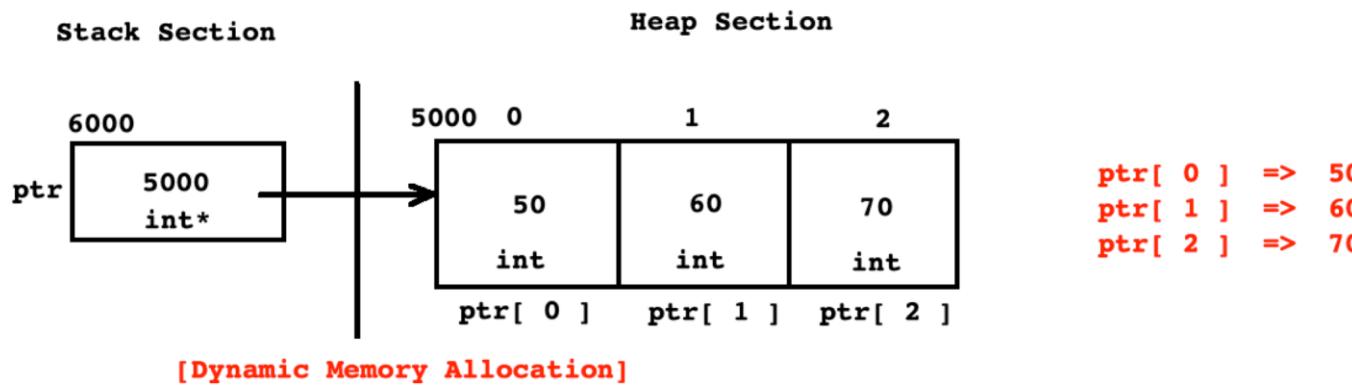
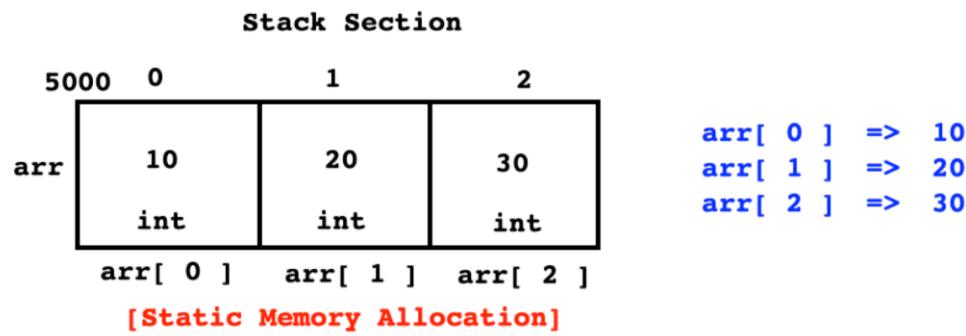
In C/C++

Dynamic Memory allocation for array

```
int *arr = ( int* )malloc( 3 * sizeof( int ));  
//or  
int *arr = ( int* )calloc( 3, sizeof( int ));
```



Static v/s Dynamic Memory Allocation In C/C++



Array Declaration and Initialization In C

- `int arr[3];` //OK : Declaration
- `int arr[3] = { 10, 20, 30 };` //OK : Initialization
- `int arr[] = { 10, 20, 30 };` //OK
- `int arr[3] = { 10, 20 };` //OK : Partial Initialization
- `int arr[3] = { };` //OK : Partial Initialization
- `int arr[3] = { 10, 20, 30, 40, 50 };` //Not recommended



Accessing Elements Of Array

- If we want to access elements of array then we should use integer index.
- Array index always begins with 0.

```
int arr[ 3 ] = { 10, 20, 30 };
printf("%d\n", arr[ 0 ] );
printf("%d\n", arr[ 1 ] );
printf("%d\n", arr[ 2 ] );
```

```
int arr[ 3 ] = { 10, 20, 30 };
int index;
for( index = 0; index < 3; ++ index )
    printf("%d\n", arr[ index ] );
```



Advantage and Disadvantages Of Array

- **Advantage Of Array**

1. We can access elements of array randomly.

Disadvantage Of Array

- 1. We can not resize array at runtime.
- 2. It requires continuous memory.
- 3. Insertion and removal of element from array is a time consuming job.
- 4. Using assignment operator, we can not copy array into another array
- 5. Compiler do not check array bounds(min and max index)

Array In Java

- Array is a reference type in Java. In other words, to create instance of array, new operator is required. It means that array instance get space on heap.
- **There are 3 types of array in Java:**
 1. Single dimensional array
 2. Multi dimensional array
 3. Ragged array
- **Types of loop in Java:**
 1. do-while loop
 2. while loop
 3. for loop
 4. for-each loop
- **To perform operations on array we can use following classes:**
 1. `java.util.Arrays`
 2. `org.apache.commons.lang3.ArrayUtils` (download .jar file)



Methods Of java.util.Arrays Class

Following are the methods of java.util Arrays class.(try javap java.util.Arrays)

- public static <T> List<T> asList(T... a)
- public static int binarySearch(int[] a, int key) //Overloaded
- public static int binarySearch(Object[] a, Object key)
- public static int[] copyOf(int[] original, int newLength)
- public static <T> T[] copyOf(T[] original, int newLength)
- public static int[] copyOfRange(int[] original, int from, int to)
- public static <T> T[] copyOfRange(T[] original, int from, int to)
- public static void fill(int[] a, int val)
- public static void fill(Object[] a, Object val)
- public static void fill(Object[] a, int fromIndex, int toIndex, Object val)
- public static void sort(int[] a) //Overloaded
- public static void sort(Object[] a)
- public static void parallelSort(int[] a)
- public static <T extends Comparable<? super T>> void parallelSort(T[] a)
- public static String toString(Object[] a) //Overloaded
- public static String deepToString(Object[] a)
- public static IntStream stream(int[] array) //Overloaded
- public static <T> Stream<T> stream(T[] array)



Single Dimensional Array

Reference declaration

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

Instantiation

```
int[ ] arr1 = new int[ 3 ];  
//or  
int size = 3;  
int[ ] arr2 = new int[ size ];
```

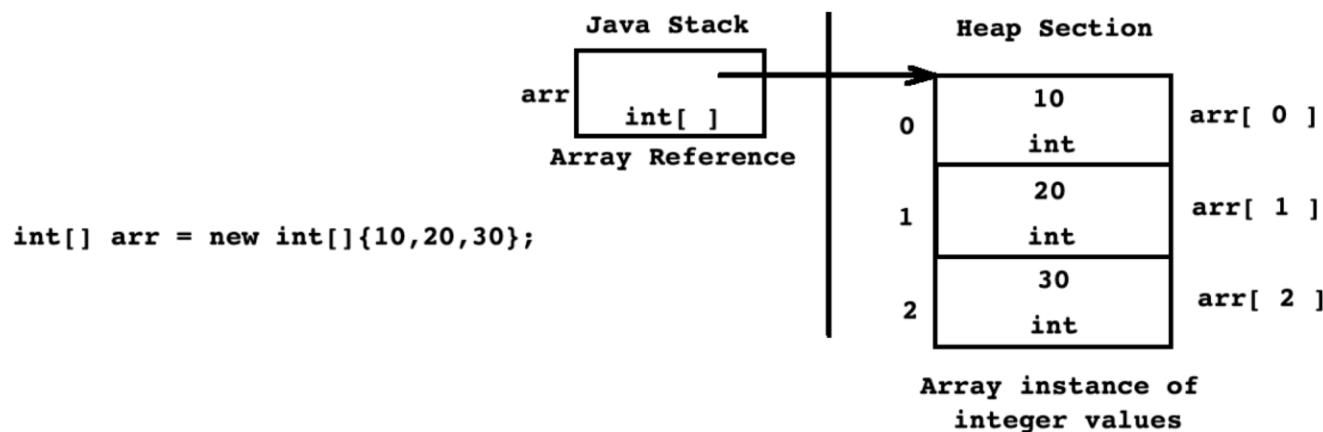
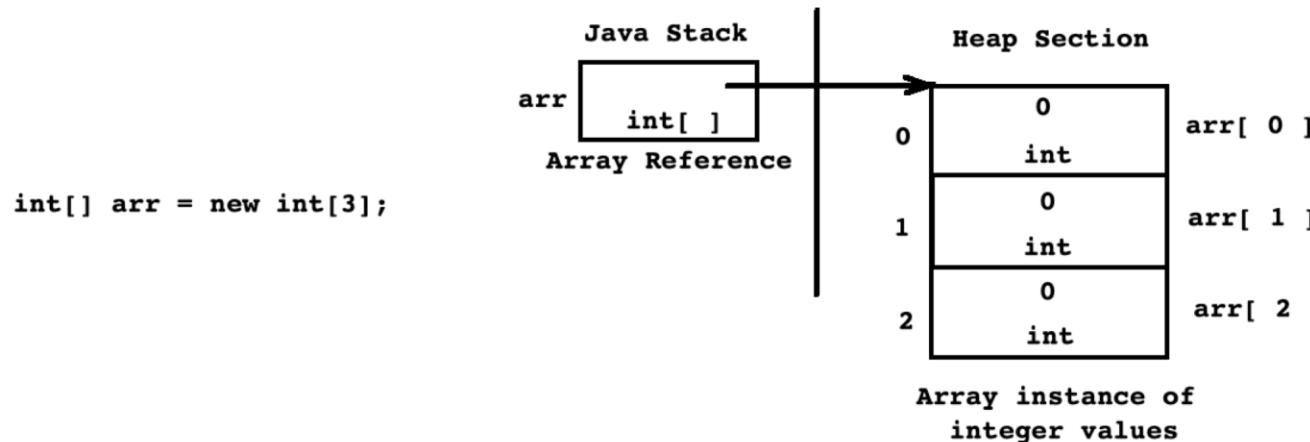
```
int[] arr1 = new int[ -3 ]; //NegativeArraySizeException  
//or  
int size = -3;  
int[] arr2 = new int[ size ]; //NegativeArraySizeException
```

Initialization

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



Single Dimensional Array



Using length Field

```
public class Program {  
    public 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 };  
        Program.printRecord(arr1);  
  
        int[] arr2 = new int[ ] { 10, 20, 30, 40, 50 };  
        Program.printRecord(arr2);  
  
        int[] arr3 = new int[ ] { 10, 20, 30, 40, 50, 60, 70 };  
        Program.printRecord(arr3);  
    }  
}
```



Using `toString()` Method

```
public static void main(String[] args) {  
    int[] arr = new int[ ] { 10, 20, 30, 40, 50 };  
    System.out.println(arr.toString()); // [I@6d06d69c  
}  
  
public static void main(String[] args) {  
    double[] arr = new double[ ] { 10.1, 20.2, 30.3, 40.4, 50.5 };  
    System.out.println(arr.toString()); // [D@6d06d69c  
}  
  
//Check the documentation of getName() method of java.lang.Class.  
public static void main(String[] args) {  
    int[] arr = new int[ ] { 10, 20, 30, 40, 50 };  
    System.out.println(Arrays.toString(arr)); // [10, 20, 30, 40, 50]  
}
```



ArrayIndexOutOfBoundsException

- Using illegal index, if we try to access elements of array then JVM throws ArrayIndexOutOfBoundsException. Consider following code:

```
public static void main(String[] args) {  
    int[] arr = new int[ ] { 10, 20, 30, 40, 50 };  
    //int element = arr[ -1 ]; //ArrayIndexOutOfBoundsException  
    //int element = arr[ arr.length ]; //ArrayIndexOutOfBoundsException  
    //int element = arr[ 7 ]; //ArrayIndexOutOfBoundsException  
}
```



ArrayStoreException

- If we try to store incorrect type of object into array then JVM throws ArrayStoreException.
- Consider the following code:

```
public class Program {  
    public static void main(String[] args) {  
        Object[] arr = new String[ 3 ];  
        arr[ 0 ] = new String("DAC"); //OK  
        arr[ 1 ] = "DMC"; //OK  
        arr[ 2 ] = new Integer(123); //Not OK : ArrayStoreException  
    }  
}
```



Sorting Array Elements

```
public class Program {  
    public static void main(String[] args) {  
        int[] arr = new int[] { 50, 10, 40, 20, 30 };  
        System.out.println((Arrays.toString(arr)));  
        Arrays.sort(arr); //The sorting algorithm is a Dual-Pivot Quicksort  
        System.out.println((Arrays.toString(arr)));  
    }  
}
```



Reference Copy and Instance Copy

Array Reference copy

```
int[] arr1 = new int[ ] { 10, 20, 30, 40, 50 };
int[] arr2 = arr1; //Reference Copy
```

Array Instance Copy(Using Arrays.copyOf())

```
int[] arr1 = new int[ ] { 10, 20, 30, 40, 50 };
int[] arr2 = Arrays.copyOf(arr1, arr1.length); //Array instance copy
```

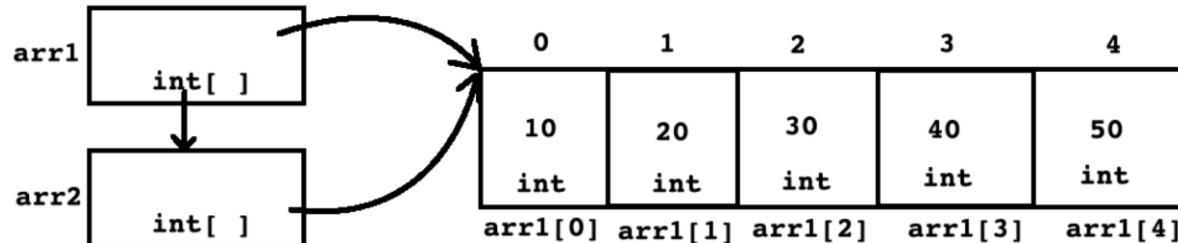
Implementation of Arrays.copyOf method:

```
public static int[] copyOf(int[] original, int newLength) {
    int[] copy = new int[newLength];
    //public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length);
    System.arraycopy(original, 0, copy, 0, Math.min(original.length, newLength));
    return copy;
}
```

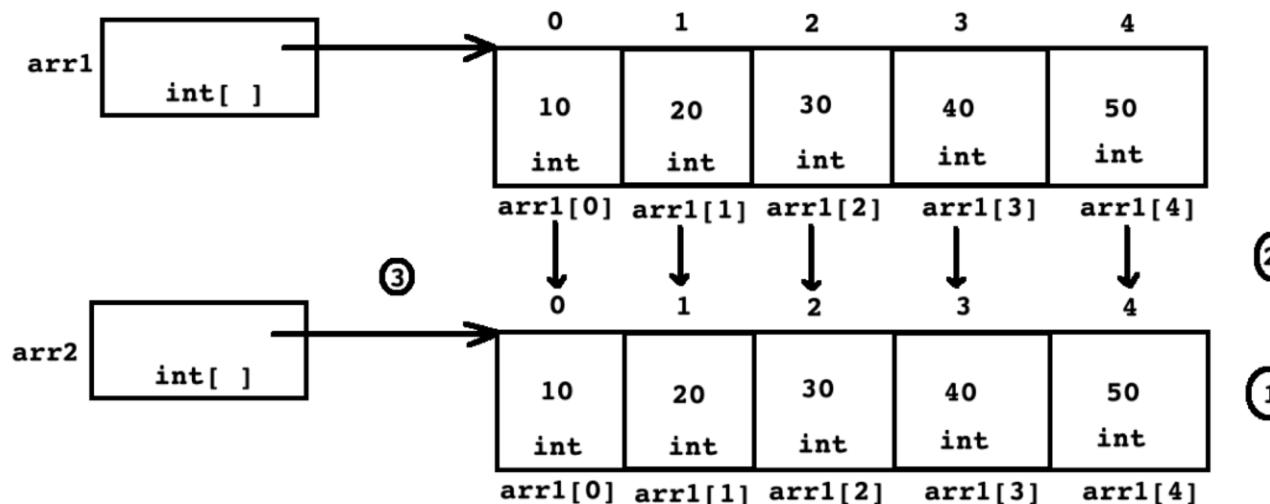


Reference Copy and Instance Copy

```
int[] arr1 = new int[ ] { 10, 20, 30, 40, 50 };  
int[] arr2 = arr1; //Reference Copy
```



```
int[] arr1 = new int[ ] { 10, 20, 30, 40, 50 };  
int[] arr2 = Arrays.copyOf(arr1, arr1.length); //Array instance copy
```



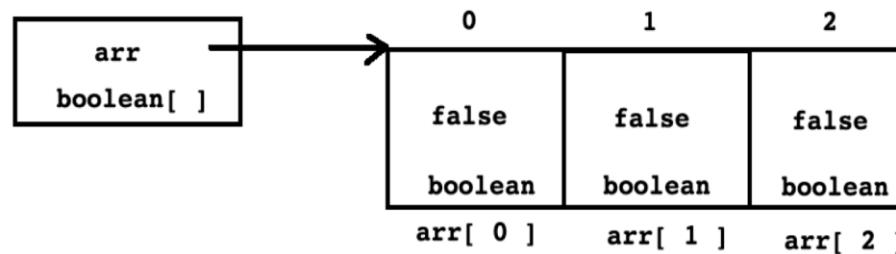
Array Of Primitive Values

```
public class Program {  
    public static void main(String[] args) {  
        boolean[] arr = new boolean[ 3 ]; //contains all false  
        int[] arr = new int[ 3 ]; //contains all 0  
        double[] arr = new double[ 3 ]; //contains all 0.0  
    }  
}
```

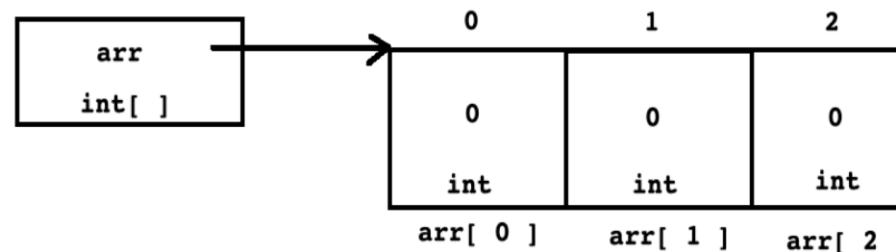


Array Of Primitive Values

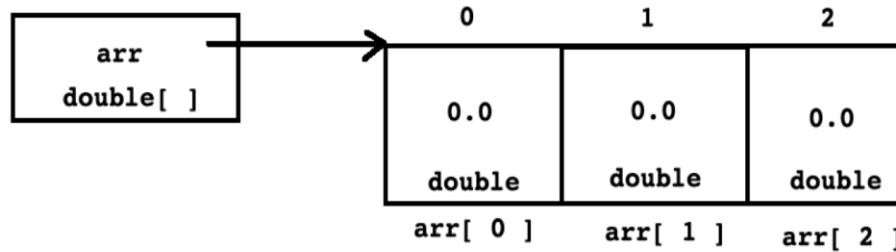
```
boolean[] arr = new boolean[3];
```



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



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



If we create array of primitive values then it's default value depends of default value of data type.

Array Of References

```
public class Program {  
    public static void main(String[] args) {  
        Date[] arr = new Date[ 3 ]; //Contains all null  
    }  
}
```



Array Of References and Instances

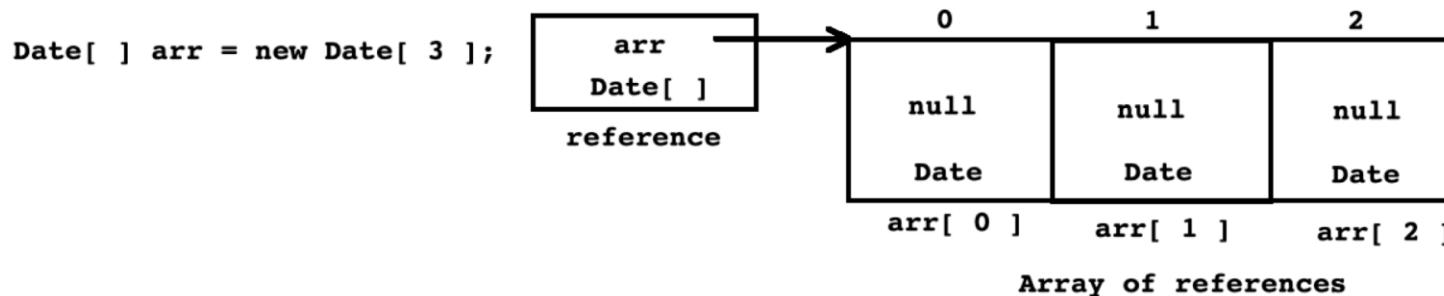
```
public class Program {  
    public static void main(String[] args) {  
        Date[] arr = new Date[ 3 ]; //Contains all null  
    }  
}
```

- Let us see how to create array of instances of non primitive type

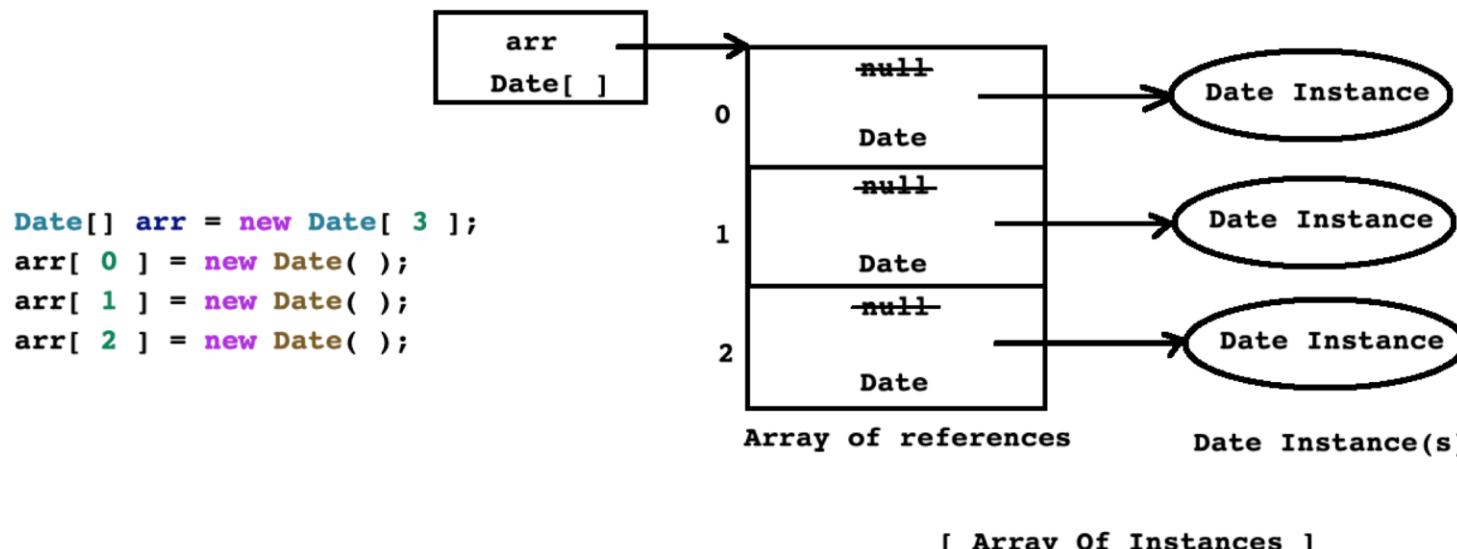
```
public class Program {  
    public static void main(String[] args) {  
        Date[] arr = new Date[ 3 ];  
        arr[ 0 ] = new Date( );  
        arr[ 1 ] = new Date( );  
        arr[ 2 ] = new Date( );  
    }  
    //or  
    public static void main(String[] args) {  
        Date[] arr = new Date[ 3 ];  
        for( int index = 0; index < arr.length; ++ index )  
            arr[ index ] = new Date( );  
    }  
}
```



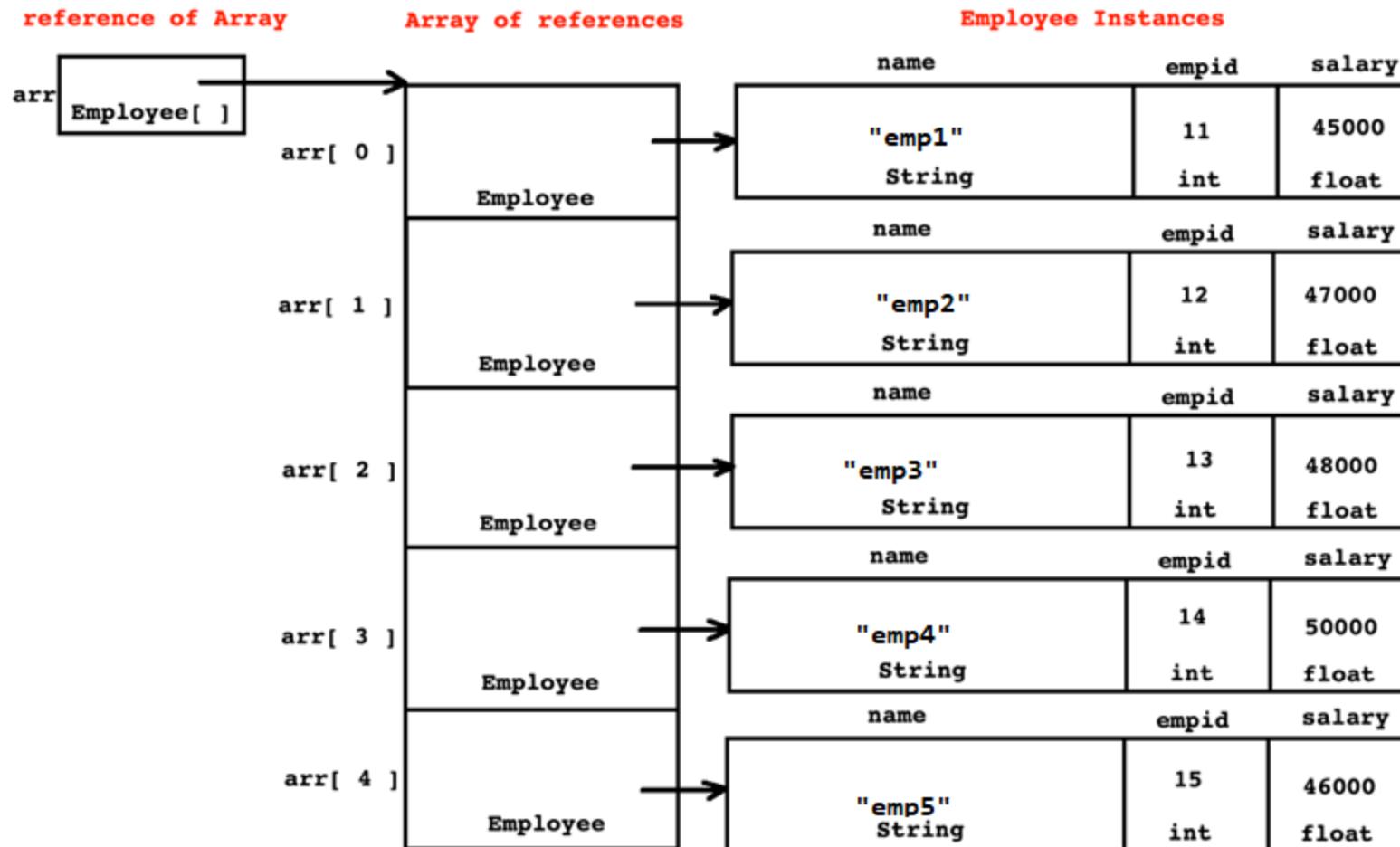
Array Of References and Instances



If we create an array of references then by default it contains null.



Array Of Instances



Multi Dimensional Array

- Array of elements where each element is array of same column size is called as multi dimensional array.

Reference declaration:

```
int arr[ ][ ]; //OK  
int [ ]arr[ ] //OK  
int[ ][ ] arr; //OK
```

Array Creation:

```
int[][] arr = new int[ 2 ][ 3 ];
```

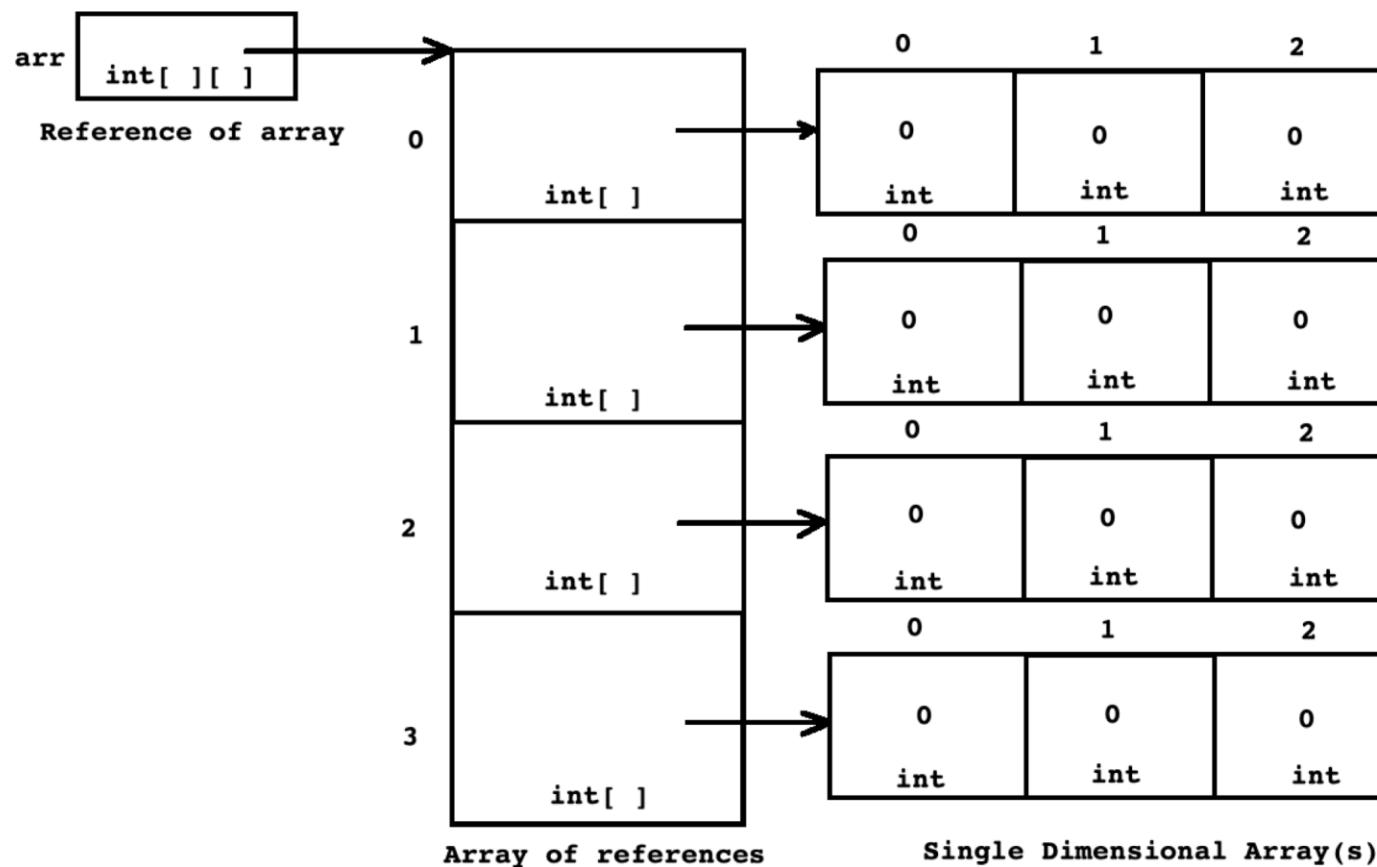
Initialization

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



Multi Dimensional Array

+ Multi Dimensional Array



Ragged Array

- A multidimensional array where column size of every array is different.

Reference declaration

```
int arr[][];  
int []arr[];  
int[][] arr;
```

Array creation

```
int[][] arr = new int[3][];  
arr[ 0 ] = new int[ 2 ];  
arr[ 1 ] = new int[ 3 ];  
arr[ 2 ] = new int[ 5 ];
```

Array Initialization

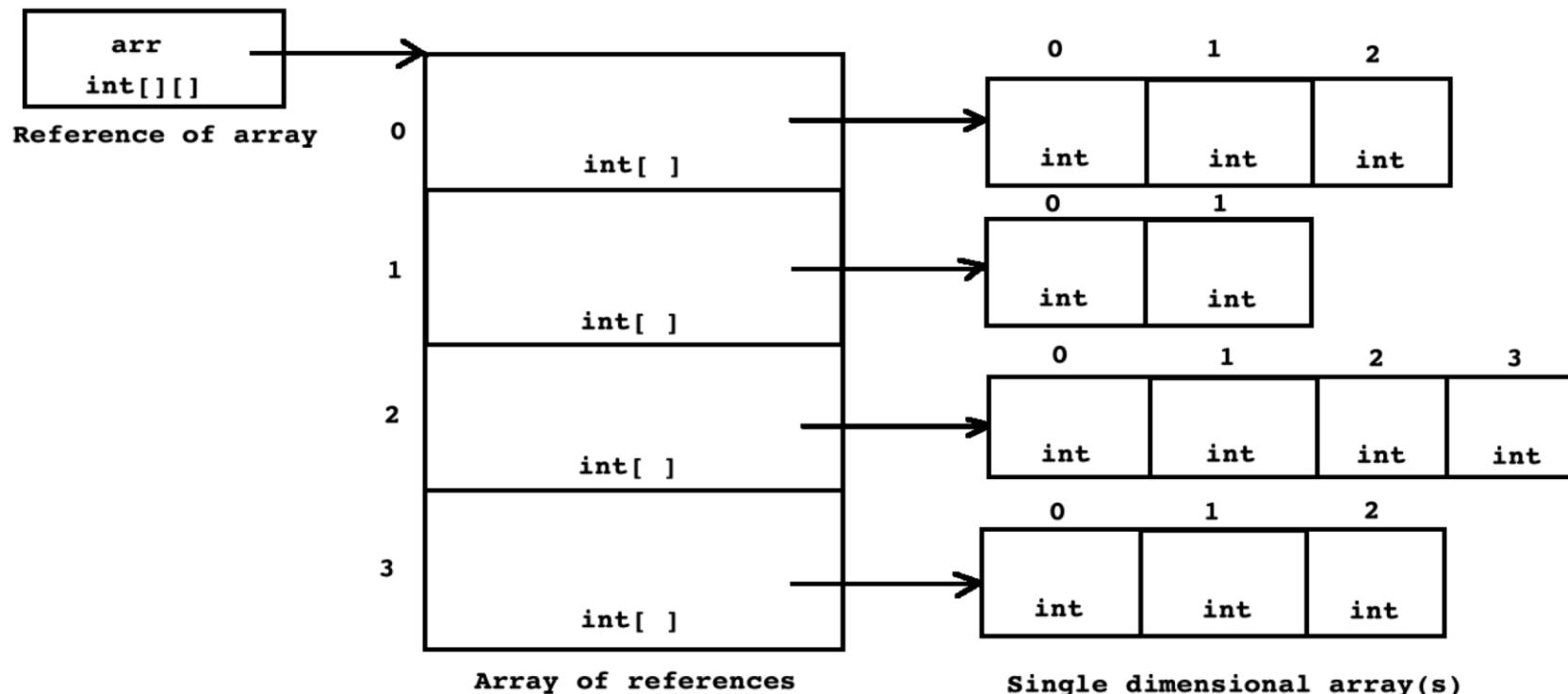
```
int[][] arr = new int[3][];  
arr[ 0 ] = new int[ ]{ 10, 20 };  
arr[ 1 ] = new int[ ]{ 10, 20, 30 };  
arr[ 2 ] = new int[ ]{ 10, 20, 30, 40, 50 };
```

```
int[][] arr = { { 1, 2 }, { 1, 2, 3 }, { 1, 2, 3, 4, 5 } };
```



Ragged Array

+ Ragged Array



Enum In C/C++ Programming language.

- According ANSI C standard, if we want to assign name to the integer constant then we should use enum.
- Enum helps developer to improve readability of source code.
- enum is keyword in C. Let us consider syntax of enum:

enum Identifier	enum Color
{	{
//enumerator-list	RED, GREEN, BLUE
};	//RED = 0, GREEN = 1, BLUE = 2
	};

```
enum Identifier  
{  
    //enumerator-list  
};
```

```
enum Color  
{  
    RED, GREEN, BLUE  
    //RED = 0, GREEN = 1, BLUE = 2  
};
```



Enum In C/C++ Programming language.

- By default, the first enumeration-constant is associated with the value 0. The next enumeration-constant in the list is associated with the value of (constant-expression + 1), unless you explicitly associate it with another value.

```
enum Channel
{
    FOX = 11,
    CNN = 25,
    ESPN = 15,
    HBO = 22,
    MAX = 30,
    NBC = 32
};

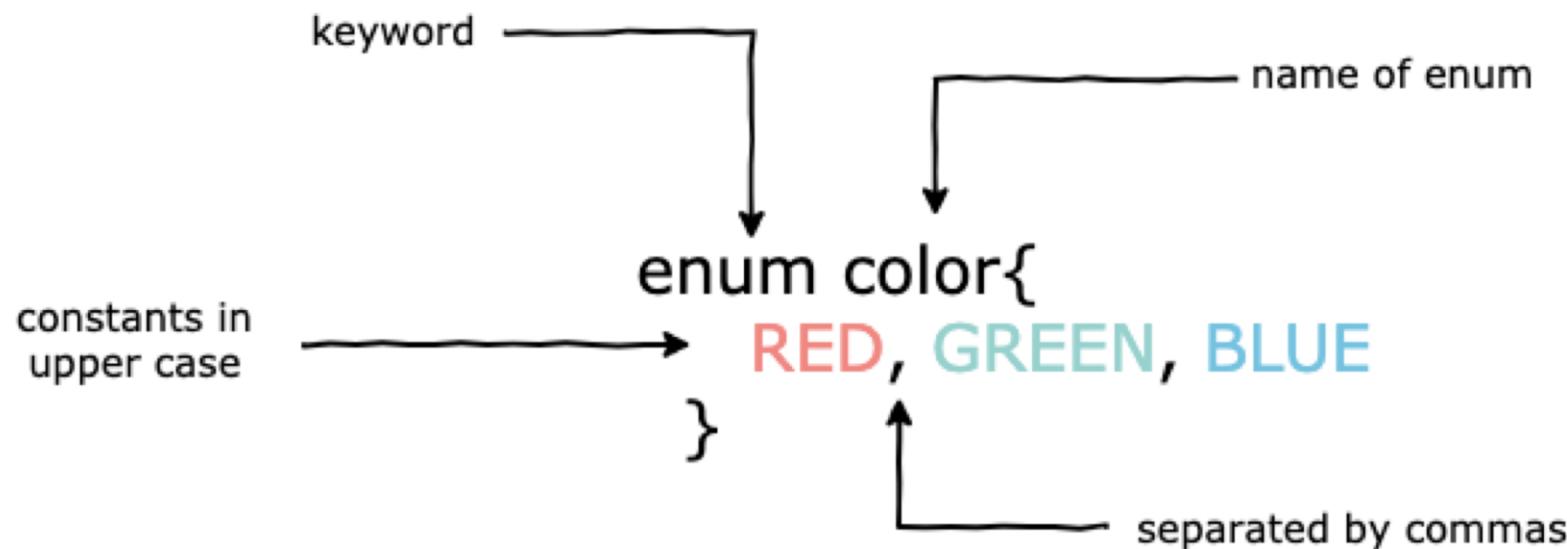
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

- constant-expression must have int type and can be negative.

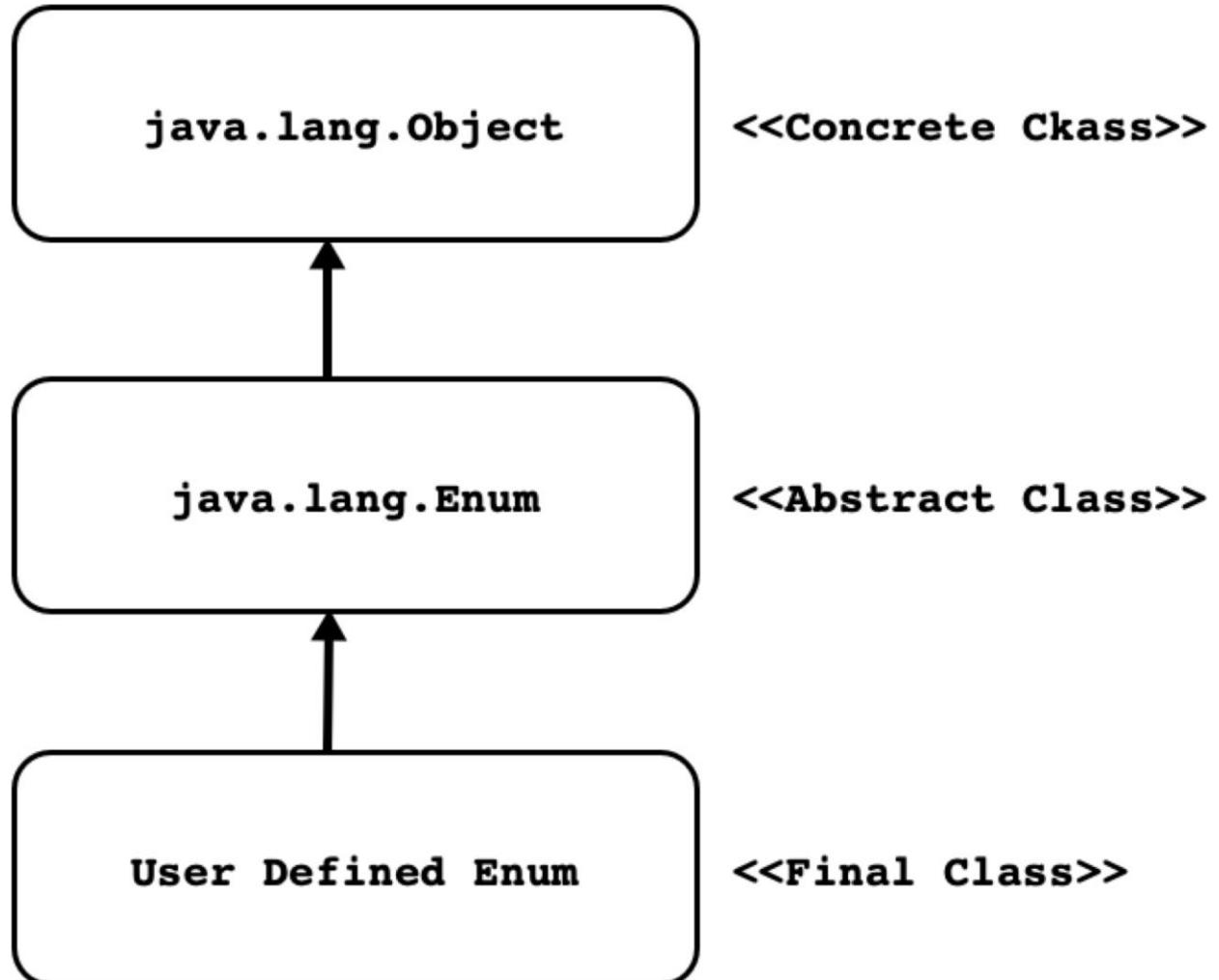


Enum In Java Programming language.

- An enum is a class that represents a group of constants.
- **Enum keyword** is used to create an enum. The constants declared inside are separated by a comma and should be in upper case.



Enum Class Hierarchy



Enum API

- Following are the methods declared in `java.lang.Enum` class:

String

[name\(\)](#)Returns the name of this enum constant, exactly as declared in its enum declaration.

int

[ordinal\(\)](#)Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).

String

[toString\(\)](#)Returns the name of this enum constant, as contained in the declaration.

static <T extends [Enum](#)<T>>
T

[valueOf\(Class<T> enumType, String name\)](#)Returns the enum constant of the specified enum type with the specified name.

Sole constructor : Programmers cannot invoke this constructor. It is for use by code emitted by the compiler in response to enum type declarations.



Enum for the compiler

Java Source Code

```
enum Color{
    RED, GREEN, BLUE
}
class Program{
    public static void main(String[] args) {
        Color color = Color.GREEN;
    }
}
```

Compiled Code

```
final class Color extends Enum<Color> {
    public static final Color RED;
    public static final Color GREEN;
    public static final Color BLUE;
    public static Color[] values();
    public static Color valueOf(String name);
}
```



Properties of enum

1. Similar to a class, an enum can have objects and methods. The only difference is that enum constants are public, static and final by default. Since it is final, we can't extend enums
2. It cannot extend other classes since it already extends the `java.lang.Enum` class.
3. It can implement interfaces.
4. The enum objects cannot be created explicitly and hence the enum constructor cannot be invoked directly.
5. It can only contain concrete methods and no abstract methods.



Application of enum

1. enum is used for values that are not going to change e.g. names of days, colors in a rainbow, number of cards in a deck etc.
2. enum is commonly used in switch statements and below is an example of it:

```
class Program {  
    enum color {  
        RED, GREEN, BLUE  
    }  
    public static void main(String[] args) {  
        color x = color.GREEN; // storing value  
        switch(x) {  
            case RED:  
                System.out.println("x has RED color");  
                break;  
            case GREEN:  
                System.out.println("x has GREEN color");  
                break;  
            case BLUE:  
                System.out.println("x has BLUE color");  
                break;  
        }  
    }  
}
```





Thank you.

Rohan . paramane@sunbeaminfo . com





Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane



Agenda

- String
- Association
- Inheritance



java.lang.Character

- It is a final class declared in `java.lang` package.
- The `Character` class wraps a value of the primitive type `char` in an object.
- This class provides a large number of static methods for determining a character's category (lowercase letter, digit, etc.) and for converting characters from uppercase to lowercase and vice versa.
- The fields and methods of class `Character` are defined in terms of character information from the Unicode Standard.
- The `char` data type are based on the original Unicode specification, which defined characters as fixed-width 16-bit entities.



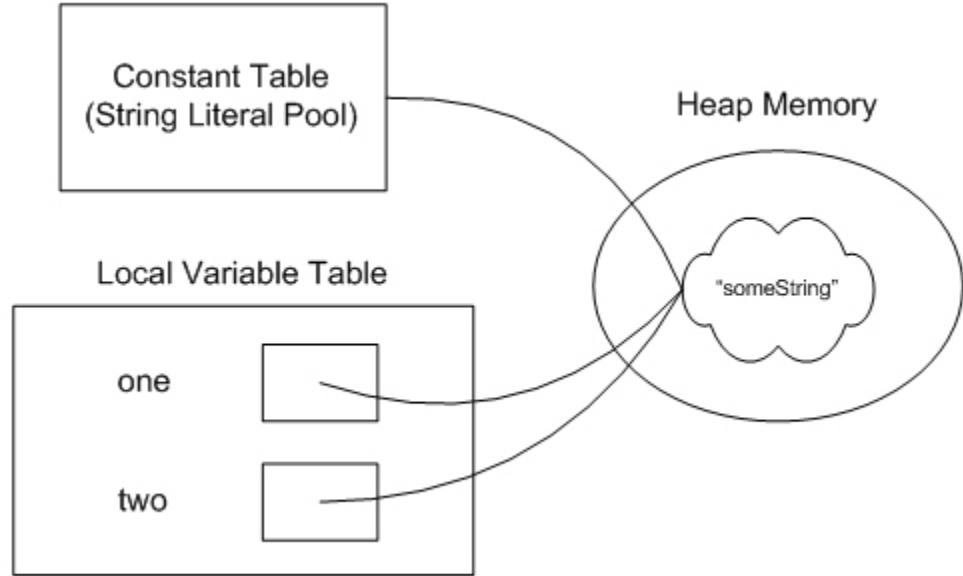
String Introduction

- Strings, which are widely used in Java programming, are a sequence of characters.
- In the Java programming language, strings are objects.
- We can use following classes to manipulate string
 - 1. **java.lang.String : immutable character sequence**
 - 2. **java.lang.StringBuffer : mutable**
 - 3. **java.lang.StringBuilder : mutable character sequence**
 - 4. **java.util.StringTokenizer**

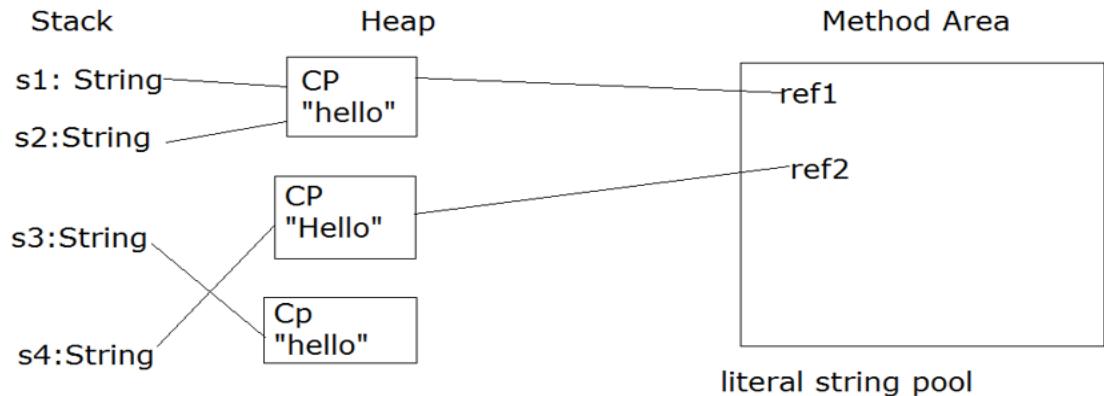


Literal Vs Non Literal

Two ways to create a String in Java which are String Literal and String Object. The main difference between String Literal and String Object is that String Literal is a String created using double quotes while String Object is a String created using the new() operator.



```
Eg. String s1="hello"; // Literal  
String s2="hello"; // Literal  
String s3=new String(s1); // Non Literal  
String s4="hello"; // Literal
```



Note: JVM Class loader will scan the literal string @class loading and create string objects on heap and its reference in literal/constant string pool (Memory allocated to method area).

Pool : Sharing of resources, so multiple references for the same content string will not be kept.

Example Literal Vs Non Literal

Literal

`String s1 = "Hello World";`

Here, the s1 is referring to “Hello World” in the String pool.

If there is another statement as follows.

`String s2 = "Hello World";`

As “Hello World” already exists in the String pool, the s2 reference variable will also point to the already existing “Hello World” in the String pool. In other words, now both s1 and s2 refer to the same “Hello World” in the String pool. Therefore, if the programmer writes a statement as follows, it will display true.

`System.out.println(s1==s2);`

if you create an object using String literal it may return an existing object from String pool (a cache of String object in which is now moved to heap space in recent Java release).

Non Literal

`String s1 = new ("Hello World");`

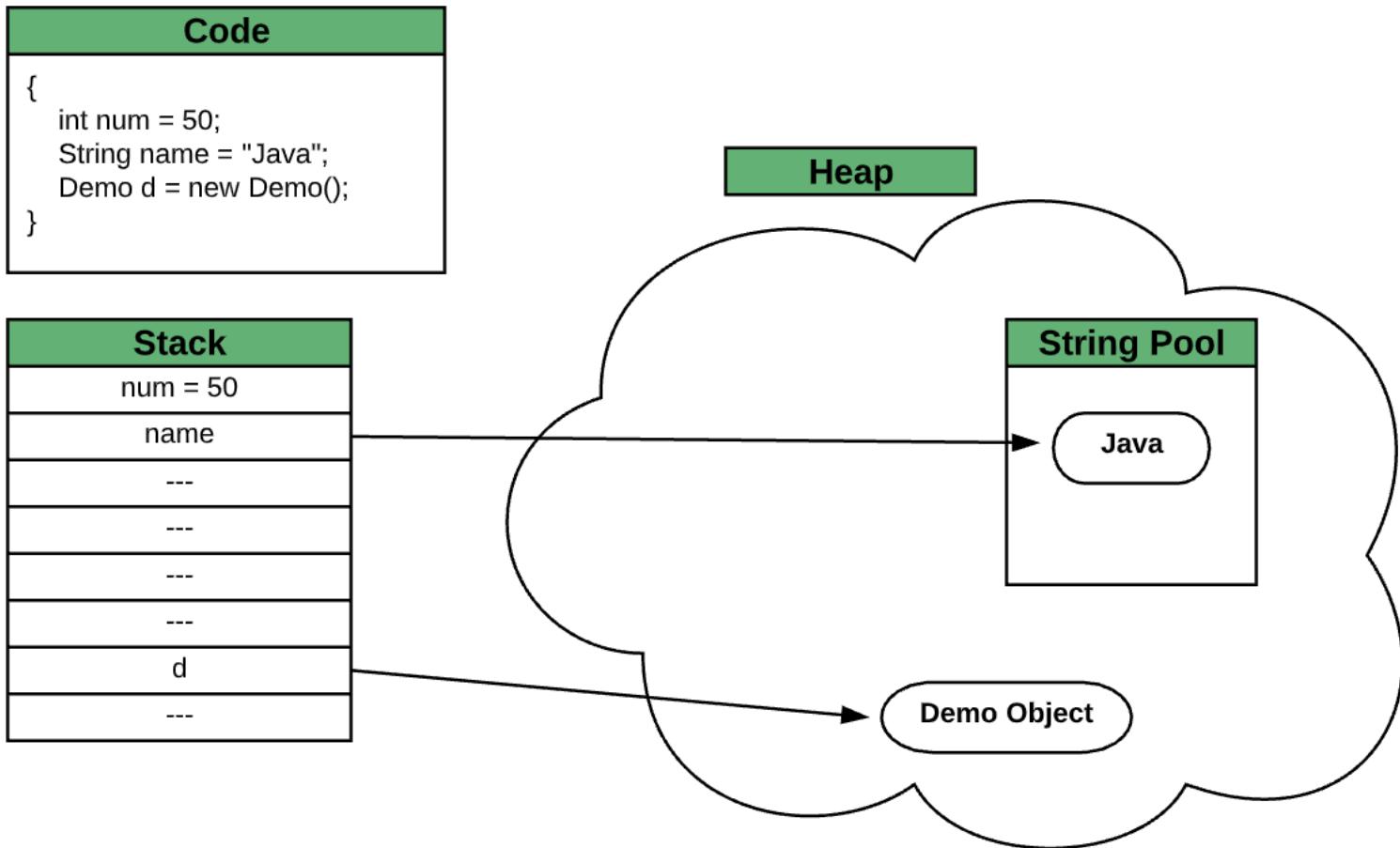
`String s2 = new ("Hello World");`

Unlike with String literals, in this case, there are two separate objects. In other words, s1 refers to one “Hello World” while s2 refers to another “Hello World”. Here, the s1 and s2 are reference variables that refer to separate String objects. Therefore, if the programmer writes a statement as follows, it will display false.

`System.out.println(s1==s2);`

When you create a String object using the new() operator, it always creates a new object in heap memory.



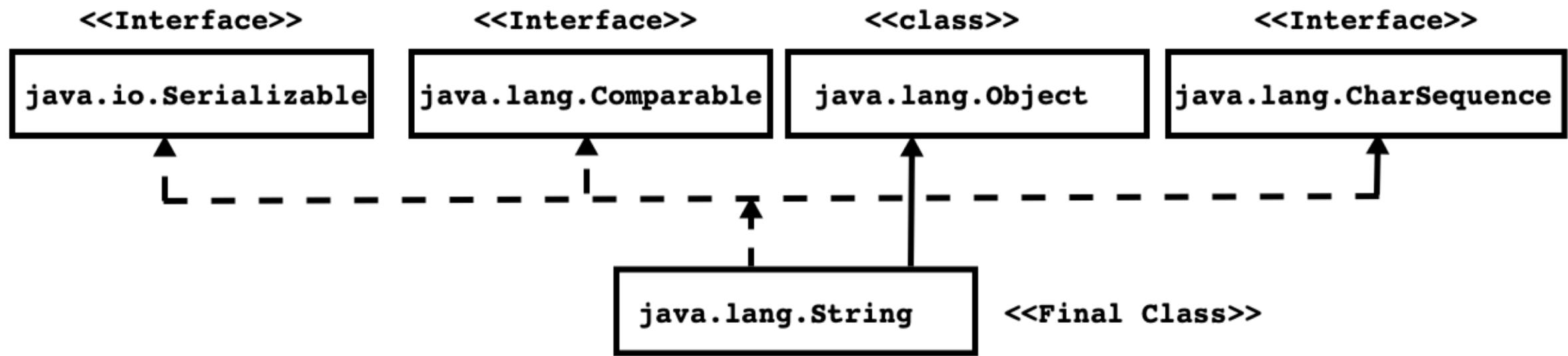


Literal Vs Non Literal example

```
String s1="hello";//s1 --> literal
String s2="hello";//nothing
String s3=new String(s1);//non literal
String s4=s3.intern();//won't add any new literal string : since "hello" already exists
String s5="he"+"llo";//won't add any new literal string : since "hello" already exists
String s6="he".concat("llo");//new non literal
System.out.println(s1==s2);//t
System.out.println(s1==s3);//f
System.out.println(s1==s4);//t
System.out.println(s1==s5);//t
System.out.println(s1==s6);//f
String s7=new String("Hello");//how many string objects are created in this line? : 2
String s8=new String("hello");//how many string objects are created in this line? : 1
```



String Class Hierarchy



String Introduction

- Serializable is a Marker interface declared in java.io package.
- Comparable is Functional interface declared in java.lang package.
 1. int compareTo(T other)
- CharSequence is interface declared in java.lang package.
 1. char charAt(int index)
 2. int length()
 3. CharSequence subSequence(int start, int end)
 4. default IntStream chars()
 5. default IntStream codePoints()
- Object is non final, concrete class declared in java.lang package.
 1. It is having 11 methods(5 Non final + 6 final)
- String is a final class declared in java.lang package.



String Introduction

- String is not a built-in or primitive type. It is a class, hence considered as non primitive/reference type.
- We can create instance of String with and W/o new operator.
 - `String str = new String("Rohan"); //String Instance`
 - `String str = "SunBeam";`
- `String str = "SunBeam"`, is equivalent to:
 - `char[] data = new char[]{ 'S', 'u', 'n', 'B', 'e', 'a', 'm' };`
 - `String str = new String(data);`



String concatenation

- If we want to concatenate Strings then we should use concat() method:

➤ "public String concat(String str)"

- Consider following Example:

```
String s1 = "SunBeam";
String s2 = "Pune/Karad";
String s3 = s1.concat( s2 );
```

- The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

```
String s1 = "SunBeam";
String s2 = s1 +" Pune/Karad";
```

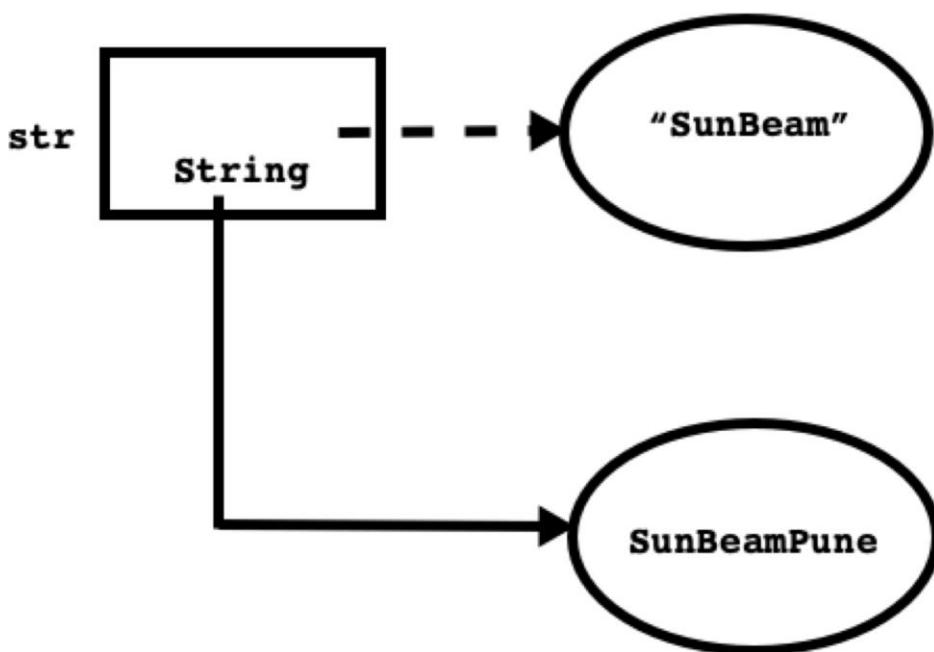
-
-
- int pinCode = 411057;
- String str = "Pune,"+pinCode;



Immutable Strings

- Strings are constant; their values cannot be changed after they are created.
- Because String objects are immutable they can be shared.

```
String str = "SunBeam";  
  
str = str + "Pune";
```



A Strategy for Defining Immutable Objects

1. Don't provide "setter" methods – methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
 - o Don't provide methods that modify the mutable objects.
 - o Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.



String Class Constructors

1. public String()
2. public String(byte[] bytes)
3. public String(char[] value)
4. public String(String original)
5. public String(StringBuffer buffer)
6. public String(StringBuilder builder)



String Class Methods

1. public char charAt(int index)
2. public int compareTo(String anotherString)
3. public String concat(String str)
4. public boolean equalsIgnoreCase(String anotherString)
5. public boolean startsWith(String prefix)
6. public boolean endsWith(String suffix)
7. public static String format(String format, Object... args)
8. public byte[] getBytes()
9. public int indexOf(int ch)
10. public int indexOf(String str)
11. public String intern()
12. public boolean isEmpty()
13. public int length()
14. public boolean matches(String regex)



String Class Methods

```
15. public String[] split(String regex)  
16. public String substring(int beginIndex)  
17. public String substring(int beginIndex, int endIndex)  
18. public char[] toCharArray()  
19. public String toLowerCase()  
20. public String toUpperCase()  
21. public String trim()  
22. public static String valueOf(Object obj)
```

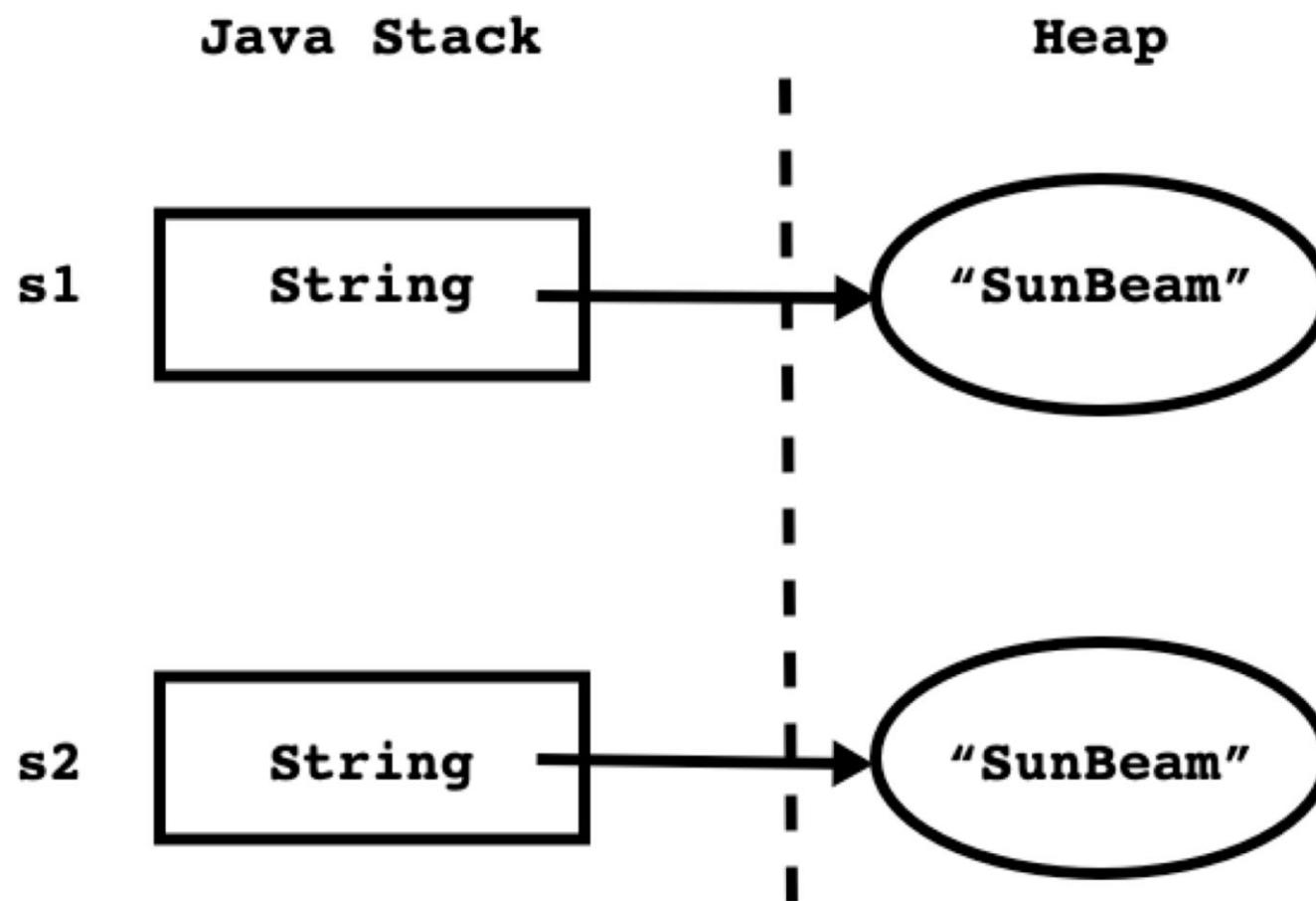


String Twister

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



String Twister



String Twister

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

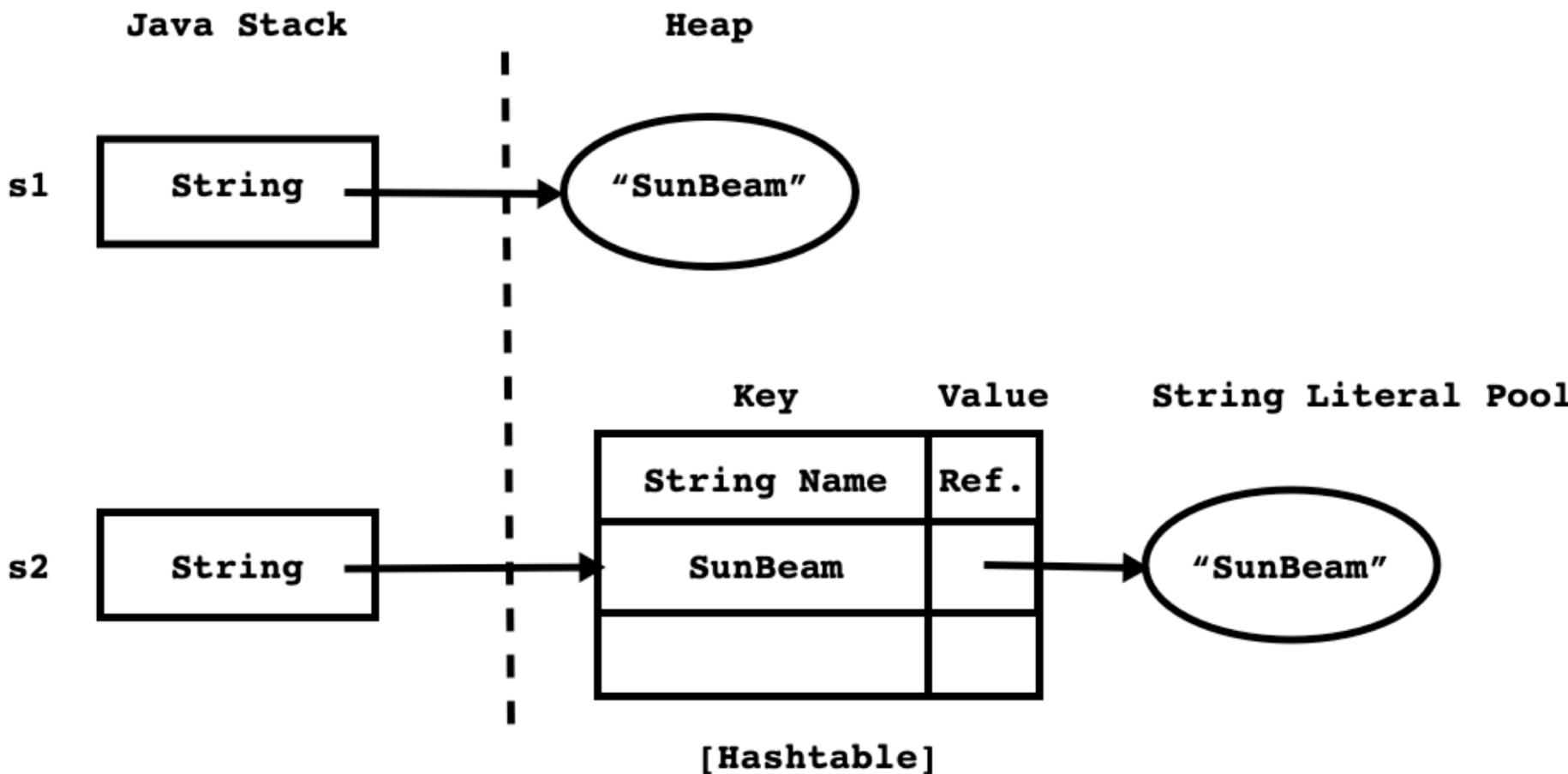


String Twister

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



String Twister



String Twister

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

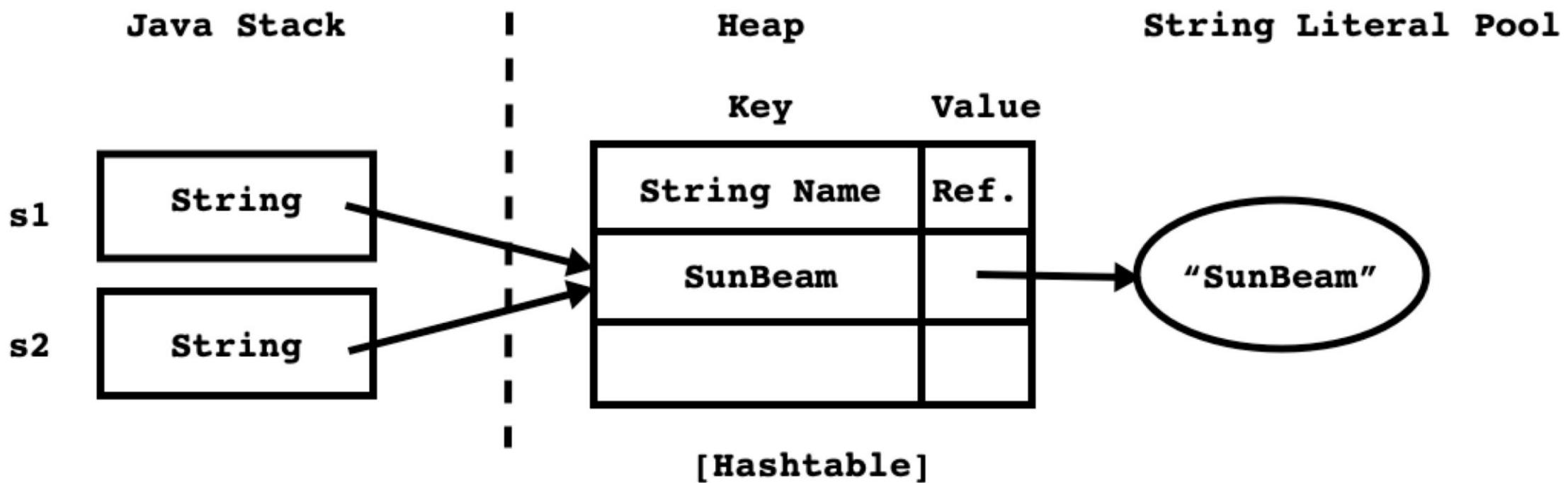


String Twister

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



String Twister



String Twister

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



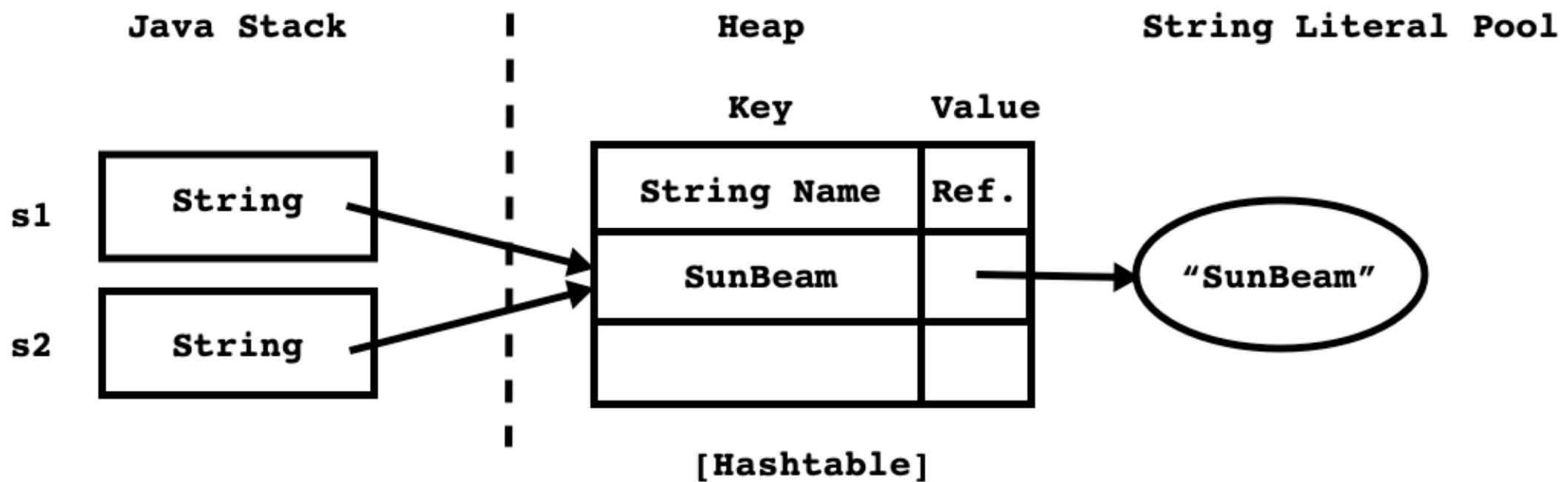
String Twister

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



String Twister

- Constant expression gets evaluated at compile time.
 - "int result = 2 + 3;" becomes "int result = 5;" at compile time
 - "String s2 = "Sun"+"Beam";" becomes "String s2="SunBeam";" at compile time.

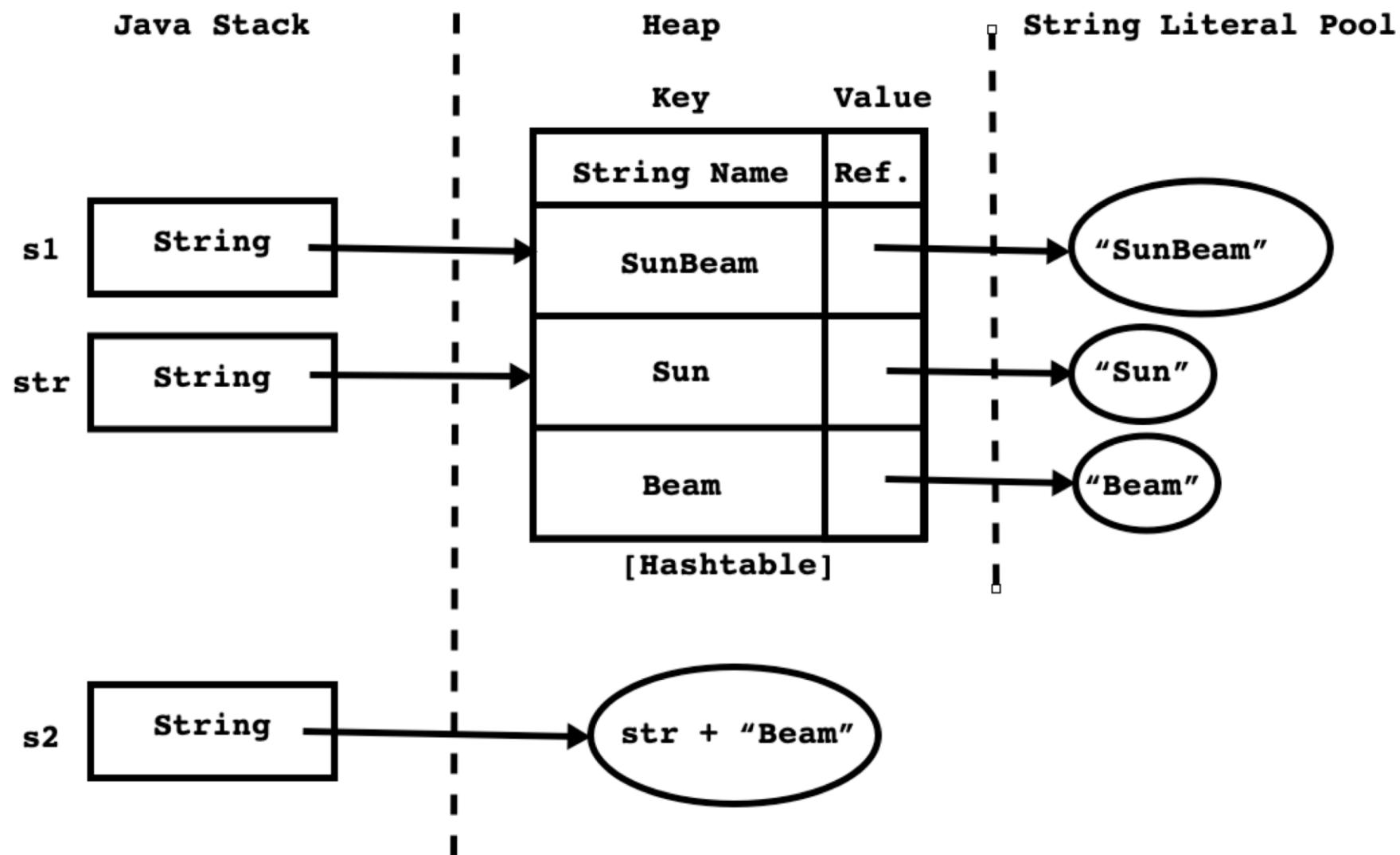


String Twister

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



String Twister



String Twister

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



String Twister

```
package p1;
public class A {
    public static final String str = "Hello";
}
```

```
package test;
class B {
    public static final String str = "Hello";
}
```

```
package test;
public class Program {
    public static final String str = "Hello";
    public static void main(String[] args) {
        String str = "Hello";
    }
}
```



String Twister

```
public class Program {  
    public static final String str = "Hello";  
    public static void main(String[] args) {  
        String str = "Hello";  
        System.out.println(A.str == B.str);          //true  
        System.out.println(A.str == Program.str);    //true  
        System.out.println(A.str == str);            //true  
        System.out.println(B.str == Program.str);    //true  
        System.out.println(B.str == str);            //true  
        System.out.println(Program.str == str);      //true  
    }  
}
```



String Twister

```
public class Program {  
    public static void main(String[] args) {  
        String str = "SunBeam";  
        //char ch = str.charAt( 0 ); //S  
        //char ch = str.charAt( 6 ); //m  
        //char ch = str.charAt(-1); //StringIndexOutOfBoundsException  
        char ch = str.charAt( str.length() ); //StringIndexOutOfBoundsException  
        System.out.println(ch);  
    }  
}
```



StringBuffer versus StringBuilder

- StringBuffer and StringBuilder are final classes.
- It is declared in `java.lang` package.
- It is used to create mutable string instance.
- `equals()` and `hashCode()` method is not overridden inside it.
- We can create instances of these classes using `new` operator only.
- Instances get space on Heap.
- **StringBuffer implementation is thread safe whereas StringBuilder is not.**
- **StringBuffer is introduced in JDK1.0 and StringBuilder is introduced in JDK 1.5.**



StringBuffer Twister

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



StringBuffer Twister

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



StringBuffer Twister

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



StringBuffer Twister

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



StringBuffer Twister

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



StringBuilder Twister

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

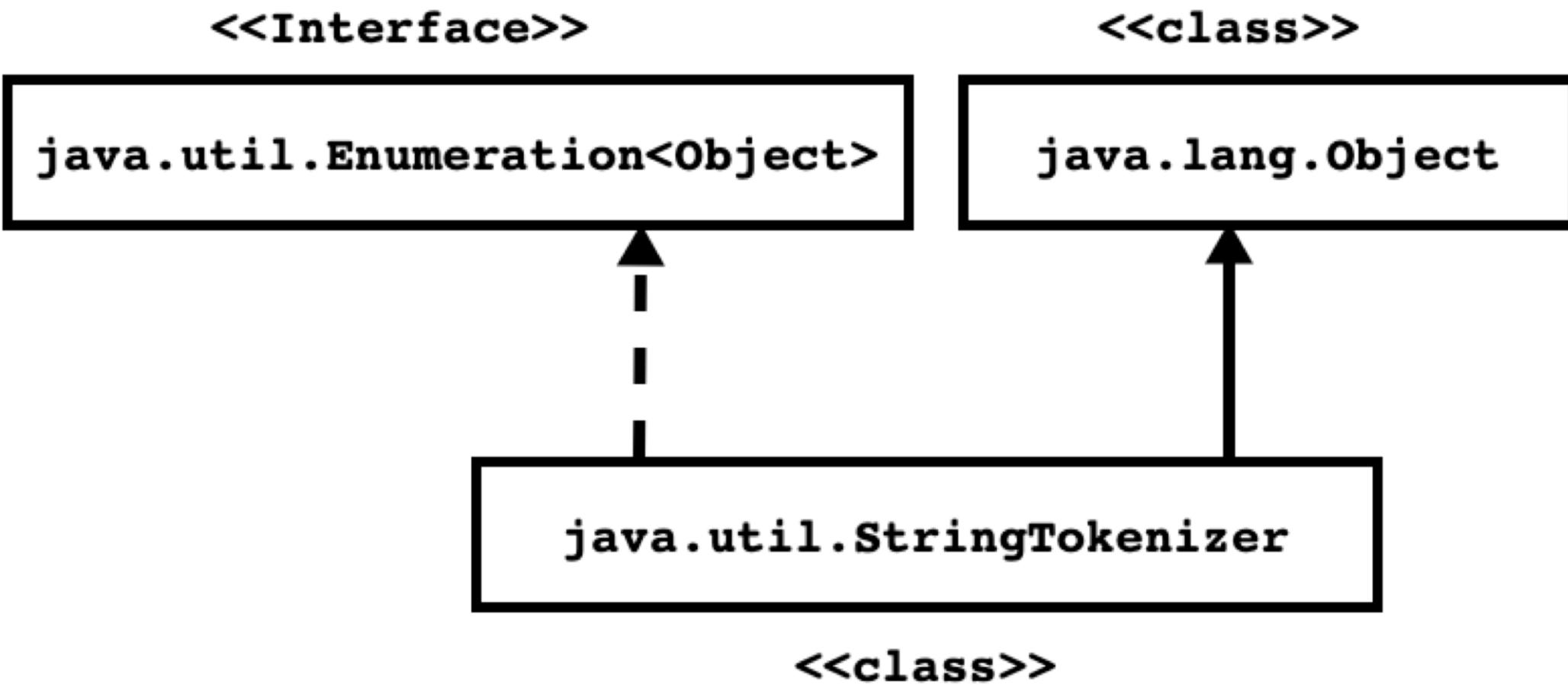


StringBuilder Twister

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



StringTokenizer



StringTokenizer

- The string tokenizer class allows an application to break a string into tokens.
- Methods of `java.util.Enumeration<E>` interface
 1. `boolean hasMoreElements()`
 2. `E nextElement()`
- Methods of `java.util.StringTokenizer` class
 1. `public int countTokens()`
 2. `public boolean hasMoreTokens()`
 3. `public String nextToken()`
 4. `public String nextToken(String delim)`



StringTokenizer

```
public class Program {  
    public static void main(String[ ] args) {  
        String str = "SunBeam Infotech Pune";  
        StringTokenizer stk = new StringTokenizer(str);  
        String token = null;  
        while( stk.hasMoreTokens() ) {  
            token = stk.nextToken();  
            System.out.println(token);  
        }  
    }  
}
```

Output

SunBeam
Infotech
Pune



StringTokenizer

```
public class Program {  
    public static void main(String[] args) {  
        String str = "www.sunbeaminfo.com";  
        String delim = ".";  
        StringTokenizer stk = new StringTokenizer(str, delim);  
        String token = null;  
        while( stk.hasMoreTokens() ) {  
            token = stk.nextToken();  
            System.out.println(token);  
        }  
    }  
}
```

Output

```
www  
sunbeaminfo  
com
```



StringTokenizer

```
import java.util.Scanner;
import java.util.StringTokenizer;

public class Day6_5
{
    static Scanner sc = new Scanner(System.in);
    public static void main(String[] args)
    {
        String str = "https://admission.sunbeaminfo.com/aspx/RegistrationForm.aspx?BatchID=J8BwSw7MbJHgHVtHZgIU1A==";

        String delim = "/:-.=/#";
        StringTokenizer stk = new StringTokenizer(str, delim, true);

        String token = null;
        while( stk.hasMoreTokens() ) {
            token = stk.nextToken();
            System.out.println(token);
        }
    }
}
```

OUTPUT

https
:
/
/
admission
.
sunbeaminfo
.
com
/
aspx
/
RegistrationForm
.
aspx?BatchID
=
J8BwSw7MbJHgHVtHZgIU1A
=
=



Advantages of OOPS

1. To achieve simplicity
2. To achieve data hiding and data security.
3. To minimize the module dependency so that failure in single part should not stop complete system.
4. To achieve reusability so that we can reduce development time/cost/efforts.
5. To reduce maintenance of the system.
6. To fully utilize hardware resources.
7. To maintain state of object on secondary storage so that failure in system should not impact on data.



Major and Minor pillars of oops

- 4 Major pillars
 - 1. Abstraction
 - 2. Encapsulation
 - 3. Modularity
 - 4. Hierarchy
- 3 Minor Pillars
 - 1. Typing
 - 2. Concurrency
 - 3. Persistence



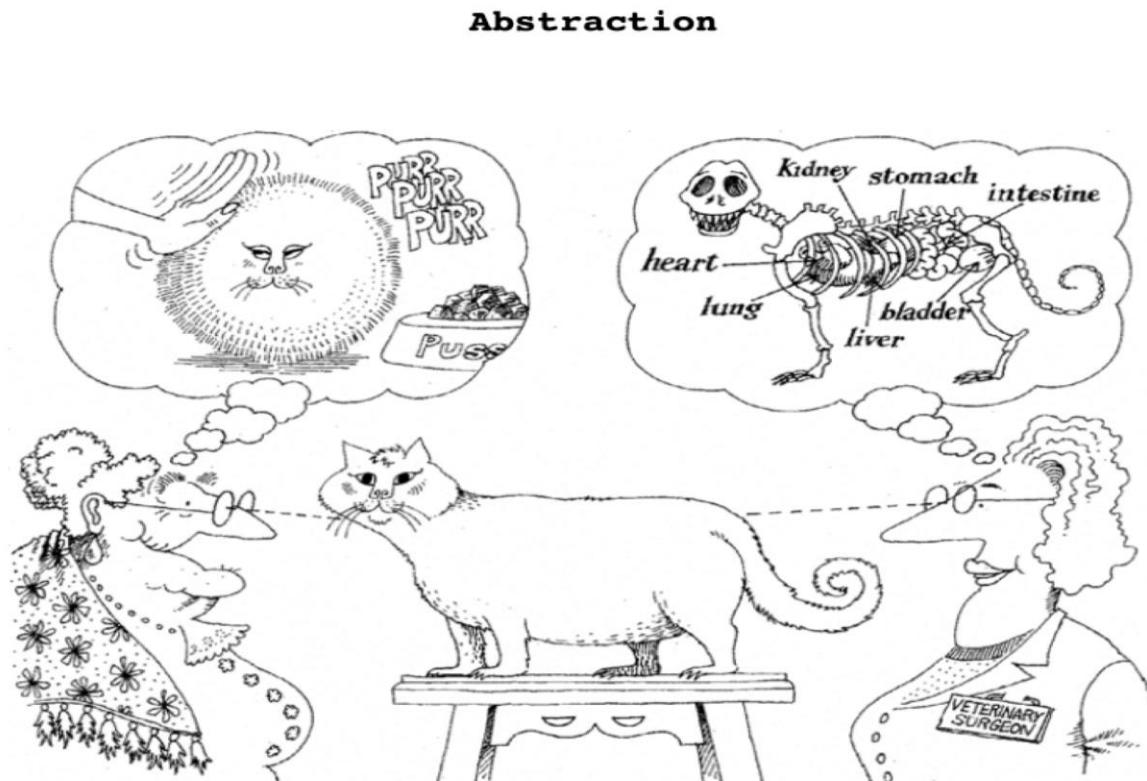
Abstraction

- It is a major pillar of oops.
- **It is a process of getting essential things from object.**
- It describes outer behaviour of the object.
- Abstraction focuses on some essential characteristics of object relative to the perspective of viewer. In other words, abstraction changes from user to user.
- **Using abstraction, we can achieve simplicity.**
- Abstraction in Java

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



Abstraction



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

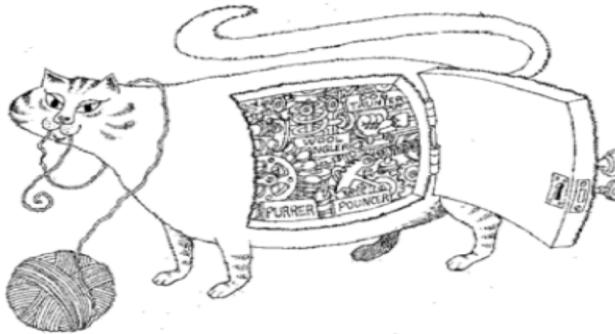
Encapsulation

- It is a major pillar of oops.
- Definition:
 1. **Binding of data and code together is called encapsulation.**
 2. To achieve abstraction, we should provide some implementation. It is called encapsulation.
- Encapsulation represents, internal behaviour of the object.
- **Using encapsulation we can achieve data hiding.**
- Abstraction and encapsulation are complementary concepts: **Abstraction focuses on the observable behaviour** of an object, whereas **encapsulation focuses on the implementation** that gives rise to this behaviour.



Encapsulation

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.

Modularity

- It is a major pillar of oops.
- It is the process of developing complex system using small parts.
- **Using modularity, we can reduce module dependency.**
- We can implement modularity by creating library files.
 - .lib/.a, .dll / .so files
 - .jar/.war/.ear in java



Hierarchy

- It is a major pillar of oops.
- **Level / order / ranking of abstraction is called hierarchy.**
- Main purpose of hierarchy is **to achieve reusability**.
- Advantages of code reusability
 1. We can reduce development time.
 2. We can reduce development cost.
 3. We can reduce developers effort.
- Types of hierarchy:
 1. **Has-a** / Part-of => Association
 2. **Is-a** / Kind-of => Inheritance / Generalization
 3. **Use-a** => Dependency
 4. **Creates-a** => Instantiation



Typing

- It is a minor pillar of oops.
- Typing is also called as polymorphism.
- Polymorphism is a Greek word. Polymorphism = Poly(many) + morphism(forms).
- **An ability of object to take multiple forms is called polymorphism.**
- **Using polymorphism, we can reduce maintenance of the system.**
- Types of polymorphism:
 - **Compile time polymorphism**
 - It is also calling static polymorphism / **Early binding** / Weak Typing / False polymorphism.
 - We can achieve it using:
 1. **Method Overloading**
 - **Run time polymorphism**
 - It is also calling dynamic polymorphism / **Late binding** / Strong Typing / True polymorphism.
 - We can achieve it using:
 1. **Method Overriding**.



Concurrency

- It is a minor pillar of oops.
- In context of operating system, it is called as multitasking.
- It is the process of executing multiple task simultaneously.
- Main purpose of concurrency is to utilise CPU efficiently.
- In Java, we can achieve concurrency using thread.



Persistence

- It is a minor pillar of oops.
- It is process of maintaining state of object on secondary storage.
- In Java, we can achieve Persistence using file and database.



Association

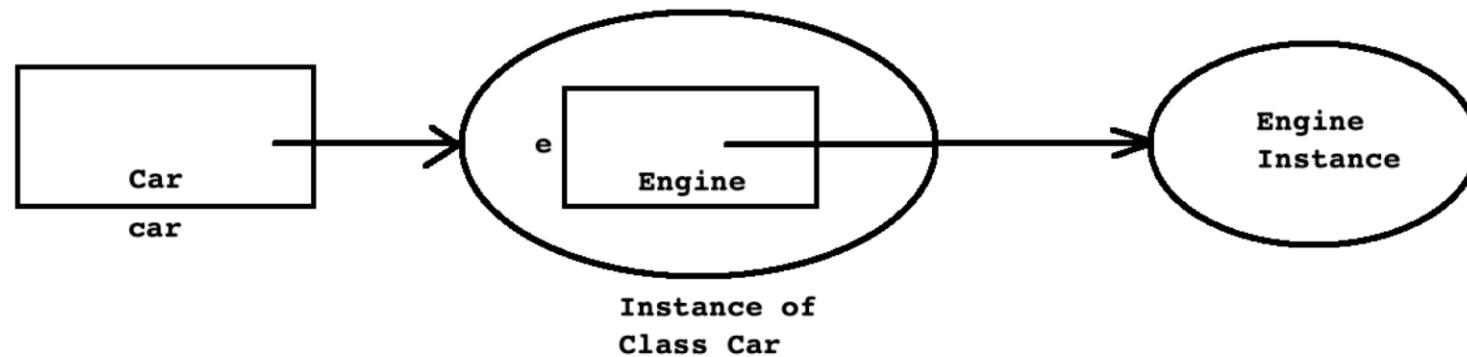
- If has-a relationship is exist between the types then we should use association.
- Example
 - 1. Car has a engine
 - 2. Room has a chair
- Let us consider example of car and engine:
 - 1. Car has a engine
 - 2. Engine is part of Car.
- If object-instance is a part/component of another instance then it is called as association.
- To implement association, we should declare instance of a class as a field inside another class.



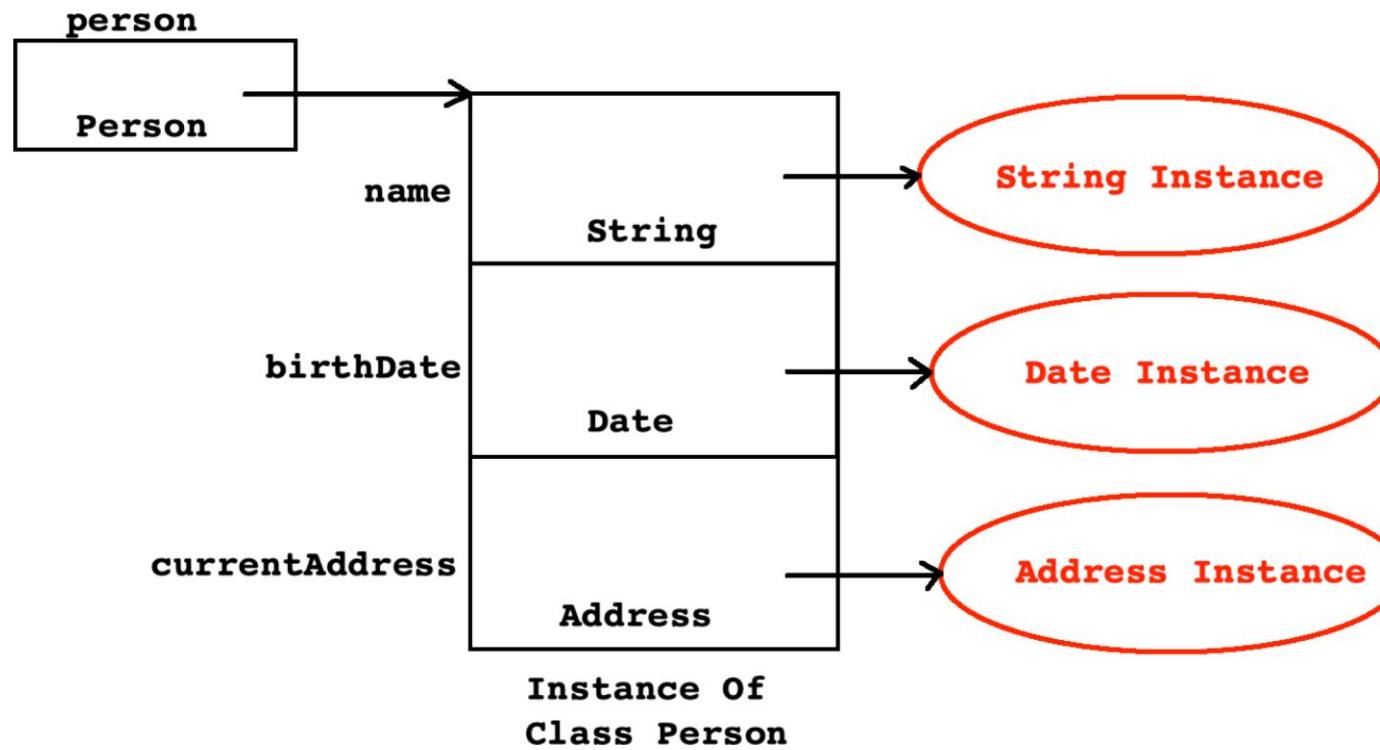
Association In Java

- Engine is part of Car

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



Association In Java



Inheritance

- If "is-a" relationship is exist between the types then we should use inheritance.
- Inheritance is also called as generalization.
- Example
 - 1. Manager is a employee
 - 2. Book is a product
 - 3. Triangle is a shape
 - 4. SavingAccount is a account.

```
class Employee{ //Parent class
    //TODO
}

class Manager extends Employee{ //Child class
    //TODO
}
//Here class Manager is extended from class Employee.
```



Inheritance

- If we want to implement inheritance then we should use extends keyword.
- In Java, parent class is called as super class and child class is called as sub class.
- Java do not support private and protected mode of inheritance
- If Java, class can extend only one class. In other words, multiple class inheritance is not allowed.
- Consider following code:

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

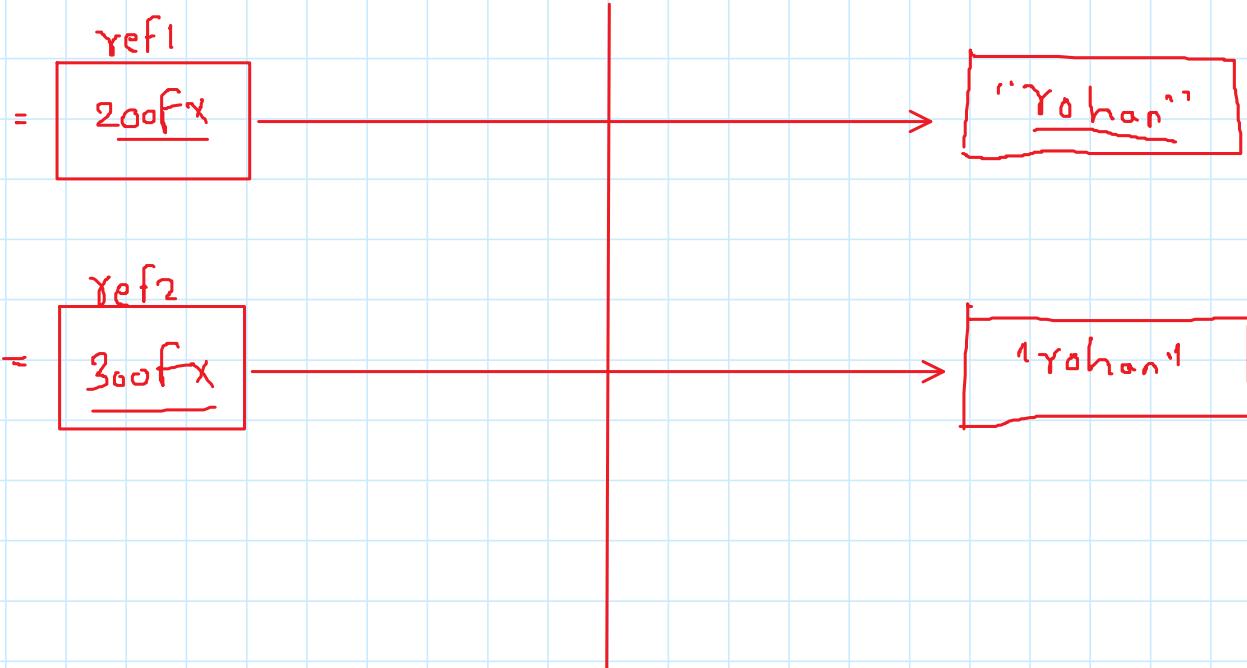




Thank you.

Rohan . paramane@sunbeaminfo . com





String name1 = "Rohan";

String name2 = "Rohan";

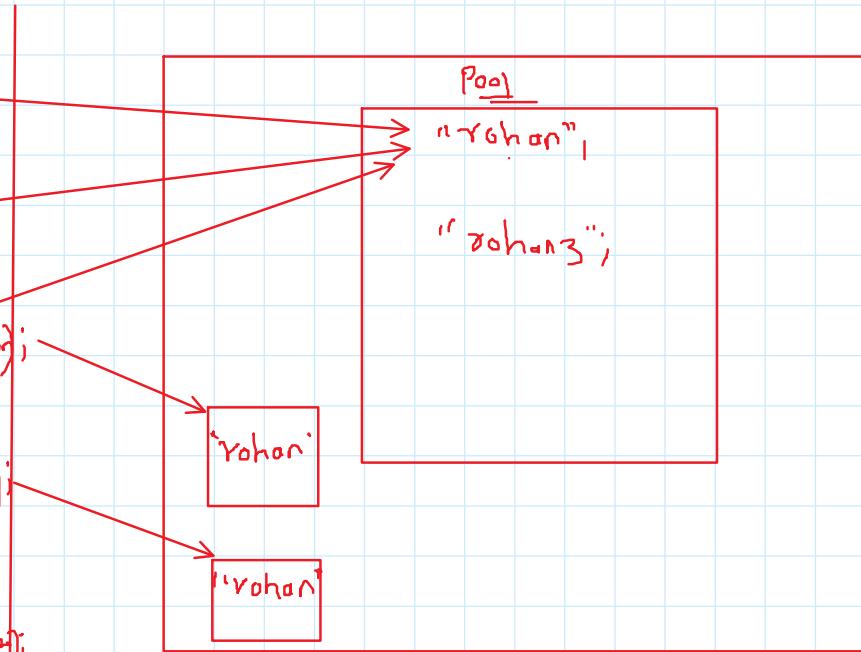
String name3 = new String("Rohan");
Name3 = "Rohan";

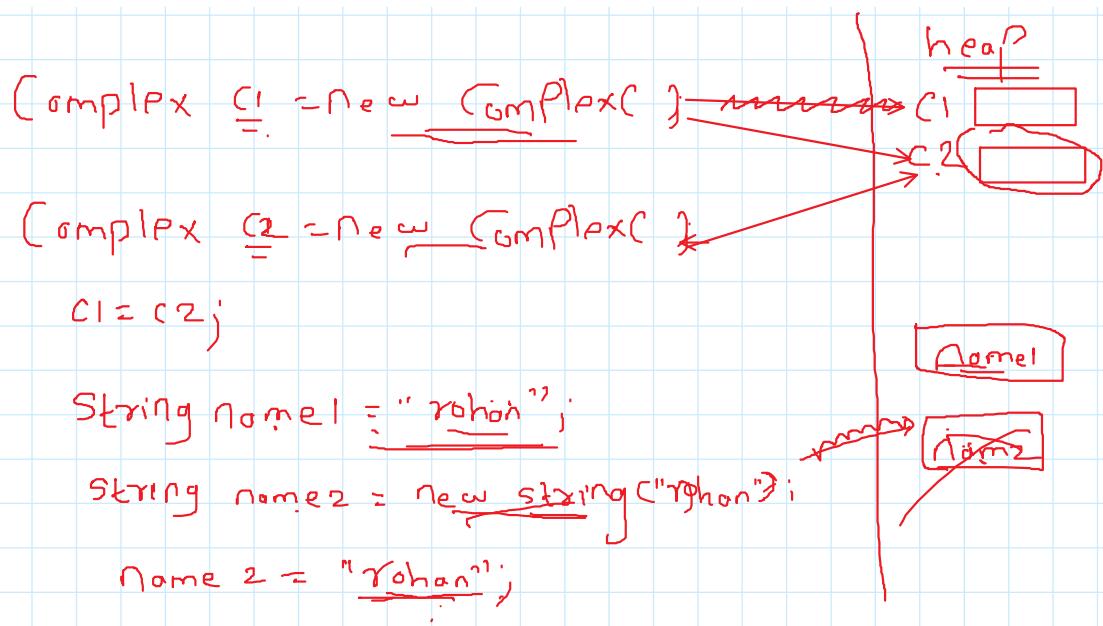
String name4 = new String("rohan");

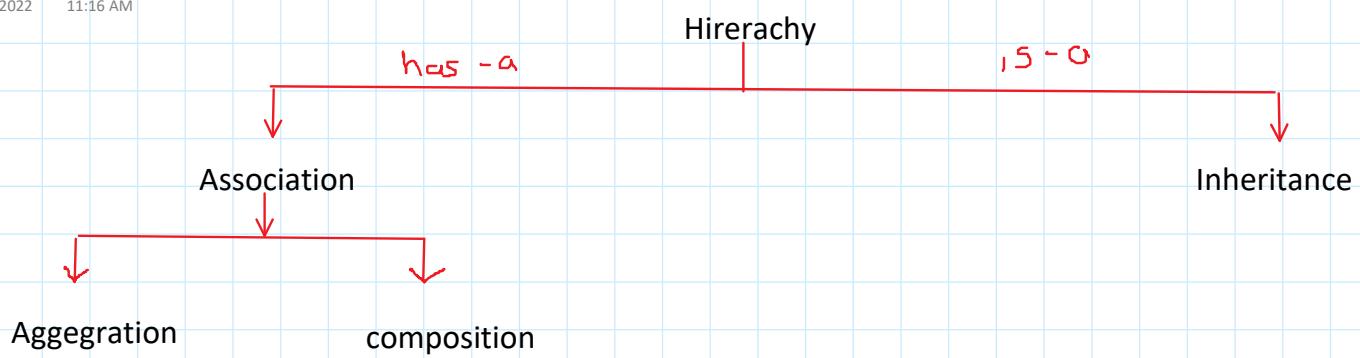
Complex c1 = new Complex(1, 2);

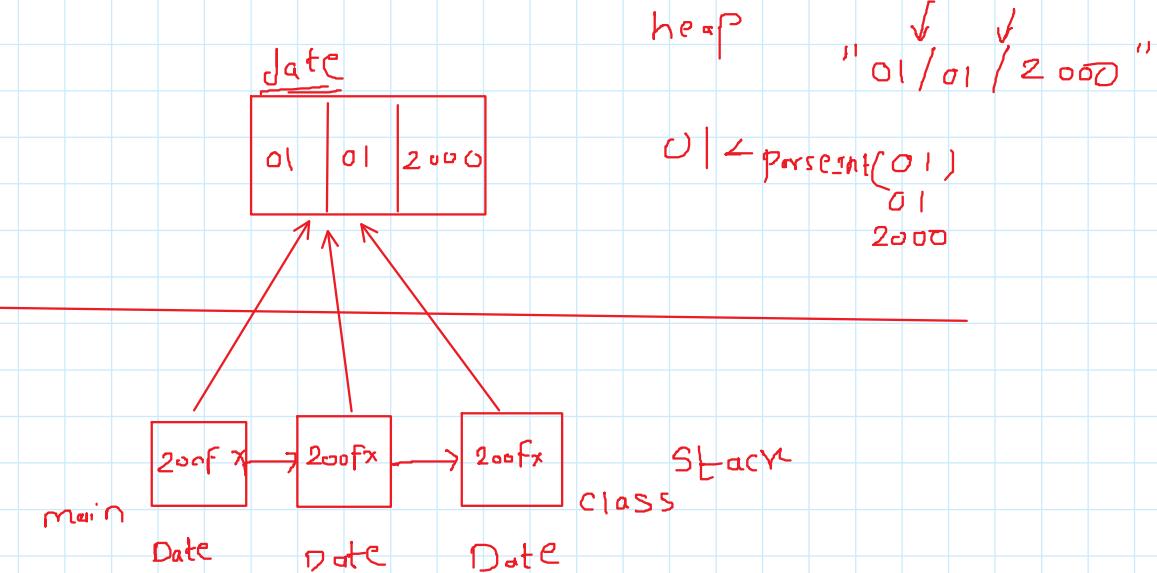
c1.real = 50;

c1 = 50;











Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane

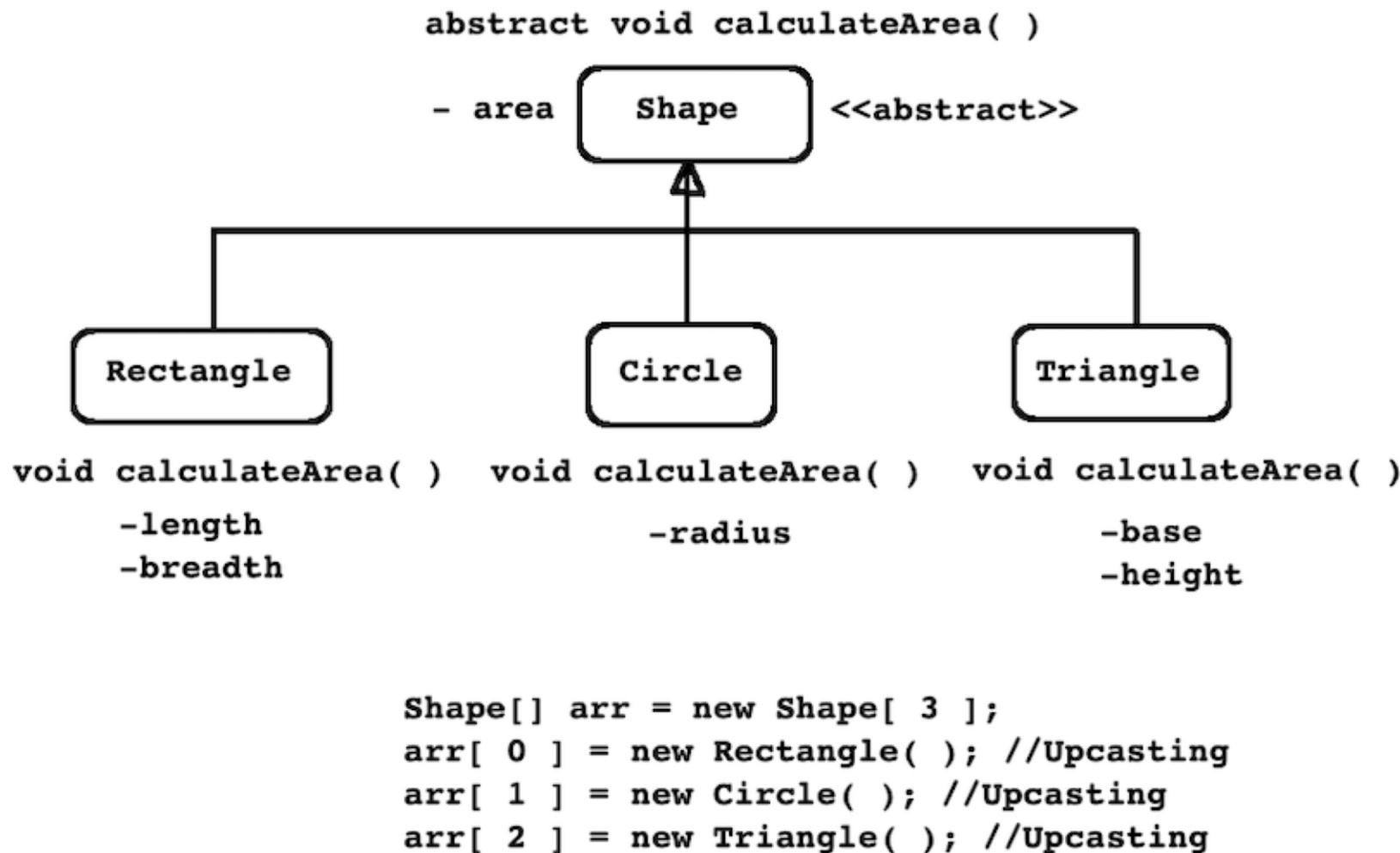


Agenda

- Upcasting and Downcasting
- Abstract Class



Abstract Class



Abstract Class

1. If "is-a" relationship is exist between super type and sub type and if we want same method design in all the sub types then super type must be abstract.
 2. Using abstract class, we can group instances of related type together
 3. Abstract class can extend only one abstract/concrete class.
 4. We can define constructor inside abstract class.
 5. Abstract class may or may not contain abstract method.
- **Hint :** In case of inheritance if state is involved in super type then it should be abstract.



Upcasting

When the reference variable of super class refers to the object of subclass, it is known as widening or **upcasting in java**.

when subclass object type is converted into superclass type, it is called widening or upcasting.



```
Superclass s = new SubClass();
```

Up casting : Assigning child class object to parent class reference .

Syntax for up casting : **Parent p = new Child();**

Here **p** is a parent class reference but point to the child object. *This reference p can access all the methods and variables of parent class but only overridden methods in child class.*

Upcasting gives us the flexibility to access the parent class members, but it is not possible to access all the child class members using this feature. Instead of all the members, we can access some specified members of the child class. For instance, we can only access the overridden methods in the child class.

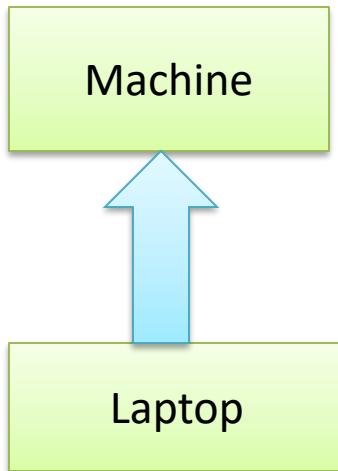


Downcasting

Down casting : Assigning parent class reference (which is pointing to child class object) to child class reference .

Syntax for down casting : **Child c = (Child)p;**

Here **p** is pointing to the object of child class as we saw earlier and now we cast this parent reference **p** to child class reference **c**. Now this child class reference **c** can access all the methods and variables of child class as well as parent class.



For example, if we have two classes, **Machine** and **Laptop** which extends **Machine** class. Now for upcasting, every laptop will be a machine but for downcasting, every machine may not be a laptop because there may be some machines which can be **Printer**, **Mobile**, etc.
Downcasting is not always safe, and we explicitly write the class names before doing downcasting. So that it won't give an error at compile time but it may throw **ClassCastException** at run time, if the parent class reference is not pointing to the appropriate child class.



Explanation

```
Machine machine = new Machine();
```

```
Laptop laptop = (Laptop)machine;//this won't give an error while compiling
```

//laptop is a reference of type Laptop and machine is a reference of type Machine and points to Machine class Object .So logically assigning machine to laptop is invalid because these two classes have different object structure.And hence throws ClassCastException at run time .

To remove ClassCastException we can use instanceof operator to check right type of class reference in case of down casting .

```
if(machine instanceof Laptop)
{
    Laptop laptop = machine; //here machine must be pointing to Laptop class object .
}
```



Polymorphism

- The ability to have many different forms
- An object always has only one form
- A reference variable can refer to objects of different forms



Method Binding

Static Binding

- Static binding
- Compile time
- early binding
- Resolved by java compiler
- Achieved via method overloading

Example :

In class Test :

```
void test(int i,int j){...}
void test(int i) {...}
void test(double i){...}
void test(int i,double j,boolean flag){...}
int test(int a,int b){...} //javac error
```

```
class A {
    A getInstance()
    {
        return new A();
    }
}
```

```
class B extends A
{
    B getInstance()
    {
        return new B();
    }
}
```

Dynamic Binding

- Dynamic binding
- Run time / Late binding
- Resolved by java runtime environment
- Achieved by method overriding (Dynamic method dispatch)
- Method Overriding is a Means of achieving run-time polymorphism

All java methods can be overridden : if they are not marked as private or static or final
Super-class form of method is called as overridden method

sub-class form of method is called as overriding form of the method

Example :



Run time polymorphism or Dynamic method dispatch

- Super -class ref. can directly refer to sub-class object(direct=w/o type casting) as its the example of up-casting(similar to widening auto. conversion) .
- When such a super class ref is used to invoke the overriding method then the method to send for execution that decision is taken by JRE & not by compiler.
- In such case overriding form of the method(sub-class version) will be dispatched for exec.
- Javac resolves the method binding by the type of the reference & JVM resolves the method binding by type of the object it's referring to.
- Super class reference can directly refer to sub-class instance BUT it can only access the members declared in super-class directly.
- eg : A ref=new B();
ref.show(); // this will invoke the sub-class: overriding form of the show () method

Java compiler resolves method binding by type of the reference & JVM resolves it by the type of the object, reference is referring to.





Thank you.

Rohan . paramane@sunbeaminfo . com





Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane



Agenda

- Upcasting and Downcasting
- Abstract Class



Interface

- In Java, an **interface** is a blueprint or template of a class. It is much similar to the Java class but the only difference is that it has abstract methods and static constants.
- An interface provides specifications of what a class should do or not and how it should do. An interface in Java basically has a set of methods that class may or may not apply.
- It also has capabilities to perform a function. The methods in interfaces do not contain any body.
- An interface in Java is a mechanism which we mainly use to achieve abstraction and multiple inheritances in Java.
- An interface provides a set of specifications that other classes must implement.
- We can implement multiple Java Interfaces by a Java class. All methods of an interface are implicitly public and abstract. The word abstract means these methods have no method body, only method signature.
- Java Interface also represents the IS-A relationship of inheritance between two classes.
- An interface can inherit or extend multiple interfaces.
- We can implement more than one interface in our class.

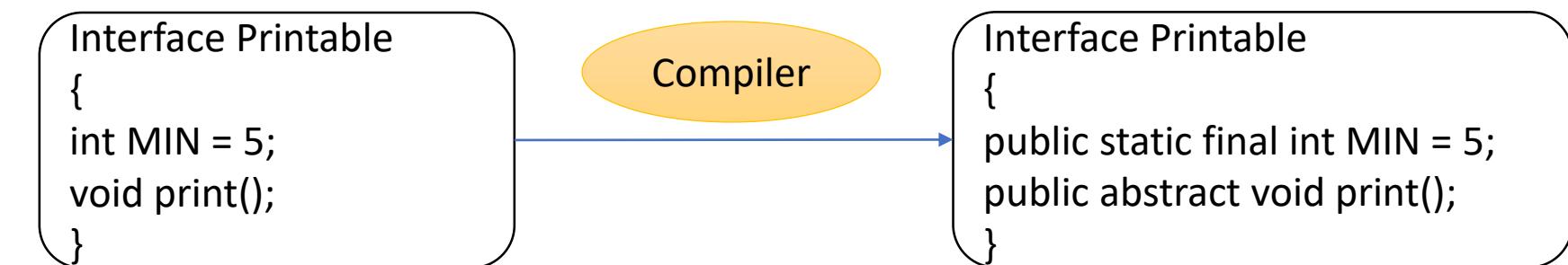


Interface Vs Class

- Unlike a class, you cannot instantiate or create an object of an interface.
- All the methods in an interface should be declared as abstract.
- An interface does not contain any constructors, but a class can.
- An interface cannot contain instance fields. It can only contain the fields that are declared as both static and final.
- An interface can not be extended or inherited by a class; it is implemented by a class.
- An interface cannot implement any class or another interface.

Syntax Interface

```
interface interface-name  
{  
//abstract methods  
}
```



Interface

- Set of rules are called specification/standard.
- It is a contract between service consumer and service provider.
- If we want to define specification for the sub classes then we should define interface.
- Interface is non primitive type which helps developer:
 1. To build/develop trust between service provider and service consumer.
 2. To minimize vendor dependency.
- interface is a keyword in Java.

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



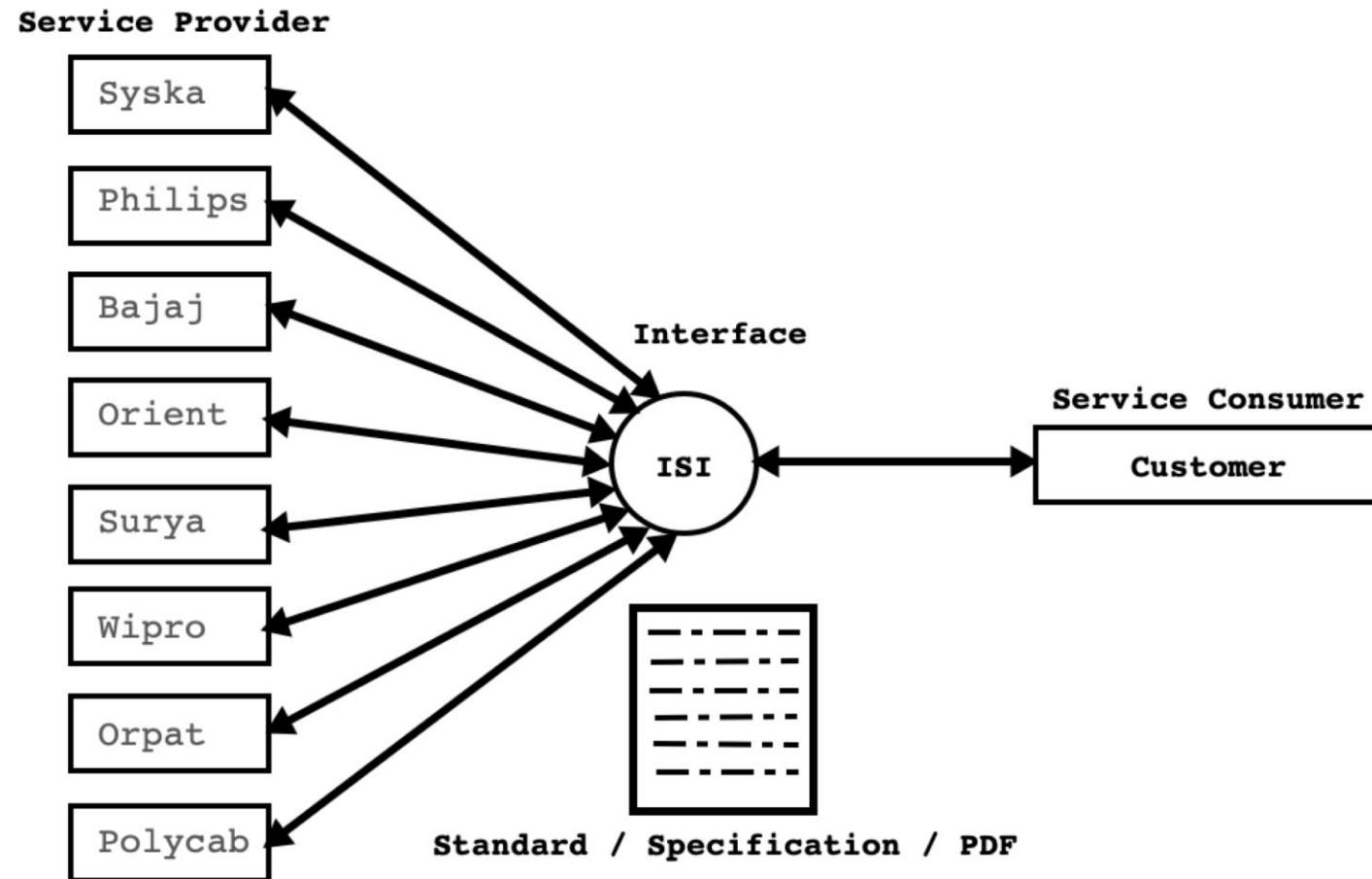
Interface (Features Considering Java8 Interface)

- Interface can contain:
 1. Nested interface
 2. Field
 3. Abstract method
 4. Default method
 5. Static method
- Interfaces cannot have constructors.
- We can create reference of interface but we can not create instance of interface.
- We can declare fields inside interface. Interface fields are by default public static and final.
- We can write methods inside interface. Interface methods are by default considered as public and abstract.

```
interface Printable{  
    int number = 10; //public static final int number = 10;  
    void print( ); //public abstract void print( );  
}
```



Interface



Interface

- If we want to implement rules of interface then we should use implements keyword.
- It is mandatory to override, all the abstract methods of interface otherwise sub class can be considered as abstract.

```
interface Printable{  
    int number = 10;  
    void print( );  
}
```

```
* Solution 1  
abstract class Test implements Printable{  
}
```

```
* Solution 2  
class Test implements Printable{  
    @Override  
    public void print( ){  
        //TODO  
    }  
}
```



Interface Implementation Inheritance

```
interface Printable{
    int number = 10;
    //public static final int number = 10;
    void print( );
    //public abstract void print( );
}

class Test implements Printable{
    @Override
    public void print() {
        System.out.println("Number : "+Printable.number);
    }
}

public class Program {
    public static void main(String[] args) {
        Printable p = new Test(); //Upcasting
        p.print(); //Dynamic Method Dispatch
    }
}
```



Interface Syntax

Interface : I1, I2, I3

Class : C1, C2, C3

- * I2 implements I1 //Incorrect
- * I2 extends I1 //correct : Interface inheritance
- * I3 extends I1, I2 //correct : Multiple interface inheritance
- * C2 implements C1 //Incorrect
- * C2 extends C1 //correct : Implementation Inheritance
- * C3 extends C1,C2 //Incorrect : Multiple Implementation Inheritance
- * I1 extends C1 //Incorrect
- * I1 implements C1 //Incorrect
- * c1 implements I1 //correct : Interface implementation inheritance
- * c1 implements I1,I2 //correct : Multiple Interface implementation inheritance
- * c2 implements I1,I2 extends C1 //Incorrect
- * c2 extends C1 implements I1,I2 //correct



Types of inheritance

- **Interface Inheritance**

- During inheritance if super type and sub type is interface then it is called interface inheritance.
 1. Single Inheritance(Valid in Java)
 2. Multiple Inheritance(Valid in Java)
 3. Hierarchical Inheritance(Valid in Java)
 4. Multilevel Inheritance(Valid in Java)

- **Implementation Inheritance**

- During inheritance if super type and sub type is class then it is called implementation inheritance.
 1. Single Inheritance(Valid in Java)
 2. Multiple Inheritance(Invalid in Java)
 3. Hierarchical Inheritance(Valid in Java)
 4. Multilevel Inheritance(Valid in Java)



Variable Arity/Argument Method

```
private static sum( int... arguments ){
    int result = 0;
    for( int element : arguments )
        result = result + element;
    return result;
}
public static void main(String[] args) {
    int result = 0;
    result = Program.sum( );      //OK
    result = Program.sum( 10, 20, 30 );    //OK
    result = Program.sum( 10, 20, 30, 40, 50 );    //OK
    result = Program.sum( 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 );    //OK
}
```

- Consider Examples from Java API:
 1. public PrintStream printf(String format, Object... args);
 2. public static String format(String format, Object... args);
 3. public Object invoke(Object obj, Object... args);





Thank you.

Rohan . paramane@sunbeaminfo . com





Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane



Agenda

- Exception Handling



Operating System Resources

- Following are the operating system resources that we can use it in the program:
 1. Memory (RAM)
 2. File
 3. Thread
 4. Socket
 5. Connection
 6. IO Devices etc.
- Since OS resources are limited, we should handle it carefully. In other words, we should avoid their leakage.



Resource Type and resource in Java

- AutoCloseable is interface declared in java.lang package.
- Methods:
 1. void close() throws Exception
 2. This method is invoked automatically on objects managed by the try-with-resources statement.
- java.io.Closeable is sub interface of java.lang.AutoCloseable interface.
- Methods:
 1. void close() throws IOException
 2. This method is invoked automatically on objects managed by the try-with-resources statement.



Resource Type and resource in Java

```
//Class Test => Resource Type
class Test implements AutoCloseable{
    private Scanner sc;
    public Test() {
        this.sc = new Scanner(System.in);
    }
    //TODO
    @Override
    public void close() throws Exception {
        this.sc.close();
    }
}
public class Program {
    public static void main(String[] args) {
        Test t = null;
        t = new Test( );      //Resource
    }
}
```



Resource Type and resource in Java

- In the context of exception handling, any class which implements `java.lang.AutoCloseable` or its sub interface(e.g. `java.io.Closeable`) is called resource type and its instance is called as resource.
- We can use instance of only resource type inside try-with-resource.
- `java.util.Scanner` class implements `java.io.Closeable` interface. Hence Scanner class is called as resource type.



Exception Handling

- **Why we should handle exception**

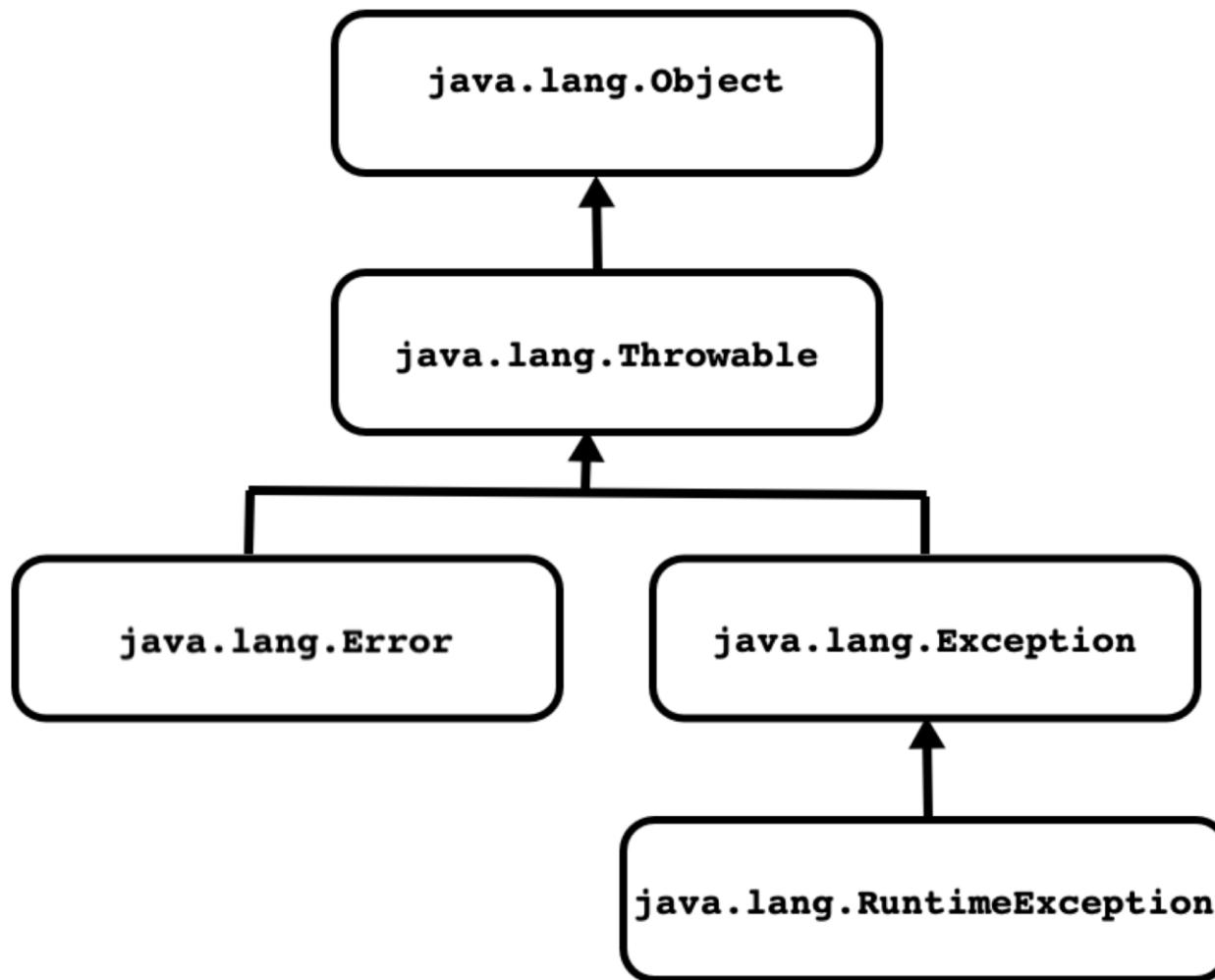
1. To handle all runtime errors at single place. It helps developer to reduces maintenance.
2. To avoid resource leakage/ to manage OS resources carefully.

- **How can we handle exception in Java?**

1. try
2. catch
3. throw
4. throws
5. finally



Exception Handling



Throwable Class

- It is a class declared in `java.lang` package.
- The `Throwable` class is the super class of all errors and exceptions in the Java language.
- Only instances 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.

```
throw 0;      //Not OK

int x = 0;
throw x;      //Not OK

class Test{
}
throw new Test(); //Not OK

class MyException extends Throwable{
}
throw new MyException(); //OK
```



Throwable Class

- Constructors of Throwable class:

1. public Throwable()

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

2. public Throwable(String message)

```
Throwable t1 = new Throwable( "exception message" );
```

3. public Throwable(Throwable cause)

```
Throwable cause = new Throwable();
```

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

4. public Throwable(String message, Throwable cause)

```
Throwable cause = new Throwable();
```

```
Throwable t1 = new Throwable( "exception message", cause );
```



Throwable Class

- **Methods of Throwable class:**

1. public Throwable initCause(Throwable cause)
2. public Throwable getCause()
3. public String getMessage()
4. public void printStackTrace()
5. public void printStackTrace(PrintStream s)
6. public void printStackTrace(PrintWriter s)



Error

- `java.lang.Error` is a sub class of `Throwable` class.
- It gets generated due to environmental condition/Runtime environment(For Example, problem in RAM/JVM, Crashing HDD etc.).
- We can not recover from error hence we should not try to catch error. But can write try-catch block to handle error.
- Example:
 1. `VirtualMachineError`
 2. `OutOfMemoryError`
 3. `InternalError`
 4. `StackOverflowError`



Exception

- `java.lang.Exception` is a sub class of `Throwable` class.
- It gets generated due to application.
- We can recover from exception hence it is recommended to write try-catch block to handle exception in Java.
- Example:
 1. `NumberFormatException`
 2. `NullPointerException`
 3. `NegativeArraySizeException`
 4. `ArrayIndexOutOfBoundsException`
 5. `ArrayStoreException`
 6. `IllegalArgumentException`
 7. `ClassCastException`



Types Of Exception

- **Unchecked Exception**
 - `java.lang.RuntimeException` and all its sub classes are considered as unchecked exception.
 - **It is not mandatory to handle unchecked exception.**
 - Example:
 1. `NullPointerException`
 2. `ClassCastException`
 3. `ArrayIndexOutOfBoundsException`
 - During the execution of arithmetic operation, if any exceptional situation occurs then JVM throws `ArithmaticException`.
- **Checked Exception**
 - `java.lang.Exception` and all its sub classes except `java.lang.RuntimeException` are considered as checked exception.
 - **It is mandatory to handle checked exception.**
 - Example:
 1. `java.lang.CloneNotSupportedException`
 2. `java.lang.InterruptedIOException`



Exception Handling

- **try**
 - It is a keyword in Java.
 - If we want to keep watch on statements for the exception then we should put all such statements inside try block/handler.
 - try block must have at least one:
 1. catch block or
 2. finally block or
 3. Resource
 - We can not define try block after catch or finally block.



Exception Handling

- **Catch**

- It is a keyword in Java.
- If we want to handle exception then we should use catch block/handler
- Only Throwable class or one of its subclasses can be the argument type in a catch clause.
- Catch block can handle exception thrown from try block only.
- For single try block we can define multiple catch block.
- Multi-catch block allows us to handle multiple specific exception inside single catch block.

```
try {  
    //TODO  
}catch (ArithmaticException | InputMismatchException e) {  
    e.printStackTrace( );  
}
```



Exception Handling

Let us consider hierarchy of ArithmeticException class:

- java.lang.Exception
 - java.lang.RuntimeException
 - java.lang.ArithmetricException

```
ArithmetricException e1 = new ArithmetricException( ); //OK
```

```
RuntimeException e2 = new ArithmetricException( ); //OK : Upcasting
```

```
Exception e3 = new ArithmetricException( ); //OK : Upcasting
```

Let us consider hierarchy of InterruptedException class:

- java.lang.Exception
 - java.lang.Interruptedexception

```
InterruptedException e1 = new InterruptedException( );
```

```
Exception e2 = new InterruptedException( ); //OK : Upcasting
```



Exception Handling

- A catch block, which can handle all type of exception is called generic catch block.
- Exception class reference variable can contain reference of instance of any checked as well as unchecked exception. Hence to write generic catch block, we should use `java.lang.Exception` class.

```
try{  
  
}catch( Exception ex ){ //Generic catch block  
    ex.printStackTrace( );  
}
```



Exception Handling

- In case of hierarchy, It is necessary to handle all sub type of exception first.

```
try {
    //TODO
}catch (ArithmaticException e) {
    e.printStackTrace();
}catch (RuntimeException e) {
    e.printStackTrace();
}catch (Exception e) {
    e.printStackTrace();
}
```



Exception Handling

- **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 Java Virtual Machine or can be thrown by the Java throw statement.
 - throw statement is a jump statement.



Exception Handling

- **finally**

- It is a keyword in Java.
- If we want to release local resources then we should use finally block.
- We can not define finally block before try and catch block.
- Try block may have only one finally block.
- JVM always execute finally block.
- If we call System.exit(0) inside try block and catch block then JVM do not execute finally block.



Exception Handling

- **throws**
 - It is a keyword in Java.
 - If we want to redirect/delegate exception from one method to another then we should use throws clause.
 - Consider declaration of following methods:
 1. public static int parseInt(String s) throws NumberFormatException
 2. public static void sleep(long millis) throws InterruptedException



Exception Handling

- **try-with-resources**
 - The try-with-resources statement is a try statement that declares one or more resources.
 - A **resource** is an object that must be closed after the program is finished with it.
 - The try-with-resources statement ensures that each resource is closed at the end of the statement.
 - Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.

```
public static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```



Custom Exception

- JVM can not understand, exceptional situations/conditions of business logic. If we want to handle such exceptional conditions then we should use custom exceptions.

Custom unchecked exception

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

Custom checked exception

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





Thank you.

Rohan . paramane@sunbeaminfo . com

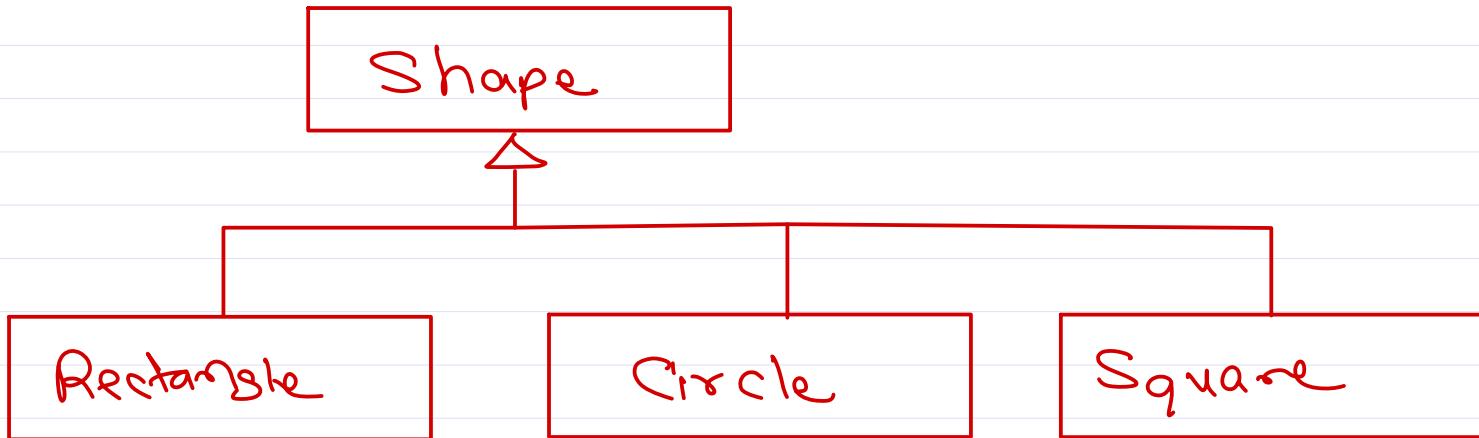




Core Java

Trainer: Nilesh Ghule





class Box < T extends Shape> {

 T obj;

 wid set (T ob) {

 obj = ob;

 T get () {

 3 return obj;

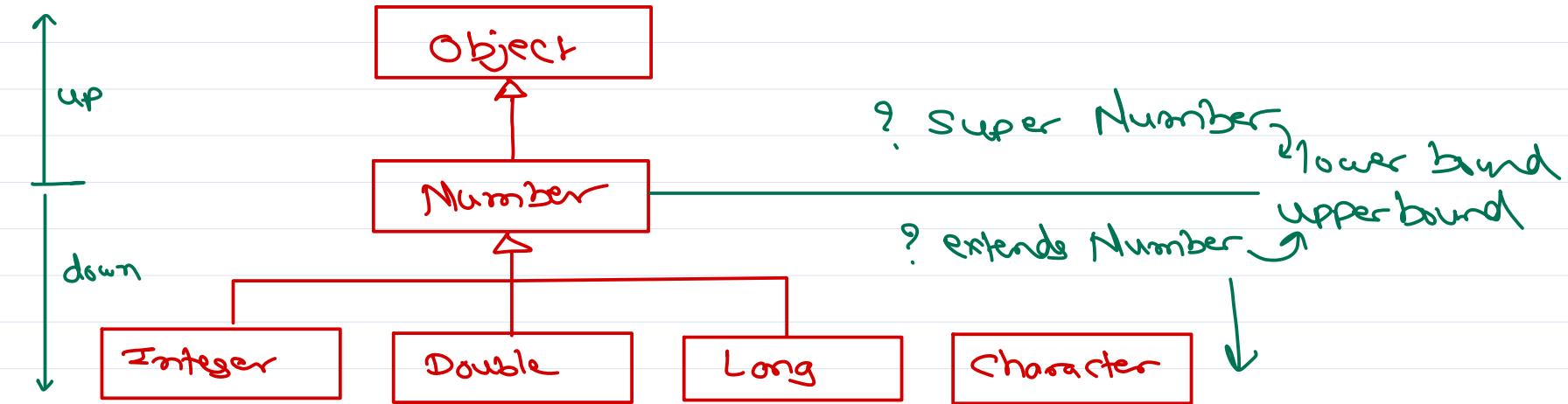
 wid display () {

 sysout (obj. calcArea());

 3

Assignment

- ✓ Box<Rectangle> b1 = new Box<>();
- ✓ Box<Circle> b2 = new Box<>();
- ✓ Box<Shape> b3 = new Box<>();
- ✗ Box<Date> b4 = new Box<>();
↳ Compiler error.





Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>



Core Java

Agenda

- Generic Programming
 - Introduction
 - Generic Classes
 - Advantages of Generics
 - Bounded & Unbounded generic types
 - Upper & Lower bounded generic types
 - Generic Methods
 - Generics Limitations
- Comparable vs Comparator interfaces

Generic Programming

- Code is said to be generic if same code can be used for various (practically all) types.
- Best example:
 - Data structure e.g. Stack, Queue, Linked List, ...
 - Algorithms e.g. Sorting, Searching, ...

Introduction

- Two ways to do Generic Programming in Java
 - using `java.lang.Object` class
 - using Generics

Using `java.lang.Object`

```
```Java
class Box {
 private Object obj;
 public void set(Object obj) {
 this.obj = obj;
 }
 public Object get() {
 return this.obj;
 }
}
```
```Java
Box b1 = new Box();
b1.set("Nilesh");
String obj1 = (String)b1.get();
System.out.println("obj1 : " + obj1);

Box b2 = new Box();
b2.set(new Date());
Date obj2 = (Date)b2.get();
System.out.println("obj2 : " + obj2);

Box b3 = new Box();
b3.set(new Integer(11));
String obj3 = (String)b3.get(); // ClassCastException
System.out.println("obj3 : " + obj3);
```

```

Using Generics

- Similar to templates in C++.

- We can implement generic classes, interfaces, and methods (as per requirement).
- Added in Java 5.0.

Generic Classes

- Implementing a generic class

```
class Box<TYPE> {  
    private TYPE obj;  
    public void set(TYPE obj) {  
        this.obj = obj;  
    }  
    public TYPE get() {  
        return this.obj;  
    }  
}
```

```
Box<String> b1 = new Box<String>();  
b1.set("Nilesh");  
String obj1 = b1.get();  
System.out.println("obj1 : " + obj1);  
  
Box<Date> b2 = new Box<Date>();  
b2.set(new Date());  
Date obj2 = b2.get();  
System.out.println("obj2 : " + obj2);  
  
Box<Integer> b3 = new Box<Integer>();  
b3.set(new Integer(11));  
String obj3 = b3.get(); // Compiler Error  
System.out.println("obj3 : " + obj3);
```

- Instantiating generic class

```
Box<String> b1 = new Box<String>(); // okay

Box<String> b2 = new Box<>(); // okay -- type inference

Box<> b3 = new Box<>(); // compiler error

Box<Object> b4 = new Box<String>(); // compiler error

Box b5 = new Box(); // okay -- if generic type not mentioned, raw type (Object) is considered

Box<Object> b6 = new Box<Object>(); // okay -- can store object of any type -- not usually needed
```

Advantages of Generics

- Stronger type checking at compile time i.e. type-safe coding.
- Explicit type casting is not required.
- Generic data structure and algorithm implementation.

Generic types naming convention

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value
6. S,U,R : Additional type param

Bounded generic types

- Bounded generic param restricts data type that can be used as type argument.
- Decided by the developer of the generic class.

```
class Box<T extends Number> {  
    private T obj;  
    public T get() {  
        return this.obj;  
    }  
    public void set(T obj) {  
        this.obj = obj;  
    }  
}
```

- The Box<> can now be used only for the classes inherited from the Number class.

1. Box<Number> b1 = new Box<>(); // okay
2. Box<Boolean> b2 = new Box<>(); // compiler error
3. Box<Character> b3 = new Box<>(); // compiler error
4. Box<String> b4 = new Box<>(); // compiler error
5. Box<Integer> b5 = new Box<>(); // okay
6. Box<Double> b6 = new Box<>(); // okay
7. Box<Date> b7 = new Box<>(); // compiler error

Unbounded generic types

- Unbounded generic type is indicated with wild-card "?".
- Can be given while declaring generic class reference.

```
class Box<T> {  
    private T obj;  
    public Box(T obj) {  
        this.obj = obj;  
    }  
    public T get() {  
        return this.obj;  
    }  
    public void set(T obj) {  
        this.obj = obj;  
    }  
}
```

```
public static void printBox(Box<?> b) {  
    Object obj = b.get();  
    System.out.println("Box contains: " + obj);  
}
```

```
Box<String> sb = new Box<String>("DAC");  
printBox(sb); // ?  
Box<Integer> ib = new Box<Integer>(100);  
printBox(ib); // ?  
Box<Date> db = new Box<Date>(new Date());  
printBox(db); // ?  
Box<Float> fb = new Box<Float>(200.5);  
printBox(fb); // ?
```

Upper bounded generic types

- Generic param type can be the given class or its sub-class.

```
public static void printBox(Box<? extends Number> b) {  
    Object obj = b.get();  
    System.out.println("Box contains: " + obj);  
}
```

```
Box<String> sb = new Box<String>("DAC");  
printBox(sb); // ?  
Box<Integer> ib = new Box<Integer>(100);  
printBox(ib); // ?  
Box<Date> db = new Box<Date>(new Date());  
printBox(db); // ?  
Box<Float> fb = new Box<Float>(200.5);  
printBox(fb); // ?
```

Lower bounded generic types

- Generic param type can be the given class or its super-class.

```
public static void printBox(Box<? super Integer> b) {  
    Object obj = b.get();  
    System.out.println("Box contains: " + obj);  
}
```

```
Box<String> sb = new Box<String>("DAC");  
printBox(sb); // ?
```

```
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // ?
Box<Date> db = new Box<Date>(new Date());
printBox(db); // ?
Box<Float> fb = new Box<Float>(200.5f);
printBox(fb); // ?
```

Generic Methods

- Generic methods are used to implement generic algorithms.
- Example:

```
// non type-safe
void printArray(Object[] arr) {
    for(Object ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
// type-safe
<T> void printArray(T[] arr) {
    for(T ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
String[] arr1 = { "John", "Dagny", "Alex" };
printArray(arr1); // printArray<String> -- String type is inferred

Integer[] arr2 = { 10, 20, 30 };
printArray(arr2); // printArray<Integer> -- Integer type is inferred
```

Generics Limitations

1. Cannot instantiate generic types with primitive Types. Only reference types are allowed.

```
ArrayList<Integer> list = new ArrayList<Integer>(); // okay
ArrayList<int> list = new ArrayList<int>(); // compiler error
```

2. Cannot create instances of Type parameters.

```
ArrayList<Integer> list = new ArrayList<Integer>(); // okay
ArrayList<T> list = new ArrayList<T>(); // compiler error
```

3. Cannot declare static fields with generic type parameters.

```
class Box<T> {
    private T obj; // okay
    private static T object; // compiler error
    // ...
}
```

4. Cannot use casts or instanceof with generic Type params.

```
if(obj instanceof T) { // compiler error  
    newobj = (T)obj; // compiler error  
}
```

5. Cannot Create arrays of generic parameterized Types

```
T[] arr = new T[5]; // compiler error
```

6. Cannot create, catch, or throw Objects of Parameterized Types

```
throw new T(); // compiler error  
  
try {  
    // ...  
} catch(T ex) { // compiler error  
    // ...  
}
```

7. Cannot overload a method just by changing generic type. Because after erasing/removing the type param, if params of two methods are same, then it is not allowed.

```
public void printBox(Box<Integer> b) {  
    // ...  
}  
public void printBox(Box<String> b) { // compiler error
```

```
// ...  
}
```

Comparable<T> vs Comparator<T>

Comparable<T>

- Standard for comparing the current object to the other object.
- Has single abstract method `int compareTo(T other);`
- In `java.lang` package.
- Used by various methods like `Arrays.sort(Object[])`, ...

```
// pre-defined interface  
interface Comparable<T> {  
    int compareTo(T other);  
}
```

```
class Employee implements Comparable<Employee> {  
    private int empno;  
    private String name;  
    private int salary;  
    // ...  
    public int compareTo(Employee other) {  
        int diff = this.empno - other.empno;  
        return diff;  
    }  
}
```

```
Employee e1 = new Employee(1, "Sarang", 50000);
Employee e2 = new Employee(2, "Nitin", 40000);
int diff = e1.compareTo(e2);
```

```
Employee[] arr = { ... };
Arrays.sort(arr);
for(Employee e:arr)
    System.out.println(e);
```

Comparator<>

- Standard for comparing two (other) objects.
- Has single abstract method `int compare(T obj1, T obj2);`
- In `java.util` package.
- Used by various methods like `Arrays.sort(T[], comparator)`, ...

```
// pre-defined interface
interface Comparator<T> {
    int compare(T obj1, T obj2);
}
```

```
class EmployeeSalaryComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
```

```
        if(e1.getSalary() == e2.getSalary())
            return 0;
        if(e1.getSalary() > e2.getSalary())
            return +1;
        return -1;
    }
}
```

Multi-level sorting

```
class Employee implements Comparable<Employee> {
    private int empno;
    private String name;
    private String designation;
    private int department;
    private int salary;
    // ...
}
```

```
// Multi-level sorting -- 1st level: department, 2nd level: designation, 3rd level: name
class CustomComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        int diff = e1.getDepartment().compareTo(e2.getDepartment());
        if(diff == 0)
            diff = e1.getDesignation().compareTo(e2.getDesignation());
        if(diff == 0)
            diff = e1.getSalary() - e2.getSalary();
        return diff;
    }
}
```

```
Employee[] arr = { ... };
Arrays.sort(arr, new CustomComparator());
// ...
```

Assignment

- Implement generic class Box so that it can store any Shape in it. (Refer slides).
- Write a generic static method to find minimum from an array of Number.
- Write a generic sort method for implementing selection sort algorithm with given comparator. Refer code below.

```
<T> selectionSort(T[] arr, Comparator c) {
    for(int i=0; i<arr.length; i++) {
        for(int j=i+1; j<arr.length; j++) {
            if(c.compare(arr[i], arr[j]) > 0) {
                T temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

- Use Arrays.sort() to sort array of Students using Comparator. The 1st level sorting should be on city (desc), 2nd level sorting should be on marks (desc), 3rd level sorting should be on name (asc).

```
class Student {
    private int roll;
    private String name;
```

```
private String city;  
private double marks;  
// ...  
}
```

SUNBEAM INFOTECH



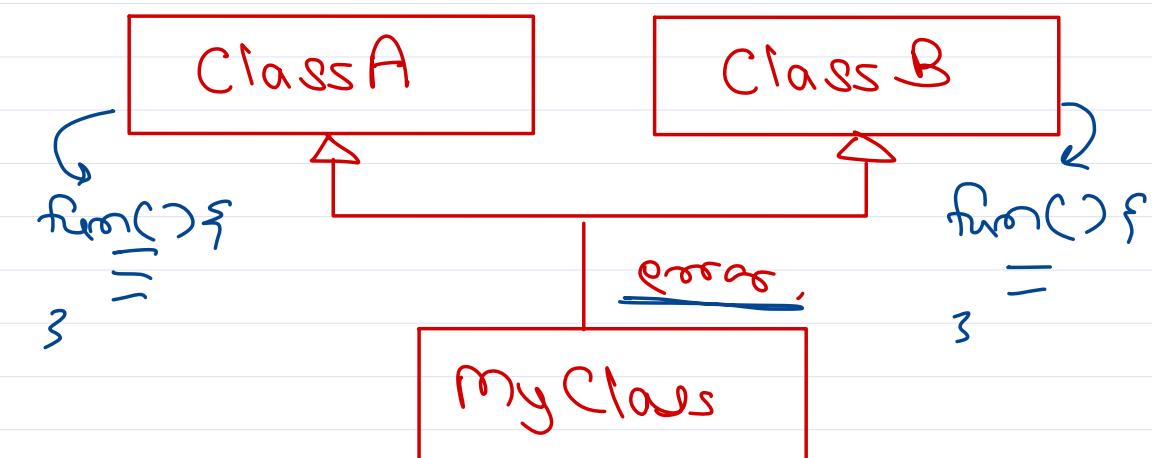
Core Java

Trainer: Nilesh Ghule

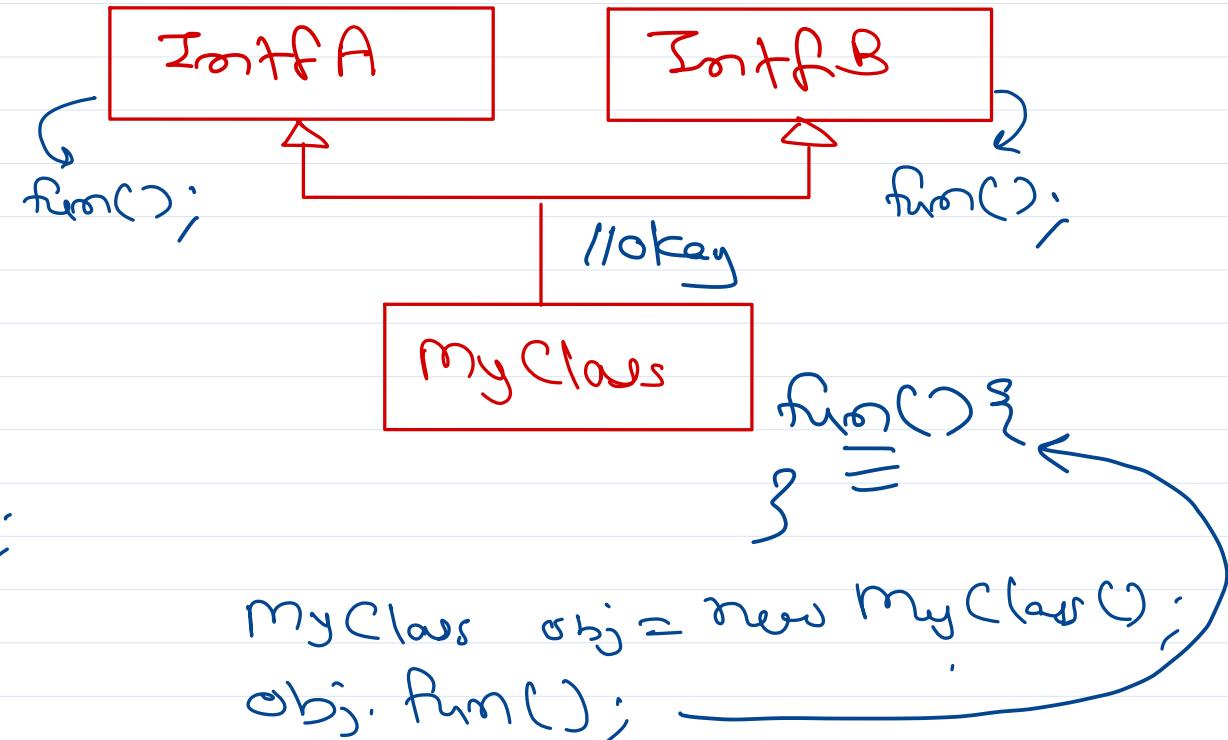


Sunbeam Infotech

www.sunbeaminfo.com



`MyClass obj = new MyClass();`
`obj. func();` → ambiguity



`MyClass obj = new MyClass();`
`obj. func();`



Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>



Core Java

Agenda

- Quick Revision
- Member classes
- Java 8 Interfaces
 - Default methods
 - Static methods
 - Functional Interface
 - Built-in functional interfaces
 - Lambda expressions
 - Method references

Quick Revision

- Generic programming enable us to write common code for different data types.
- Generic classes, interfaces and methods.
- Java generics work with reference types only.
- For generic classes/interfaces generic type needs to be given while instantiating it; while for generic methods type is inferred automatically.
- Comparable and Comparator are generic interfaces that provide standard way of comparing two objects.
- Java generics are most used with data structures and algorithms (collections).

Member/Nested classes

- By default all Java classes are top-level.
- In Java classes can be written inside another class/method. They are Member classes.
- Four types of member classes
 - Static member classes
 - Non-static member class

- Local class
- Anonymous Inner class
- When .java file is compiled, separate .class file created for outer class as well as inner class.

Static member classes

- Like other static members of the class (belong to the class, not the object).
- Accessed using outer class (Doesn't need the object of outer class).
- Can access static (private) members of the outer class directly.
- The static member classes can be private, public, protected, or default.

```
class Outer {  
    private int nonStaticField = 10;  
    private static int staticField = 20;  
    public static class Inner {  
        public void display() {  
            System.out.println("Outer.nonStaticField = " + nonStaticField); // error  
            System.out.println("Outer.staticField = " + staticField); // okay  
        }  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        Outer.Inner obj = new Outer.Inner();  
        obj.display();  
    }  
}
```

Non-static member classes/Inner classes

- Like other non-static members of the class (belong to the object-instance of Outer class).

- Accessed using outer class object (Object of outer class is must).
- Can access static & non-static (private) members of the outer class directly.

```
class Outer {  
    private int nonStaticField = 10;  
    private static int staticField = 20;  
    class Inner {  
        public void display() {  
            System.out.println("Outer.nonStaticField = " + nonStaticField); // okay  
            System.out.println("Outer.staticField = " + staticField); // okay  
        }  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        Outer outObj = new Outer();  
        Outer.Inner obj = outObj.new Inner();  
        // Outer.Inner obj = new Outer().new Inner();  
        obj.display();  
    }  
}
```

- If Inner class member has same name as of outer class member, it shadows (hides) the outer class member. Such Outer class members can be accessed explicitly using `Outer.this`.

Static member class and Non-static member class -- Application

```
// top-level class  
class LinkedList {  
    // static member class  
    static class Node {
```

```
private int data;
private Node next;
// ...
}
private Node head;
// non-static member class
class Iterator {
    private Node trav;
    // ...
}
// ...
public void display() {
    Node trav = head;
    while(trav != null) {
        System.out.println(trav.data);
        trav = trav.next;
    }
}
```

Local class

- Like local variables of a method.
- The class scope is limited to the enclosing method.
- If enclosed in static method, behaves like static member class. If enclosed in non-static method, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.
- We can create any number of objects of local classes within the enclosing method.

```
public class Main {
    private int nonStaticField = 10;
    private static int staticField = 20;
    public static void main(String[] args) {
```

```
final int localField1 = 1;
int localField2 = 2;
int localField3 = 3;
localField3++;
// local class
class Inner {
    public void display() {
        System.out.println("Outer.nonStaticField = " + nonStaticField); //error
        System.out.println("Outer.staticField = " + staticField); // okay
        System.out.println("Main.localField1 = " + localField1); // okay
        System.out.println("Main.localField2 = " + localField2); // okay
        System.out.println("Main.localField3 = " + localField3); // error
    }
}
Inner obj = new Inner();
obj.display();
//new Inner().display();
}
```

Annoymous Inner class

- Creates a new class inherited from the given class/interface and its object is created.
- If in static context, behaves like static member class. If in non-static context, behaves like non-static member class.
- Along with Outer class members, it can also access () final local variables of the enclosing method.

```
// (named) local class
class EmpnoComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
}
```

```
Arrays.sort(arr, new EmpnoComparator()); // anonymous obj of local class

// A new Anonymous inner class is created which is inherited from Comparator.
// it's named object is created "cmp" and passed to sort() method
Comparator<Employee> cmp = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getName().compareTo(e2.getName());
    }
};
Arrays.sort(arr, cmp); // sort by name

// A new Anonymous inner class is created which is inherited from Comparator
// a new Anonymous object of that Anonymous inner class is created and passed to sort() method.
Arrays.sort(arr, new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getSalary() - e2.getSalary();
    }
}); // sort by salary
```

Java 8 Interfaces

- Before Java 8
 - Interfaces are used to design specification/standards. It contains only declarations – public abstract.

```
interface Geometry {
    /*public static final*/ double PI = 3.14;
    /*public abstract*/ int calcRectArea(int length, int breadth);
    /*public abstract*/ int calcRectPeri(int length, int breadth);
}
```

- As interfaces doesn't contain method implementations, multiple interface inheritance is supported (no ambiguity error).
- Interfaces are immutable. One should not modify interface once published.

- Java 8 added many new features in interfaces in order to support functional programming in Java. Many of these features also contradicts earlier Java/OOP concepts.

Default methods

- Java 8 allows default methods. If method is not overridden, its default implementation in interface is considered.
- This allows adding new functionalities into existing interfaces without breaking old implementations e.g. Collection, Comparable, ...

```
interface Emp {  
    double getSal();  
    default double calcIncentives() {  
        return 0.0;  
    }  
}  
class Manager implements Emp {  
    // ...  
    double calcIncentives() {  
        return getSal() + getSal() * 0.2;  
    }  
}  
class Clerk implements Emp {  
    // ...  
}
```

```
new Manager().calcIncentives();  
new Clerk().calcIncentives();
```

- However default methods will lead to ambiguity errors as well, if same default method is available from multiple interfaces.
- Superclass same method get higher priority, while other interface same method will lead to error.
 - Super-class wins! Super-interfaces clash!!

```
interface Displayable {  
    default void show() {  
        System.out.println("Displayable.show() called");  
    }  
}  
interface Printable {  
    default void show() {  
        System.out.println("Printable.show() called");  
    }  
}  
class FirstClass implements Displayable, Printable { // compiler error -- ambiguity error  
    // ...  
}  
class Main {  
    public static void main(String[] args) {  
        FirstClass obj = new FirstClass();  
        obj.show();  
    }  
}
```

```
interface Displayable {  
    default void show() {  
        System.out.println("Displayable.show() called");  
    }  
}  
interface Printable {  
    default void show() {  
        System.out.println("Printable.show() called");  
    }  
}  
class Superclass {  
    public void show() {
```

```
        System.out.println("Superclass.show() called");
    }
}
class SecondClass extends Superclass implements Displayable, Printable {
    // ...
}
class Main {
    public static void main(String[] args) {
        SecondClass obj = new SecondClass();
        obj.show();
    }
}
```

- A class can invoke methods of super interfaces using InterfaceName.super.

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class FourthClass implements Displayable, Printable {
    @Override
    public void show() {
        System.out.println("FourthClass.show() called");
        Displayable.super.show();
        Printable.super.show();
    }
}
```

```
class Main {  
    public static void main(String[] args) {  
        FourthClass obj = new FourthClass();  
        obj.show(); // calls FourthClass method  
    }  
}
```

Static methods

- Before Java 8 interfaces allowed public static final fields.
- Java 8 also allows the static methods in interfaces.
- They act as helper methods and thus eliminates need of helper classes like Collections, ...

```
interface Emp {  
    double getSal();  
    public static double calcTotalSalary(Emp[] a) {  
        double total = 0.0;  
        for(int i=0; i<a.length; i++)  
            total += a[i].getSal();  
        return total;  
    }  
}
```

Functional Interface

- If interface contains exactly one abstract method (SAM), it is said to be functional interface.
- It may contain additional default & static methods. E.g. Comparator, Runnable, ...
- @FunctionalInterface annotation does compile time check, whether interface contains single abstract method. If not, raise compile time error.

```
@FunctionalInterface // okay
interface Foo {
    void foo();
}

@FunctionalInterface // okay
interface FooBar1 {
    void foo();
    default void bar() {
        /*... */
    }
}

@FunctionalInterface // error
interface FooBar2 {
    void foo();
    void bar();
}

@FunctionalInterface // error
interface FooBar3 {
    default void foo() {
        /*... */
    }
    default void bar() {
        /*... */
    }
}
```

- Functional interfaces forms foundation for Java lambda expressions and method references.

Built-in functional interfaces

- New set of functional interfaces given in `java.util.function` package.
 - `Predicate`: `test: T -> boolean`
 - `Function<T, R>`: `apply: T -> R`

- BiFunction<T,U,R>: apply: (T,U) -> R
 - UnaryOperator: apply: T -> T
 - BinaryOperator: apply: (T,T) -> T
 - Consumer: accept: T -> void
 - Supplier: get: () -> T
- For efficiency primitive type functional interfaces are also supported e.g. IntPredicate, IntConsumer, IntSupplier, IntToDoubleFunction,ToIntFunction, ToIntBiFunction, IntUnaryOperator, IntBinaryOperator.

Lambda expressions

- Traditionally Java uses anonymous inner classes to compact the code. For each inner class separate .class file is created.
- However code is complex to read and un-efficient to execute.
- Lambda expression is short-hand way of implementing functional interface.
- Its argument types may or may not be given. The types will be inferred.
- Lambda expression can be single liner (expression not statement) or multi-liner block { ... }.

```
Arrays.sort(arr, new Comparator<Employee>(){  
    public int compare(Employee e1, Employee e2) {  
        return e1.getSalary() - e2.getSalary();  
    }  
});
```

```
Arrays.sort(arr, (Employee e1, Employee e2) -> {  
    int diff = e1.getSalary() - e2.getSalary();  
    return diff;  
});
```

```
Arrays.sort(arr, (e1, e2) -> {
    int diff = e1.getSalary() - e2.getSalary();
    return diff;
});
```

```
Arrays.sort(arr, (e1, e2) -> {
    return e1.getSalary() - e2.getSalary();
});
```

```
Arrays.sort(arr, (e1, e2) -> e1.getSalary() - e2.getSalary());
```

- Practically lambda expressions are used to pass as argument to various functions.
- Lambda expression enable developers to write concise code (single liners recommended).

Non-capturing lambda expression

- If lambda expression result entirely depends on the arguments passed to it, then it is non-capturing (self-contained).

```
BinaryOperator<Integer> op1 = (a,b) -> a + b;
```

Capturing lambda expression

- If lambda expression result also depends on additional variables in the context of the lambda expression passed to it, then it is capturing.

```
int c = 2;
BinaryOperator<Integer> op2 = (a,b) -> a + b + c;
```

- Here variable c is bound (captured) into lambda expression. So it can be accessed even out of scope. Internally it is associated with the method.
- In some languages, this is known as Closures.

Method references

- If lambda expression involves single method call, it can be shortened by using method reference.
- Method references are converted into instances of functional interfaces.
- Method reference can be for class static method, class non-static method, object non-static method or constructor.

Examples

- Class static method: Integer::sum [(a,b) -> Integer.sum(a,b)]
 - Both lambda param passed to static function explicitly
- Class non-static method: String::compareTolgnoreCase [(a,b) -> a. compareTolgnoreCase(b)]
 - First lambda param become implicit param of the function and second is passed explicitly.
- Object non-static method: System.out::println [x -> System.out.println(x)]
 - Lambda param is passed to function explicitly.
- Constructor: Date::new [() -> new Date()]
 - Lambda param is passed to constructor explicitly.

Assignment

- Create an interface Emp with abstract method `double getSal()` and a default method `default double calcIncentives()`. The default method simply returns 0.0. Create a class Manager (with fields basicSalary and dearanceAllowance) inherited from Emp. In this class override getSal() method (basicSalary + dearanceAllowance) as well as calcIncentives() method (20% of basicSalary). Create another class Labor (with fields hours and rate) inherited from Emp interface. In this class override getSal() method (hours * rate) as well as calcIncentives() method (5% of salary if hours > 300, otherwise no incentives). Create another class Labor (with field salary) inherited from Emp interface. In this class override getSal() method (salary). Do not override, calcIncentives() in Clerk

class. In Emp interface create a static method `static double calcTotalIncome(Employee arr[])` that calculate total income (salary + incentives) of all employees in the given array.

- Use following method to count number of strings with length > 6 in given array.

```
public static int countIf(String[] arr, Predicate<String> cond) {  
    int count = 0;  
    for(String str: arr) {  
        if(cond.test(str))  
            count++;  
    }  
    return count;  
}
```

```
public static void main(String[] args) {  
    String[] arr = { "Nilesh", "Shubham", "Pratik", "Omkar", "Prashant" };  
    // call countIf() to count number of strings have length more than 6 -- using lambda expressions  
}
```



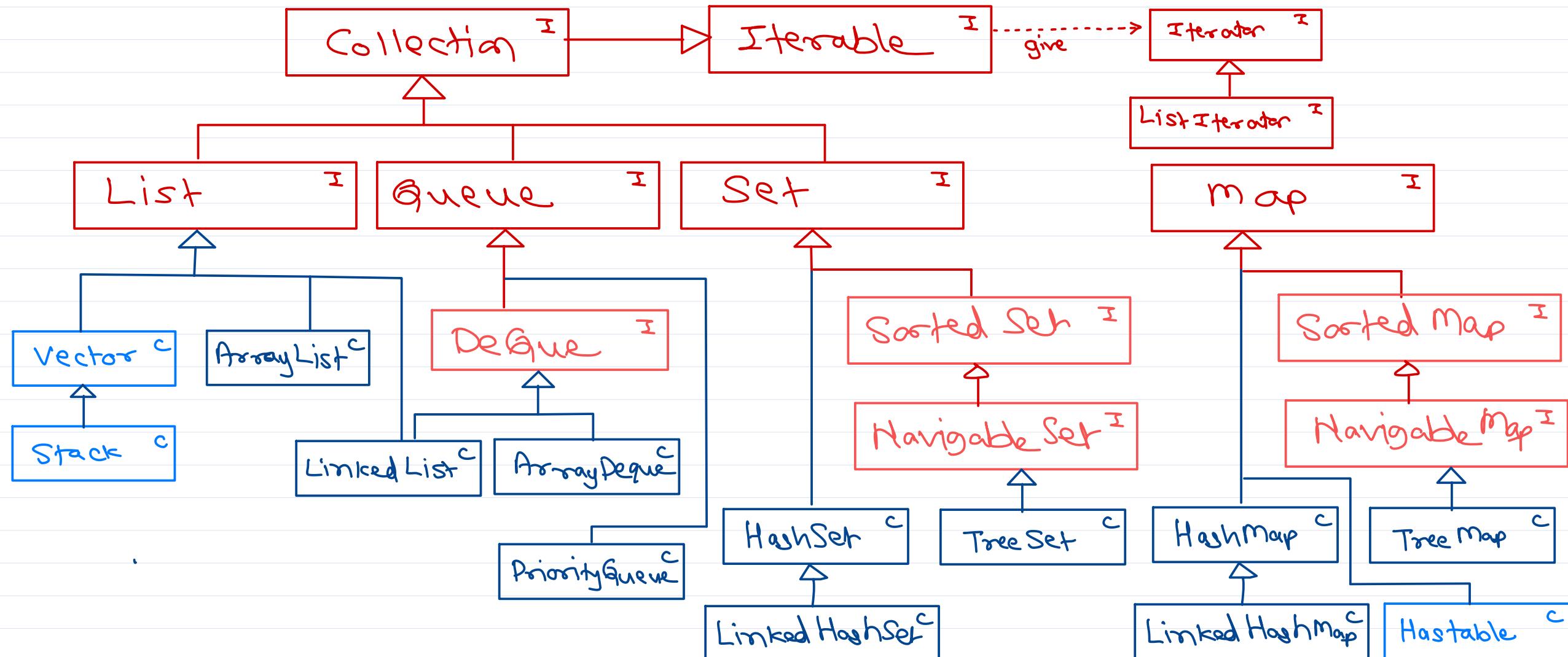
Core Java

Trainer: Nilesh Ghule



Sunbeam Infotech

www.sunbeaminfo.com



Application

Java API(Library)

```

class ArrayList {
    Iterator iterator() {
        return new MyIterator();
    }
}

itr = list.iterator();
while (itr.hasNext()) {
    e = itr.next();
}

```

Diagram illustrating the delegation pattern:

- The `ArrayList` class delegates the `iterator()` method to its `MyIterator` inner class.
- The `MyIterator` class implements the `Iterator` interface, delegating the `hasNext()` and `next()` methods to its own implementation.
- Red annotations show the delegation paths: `list.iterator()` points to the `ArrayList.iterator()` method, and `itr.next()` points to the `MyIterator.next()` method.
- Red numbers (3) are placed near the delegation points to indicate the flow of control.

Application

Java API(Library)

```

class ArrayList {
    private int len;
    forEach(Consumer c) {
        for (int i=0; i<len; i++) {
            c.accept(arr[i]);
        }
    }
}

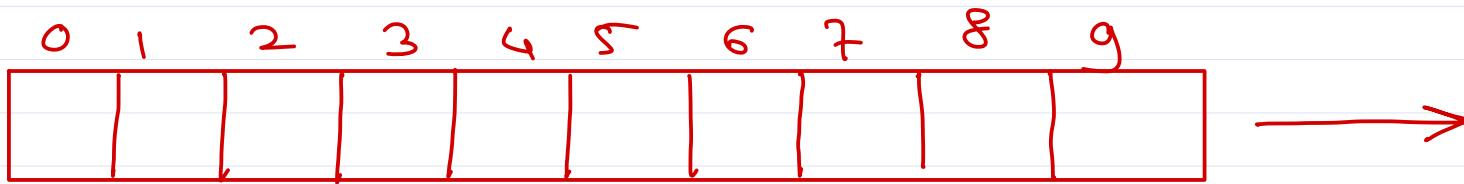
list.forEach(e->System.out.println(e));

```

Diagram illustrating the Java Stream API's `forEach` method:

- The `list.forEach` method delegates to the `forEach` method of the `ArrayList` class.
- The `forEach` method of `ArrayList` delegates to the `accept` method of the `Consumer` interface.
- Red annotations show the delegation paths: `list.forEach` points to the `ArrayList.forEach` method, and `e->System.out.println(e)` points to the `Consumer.accept` method.
- Red numbers (3) are placed near the delegation points to indicate the flow of control.





ArrayList list = new ArrayList();

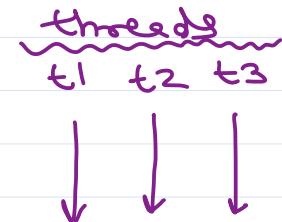
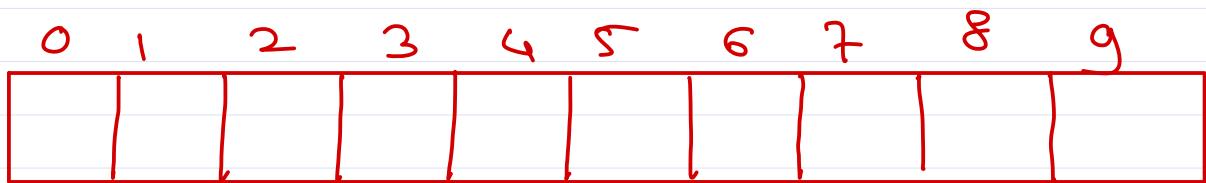
↳ initial capacity = 10

for (i=1; i<=20; i++)

list.add(i);

↳ if capacity is full, AL will grow

to new capacity = old capacity + $\frac{\text{old capacity}}{2}$



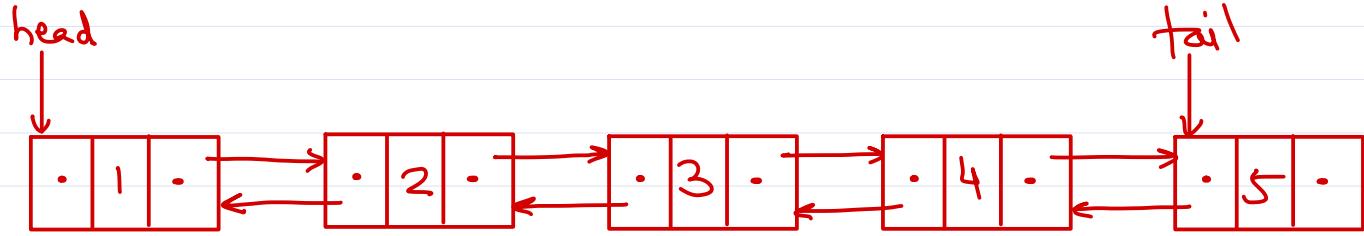
Vector `list = new vector();`
 \hookrightarrow initial capacity = 10

`for (i=1; i<=20; i++)`

`list.add(i);`

\hookrightarrow if capacity is full, AL will grow
 to new capacity = old capacity + addCapacity

vector is
 thread safe.
 (synchronized)



LinkedList list = new LinkedList();

```
for ( i=1; i<=5 ;i++)  
    list.add(i);
```



Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>



Core Java

Agenda

- Quick Revision
- Java collection framework
 - Introduction
 - Hierarchy
 - List interface and classes

Quick Revision

- Member classes
 - Static member class
 - Non-static member class
 - Local class
 - Anonymous inner class
- Java 8 Interfaces
 - default methods
 - static methods
 - Functional interface
 - Lambda expression
 - Method reference

Lambda expressions

Non-capturing lambda expression

- If lambda expression result entirely depends on the arguments passed to it, then it is non-capturing (self-contained).

```
BinaryOperator<Integer> op1 = (a,b) -> a + b;
```

- In functional programming, such functions/lambdas are referred as pure functions.

Capturing lambda expression

- If lambda expression result also depends on additional variables in the context of the lambda expression passed to it, then it is capturing.

```
int c = 2; // must be effectively final
BinaryOperator<Integer> op2 = (a,b) -> a + b + c;
```

- Here variable c is bound (captured) into lambda expression. So it can be accessed even out of scope. Internally it is associated with the method/expression.
- In some functional languages, this is known as Closures.

Method references

- If lambda expression involves single method call, it can be shortened by using method reference.
- Method references are converted into instances of functional interfaces.
- Method reference can be for class static method, class non-static method, object non-static method or constructor.

Java Collection Framework

Introduction

- Collection framework is Library of reusable data structure classes that is used to develop application.
- Main purpose of collection framework is to manage data/objects in RAM efficiently.
- Collection framework was introduced in Java 1.2 and type-safe implementation is provided in 5.0 (using generics).
- java.util package.
- Java collection framework provides

- Interfaces -- defines standard methods for the collections.
- Implementations -- classes that implements various data structures.
- Algorithms -- helper methods like searching, sorting, ...

Collection Hierarchy

- Interfaces: Iterable, Collection, List, Queue, Set, Map, Deque, SortedSet, SortedMap.
- Implementations: ArrayList, LinkedList, HashSet, HashMap, ...
- Algorithms: sort(), reverse(), max(), min(), ... -> in Collections class static methods

Iterable interface

- To traverse any collection it provides an Iterator.
- Enable use of for-each loop.
- In java.lang package
- Methods
 - Iterator iterator()
 - default Spliterator spliterator()
 - default void forEach(Consumer<? super T> action)

Collection interface

- Root interface in collection framework interface hierarchy.
- Most of collection classes are inherited from this interface (indirectly).
- Provides most basic/general functionality for any collection
- Abstract methods
 - 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()
 - Iterator iterator()
- Default methods
 - default Stream stream()
 - default Stream parallelStream()
 - default boolean removeIf(Predicate<? super E> filter)

List interface

- Ordered/sequential collection.
- Implementations: ArrayList, Vector, Stack, LinkedList, etc.
- List can contain duplicate elements.
- List can contain multiple null elements.
- Elements can be accessed sequentially (bi-directional using Iterator) or randomly (index based).
- Abstract methods
 - void add(int index, E element)
 - boolean addAll(int index, Collection<? extends E> c)
 - E get(int index)
 - E set(int index, E element)
 - int indexOf(Object o)
 - int lastIndexOf(Object o)
 - ListIterator listIterator()
 - ListIterator listIterator(int index)
 - E remove(int index)
 - List subList(int fromIndex, int toIndex)

- To store objects of user-defined types in the list, you must override equals() method for the objects. It is mandatory while searching operations like contains(), indexOf(), lastIndexOf().

ArrayList class

- Internally ArrayList is dynamically growable array.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Default initial capacity of ArrayList is 10. If it gets filled then its capacity gets increased by half of its existing capacity.
- Primary use
 - Random access is very fast
 - Add/remove at the end of list
- Internals (for experts)
 - <https://www.javatpoint.com/internal-working-of-arraylist-in-java>

Vector class

- Legacy collection class (since Java 1.0), modified for collection framework (List interface).
- Internally Vector is dynamically growable array.
- Elements can be traversed using Enumeration, Iterator, ListIterator, or using index.
- Default initial capacity of vector is 10. If it gets filled then its capacity gets increased by its existing capacity.
- Synchronized collection -- Thread safe but slower performance
- Primary use
 - Random access (in multi-threaded applications)
 - Add/remove at the end of list (in multi-threaded applications)

NOTE:

- To perform multiple tasks concurrently within a single process, threads are used (thread based multi-tasking or multi-threading).
- When multiple threads are accessing same resource at the same time, the race condition may occur. Due to this undesirable/unexpected results will be produced.

* To avoid this, OS/JVM provides synchronization mechanism. It will provide thread-safe access to the resource (the other threads will be blocked).

Iterator vs Enumeration

- Enumeration
 - Since Java 1.0
 - Methods
 - boolean hasMoreElements()
 - E nextElement()
 - Example

```
Enumeration<E> e = v.elements();
while(e.hasMoreElements()) {
    E ele = e.nextElement();
    System.out.println(ele);
}
```

- Enumeration behaves similar to fail-safe iterator.
- Iterator
 - Part of collection framework (1.2)
 - Methods
 - boolean hasNext()
 - E next()
 - void remove()
 - Example

```
Iterator<E> e = v.iterator();
while(e.hasNext()) {
```

```
E ele = e.next();
System.out.println(ele);
}
```

- ListIterator
 - Part of collection framework (1.2)
 - Inherited from Iterator
 - Bi-directional access
 - Methods
 - boolean hasNext()
 - E next()
 - int nextIndex()
 - boolean hasPrevious()
 - E previous()
 - int previousIndex()
 - void remove()
 - void set(E e)
 - void add(E e)

Fail-fast vs Fail-safe Iterator

- If state of collection is modified (add/remove operation other than iterator methods) while traversing a collection using iterator and iterator methods fails (with ConcurrentModificationException), then iterator is said to be Fail-fast.
 - e.g. Iterators from ArrayList, LinkedList, Vector, ...
- If iterator allows to modify the underlying collection (add/remove operation other than iterator methods) while traversing a collection (NO ConcurrentModificationException), then iterator is said to be Fail-safe.
 - e.g. Iterators from CopyOnWriteArrayList, ...

Stack class

- Legacy collection class, inherited from Vector class.

- Methods
 - boolean empty()
 - E peek()
 - E pop()
 - E push(E item)
 - int search(Object o)
- Synchronized collection -- Thread safe but slower performance
- Use ArrayDeque<> for better performance.

LinkedList class

- Internally LinkedList is doubly linked list.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Primary use
 - Add/remove elements (anywhere)
- Inherited from List<>, Deque<>.

Synchronized vs Un同步ized collections

- Synchronized collections are thread-safe and sync checks cause slower execution.
- Legacy collections were synchronized.
 - Vector
 - Stack
 - Hashtable
 - Properties
- Collection classes in collection framework are non-synchronized (for better performance).
- Collection classes can be converted to synchronized collection using Collections class methods.
 - syncList = Collections.synchronizedList(list)
 - syncSet = Collections.synchronizedSet(set)
 - syncMap = Collections.synchronizedMap(map)

Assignments

1. Store book details in a library in a list -- ArrayList.
 - Book details: isbn(string), price(double), authorName(string), quantity(int)
 - Write a menu driven program to
 1. Add new book in list
 2. Display all books in forward order
 3. Display all books in reverse order
 4. Search a book with given isbn (hint - indexOf())
 5. Delete a book at given index.
 6. Sort all books by price in desc order
 7. Replace book at given index with a new book (input from user)
 8. Remove all books with price < 200. (hint - removeIf())
2. Create an array of Strings. Reverse it using a stack. Hint --
 - step 1: push all elements of the array on stack (one by one).
 - step 2: pop elements from stack one by one and restore in array.
3. Create a list of strings. Find the string with highest length. Hint -- Collections.max()



Core Java

Trainer: Nilesh Ghule



Sunbeam Infotech

www.sunbeaminfo.com

temp
object to find

ArrayList - indexOf(obj) {

for(i=0 ; i<size ; i++) {

if(obj == arr[i])

if(arr[i].equals(obj))

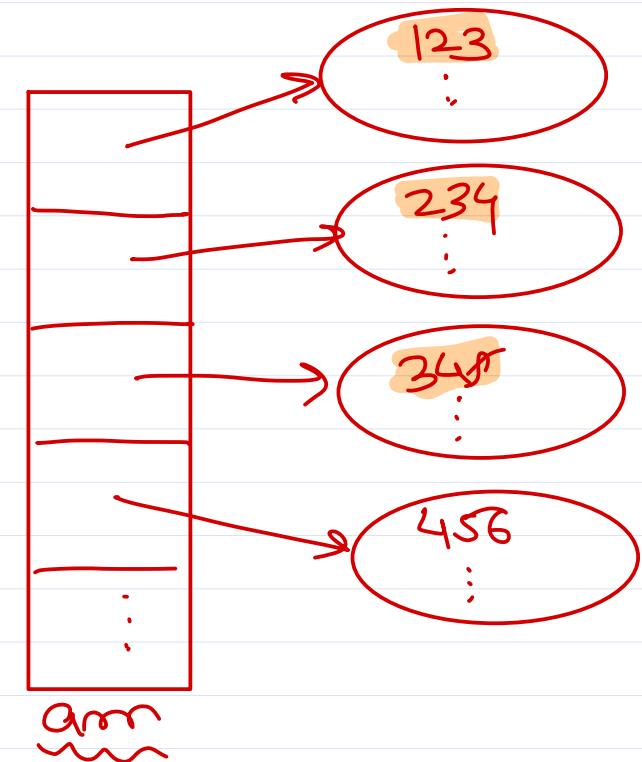
return i;

}

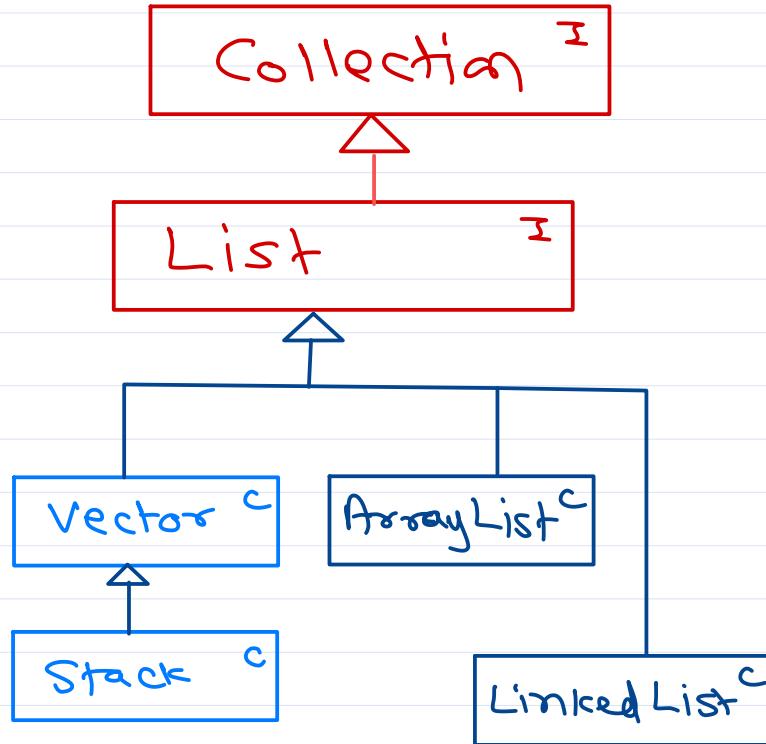
return -1;

}

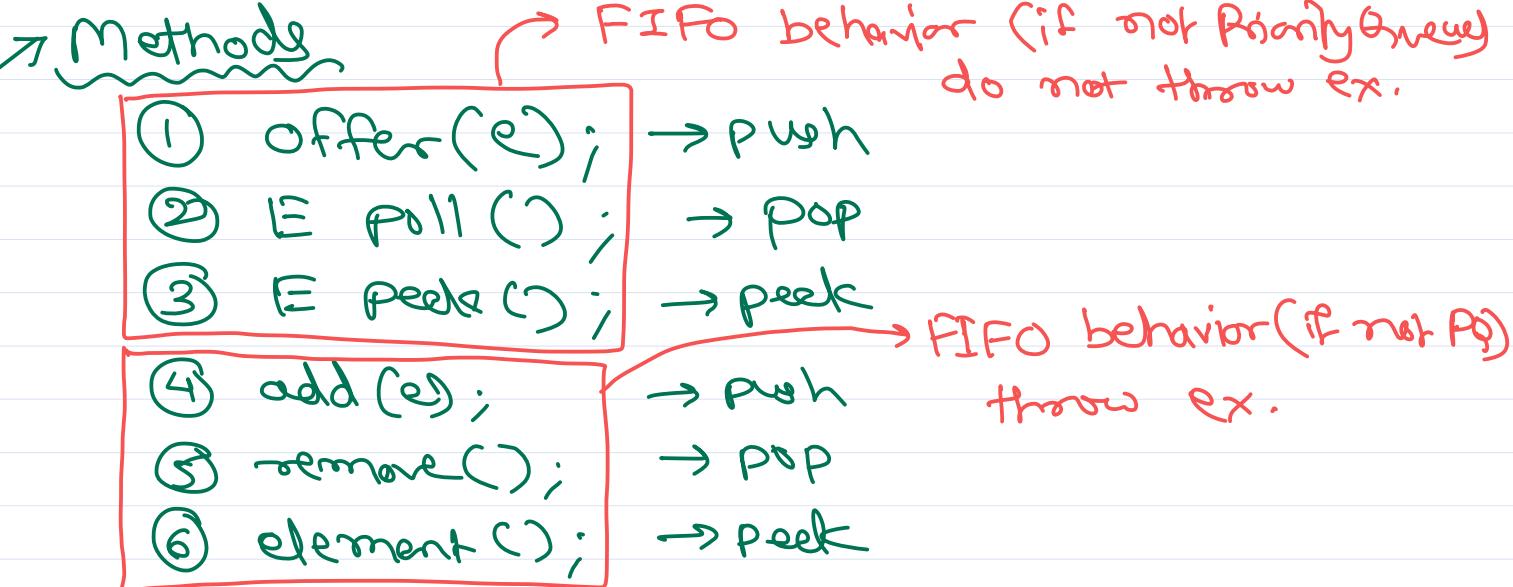
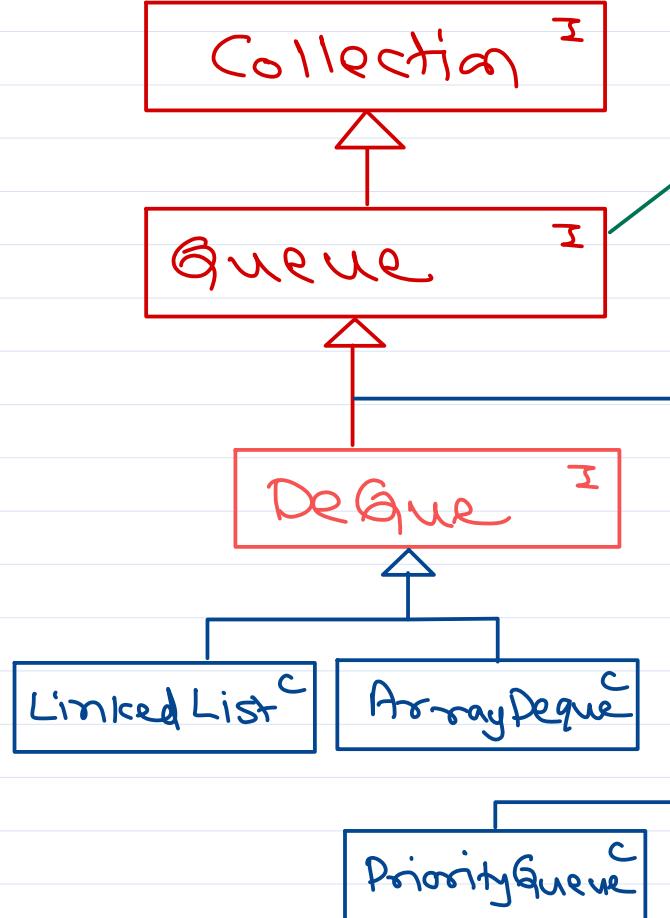
ArrayList
list



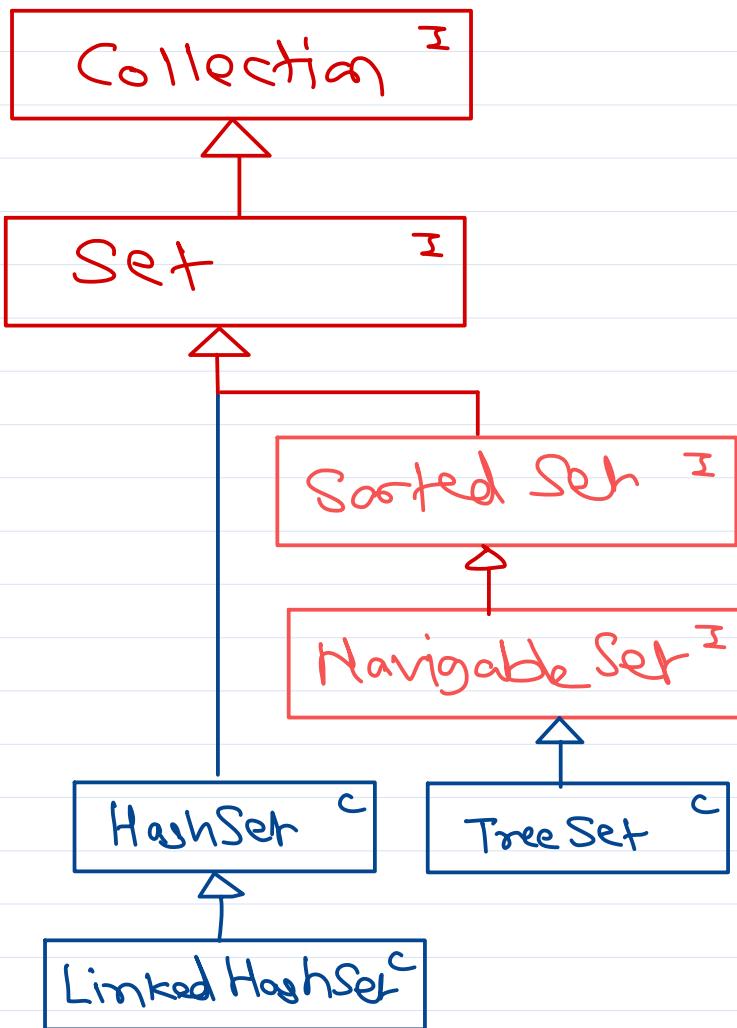
List hierarchy



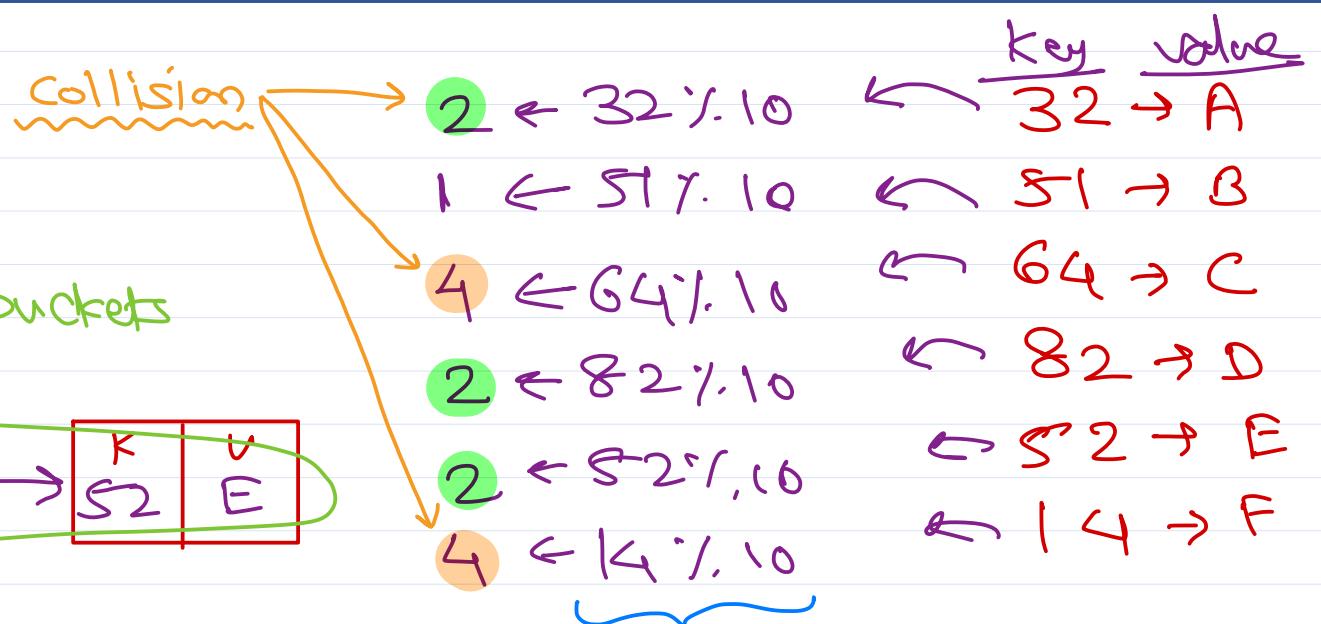
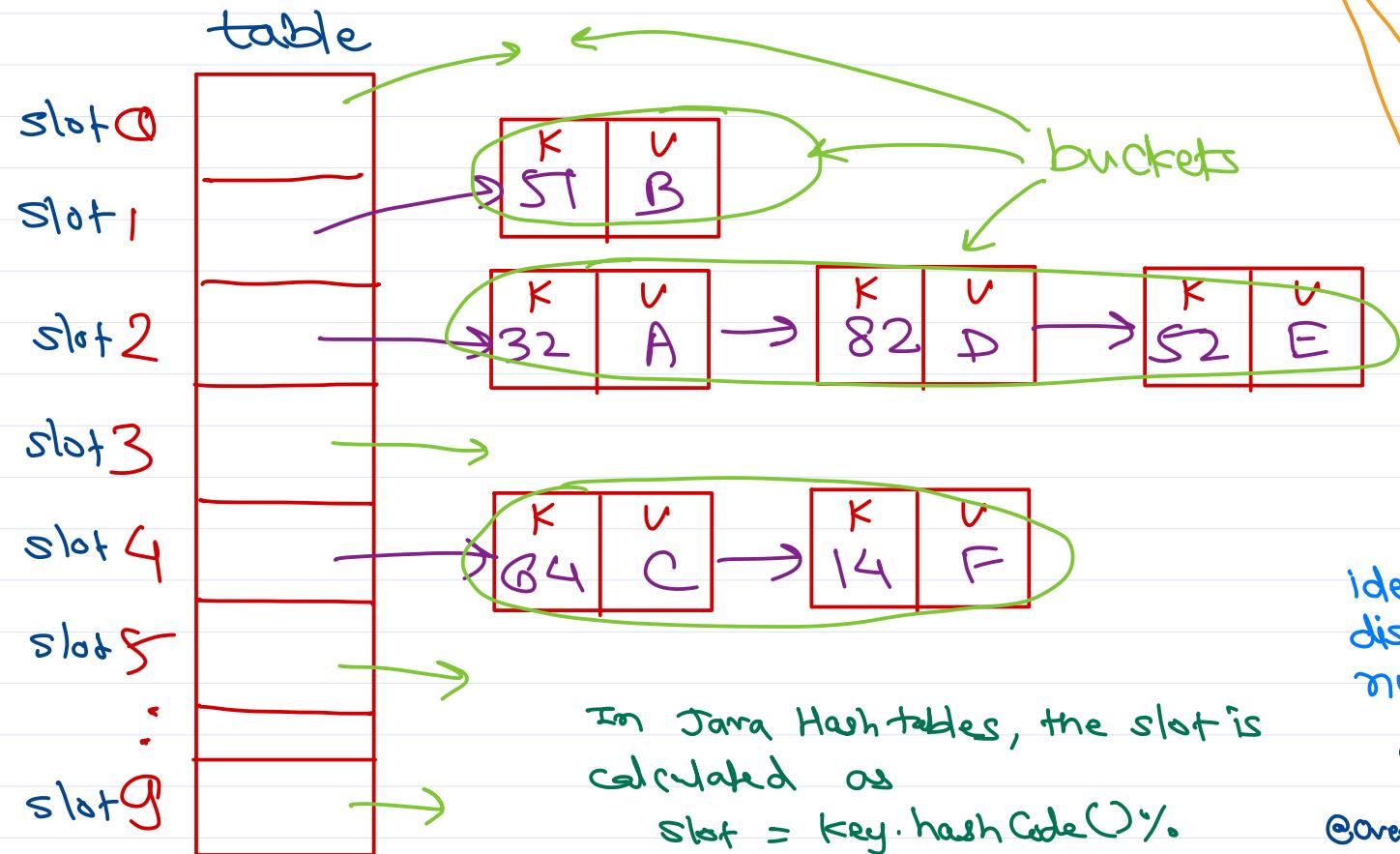
Queue hierarchy



Set hierarchy



load factor = $\frac{\text{num of entries}}{\text{num of buckets}}$



hash function.

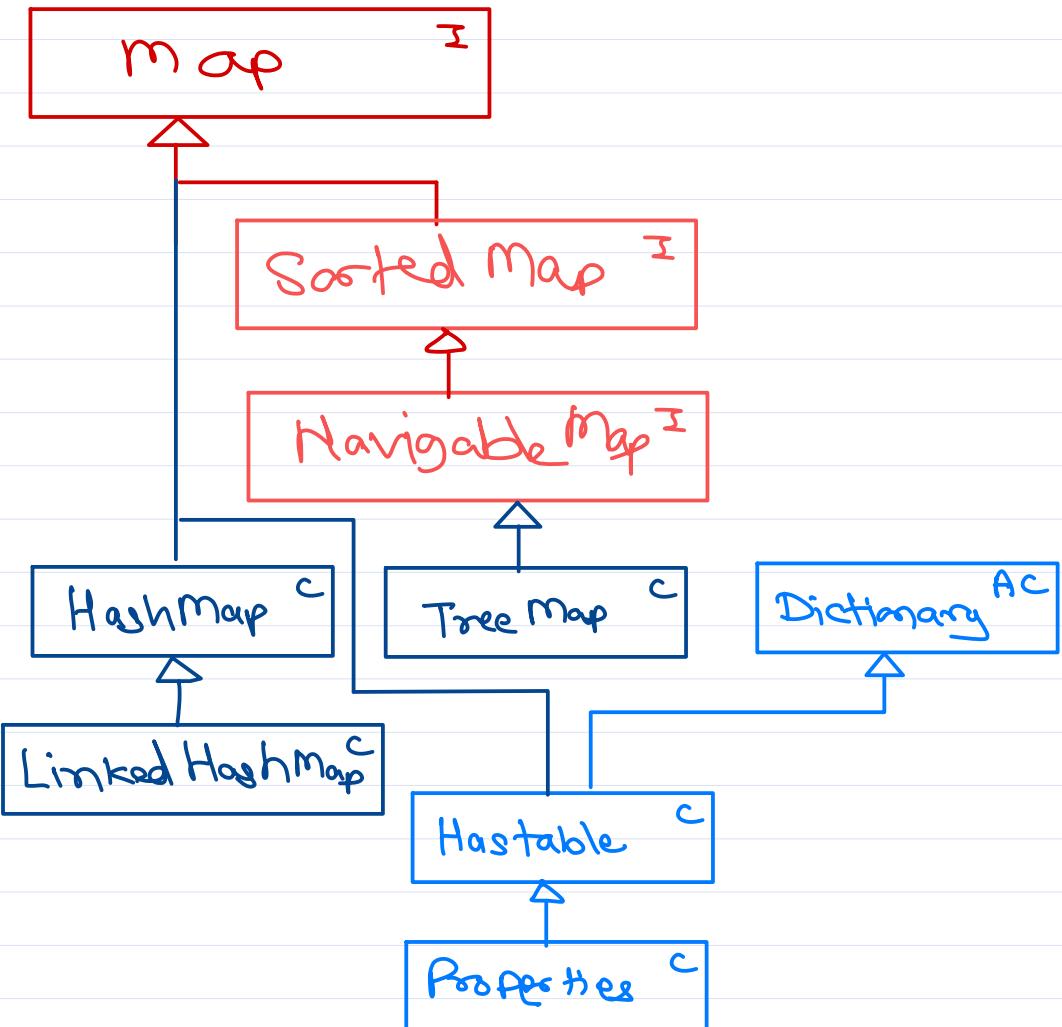
ideal hash fn produce diff num for each distinct key. Mathematically multiplication with prime num yields uniform distribution,

$$\text{class Distance} \{ h(f, i) = f + i * 31$$

```
int feet,inches;
@Override int hashCode() {
    int r=f;
    r+=i*31;
    return r;
}
```



Map hierarchy



```
Collection<String> c = new ArrayList<>();  
c.add("A");  
c.add("X");  
c.add("P");  
c.add(null);  
c.add(null);  
c.add(null);  
System.out(c);
```

try if nulls are
allowed in collections





Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

Core Java

Agenda

- Quick Revision
- Java collection framework
 - Queue interface and classes
 - Set interface and classes
 - Map interface and classes
 - Collections class
- Java 8 Streams
 - Functional programming
 - Stream operations

Quick Revision

- Java collection framework Hierarchy
- List interface and classes
- Fail-fast vs Fail-safe iterator
- indexOf() internals -- Demo14_01

Day13 Assignment

```
list.sort((b1,b2) -> {
    if(b1.getPrice() == b2.getPrice())
        return 0;
    if(b1.getPrice() > b2.getPrice())
        return +1;
    return -1;
});
```

OR

```
list.sort((b1,b2) -> (int)Math.signum(b1.getPrice() - b2.getPrice()));
```

Java collection framework

Collections class

- Helper/utility class that provides several static helper methods
- Methods
 - List reverse(List list);
 - List shuffle(List list);
 - E max(Collection list, Comparator cmp);
 - E min(Collection list, Comparator cmp);
 - List synchronizedList(List list);

Queue interface

- Represents queue data structure.
- Implementations: LinkedList, ArrayDeque, PriorityQueue.
- Can be accessed using iterator, but no random access.
- Methods
 - boolean add(E e) - throw IllegalStateException if full.
 - E remove() - throw NoSuchElementException if empty
 - E element() - throw NoSuchElementException if empty
 - boolean offer(E e) - return false if full.
 - E poll() - returns null if empty
 - E peek() - returns null if empty

Deque interface

- Represents double ended queue data structure i.e. add/delete can be done from both the ends.
- Can used as Queue as well as Stack.
- Methods
 - boolean offerFirst(E e)
 - E pollFirst()
 - E peekFirst()
 - boolean offerLast(E e)
 - E pollLast()
 - E peekLast()

ArrayDeque class

- Internally ArrayDeque is dynamically growable array.

LinkedList class

- Internally LinkedList is doubly linked list.

PriorityQueue class

- Internally PriorityQueue is a binary heap data structures.
- Elements with highest priority is deleted first (NOT FIFO).
- Elements should have natural ordering or need to provide comparator.

Set interface

- Collection of unique elements (NO duplicates allowed).
- Implementations: HashSet, LinkedHashSet, TreeSet.
- Elements can be accessed using an Iterator.
- Abstract methods (same as Collection interface)

- add() returns false if element is duplicate

HashSet class

- Non-ordered set (elements stored in any order)
- Elements must implement equals() and hashCode()
- Fast execution

LinkedHashSet class

- Ordered set (preserves order of insertion)
- Elements must implement equals() and hashCode()
- Slower than HashSet

SortedSet interface

- Use natural ordering or Comparator to keep elements in sorted order
- Methods
 - E first()
 - E last()
 - SortedSet headSet(E toElement)
 - SortedSet subSet(E fromElement, E toElement)
 - SortedSet tailSet(E fromElement)

NavigableSet interface

- Sorted set with additional methods for navigation
- Methods
 - E higher(E e)
 - E lower(E e)
 - E pollFirst()
 - E pollLast()

- NavigableSet descendingSet()
- Iterator descendingIterator()

TreeSet class

- Sorted navigable set (stores elements in sorted order)
- Elements must implement Comparable or provide Comparator
- Slower than HashSet and LinkedHashSet
- It is recommended to have consistent implementation for Comparable (Natural ordering) and equals() method i.e. equality and comparison should done on same fields.
- If need to sort on other fields, use Comparator.

```
class Book implements Comparable<Book> {  
    private String isbn;  
    private String name;  
    // ...  
    public int hashCode() {  
        return isbn.hashCode();  
    }  
    public boolean equals(Object obj) {  
        if(!(obj instanceof Book))  
            return false;  
        Book other=(Book)obj;  
        if(this.isbn.equals(other.isbn))  
            return true;  
        return false;  
    }  
    public int compareTo(Book other) {  
        return this.isbn.compareTo(other.isbn);  
    }  
}
```

```
// Store in sorted order by name  
set = new TreeSet<Book>((b1,b2) -> b1.getName().compareTo(b2.getName()));
```

```
// Store in sorted order by isbn  
set = new TreeSet<Book>();
```

HashTable Data structure

- Hashtable stores data in key-value pairs so that for the given key, value can be searched in fastest possible time.
- Internally hash-table is a table(array), in which each slot(index) has a bucket(collection). Key-value entries are stored in the buckets depending on hash code of the key.
- Load factor = Number of entries / Number of buckets.
- Examples
 - Key=pincode, Value=city/area
 - Key=Employee, Value=Manager
 - Key=Department, Value=list of Employees

hashCode() method

- Object class has hashCode() method, that returns a unique number for each object (by converting its address into a number).
- To use any hash-based data structure hashCode() and equals() method must be implemented.
- If two distinct objects yield same hashCode(), it is referred as collision. More collisions reduce performance.
- Most common technique is to multiply field values with prime numbers to get uniform distribution and lesser collisions.
- hashCode() overriding rules
 - hash code should be calculated on the fields that decides equality of the object.
 - hashCode() should return same hash code each time unless object state is modified.
 - If two objects are equal (by equals()), then their hash code must be same.
 - If two objects are not equal (by equals()), then their hash code may be same (but reduce performance).

Map interface

- Collection of key-value entries (Duplicate keys not allowed).
- Implementations: HashMap, LinkedHashMap, TreeMap.
- The data can be accessed as set of keys, collection of values, and/or set of key-value entries.
- Map.Entry<K,V> is nested interface of Map<K,V>.
 - K getKey()
 - V getValue()
 - V setValue(V value)
- Abstract methods
 - 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()
- Maps not considered as true collection, because it is not inherited from Collection interface.

HashMap class

- Non-ordered map (entries stored in any order -- as per hash code of key)
- Keys must implement equals() and hashCode()
- Fast execution
- Mostly used Map implementation

LinkedHashMap class

- Ordered map (preserves order of insertion)
- Keys must implement equals() and hashCode()
- Slower than HashSet
- Since Java 1.4

TreeMap class

- Sorted navigable map (stores entries in sorted order of key)
- Keys must implement Comparable or provide Comparator
- Slower than HashMap and LinkedHashMap
- Internally based on Red-Black tree.
- Doesn't allow null key (allows null value though).

Hashtable class

- Similar to HashMap class.
- Legacy collection class (since Java 1.0), modified for collection framework (Map interface).
- Synchronized collection -- Thread safe but slower performance
- Inherited from java.util.Dictionary abstract class (it is Obsolete).

Java 8 Streams

Functional Programming

- Functions are First class Values (as equal as int or String).
 - Functions like other languages.
 - Functions as variables/objects.
 - Functions as argument or return value of function.
 - Functions defined in other function.
 - Functions as anonymous.

- Output of functions are mapped to their inputs.
 - Immutability – produce new result (instead of changing in place)
 - Function result depends solely on the input and also doesn't modify state of args or other objects.
 - Functions should be referentially transparent i.e. it should be easily replaceable by its result without changing program semantic.
 - Such functions are said to be side-effect free functions or pure functions.
- Why functional programming?
 - Developer's productivity: better reusability and modularity.
 - Simplified testing: functions without side effects.
 - Scalability: easily portable to multi-core and distributed computing (no sync. issues)
- Limitations of functional programming
 - Difficult to understand: new and complex programming paradigm for beginners.
 - Tricky: pure functions are not obvious in complex programs and tricky reusability.
 - Resource hungry: Needs more memory to store state and also more computing.
- Scope for functional programming
 - CPU and memory resources are cheaper (cloud computing).
 - Multi-core and distributed programming is common (huge data processing).
 - Popular languages are functional or support functional.
- Java functional programming is blended into object oriented programming.

Introduction

- Java 8 Stream is NOT IO streams.
- Java streams represents pipeline of operations through which data is processed.
- Stream operations are of two types
 - Intermediate operations: Yields another stream.
 - filter()
 - map(), flatMap()
 - limit(), skip()
 - sorted(), distinct()
 - Terminal operations: Yields some result.
 - reduce()

- forEach()
- collect(), toArray()
- count(), max(), min()
- Stream operations are higher order functions (take functional interfaces as arg).

Java stream characteristics

- No storage: Stream is an abstraction. Stream doesn't store the data elements. They are stored in source collection or produced at runtime.
- Immutable: Any operation doesn't change the stream itself. The operations produce new stream holding the results.
- Lazy evaluation: Streams are evaluated only if they have terminal operations. If terminal operation is not given, stream is not processed.
- Not reusable: Streams processed once (terminal operation) cannot be processed again.

Stream creation

- Collection interface: stream() or parallelStream()
- Arrays class: Arrays.stream()
- Stream interface: static of() method
- Stream interface: static generate() method
- Stream interface: static iterate() method
- Stream interface: static empty() method
- nio Files class: static Stream lines(filePath) method

Assignment

1. Store few books (hardcoded values with yesterday's Book class) in a HashSet and display them using forEach() method. If any book with duplicate isbn is added, what will happen? Books are stored in which order?
2. In above assignment use LinkedHashSet instead of HashSet. If any book with duplicate isbn is added, what will happen? Books are stored in which order?
3. In above assignment use TreeSet instead of LinkedHashSet. Use natural ordering for the Book. If any book with duplicate isbn is added, what will happen? Books are stored in which order?
4. Use TreeSet to store all books in descending order of price. Natural ordering for the Book should be isbn (do not change it). Display them using forEach().

5. Store Books in HashMap<> so that for given isbn, book can be searched in fastest possible time. Do we need to write equals() and hashCode() in Book class?

Hint:

```
// declare map  
Map<String,Book> map = new HashMap<>();  
// insert in map  
Book b = new Book(...);  
map.put(b.getIsbn(), b);  
// find in map  
String isbn = sc.next();  
Book f = map.get(isbn);
```

6. In which collection classes null is not allowed? Duplicate null is not allowed? Multiple nulls are allowed?

7. Create a stream of random numbers as follows. Do the operations given below.

```
Stream<Double> strm = Stream  
    .generate(Stream.generate(() -> Math.random()))  
    .limit(25);
```

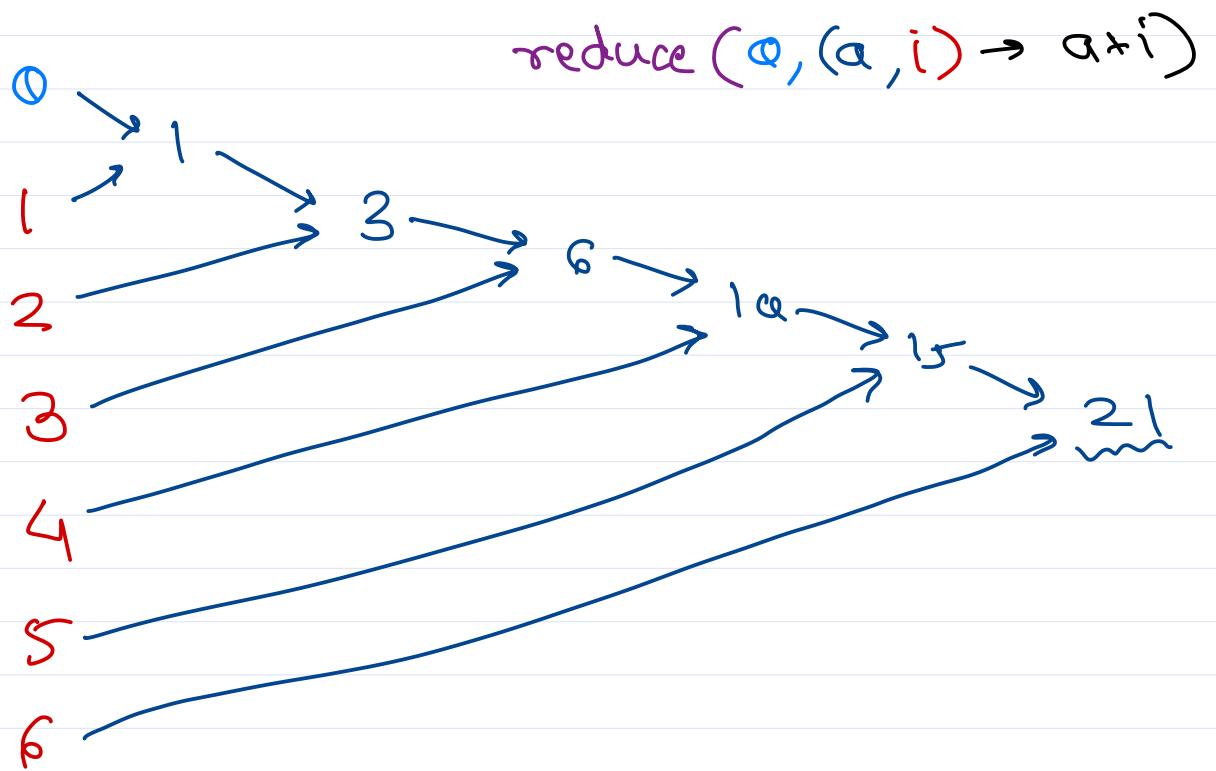
- stream1 - Display all numbers in stream.
- stream2 - Convert stream into stream of ints in range of 1 to 100 and then Display them.
- stream3 - Convert stream into stream of ints in range of 1 to 100, then get all odd numbers, finally find the minimum number using min() action.
- stream4 - Convert stream into stream of ints in range of 1 to 100, then display 5 smallest numbers from it.

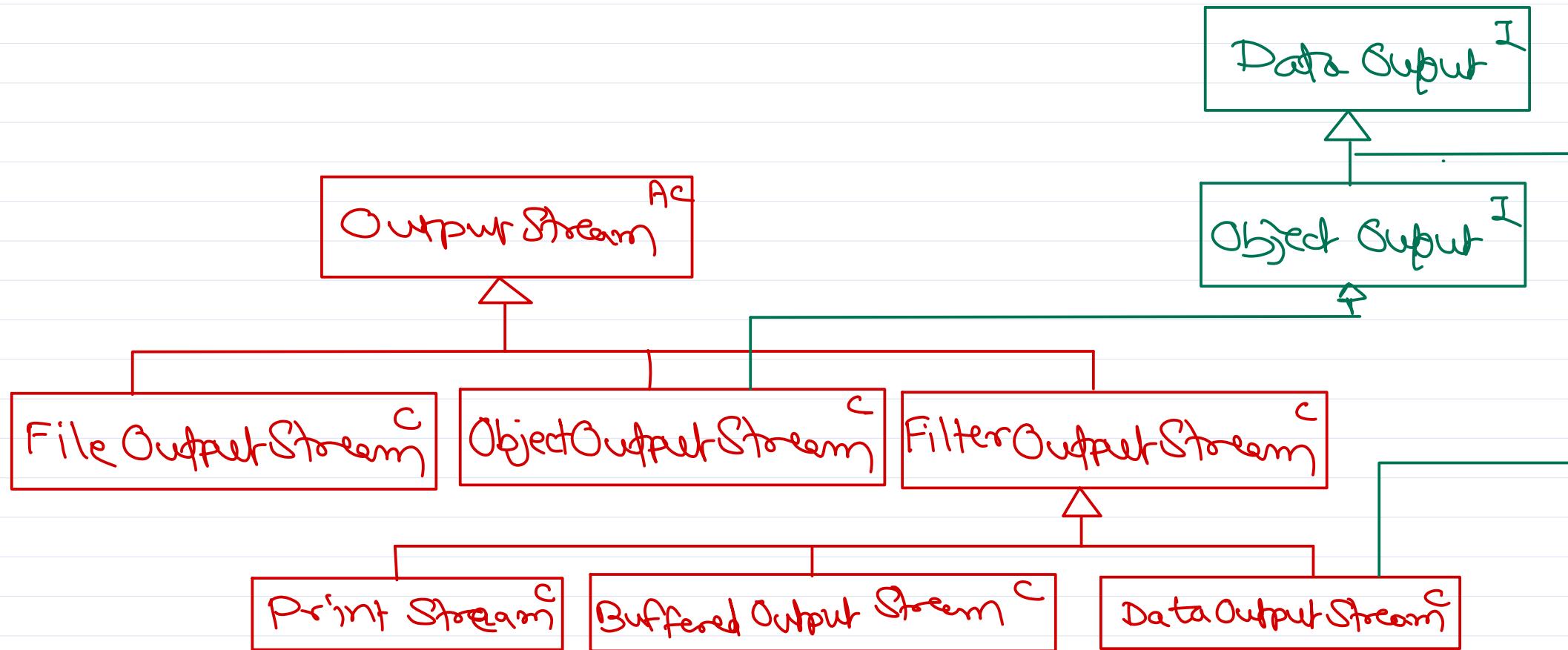


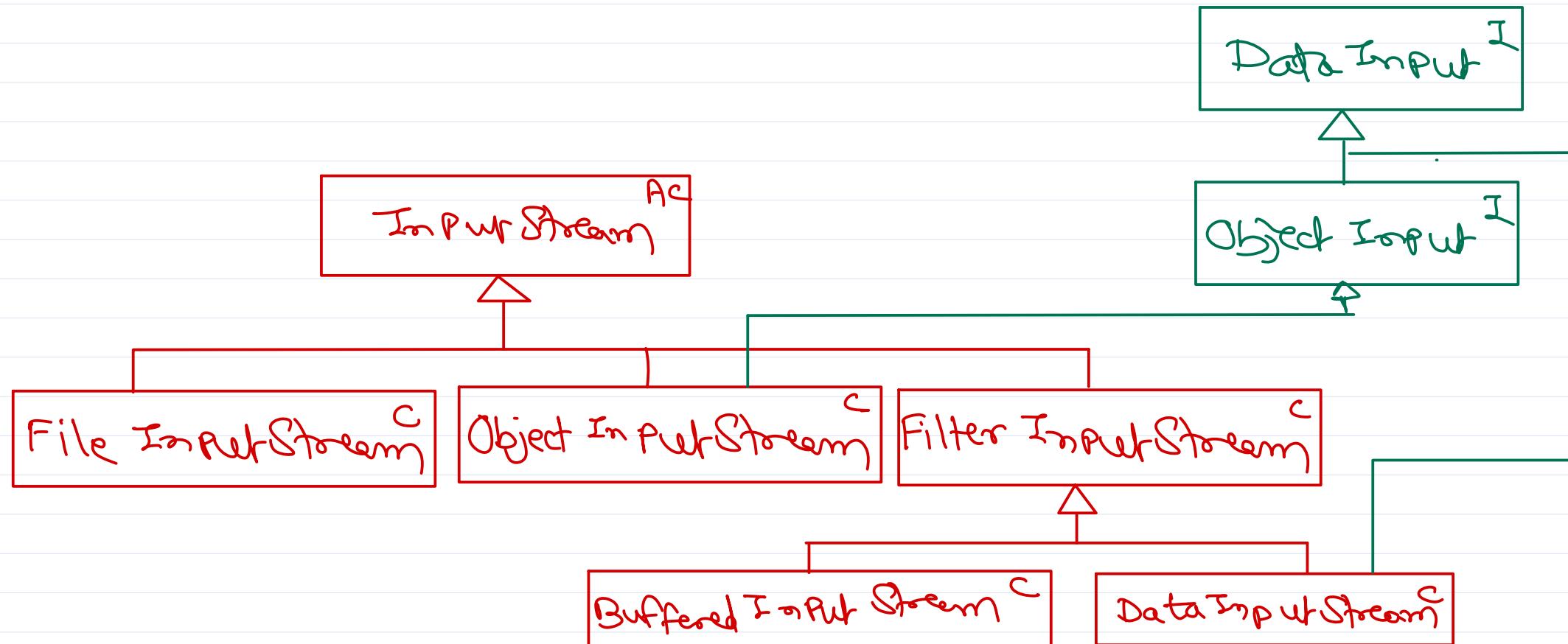
Core Java

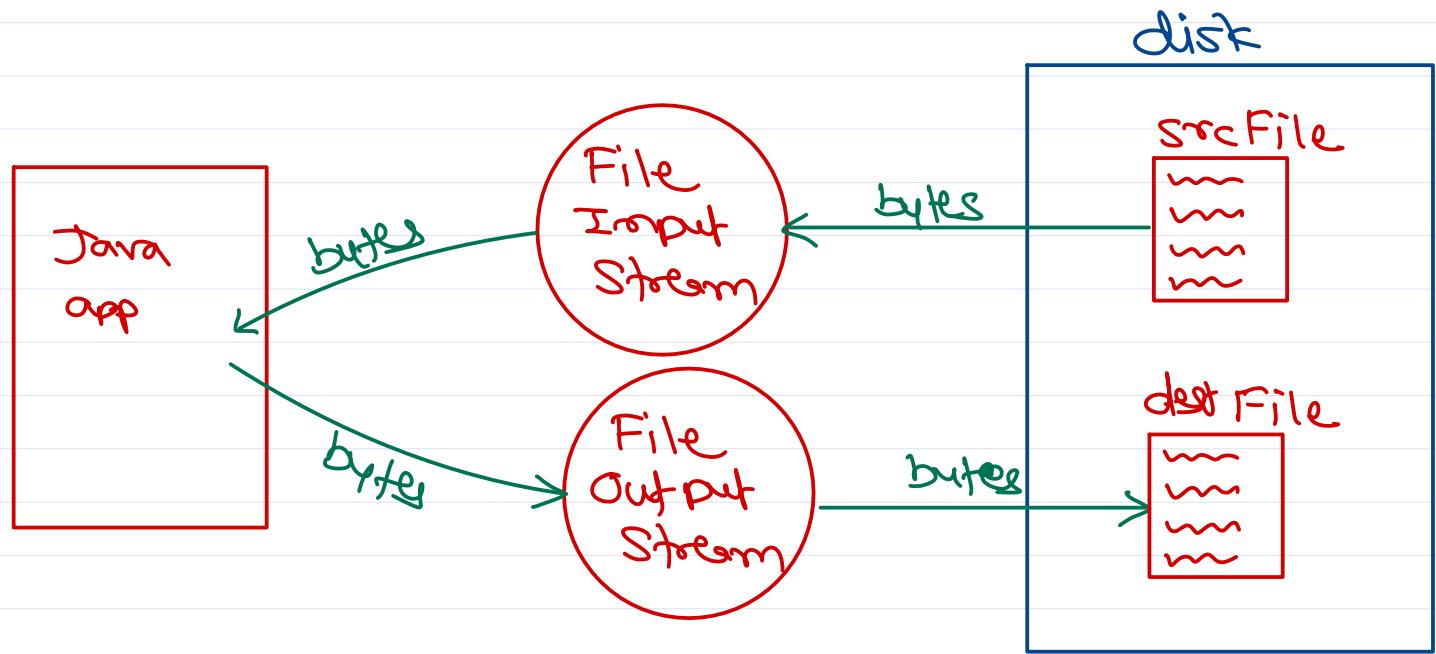
Trainer: Nilesh Ghule







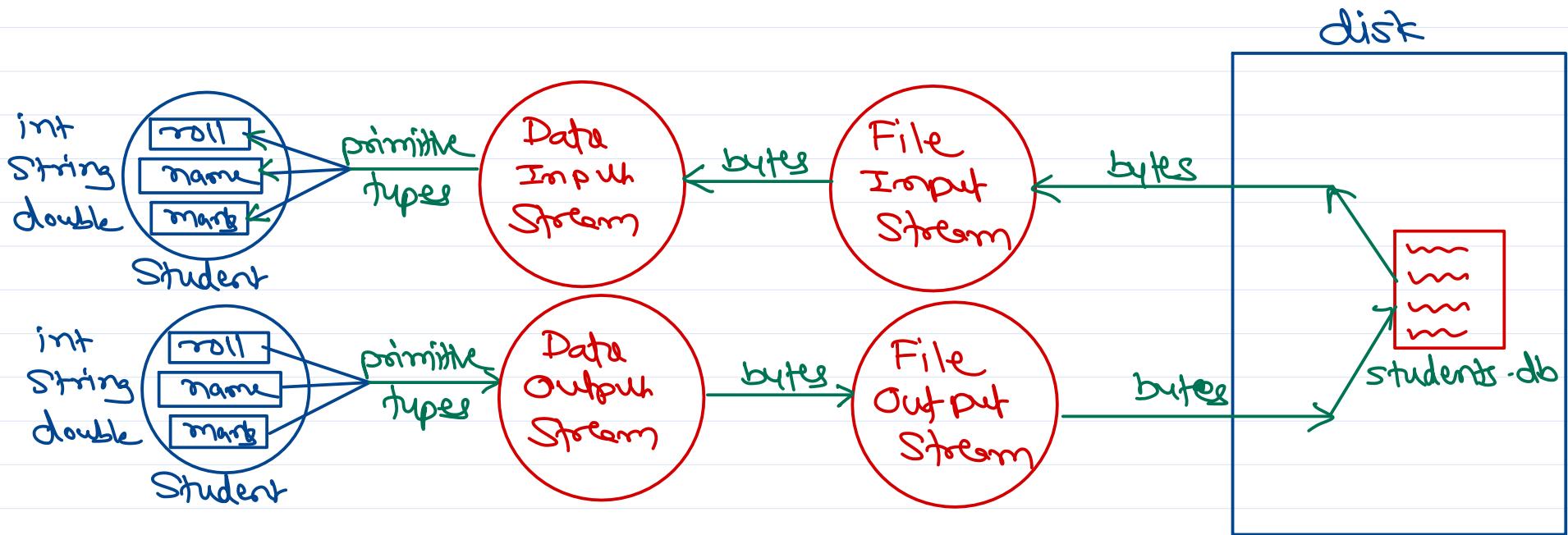




```

int b;
InputStream in=new FileInputStream("src.txt");
OutputStream out=new FileOutputStream("dest.txt");
while((b=in.read()) != -1)
{
    out.write(b);
}
out.close();
in.close();

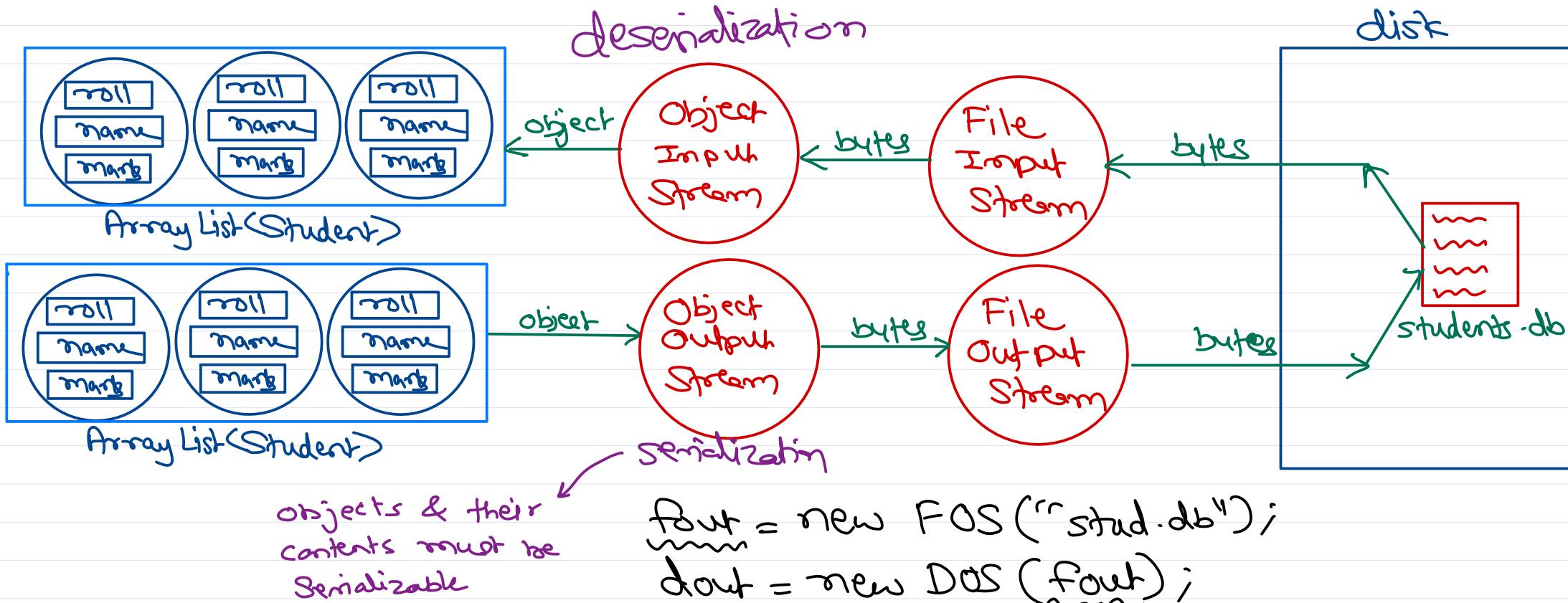
```



```

fout = new FOS("stud.db");
dout = new DOS(fout);

```





Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>



Core Java

Agenda

- Quick Revision
- Java 8 Streams
- Java IO framework

Quick Revision

- Java collection framework
 - Queue interface and classes
 - Set interface and classes
 - Map interface and classes
 - Collections class
- Java 8 Streams
 - Functional programming

Java 8 Streams

- Streams follow functional programming model in Java 8.
- The functional programming is based on functional interface (SAM).
- Number of predefined functional interfaces added in Java 8. e.g. Consumer, Supplier, Function, Predicate, ...
- Lambda expression is short-hand way of implementing SAM -- arg types & return type are inferred.

Stream creation

- Collection interface: stream() or parallelStream()

```
List<String> list = new ArrayList<>();  
// ...  
Stream<String> strm = list.stream();
```

- Arrays class: Arrays.stream()
- Stream interface: static of() method
- Stream interface: static generate() method

```
Stream<Double> strm = Stream.generate(() -> Math.random()).limit(25);
```

```
Random r = new Random();  
Stream<Integer> strm = Stream.generate(() -> r.nextInt(1000)).limit(10);
```

- Stream interface: static iterate() method

```
Stream<Integer> strm = Stream.iterate(1, i -> i + 1).limit(10);
```

- Stream interface: static empty() method
- nio Files class: static Stream lines(filePath) method

Stream operations

- Create Stream

```
String[] names = {"Smita", "Rahul", "Rachana", "Amit", "Shraddha", "Nilesh", "Rohan", "Pradnya", "Rohan", "Sarika",  
"Prerana"};  
Stream strm = Stream.of(names);
```

- filter() -- Get all names ending with "a"

- Predicate: (T) -> boolean

```
strm.filter(s -> s.endsWith("a"))  
    .forEach(s -> System.out.println(s));
```

- map() -- Convert all names into upper case

- Function<T,R>: (T) -> R

```
strm.map(s -> s.toUpperCase())  
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in ascending order

- String class natural ordering is ascending order.
 - sorted() is a stateful operation (i.e. needs all element to sort).

```
strm.sorted()  
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in descending order

- Comparator: $(T,T) \rightarrow \text{int}$

```
strm.sorted((s1,s2) -> s2.compareTo(s1))
    .forEach(s -> System.out.println(s));
```

- skip() & limit() -- leave first 2 names and print next 4 names

```
strm
    .skip(2)
    .limit(4)
    .forEach(s -> System.out.println(s));
```

- distinct() -- remove duplicate names

- duplicates are removed according to equals().

```
strm.distinct()
    .forEach(s -> System.out.println(s));
```

- count() -- count number of names

- terminal operation: returns long.

```
cnt = strm.count();
```

- max() -- find the max string

- terminal operation

```
Optional<String> res = strm
    .max((m,i) -> m.compareTo(i));
if(res.isPresent())
    System.out.println(res.get());
else
    System.out.println("No max found");
```

- reduce() -- concat all the strings with length <= 5

- terminal operation:

```
String res = strm // "Smita", "Rahul", "Rachana", "Amit", "Shraddha", "Nilesh", "Rohan", "Pradnya", "Rohan",
"Sarika", "Prerana"
    .filter(s -> s.length() <= 5)
    // "Smita", "Rahul", "Amit", "Rohan", "Rohan"
    .reduce("", (acc,str) -> acc + str);
    // (((("") + "Smita") + "Rahul") + "Amit") + "Rohan" + "Rohan"
```

```
Stream<Integer> strm = Stream.iterate(1, i -> i + 1).limit(6);
Integer result = strm.reduce(0, (a,i) -> a + i);
```

- collect() -- collects all stream elements into an collection (list, set, or map)

```
List<String> list = strm.collect(Collectors.toList());
```

Optional<> type

- Few stream operations yield Optional<> value.
- Optional value is a wrapper/box for object of T type or no value.
- It is safer way to deal with null values.
- Get Optional<> value:
 - optValue = opt.get();
 - optValue = opt.orElse(defValue);
- Consuming Optional<> value:
 - opt.isPresent();
 - opt.ifPresent(consumer);

Collect Stream result

- Collecting stream result is terminal operation.
- Object[] toArray()
- R collect(Collector)
 - Collectors.toList(), Collectors.toSet(), Collectors.toCollection(), Collectors.joining()
 - Collectors.toMap(key, value)

Stream of primitive types

- Efficient in terms of storage and processing. No auto-boxing and unboxing is done.
- IntStream class
 - IntStream.of() or IntStream.range() or IntStream.rangeClosed() or Random.ints()
 - sum(), min(), max(), average(), summaryStatistics(),
 - OptionallInt reduce().

Reference

- <https://winterbe.com/posts/2014/03/16/java-8-tutorial/>

Java IO framework

- Input/Output functionality in Java is provided under package `java.io` and `java.nio` package.
- IO framework is used for File IO, Network IO, Memory IO, and more.

Java File IO

- File is a collection of data and information on a storage device.
- File = Data + Metadata
- Two types of APIs are available file handling
 - `FileSystem API` -- Accessing/Manipulating Metadata
 - `File IO API` -- Accessing/Manipulating Contents/Data

java.io.File class

- A path (of file or directory) in file system is represented by "File" object.
- Provides `FileSystem` APIs
 - `String[] list()` -- return contents of the directory
 - `File[] listFiles()` -- return contents of the directory
 - `boolean exists()` -- check if given path exists
 - `boolean mkdir()` -- create directory
 - `boolean mkdirs()` -- create directories (child + parents)
 - `boolean createNewFile()` -- create empty file
 - `boolean delete()` -- delete file/directory
 - `boolean renameTo(File dest)` -- rename file/directory
 - `String getAbsolutePath()` -- returns full path (drive:/folder/folder/...)
 - `String getPath()` -- return path
 - `File getParentFile()` -- returns parent directory of the file
 - `String getParent()` -- returns parent directory path of the file
 - `String getName()` -- return name of the file/directory
 - `static File[] listRoots()` -- returns all drives in the systems.

- long getTotalSpace() -- returns total space of current drive
- long getFreeSpace() -- returns free space of current drive
- long getUsableSpace() -- returns usable space of current drive
- boolean isDirectory() -- return true if it is a directory
- boolean isFile() -- return true if it is a file
- boolean isHidden() -- return true if the file is hidden
- boolean canExecute()
- boolean canRead()
- boolean canWrite()
- boolean setExecutable(boolean executable) -- make the file executable
- boolean setReadable(boolean readable) -- make the file readable
- boolean setWritable(boolean writable) -- make the file writable
- long length() -- return size of the file in bytes
- long lastModified() -- last modified time
- boolean setLastModified(long time) -- change last modified time

Java File IO

- Java File IO is done with Java IO streams.
- Stream is abstraction of data source/sink.
- Java supports two types of IO streams
 - Byte streams (binary files) -- byte by byte read/write
 - Character streams (text files) -- char by char read/write
- All these streams are AutoCloseable (so can be used with try-with-resource construct)

Binary/Byte Streams

- Byte streams are represented by InputStream and OutputStream class.
- OutputStream class -- write operation
 - void close() -- close the stream
 - void flush() -- writes data (in memory) to underlying stream/device.

- void write(byte[] b) -- writes byte array to underlying stream/device.
- abstract void write(int b) -- writes a byte to underlying stream/device.
- OutputStream Sub-classes
 - FileOutputStream, ObjectOutputStream, DataOutputStream, PrintStream, BufferedOutputStream, etc.
- InputStream class -- read operation
 - void close() -- close the stream
 - int available() -- return number of bytes available in memory.
 - int read(byte[] b) --
 - abstract int read() -- reads a byte from the underlying device/stream. Returns -1
- InputStream Sub-classes
 - FileInputStream, ObjectInputStream, DataInputStream, BufferedInputStream, etc.

Chaining IO Streams

- Each IO stream object performs a specific task.
 - FileOutputStream -- Write the given bytes into the file (on disk).
 - BufferedOutputStream -- Hold multiple elements in a temporary buffer before flushing it to underlying stream/device. Improves performance.
 - DataOutputStream -- Convert primitive types into sequence of bytes. Inherited from DataOutput interface.
 - ObjectOutputStream -- Convert object into sequence of bytes. Inherited from ObjectOutputStream interface.
 - PrintStream -- Convert given input into formatted output.
 - Note that input streams does the counterpart of OutputStream class hierarchy.
- Streams can be chained to fulfil application requirements.

Assignment

- Write a menu driven program to store books, display books, sort books (as in previous assignment). Provide two menus for storing it into file and loading it back from the menu.
- Calculate the factorial of the given number using stream operations.
- Create a program to calculate sum of integers.
- Create an IntStream to represent numbers from 1 to 10. Call various functions like sum(), summaryStatistics() and observe the output.



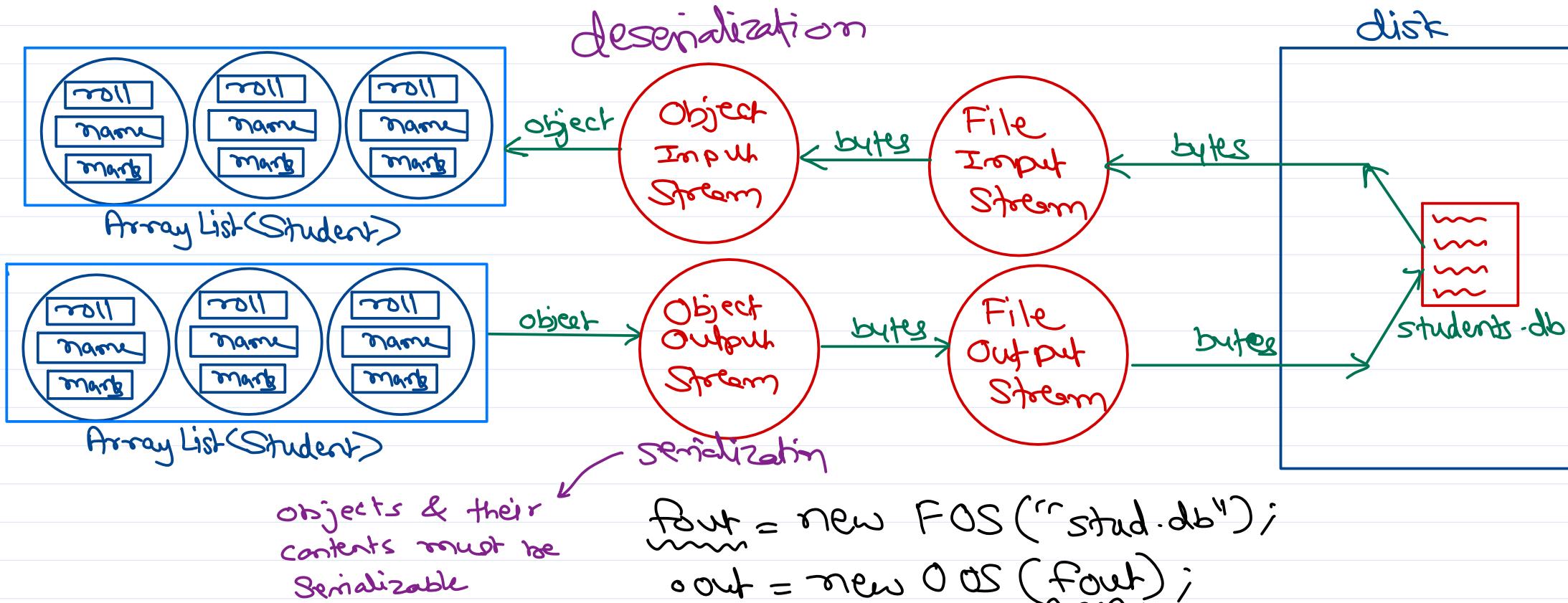
Core Java

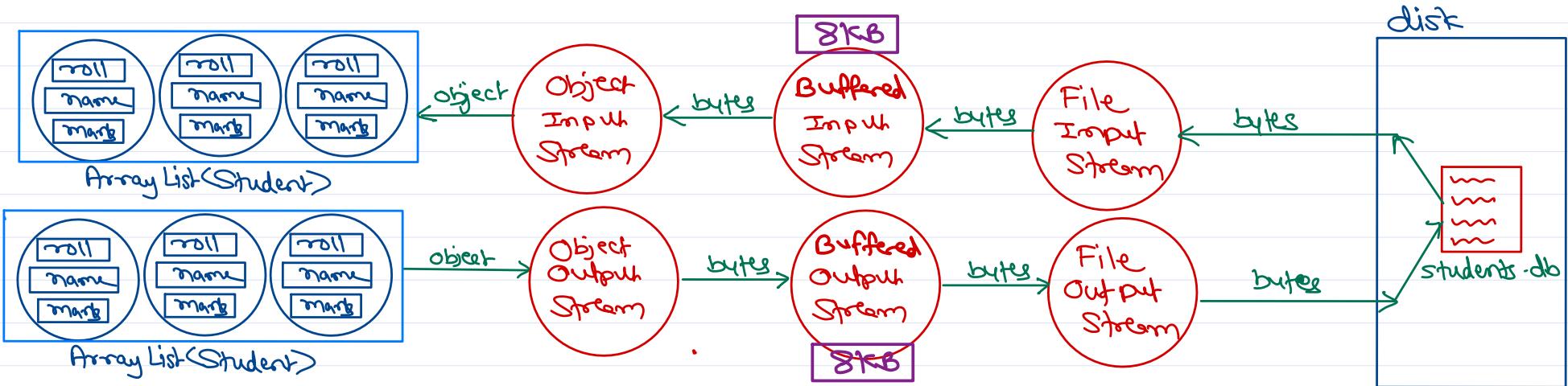
Trainer: Nilesh Ghule



Sunbeam Infotech

www.sunbeaminfo.com

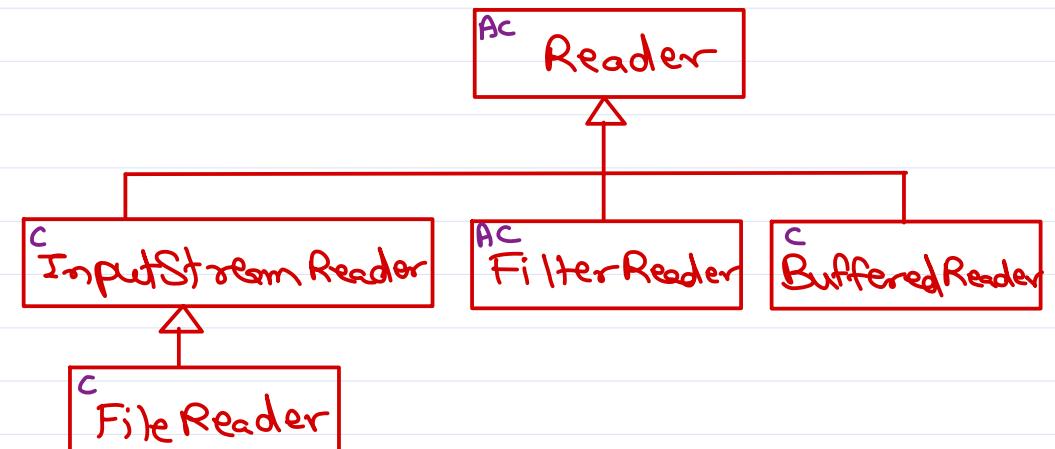
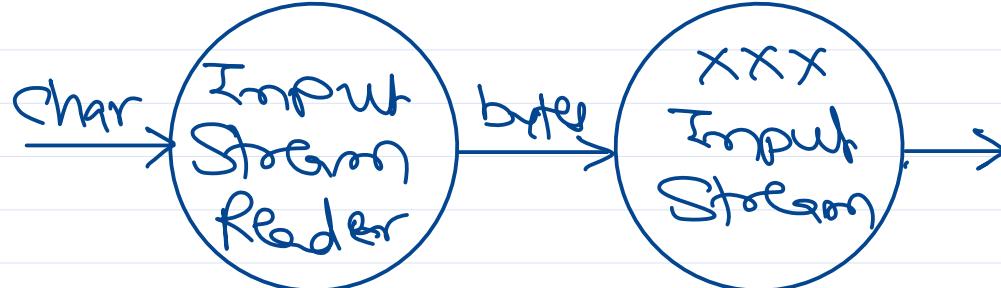
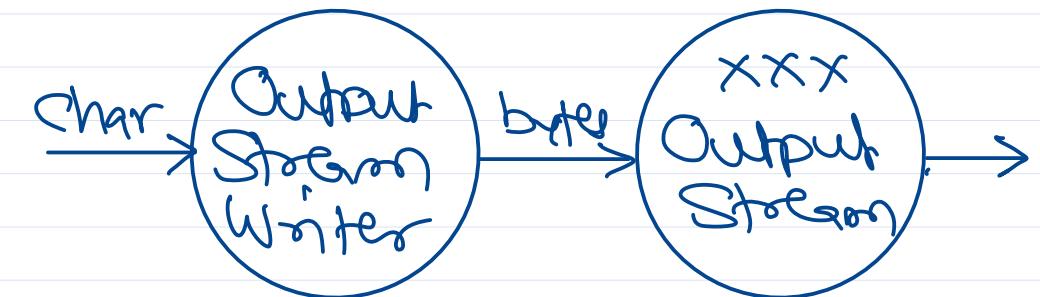
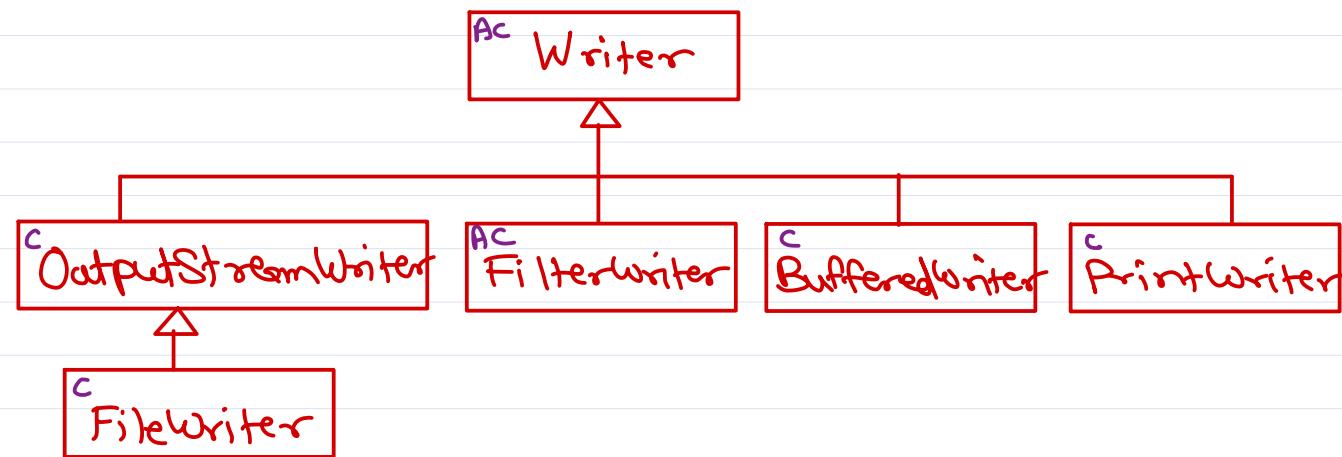


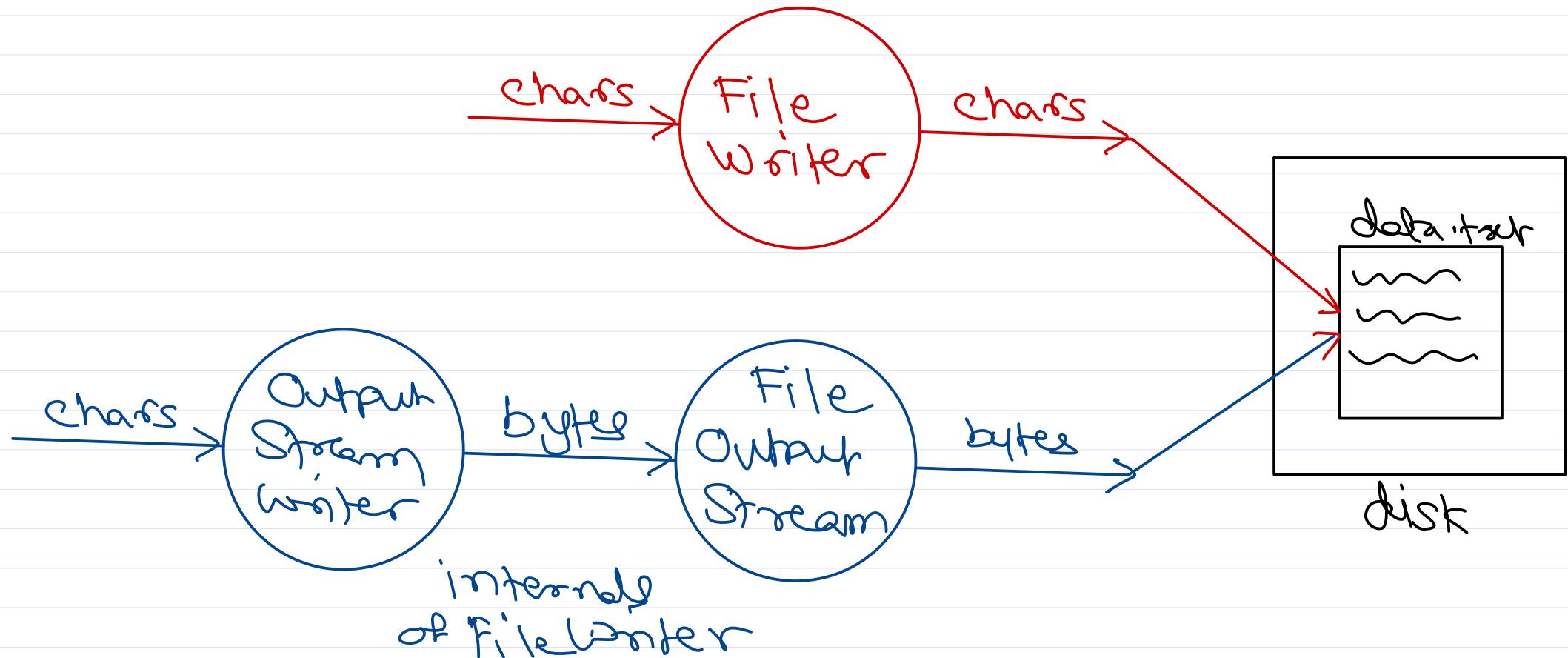


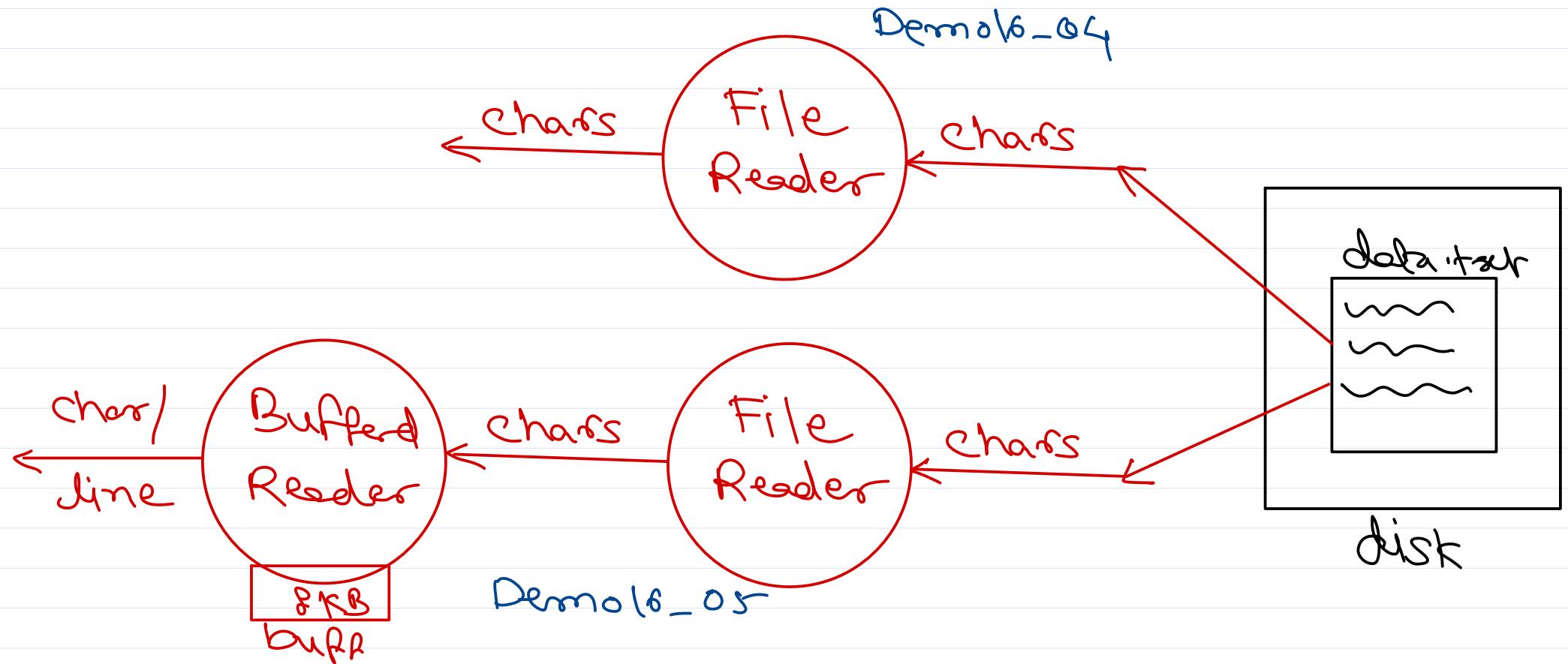
```

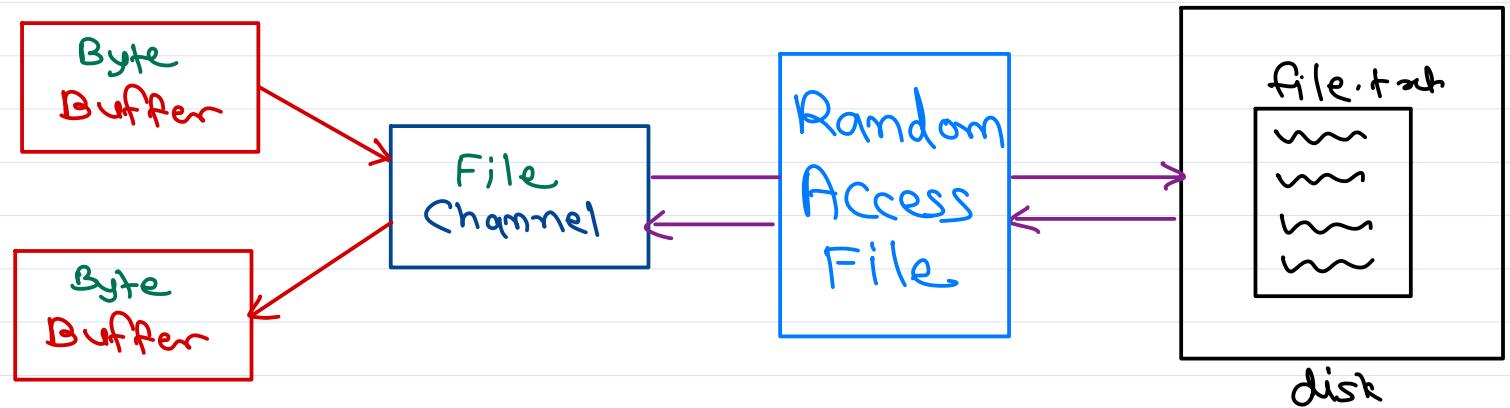
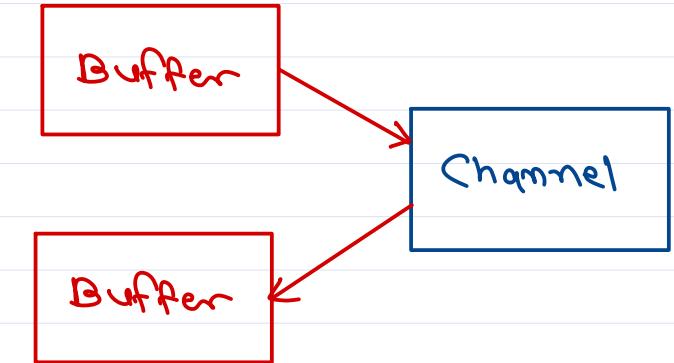
fout = new FOS("stud.db");
bowt = new BOS(fout);
oout = new OOS(bout);
    
```













Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>



Core Java

Agenda

- Quick Revision
- Java IO framework
- Java NIO
- Reflection

Quick Revision

- Java 8 Streams
 - Most of stream operations are based on functional interfaces.
 - Intermediate operations -- returns Stream
 - map() -- Function -- process each record in the stream and convert to some result
 - filter() -- Predicate -- check some condition on each record
 - sorted() -- Comparator -- compare two objects in the stream to sort elements
 - distinct() -- skip duplicate
 - limit(), skip()
 - Terminal operations -- returns non-Stream
 - reduce() -- BinaryOperator -- accumulate the result
 - collect() -- Collector object -- convert whole stream into some collection
 - max() -- Comparator -- compare two objects in the stream to find the max
 - count()
- Java IO
 - File system API are provided by java.io.File object.
 - exists(), isDirectory(), isFile(), listFiles(), length(), canRead(), getName(), ...
 - File IO -- binary stream -- InputStream & OutputStream
 - FileInputStream & FileOutputStream -- read/write bytes into file
 - DataInputStream & DataOutputStream -- convert primitive types from/to bytes

- primitive type --> DataOutputStream --> bytes --> FileOutputStream --> file.
 - DataOutput interface provides methods for conversion - writeInt(), writeUTF(), writeDouble(), ...
- primitive type <-- DataInputStream <-- bytes <-- FileInputStream <-- file.
 - DataInput interface provides methods for conversion - readInt(), readUTF(), readDouble(), ...
- ObjectInputStream & ObjectOutputStream -- convert java object from/to bytes
 - primitive type --> ObjectOutputStream --> bytes --> FileOutputStream --> file.
 - ObjectOutput interface provides method for conversion - writeObject().
 - primitive type <-- ObjectInputStream <-- bytes <-- FileInputStream <-- file.
 - ObjectInput interface provides methods for conversion - readObject().

Java IO

Serialization

- Converting state of object into a sequence of bytes is referred as **Serialization**. The sequence of bytes includes object data as well as metadata.
- Serialized data can be further saved into a file (using FileOutputStream) or sent over the network (Marshalling process).
- Converting (serialized) bytes back to the Java object is referred as **Deserialization**.
- These bytes may be received from the file (using FileInputStream) or from the network (Unmarshalling process).

ObjectOutput/ObjectInput interface

- interface ObjectOutput extends DataOutput
 - writeObject(obj)
- interface ObjectInput extends DataInput
 - obj = readObject()

Serializable interface

- Object can be serialized only if class is inherited from Serializable interface; otherwise writeObject() throws NotSerializableException.
- Serializable is a marker interface.

transient fields

- `writeObject()` serialize all non-static fields of the class. If fields are objects, then they are also serialized.
- If any field is intended not to serialize, then it should be marked as "transient".
- The transient and static fields are not serialized.

serialVersionUID field

- Each serializable class is associated with a version number, called a serialVersionUID.
- It is recommended that programmer should define it as a static final long field (with any access specifier). Any change in class fields expected to modify this serialVersionUID.

```
private static final long serialVersionUID = 1001L;
```

- During deserialization, this number is verified by the runtime to check if right version of the class is loaded in the JVM. If this number mismatched, then `InvalidClassException` will be thrown.
- If a serializable class does not explicitly declare a serialVersionUID, then the runtime will calculate a default serialVersionUID value for that class (based on various aspects of the class described in the Java(TM) Object Serialization specs).

Buffered streams

- Each `write()` operation on `FileOutputStream` will cause data to be written on disk (by OS). Accessing disk frequently will reduce overall application performance. Similar performance problems may occur during network data transfer.
- `BufferedOutputStream` classes hold data into a in-memory buffer before transferring it to the underlying stream. This will result in better performance.
 - Java object --> `ObjectOutputStream` --> `BufferedOutputStream` --> `FileOutputStream` --> file on disk.
- Data is sent to underlying stream when buffer is full or `flush()` called explicitly.
- `BufferedInputStream` provides a buffering while reading the file.
- The buffer size can be provided while creating the respective objects.

PrintStream class

- Produce formatted output (in bytes) and send to underlying stream.

- Formatted output is done using methods print(), println(), and printf().
- System.out and System.err are objects of PrintStream class.

Scanner class

- Added in Java 5 to get the formatted input.
- It is java.util package (not part of java io framework).

```
Scanner sc = new Scanner(inputStream);
// OR
Scanner sc = new Scanner(inputFile);
```

Character streams

- Character streams are used to interact with text file.
- Java char takes 2 bytes (unicode), however char stored in disk file may take 1 or more bytes depending on char encoding.
 - <https://www.w3.org/International/questions/qa-what-is-encoding>
- The character stream does conversion from java char to byte representation and vice-versa (as per char encoding).
- The abstract base classes for the character streams are the Reader and Writer class.
- Writer class -- write operation
 - void close() -- close the stream
 - void flush() -- writes data (in memory) to underlying stream/device.
 - void write(char[] b) -- writes char array to underlying stream/device.
 - void write(int b) -- writes a char to underlying stream/device.
- Writer Sub-classes
 - FileWriter, OutputStreamWriter, PrintWriter, BufferedWriter, etc.
- Reader class -- read operation
 - void close() -- close the stream
 - int read(char[] b) -- reads char array from underlying stream/device
 - int read() -- reads a char from the underlying device/stream. Returns -1

- Reader Sub-classes
 - FileReader, InputStreamReader, BufferedReader, etc.

Java NIO

- Java NIO (New IO) is an alternative IO API for Java.
- Java NIO offers a different IO programming model than the traditional IO APIs.
- Since Java 7.
- Java NIO enables you to do non-blocking (not fully) IO.
- Java NIO consist of the following core components:
 - Channels
 - Buffers
 - Selectors
- Java NIO also provides helper classes Path & Files.
 - exists()
 - ...

Path and Files

- A Java Path instance represents a path in the file system. A path can point to either a file or a directory. A path can be absolute or relative.

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

- Files class (java.nio.file.Files) provides several methods for manipulating files in the file system.

Channels and Buffers

- All IO in NIO starts with a Channel. A Channel is a bit like a stream. From the Channel data can be read into a Buffer. Data can also be written from a Buffer into a Channel.

NIO Channels

- Java NIO Channels are similar to streams with a few differences:
 - You can both read and write to a Channel. Streams are typically one-way (read or write).
 - Channels can be read and written asynchronously.
 - Channels always read to, or write from, a Buffer.
- Channel Examples
 - FileChannel
 - DatagramChannel // UDP protocol
 - SocketChannel, ServerSocketChannel // TCP protocol

NIO Buffers

- A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.
- Using a Buffer to read and write data typically follows this 4-step process:
 - Write data into the Buffer
 - Call buffer.flip()
 - Read data out of the Buffer
 - Call buffer.clear() or buffer.compact()
- Buffer Examples
 - ByteBuffer
 - CharBuffer
 - DoubleBuffer
 - FloatBuffer
 - IntBuffer
 - LongBuffer
 - ShortBuffer

Channel and Buffer Example

```
RandomAccessFile aFile = new RandomAccessFile("somefile.txt", "rw");
FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf); // write data into buffer (from channel)
while (bytesRead != -1) {
    System.out.println("Read " + bytesRead);
    buf.flip(); // switch buffer from write mode to read mode

    while(buf.hasRemaining()){
        System.out.print((char) buf.get()); // read data from the buffer
    }

    buf.clear(); // clear the buffer
    bytesRead = inChannel.read(buf);
}
aFile.close();
```

Java NIO vs Java IO

- IO: Stream-oriented
- NIO: Buffer-oriented
- IO: Blocking IO
- NIO: Non-blocking IO

Reflection

Concept

- Each class is associate a metadata.
 - .class = class header + meta data + byte code + ...

- metadata includes
 - name, type (class, interface, enum), flags (abstract/final/static), access specifier (public, default, ...)
 - super class and interfaces information
 - fields info (name, type, access specifier, final/static, ...)
 - constructor info (access specifier, args ...)
 - methods info (name, return type, access specifier, final/static, args, ...)
 - annotations info
- This metadata can be inspected and used for dynamic invocation.

java.lang.Class object

- java.lang.Class object is associated with each class loaded in JVM. This object stores complete metadata about that class.
- To access java.lang.Class for the given class.
 - method1: If class name is given as a string (from Scanner, config file, ...)

```
String className = "..."; // e.g. java.lang.Object  
Class c = Class.forName(className); // internally loads the class (by classloader) if not loaded already
```

- method2: If class is present in the project/classpath and need to access in code.

```
Class c = ClassName.class; // .class is like a hidden static member of the class
```

- method3: If object of the class is given

```
Class c = obj.getClass();
```

Inspecting metadata

```
```Java
String className = c.getName();
Class superClass = c.getSuperclass();
Class[] superInterfaces = c.getInterfaces();

Field[] fields = c.getFields(); // c.getDeclaredFields();
for(Field field : fields)
 System.out.println(field.toString());

Method[] methods = c.getMethods(); // c.getDeclaredMethods();
for(Method method : methods)
 System.out.println(method.toString());

Constructor[] ctors = c.getConstructors(); // c.getDeclaredConstructors();
for(Constructor ctor : ctors)
 System.out.println(ctors.toString());

Annotation[] annotations = c.getAnnotations(); // c.getDeclaredAnnotations();
for(Constructor ann : annotations)
 System.out.println(ann.toString());
...```

```

## Applications of Reflection

- Get information about a class - Intellisense, Display information (like javap), ...
- Create object dynamically and invoke methods - Used in all advanced java frameworks (like Java EE, Spring, Hibernate, ...)
- Access private members of the class

## Invoking method dynamically

```
```Java
public class Middleware {
    public static Object invoke(String className, String methodName, Class[] methodParamTypes, Object[] methodArgs) throws
Exception {
        // load the given class
        Class c = Class.forName(className);
        // create object of that class
        Object obj = c.newInstance(); // also invokes param-less constructor
        // find the desired method
        Method method = c.getDeclaredMethod(methodName, methodParamTypes);
        // allow to access the method (irrespective of its access specifier)
        method.setAccessible(true);
        // invoke the method on the created object with given args & collect the result
        Object result = method.invoke(obj, methodArgs);
        // return the results
        return result;
    }
}
```
```Java
// invoking method statically
Date d = new Date();
String result = d.toString();
```
```Java
// invoking method dynamically
String result = Middleware.invoke("java.util.Date", "toString", null, null);
```

```

## Reflection Tutorial

- You may refer this after lab hours.
  - [https://youtu.be/lAoNJ\\_7LD44](https://youtu.be/lAoNJ_7LD44)

## Assignment

1. Write a program that inputs 4 lines and stored them in a text file. Use BufferedWriter class.
2. Write a program that reads data from emp.csv file in a Stream. Calculate total salary of all clerk in dept 20 using stream apis.
3. Create a class Arithmetic in your project with no fields and methods int add(int, int), int subtract(int, int), int multiply(int, int), etc. Access the its metadata using Arithmetic.class and display it.
4. Write all native methods of java.lang.Object in your notebook.

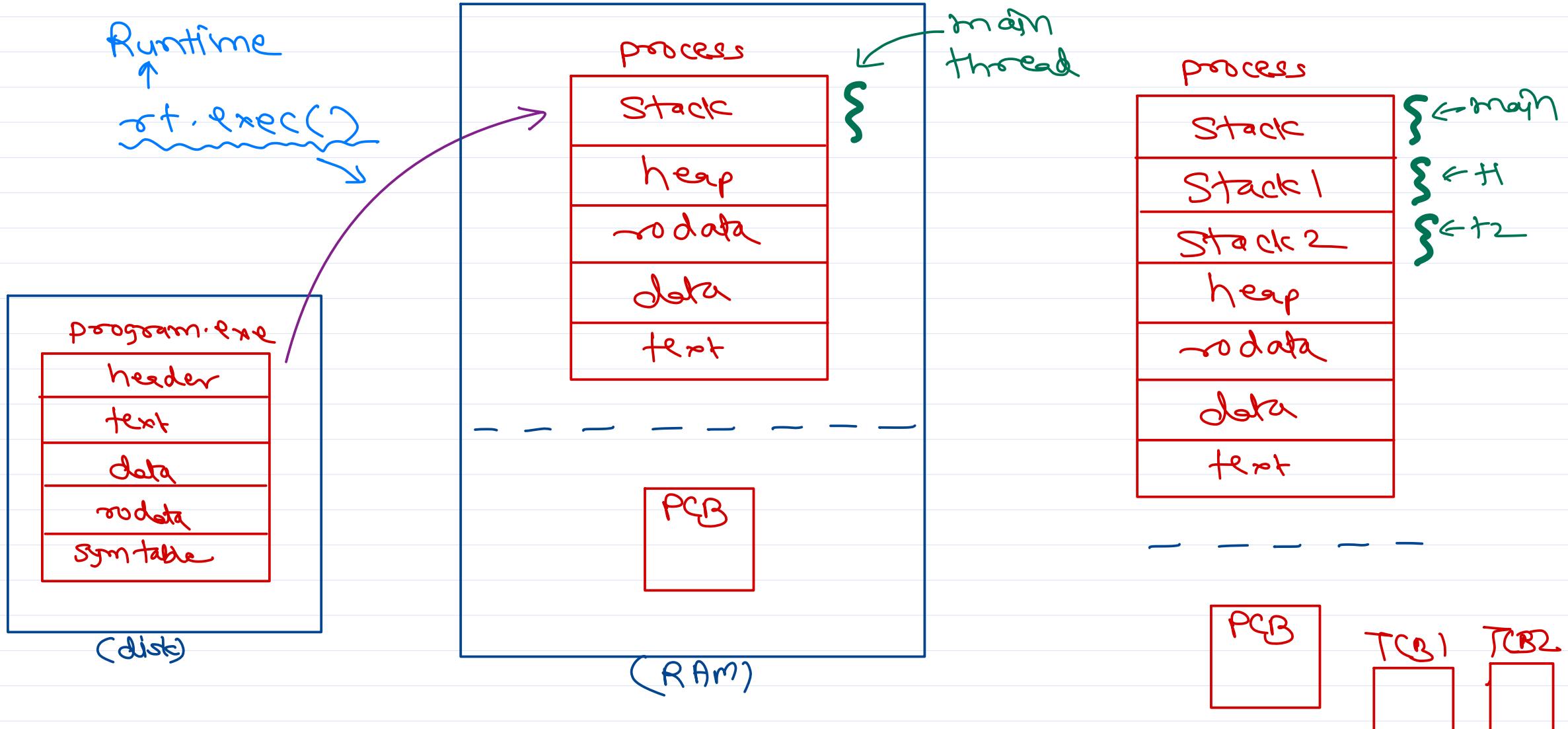
SUNBEAM INFO TECH



# Core Java

*Trainer: Nilesh Ghule*







*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>



# Core Java

---

## Agenda

- Quick Revision
- Reflection
- Annotation
- Process vs Thread
- Multi-threading

## Quick Revision

- Serialization (transient, serialVersionUID)
- Character streams
- Java NIO (Files, Channel & Buffers)
- Reflection

## Day15 Assignment

1. Write a program that inputs 4 lines and stored them in a text file. Use BufferedWriter class.

```
try(FileWriter fw = new FileWriter("file.txt")) {
 try(BufferedWriter bw = new BufferedWriter(fw)) {
 Scanner sc = new Scanner(System.in);
 for(int i=1; i<=4; i++) {
 System.out.print("Enter Line " + i + " : ");
 String line = sc.nextLine();
 bw.write(line);
 }
 }
}
```

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

2. Write a program that reads data from emp.csv file in a Stream. Calculate total salary of all clerk in dept 20 using stream apis.

```
public static Emp parseEmp(String line) {
 String[] parts = line.split(",");
 Emp e = new Emp();
 // ...
 return e;
}
```

```
Path empFilePath = Paths.get("emp.csv");
Stream<String> stream = Files.lines(empFilePath); // return Stream<String>
double totalSal = stream
 .map(line -> parseEmp(line)) // return Stream<Emp>
 .predicate(e -> e.getDeptno() == 20 && e.getJob().equals("CLERK")) // return Stream<Emp> -- clerks in dept 20
 .map(e -> e.getSal()) // return Stream<Double>
 .reduce(0.0, (a,b)->a + b);
```

3. Create a class Arithmetic in your project with no fields and methods int add(int, int), int subtract(int, int), int multiply(int, int), etc. Access the its metadata using Arithmetic.class and display it.

```
class Arithmetic {
 public Integer add(Integer a, Integer b) {
 return a + b;
 }
 public Integer subtract(Integer a, Integer b) {
```

```
 return a - b;
 }
 private Double multiply(Double a, Double b) {
 return a * b;
 }
 public String toString() {
 return "Arithmetric class";
 }
}
```

```
Class<?> c = Arithmetric.class;

String className = c.getName();
Class superClass = c.getSuperclass();

Method[] methods = c.getDeclaredMethods();
for(Method method : methods)
 System.out.println(method.toString());
```

4. Write all native methods of java.lang.Object in your notebook.

- public final native java.lang.Class<?> getClass();
- public native int hashCode();
- protected native java.lang.Object clone() throws java.lang.CloneNotSupportedException;
- public final native void notify();
- public final native void notifyAll();
- public final native void wait(long) throws java.lang.InterruptedException;

## Reflection

### Applications of Reflection

- Get information about a class - Intellisense, Display information (like javap), ...

```
cmd> javap java.lang.Object
show all public methods/fields

cmd> javap java.lang.Object
show all methods/fields

cmd> javap -v java.lang.Object
detailed class file contents including byte code, constant pool, ...
```

- Create object dynamically and invoke methods - Used in all advanced java frameworks (like Java EE, Spring, Hibernate, ...)
- Access private members of the class

### Invoking method dynamically

```
```java  
public class Middleware {  
    public static Object invoke(String className, String methodName, Class[] methodParamTypes, Object[] methodArgs) throws  
Exception {  
    // load the given class  
    Class c = Class.forName(className);  
    // create object of that class  
    Object obj = c.newInstance(); // also invokes param-less constructor  
    // find the desired method  
    Method method = c.getDeclaredMethod(methodName, methodParamTypes);  
    // allow to access the method (irrespective of its access specifier)  
    method.setAccessible(true);  
    // invoke the method on the created object with given args & collect the result  
    Object result = method.invoke(obj, methodArgs);  
    // return the results
```

```
        return result;
    }
}

```
```Java
// invoking method statically
Date d = new Date();
String result = d.toString();
```
```
```Java
// invoking method dynamically
String result = Middleware.invoke("java.util.Date", "toString", null, null);
```
```
```

## Reflection Tutorial

- You may refer this after lab hours.
  - [https://youtu.be/lAoNJ\\_7LD44](https://youtu.be/lAoNJ_7LD44)

## Annotations

- Added in Java 5.0.
- Annotation is a way to associate metadata with the class and/or its members.
- Annotation applications
  - Information to the compiler
  - Compile-time/Deploy-time processing
  - Runtime processing
- Annotation Types
  - Marker Annotation: Annotation is not having any attributes.
    - @Override, @Deprecated, @FunctionalInterface ...
  - Single value Annotation: Annotation is having single attribute -- usually it is "value".

- @SuppressWarnings("deprecated"), ...
- Multi value Annotation: Annotation is having multiple attribute
  - @RequestMapping(method = "GET", value = "/books"), ...

## Pre-defined Annotations

- @Override
  - Ask compiler to check if corresponding method (with same signature) is present in super class.
  - If not present, raise compiler error.
- @FunctionalInterface
  - Ask compiler to check if interface contains single abstract method.
  - If zero or multiple abstract methods, raise compiler error.
- @Deprecated
  - Inform compiler to give a warning when the deprecated type/member is used.
- @SuppressWarnings
  - Inform compiler not to give certain warnings: e.g. deprecation, rawtypes, unchecked, serial, unused
  - @SuppressWarnings("deprecation")
  - @SuppressWarnings({"rawtypes", "unchecked"})
  - @SuppressWarnings("serial")
  - @SuppressWarnings("unused")

## Meta-Annotations

- Annotations that apply to other annotations are called meta-annotations.
- Meta-annotation types defined in java.lang.annotation package.

### @Retention

- RetentionPolicy.SOURCE
  - Annotation is available only in source code and discarded by the compiler (like comments).
  - Not added into .class file.

- Used to give information to the compiler.
- e.g. @Override, ...
- RetentionPolicy.CLASS
  - Annotation is compiled and added into .class file.
  - Discared while class loading and not loaded into JVM memory.
  - Used for utilities that process .class files.
  - e.g. Obfuscation utilities can be informed not to change the name of certain class/member using @SerializedName, ...
- RetentionPolicy.RUNTIME
  - Annotation is compiled and added into .class file. Also loaded into JVM at runtime and available for reflective access.
  - Used by many Java frameworks.
  - e.g. @RequestMapping, @Id, @Table, @Controller, ...

#### **@Target**

- Where this annotation can be used.
- ANNOTATION\_TYPE, CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE, TYPE\_PARAMETER, TYPE\_USE
- If annotation is used on the other places than mentioned in @Target, then compiler raise error.

#### **@Documented**

- This annotation should be documented by javadoc or similar utilities.

#### **@Repeatable**

- The annotation can be repeated multiple times on the same class/target.

#### **@Inherited**

- The annotation gets inherited to the sub-class and accessible using c.getAnnotation() method.

#### **Custom Annotation**

- Annotation to associate developer information with the class and its members.

```
@Inherited
@Retention(RetentionPolicy.RUNTIME) // the def attribute is considered as "value" = @Retention(value =
RetentionPolicy.RUNTIME)
@Taget({TYPE, CONSTRUCTOR, FIELD, METHOD}) // { } represents array
@interface Developer {
 String firstName();
 String lastName();
 String company() default "Sunbeam";
 String value() default "Software Engg";
}

@Repeatable
@Retention(RetentionPolicy.RUNTIME)
@Taget({TYPE})
@interface CodeType {
 String[] value();
}
```

```
//@Developer(firstName="Nilesh", lastName="Ghule", value="Technical Director") // compiler error -- @Developer is not
@Repeatable
@CodeType({"businessLogic", "algorithm"})
@Developer(firstName="Nilesh", lastName="Ghule", value="Technical Director")
class MyClass {
 //
 @Developer(firstName="Shubham", lastName="Borle", company="Sunbeam Karad ")
 private int myField;
 @Developer(firstName="Rahul", lastName="Sansuddi")
 public MyClass() {

}
```

```
@Developer(firstName="Shubham", lastName="Borle", company="Sunbeam Karad ")
public void myMethod() {
 @Developer(firstName="James", lastName="Bond") // compiler error
 int localVar = 1;
}
}
```

```
// @Developer is inherited
@CodeType("frontEnd")
@CodeType("businessLogic") // allowed because @CodeType is @Repeatable
class YourClass extends MyClass {
 ...
}
```

## Annotation processing (using Reflection)

```
Annotation[] anns = MyClass.class.getDeclaredAnnotations();
for (Annotation ann : anns) {
 System.out.println(ann.toString());
 if(ann instanceof Developer) {
 Developer devAnn = (Developer) ann;
 System.out.println(" - Name: " + devAnn.firstName() + " " + devAnn.lastName());
 System.out.println(" - Company: " + devAnn.company());
 System.out.println(" - Role: " + devAnn.value());
 }
}
System.out.println();

Field field = MyClass.class.getDeclaredField("myField");
anns = field.getAnnotations() ;
```

```
for (Annotation ann : anns)
 System.out.println(ann.toString());
System.out.println();

//annts = YourClass.class.getDeclaredAnnotations();
annts = YourClass.class.getAnnotations();
for (Annotation ann : annts)
 System.out.println(ann.toString());
System.out.println();
```

## Process vs Threads

### Program

- Program is set of instructions given to the computer.
- Executable file is a program.
- Executable file contains text, data, rodata, symbol table, exe header.

### Process

- Process is program in execution.
- Program (executable file) is loaded in RAM (from disk) for execution. Also OS keep information required for execution of the program in a struct called PCB (Process Control Block).
- Process contains text, data, rodata, stack, and heap section.

### Thread

- Threads are used to do multiple tasks concurrently within a single process.
- Thread is a lightweight process.
- When a new thread is created, a new TCB is created along with a new stack. Remaining sections are shared with parent process.

### Process vs Thread

- Process is a container that holds resources required for execution and thread is unit of execution/scheduling.
- Each process have one thread created by default -- called as main thread.

## Process creation (Java)

- In Java, process can be created using Runtime object.
- Runtime object holds information of current runtime environment that includes number of processors, JVM memory usage, etc.
- Current runtime can be accessed using static getRuntime() method.

```
Runtime rt = Runtime.getRuntime();
```

- The process is created using exec() method, which returns the Process object. This object represents the OS process and its waitFor() method wait for the process termination (and returns exit status).

```
String[] args = { "/path/of/executable", "cmd-line arg1", ... };
Process p = Runtime.exec(args);
int exitStatus = p.waitFor();
```

## Multi-threading (Java)

- Java applications are always multi-threaded.
- When any java application is executed, JVM creates (at least) two threads.
  - main thread -- executes the application main()
  - GC thread -- does garbage collection (release unreferenced objects)
- Programmer may create additional threads, if required.

### Thread creation

- To create a thread

- step 1: Implement a thread function (task to be done by the thread)
- step 2: Create a thread (with above function)
- Method 1: extends Thread

```
class MyThread extends Thread {
 @Override
 public void run() {
 // task to be done by the thread
 }
}
```

```
MyThread th = new MyThread();
th.start();
```

- Method 2: implements Runnable

```
class MyRunnable implements Runnable {
 @Override
 public void run() {
 // task to be done by the thread
 }
}
```

```
MyRunnable runnable = new MyRunnable();
Thread th = new Thread(runnable);
th.start();
```

- Java doesn't support multiple inheritance. If your class is already inherited from a super class, you cannot extend it from Thread class. Prefer Runnable in this case; otherwise you may choose any method.

```
// In Java GUI application is inherited from Frame class.
// to create run() in the same class, you must use Runnable
class MyGuiApplication extends Frame implements Runnable {
 // ...
 public void run() {
 // ...
 }
 // ...
}
```

## Daemon threads

- By default all threads are non-daemon threads (including main thread).
- We can make a thread as daemon by calling its setDaemon(true) method -- before starting the thread.
- Daemon threads are also called as background threads and they support/help the non-daemon threads.
- When all non-daemon threads are terminated, the Daemon threads get automatically terminated.



# Core Java

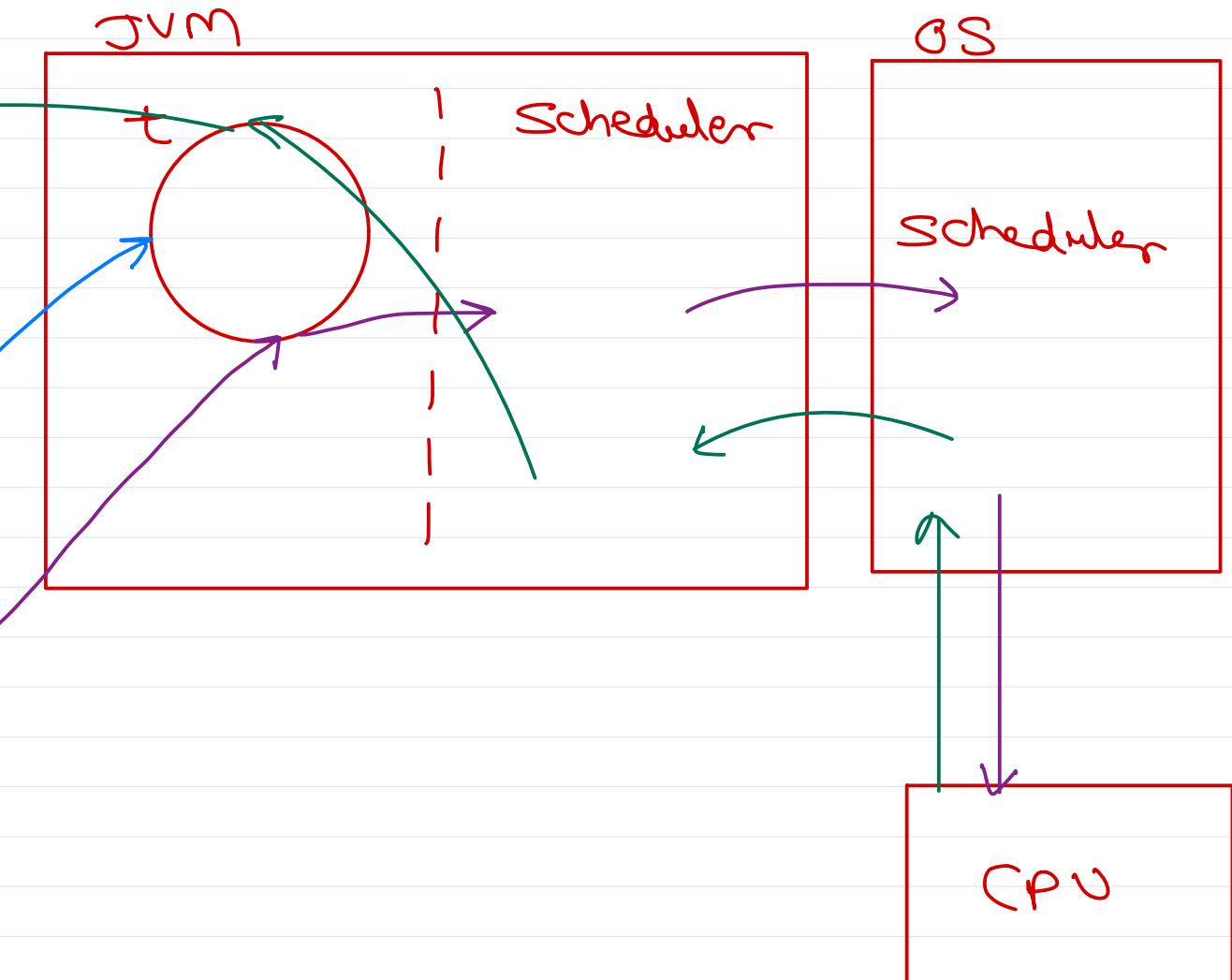
*Trainer: Nilesh Ghule*



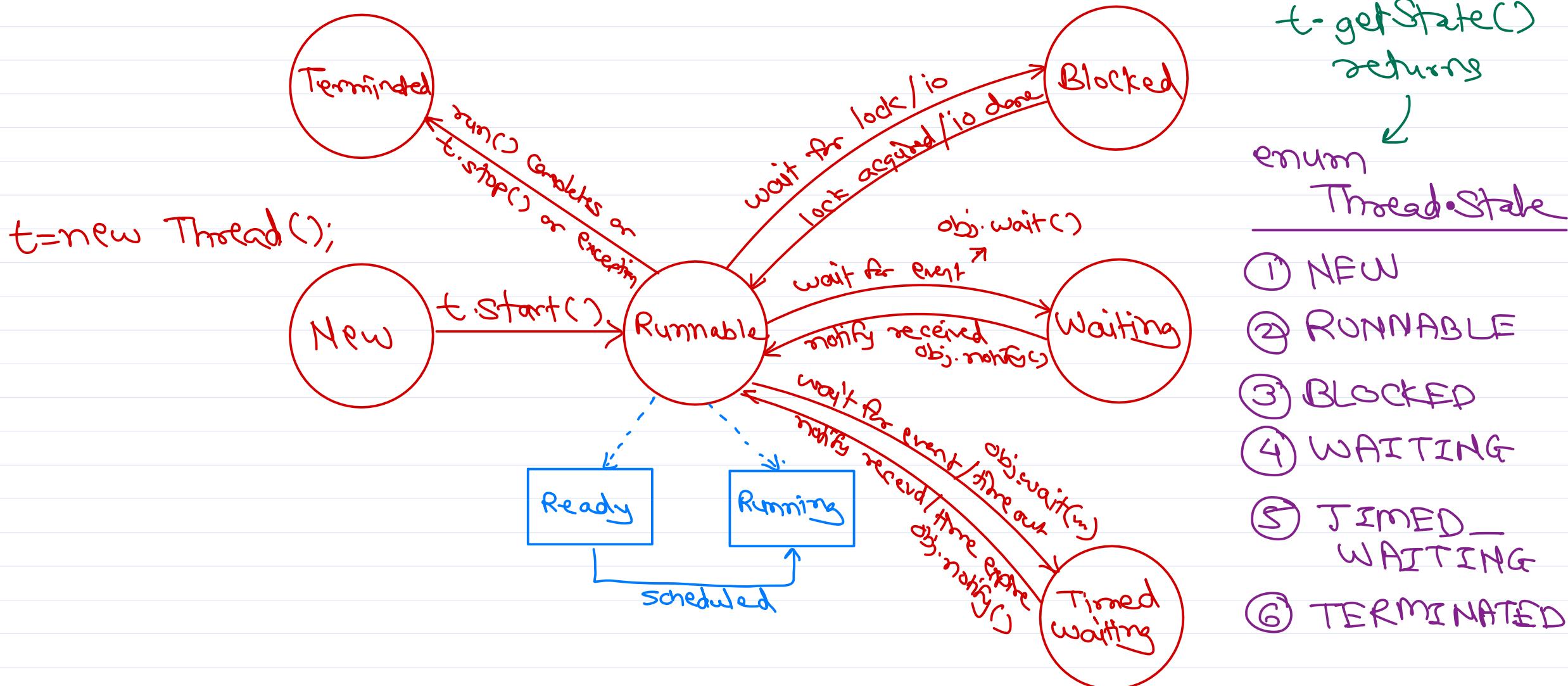
# start() vs run()

```
class My Runnable implements Runnable {
 void run() {}
}
```

① Thread t = new Thread(m);  
 t.start(); ②



# Thread life cycle



deposit Thread:

①

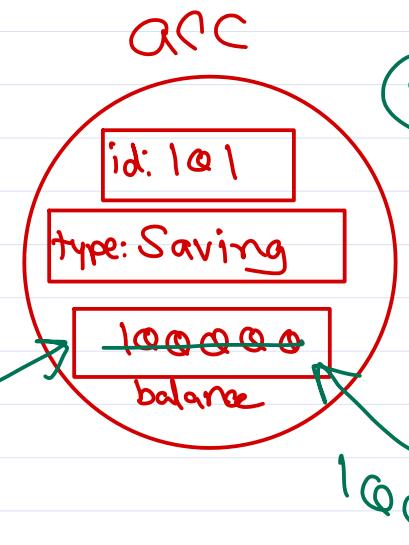
deposit()

101000 100000 + 1000

newbal = balance + amnt;

balance = newbal;

101000



withdrawThread:

②

withdraw()

101000 101000 - 1000

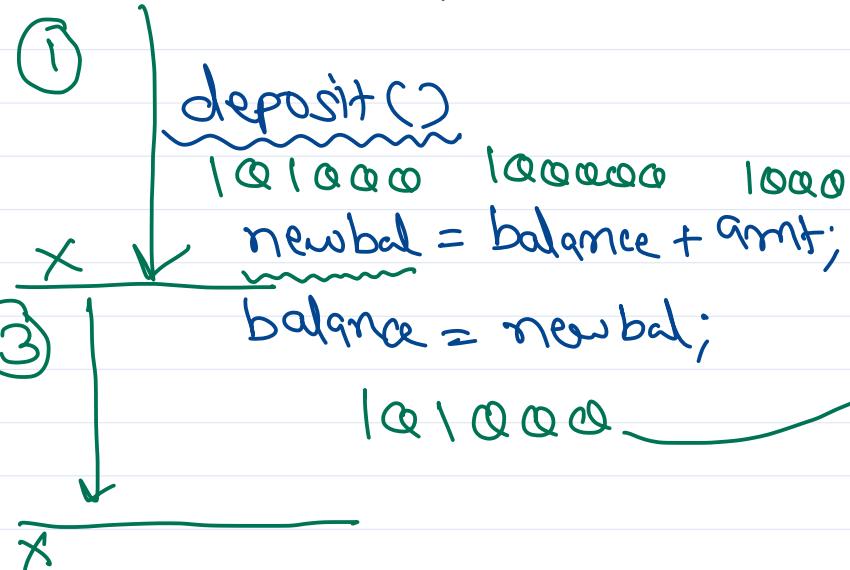
newbal = balance - amnt;

balance = newbal;

100000



deposit Thread:



ACC

id: 101

type: Saving

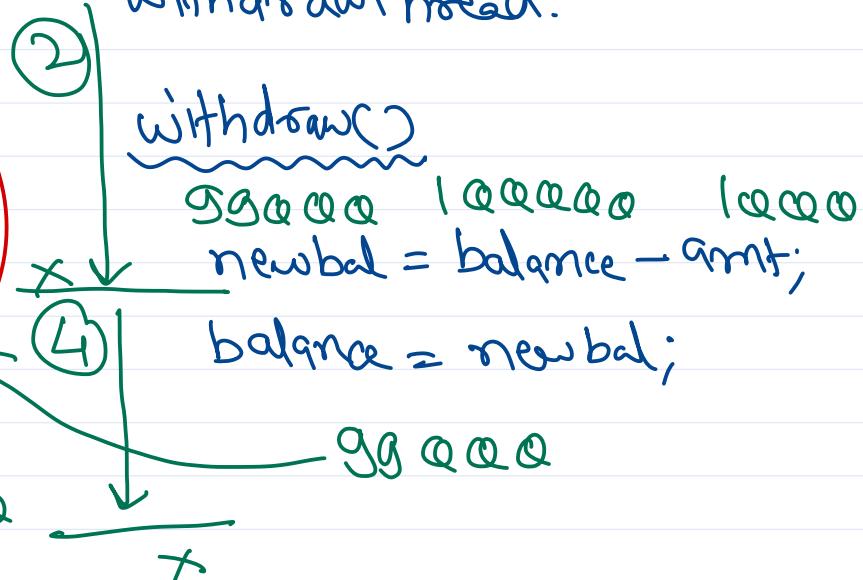
100000

balance

101000

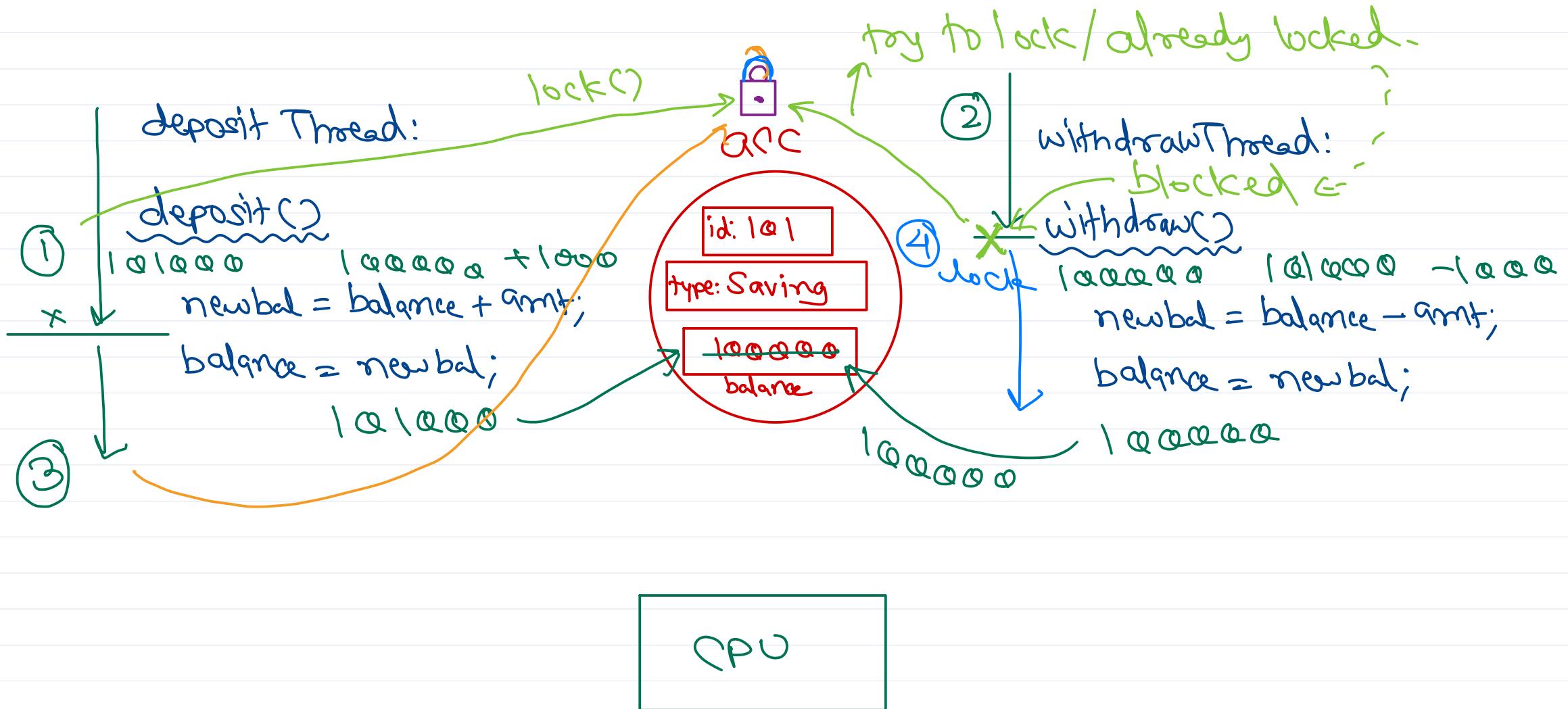
99000

withdrawThread:



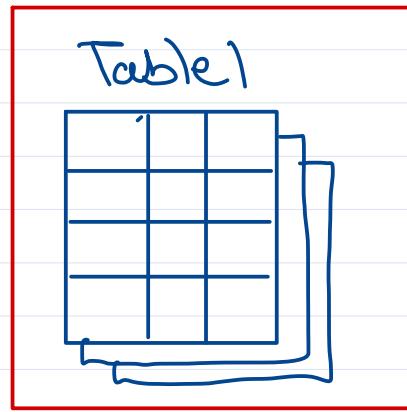
CPU



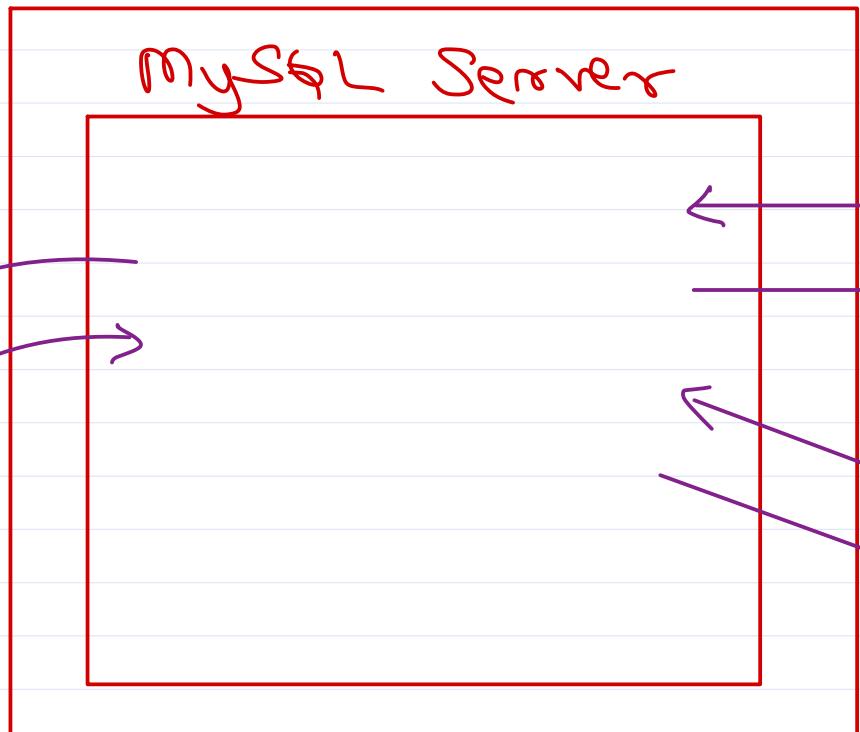


## RDBMS

- ① MySQL
- ② Oracle
- ③ MS-SQL
- ④ PostgreSQL
- ⑤ Derby
- ⑥ ...



disk



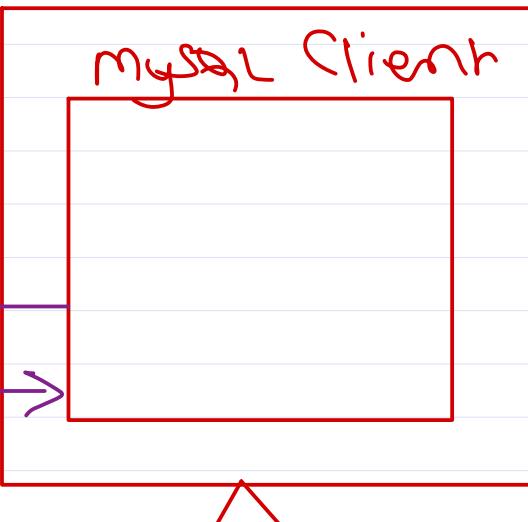
MySQL Server

SQL

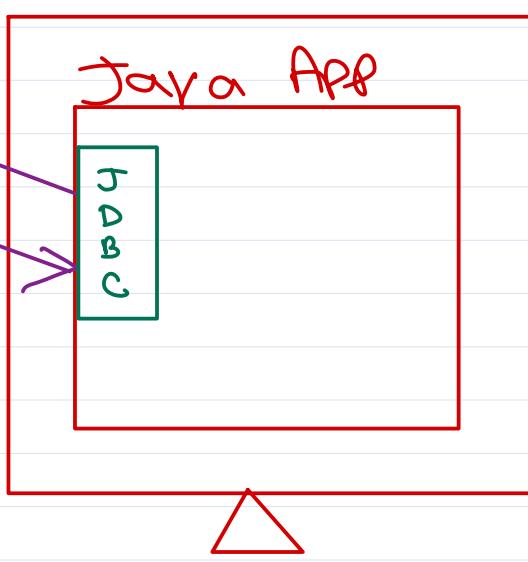
SQL

SQL

- ① DML
- ② DQL
- ③ DDL
- ④ TCL
- ⑤ DCL



MySQL Client



Java App



*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

# Core Java

---

## Agenda

- Multi-threading
- Garbage collection
- RDBMS basics

## Platform Independence

- Java is architecture neutral i.e. can work on various CPU architectures like x86, ARM, SPARC, PPC, etc (if JVM is available on those architectures).
- Java is NOT fully platform independent. It can work on various platforms like Windows, Linux, Mac, UNIX, etc (if JVM is available on those platforms).
- Few features of Java remains platform dependent.
  - Multi-threading (Scheduling, Priority)
  - File IO (Performance, File types, Paths)
  - AWT GUI (Look & Feel)
  - Networking (Socket connection)

## Multi-threading

### **start() vs run()**

- run():
  - Programmer implemented code to be executed by the thread.
- start():
  - Pre-defined method in Thread class.
  - When called, the thread object is submitted to the (JVM/OS) scheduler. Then scheduler select the thread for execution and invoke its run() method.

## Thread life cycle

- NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, TERMINATED
  - NEW: New thread object created (not yet started its execution).
  - RUNNABLE: Thread is running on CPU or ready for execution. Scheduler picks ready thread and dispatch it on CPU.
  - BLOCKED: Thread is waiting for lock to be released. Thread blocks due to synchronized block/method.
  - WAITING: Thread is waiting for the notification. Waiting thread release the acquired lock.
  - TIMED\_WAITING: Thread is waiting for the notification or timeout duration. Waiting thread release the acquired lock.
  - TERMINATED: Thread terminates when run() method is completed, stopped explicitly using stop(), or an exception is raised while executing run().

## Thread methods

- static Thread currentThread()
  - Returns a reference to the currently executing thread object.
- static void sleep(long millis)
  - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- static void yield()
  - A hint to the scheduler that the current thread is willing to yield its current use of a processor.
- Thread.State getState()
  - Returns the state of this thread.
  - State can be NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, TERMINATED
- void run()
  - If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called. If thread class extends from Thread class, this method should be overridden. The default implementation is empty.

- void start()
  - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- void join()
  - Waits for this thread to die/complete.
- boolean isAlive()
  - Tests if this thread is alive.
- void setDaemon(boolean daemon);
  - Marks this thread as either a daemon thread (true) or a user thread (false).
- boolean isDaemon()
  - Tests if this thread is a daemon thread.
- long getId()
  - Returns the identifier of this Thread.
- void setName(String name)
  - Changes the name of this thread to be equal to the argument name.
- String getName()
  - Returns this thread's name.
- void setPriority(int newPriority)
  - Changes the priority of this thread.
  - In Java thread priority can be 1 to 10.
  - May use predefined constants MIN\_PRIORITY(1), NORM\_PRIORITY(5), MAX\_PRIORITY(10).

- int getPriority()
  - Returns this thread's priority.
- ThreadGroup getThreadGroup()
  - Returns the thread group to which this thread belongs.
- void interrupt()
  - Interrupts this thread -- will raise InterruptedException in the thread.
- boolean isInterrupted()
  - Tests whether this thread has been interrupted.

## Synchronization

- When multiple threads try to access same resource at the same time, it is called as Race condition.
- Example: Same bank account undergo deposit() and withdraw() operations simultaneously.
- It may yield in unexpected/undesired results.
- This problem can be solved by Synchronization.
- The synchronized keyword in Java provides thread-safe access.
- In Java synchronization internally use the Monitor object associated with any object. It provides lock/unlock mechanism.
- "synchronized" can be used for block or method.
- It acquires lock on associated object at the start of block/method and release at the end. If lock is already acquired by other thread, the current thread is blocked (until lock is released by the locking thread).
- "synchronized" non-static method acquires lock on the current object i.e. "this". Example:

```
class Account {
 // ...
 public synchronized void deposit(double amount) {
 double newBalance = this.balance + amount;
```

```
 this.balance = newBalance;
 }
 public synchronized void withdraw(double amount) {
 double newBalance = this.balance - amount;
 this.balance = newBalance;
 }
}
```

- "synchronized" static method acquires lock on metadata object of the class i.e. MyClass.class. Example:

```
class MyClass {
 private static int field = 0;
 // called by incThread
 public synchronized static void incMethod() {
 field++;
 }
 // called by decThread
 public synchronized static void decMethod() {
 field--;
 }
}
```

- "synchronized" block acquires lock on the given object.

```
// assuming that no method in Account class is synchronized.

// thread1
synchronized(acc) {
 acc.deposit(1000.0);
}
```

```
// thread2
synchronized(acc) {
 acc.withdraw(1000.0);
}
```

- Alternatively lock can be acquired using ReentrantLock since Java 5.0. Example code:

```
class Example {
 private final ReentrantLock rl = new ReentrantLock();
 public void method() {
 rl.lock();
 try {
 // ...
 }
 finally {
 rl.unlock();
 }
 }
}
```

- Synchronized collections
  - Synchronized collections (e.g. Vector, Hashtable, ...) use synchronized keyword (block/method) to handle race conditions.

## Inter-thread communication

- wait()
  - Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
  - The current thread must own this object's monitor i.e. wait() must be called within synchronized block/method.
  - The thread releases ownership of this monitor and waits until another thread notifies.
  - The thread then waits until it can re-obtain ownership of the monitor and resumes execution.
- notify()

- Wakes up a single thread that is waiting on this object's monitor.
  - If multiple threads are waiting on this object, one of them is chosen to be awakened arbitrarily.
  - The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
  - This method should only be called by a thread that is the owner of this object's monitor.
- notifyAll()
    - Wakes up all threads that are waiting on this object's monitor.
    - The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.
    - This method should only be called by a thread that is the owner of this object's monitor.

## RDBMS

- DBMS is specialized software for database management i.e. CRUD operations.
- RDBMS (Relational DBMS) stores all data in form of tables (rows and columns).
- Each row represent a record, while column represents attributes of the data. The column is associated with some data types e.g. INT, CHAR, DOUBLE, DATE.
- RDBMS understand SQL (Structured Query Language) only. SQL queries can be categorised as
  - DML: INSERT, UPDATE, and DELETE operations
  - DQL: SELECT operations
  - DDL: CREATE TABLE, ALTER TABLE, ...
  - DCL: GRANT and REVOKE permissions
  - TCL: COMMIT, ROLLBACK
- RDBMS is server-client architecture. Server does whole database management as per client requests.
- Popular RDBMS: MySQL, Oracle, MS-SQL, PostgreSQL, etc.

## MySQL Installation

- Download MySQL community edition.
  - <https://dev.mysql.com/downloads/mysql/>
- Install the server (refer PDF).
  - You may choose "Developer Default".
  - Ensure that Visual Studio redistributable is downloaded and installed.
  - Enter root password e.g. "manager".

- This PC (right click) --> Properties --> Advanced System Settings --> Advanced --> Environment Variable --> PATH --> Edit --> New -- C:\Program Files\MySQL\MySQL Shell 8.0\bin\ --> OK
- On new terminal (command prompt or powershell) continue the next command.

```
cmd> mysql -u root -pmanager
```

```
CREATE DATABASE test;
USE test;

CREATE TABLE students(roll INT, name CHAR(40), marks DOUBLE);

INSERT INTO students VALUES(1, 'Nilesh', 77.00);
INSERT INTO students VALUES(2, 'Vishal', 87.43);
INSERT INTO students VALUES(3, 'Smita', 95.34);
INSERT INTO students VALUES(4, 'Yogesh', 79.65);

SELECT * FROM students;

SELECT roll, name FROM students;
```

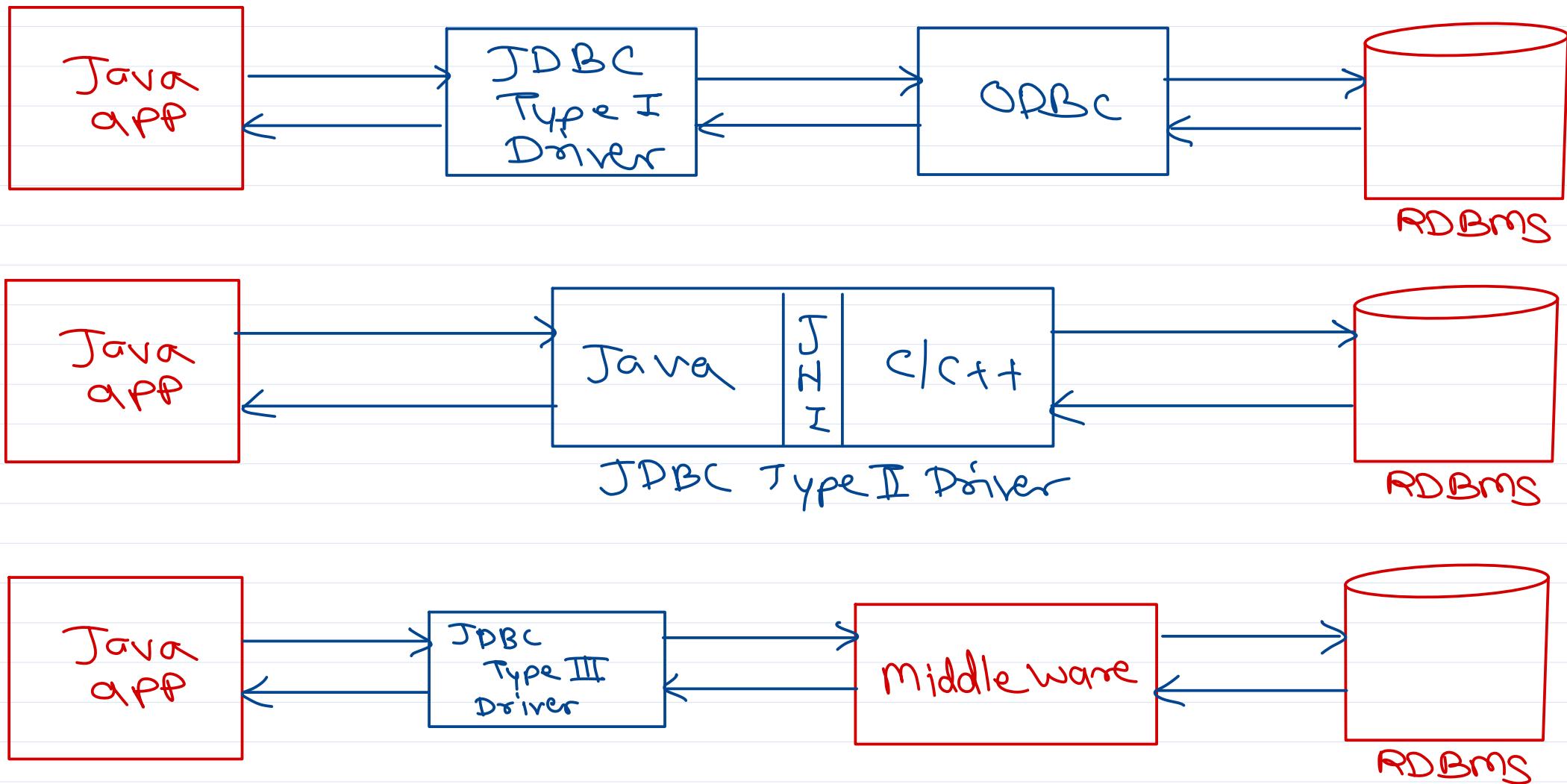


# Core Java

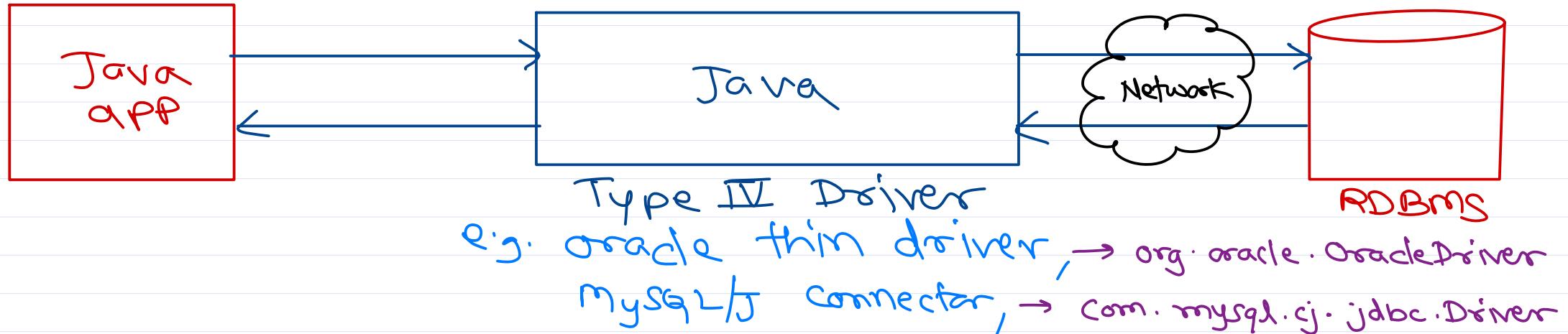
*Trainer: Nilesh Ghule*



# JDBC Drivers



# JDBC Drivers



# JDBC Specification

JDBC is a specification.

- interfaces }  
- helper classes }      java.sql package

## JDBC interfaces

- ① Driver : Connect to db over the network.
- ② Connection : represent db connection (encapsulate socket).  
used to create query(statement, tx mgmt.)
- ③ Statement : represent SQL query & execute it.  
returns num of rows affected ← ExecuteUpdate() – non-SELECT queries  
returns ResultSet ← ExecuteQuery() – SELECT queries
- ④ ResultSet : represent db response - multiple rows/cols.  
used to access result row by row.

## JDBC Helper classes

Driver Manager: Manage JDBC drivers.

Use appropriate driver to connect to db.

Interfaces are implemented by JDBC drivers.

e.g. com.mysql.cj.jdbc.

Driver class is inherited from java.sql.Driver interface.



# JDBC Programming

① Add MySQL JDBC driver Jar into appm CLASSPATH.

eclipse project → properties → Java Build Path → Libraries → Add External Jars  
→ Select MySQL-Jar.

1. Load & register MySQL driver class (one-time).

Class.forName("com.mysql.cj.jdbc.Driver"); → static block

2. Create a JDBC connection.

Con=DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "manager");  
server name ↓ port ← db name  
db url user password

3. Create a Statement.

stmt=con.createStatement();

4. Execute the stmt & process the result.

cnt = stmt.executeUpdate("DELETE FROM students");

OR

rs = stmt.executeQuery("SELECT \* FROM students");

while(rs.next())

System.out.println(rs.getInt("roll") + rs.getString("name") + rs.getDouble("marks"));

rs.close();

5. Close stmt & connection.

stmt.close();  
con.close();





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

# Core Java

## Agenda

- SQL queries
- JDBC Programming

## SQL

```
cmd> mysql -u root -pmanager
```

```
USE test;

SHOW TABLES;

SELECT * FROM students;

\! cls

UPDATE students SET marks=82.13 WHERE roll=4;

SELECT * FROM students;

SELECT * FROM students WHERE roll=4;

UPDATE students SET marks=85.13, name='Yogeshwar' WHERE roll=4;

SELECT * FROM students WHERE roll=4;
```

```
DELETE FROM students WHERE roll=4;

SELECT * FROM students;
```

## Java Database Connectivity (JDBC)

- RDBMS understand SQL language only.
- JDBC driver converts Java requests in database understandable form and database response in Java understandable form.
- JDBC drivers are of 4 types
  - Type I - Jdbc Odbc Bridge driver
    - ODBC is standard of connecting to RDBMS (by Microsoft).
    - Needs to create a DSN (data source name) from the control panel.
    - From Java application JDBC Type I driver can communicate with that ODBC driver (DSN).
    - The driver class: sun.jdbc.odbc.JdbcOdbcDriver -- built-in in Java.
    - database url: jdbc:odbc:dsn
    - Advantages:
      - Can be easily connected to any database.
    - Disadvantages:
      - Slower execution (Multiple layers).
      - The ODBC driver needs to be installed on the client machine.
  - Type II - Partial Java/Native driver
    - Partially implemented in Java and partially in C/C++. Java code calls C/C++ methods via JNI.
    - Different driver for different RDBMS. Example: Oracle OCI driver.
    - Advantages:
      - Faster execution
    - Disadvantages:
      - Partially in Java (not truly portable)
      - Different driver for Different RDBMS
  - Type III - Middleware/Network driver
    - Driver communicate with a middleware that in turn talks to RDBMS.

- Example: WebLogic RMI Driver
- Advantages:
  - Client coding is easier (most task done by middleware)
- Disadvantages:
  - Maintaining middleware is costlier
  - Middleware specific to database
- Type IV
  - Database specific driver written completely in Java.
  - Fully portable.
  - Most commonly used.
  - Example: Oracle thin driver, MySQL Connector/J, ...

## MySQL Programming Steps

- Refer slides

## MySQL Driver Download

- <https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.30>

## SQL Injection

- Building queries by string concatenation is inefficient as well as insecure.
- Example:

```
roll = sc.nextLine();
sql = "DELETE FROM students WHERE roll="+roll;
```

- If user input "1", then effective SQL will be "DELETE FROM students WHERE roll=1". This will delete single row from the RDBMS.

- If user input "1 OR 1", then effective SQL will be "DELETE FROM students WHERE roll=1 OR 1". Here "1" represent true condition and kit will delete all rows from the RDBMS.
- In Java, it is recommended NOT to use Statement and building SQL by string concatenation. Instead use PreparedStatement.

## PreparedStatement

- PreparedStatement represents parameterized queries.

```
String sql = "SELECT * FROM students WHERE name=?";
PreparedStatement stmt = con.prepareStatement(sql);
System.out.print("Enter name to find: ");
String name = sc.next();
stmt.setString(1, name);
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
 int roll = rs.getInt("roll");
 String name = rs.getString("name");
 double marks = rs.getDouble("marks");
 System.out.printf("%d, %s, %.2f\n", roll, name, marks);
}
```

- The same PreparedStatement can be used for executing multiple queries. There is no syntax checking repeated. This improves the performance.

## Properties

- java.util.Properties is a legacy java collection.
- java.util.Properties is inherited from java.util.Hashtable.
- It stores key value pairs (Strings). Mainly used for storing config/settings.
- Typically loaded from some.properties files.

```
Properties props = new Properties(); // empty collection
InputStream in = Util.class.getResourceAsStream("/some.properties");
props.load(in); // read key-value from some.properties and load into the props collection.
```

```
String value = props.getProperty(key);
```

SUNBEAM INFOTECH