



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">- OpenJDK (Open Java Development Kit) is a free and open source implementation of the Java Platform. It is the result of an effort Sun Microsystems began in 2006.
Java SE 9	September 21, 2017	<ul style="list-style-type: none">- Java Language and Virtual Machine Specifications: https://docs.oracle.com/javase/specs/
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.

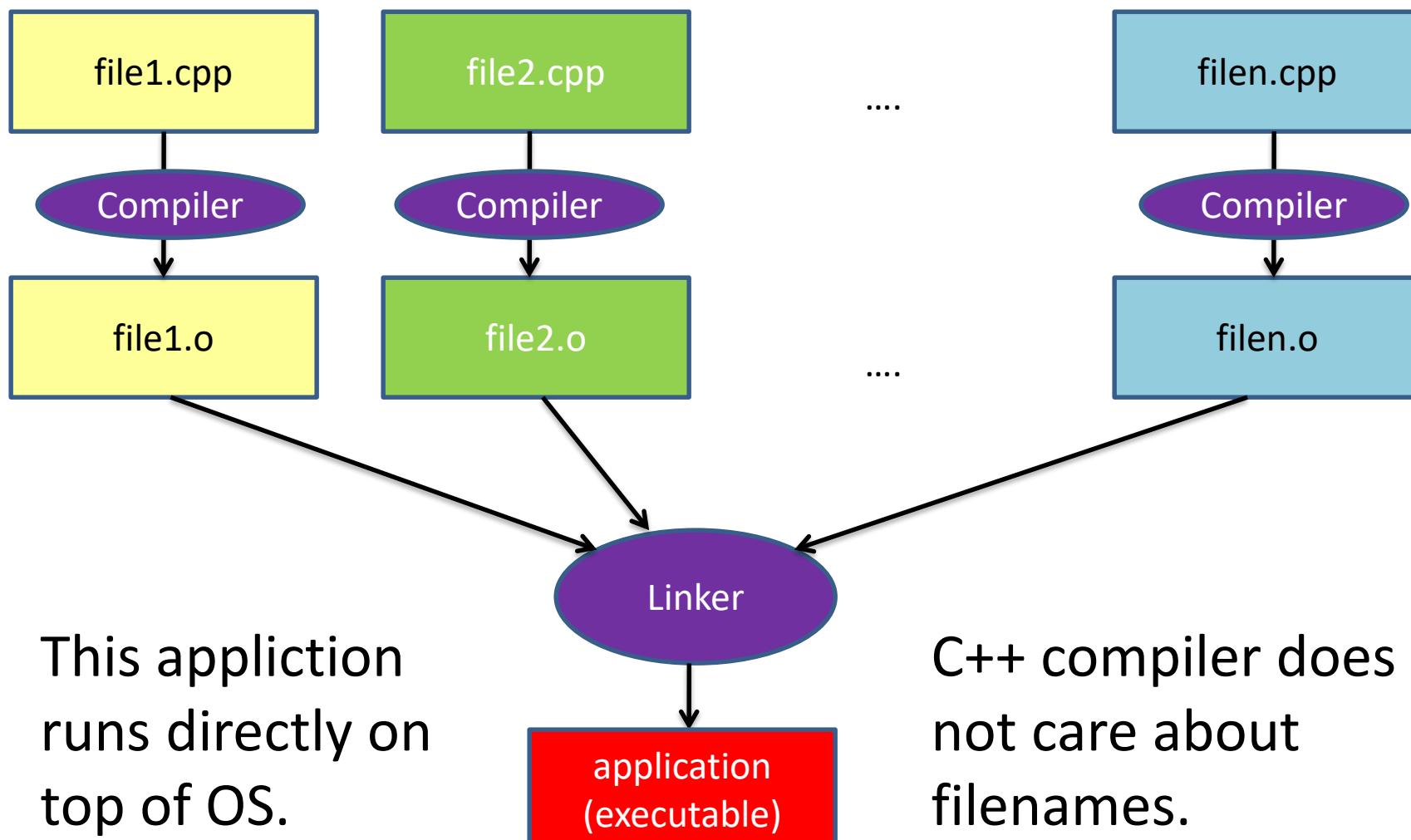


SDK, JDK, JRE, JVM

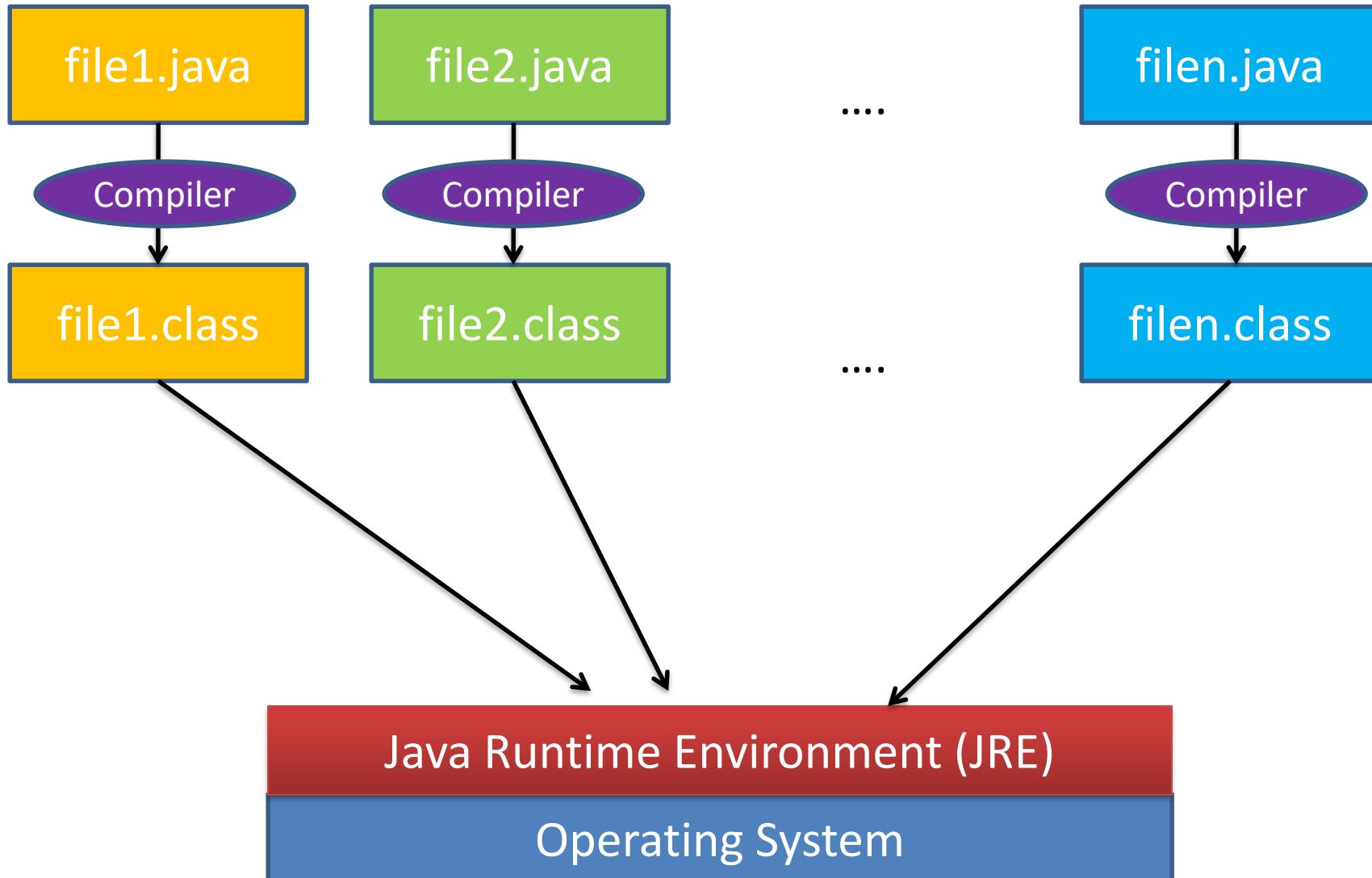
- **SDK** = Development Tools + Documentation + Libraries + Runtime Environment.
- **JDK** = Java Development Tools + Java Docs + **rt.jar** + **JVM**.
 - JDK : Java Development Kit.
 - It is a software, that need to be install on developers machine.
 - We can download it from oracle official website.
- **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
```

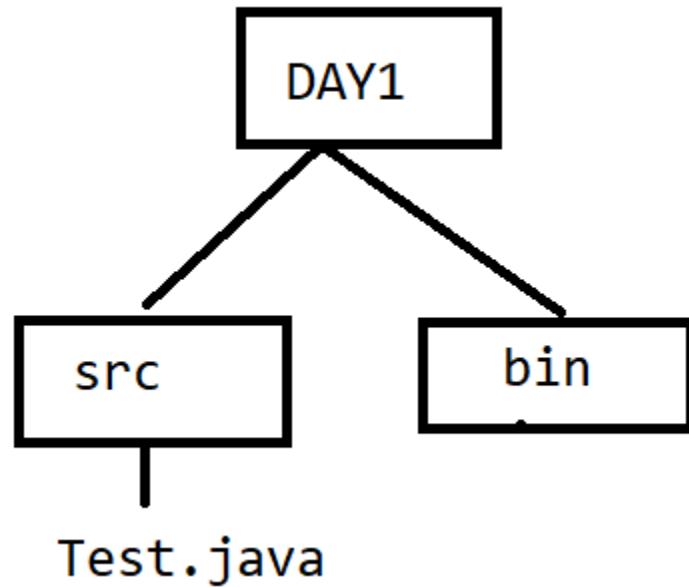
```
System      : Final class declared in java.lang package  
out         : public static final field of System class. Type of out is PrintStream  
println     : Non static method of java.io.PrintStream class
```

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();
}
```

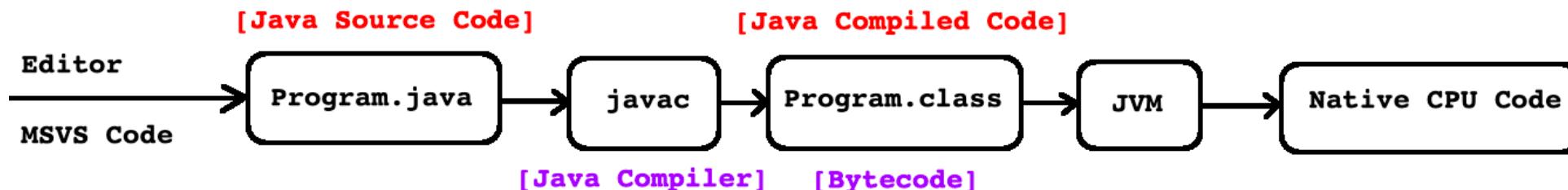


Stream

- 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.



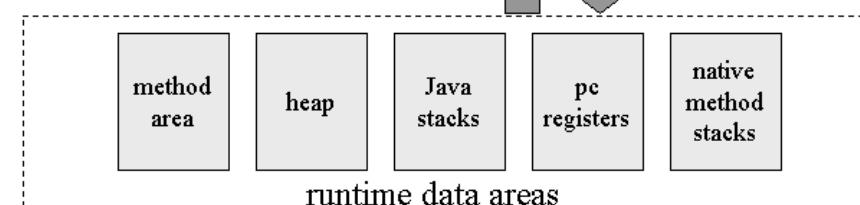
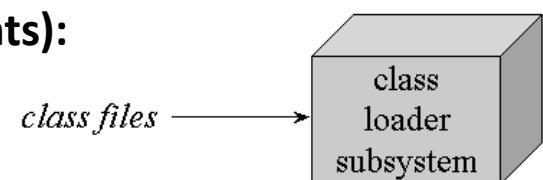
Java application flow of execution



Components of JVM (Major 3 Components):

1. Class Loader Sub System

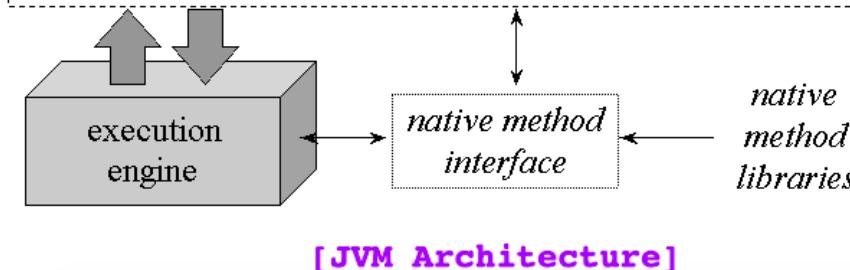
- Bootstrap class loader
- Extension classLoader
- Application classLoader
- User Defined class Loader



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.

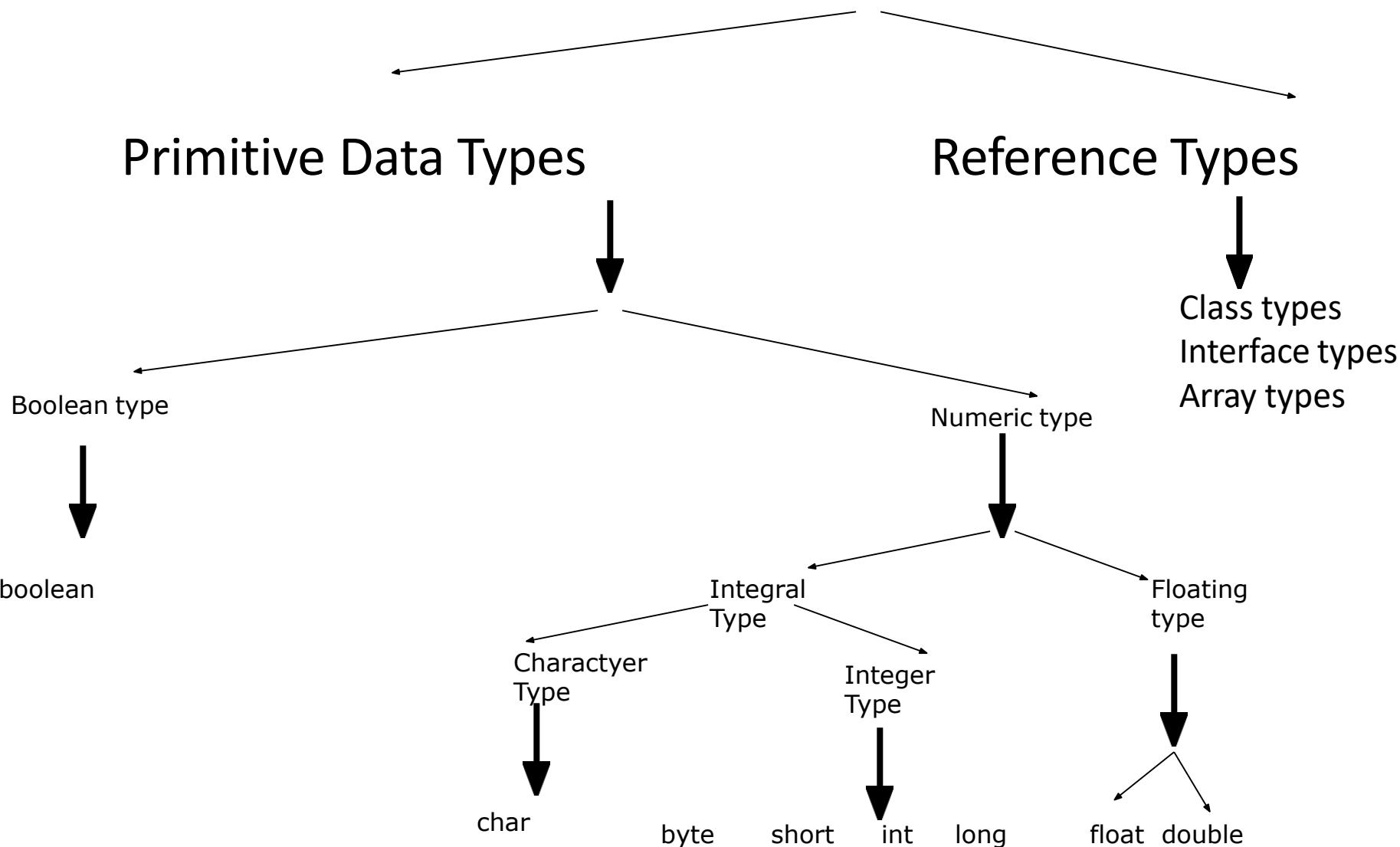


Entry point method

- 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 .



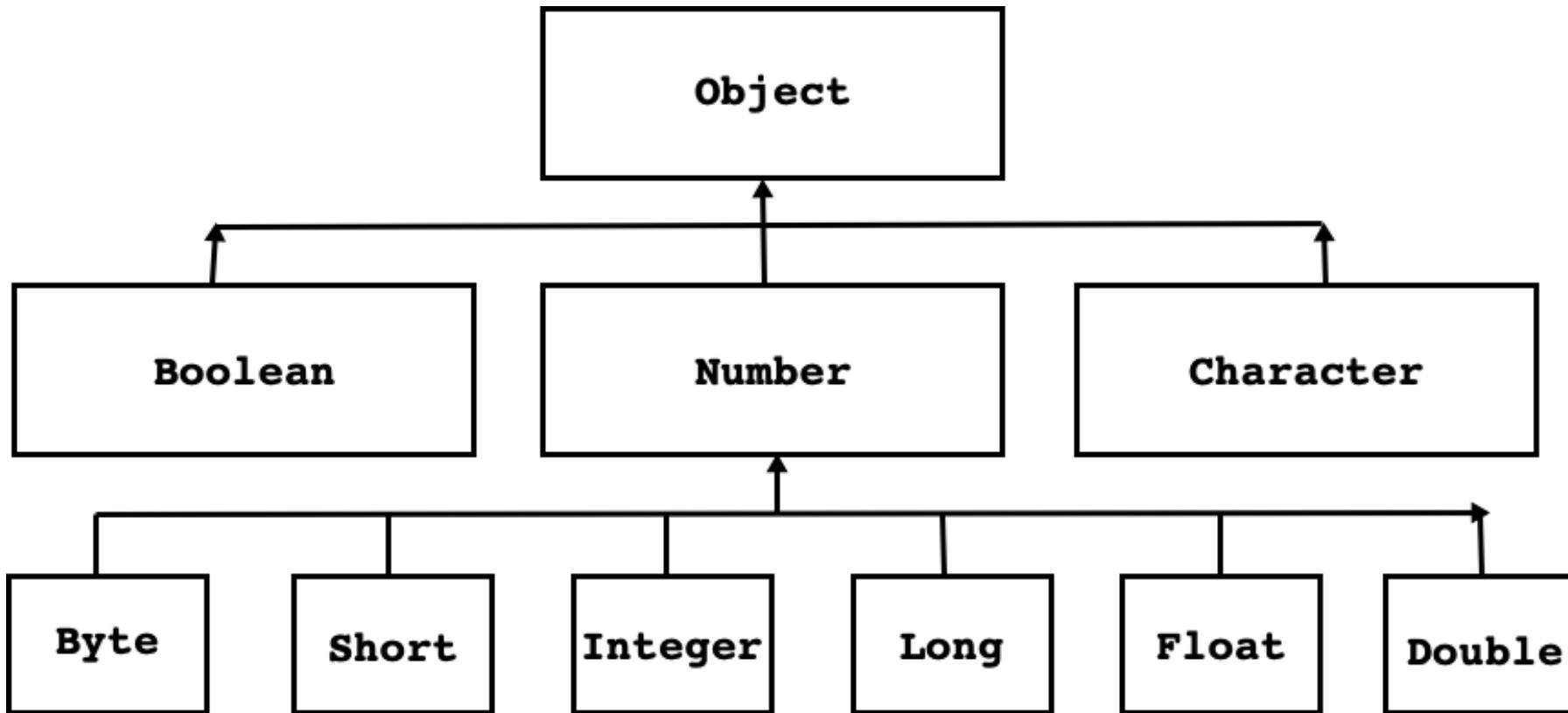
Wrapper class

- 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

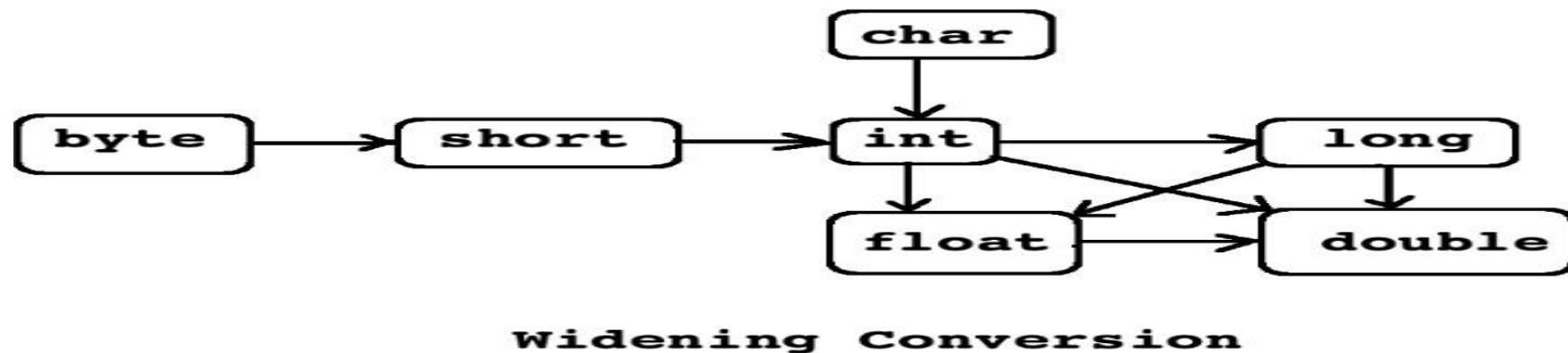
- Process of converting value of variable of narrower type into wider type is called widening.
- E.g. Converting int to double
-

```
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.

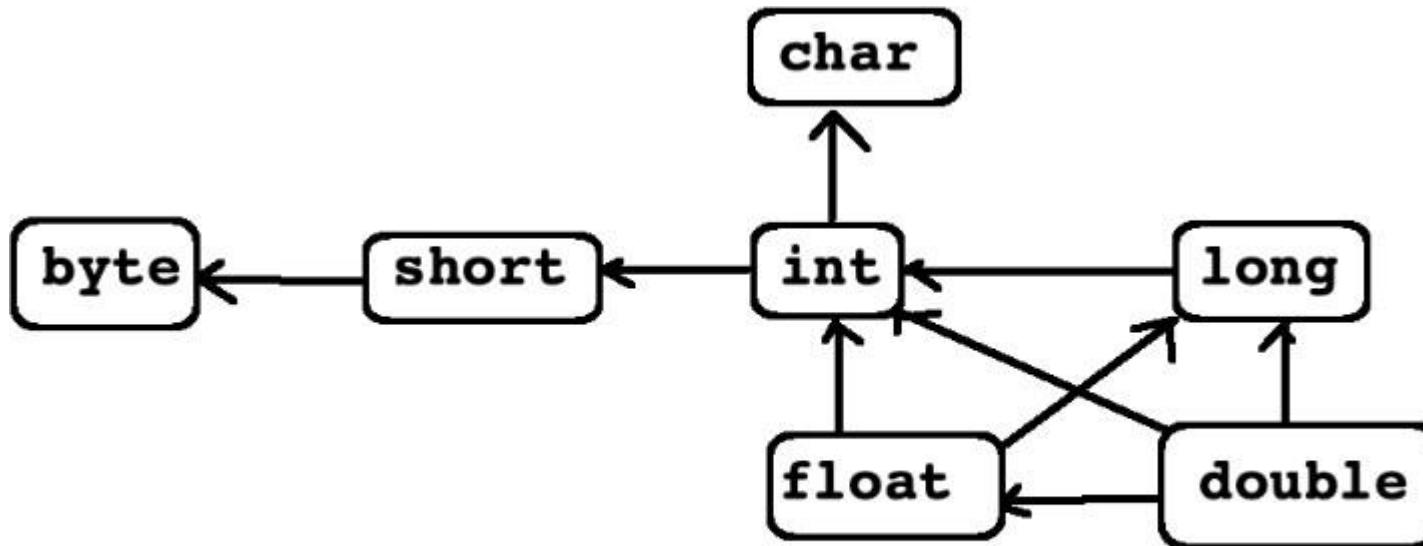
```
public static void main(String[] args) {  
    double num1 = 10.5;  
  
    int num2 = ( int )num1; //Narrowing : OK  
    //int num2 = num1; //Narrowing : NOT OK  
  
    System.out.println("Num2      :      "+num2);  
}
```

- 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





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**.

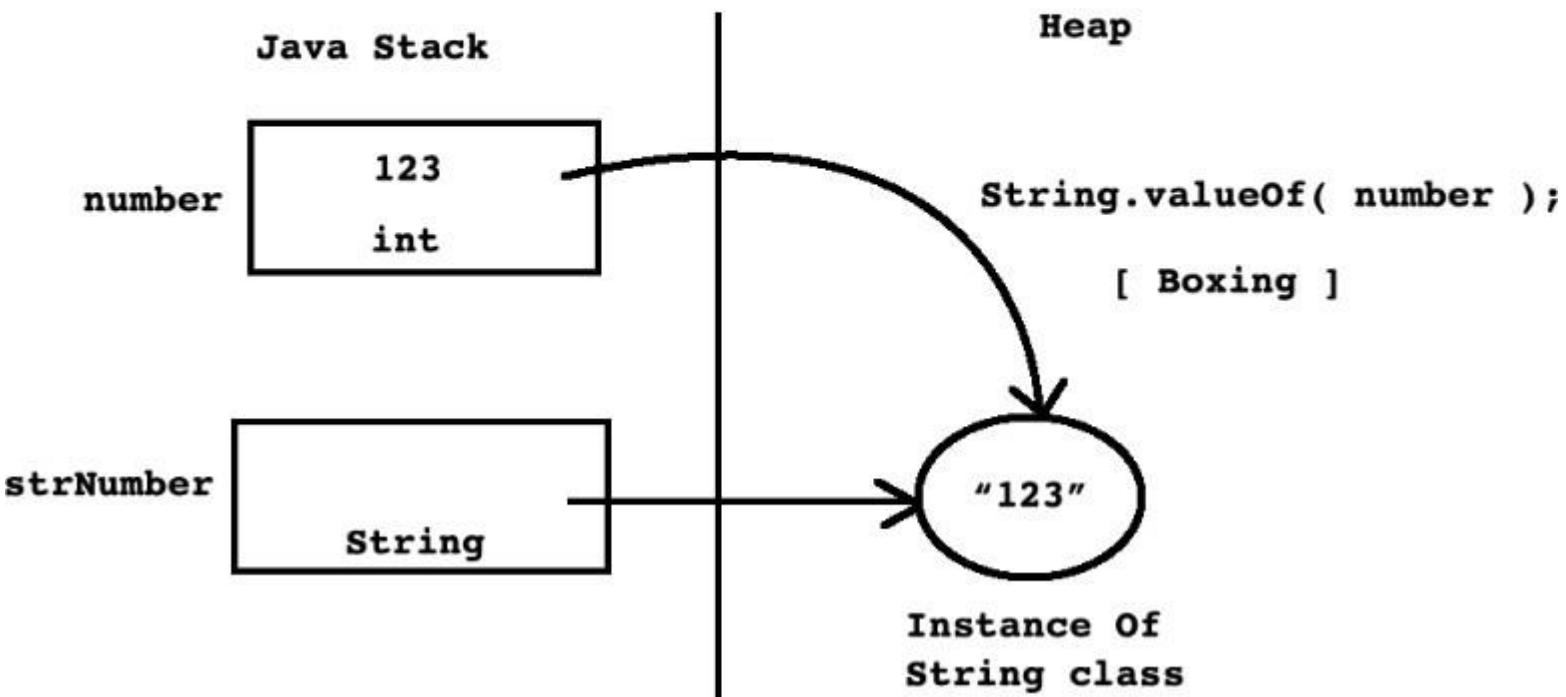
```
public static void main(String[] args) {  
    int number = 123;  
    //String str = Integer.toString( number );      //Boxing : OK  
    String str = String.valueOf(number);           //Boxing : OK  
    System.out.println("Str : " +str);  
}
```

- 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);  
}
```

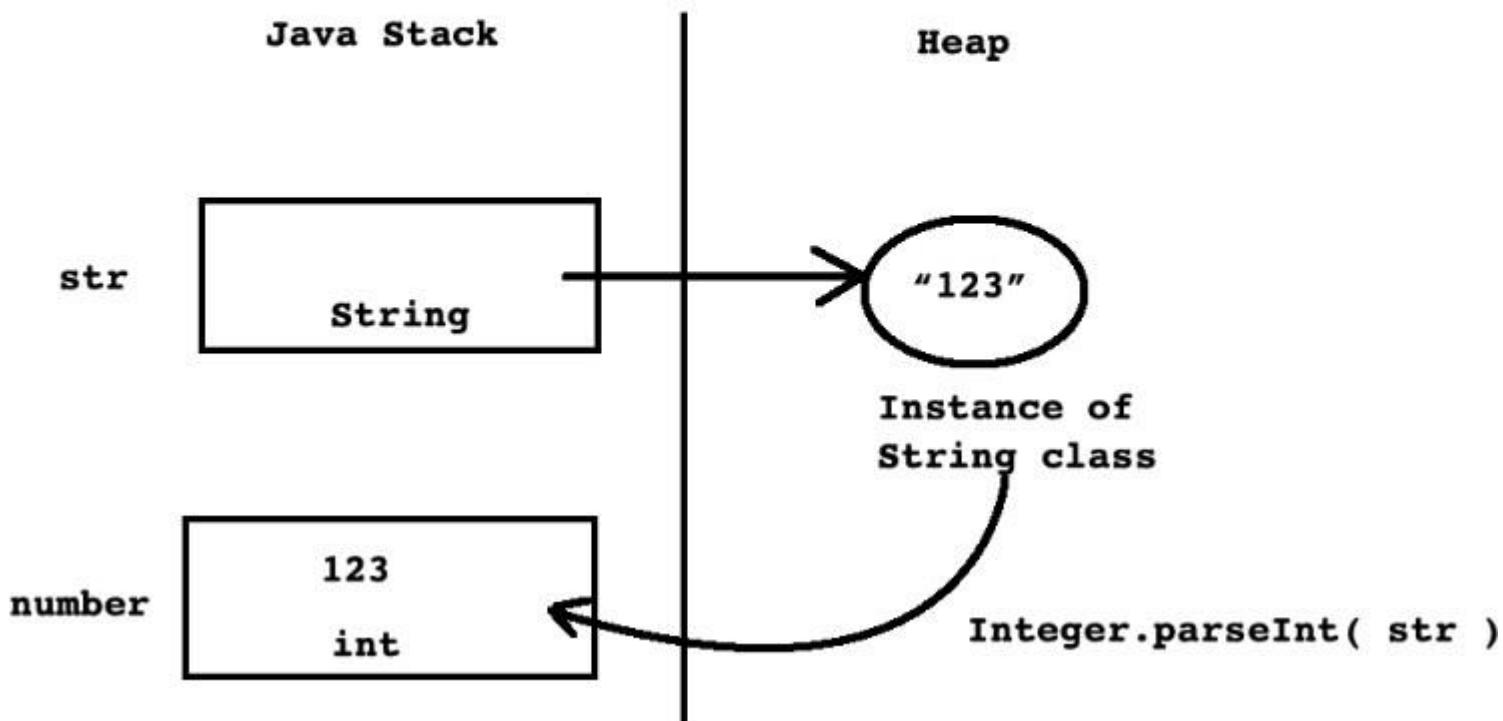
- If string does not contain parseable numeric value then **parseXXX()** method throws **NumberFormatException**.

```
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.

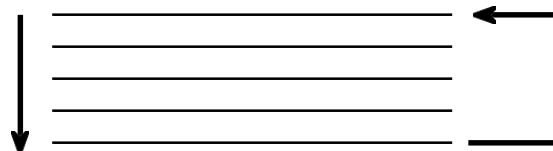
```
if  
(expression)  
    statement;  
else  
{  
    statements;  
}
```

→ 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





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.

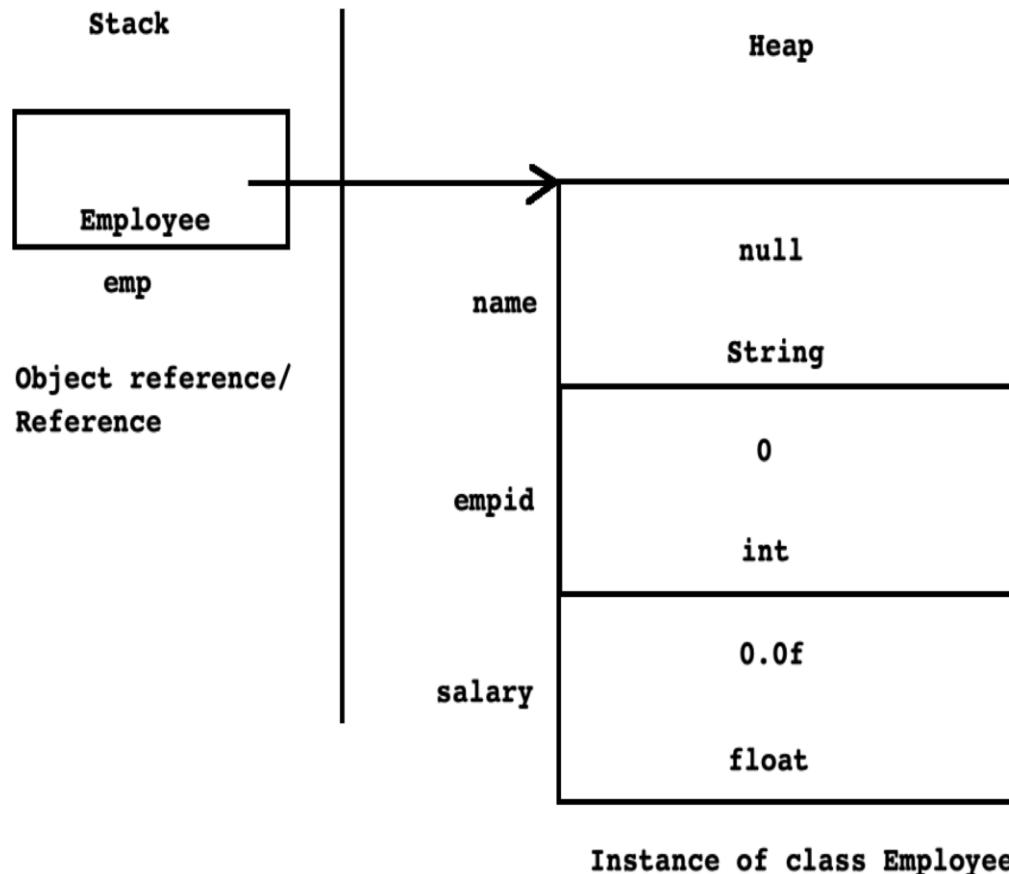


Instantiation

- 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.**

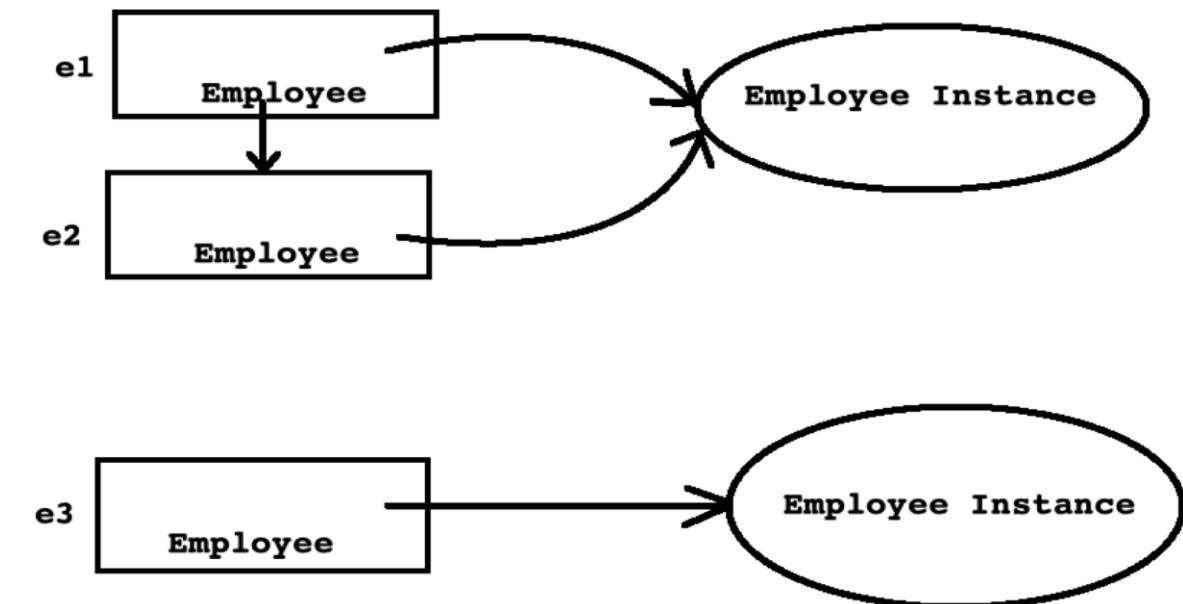


Instantiation



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



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.



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





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

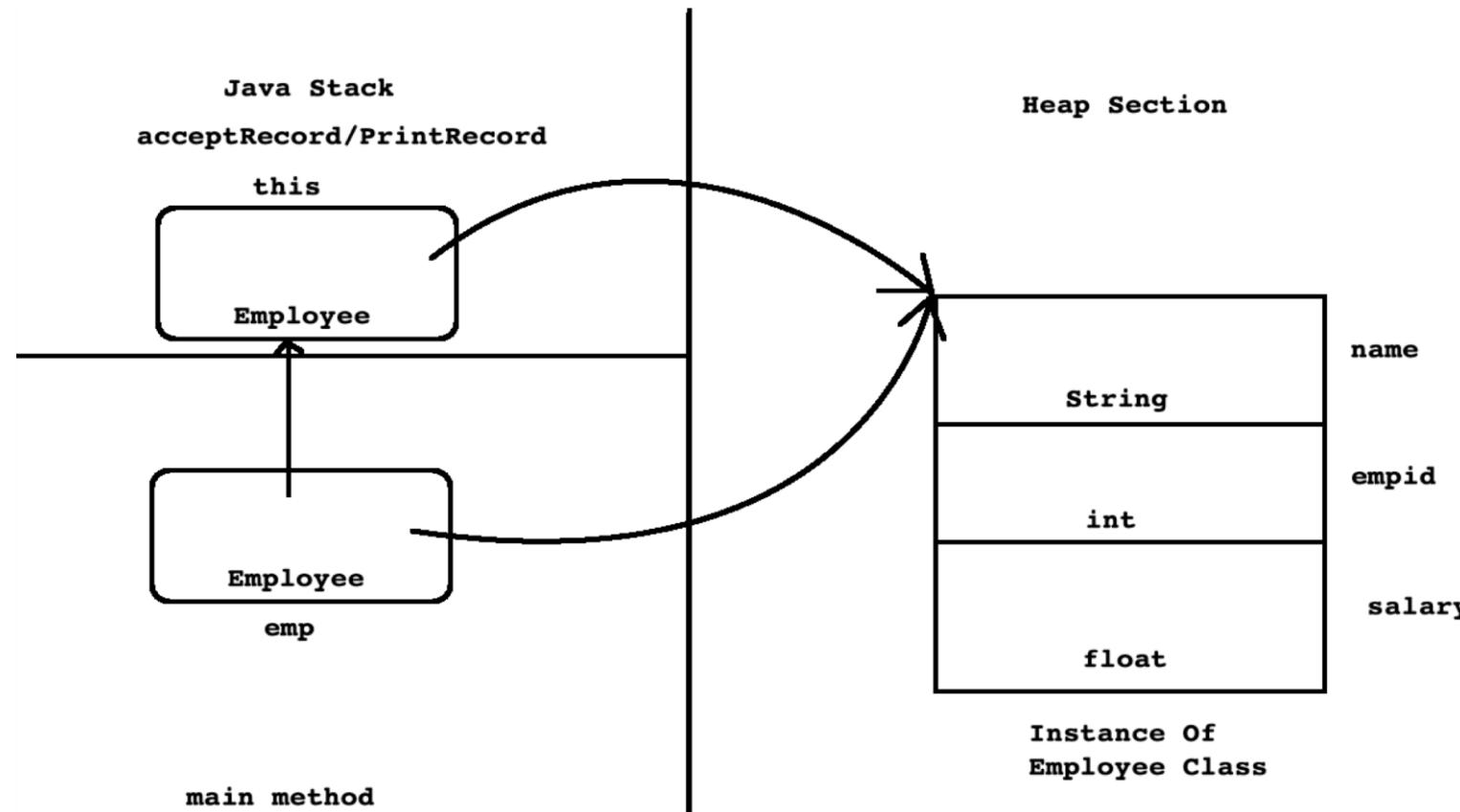
- If we call non static method on instance(actually object reference) then compiler implicitly pass, reference of current/calling instance as a argument to the method implicitly. To store reference of current/calling instance, compiler implicitly declare one reference as a parameter inside method. It 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);  
    }  
}
```

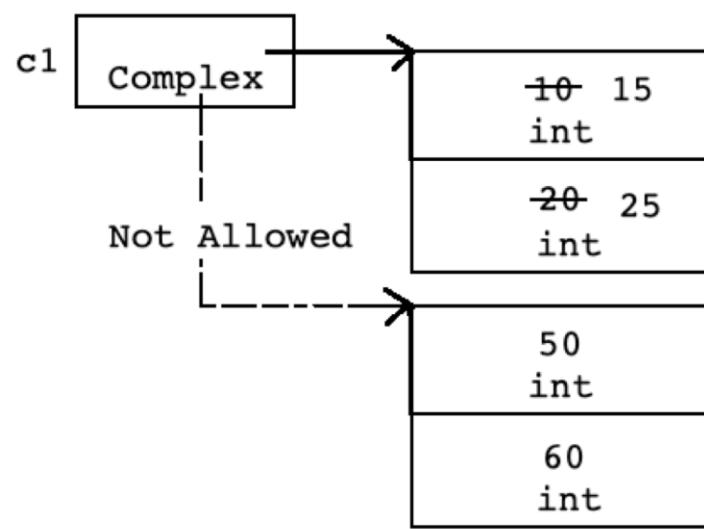
- If we want to declare any field final then we should declare it static also.



Final Reference Variable

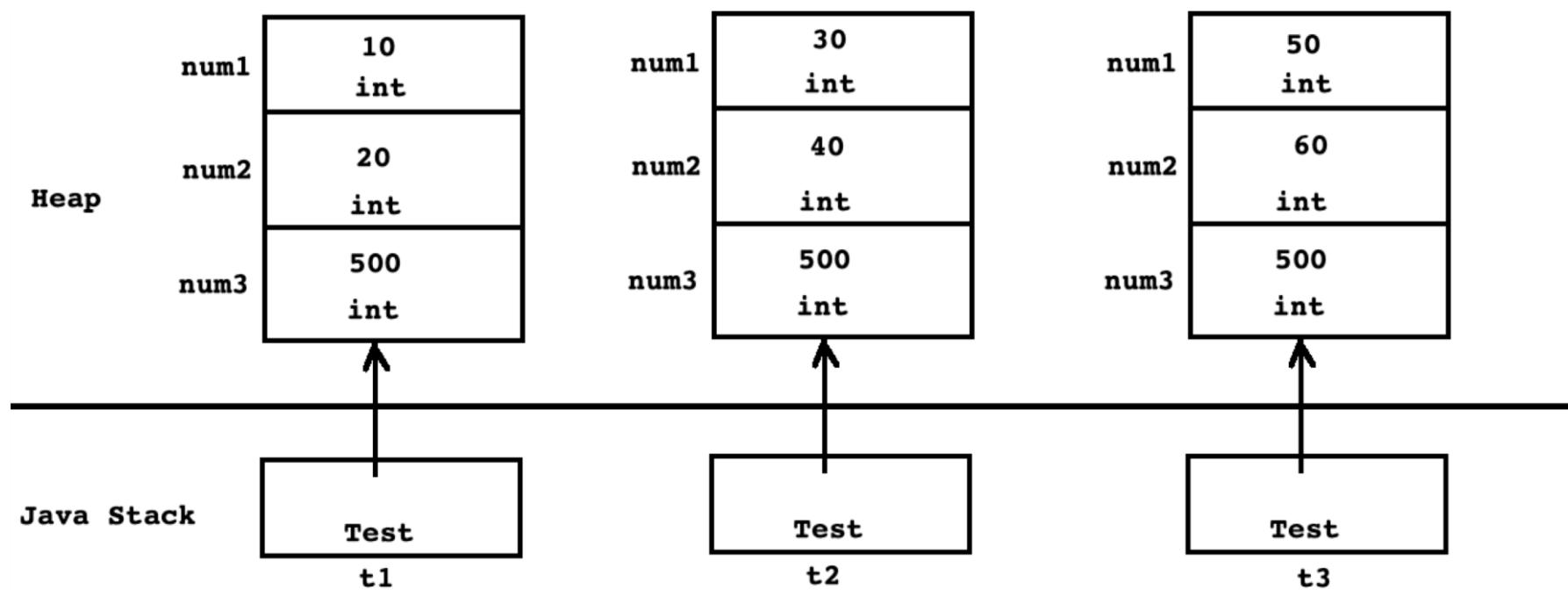
- In Java, we can declare reference final but we can not declare instance final.

```
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  
}
```

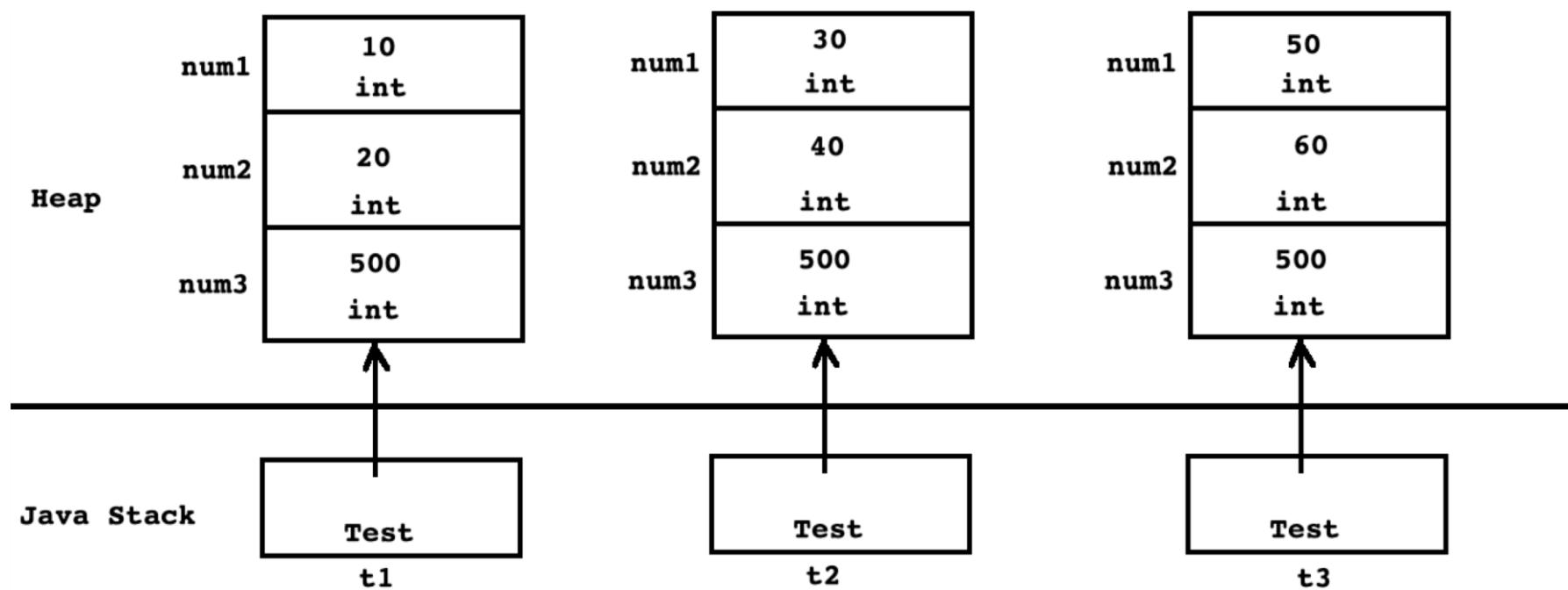


- We can declare method final. It is not allowed to override final method in sub class.
- We can declare class final. It is not allowed to extend final class.

Static Field

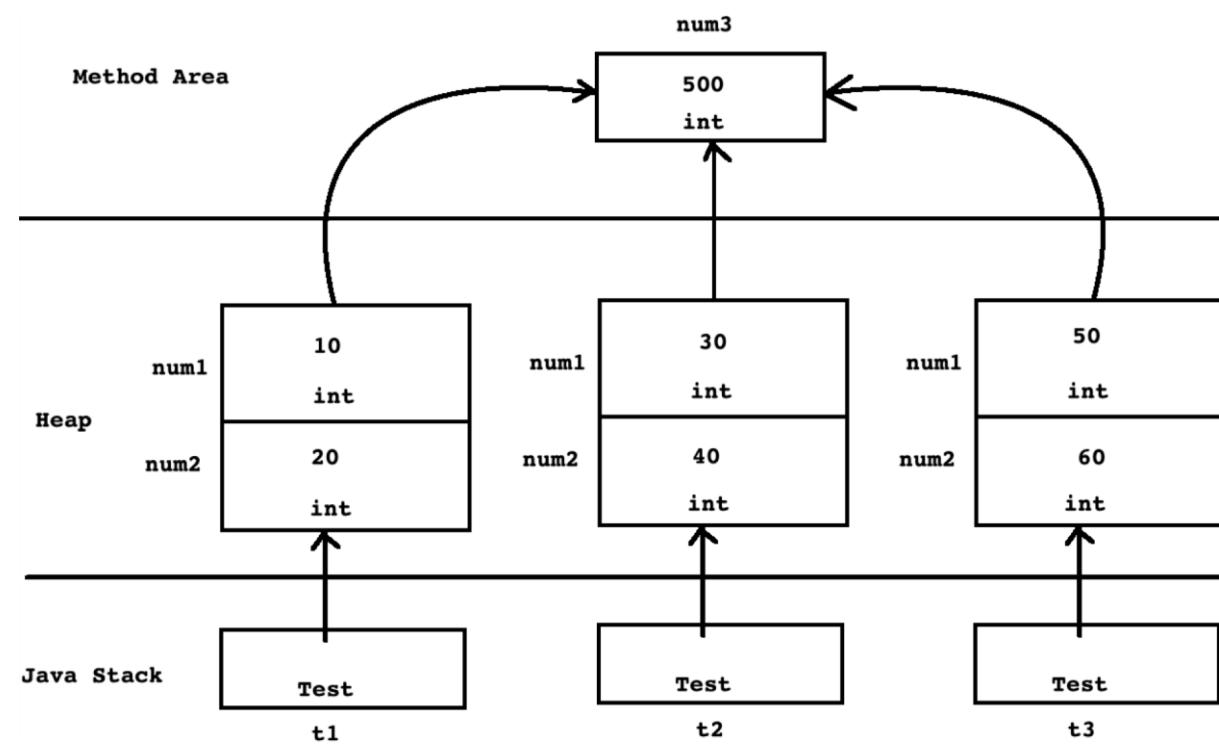


Static Field



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
```

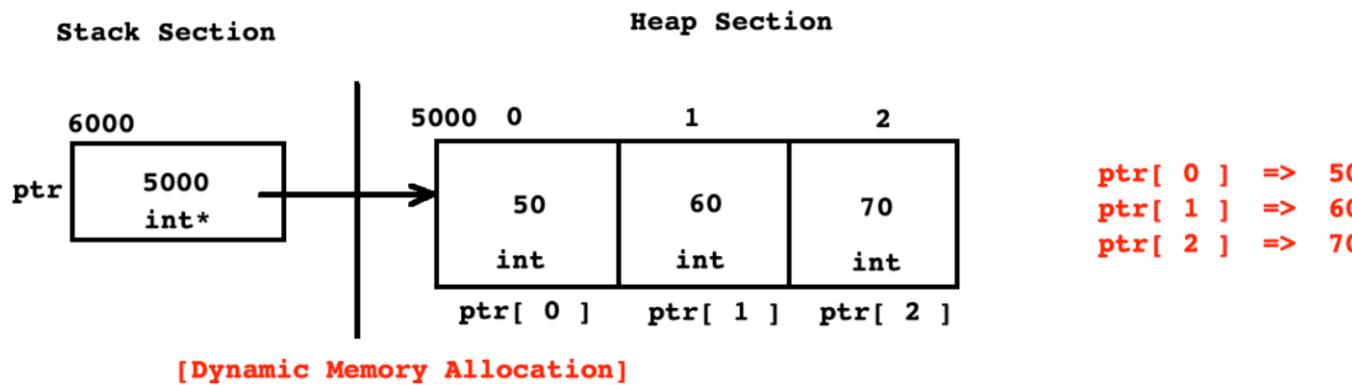
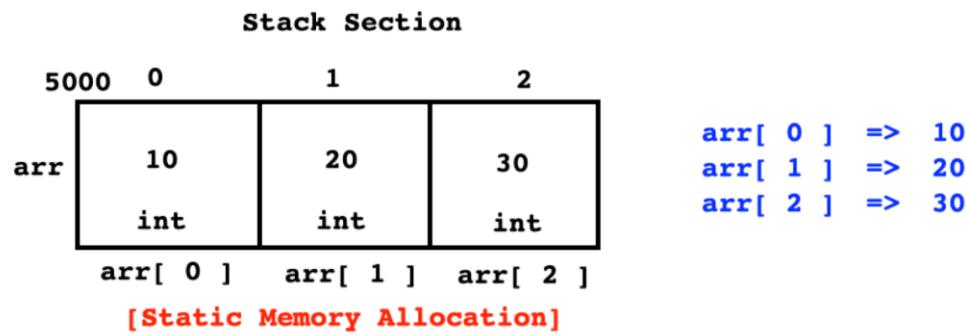
In C/C++

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 ];
```

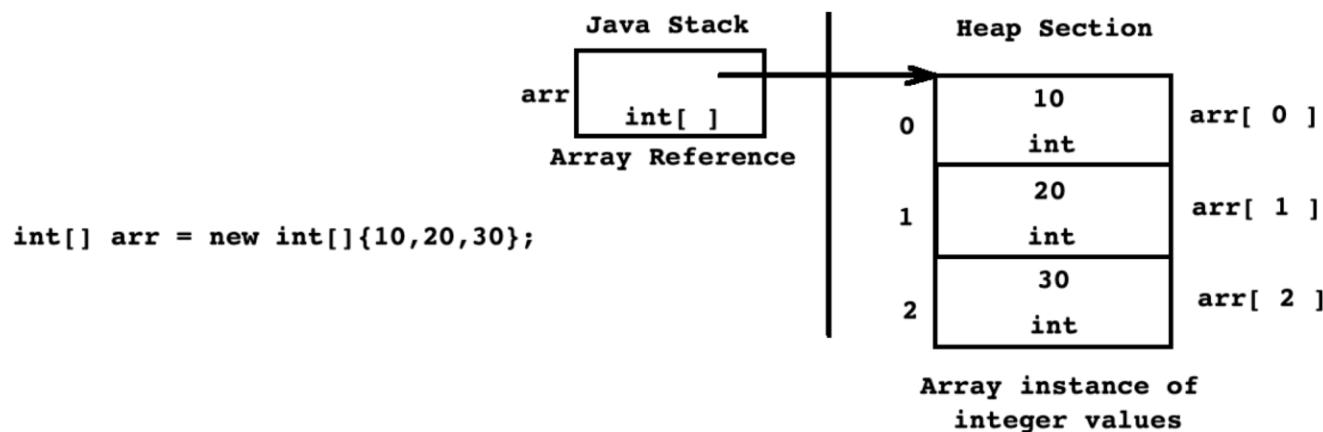
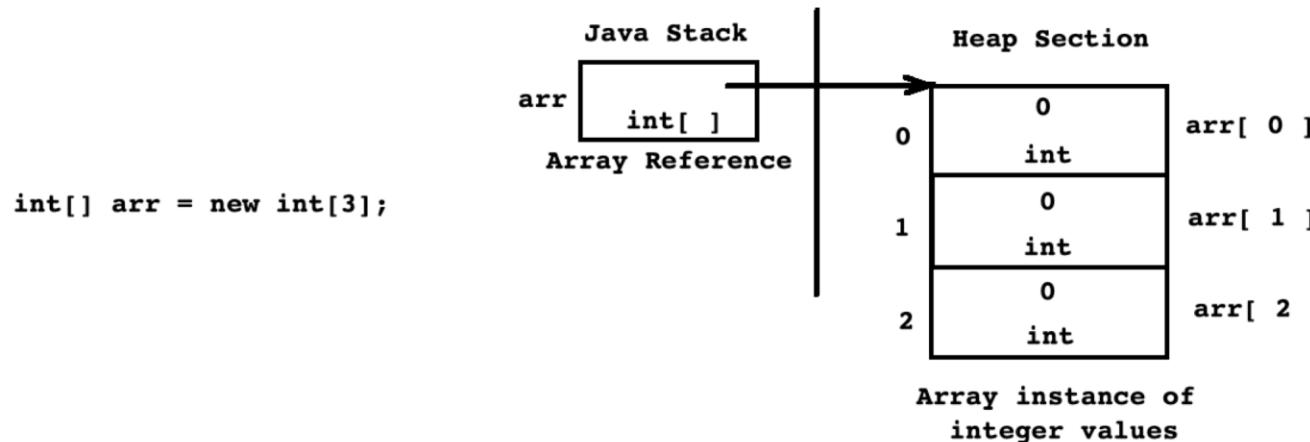
```
int[] arr1 = new int[ -3 ]; //NegativeArraySizeException  
//or  
int size = -3;  
int[] arr2 = new int[ size ]; //NegativeArraySizeException
```

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
```

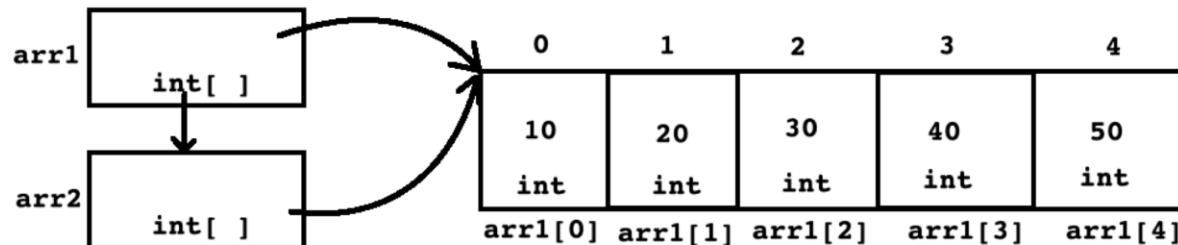
Implementation of Arrays.copyOf method:

```
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;
}
```

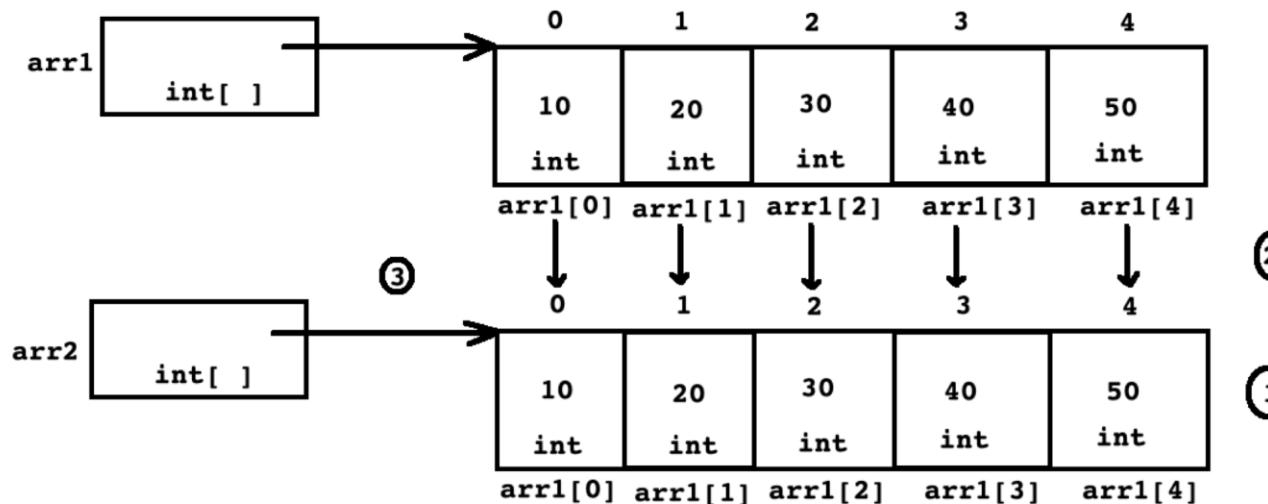


Reference Copy and Instance Copy

```
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
```



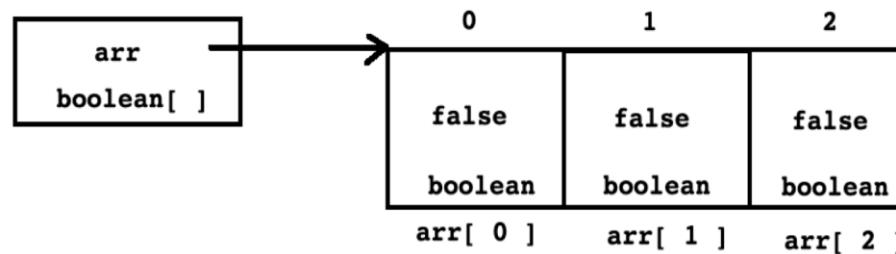
Array Of Primitive Values

```
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  
    }  
}
```

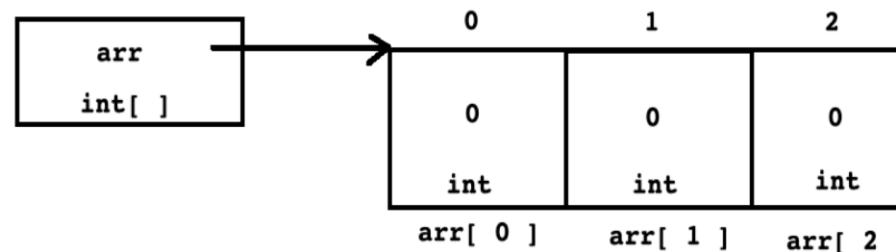


Array Of Primitive Values

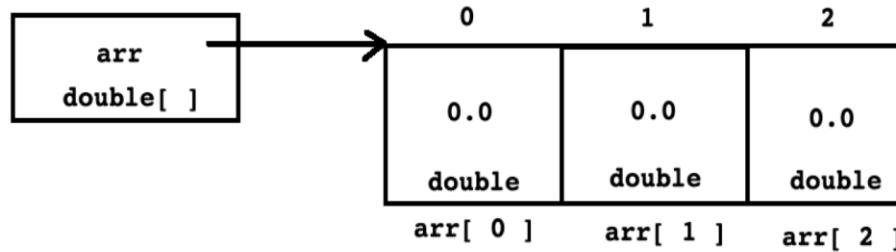
```
boolean[] arr = new boolean[3];
```



```
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

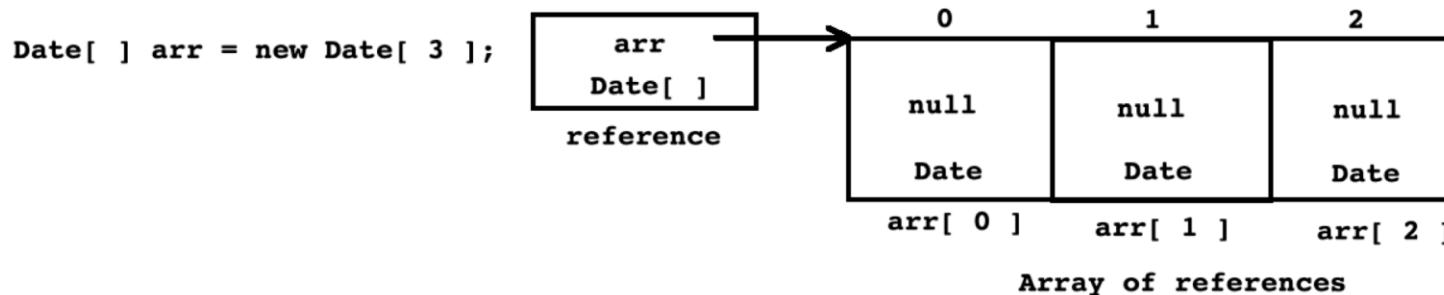
```
public class Program {  
    public static void main(String[] args) {  
        Date[] arr = new Date[ 3 ]; //Contains all null  
    }  
}
```

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

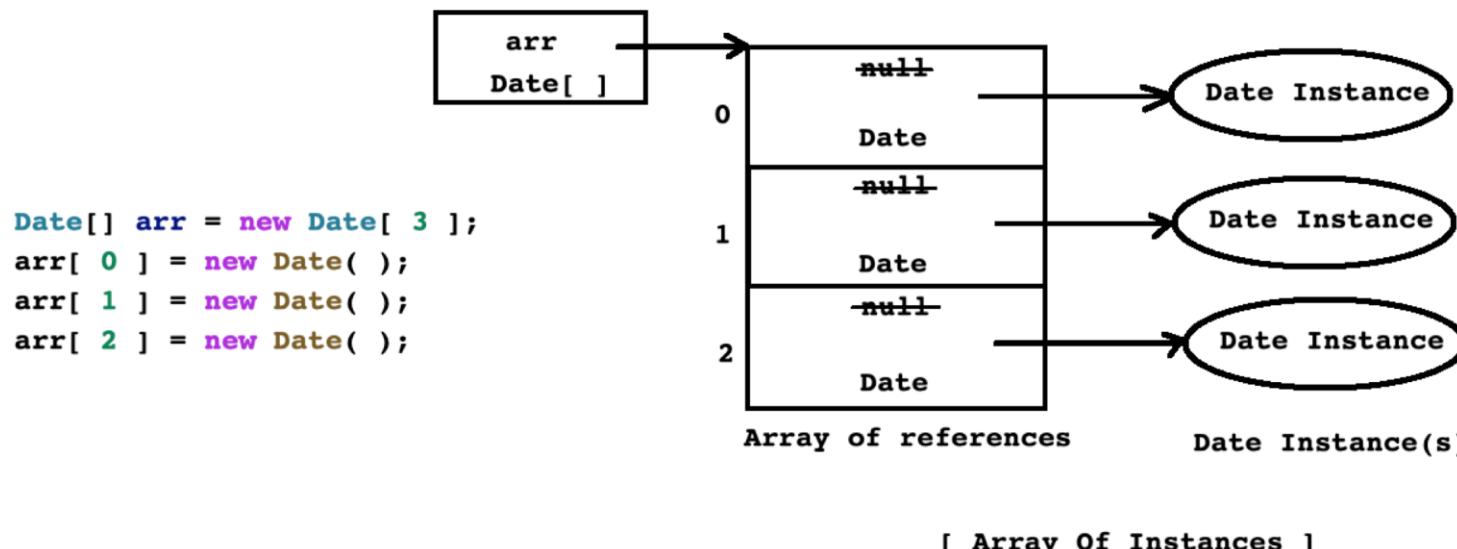
```
public class Program {  
    public static void main(String[] args) {  
        Date[] arr = new Date[ 3 ];  
        arr[ 0 ] = new Date( );  
        arr[ 1 ] = new Date( );  
        arr[ 2 ] = new Date( );  
    }  
    //or  
    public static void main(String[] args) {  
        Date[] arr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



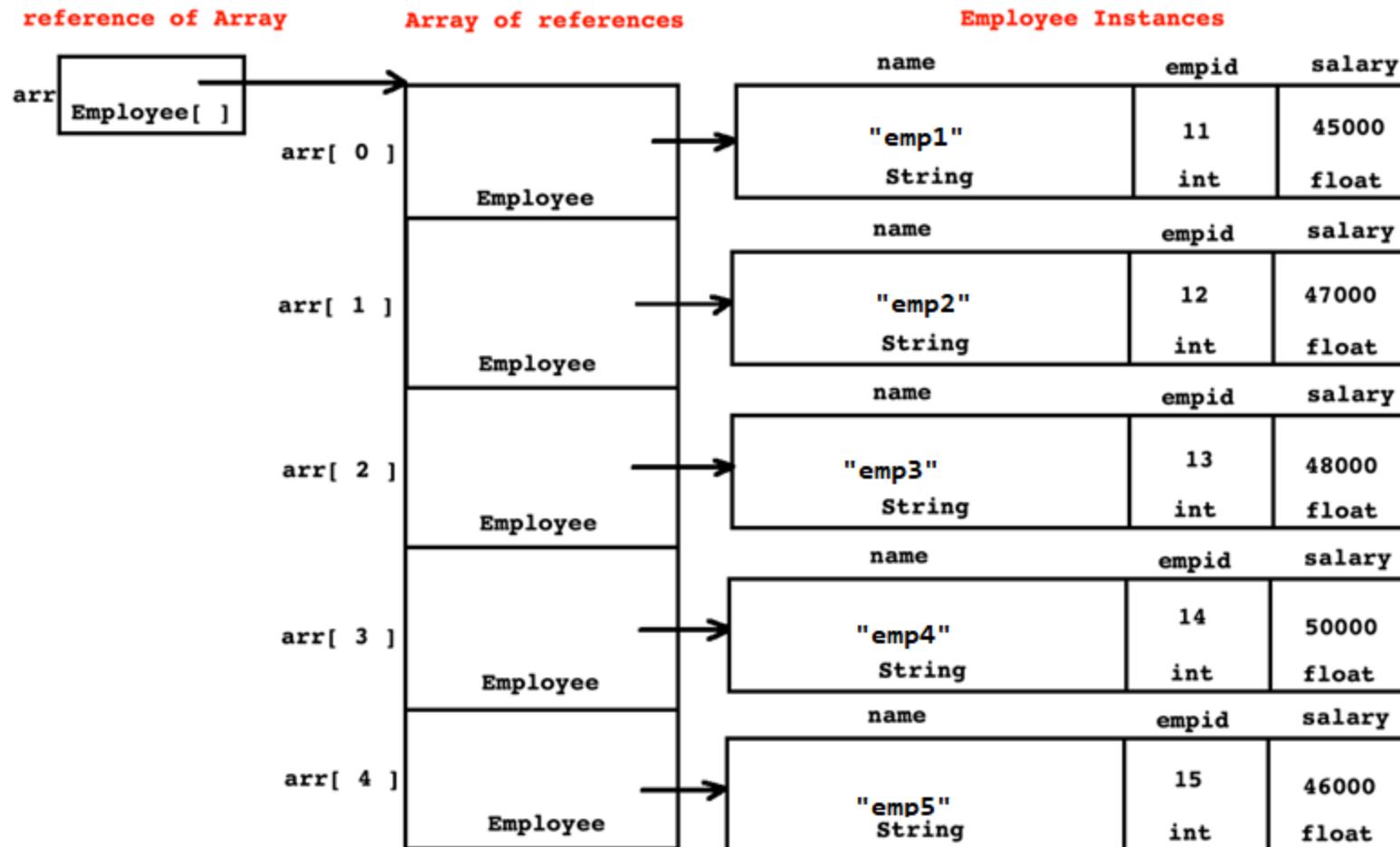
Array Of References and Instances



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 ];
```

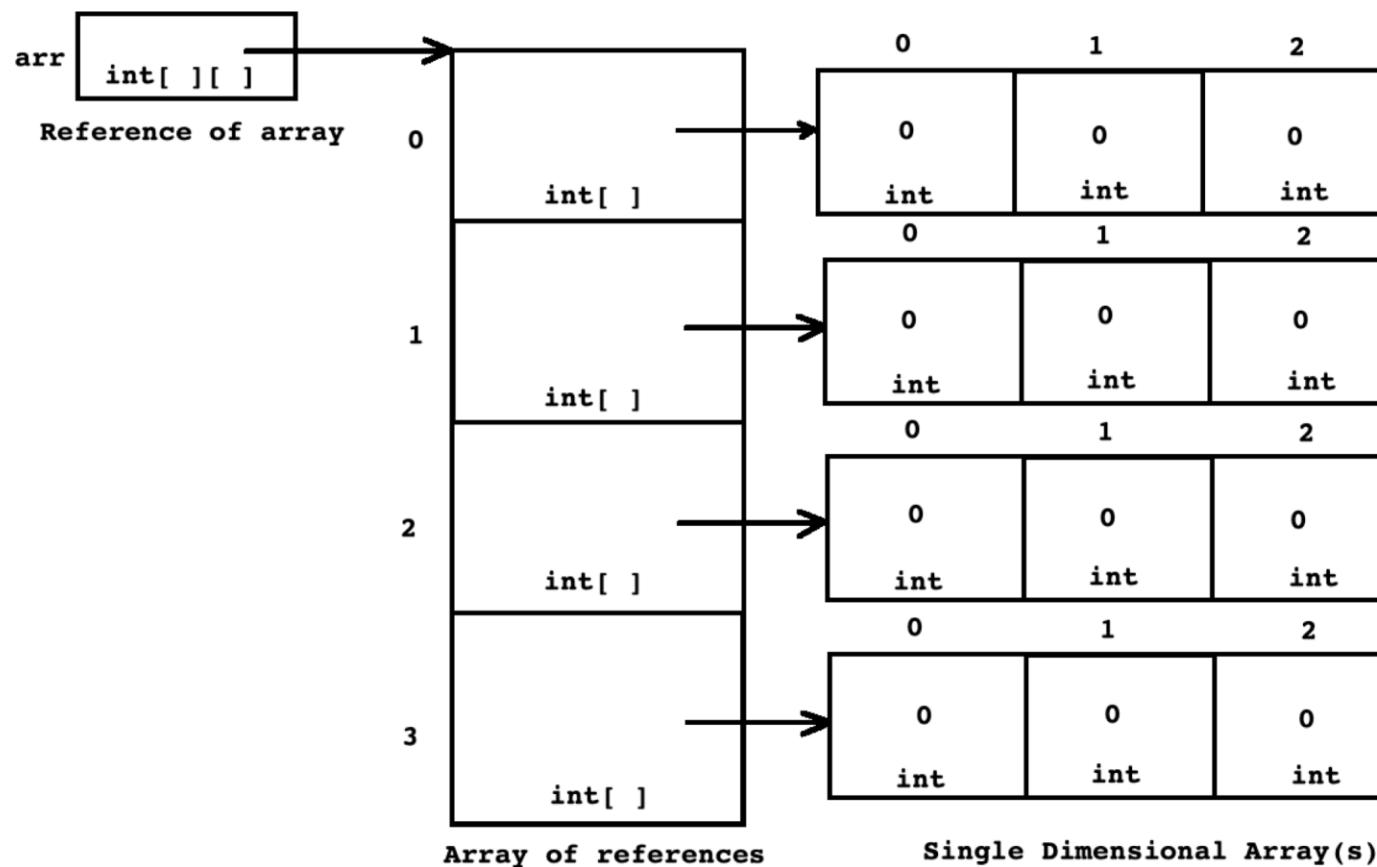
Initialization

```
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

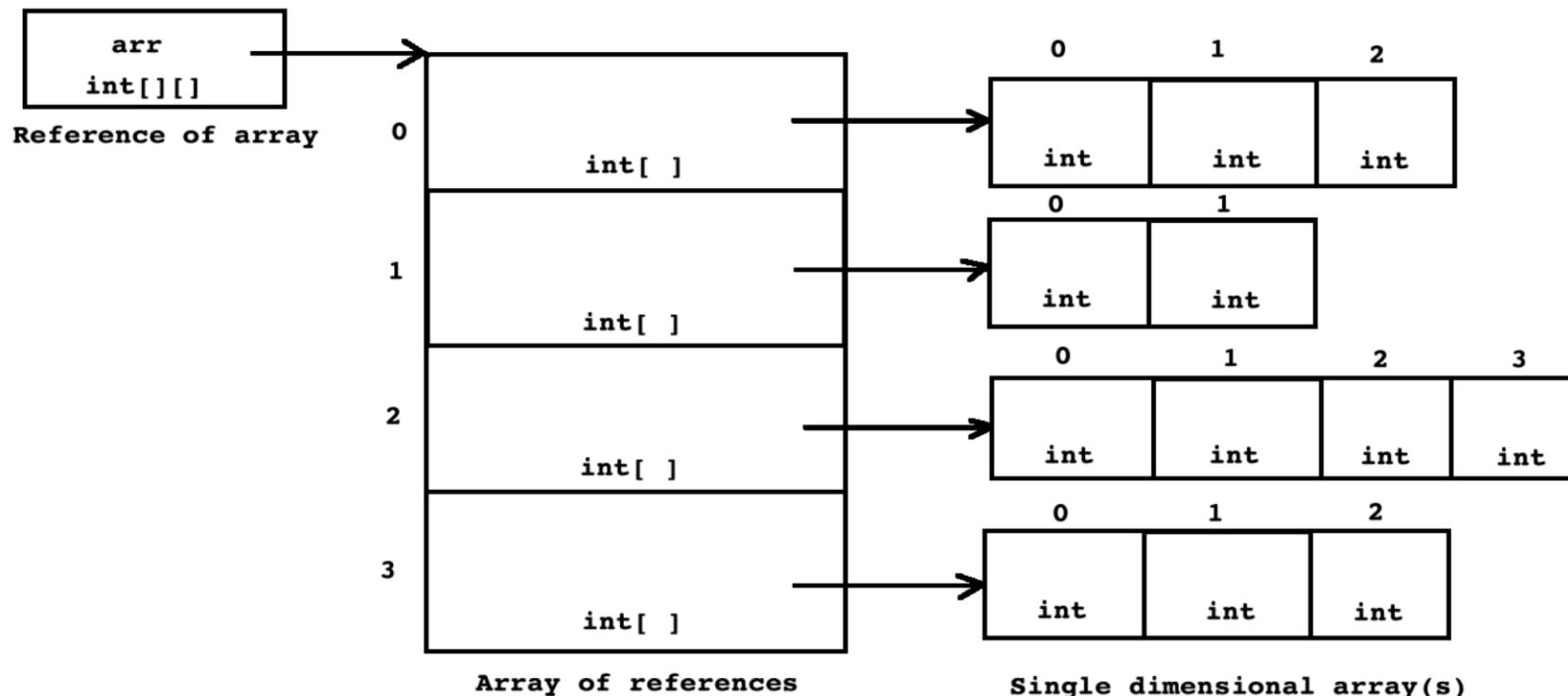
```
int[][] arr = new int[3][];  
arr[ 0 ] = new int[ ]{ 10, 20 };  
arr[ 1 ] = new int[ ]{ 10, 20, 30 };  
arr[ 2 ] = new int[ ]{ 10, 20, 30, 40, 50 };
```

```
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
};

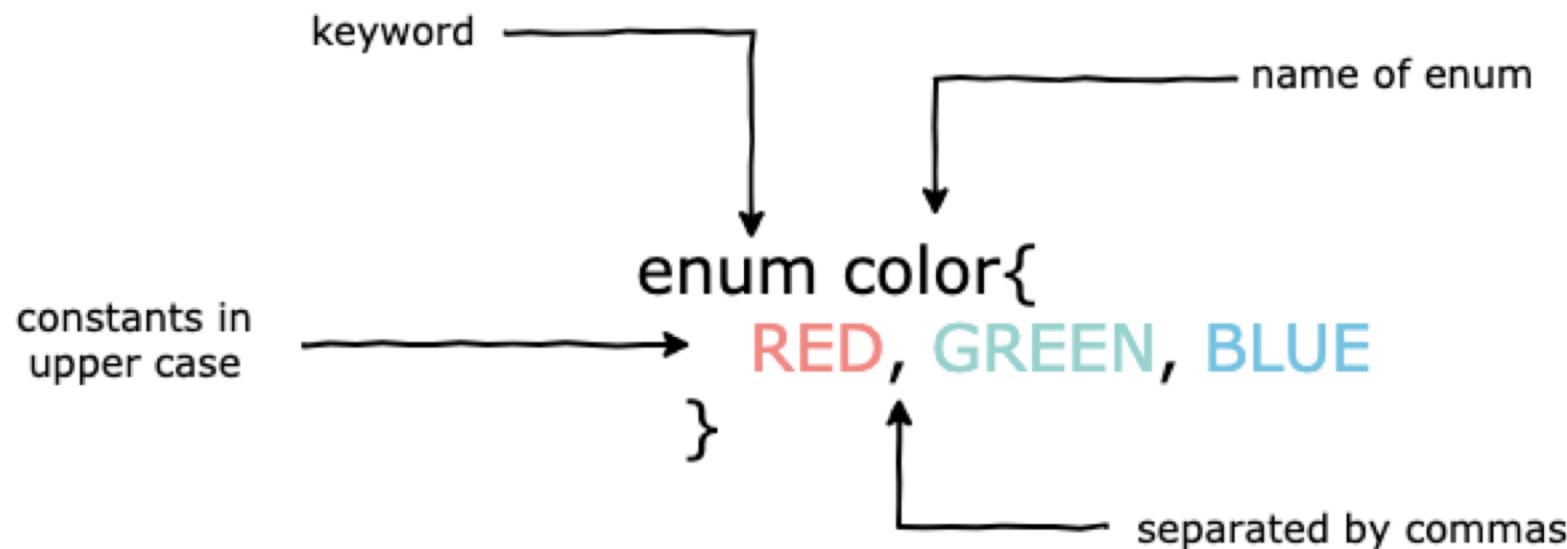
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

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

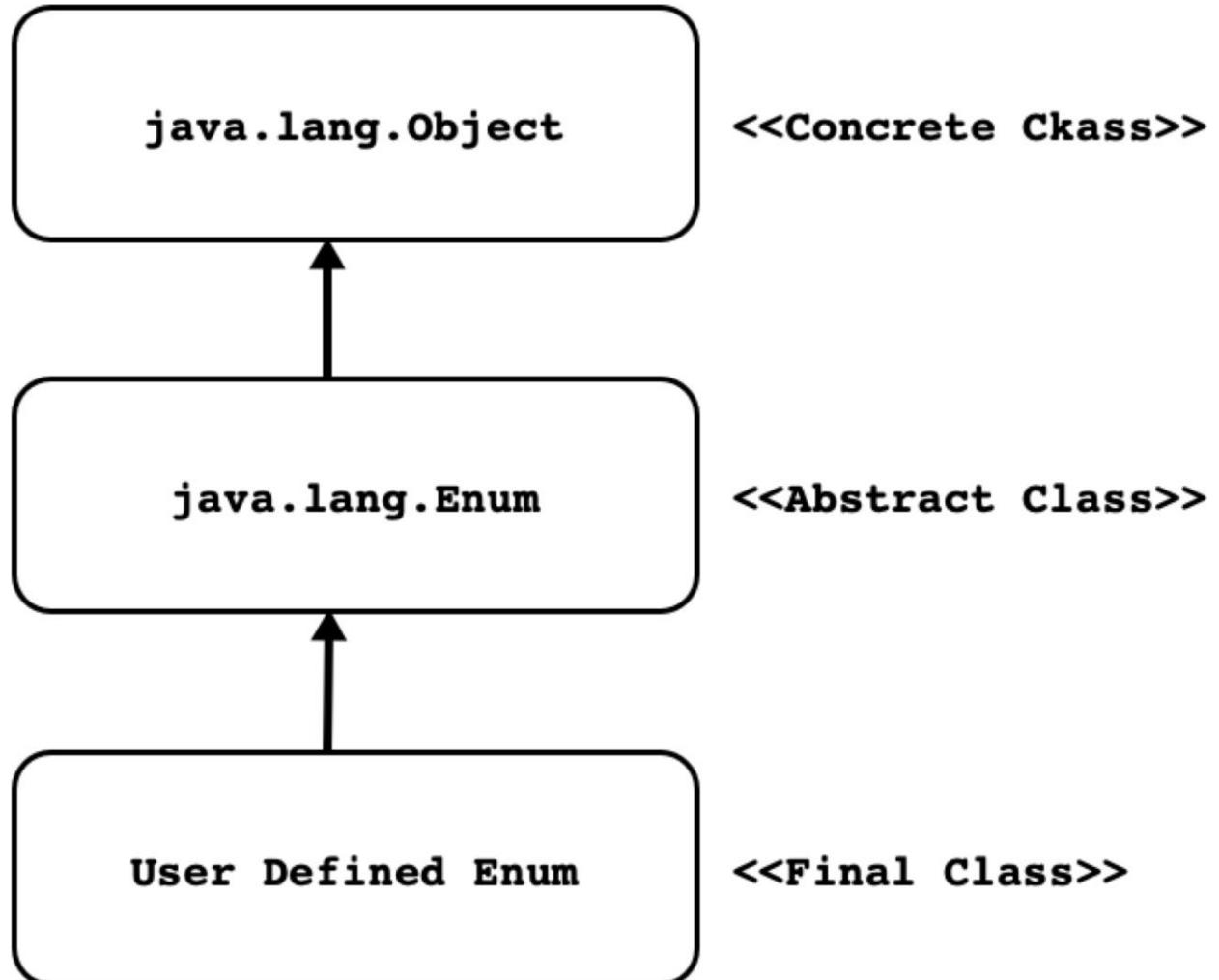


Enum In Java Programming language.

- 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.



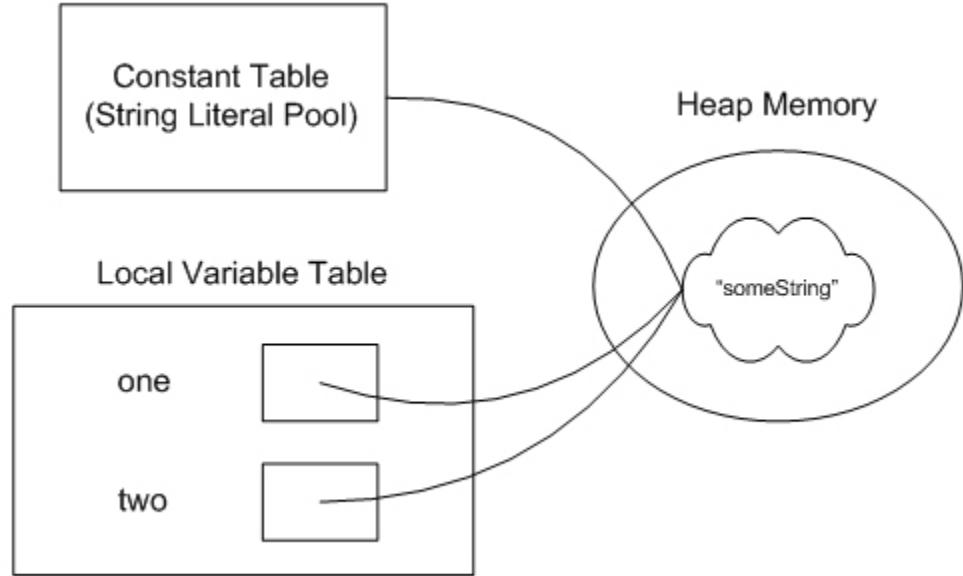
String Introduction

- 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**

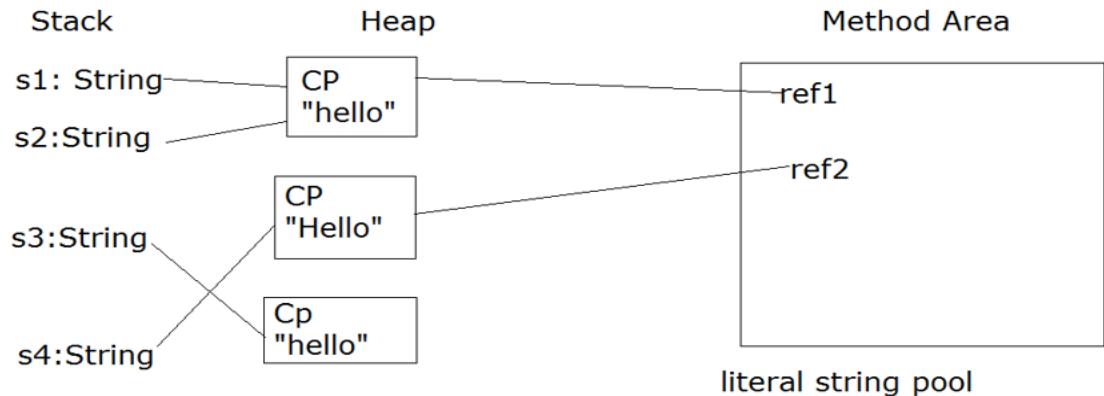


Literal Vs Non Literal

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");`

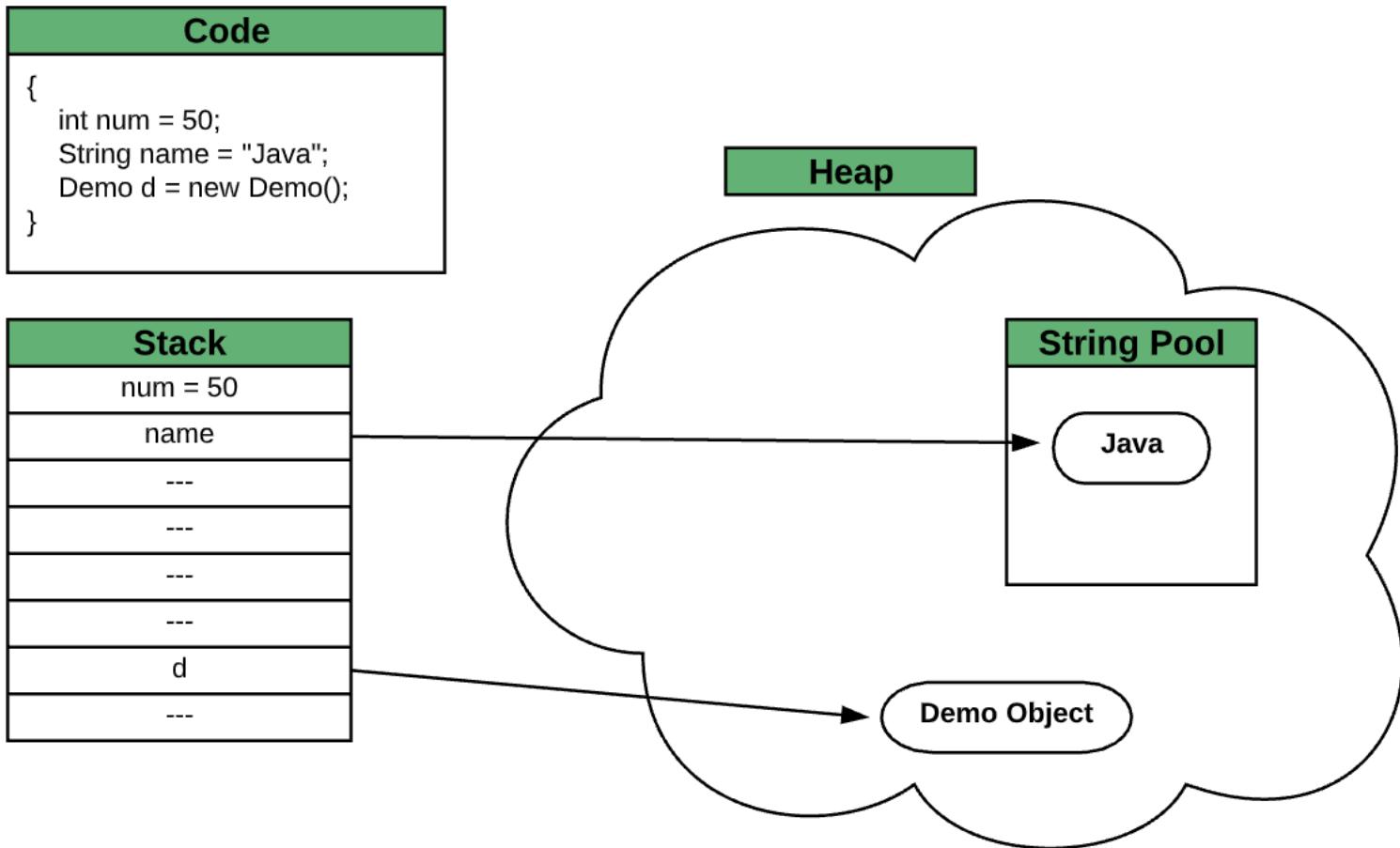
`String s2 = new ("Hello World");`

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.



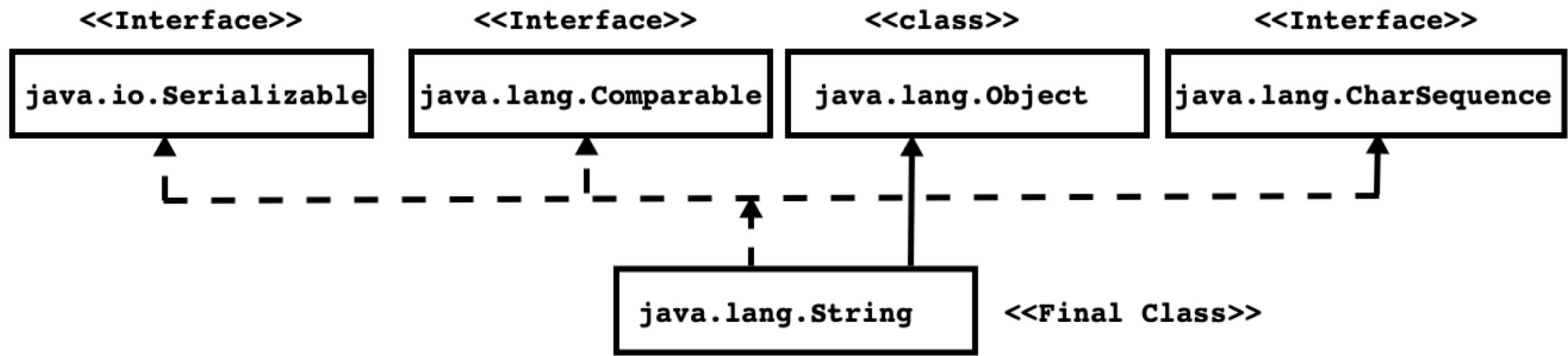


Literal Vs Non Literal example

```
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";
```

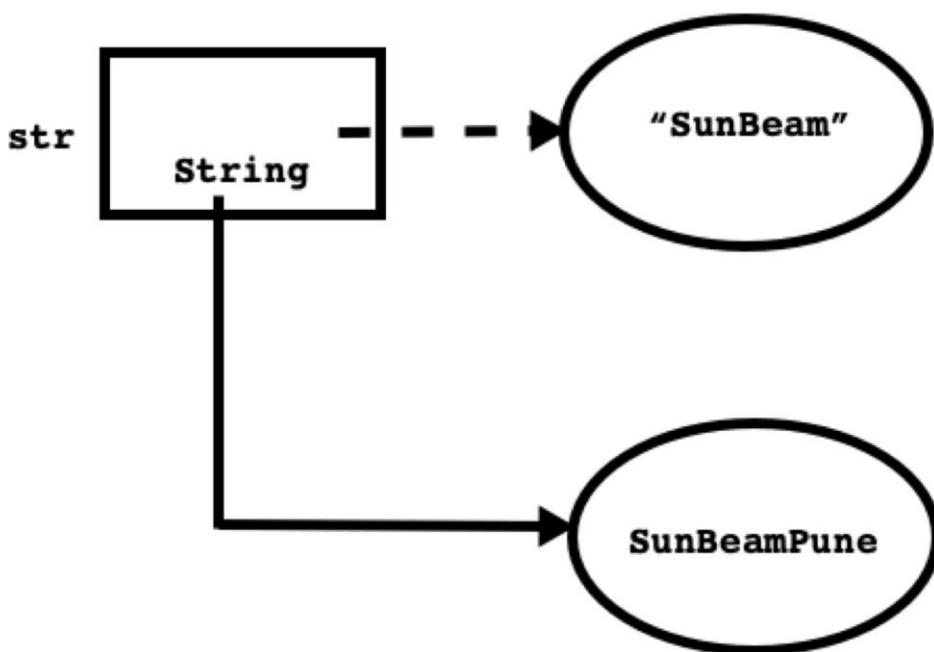
-
-
- int pinCode = 411057;
- String str = "Pune,"+pinCode;



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)
```

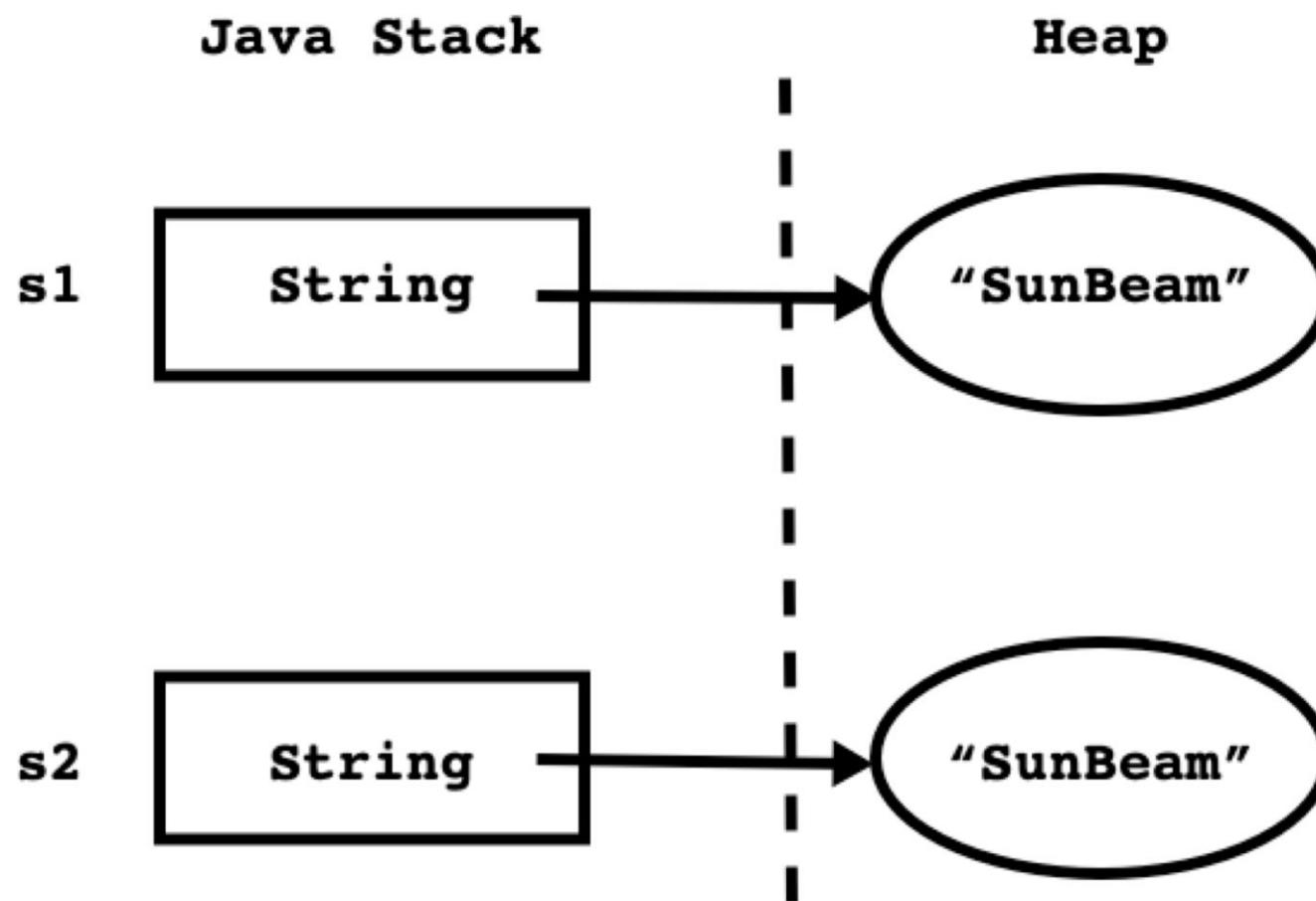


String Twister

```
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  
    }  
}
```

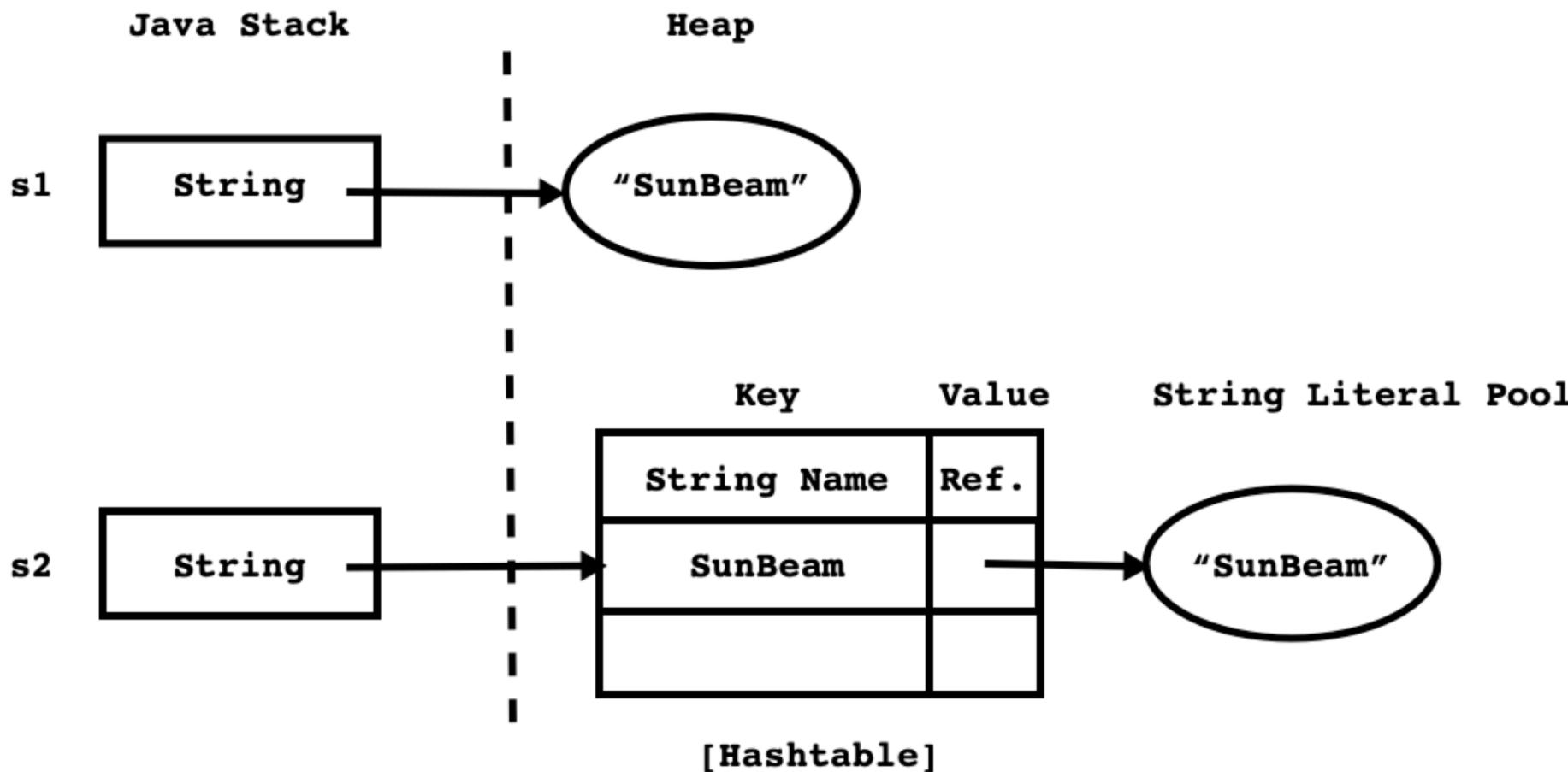


String Twister

```
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  
    }  
}
```

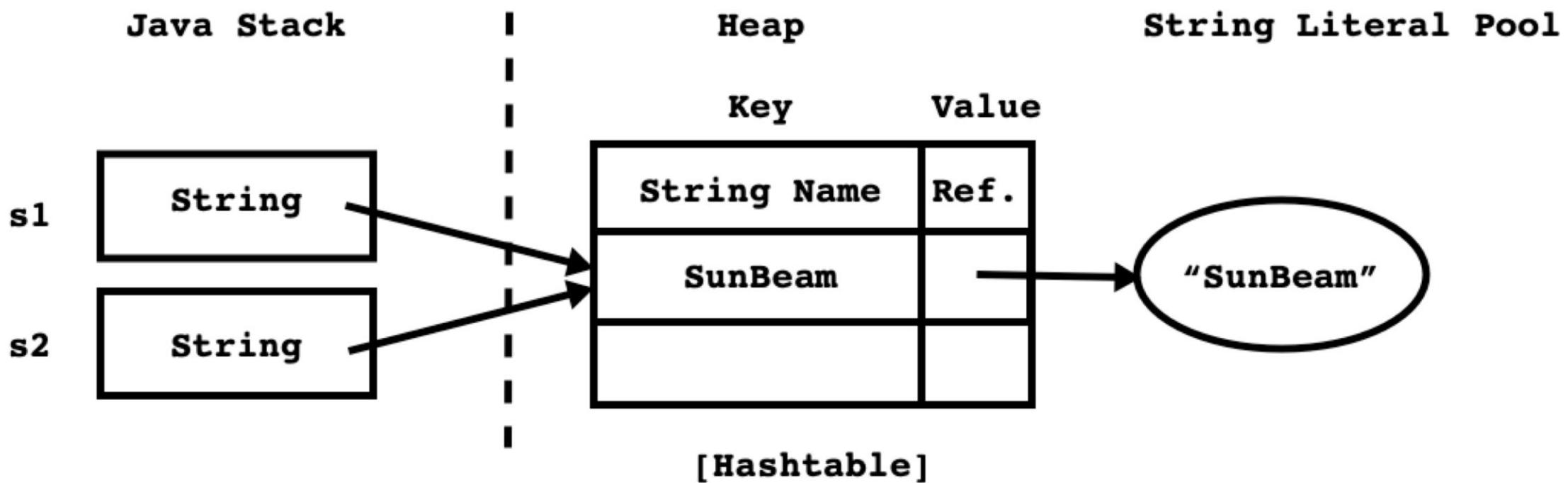


String Twister

```
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  
    }  
}
```



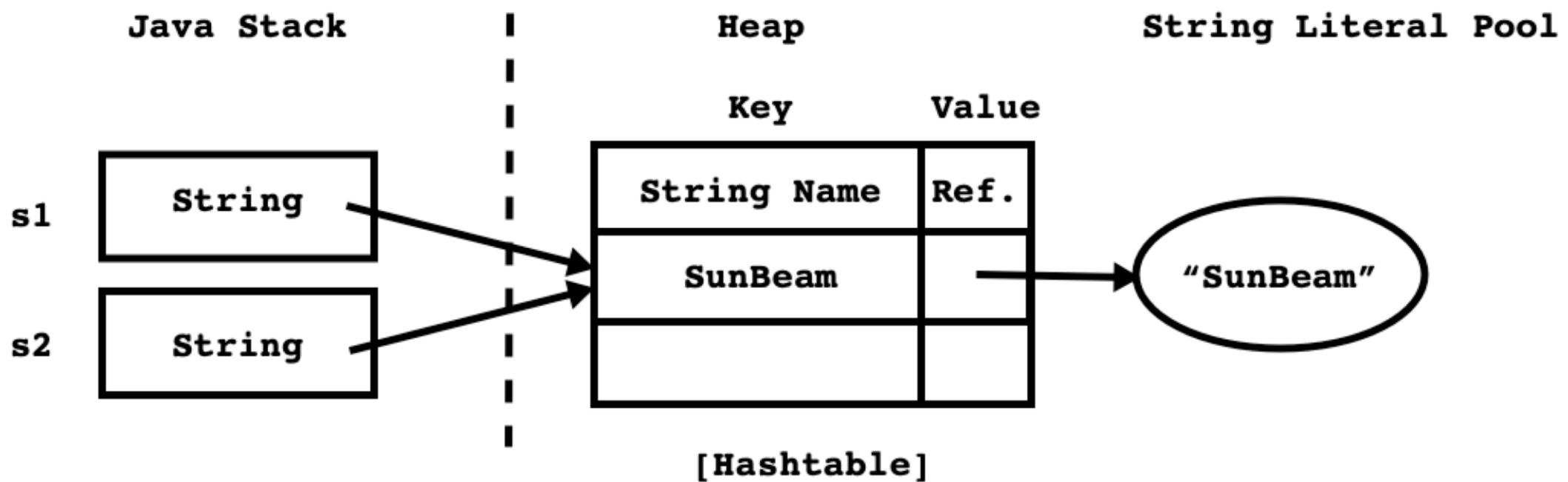
String Twister

```
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.

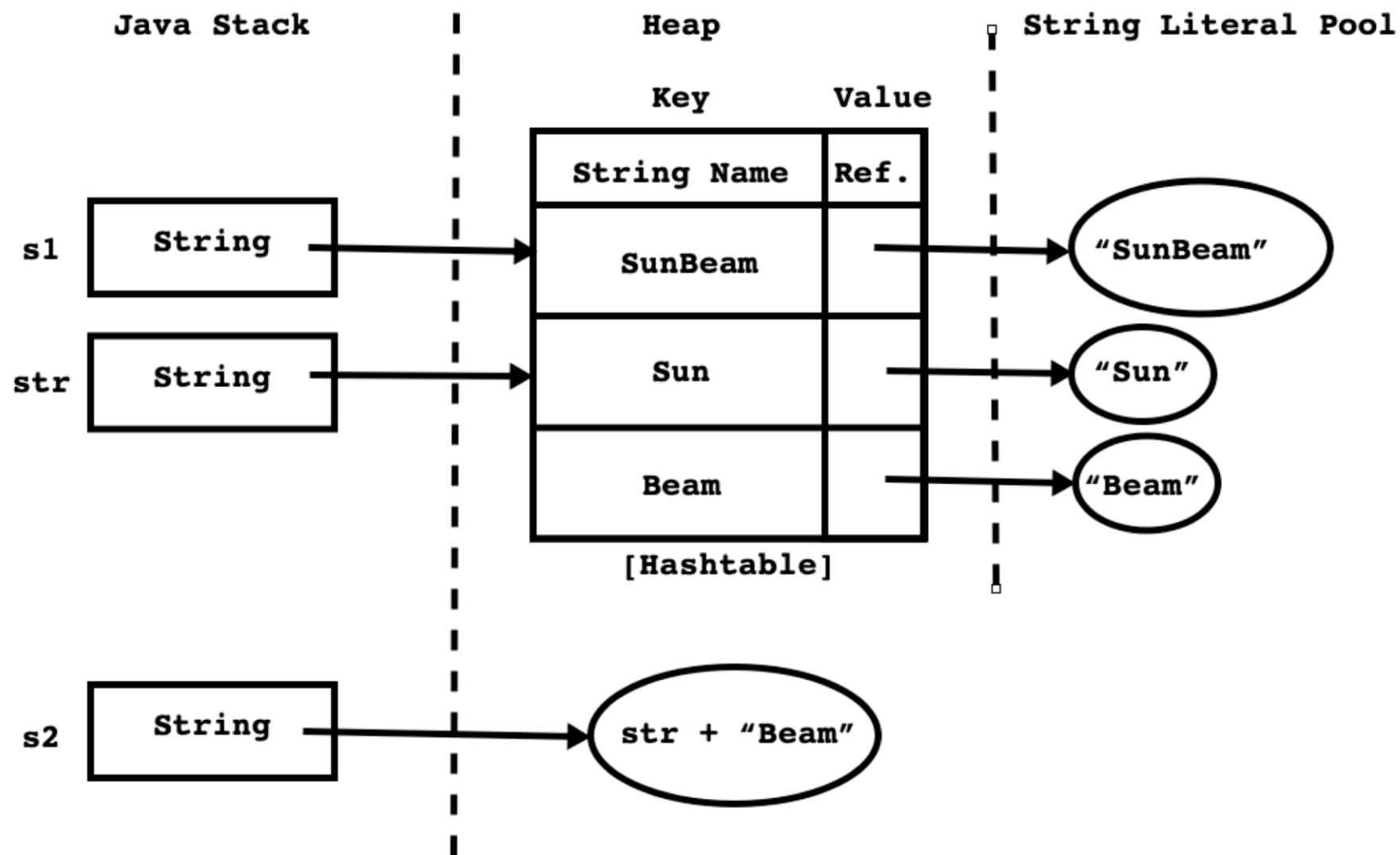


String Twister

```
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  
    }  
}
```

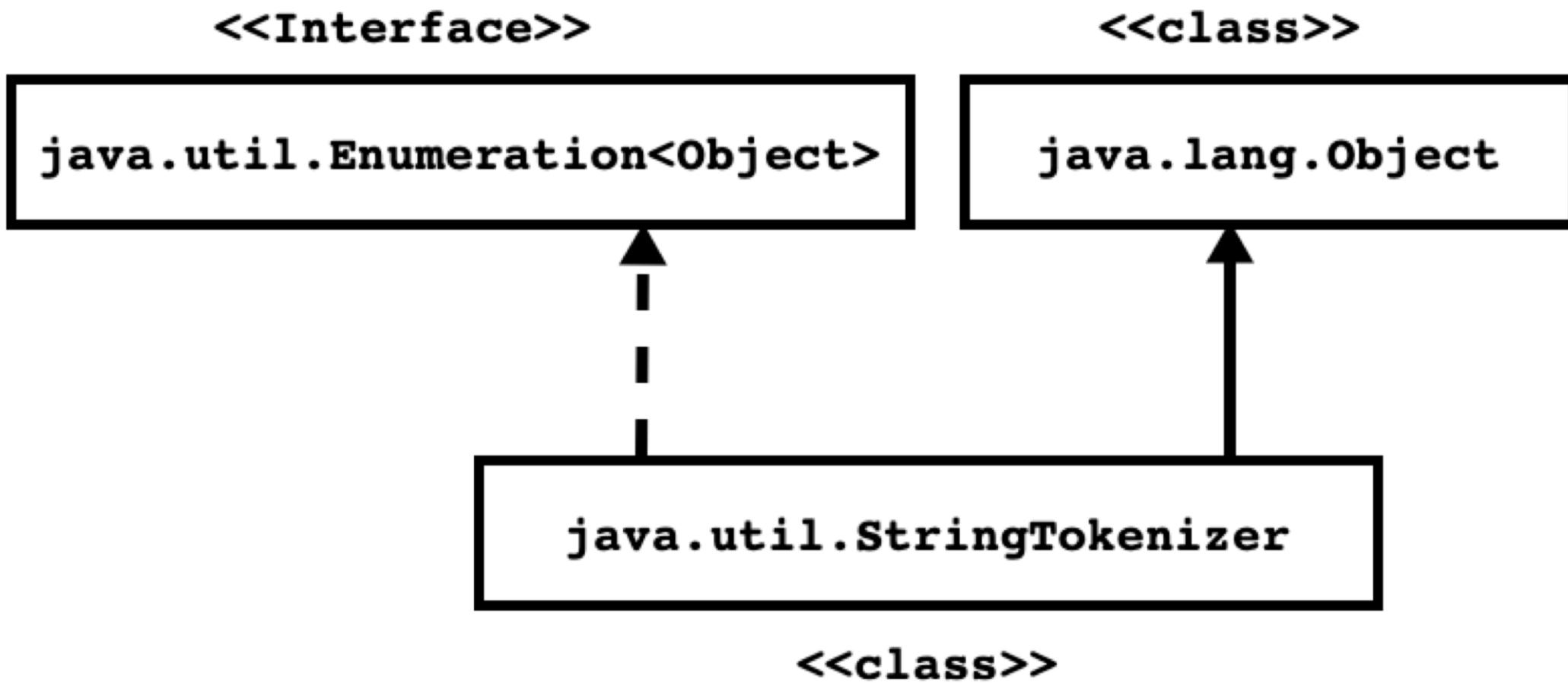


StringBuilder Twister

```
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



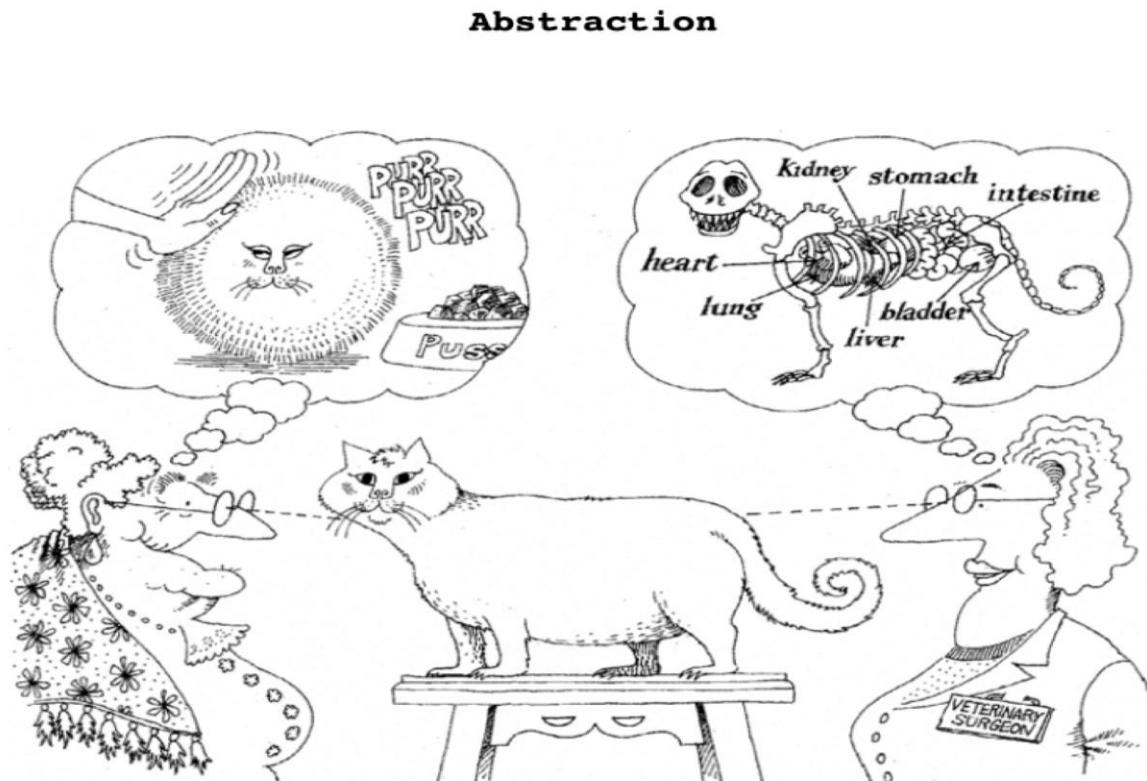
Abstraction

- 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.

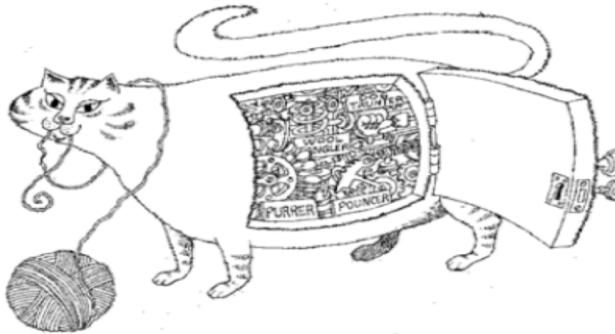
Encapsulation

- 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

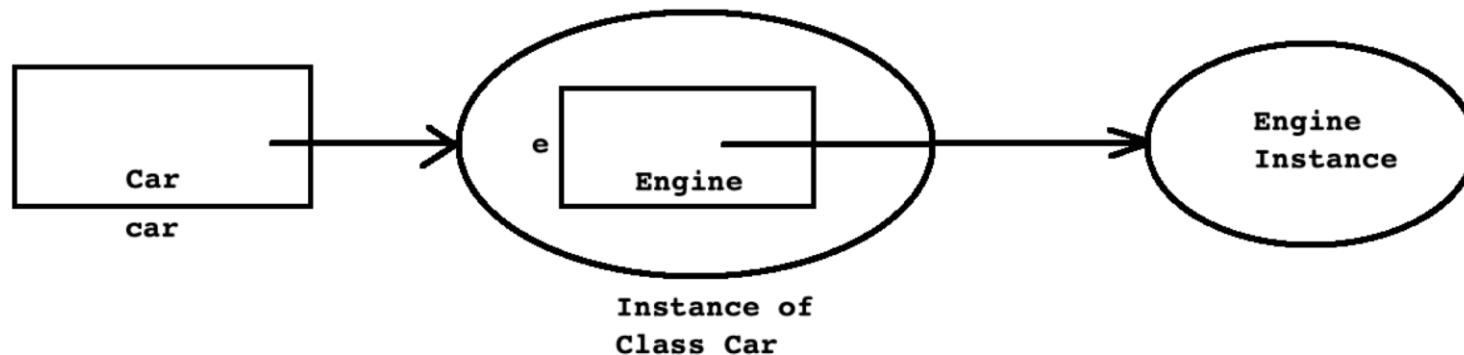
- If has-a relationship is exist between the types then we should use association.
- Example
 - 1. Car has a engine
 - 2. Room has a chair
- Let us consider example of car and engine:
 - 1. Car has a engine
 - 2. Engine is part of Car.
- If object-instance is a part/component of another instance then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



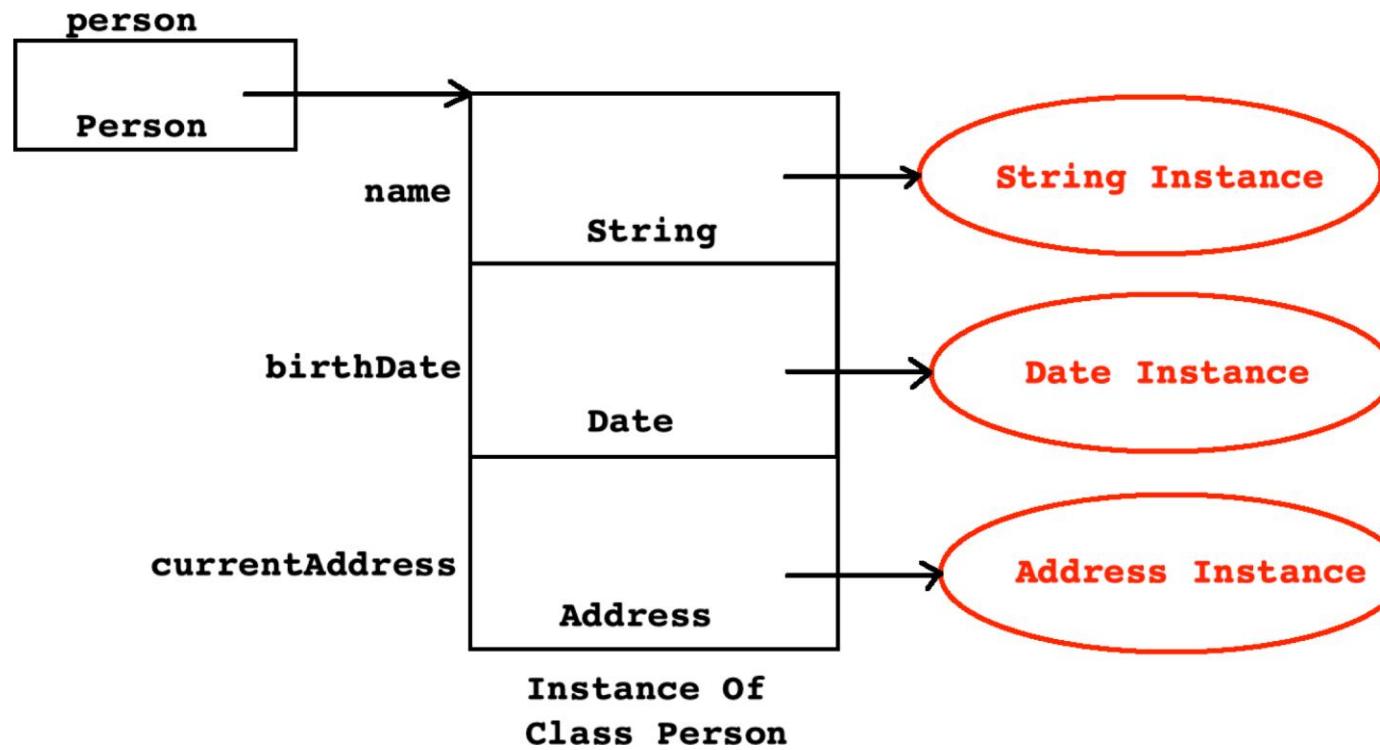
Association In Java

- 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





Object Oriented Programming with Java 8

PG-KDAC

Rohan Paramane

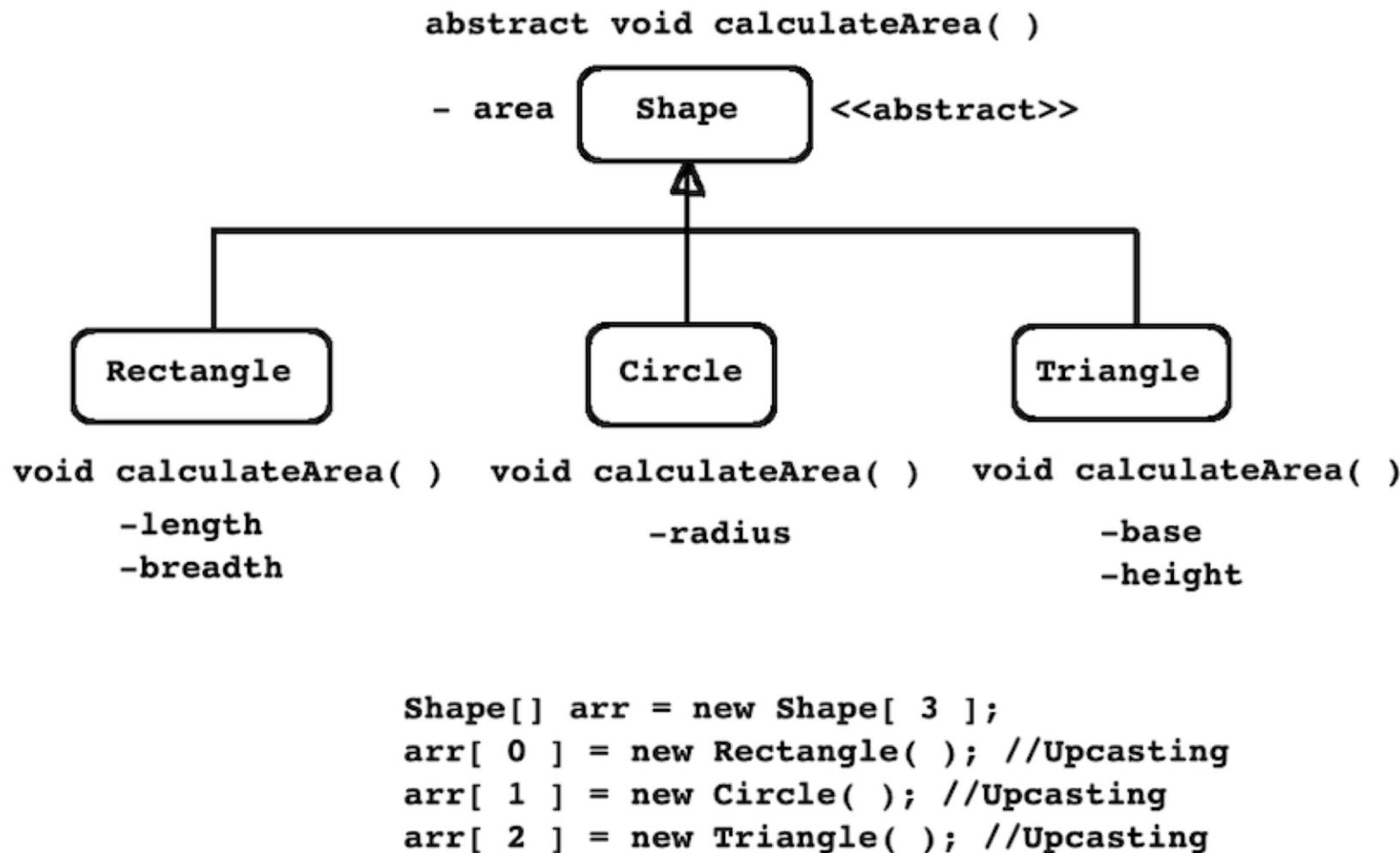


Agenda

- 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.

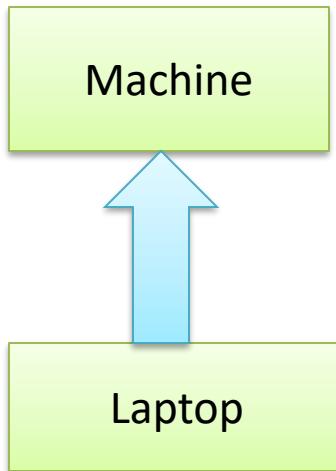


Downcasting

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.

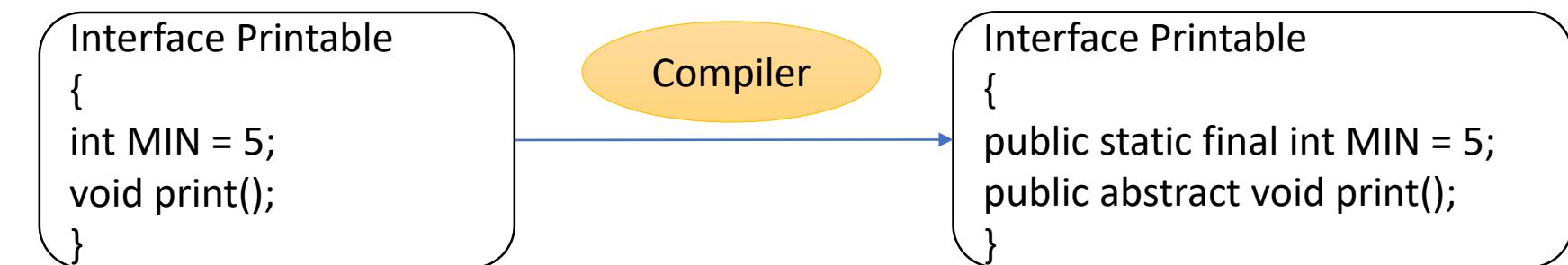


Interface Vs Class

- 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  
}
```



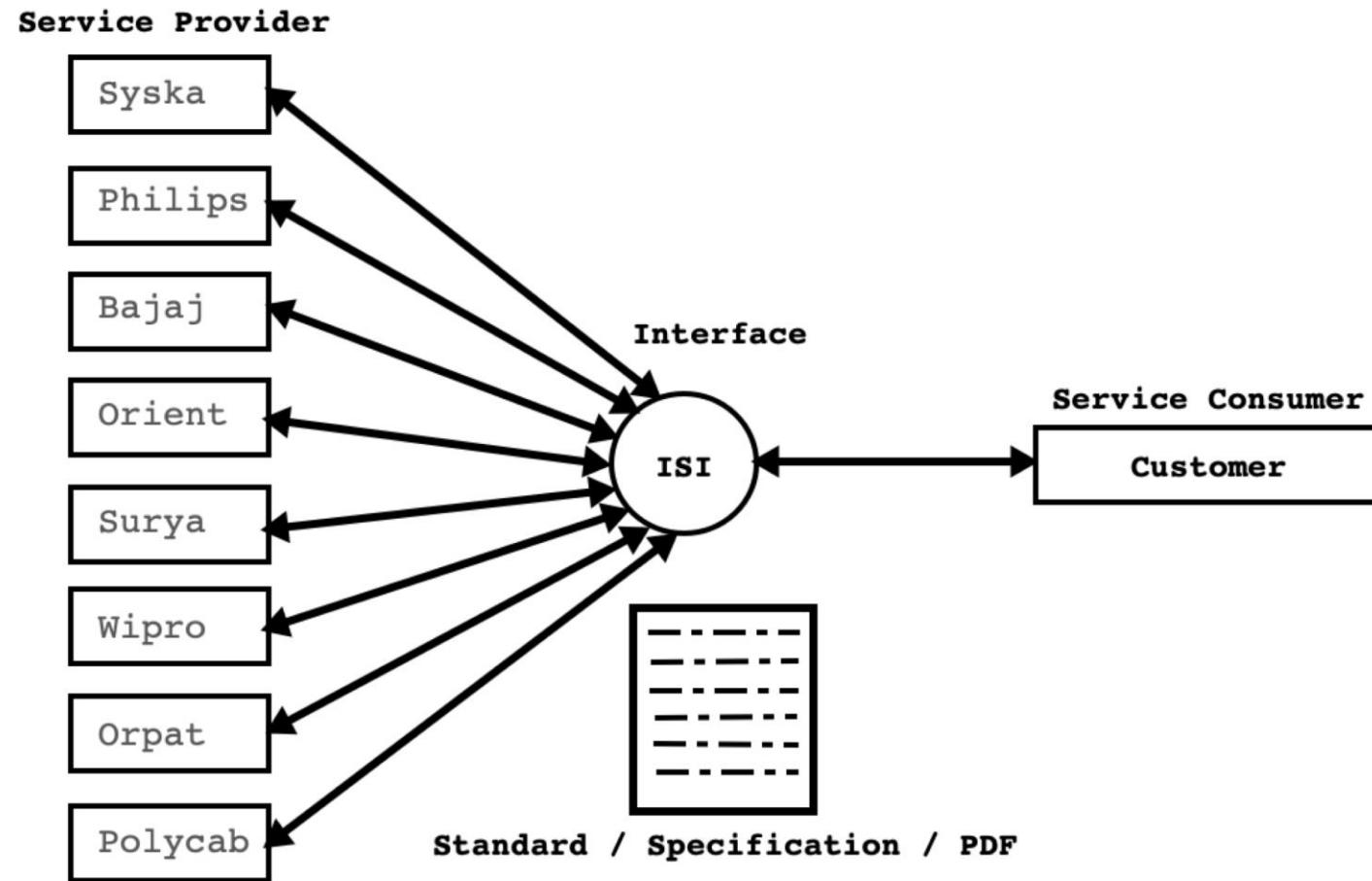
Interface (Features Considering Java8 Interface)

- 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**

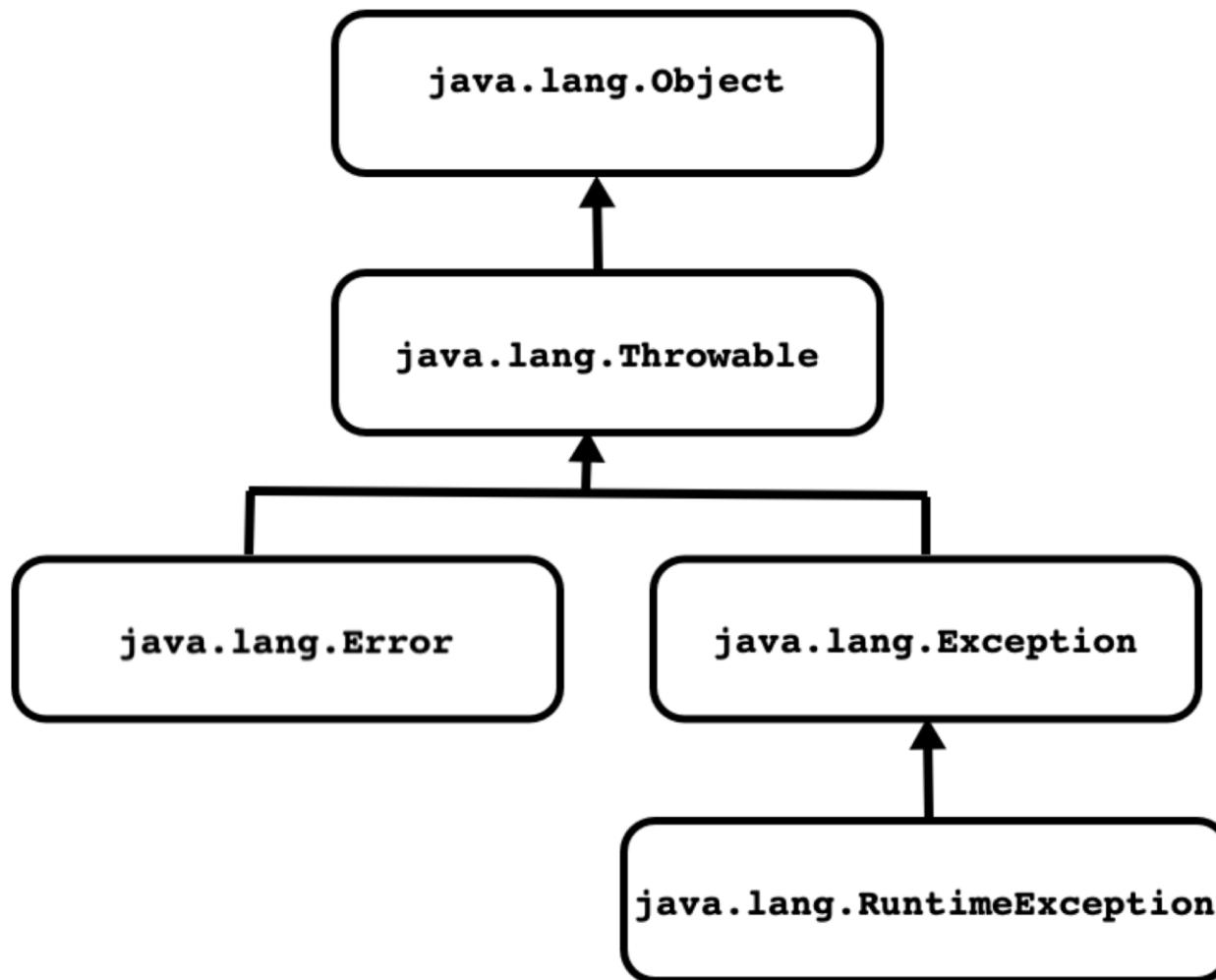
1. To handle all runtime errors at single place. It helps developer to reduces maintenance.
2. To avoid resource leakage/ to manage OS resources carefully.

- **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





Object Oriented Programming with Java 8

Akshita Chanchlani



Agenda

- Boxing
- UnBoxing
- Generic programming using Object class
- Wrapper class hierarchy(Revision)
- Generic Programming Using Generics
- Why Generics
- Type Parameters
- Bounded Type Parameter
- Wild Card
- Generic Method
- Restrictions on generics.



Generic Programming

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

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



Generic Programming(Using Object class)

```
public static void main(String[] args) {  
    Box b1 = new Box( );  
    b1.setObject(123); //b1.setObject(Integer.valueOf(123));  
    Integer n1 = (Integer) b1.getObject(); //Downcasting  
    int n2 = n1.intValue();  
}  
  
public static void main2(String[] args) {  
    Box b1 = new Box( );  
    b1.setObject( new Date());  
    Date date = (Date) b1.getObject(); //Downcasting  
    System.out.println(date.toString());  
}  
  
public static void main(String[] args) {  
    Box b1 = new Box( );  
    b1.setObject( new Date() );  
    String str = (String) b1.getObject(); //Downcasting : ClassCastException  
}
```

- Using `java.lang.Object` class, we can write generic code but we can not write type-safe generic code.
- If we want to write type safe generic code then we should use generics.



Generic Programming(Using Generics)

```
class Box<T>{ //T : Type Parameter
    private T object;
    public T getObject() {
        return object;
    }
    public void setObject(T object) {
        this.object = object;
    }
}
```

```
Box<Date> b1 = new Box<Date>( ); //Date : Type Argument

Box<Date> b1 = new Box<>( ); //Type Inference
```

- By passing, data type as a argument, we can write generic code in Java. Hence parameterized class/type is called as generics.
- If we specify type argument during declaration of reference variable then specifying type argument during instance creation is optional. It is also called as type inference.



Generic Programming(Using Generics)

1. Box<String> b1 = new Box<String>(); //OK
2. Box<String> b2 = new Box<>(); //OK
3. Box<Object> b3 = new Box<String>(); //NOT OK
4. Box<Object> b4 = new Box<Object>(); //OK



Generic Programming(Using Generics)

- If we instantiate parameterized type without type argument then `java.lang.Object` is considered as default type argument.

```
Box b1 = new Box();      //Class Box : Raw Type  
//Box<Object> b1 = new Box<>();
```

- If we instantiate parameterized type without type argument then parameterized type is called raw type.
- During the instantiation of parameterized class, type argument must be reference type.

```
//Box<int> b1 = new Box<>(); //Not OK  
Box<Integer> b1 = new Box<>(); //OK
```



Why Generics?

1. It gives us stronger type checking at compile time. In other words, we can write type safe code.
2. No need of explicit type casting.
3. We can implement generic data structure and algorithm.

Syntax -- similar to c++ templates (angle brackets)

eg : ArrayList<Emp> , HashMap<Integer,Account>

1. Syntax is different than C++ for nested collections only.

A generic class means that the class declaration includes a type parameter.

Eg : class MyGeneric<T>

```
{  
    private T ref;  
}
```

Eg. class MyGeneric<T,U> {...}

T ,U ---type --- ref type

ArrayList<Emp>



Type Parameters Used In Java API

- 1. T : Type
- 2. N : Number
- 3. E : Element
- 4. K : Key
- 5. V : Value
- 6. S, U : Second Type Parameter Names



Bounded Type Parameter

- If we want to put restriction on data type that can be used as type argument then we should specify bounded type parameter.

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

- Specifying bounded type parameter is a job of class implementor.



Bounded Type Parameter

1. Box<Number> b1 = new Box<>(); //OK
2. Box<Boolean> b2 = new Box<>(); //Not OK
3. Box<Character> b3 = new Box<>(); //Not OK
4. Box<String> b4 = new Box<>(); //Not OK
5. Box<Integer> b5 = new Box<>(); //OK
6. Box<Double> b6 = new Box<>(); //OK
7. Box<Date> b7 = new Box<>(); //Not OK



Wild Card

- In generics "?" is called as wild card which represents unknown type.
- **Types of wild card:**
 1. Unbounded wild card
 2. Upper bounded wild card
 3. Lower bounded wilds card



Unbounded Wild Card

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

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

```
public static void main(String[] args) {  
    ArrayList<Integer> integerList = Program.getIntegerList();  
    Program.printRecord( integerList ); //OK  
  
    ArrayList<Double> doubleList = Program.getDoubleList();  
    Program.printRecord( doubleList ); //OK  
  
    ArrayList<String> stringList = Program.getStringList();  
    Program.printRecord( stringList ); //OK  
}
```



Upper Bounded Wild Card

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

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

```
public static void main(String[] args) {  
    ArrayList<Integer> integerList = Program.getIntegerList();  
    Program.printRecord( integerList ); //OK  
  
    ArrayList<Double> doubleList = Program.getDoubleList();  
    Program.printRecord( doubleList ); //OK  
  
    ArrayList<String> stringList = Program.getStringList();  
    Program.printRecord( stringList ); //Not OK  
}
```



Lower Bounded Wild Card

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

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

```
public static void main(String[] args) {  
    ArrayList<Integer> integerList = Program.getIntegerList();  
    Program.printRecord( integerList ); //OK  
  
    ArrayList<Double> doubleList = Program.getDoubleList();  
    Program.printRecord( doubleList ); //NOT OK  
  
    ArrayList<String> stringList = Program.getStringList();  
    Program.printRecord( stringList ); //NOT OK  
}
```



Generic Method

```
private static <T> void print( T obj ) {  
    System.out.println(obj);  
}
```

```
private static <T extends Number> void print( T obj ) {  
    System.out.println(obj);  
}
```



Restrictions on Generics

1. Cannot Instantiate Generic Types with Primitive Types
2. Cannot Create Instances of Type Parameters
3. Cannot Declare Static Fields Whose Types are Type Parameters
4. Cannot Use Casts or instanceof with Parameterized Types
5. Cannot Create Arrays of Parameterized Types
6. Cannot Create, Catch, or Throw Objects of Parameterized Types
7. Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type



Collection Framework



Collection Framework

- Every value/data stored in data structure is called element.
- A group of data (Object reference) handled as a single unit.
- Framework is library of reusable classes/interfaces that is used to develop application.
- **Library of reusable data structure classes that is used to develop java application is called collection framework.**
- Main purpose of collection framework is to manage data in RAM efficiently.
- Consider following Example:
 1. Person has-a birthdate
 2. Employee is a person
- In java, collection instance do not contain instances rather it contains reference of instances.
- If we want to use collection framework them we should **import java.util** package.

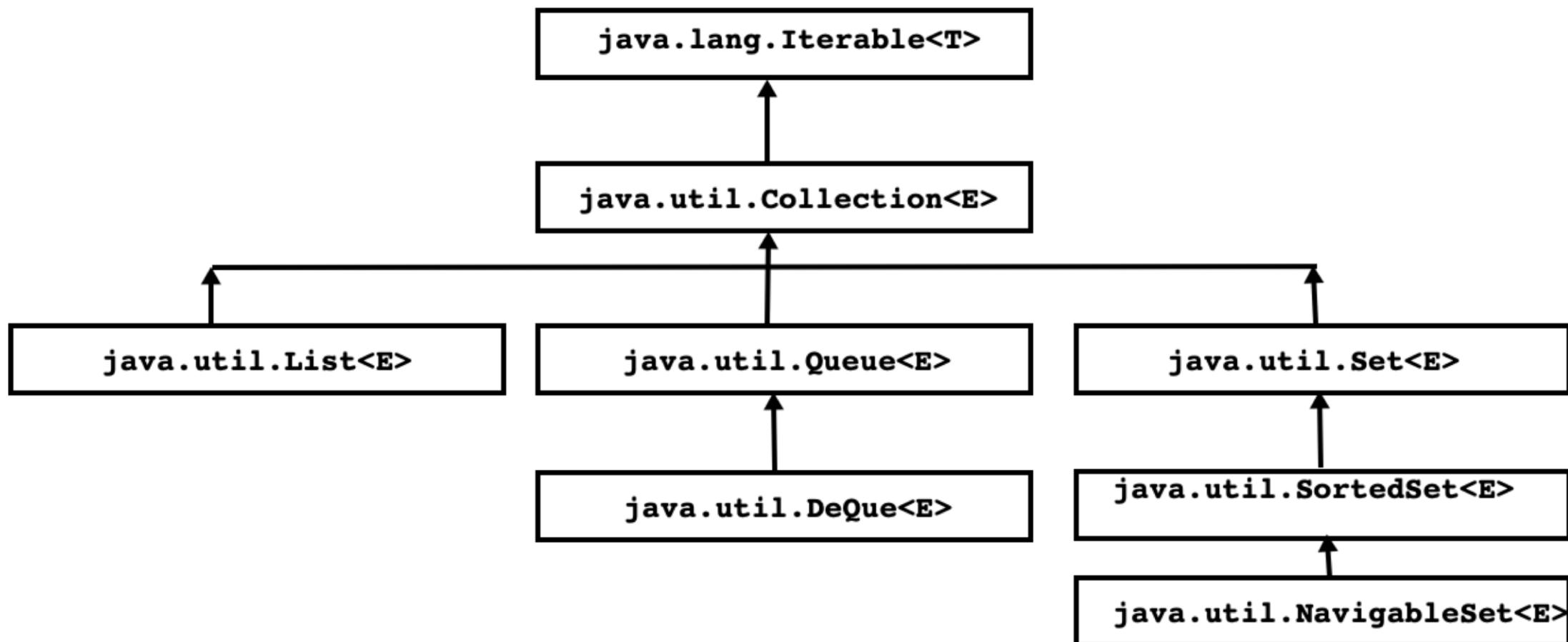


Collection Framework

- Collection frameworks encapsulates the algorithms
- Programmer don't have to worry about the implementation of the various collections types.
- Nevertheless, still collection framework is encourages programmer to extend members of collection to create a specific implementation where one is needed

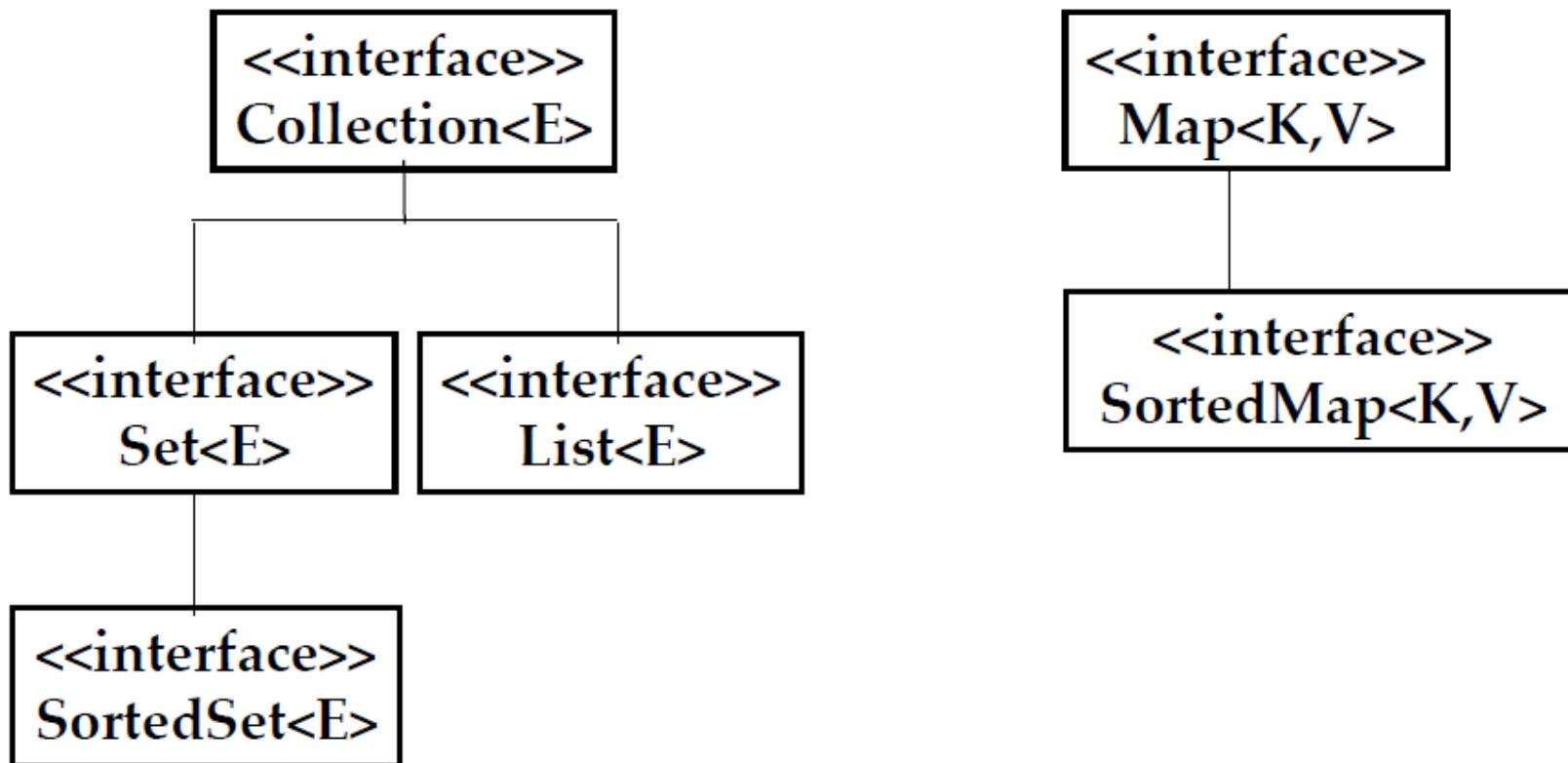


Collection Interface Hierarchy



Collection Framework Interface Hierarchy





Collection framework is in `java.util` package



Iterable<T>

- It is a interface declared in java.lang package.
- All the collection classes implements Iterable interface hence we can traverse it using for each loop
- Methods of Iterable interface:
 1. Iterator<T> **iterator()**
 2. default Spliterator<T> **spliterator()**
 3. default void **forEach**(Consumer<? super T> action)



Collection<E>

- Collection<E> is interface declared in java.util package.
- It is sub interface of Iterable interface.
- It is **root interface** in collection framework interface hierarchy.
- Default methods of Collection interface
 - 1. default Stream<E> **stream()**
 - 2. default Stream<E> **parallelStream()**
 - 3. default boolean **removeIf**(Predicate<? super E> filter)



Collection<E>

- Abstract Methods of Collection Interface

1. boolean **add**(E e)
2. boolean **addAll**(Collection<? extends E> c)
3. void **clear**()
4. boolean **contains**(Object o)
5. boolean **containsAll**(Collection<?> c)
6. boolean **isEmpty**()
7. boolean **remove**(Object o)
8. boolean **removeAll**(Collection<?> c)
9. boolean **retainAll**(Collection<?> c)
10. int **size**()
11. Object[] **toArray**()
12. <T> T[] **toArray**(T[] a)

There is no direct concrete implementation class for Collection interface but exists for sub interface

This interface supports the most basic and general operations and queries that can be performed on a group of objects.



List<E>

- It is sub interface of java.util.Collection interface.
- It is ordered/sequential collection.
- **ArrayList, Vector, Stack, LinkedList etc. implements List interface. It generally referred as "List collections".**
- List collection can contain duplicate element as well multiple elements.
- Using integer index, we can access elements from List collection.
- We can traverse elements of List collection using Iterator as well as ListIterator.
- A list is a collection in which duplicate elements are allowed, and where the order is significant null
- List interface inherits from Collection, but changes the semantics of some methods, and adds new methods. The list starts its index at 0.

<a,b,c>, <c,a,b>and <a,b,b,c>are different lists

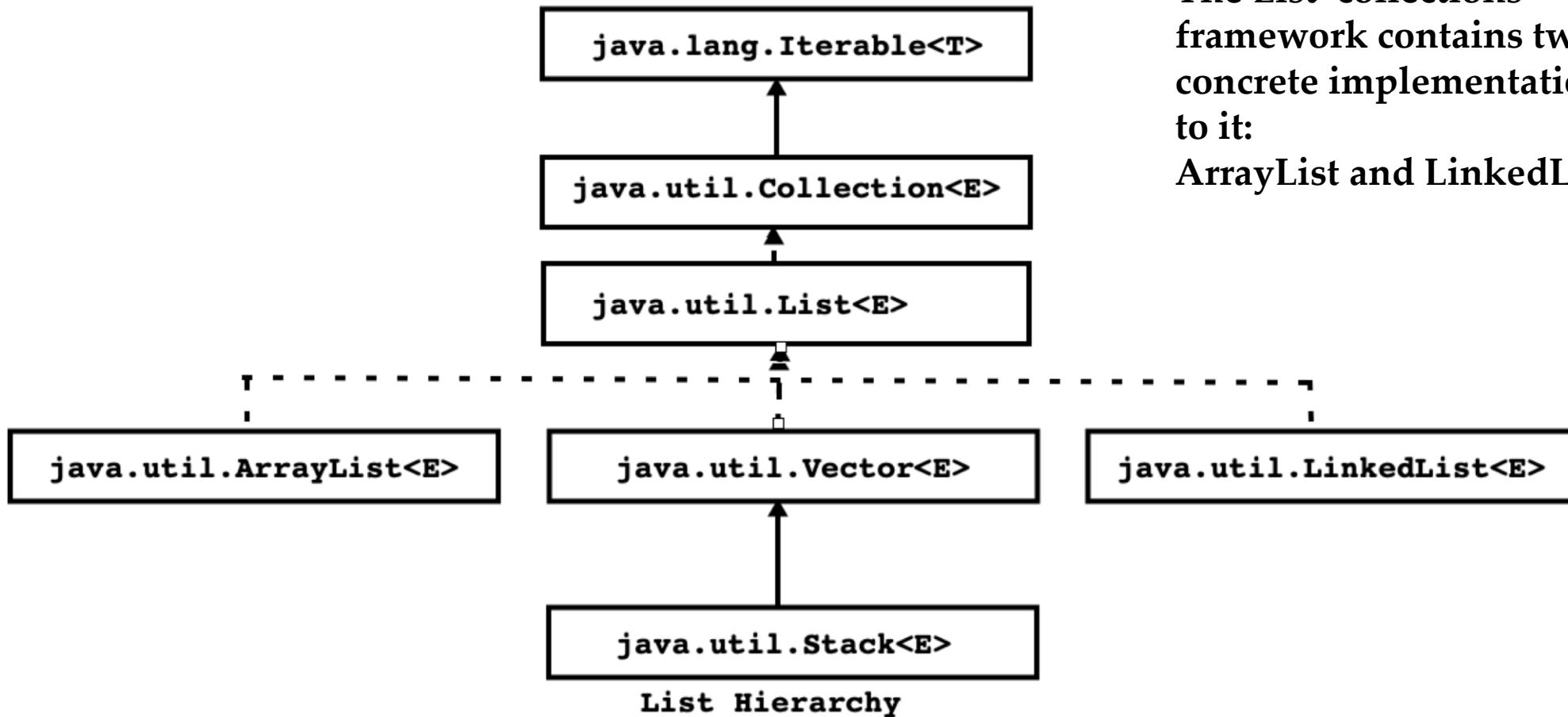


List<E>

- Abstract methods of List Interface
 1. void **add**(int index, E element)
 2. boolean **addAll**(int index, Collection<? extends E> c)
 3. E **get**(int index)
 4. int **indexOf**(Object o)
 5. int **lastIndexOf**(Object o)
 6. ListIterator<E> **listIterator**()
 7. ListIterator<E> **listIterator**(int index)
 8. E **remove**(int index)
 9. E **set**(int index, E element)
 10. List<E> **subList**(int fromIndex, int toIndex)



List Interface Hierarchy



The List collections framework contains two concrete implementations to it:
`ArrayList` and `LinkedList`.

ArrayList<E>

- It is resizable array.
- It implements List<E>, RandomAccess, Cloneable, Serializable interfaces.
- It is List collection.
- It is unsynchronized collection. Using "Collections.synchronizedList" method, we can make it synchronized.

➤ List list = Collections.synchronizedList(new ArrayList(...));

- Initial capacity of ArrayList is 10. If ArrayList is full then its capacity gets increased by half of its existing capacity.
- ArrayList is recommended when you're adding and removing elements only to the end of the collection satisfies you and when you wish to access elements using their indices.
- It is less time-consuming than LinkedList is.



Vector<E>

- It is resizable array.
- It implements List<E>, RandomAccess, Cloneable, Serializable.
- It is List collection.
- It is synchronized collection.
- Default capacity of vector is 10. If vector is full then its capacity gets increased by its existing capacity.
- We can traverse elements of vector using Iterator, ListIterator as well as Enumeration.
- **Note:** If we want to manage elements of non final type inside Vector then non final type should override "equals" method.
- It is also called as growable array.



Vector<E>

- Duplicate elements are allowed in the vector class.
- It preserves the insertion order in Java.
- Null elements are allowed in the Java vector class.
- Heterogeneous elements are allowed in the vector class. Therefore, it can hold elements of any type and any number.
- Most of the methods present in the vector class are synchronized. That means it is a thread-safe. Two threads cannot access the same vector object at the same time. Only one thread can access can enter to access vector object at a time.
- Vector class is preferred where we are developing a multi-threaded application but it gives poor performance because it is thread-safety.
- Vector is rarely used in a non-multithreaded environment due to synchronized which gives you poor performance in searching, adding, delete, and update of its element.
- It can be iterated by a simple for loop, Iterator, ListIterator, and Enumeration.
- Vector is the best choice if the frequent operation is retrieval (getting).



Synchronized Collections

- 1. **Vector**
- 2. **Stack** (Sub class of Vector)
- 3. **Hashtable**
- 4. **Properties** (Sub class of Hashtable)



Enumeration<E>

- It is interface declared in `java.util` package.
- Methods of Enumeration I/F
 - 1. `boolean hasMoreElements()`
 - 2. `E nextElement()`
- It is used to traverse collection only in forward direction. During traversing, we can add, set or remove element from collection.
- It is introduced in jdk 1.0.
- "`public Enumeration<E> elements()`" is a method of `Vector` class.

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



Iterator<E>

- It is a interface declared in `java.util` package.
- It is used to traverse collection only in forward direction. During traversing, we can not add or set element but we can remove element from collection.
- Methods of Iterator
 - 1. `boolean hasNext()`
 - 2. `E next()`
 - 3. `default void remove()`
 - 4. `default void forEachRemaining(Consumer<? super E> action)`
- It is introduced in jdk 1.2

```
Integer element = null;
Iterator<Integer> itr = v.iterator();
while( itr.hasNext()){
    element = itr.next();
    System.out. println(element);
}
```

ListIterator<E>

- It is sub interface of Iterator interface.
- It is used to traverse only List Collection in bidirectional.
- During traversing, we can add, set as well as remove element from collection.
- It is introduced in jdk 1.2
- Methods of ListIterator
 - 1. boolean hasNext()
 - 2. E next()
 - 3. boolean hasPrevious()
 - 4. E previous()
 - 5. void add(E e)
 - 6. void set(E e)
 - 7. void remove()



ListIterator<E>

```
Integer element = null;
ListIterator<Integer> itr = v.listIterator();
while( itr.hasNext()){
    element = itr.next();
    System.out.print(element+ " ");
}

while( itr.hasPrevious()){
    element = itr.previous();
    System.out.print(element+ " ");
}
```



Types of Iterator

- **Fail Fast Iterator**
- During traversing, using collection reference, if we try to modify state of collection and if iterator do not allows us to do the same then such iterator is called "Fail Fast" Iterator. In this case JVM throws **ConcurrentModificationException**.

```
Integer element = null;
Iterator<Integer> itr = v.iterator();
while( itr.hasNext()){
    element = itr.next();
    System.out.print(element+ " ");
    if( element == 50 )
        v.add(60); //ConcurrentModificationException
}
```



Types of Iterator

- **Fail Safe Iterator**
- During traversing, if iterator allows us to do changes in underlying collection then such iterator is called fail safe iterator.

```
Integer element = null;
Enumeration<Integer> e = v.elements();
while( e.hasMoreElements()){
    element = e.nextElement();
    System.out.print(element+" ");
    if( element == 50 )
        v.add(60); //OK
}
```



Stack<E>

- It is linear data structure which is used to manage elements in Last In First Out order.
- It is sub class of Vector class.
- It is synchronized collection.
- It is List Collection.
- Methods of Stack class
 - 1. public boolean empty()
 - 2. public E push(E item)
 - 3. public E peek()
 - 4. public E pop()
 - 5. public int search(Object o)
- * Since it is synchronized collection, it slower in performance.
- * For high performance we should use ArrayDeque class.



LinkedList<E>

- It is a List collection.
- It implements List<E>, Deque<E>, Cloneable and Serializable interface.
- Its implementation is depends on Doubly linked list.
- It is unsynchronized collection. Using Collections.synchronizedList() method, we can make it synchronized.

➤ **List list = Collections.synchronizedList(new LinkedList(...));**

- Note : If we want to manage elements of non-final type inside LinkedList then non final type should override "equals" method.
- Instantiation

➤ **List<Integer> list = new LinkedList<>();**

LinkedList is best when add and remove operations happen anywhere, not only at the end.



LinkedList<E>

Arrays have certain limitations such as:

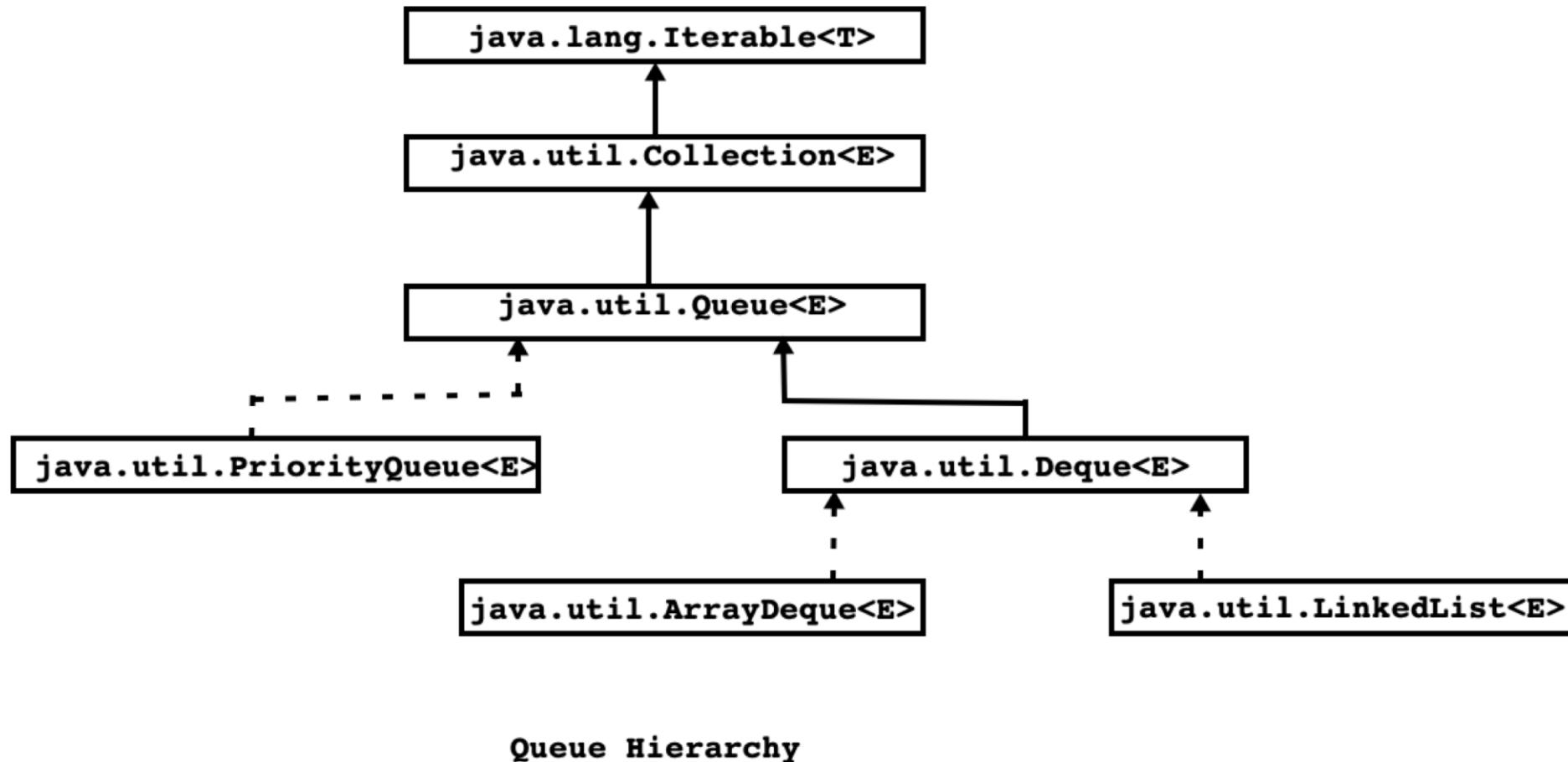
- Size of the array is fixed
- Array elements need contiguous memory locations
- Inserting an element in an array is performance wise expensive
- deleting an element from the array is also a performance wise expensive

Linked List by providing following features:

- Allows **dynamic memory allocation**
- elements **don't need contiguous memory locations**
- Insert and delete operations in the Linked list are not performance wise expensive because adding and deleting an element from the linked list doesn't require element shifting, only the pointer of the previous and the next node requires change.



Queue Interface Hierarchy



Queue<E>

- It is interface declared in `java.util` package.
- It is sub interface of `Collection` interface.
- It is introduced in jdk 1.5

Summary of Queue methods

	<i>Throws exception</i>	<i>Returns special value</i>
Insert	<code>add(e)</code>	<code>offer(e)</code>
Remove	<code>remove()</code>	<code>poll()</code>
Examine	<code>element()</code>	<code>peek()</code>

Since the Queue is an interface, we cannot provide the direct implementation of it.

Classes that implement Queue Interface in Java:

1.Priority Queue

2.Dequeue



Queue Interface Methods

add()

- Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.

offer()

- Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.

element()

- Returns the head of the queue. Throws an exception if the queue is empty.

peek()

- Returns the head of the queue. Returns null if the queue is empty.

remove()

- Returns and removes the head of the queue. Throws an exception if the queue is empty.

poll()

- Returns and removes the head of the queue. Returns null if the queue is empty.



Queue<E>

```
Queue<Integer> que = new ArrayDeque<>();
que.offer(10);
que.offer(20);
que.offer(30);

Integer ele = null;
while( !que.isEmpty()){
    ele = que.peek();
    //TODO : processing
    que.poll();
}
```

```
Queue<Integer> que = new ArrayDeque<>();
que.add(10);
que.add(20);
que.add(30);

Integer ele = null;
while( !que.isEmpty()){
    ele = que.element();
    //TODO : Processing
    que.remove();
}
```



Deque<E>

- It is usually pronounced "deck".
- It is sub interface of Queue.
- It is introduced in jdk 1.6
- If we want to perform operations from bidirection then we should use Deque interface.

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	<i>Throws exception</i>	<i>Special value</i>	<i>Throws exception</i>	<i>Special value</i>
Insert	<code>addFirst(e)</code>	<code>offerFirst(e)</code>	<code>addLast(e)</code>	<code>offerLast(e)</code>
Remove	<code>removeFirst()</code>	<code>pollFirst()</code>	<code>removeLast()</code>	<code>pollLast()</code>
Examine	<code>getFirst()</code>	<code>peekFirst()</code>	<code>getLast()</code>	<code>peekLast()</code>



Deque<E>

- Deque Interface is a linear collection that supports element insertion and removal at both ends.
- The class which implements this interface is ArrayDeque.
- It extends the Queue interface.
- Deque is an interface and has two implementations: LinkedList and ArrayDeque.
- **Creating a Deque**
Syntax : `Deque dq = new LinkedList();`
`Deque dq = new ArrayDeque();`



ArrayDeque

ArrayDeque class provides the facility of using deque and resizable-array.
It inherits the AbstractCollection class and implements the Deque interface.

Syntax : `Deque<String> arr = new ArrayDeque<String>();`



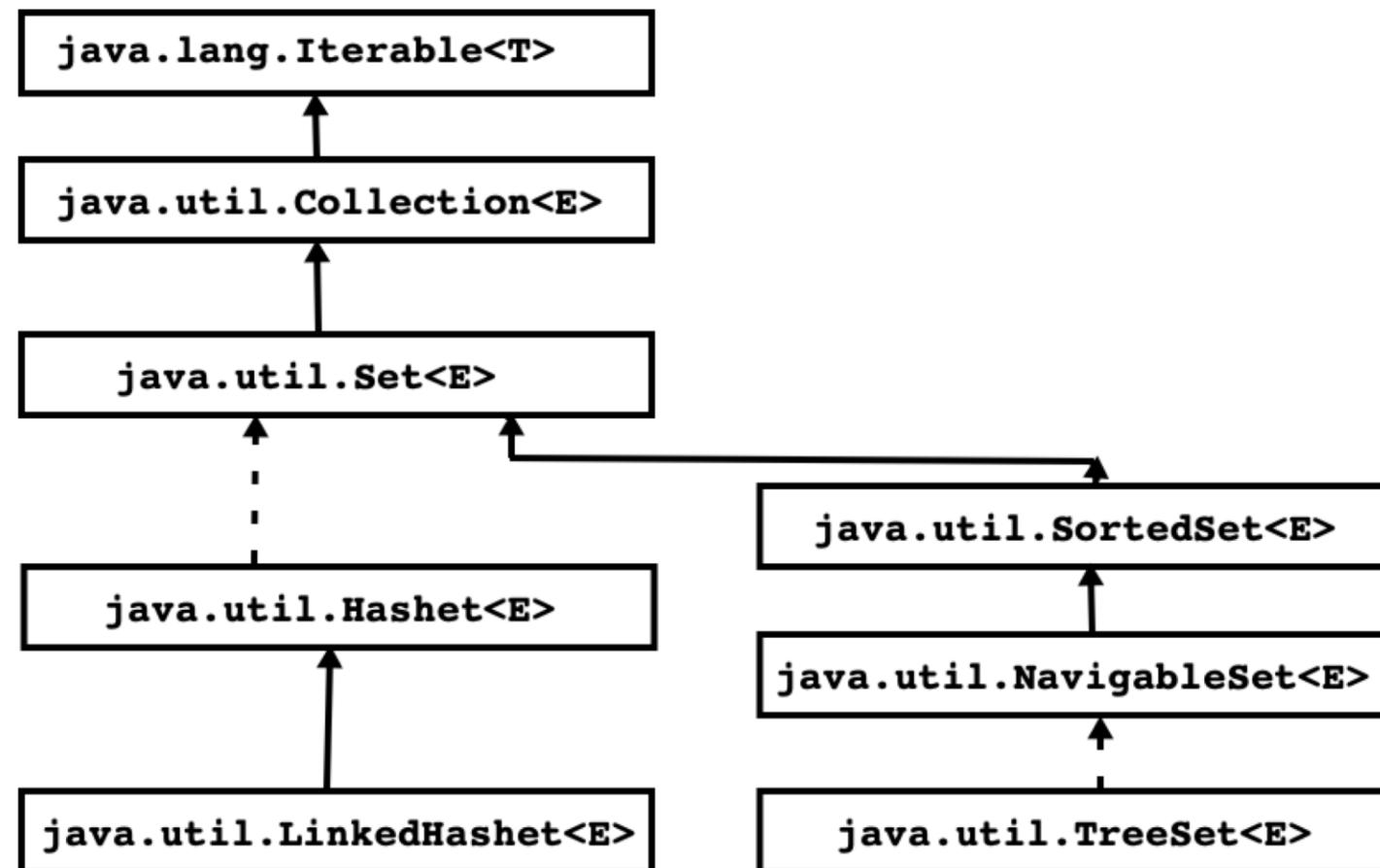
Priority Queue

- PriorityQueue class provides the functionality of the heap data structure.
- The PriorityQueue class provides the facility of using a queue.
- It does not order the elements in a FIFO manner.
- It is based on Priority Heap.
- The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

Syntax : PriorityQueue<Integer> numbers = new PriorityQueue<Integer>();
// it will create PriorityQueue without any arguments
//head of the queue will be smallest element of the queue
//elements are removed in ascending order from the queue.



Set Interface Hierarchy



Set<E>

- It is sub interface of `java.util.Collection` interface.
- `HashSet`, `LinkedHashSet`, `TreeSet` etc. implements Set interface. It is also called as Set collection.
- Set collections do not contain duplicate elements.
- Set will not maintain any order for elements
- While adding new element it is using Object's `equals()` and `hashCode()` method to check , if such element is there or not in set

**There are three concrete Set implementations that are part of the Collection Framework:
HashSet, TreeSet, and LinkedHashSet.**



HashSet and TreeSet, LinkedHashSet

- You use HashSet, which maintains its collection in an unordered manner.
- If this doesn't suit your needs, you can use TreeSet.
- A TreeSet keeps the elements in the collection in sorted order
- While HashSet has an undefined order for its elements, LinkedHashSet supports iterating through its elements in the order they were inserted.
- Understand that the additional features provided by TreeSet and LinkedHashSet add to the runtime costs.



TreeSet<E>

- It is Set collection.
- It can not contain duplicate element as well as null element.
- It is sorted collection.
- Its implementation is based on TreeMap
- It is unsynchronized collection.
- Using "Collections.synchronizedSortedSet()" method we can make it synchronized.

➤ **SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));**

- It is introduced in jdk 1.2
- Note : If we want to manage elements of non final type inside TreeSet then non final type should implement Comparable interface.
- Instantiation

➤ **Set<Integer> set = new TreeSet<>();**



Hashing

- Hashing is a searching algorithm which is used to search element in constant time(faster searching) .
- In case array, if we know index of element then we can locate it very fast.
- Hashing technique is based on "hashcode".
- Hashcode is not a reference or address of the object rather it is a logical integer number that can be generated by processing state of the object.
- Generating hashcode is a job of hash function/method.
- Generally hashcode is generated using prime number.

```
//Hash Method
private static int getHashCode(int data)
{
    int result = 1;
    final int PRIME = 31;
    result = result * data + PRIME * data;
    return result;
}
```



Hashing

- If state of object-instance is same then we will get same hashcode.
- Hashcode is required to generate slot.
- If state of objects are same then their hashcode and slot will be same.
- By processing state of two different object's , if we get same slot then it is called collision.
- Collision resolution techniques:
 - Seperate Chaining / Open Hashing
 - Open Addressing / Close Hashing
 - 1. Linear Probing
 - 2. Quadratic Probing
 - 3. Double Hashing / Rehashing



Hashing

- Collection(LinkedList/Tree) maintained per slot is called bucket.
- Load Factor = (Count of bucket / Total elements);
- **In hashCode based collection, if we want manage elements of non final type then reference type should override equals() and hashCode() method.**
- hashCode() is non final method of java.lang.Object class.
- Syntax:
 - public native int hashCode();
- On the basis of state of the object, we want to generate hashCode then we should override hashCode() method in sub class.
- The hashCode method defined by class Object does return distinct integers for distinct objects. This is typically implemented by converting the internal address of the object into an integer.



HashSet<E>

- It Set Collection.
- It can not contain duplicate elements but it can contain null element.
- It's implementation is based on HashTable.
- It is unordered collection.
- It is unsynchronized collection. Using Collections.synchronizedSet() method, we can make it synchronized.
- It is introduced in jdk 1.2
- Note : If we want to manage elements of non final type inside HashSet then non final type should override equals and hashCode() method.
- Instantiation:
 - **Set<Integer> set = new HashSet<>();**



LinkedHashSet<E>

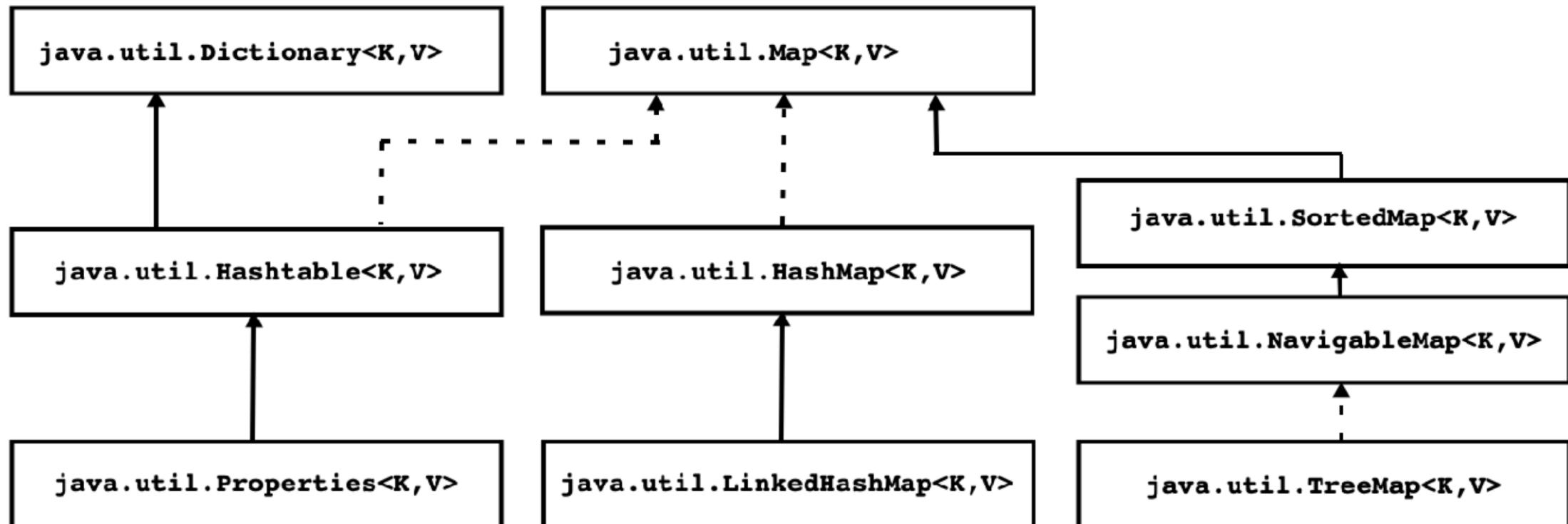
- It is sub class of HashSet class.
- Its implementation is based on linked list and Hashtable.
- It is ordered collection.
- It is unsynchronized collection. Using Collections.synchronizedSet() method we can make it synchronized.

➤ **Set s = Collections.synchronizedSet(new LinkedHashSet(...));**

- It is introduced in jdk 1.4
- It can not contain duplicate element but it can contain null element.



Map Interface Hierarchy



Dictionary<K,V>

- It is abstract class declared in java.util package.
- It is super class of Hashtable.
- It is used to store data in key/value pair format.
- It is not a part of collection framework
- It is introduced in jdk 1.0
- Methods:
 1. public abstract boolean isEmpty()
 2. public abstract V put(K key, V value)
 3. public abstract int size()
 4. public abstract V get(Object key)
 5. public abstract V remove(Object key)
 6. public abstract Enumeration<K> keys()
 7. public abstract Enumeration<V> elements()
- Implementation of Dictionary is Obsolete.



Map<K,V>

- It is part of collection framework but it doesn't extend Collection interface.
- HashMap, Hashtable, TreeMap etc are Map collection's.
- This interface takes the place of the Dictionary class, which was a totally ~~Map collection stores data in key/value pair format~~ interface.
- In map we can not insert duplicate keys but we can insert ~~duplicate values~~.
- It is introduced in jdk 1.2
- Map.Entry<K,V> is nested interface of Map<K,V>. ~~duplicate values~~.
- Following are abstract methods of Map.Entry interface.
 1. K getKey()
 2. V getValue()
 3. V setValue(V value)



Map<K,V>

- Abstract Methods of Map<K,V>

1. boolean isEmpty()
2. V put(K key, V value)
3. void putAll(Map<? extends K, ? extends V> m)
4. int size()
5. boolean containsKey(Object key)
6. boolean containsValue(Object value)
7. V get(Object key)
8. V remove(Object key)
9. void clear()
10. Set<K> keySet()
11. Collection<V> values()
12. Set<Map.Entry<K,V>> entrySet()

- An instance, whose type implements Map.Entry<K,V> interface is called entry instance.

Map Implementation
Collection framework gives two classes
HashMap and TreeMap



HashMap , TreeMap

- By default, choose HashMap, it serves the most needs.
- TreeMap implementation will maintain the keys of the map in a sorted order.
- it's better to simply keep everything in a HashMap while adding, and create a TreeMap at the end:
- Map <String, String> map = new HashMap<String, String>(); // Add and remove elements from unsorted map
map.put("Foo", "Bar"); map.put("Bar", "Foo"); map.remove("Foo"); map.put("Foo", "Baz"); // Then sort before displaying elements// in sorted order
map = new TreeMap(map);



Hashtable<K,V>

- It is Map<K,V> collection which extends Dictionary class.
- It can not contain duplicate keys but it can contain duplicate values.
- In Hashtable, Key and value can not be null.
- It is synchronized collection.
- It is introduced in jdk 1.0
- In Hashtable, if we want to use instance non final type as key then it should override equals and hashCode method.



HashMap<K,V>

- It is map collection
- It's implementation is based on Hashtable.
- It can not contain duplicate keys but it can contain duplicate values.
- In HashMap, key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method, we can make it synchronized.
 - Map m = Collections.synchronizedMap(new HashMap(...));
- It is introduced in jdk 1.2.
- Instantiation
 - Map<Integer, String> map = new HashMap<>();
- **Note : In HashMap, if we want to use element of non final type as a key then it should override equals() and hashCode() method.**



LinkedHashMap<K,V>

- It is sub class of HashMap<K,V> class
- Its implementation is based on LinkedList and Hashtable.
- It is Map collection hence it can not contain duplicate keys but it can contain duplicate values.
- In LinkedHashMap, key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method we can make it synchronized.

➤ **Map m = Collections.synchronizedMap(new LinkedHashMap(...));**

- LinkedHashMap maintains order of entries according to the key.
- Instantiation:

➤ **Map<Integer, String> map = new LinkedHashMap<>();**
- It is introduced in jdk 1.4

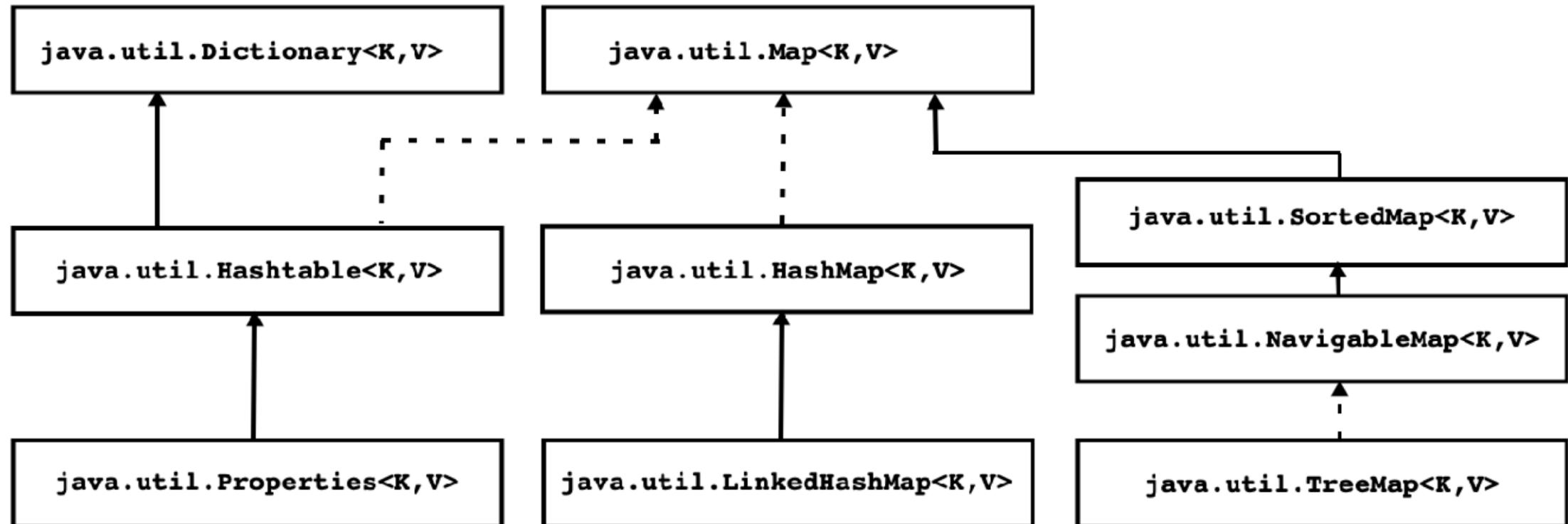


TreeMap<K,V>

- It is map collection.
- It can not contain duplicate keys but it can contain duplicate values.
- In TreeMap, key must not be null but value can be null.
- Implementation of TreeMap is based on Red-Black Tree.
- It maintains entries in sorted form according to the key.
- It is unsynchronized collection. Using Collections.synchronizedSortedMap() method, we can make it synchronized.
 - **SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));**
- Instantiation:
 - **Map<Integer, String> map = new TreeMap<>();**
- It is introduced in jdk 1.2
- Note : In TreeMap, if we want to use element of non final type as a key then it should implement Comparable interface.



Map Interface Hierarchy



- Map is not considered to be a true collection, as the Map interface does not extend the *Collection* interface.
- *Map* is not a true collection, its characteristics and behaviors are different than the other collections like *List* or *Set*.

Dictionary<K,V>

- It is abstract class declared in java.util package.
- It is super class of Hashtable.
- It is used to store data in key/value pair format.
- It is not a part of collection framework
- It cannot contain duplicate keys and each key can map to at most one value
- Methods:
 1. public abstract boolean isEmpty()
 2. public abstract V put(K key, V value)
 3. public abstract int size()
 4. public abstract V get(Object key)
 5. public abstract V remove(Object key)
 6. public abstract Enumeration<K> keys()
 7. public abstract Enumeration<V> elements()
- Implementation of Dictionary is Obsolete.



Map<K,V>

- It is part of collection framework but it doesn't extend Collection interface.
- This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
- HashMap, Hashtable, TreeMap etc are Map collection's.
- Map collection stores data in key/value pair format.
- In map we can not insert duplicate keys but we can insert duplicate values.
- It maps keys to values, or is a collection of Key-Value pairs
- Map.Entry<K,V> is nested interface of Map<K,V>.
- Following are abstract methods of Map.Entry interface.
 1. K getKey()
 2. V getValue()
 3. V setValue(V value)



Map<K,V>

- Abstract Methods of Map<K,V>
 1. boolean isEmpty()
 2. V put(K key, V value)
 3. void putAll(Map<? extends K, ? extends V> m)
 4. int size()
 5. boolean containsKey(Object key)
 6. boolean containsValue(Object value)
 7. V get(Object key)
 8. V remove(Object key)
 9. void clear()
 10. Set<K> keySet()
 11. Collection<V> values()
 12. Set<Map.Entry<K,V>> entrySet()
- An instance, whose type implements Map.Entry<K,V> interface is called entry instance.



When to use Map<K,V>

Some implementations allow null key and null value (*HashMap* and *LinkedHashMap*) but some does not (*TreeMap*).

The order of a map depends on specific implementations,
e.g *TreeMap* and *LinkedHashMap* have predictable order, while *HashMap* does not.

Maps are perfect for key-value association mapping such as dictionaries. Use Maps when you want to retrieve and update elements by keys, or perform look-ups by keys.

- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

```
Map<Integer, String> hashMap = new HashMap<>();  
Map<Integer, String> linkedHashMap = new LinkedHashMap<>();  
Map<Integer, String> treeMap = new TreeMap<>();
```



Characterstics Map<K,V>

- **HashMap:** this implementation uses a hash table as the underlying data structure. It implements all of the Map operations and allows null values and one null key. This class is roughly equivalent to Hashtable - a legacy data structure before Java Collections Framework, but it is not synchronized and permits nulls. HashMap does not guarantee the order of its key-value elements. Therefore, consider to use a HashMap when order does not matter and nulls are acceptable.
- **LinkedHashMap:** this implementation uses a hash table and a linked list as the underlying data structures, thus the order of a LinkedHashMap is predictable, with insertion-order as the default order. This implementation also allows nulls like HashMap. So consider using a LinkedHashMap when you want a Map with its key-value pairs are sorted by their insertion order.
- **TreeMap:** this implementation uses a red-black tree as the underlying data structure. A TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at creation time. This implementation does not allow nulls. So consider using a TreeMap when you want a Map sorts its key-value pairs by the natural order of the keys (e.g. alphabetic order or numeric order), or by a custom order you specify. So far you have understood the key differences of the 3 major Map's implementations. And the code examples in this tutorial are around them.



Hashtable<K,V>

- It is Map<K,V> collection which extends Dictionary class.
- It can not contain duplicate keys but it can contain duplicate values.
- In Hashtable, Key and value can not be null.
- It is synchronized collection.
- It is introduced in jdk 1.0
- In Hashtable, if we want to use instance non final type as key then it should override equals and hashCode method.



HashMap<K,V>

- It is map collection
- It's implementation is based on Hashtable.
- It can not contain duplicate keys but it can contain duplicate values.
- In HashMap, key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method, we can make it synchronized.
 - Map m = Collections.synchronizedMap(new HashMap(...));
- It is introduced in jdk 1.2.
- Instantiation
 - Map<Integer, String> map = new HashMap<>();
- **Note : In HashMap, if we want to use element of non final type as a key then it should override equals() and hashCode() method.**



LinkedHashMap<K,V>

- It is sub class of HashMap<K,V> class
- Its implementation is based on LinkedList and Hashtable.
- It is Map collection hence it can not contain duplicate keys but it can contain duplicate values.
- In LinkedHashMap, key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method we can make it synchronized.

➤ **Map m = Collections.synchronizedMap(new LinkedHashMap(...));**

- LinkedHashMap maintains order of entries according to the key.
- Instantiation:

➤ **Map<Integer, String> map = new LinkedHashMap<>();**
- It is introduced in jdk 1.4



TreeMap<K,V>

- It is map collection.
- It can not contain duplicate keys but it can contain duplicate values.
- In TreeMap, key must not be null but value can be null.
- Implementation of TreeMap is based on Red-Black Tree.
- It maintains entries in sorted form according to the key.
- It is unsynchronized collection. Using Collections.synchronizedSortedMap() method, we can make it synchronized.
 - **SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));**
- Instantiation:
 - **Map<Integer, String> map = new TreeMap<>();**
- It is introduced in jdk 1.2
- Note : In TreeMap, if we want to use element of non final type as a key then it should implement Comparable interface.





Thank you.

akshita.Chanchlani@sunbeaminfo.com





Object Oriented Programming with Java 8

Akshita Chanchlani

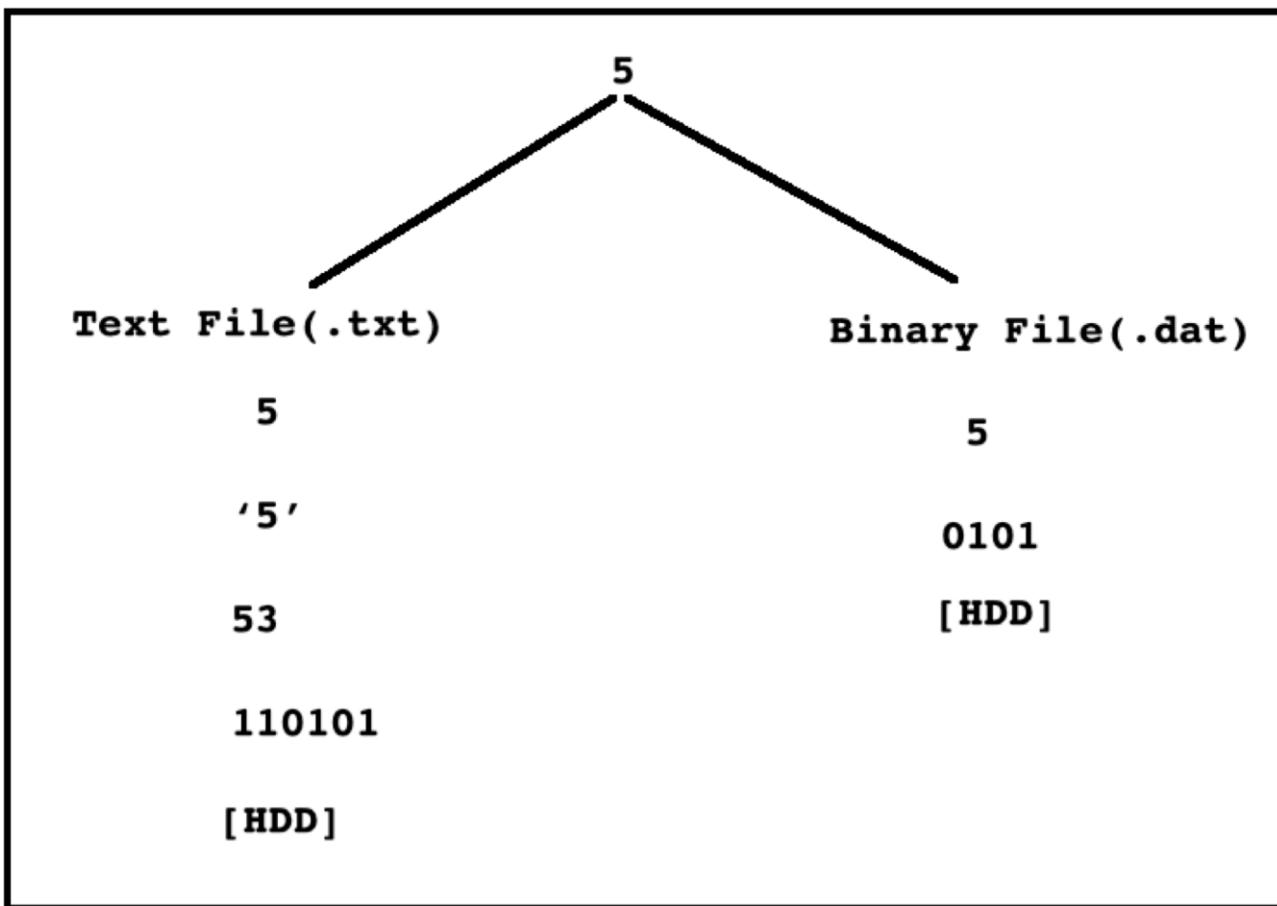


Introduction

- **Variable**
 - It is temporary container, which is used to store data on RAM.
- **File**
 - It is permanent container, which is used to store data on HDD.
- File is operating system resource/non java resource.
- General types of file:
 1. Text File
 2. Binary File



Text File versus Binary File



Text File

- .txt, .rtf, .doc/.docx, .html/.css/.js/.xml, .c/cpp/.java etc.
- We can read text file using any text editor (example notepad).
- It contains data in human readable format.
- It requires more processing hence it is slower in performance.



Binary File

- .jpg/.jpeg/.png, .mp3/.mp4, .o/.obj, .class etc.
- To read binary file we need to use specific program.
- It doesn't contains data in human readable format.
- It requires less processing hence it is faster in performance.



Absolute Path versus Relative Path

- A path of a file from root directory is called as absolute path.
 - Example : **D:\JavaSE8\Demo\src\Program.java**
- A path of a file from current directory is called as relative path.
 - Example : **./src/Program.java**
- Path represents location of file/directory on HDD. It contains:
 1. Root directory name
 2. Sub directories
 3. File Name
 4. Path separator character
 5. Windows : \
 6. Unix\Linux : /

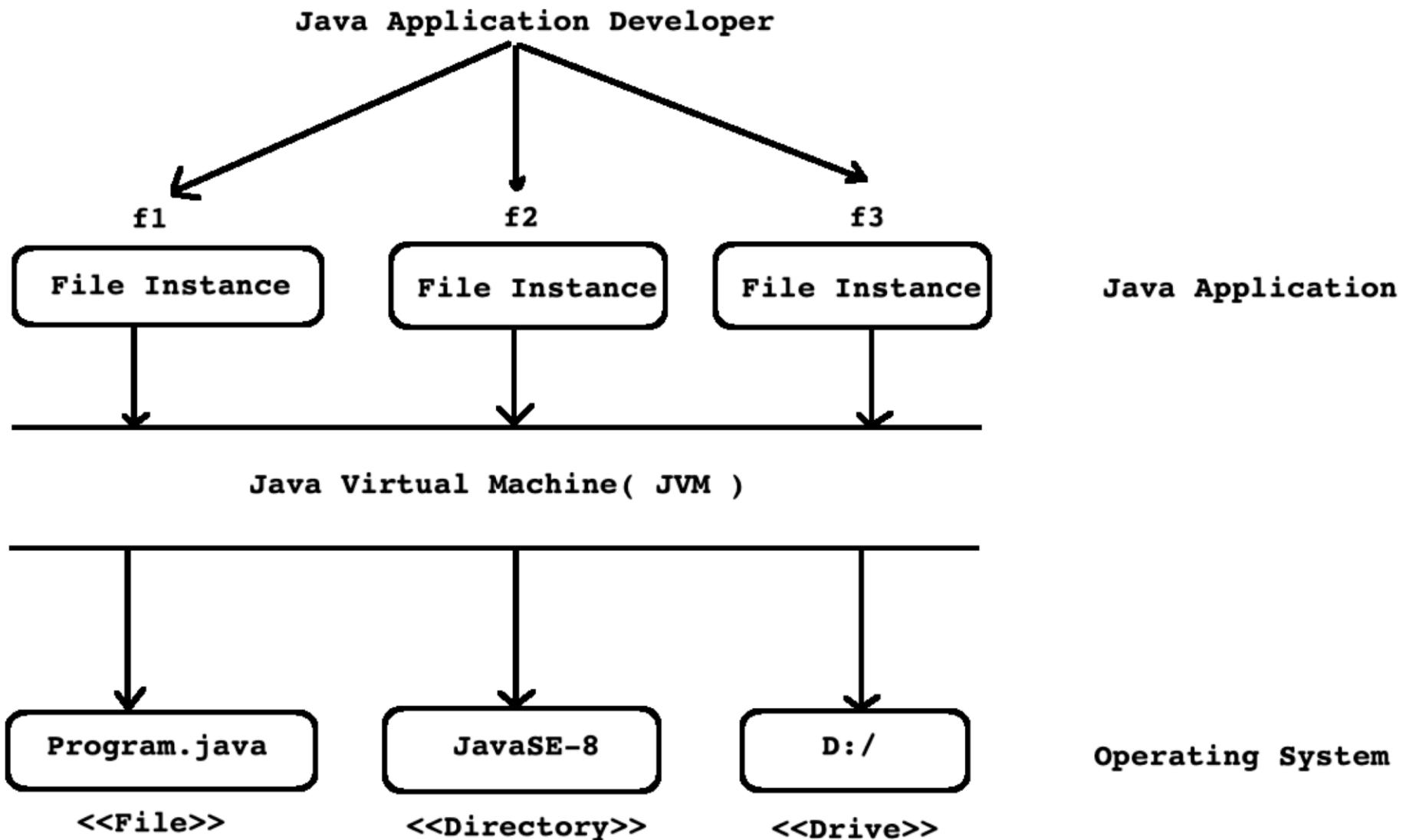


File Handling In Java.

- If we want to manage file in Java then we should use types declared in **java.io** and **java.nio** package.
- Console is a class declared in `java.io` package which represents console/terminal associated with JVM
- File is a class declared in `java.io` package which represents Operating system Drive/directory/file.
- We should use File class:
 1. To create new empty file
 2. To create new empty directory
 3. To delete existing file/directory
 4. To read metadata of file/folder or directory/drive



File Handling In Java.



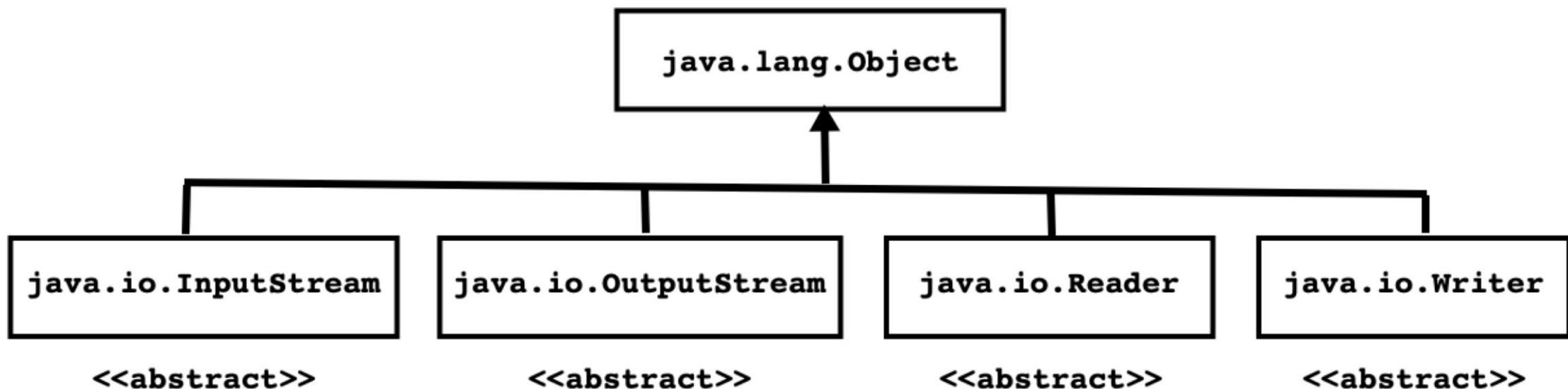
Methods of java.io.File class

1. To create new empty file
 - public boolean **createNewFile()** throws IOException
2. To create new empty directory
 - public boolean **mkdir()**
3. To delete existing file/directory
 - public boolean **delete()**
4. To read metadata of file/directory/drive
 - public String **getName()**
 - public String **getParent()**
 - public String **getPath()**
 - public long **length()**
 - public String[] **list()**
 - public File[] **listFiles()**
 - public long **getTotalSpace()**
 - public long **getUsableSpace()**
 - public long **getFreeSpace()**
 - public boolean **isFile()**
 - public boolean **isDirectory()**
 - public boolean **isHidden()**



File I/O

- If we want to read/write data to and from file then we should use stream.



- Stream is an abstraction(instance/object) which either consume(read)/produce(write) information from source to destination.
- InputStream, OutputStream and their sub classes are used to process binary file.
- Reader, Writer and their sub classes are used to process text file.
- Since all these classes implements Closeable interface, we can use its instance in try with resource syntax.



OutputStream Hierarchy

- java.lang.object
 - java.io.OutputStream (implements java.io.Closeable, java.io.Flushable)
 - java.io.FileoutputStream
 - java.io.FilteroutputStream
 - java.io.BufferedoutputStream
 - java.io.DataoutputStream (implements java.io.Dataoutput)
 - java.io.PrintStream (implements java.lang.Appendable, java.io.Closeable)
 - java.io.ObjectoutputStream (implements java.io.Objectoutput)



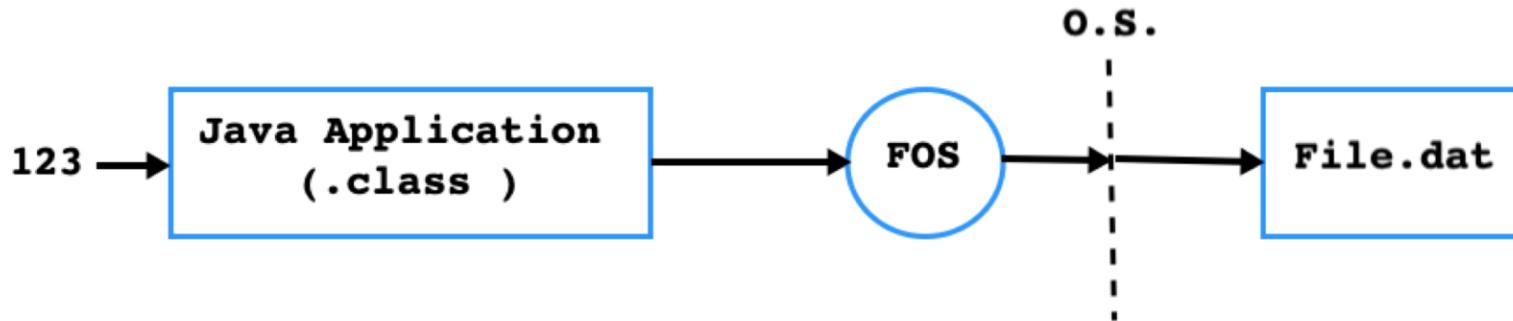
InputStream Hierarchy

- java.lang.object
 - java.io.InputStream (implements java.io.Closeable)
 - java.io.FileInputStream
 - java.io.FilterInputStream
 - java.io.BufferedInputStream
 - java.io.DataInputStream (implements java.io.DataInput)
 - java.io.ObjectInputStream (implements java.io.ObjectInput)

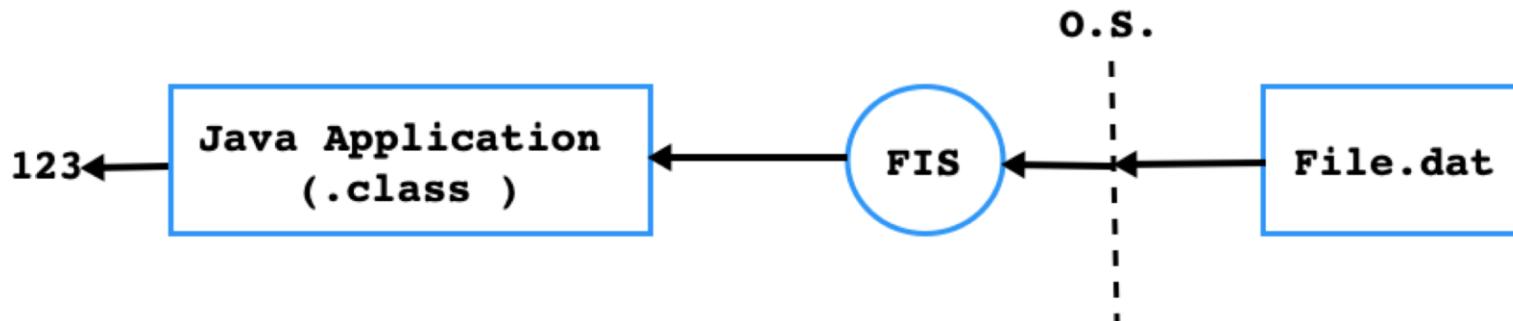


FileInputStream and FileOutputStream

- Using FileOutputStream Instance, we can write single byte at a time into binary file.

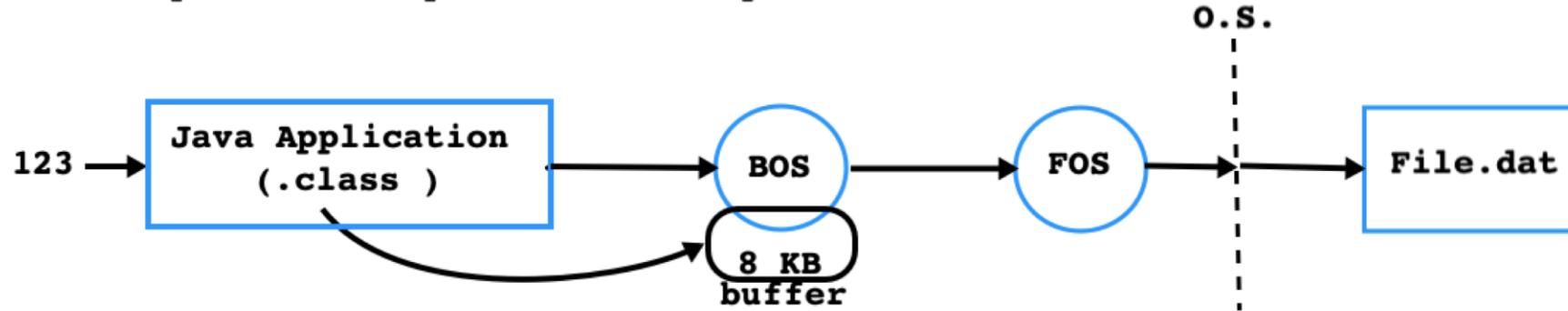


- Using FileInputStream Instance, we can read single byte at a time from binary file.

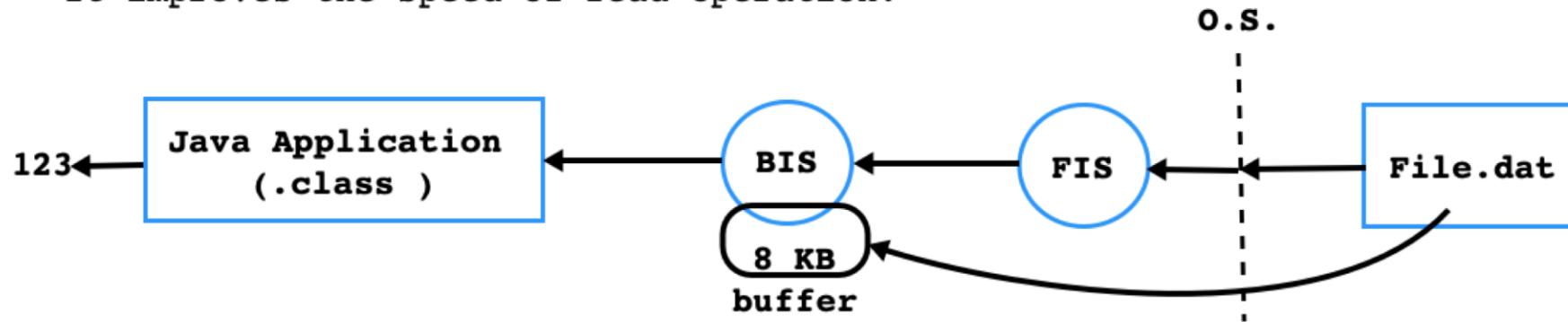


BufferedInputStream and BufferedOutputStream

- Using BufferedOutputStream Instance, we can write multiple bytes at a time into binary file.
- It improves the speed of write operation.

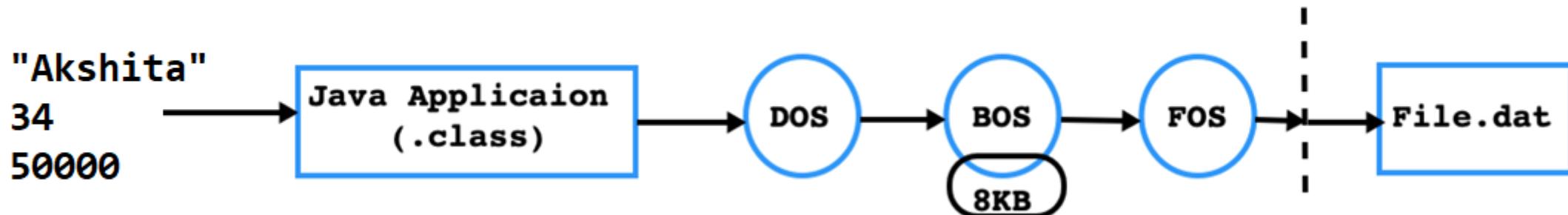


- Using BufferedInputStream Instance, we can read multiple bytes at a time from binary file.
- It improves the speed of read operation.

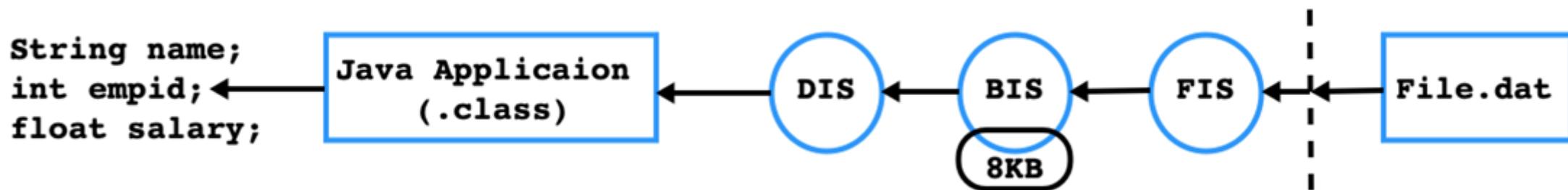


DataInputStream and DataOutputStream

- DataOutputStream instance is used to convert primitive values into binary data.

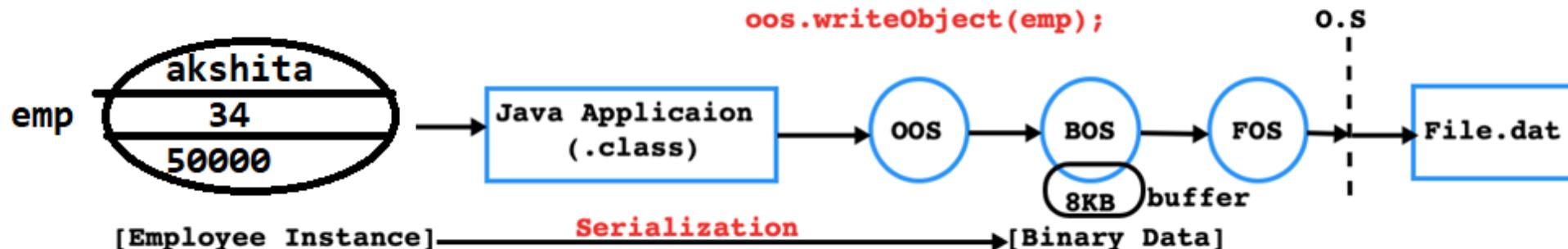


- DataInputStream instance is used to convert binary data into primitive values.

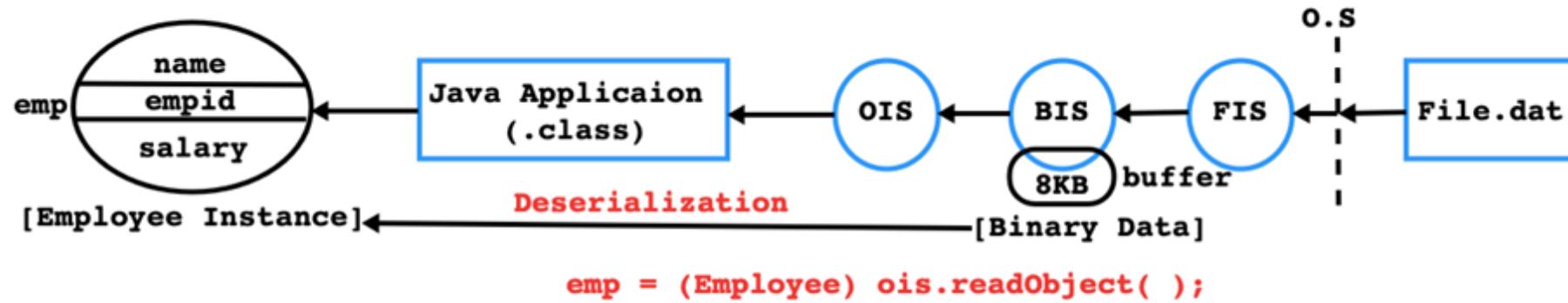


ObjectInputStream and ObjectOutputStream

- ObjectOutputStream instance is used to convert State of Java Instance into binary data.



- ObjectInputStream instance is used to convert binary into Java Instance .



Serialization

- ObjectOutput is a subinterface of DataOutput interface.
 - "**void writeObject(Object obj) throws IOException**" is a method of ObjectOutput interface.
- ObjectOutputStream class implements ObjectOutput interface.
- If we want to convert state of instance of class into binary data then we should use ObjectOutputStream class.
- This ***process of converting state of Java instance into binary data is called as Serialization.***
- If we want to serialize state of Java instance then its type/class must implement java.io.Serializable interface. Otherwise writeObject() method will throw NotSerializableException.
- Serializable is a marker interface.



Serialization

- In case of association, type of contained instance must also implement Serializable interface.

```
class Date implements Serializable{  
    //TODO : Declare fields and methods  
}  
  
class Employee implements Serializable{  
    private Date joinDate; //Date must implement Serializable interface.  
    //TODO : constructor, getter and setter  
}
```

- transient is a keyword in Java.
- If we declare any field transient then JVM do not serialize its state.
- JVM do not serialize state of static and transient field.



Deserialization

- objectInput is a subinterface of DataInput interface.
 - **"Object readObject() throws ClassNotFoundException, IOException"** is a method of objectInput interface.
- objectInputStream class implements objectInput interface.
- If we want to convert binary data into state of instance of class then we should use objectInputStream class.
- This ***process of converting binary data into state of Java instance is called as deserialization.***



SerialVersionUID

- The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException.
- A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:
 - **ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;**



SerialVersionUID

- If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values

```
class Employee implements Serializable{  
    private static final long serialVersionUID = 7504310288177453321L;  
}
```



Writer Hierarchy

- `java.lang.Object`
 - `java.io.Writer` (implements `java.lang.Appendable`,
`java.io.Closeable`, `java.io.Flushable`)
 - `java.io.BufferedWriter`
 - `java.io.CharArrayWriter`
 - `java.io.FilterWriter`
 - `java.io.OutputStreamWriter`
 - `java.io.FileWriter`
 - `java.io.PipedWriter`
 - `java.io.PrintWriter`
 - `java.io.StringWriter`



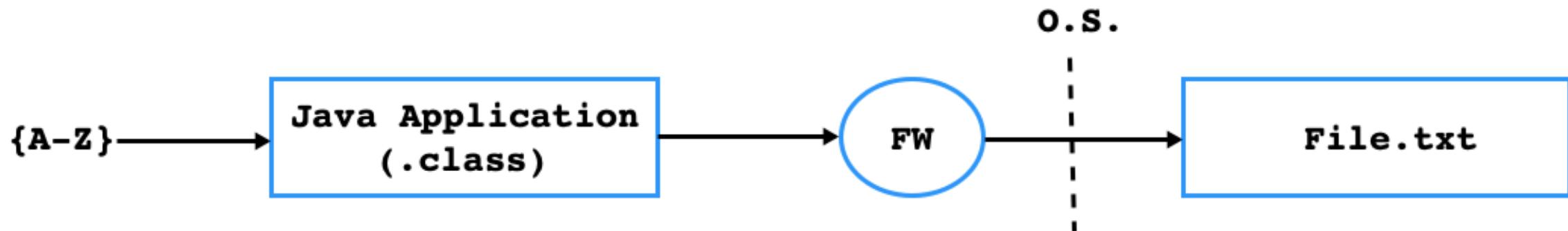
Reader Hierarchy

- java.lang.Object
 - java.io.Reader (implements java.io.Closeable, java.lang.Readable)
 - java.io.BufferedReader
 - java.io.LineNumberReader
 - java.ioCharArrayReader
 - java.io.FilterReader
 - java.io.PushbackReader
 - java.io.InputStreamReader
 - java.io.FileReader
 - java.io.PipedReader
 - java.io.StringReader



FileWriter and FileReader

- `FileWriter` instance is used to write single character at a time inside file.

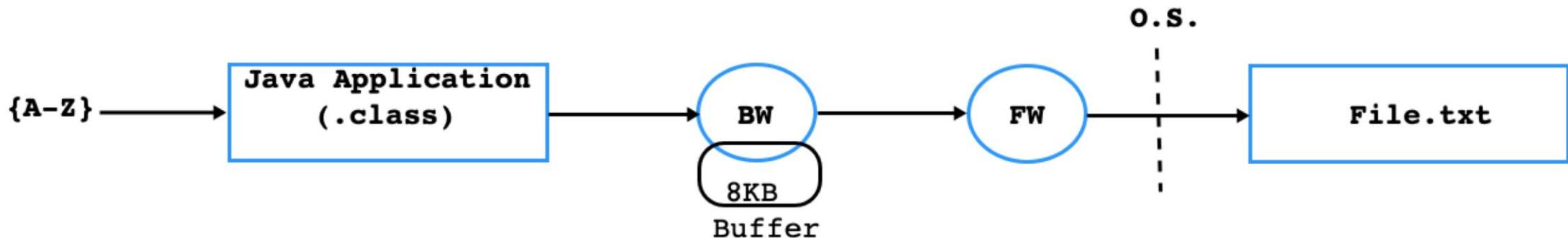


- `FileReader` instance is used to read single character at a time from file.

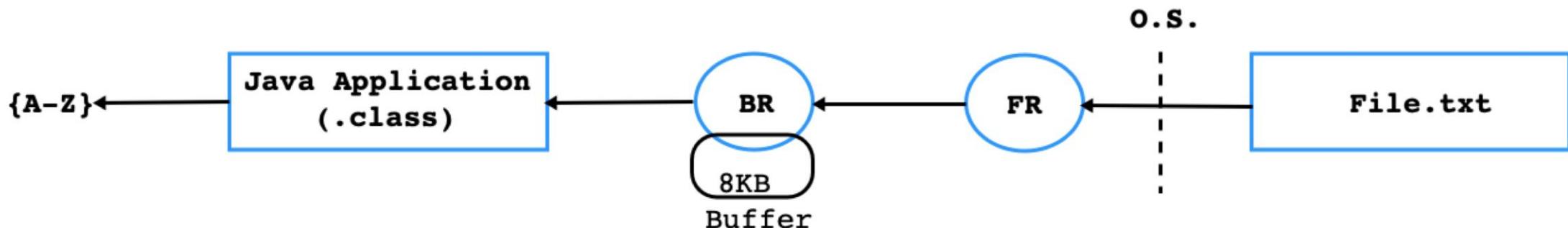


BufferedReader and BufferedWriter

- Using BufferedWriter instance, we can write multiple characters inside file.



- Using BufferedReader instance, we can read multiple characters from file.





Thank You.





Object Oriented Programming with Java 8

Akshita Chanchlani



Annotations

- *Annotations*, a form of metadata, provide data about a program.
- Annotations have a number of uses, among them:
 1. **Information for the compiler** – Annotations can be used by the compiler to detect errors or suppress warnings.
 2. **Compile-time and deployment-time processing** – Software tools can process annotation information to generate code, XML files, and so forth.
 3. **Runtime processing** – Some annotations are available to be examined at runtime.



Types of Annotation

1. Marker Annotation

- Annotation without element is called marker annotation
- Example : @Entity, @Id, @Override, @Deprecated

2. Single-value Annotation

- Annotation which is having single element is called single value annotation.
- Example : @Table(name = "employees")

3. Multi-value Annotation

- Annotation which is having multiple elements is called multi value annotation.
- Example: @Entity(tableName = "vehicles", primaryKey="id")



Example Single value and Multiple Value Annotation

Single-value Annotation

```
@interface Demo{ int value(); } // declaration  
@interface Demo{ int value() default 0; } // declaration along with default value  
@Demo(value=10) // applying the single value annotation
```

Multi-value Annotation

```
@interface Emp{  
int value1();  
String value2();  
String value3();  
}  
}
```

```
@interface Emp{  
int value1() default 1;  
String value2() default "";  
String value3() default "ABC";  
}
```

```
@Emp(value1=20,value2="Akshit  
a",value3="Pune")
```



Predefined Annotation Types

- A set of annotation types are predefined in the Java SE API. Some annotation types are used by the Java compiler, and some apply to other annotations.
- Annotation Types Used by the Java Language
 1. @Deprecated
 2. @Override
 3. @SuppressWarnings
 4. @SafeVarargs
 5. @FunctionalInterface



Predefined Annotation Types

- Annotations that apply to other annotations are called *meta-annotations*.
- There are several meta-annotation types defined in `java.lang.annotation` package.
- Annotations that apply to other annotations:
 1. `@Retention`
 2. `@Documented`
 3. `@Target`
 4. `@Inherited`
 5. `@Repeatable`
 6. `@Native`



Custom Annotation Type

- The annotation type definition looks similar to an interface definition where the keyword interface is preceded by the at sign (@) (@ = AT, as in annotation type).
- Annotation types are a form of *interface*.
- There are few points that we should remember
 1. Element definition should not have any parameter.
 2. Element definition not have any throws clauses
 3. Element definition should return one of the following:
 1. primitive data types,
 2. String,
 3. Class, enum or array of these data types.
 4. We should attach @ just before interface keyword to define annotation.
 5. It may assign a default value to the method.



Retention Policy

- **RetentionPolicy.SOURCE**

- The marked annotation is retained only in the source level and is ignored by the compiler.

- **RetentionPolicy.CLASS**

- The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).

- **RetentionPolicy.RUNTIME**

- The marked annotation is retained by the JVM so it can be used by the runtime environment.



Annotation Target

1. ANNOTATION_TYPE
2. CONSTRUCTOR
3. FIELD
4. LOCAL_VARIABLE
5. METHOD
6. PACKAGE
7. PARAMETER
8. TYPE
9. TYPE_PARAMETER
10. TYPE_USE





Thank You.





Object Oriented Programming with Java 8

Reflection

Akshita Chanchlani



Metadata

- A data which describes other data / data about data is called metadata.
- **Consider metadata for Interface**
 - What is name of the interface.
 - In which package it is declared.
 - Which is super interface of that interface.
 - Which annotations are used on interface.
 - Which is access modifier of the interface.
 - Which members are declared inside interface.



Metadata

- **Consider metadata for Class**

- What is name of the class?
- In which package it is declared?
- Which is super class of it?
- Which interfaces it has implemented?
- Which is access modifier of it?
- Whether it is abstract/final?
- Which annotations are used on it?
- Which members are declared inside class?



Metadata

- **Consider metadata for Field**

- What is name of the field?
- In which class it is declared?
- Which is access modifier of the field?
- Which modifiers used on field(static/final/transient) ?
- Which is type of field?
- Whether it is inherited/declared-only field?
- Which annotations are used on fields?



Metadata

- **Consider metadata for Method**

- What is name of the method?
- In Which class it is declared?
- Whether it is inherited/declared-only/overridden method?
- Which is access modifier of the method?
- Which modifiers used on field(static/final/abstract/synchronized) ?
- Which is return type of the method?
- What is parameter information of the method?
- Which exceptions method can throw?
- Which annotations are used on method?



Application Of Metadata

1. Metadata removes the need for native C/C++ header and library files when compiling because .class file contains bytecode and metadata (information about types defined inside same file and types referenced from other file)
2. IDE uses metadata to help you write code. Its IntelliSense feature parses metadata to tell you what methods, fields, nested types a class contain.
3. Metadata allows an object's fields to be serialized into a memory block, sent to another machine, and then deserialized, re-creating the object's state on the remote machine.
4. Metadata allows the garbage collector to track the lifetime of objects.



Reflection

- It is a Java language feature which provides types(interfaces / classes) that we can use:
 1. To explore metadata
 2. To access private fields of the class
 3. To manage behavior of the application at runtime.
- To use reflection we should use types declared in following package :
 1. `java.lang`
 2. `java.lang.reflect`
- Java Reflection is a process of examining or modifying the run time behavior of a class at run time.
- The `java.lang.Class` class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.
- The `java.lang` and `java.lang.reflect` packages provide classes for java reflection.



Reflection

- The `java.lang.reflect` package provides classes and interfaces for obtaining reflective information about classes and objects.
- Reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use of reflected fields, methods, and constructors to operate on their underlying counterparts, within security restrictions.
- Classes in this package, along with `java.lang.Class` accommodate applications such as debuggers, interpreters, object inspectors, class browsers, and services such as object Serialization and JavaBeans that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class.



Reflection API

- Types declared in `java.lang.reflect` package:
 1. `AccessibleObject`
 2. `Array`
 3. `Constructor`
 4. `Executable`
 5. `Field`
 6. `Method`
 7. `Modifier`
 8. `Parameter`
 9. `Proxy`
 10. `ReflectPermission`
- Type declared in `java.lang` package:
 1. `Class<T>`



Class<T> class

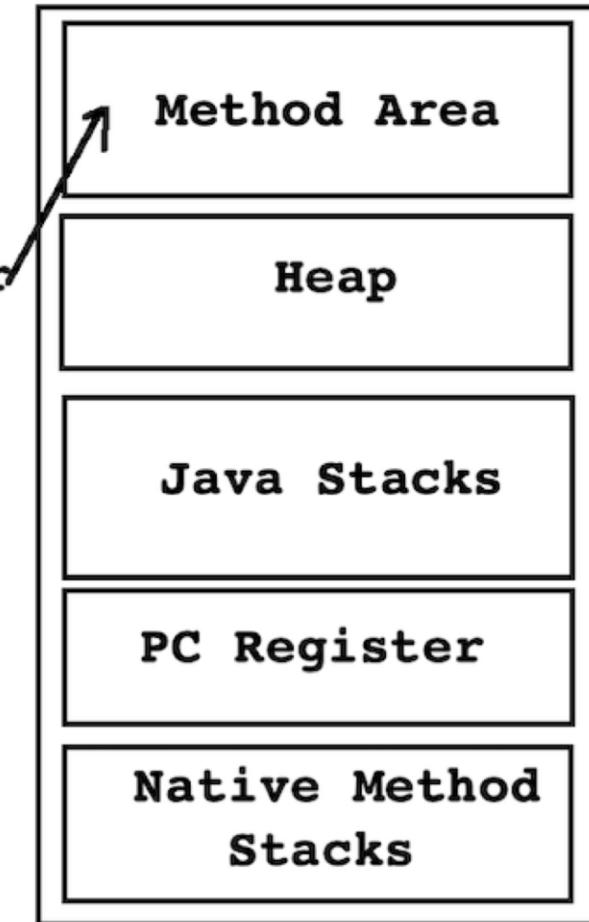
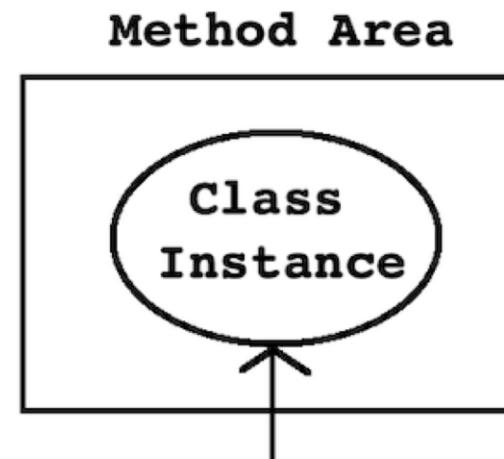
- Class<T> is a final class declared in java.lang package.
- Class has no public constructor. Instead Class objects are constructed automatically by the Java Virtual Machine.
- Instances of the class Class represent classes and interfaces in a running Java application. In other words, instance of java.lang.Class class contains metadata of loaded class.
- Methods of java.lang.Class<T>
 1. public static Class<?> forName(String className) throws ClassNotFoundException
 2. public T newInstance()
 3. public Package getPackage()
 4. public String getName()
 5. public String getSimpleName()



Flow Of Execution

Program.java → **Program.class** → **ClassLoader**

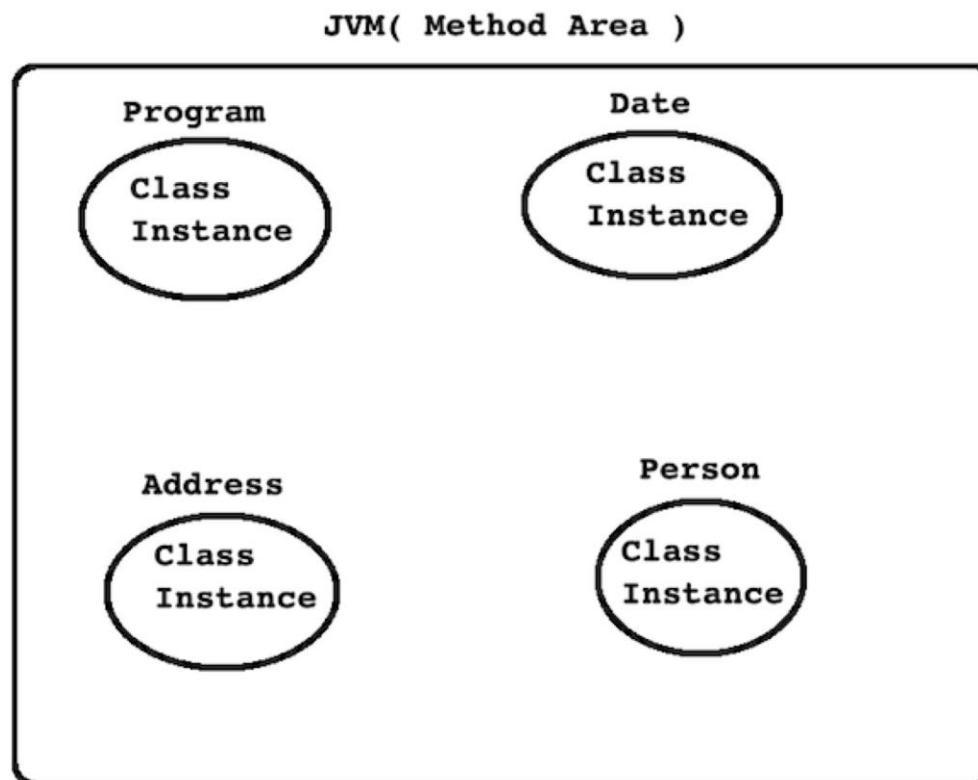
Program.class → **C.L.** →



java.lang.Class class instance associated with Program class

Flow Of Execution

```
class Date{} //Date.class
class Address{} //Address.class
class Person{} //Person.class
    String name;
    Date dob;
    Address address;
}
class Program{ //Program.class
    p.s.v.m( String[] args ){
        case 1: Date
        case 2: Address
        case 3: Person
    }
}
javac Program.java
Java Program( Press enter key ) - ClassLoader -
```



Retrieving Class Objects

- The entry point for all reflection operations is [java.lang.Class](#).
- How to get reference of Class<T> class instance associated with loaded type?

```
1. Using getClass() method
- "public Class<?> getClass( )" is a method of java.lang.Object class.

    Integer number = new Integer( 0 );
    Class<?> c = number.getClass();

2. Using .class syntax

    Class<?> c = Number.class;
    Class<?> c = Integer.class;

3. Using Class.forName( )
- public static Class<?> forName(String className) is a method of Class<T>

    public static void main(String[] args) {
        try( Scanner sc = new Scanner(System.in)){
            System.out.print("F.Q Name :   ");
            String className = sc.nextLine(); //java.io.File
            Class<?> c = Class.forName(className);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```



Get the object of Class class

There are 3 ways to get the instance of Class class.

1. `forName()` method of `Class` class
2. `getClass()` method of `Object` class
3. the `.class` syntax



1) `forName()` method of `Class` class

- is used to load the class dynamically.
- returns the instance of `Class` class.
- It should be used if you know the fully qualified name of class. This cannot be used for primitive types.

2) `getClass()` method of `Object` class

It returns the instance of `Class` class. It should be used if you know the type. Moreover, it can be used with primitives.

3) The `.class` syntax

If a type is available, but there is no instance, then it is possible to obtain a `Class` by appending ".class" to the name of the type. It can be used for primitive data types also.





Thank You.





Object Oriented Programming with Java 8 Functional Programming

Akshita Chanchlani



Functional Programming

Package : java.util.function

Functional Programming (FP) is one of the type of programming pattern that helps the process of building application by using of higher order functions, avoiding shared state, mutable data.

Functional programming is the way of writing s/w applications that uses only pure functions & immutable values.

Need of Functional Programming

1. To write more readable , maintainable , clean & concise code.
2. To use APIs easily n effectively.
3. To enable parallel processing

Declarative vs Imperative :

- Functional programming is a declarative pattern means that the program logic is expressed without explicitly describing the flow control.(you will just have to specify what's to be done)
- Imperative programs spend lines of code describing the specific steps used to achieve the desired results by following flow control. (you will have to specify what's to be done & how)
- Declarative programs remove the flow control process, and instead spend lines of code describing the data flow.



Functions as Objects

- store in a variable, pass as a parameter, return as a result.
- Only variables and objects can be stored, passed, and returned.
- Functions can be represented as objects in Java. That is called as Functional Interface.
- We implement classes from Functional Interfaces.
- Then, we can create objects.
- These objects are called **First-Class Functions**. Because It is said that it is because functions are finally being treated as *first-class citizens*. When encapsulated in an object, they can finally be stored, returned, and used as parameters to other functions.



Functions as Expressions

Functions can be expressions; therefore, they can exist inside methods, as the expressions that we create combining operators and values

We will need a new kind of expression called **Lambda Expressions**. They are functions too.



Types of Functional Programming

- Streams functional Programming
- Lambda Expressions functional Programming
- Method Reference functional Programming



Streams Functional Programming

```
objectName.stream();
```



Lambda Expressions

- Lambda expression used to represent a method interface with an expression.
- It helps to iterate, filtering and extracting data from collections.
- Lambda expression interface implementation is a [functional interface](#).
- It reduces a lot of code.
- Lambda expression treated as a function so java compiler can't create .class

Syntax:

```
(arguments) ->  
{  
//code for implementation  
}
```

Arguments: argument-list can be have values or no values

Example: arguments1, arguments2, arguments3,.....

->: Joins code implementation and arguments.



Lambda expression with a single argument

Syntax:

```
(argument1) ->
{
//code for implementation
}
```

Lambda expression without argument

Syntax:

```
() ->
{
//code for implementation
}
```



Method Reference

- Method reference used to refer to a method of functional interface.
- It is one more easy form of a lambda expression.
- If you use every time lambda expression to point a method, we can use method reference in place of method reference.

Syntax:

Class-Name:: static method name



Functional Programming Concepts

• Higher-order functions:

In functional programming, functions are to be considered as first-class citizens. That is, so far in the legacy style of coding, we can do below stuff with objects.

- We can pass **objects** to a function.
- We can create **objects** within function.
- We can return **objects** from a function.
- We can pass a **function** to a function.
- We can create a **function** within function.
- We can return a **function** from a function.

• Pure functions:

A function is called pure function if it always returns the same result for same argument values and it has no side effects like modifying an argument (or global variable) or outputting something.

• Lambda expressions

A Lambda expression is an anonymous method that has mutability at very minimum and it has only a parameter list and a body. The return type is always inferred based on the context. Also, make a note, Lambda expressions work in parallel with the functional interface.

The syntax of a lambda expression , (parameter) -> body



First-Class and Higher-Order Functions

- A programming language is said to have first-class functions if it treats functions as first-class citizens. Basically, it means that **functions are allowed to support all operations typically available to other entities**.
- These include assigning functions to variables, passing them as arguments to other functions, and returning them as values from other functions.
- This property makes it possible to define higher-order functions in functional programming. **Higher-order functions are capable of receiving function as arguments and returning a function as a result**.
- This further enables several techniques in functional programming like function composition and currying.
- Traditionally it was only possible to pass functions in Java using constructs like functional interfaces or anonymous inner classes. Functional interfaces have exactly one abstract method and are also known as Single Abstract Method (SAM) interfaces.

For example we have to provide a custom comparator to **Collections.sort** method:

```
Collections.sort(numbers, new Comparator<Integer>() {  
    @Override public int compare(Integer n1, Integer n2) {  
        return n1.compareTo(n2); } });
```



Functional Interface

- An interface which has exactly single abstract method(SAM) is called functional interface.

Eg. Runnable,Comparable,Comparator,Iterable etc.

- Annotation for Functional Interface
`@FunctionalInterface`
- Functional i/f references can be substituted by lambda expressions, method references, or constructor references.



Lambda Expression

- Lambdas are the most important new addition
- **Java treats a lambda expression as an *Object*,** which is, in fact, the true first-class citizen in Java.
- A big challenge was to introduce lambdas without requiring recompilation of existing binaries

For example above example with Lambda Expression:

```
Collections.sort(numbers, (n1, n2) -> n1.compareTo(n2));
```

this is more concise and understandable



Syntax of Java 8 Lambdas

- . A Java 8 lambda is basically a method in Java without a declaration usually written as (parameters) -> { body }. Examples,
 1. `(int x, int y) -> { return x + y; }`
 2. `x -> x * x`
 3. `() -> x`
- . A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context.
- . Parenthesis are not needed around a single parameter.
- . `()` is used to denote zero parameters.
- . The body can contain zero or more statements.
- . Braces are not needed around a single-statement body.

Benefits of Lambdas in Java 8

- Enabling functional programming
- Writing leaner more compact code
- Facilitating parallel programming
- Developing more generic, flexible and reusable APIs
- Being able to pass behaviors as well as data to functions



Example 1: Print a list of integers with a lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);  
  
intSeq.forEach(x -> System.out.println(x));
```

x -> System.out.println(x) is a lambda expression that defines an anonymous function with one parameter named x of type Integer



Example 2: A multiline lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
    x += 2;
    System.out.println(x);
});
```

Braces are needed to enclose a multiline body in a lambda expression.



Example 3: A lambda with a defined local variable

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
    int y = x * 2;
    System.out.println(y);
});
```

- Just as with ordinary functions, you can define local variables inside the body of a lambda expression

Example 4: A lambda with a declared parameter type

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach((Integer x -> {
    x += 2;
    System.out.println(x);
}));
```

- You can, if you wish, specify the parameter type.

Variable Capture

- Lambdas can interact with variables defined outside the body of the lambda
- Using these variables is called variable capture

Local Variable Capture Example

```
List<Integer> intSeq = Arrays.asList(1,2,3);

int var = 10;
intSeq.forEach(x -> System.out.println(x + var));
```

- Note: local variables used inside the body of a lambda must be final or effectively final

Static Variable Capture Example

```
public class Demo {  
    private static int var = 10;  
    public static void main(String[] args) {  
        List<Integer> intSeq = Arrays.asList(1,2,3);  
        intSeq.forEach(x -> System.out.println(x + var));  
    }  
}
```

Method References

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method

Conciseness with Method References

We can rewrite the statement

```
intSeq.forEach(x -> System.out.println(x));
```

more concisely using a method reference

```
intSeq.forEach(System.out::println);
```

What is a stream?

A stream is “a sequence of elements from a source that supports data processing operations.” Internal iteration.

Collections are held in memory space. You need the entire collection to iterate. SPACE

Streams are created on-demand and they are infinite. TIME

Declarative— More concise and readable

Composable— Greater flexibility

Parallelizable— Better performance

Stream API

- The new `java.util.stream` package provides utilities to support functional-style operations on streams of values.
- A common way to obtain a stream is from a collection:
`Stream<T> stream = collection.stream();`
- Streams can be sequential or parallel.
- Streams are useful for selecting values and performing actions on the results.

Sequence of Elements

Sequence of elements— Like a collection, a stream provides an interface to a sequenced set of values of a specific element type. Because collections are data structures, they're mostly about storing and accessing elements, but streams are **also** about expressing computations such as filter, sorted, and map.

Source

Source— Streams consume from a data-providing source such as collections, arrays, or I/O resources.

Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.

Data processing

Data processing— Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on. Stream operations can be executed either **sequentially** or in **parallel**.

Pipelining

Pipelining— Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline. This enables certain optimizations such as laziness and short-circuiting. A pipeline of operations can be viewed as a database-like query on the data source.

Stream Operations

- An intermediate operation keeps a stream open for further operations. Intermediate operations are lazy.
- A terminal operation must be the final operation on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable.

Example Intermediate Operations

- `filter` excludes all elements that don't match a Predicate.
- `map` performs a one-to-one transformation of elements using a Function.

A Stream Pipeline

A stream pipeline has three components:

1. A source such as a Collection, an array, a generator function, or an IO channel;
2. Zero or more intermediate operations; and
3. A terminal operation

Stream Example

```
int sum = widgets.stream()  
    .filter(w -> w.getColor() == RED)  
    .mapToInt(w -> w.getweight())  
    .sum();
```

Here, `widgets` is a `Collection<widget>`. We create a stream of `widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of `int` values representing the weight of each red widget. Then this stream is summed to produce a total weight.

Parting Example: Using lambdas and stream to sum the squares of the elements on a list

```
List<Integer> list = Arrays.asList(1,2,3);
```

```
int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();
```

```
System.out.println(sum);
```

- Here `map(x -> x*x)` squares each element and then `reduce((x,y) -> x + y)` reduces all elements into a single number



Thank You.

akshita.chanchlani@sunbeaminfo.com





Object Oriented Programming with Java 8

Multithreading

Akshita Chanchlani



Process Introduction

- Process(also called as task)
 - Program in execution is called as process.
 - Running instance of a program is also called as process.
- Ability of an operating system to execute single process at a time is called as single tasking operating system.
 - Example : Microsoft Disk Operating System(MSDOS)
- Ability of an operating system to execute multiple processes at a time is called as multi tasking operating system
 - Example : Microsoft Windows, Unix/Linux, Mac OS
- This ability to run multiple processes or application at a time is called as concurrency in oops.

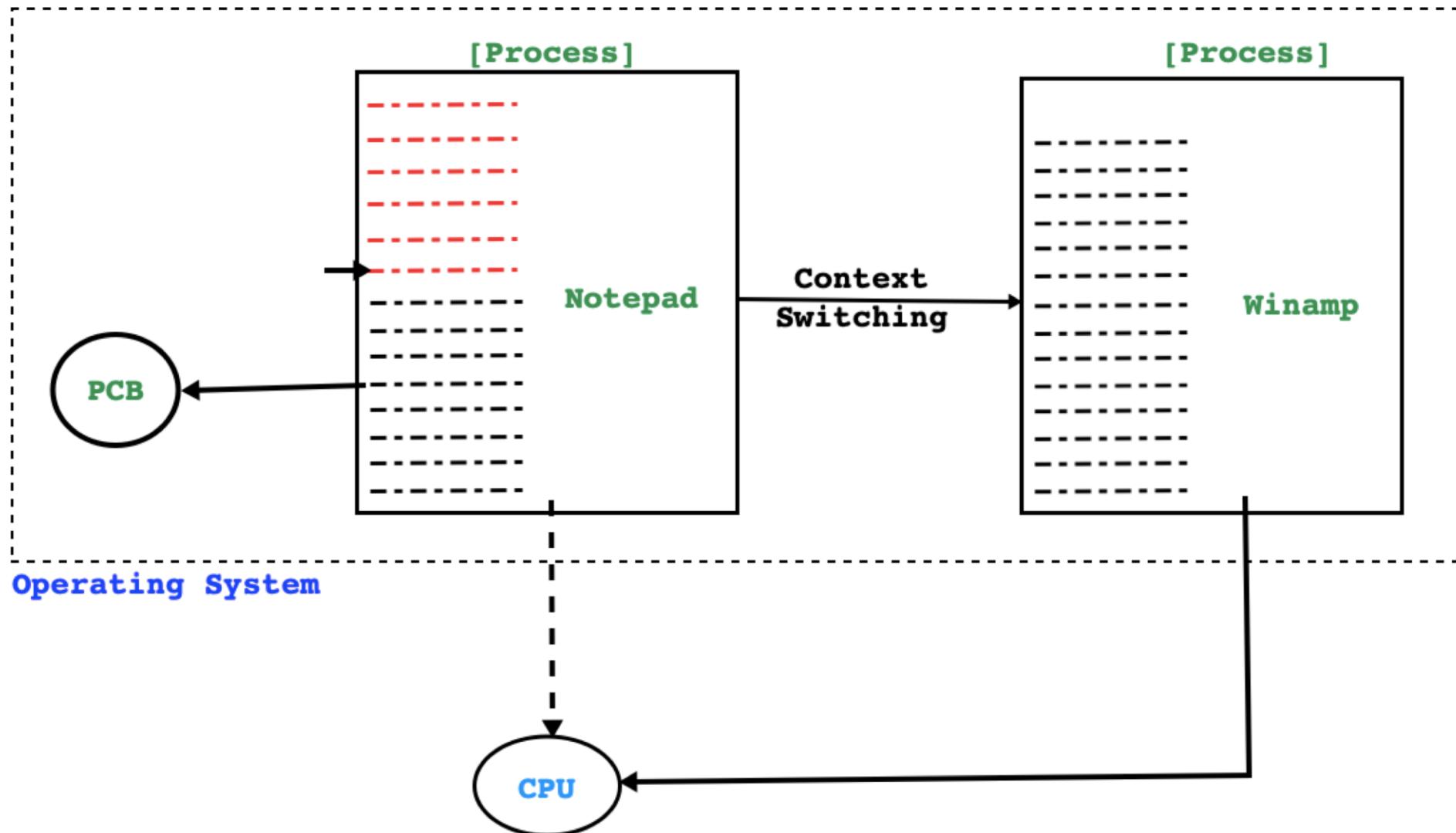


Thread Introduction

- Thread(also called as task)
 - Light weight process or sub process is also called thread.
 - In the context of Java, thread is a separate path of execution which runs independently.
- Thread is a operating system / Non Java resource.
- If we want to utilize hardware resources(e.g. CPU) efficiently then we should use thread.
- If application take help of single thread to execute application then it is called single threaded application.
- If application take help of multiple threads to execute application then it is called multi threaded application.
- Every Java application is multithread.



Process Based Multitasking

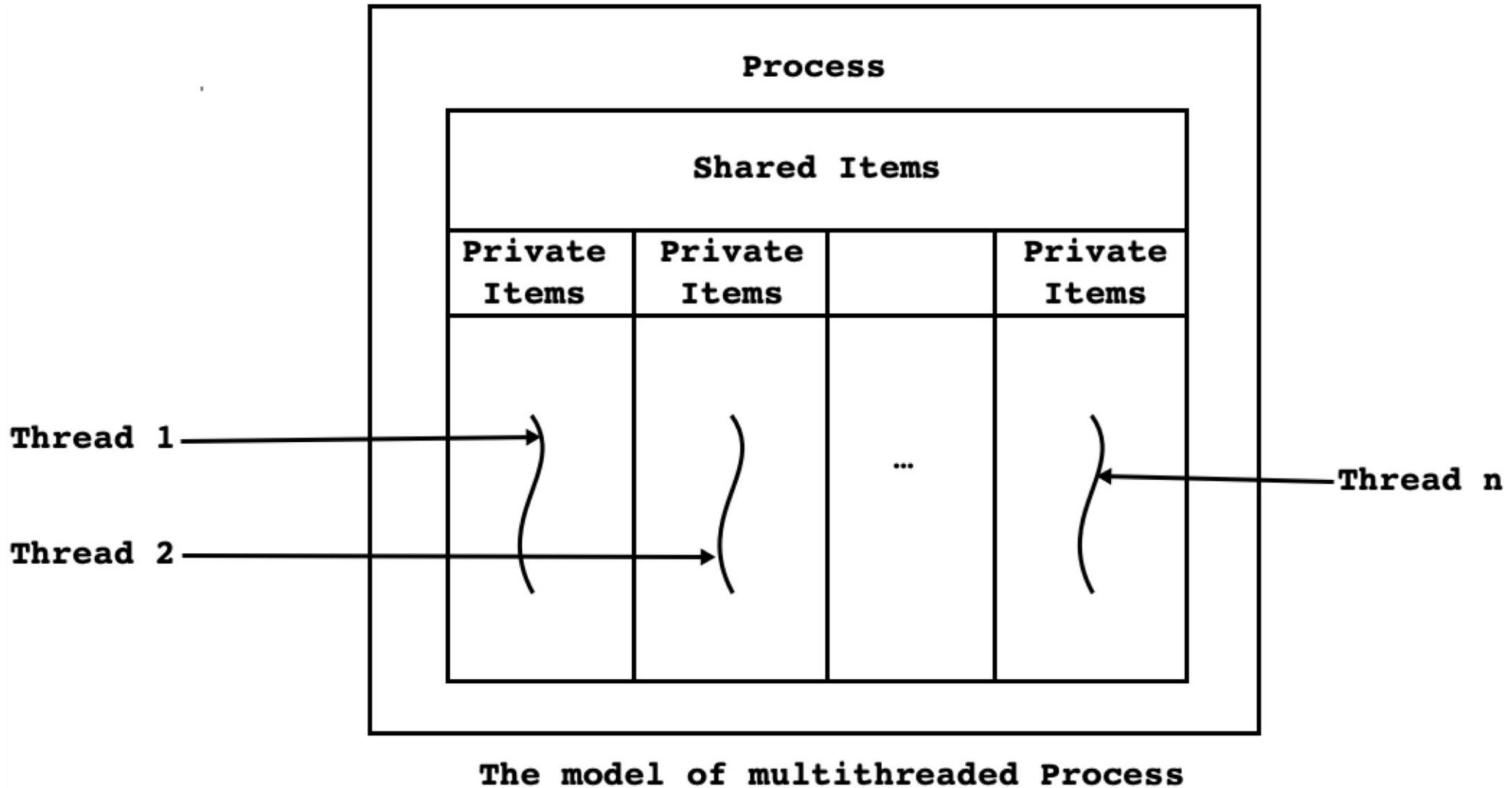


Process Based Multitasking

- If we want, operating system should execute multiple processes at the same time then OS must first save state of current process into process control block(PCB). Only after that control of CPU can be transferred to the another process. It is called as context switching.
- Creating / disposing process and context switching is a heavy task. Hence process Based multitasking is called heavy weight multitasking.



Thread Based Multitasking



Thread Based Multitasking

- In comparison with process, thread creation and disposal/termination take less time.
- To access shared items of process, thread do not require context switching. Hence thread based multitasking is called as light weight multitasking.



Java is *Multithreaded* Programming Language

- When we start execution of Java application, JVM starts execution of main thread and garbage collector.
 - Main thread
 1. It is a *user thread*(also called as non daemon thread).
 2. It is responsible for invoking main method.
 - Garbage Collector(also called as *finalizer*)
 1. It is a *daemon thread*(also called as background thread).
 2. It is responsible for deallocating/releasing/reclaiming memory of unused instances.
- Thread is an OS resource. To access OS thread in application, Java application developer need not to do native programming. Sun/Oracle developers has already developed framework to access OS thread in Java application.



Thread Types

- Java offers two types of thread:
 1. User Thread
 2. Daemon Thread
- User threads are high priority threads. JVM will not terminate until all user threads gets terminated.
- Daemon threads are low priority threads. When thread is marked as daemon, JVM doesn't wait to finish its task.
- Thread inherits properties from its parent. Main thread is user thread hence, if we create any thread from main then it is by default user thread. If we create any thread from daemon thread then it is by default daemon thread.
- Using setDaemon() method we can convert user thread to deamon or vice versa.



Thread Types

- Using `isDaemon()` method, we can check whether thread is user thread or daemon thread.
- We can use `setDaemon()` method to change type of thread. This method should be called after creating thread instance and before calling `start` method on thread instance.

```
public class Program {  
    public static void main(String[] args){  
        Thread thread = Thread.currentThread();  
        System.out.println(thread.toString()); //Thread[main,5,main]  
        if( !thread.isDaemon())  
            thread.setDaemon(true); //IllegalThreadStateException  
    }  
}
```



Thread API

- **Packages**

1. java.lang
2. java.util.concurrent

- **Interface**

- java.lang.Runnable

- **Class(es)**

- Thread
 - ThreadGroup
 - ThreadLocal

- **Enum**

- Thread.State

- **Exception**

- IllegalThreadStateException
 - IllegalMonitorStateException
 - InterruptedException



Runnable

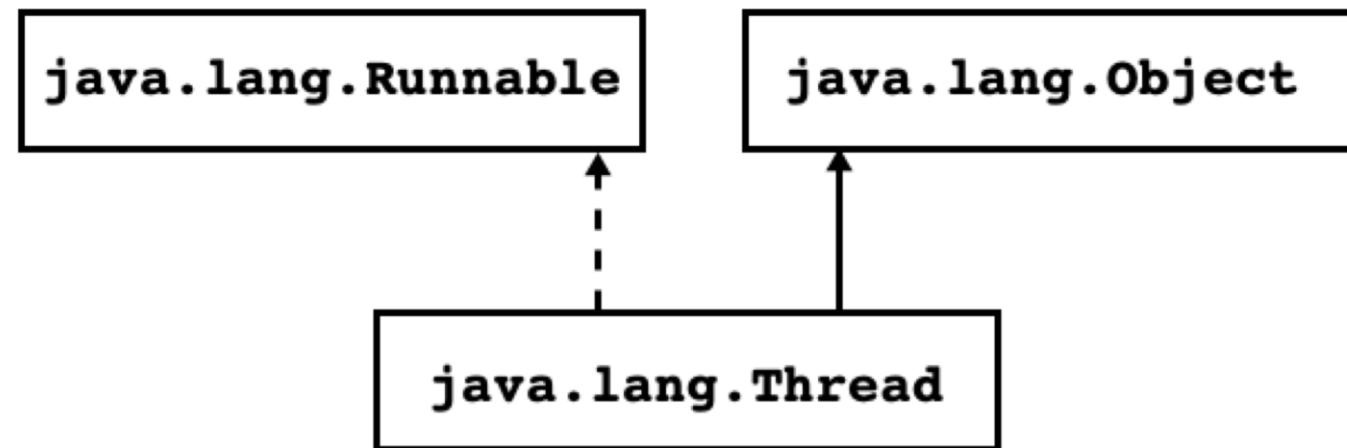
- It is a functional interface declared in `java.lang` package.
- “`void run()`” is a method / functional method of Runnable interface.
- Without sub classing `java.lang.Thread`, if we want to create thread then we should use Runnable interface.

```
class Task implements Runnable{
    private Thread thread;
    public Task( String name ) {
        this.thread = new Thread(this, name);
        this.thread.start();
    }
    @Override
    public void run() {
        //TODO : Write Business Logic Here
    }
}
```



Thread

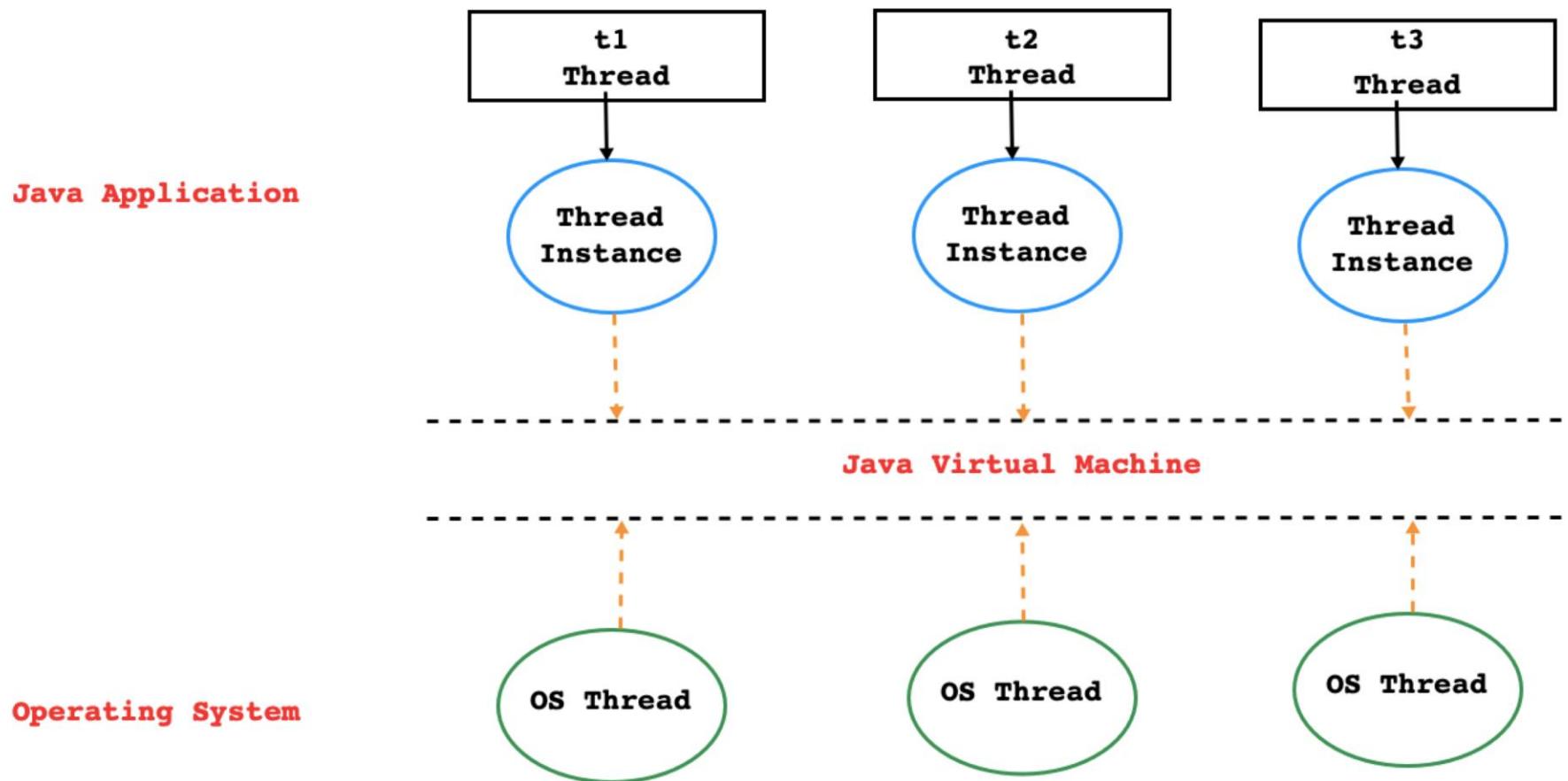
- It is a non final and concrete class declared in `java.lang` Package.
- In Java language, we can create thread using two ways:
 1. `java.lang.Runnable` interface
 2. `java.lang.Thread` class



- `Java.lang.Thread` is managed version(Java) of unmanaged thread(oS.)

Thread

- Instance of `java.lang.Thread` class is not an operating System thread. It represents operating System(actually kernel) thread.



Thread

- Thread creation using `java.lang.Thread` class:

```
class Task extends Thread{
    public Task( String name ) {
        //super( name );    //or
        this.setName(name);
        this.start();
    }
    @Override
    public void run() {
        //TODO : Write Business Logic Here
    }
}
```



Thread.State

- State is a static nested type(Enum) declared inside `java.lang.Thread` class.
- A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.
 - **NEW**
 - A thread that has not yet started is in this state.
 - **RUNNABLE**
 - A thread executing in the Java virtual machine is in this state.
 - **BLOCKED**
 - A thread that is blocked waiting for a monitor lock is in this state.
 - **WAITING**
 - A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
 - **TIMED_WAITING**
 - A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
 - **TERMINATED**
 - A thread that has exited is in this state.



Fields and Constructor's of Thread Class

- **Field(s)**

1. public static final int MIN_PRIORITY
2. public static final int NORM_PRIORITY
3. public static final int MAX_PRIORITY

- **Constructor(s)**

1. public Thread()
2. public Thread(String name)
3. public Thread(Runnable target)
4. public Thread(Runnable target, String name)
5. public Thread(ThreadGroup group, String name)
6. public Thread(ThreadGroup group, Runnable target)
7. public Thread(ThreadGroup group, Runnable target, String name)



Methods Of Thread Class

1. public static Thread **currentThread()**
2. public static void **sleep**(long millis) throws InterruptedException
3. public void **start()**
4. public static void **yield()**
5. public final String **getName()**
6. public final void **setName**(String name)
7. public final int **getPriority()**
8. public final void **setPriority**(int newPriority)
9. public Thread.State **getState()**
10. public static boolean **interrupted()**
11. public void **interrupt()**
12. public final boolean **isAlive()**
13. public final boolean **isDaemon()**
14. public final void **setDaemon**(boolean on)
15. public final void **join**(long millis) throws InterruptedException
16. public void **setUncaughtExceptionHandler**(Thread.UncaughtExceptionHandler eh)



Main Thread

```
public class Program {  
    public static void main(String[] args){  
        Thread thread = Thread.currentThread();  
        System.out.println(thread.toString()); //Thread[main,5,main]  
        System.out.println("Name : "+thread.getName()); //main  
        System.out.println("Priority: "+thread.getPriority());//5  
        System.out.println("Group : "+thread.getThreadGroup().getName());//main  
        System.out.println("State : "+thread.getState().name()); //RUNNABLE  
        System.out.println("Type : "+(thread.isDaemon()?"Daemon":"User")); //User  
        System.out.println("Status : "+(thread.isAlive()?"Alive":"Dead")); //Alive  
    }  
}
```



Finalizer / GC

```
class ResourceType{
    @Override
    protected void finalize() throws Throwable {
        Thread thread = Thread.currentThread();
        System.out.println(thread.toString()); //Thread[main,5,main]
        System.out.println("Name      : "+thread.getName()); //Finalizer
        System.out.println("Priority: "+thread.getPriority());//8
        System.out.println("Group     : "+thread.getThreadGroup().getName());//system
        System.out.println("State     : "+thread.getState().name()); //RUNNABLE
        System.out.println("Type      : "+(thread.isDaemon()?"Daemon":"User")); //Daemon
        System.out.println("Status    : "+(thread.isAlive()?"Alive":"Dead")); //Alive
    }
}
public class Program {
    public static void main(String[] args){
        ResourceType rt = new ResourceType();
        rt = null;
        System.gc();
    }
}
```



Thread Creation

Using Runnable Interface

```
class Task implements Runnable{  
    private Thread thread;  
    public Task( String name ) {  
        this.thread = new Thread(this, name);  
        this.thread.start();|  
    }  
    @Override  
    public void run() {  
        //TODO : Write Business Logic Here  
    }  
}
```

Using Thread class

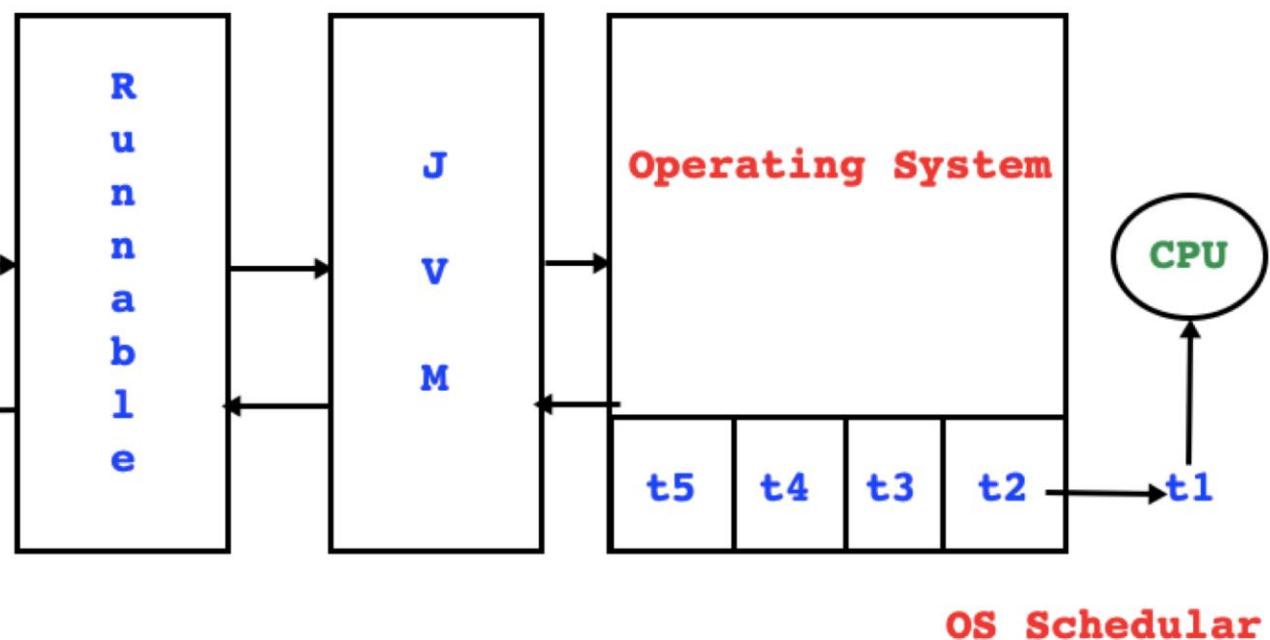
```
class Task extends Thread{  
    public Task( String name ) {  
        //super( name );    //or  
        this.setName(name);  
        this.start();  
    }  
    @Override  
    public void run() {  
        //TODO : Write Business Logic Here  
    }  
}
```



Relation Between start() and run() method.

- start() method do not call run method.
- If we call start() method on thread instance then JVM ask OS to map java thread instance to OS thread. When OS scheduler assign CPU to OS thread then OS ask JVM to invoke run() method on corresponding java instance.

```
class Task implements Runnable{
    Thread thread;
    public Task(String name) {
        thread = new Thread(this, name);
        thread.start(); -> [Programmer invoke start()]
    }
    @Override
    public void run() { <- [JVM Invoke run()]
        //TODO : Business Logic
    }
}
```



Thread termination

- If control come out of run() method then thread gets terminated.
- In following cases thread can come out of run() method:
 1. If JVM execute run method successfully.
 2. If JVM throw exception during execution of run method.
 3. If JVM execute return statement during execution of run method.



Blocking calls/Operations

- Following calls are considered as blocking calls in multithreaded programming:
 1. sleep()
 2. suspend() [**Deprecated**]
 3. wait()
 4. join
 5. input calls
- **sleep() method**
 - It is overloaded static method of java.lang.Thread class.
 - Syntax:
 - public static void sleep(long milliSeconds) throws InterruptedException
 - If we want to suspend current thread for specified number of milliseconds then we should use sleep method.
- **suspend() method**
 - It is deprecated non static method of java.lang.Thread class.
 - Syntax:
 - public final void suspend()
 - If we want to suspend any thread for infinite time then we should use suspend(). method.



Blocking calls/Operations

- Following calls are considered as blocking calls in multithreaded programming:
 1. sleep()
 2. suspend() [**Deprecated**]
 3. wait()
 4. join
 5. input calls
- **sleep() method**
 - It is overloaded static method of java.lang.Thread class.
 - Syntax:
 - public static void sleep(long milliSeconds) throws InterruptedException
 - If we want to suspend current thread for specified number of milliseconds then we should use sleep method.
- **suspend() method**
 - It is deprecated non static method of java.lang.Thread class.
 - Syntax:
 - public final void suspend()
 - If we want to suspend any thread for infinite time then we should use suspend(). method.



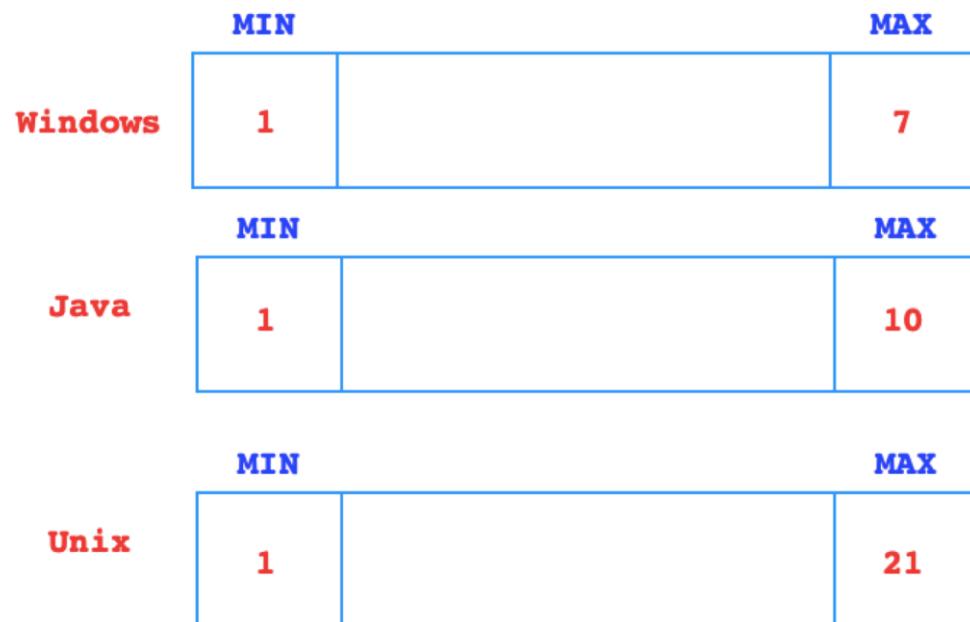
Thread Priorities

- In the Java programming language, every thread has a *priority*.
- Whenever the thread scheduler has a chance to pick a new thread, it prefers threads with higher priority.
- By default, a thread inherits the priority of the thread that constructed it. We can increase or decrease the priority of any thread with the `setPriority()` method.
- We can set the priority to any value between `MIN_PRIORITY` and `MAX_PRIORITY`.
- If the priority is not in the range `MIN_PRIORITY` to `MAX_PRIORITY` the `setPriority()` method throws **IllegalArgumentException**



Thread Priorities

- Thread priorities are *highly system dependent*.
- When the virtual machine relies on the thread implementation of the host platform, the Java thread priorities are mapped to the priority levels of the host platform, which may have more or fewer thread priority levels.
- In the Oracle JVM for Linux, thread priorities are ignored altogether—all threads have the same priority.



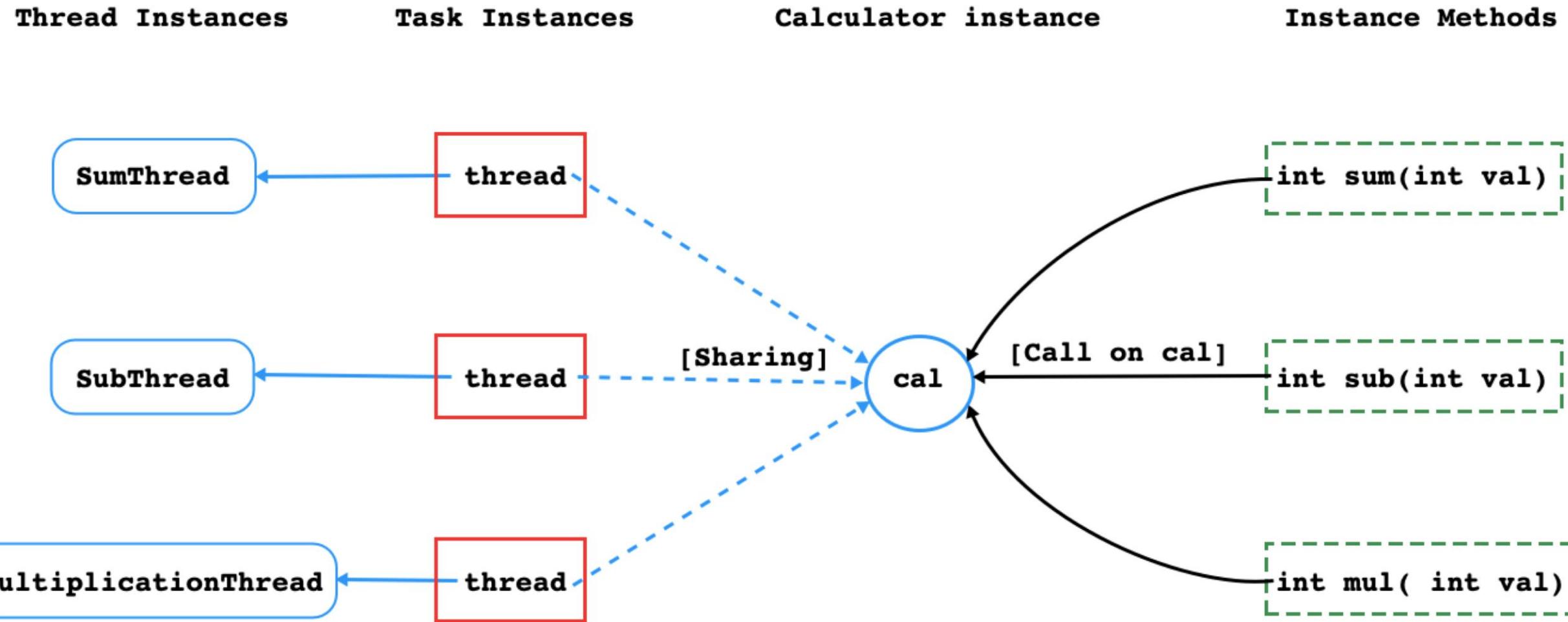
	Java	MS Windows	Unix
t1	8	7	16
t2	7	7	14
t3	2	2	4

Race Condition

- In most practical multithreaded applications, two or more threads need to share access to the same data. What happens if two threads have access to the same object and each calls a method that modifies the state of the object? As you might imagine, the threads can step on each other's toes.
- Depending on the order in which the data were accessed, corrupted objects can result. Such a situation is often called a *race condition*.
- To avoid corruption of shared data by multiple threads, you must learn how to *synchronize the access*.



Race Condition



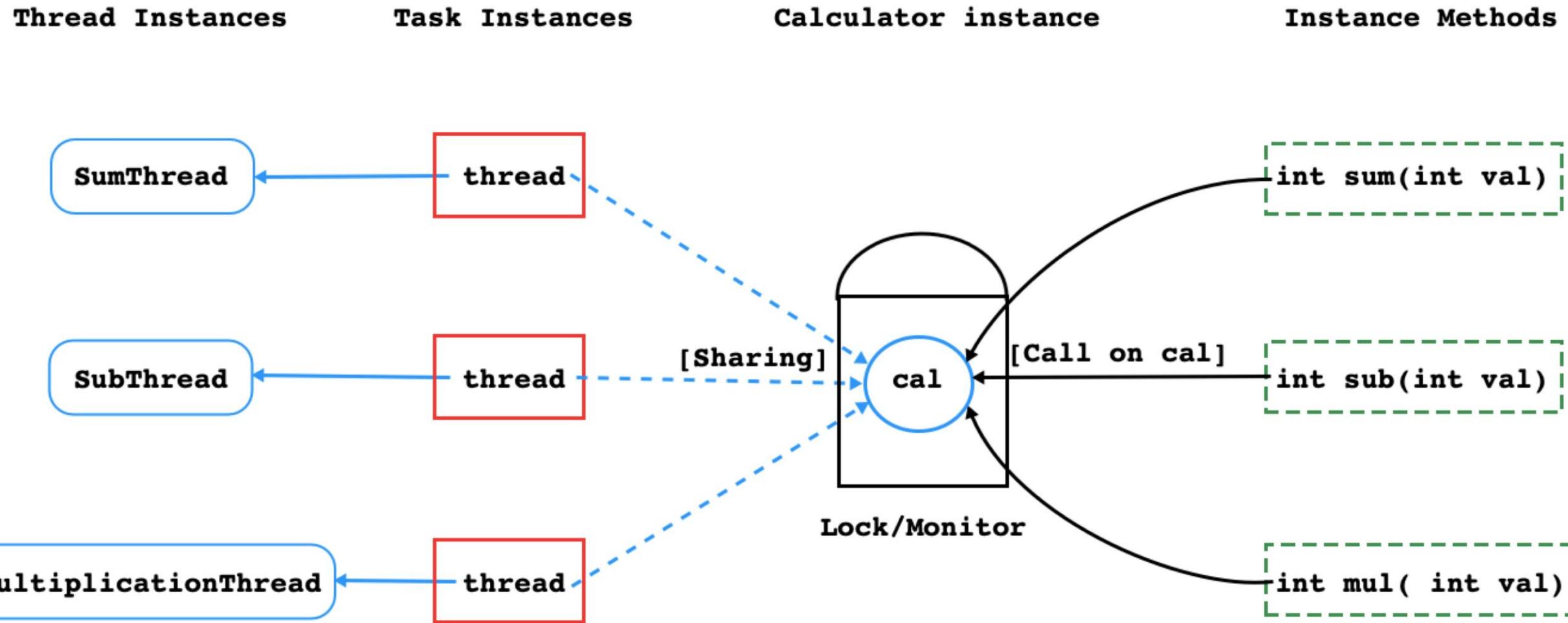
Lock Objects

- There are two mechanisms for protecting a code block from concurrent access.
 1. A synchronized keyword.
 2. The ReentrantLock class.
- The synchronized keyword automatically provides a lock as well as an associated “condition,” which makes it powerful and convenient for most cases that require explicit locking.
- The basic outline for protecting a code block with a ReentrantLock is:

```
myLock.lock(); // a ReentrantLock object try
{
    critical section
} finally {
    myLock.unlock(); //lock is unlocked even if an exception is thrown
}
```



Monitor Concept

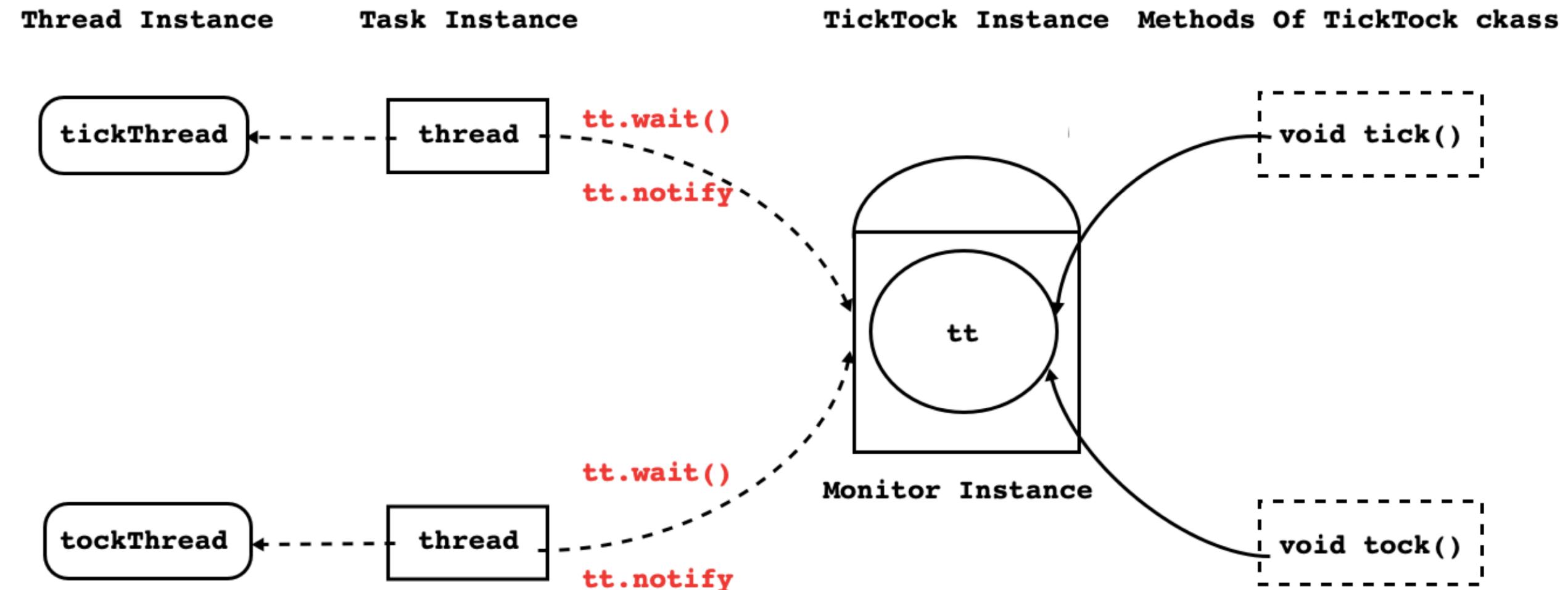


Thread Synchronization

- Using same object's monitor, if multiple threads try to communicate with each other then it is called inter thread communication. And inter thread communication represents synchronization.
- To achieve synchronization, we can use methods declared in `java.lang.Object` class:
 1. `public final void wait() throws InterruptedException`
 2. `public final void wait(long timeout) throws InterruptedException`
 3. `public final void wait(long timeout, int nanos) throws InterruptedException`
 4. `public final void notify()`
 5. `public final void notifyAll()`
- Resource locking / inter thread communication is based on monitor object associated with shared resource. Type of every shared resource is directly or indirectly extended from `java.lang.Object` class. Hence `wait/notify/notifyAll` methods are declared in `java.lang.Object` class.

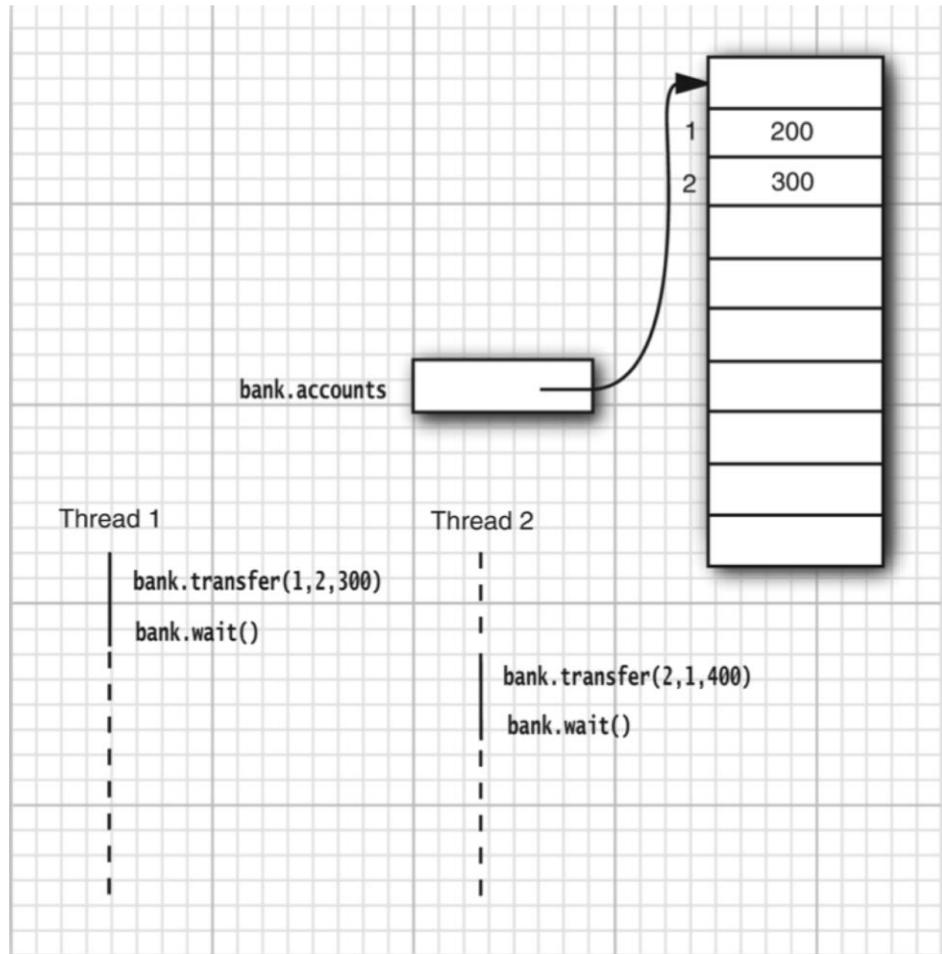


Thread Synchronization

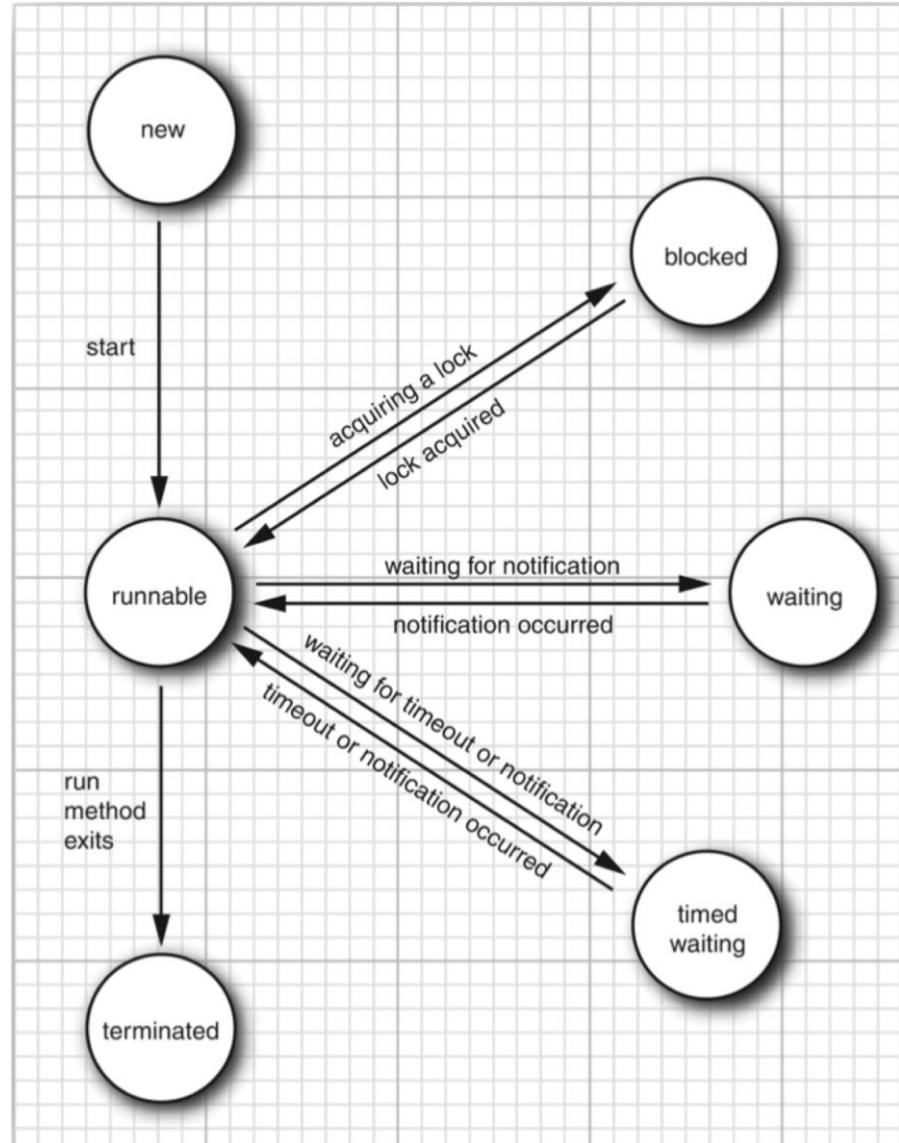


Deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.



Thread Life Cycle





Thank You.

[akshita.chanchlani@sunbeaminfo.com]

