



Java SE 8 Programmer I (1Z0-808)

Study Material



Table of Contents

1) The Features of Java	3
2) Data Types	8
3) Arrays	24
4) Types of Variables	48
5) Main Method and Command Line Arguments	76
6) Operators And Assignments	88
7) Flow Control	108
8) Declarations And Access Modifiers	126
9) OOPs And Constructors	201
10) Exception Handling	268
11) String And StringBuilder	309
12) Wrapper Classes	344
13) Collections Framework And ArrayList	353
14) Lambda Expression And Predicates	368
15) Date And Time API (Joda - Time API)	390
16) Garbage Collections	397



The Features of Java (Java BUZZWORDS)

Java language is the most powerful and high level programming language. The power of Java is described by the following 12 buzzwords.

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

1) Simple:

- Java is very simple and easy to learn (Nursery Level) programming language.
- We can write Java programs very easily.
- To learn Java no prior knowledge is required.
- Most of the complex or confusing features of other languages C,C++ like Pointers etc .. are removed in Java.

2) Platform Independent:

If we write Java program once, we can run on any platform. i.e Java follows Write Once Run and Anywhere(WORA) principle.

3) Architecture-Neutral:

Java program never communicates with the platform directly. Changes and upgrades in operating systems, processors and system resources will not force any changes in Java Programs.

4) Portable:

We can carry the java byte code to any platform without making any changes.
(Mobile Number Portability in India)



5) Secure:

- Java programs never communicate directly with the machine. First converted into byte code and then converted into machine code by the JVM
- If the byte code contains any problem, then JVM won't allow that code to run and will raise VerifyError. Internally inside JVM, ByteCode verifier is responsible to verify the byte code.
- Hence Java programs won't cause any problem to the System.

6) Object Oriented Programming Language:

- Java is Object Oriented Programming Language like C++. Most of the times in java, we have to handle everything in terms of Object.
- Java provides support for the following OOP features
 - Encapsulation
 - Inheritance
 - Polymorphism
 - etc

7) Multithreaded:

- In the case of multithreading, multiple threads can run simultaneously and can perform specified tasks simultaneously, so that performance of the application will be improved.
- Java provides inbuilt support for multi threading by providing a rich API.

8) Robust:

- Java is strongly typed language. Compiler will check each and every declaration and assignments at compile time only for type compatibility. If any problem wrt types, then at compile time only we can identify the problem.
- Java provides Garbage Collector for automatic memory management. Hence there is no chance of memory related problems.
- Java provides inbuilt Exception handling, which prevents abnormal termination of the program at runtime.
- Java is platform independent and it can run on any platform.
- Because of all these facilities, the chance of failing the program at runtime is very very less and Hence Java is Robust.

9) Distributed:

- If the application is distributed across multiple machines (JVMs), such type of application is called Distributed Application. Java provides inbuilt support for Distributed programming with RMI and EJB.



10) Compiled and Interpreted:

Java is both Compiled and Interpreted Programming language. First Java compiler compiles java code and generates machine independent Byte Code. At runtime JVM interprets this byte code into machine code and executes that machine code.

11) High Performance:

Java is relatively faster than traditional interpreted languages, since byte code is "close" to native code. But Java is still somewhat slower than C or C++.

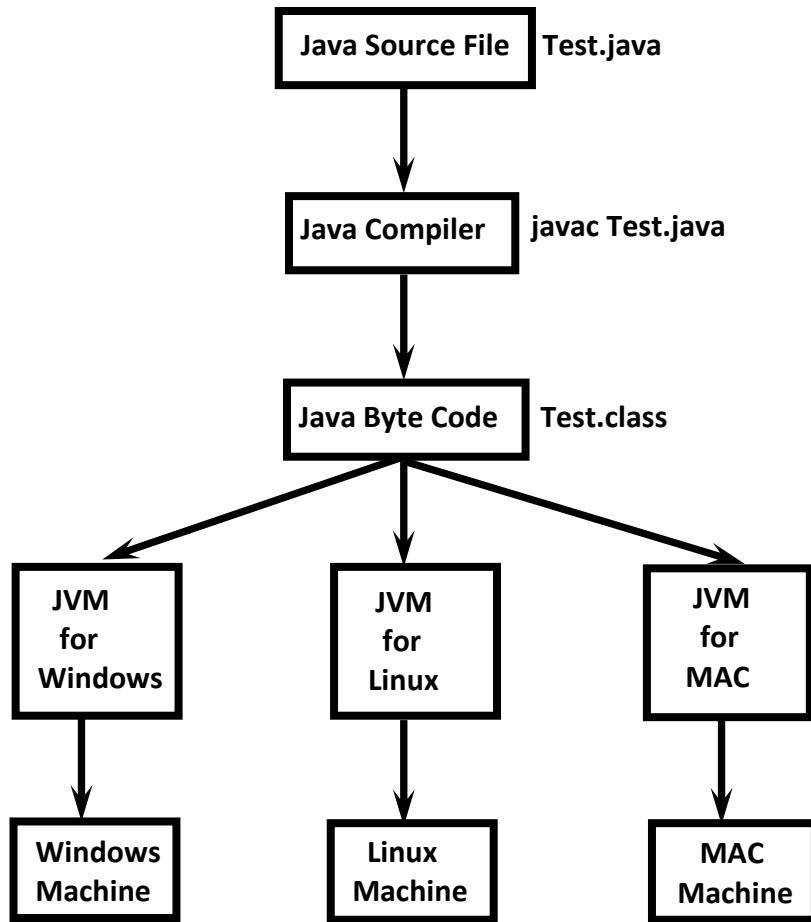
12) Dynamic:

In the case of Java programs, all .class files won't be loaded at the beginning. At runtime if JVM required any class then only the corresponding .class file will be loaded(Dynamic Loading).The main advantage is program will always get latest version of .class file and memory utilization will be improved.

Explain Platform Independent Nature of Java:

Java follows Write Once and Run anywhere policy (WORA). i.e Once we write a java program, we can run on any platform without making any changes.

First Java Source File will be compiled into ByteCode. Bytecode is an intermediate and machine independent code. JVM will interpret byte code into the corresponding machine dependent code and executes that machine code.



Note:

Java is platform independent where as JVM is platform dependent.

JDK vs JRE vs JVM:

JDK (Java Development Kit) provides environment to develop and run java applications.

JRE (Java Runtime Environment) provides environment to run java applications.

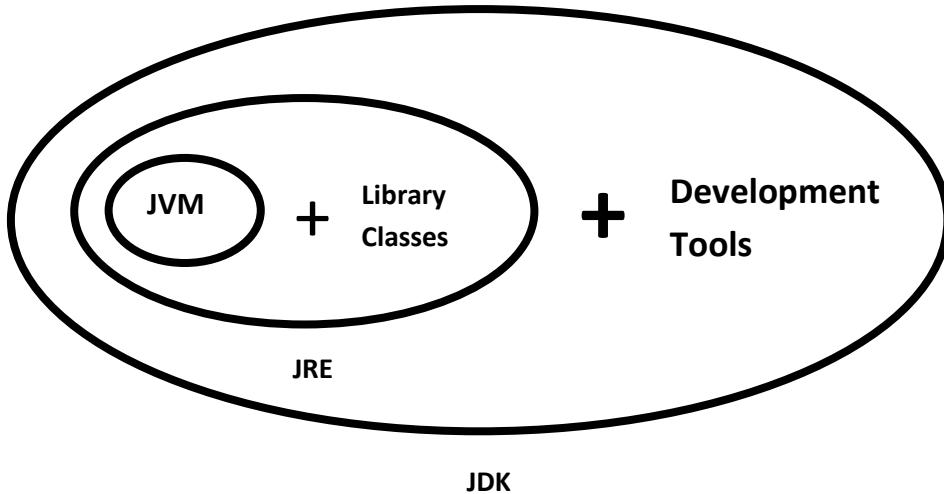
JVM (Java Virtual Machine) is an interpreter which is responsible to run java applications line by line.

Note:

JVM is the part of JRE where as JRE is the part of JDK

JDK = JRE + Development Tools

JRE = JVM + Libraries



Note:

On the Developer's Machine we have to install JDK, whereas on the client machine we have to install JRE.

Q. Which of the following is true?

- A) Java is platform dependent but JVM is platform independent
- B) Java is platform independent but JVM is platform dependent
- C) Java Byte code is platform dependent but JVM is platform independent
- D) Java Byte code is platform independent but JVM is platform dependent

Answer: B and D

Q. Which Statement is true about Java Byte code?

- A) It can run on any platform
- B) It can run on any platform only if it was compiled for that platform
- C) It can run on any platform that has the Java Runtime Environment (JRE)
- D) It can run on any platform that has a Java Compiler
- E) It can run on any platform only if that platform has both JRE and Java Compiler

Answer: C

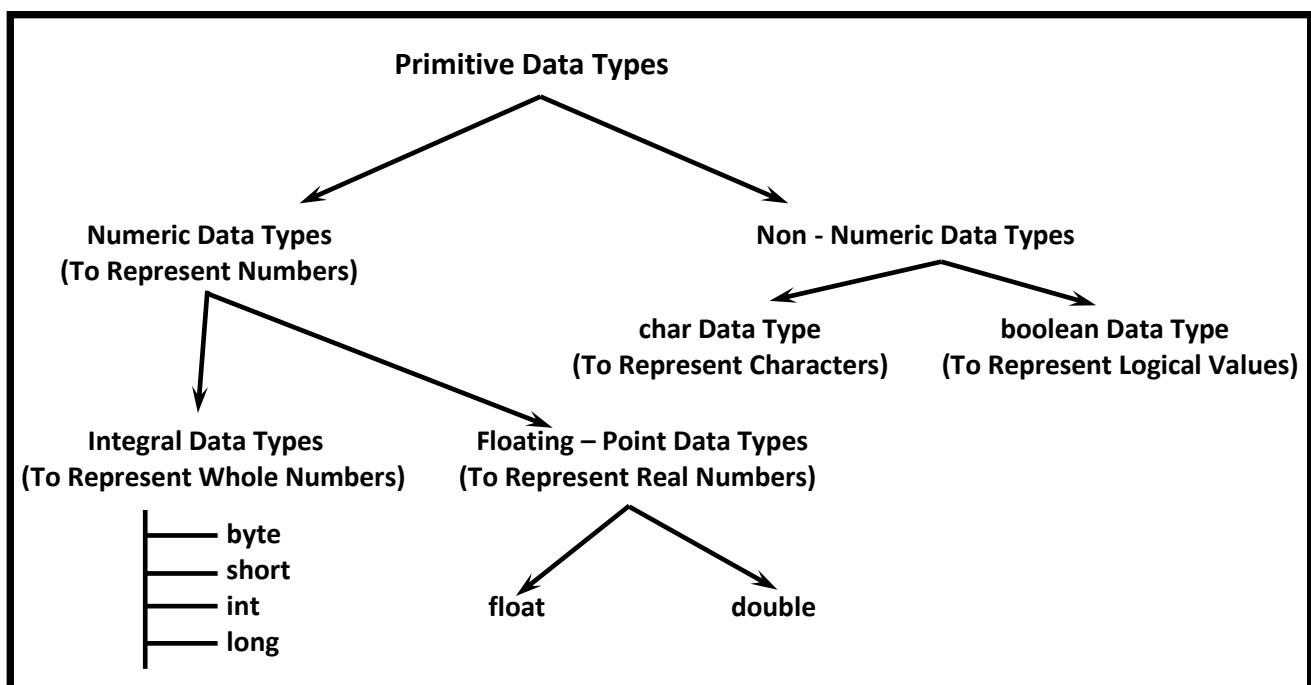


Data Types

Every variable has a type, every expression has a type and all types are strictly defined over every assignment should be checked by the compiler by the type compatibility hence java language is considered as strongly typed programming language.

Java is pure object oriented programming or not?

Java is not considered as pure object oriented programming language because several oops features (like multiple inheritance, operator overloading) are not supported by java. Moreover we are depending on primitive data types which are non objects.



Except Boolean and char all remaining data types are considered as signed data types because we can represent both "+ve" and "-ve" numbers.



Integral Data Types:

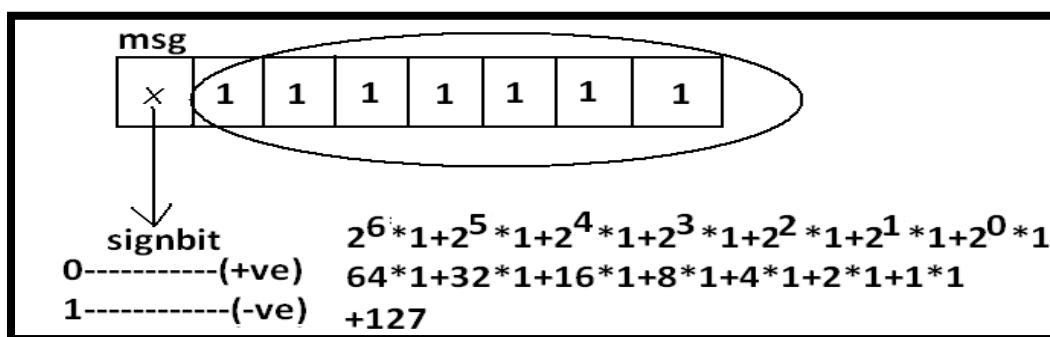
byte Data Type:

Size: 1byte (8bits)

Maxvalue: +127

Minvalue:-128

Range:-128 to 127[-2⁷ to 2⁷-1]



The most significant bit acts as sign bit. "0" means "+ve" number and "1" means "-ve" number. "+ve" numbers will be represented directly in the memory whereas "-ve" numbers will be represented in 2's complement form.

Example:

```
byte b=10;
byte b2=130;//C.E:possible loss of precision
    found : int
    required : byte
byte b=10.5;//C.E:possible loss of precision
byte b=true;//C.E:incompatible types
byte b="ashok";//C.E:incompatible types
    found : java.lang.String
    required : byte
```

Note:

byte data type is best suitable if we are handling data in terms of streams either from the file or from the network.



short Data Type:

The most rarely used data type in java is short.

Size: 2 bytes

Range: s-32768 to 32767 (-215 to 215-1)

Example:

```
short s=130;  
short s=32768;//C.E:possible loss of precision  
short s=true;//C.E:incompatible types
```

Note:

short data type is best suitable for 16 bit processors like 8086 but these processors are completely outdated and hence the corresponding short data type is also out data type.

int Data Type:

This is most commonly used data type in java.

Size: 4 bytes

Range:-2147483648 to 2147483647 (-231 to 231-1)

Example:

```
int i=130;  
int i=10.5;//C.E:possible loss of precision  
int i=true;//C.E:incompatible types
```

long Data Type:

Whenever int is not enough to hold big values then we should go for long data type.

Eg 1:

To hold the distance travelled by light in 1000 days , int may not enough, compulsory we should go for long data type.

```
long l = 186000*60*24*1000 miles
```

Eg 2:

To hold the number of characters present in a big file, int may not enough, compulsory we should go for long data type. Hence the return type of length() method is long.

```
long l=f.length();//f is a file
```

Size: 8 bytes

Range: 2^{63} to $2^{63}-1$



Note: All the above data types (byte, short, int and long) can be used to represent whole numbers. If we want to represent real numbers then we should go for floating point data types.

Floating Point Data types:

Float	double
If we want to 5 to 6 decimal places of accuracy then we should go for float.	If we want to 14 to 15 decimal places of accuracy then we should go for double.
Size:4 bytes.	Size:8 bytes.
Range:-3.4e38 to 3.4e38.	-1.7e308 to 1.7e308.
float follows single precision.	double follows double precision.

boolean data type:

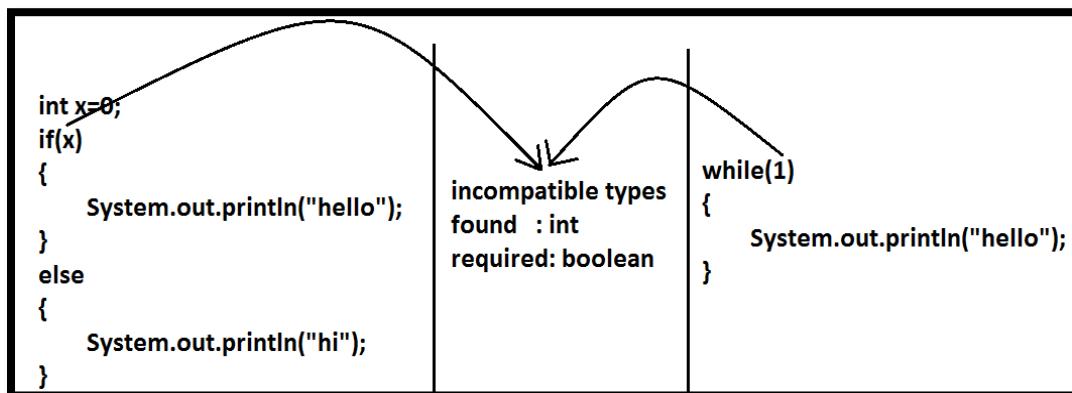
Size: Not applicable (virtual machine dependent)

Range: Not applicable but allowed values are true or false.

Q. Which of the following boolean declarations are valid?

```
boolean b=true;
boolean b=True;//C.E:cannot find symbol
boolean b="True";//C.E:incompatible types
boolean b=0;//C.E:incompatible types
```

Example 2:





char data type:

Old languages like C & C++ are ASCII code based and the number of ASCII characters are < 256. To represent these 256 characters, 8 – bits are enough and hence char size in old languages 1 byte.

But, in java we are allowed to use worldwide any alphabet character and java is Unicode based. The number of unicode characters are > 256 and <= 65536. To represent all these characters one byte is not enough compulsory we should go for 2 bytes.

Size: 2 bytes

Range: 0 to 65535

Example:

```
char ch1=97;  
char ch2=65536;//C.E:possible loss of precision
```

Summary of Java Primitive Data Type

data type	Size	Range	Corresponding Wrapper class	Default value
byte	1 byte	-2^7 to $2^7 - 1$ (-128 to 127)	Byte	0
short	2 bytes	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	Short	0
int	4 bytes	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	Integer	0
long	8 bytes	-2^{63} to $2^{63} - 1$	Long	0
float	4 bytes	-3.4e38 to 3.4e38	Float	0.0
double	8 bytes	-1.7e308 to 1.7e308	Double	0.0
boolean	Not applicable	Not applicable (but allowed values true false)	Boolean	false
char	2 bytes	0 to 65535	Character	0 (represents blank space)

Note:

The default value for the object references is "null".



Literals

Any constant value which can be assigned to the variable is called literal.

Example:

```
int x=10
```

constant value | literal
name of variable | identifier
datatype | keyword

Integral Literals:

For the integral data types (byte, short, int and long) we can specify literal value in the following ways.

1) Decimal literals(base -10):

Allowed digits are 0 to 9.

Example: int x=10;

2) Octal literals(base-8):

Allowed digits are 0 to 7. Literal value should be prefixed with zero.

Example: int x=010;

3) Hexa Decimal literals(base-16):

The allowed digits are 0 to 9, A to Z.

For the extra digits we can use both upper case and lower case characters. This is one of very few areas where java is not case sensitive.

Literal value should be prefixed with 0x(or)oX.

Example: int x=0x10;

These are the only possible ways to specify integral literal.



Which of the following are valid declarations?

```
int x=0777; //(valid)
int x=0786; //C.E:integer number too large: 0786(invalid)
int x=0xFACE; (valid)
int x=0xbeef; (valid)
int x=0xBeer; //C.E: ';' expected(invalid) //:int x=0xBeer; ^// ^
int x=0xabb2cd;(valid)
```

Example:

```
int x=10;
int y=010;
int z=0x10;
System.out.println(x+"---"+y+"---"+z); //10---8---16
```

By default every integral literal is of int type but we can specify explicitly as long type by suffixing with small "l" (or) capital "L".

Example:

```
int x=10;(valid)
long l=10L;(valid)
long l=10;(valid)
int x=10l;//C.E:possible loss of precision(invalid)
    found : long
    required : int
```

There is no direct way to specify byte and short literals explicitly. But whenever we are assigning integral literal to the byte variables and its value within the range of byte compiler automatically treats as byte literal. Similarly short literal also.

Example:

```
byte b=127;(valid)
byte b=130;//C.E:possible loss of precision(invalid)
short s=32767;(valid)
short s=32768;//C.E:possible loss of precision(invalid)
```



Floating Point Literals:

Floating point literal is by default double type but we can specify explicitly as float type by suffixing with f or F.

Example:

```
float f=123.456;//C.E:possible loss of precision(invalid)
```

```
float f=123.456f;(valid)
```

```
double d=123.456;(valid)
```

We can specify explicitly floating point literal as double type by suffixing with d or D.

Example:

```
double d=123.456D;
```

We can specify floating point literal only in decimal form and we can't specify in octal and hexadecimal forms.

Example:

```
double d=123.456;(valid)
```

```
double d=0123.456;(valid) //it is treated as decimal value but not octal
```

```
double d=0x123.456;//C.E:malformed floating point literal(invalid)
```

Which of the following floating point declarations are valid?

```
float f=123.456; //C.E:possible loss of precision(invalid)
```

```
float f=123.456D; //C.E:possible loss of precision(invalid)
```

```
double d=0x123.456; //C.E:malformed floating point literal(invalid)
```

```
double d=0xFace; (valid)
```

```
double d=0xBEEF; (valid)
```

We can assign integral literal directly to the floating point data types and that integral literal can be specified in decimal, octal and Hexa decimal form also.

Example:

```
double d=0xBEEF;
```

```
System.out.println(d); //48879.0
```

```
float f = 100f;
```

```
System.out.println(f); //100.0
```

But we can't assign floating point literal directly to the integral types.

Example:

```
int x=10.0;//C.E:possible loss of precision
```

We can specify floating point literal even in exponential form also (significant notation).



Example:

```
double d=10e2;//==>10*102(valid)
```

```
System.out.println(d);//1000.0
```

```
float f=10e2;//C.E:possible loss of precision(invalid)
```

```
float f=10e2F;(valid)
```

Boolean literals:

The only allowed values for the boolean type are true (or) false where case is important.
i.e., lower case

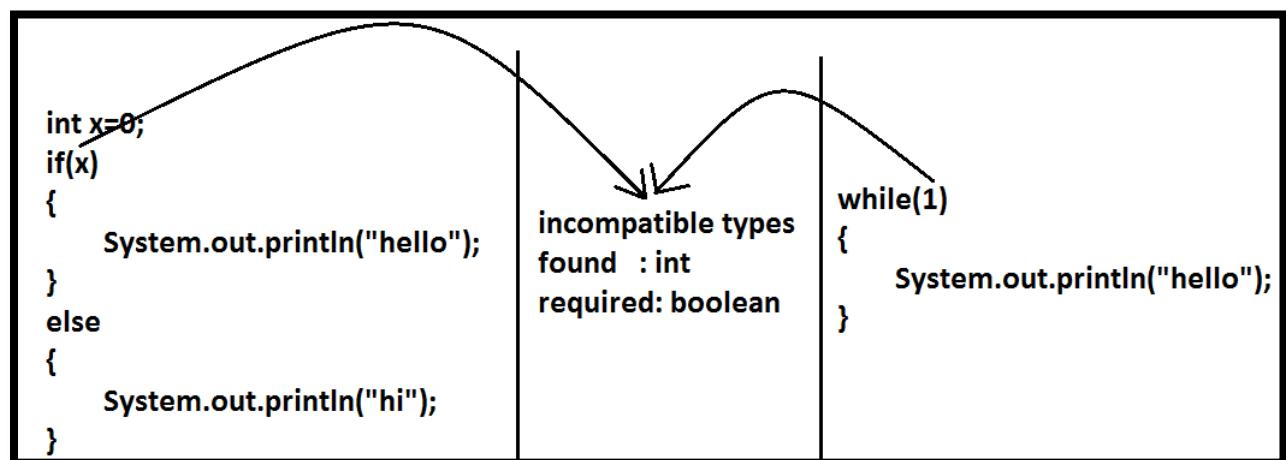
Example:

```
boolean b=true;(valid)
```

```
boolean b=0;//C.E:incompatible types(invalid)
```

```
boolean b=True;//C.E:cannot find symbol(invalid)
```

```
boolean b="true";//C.E:incompatible types(invalid)
```



Char literals:

- 1) A char literal can be represented as single character within single quotes.

Example:

```
char ch='a';(valid)
```

```
char ch=a;//C.E:cannot find symbol(invalid)
```

```
char ch="a";//C.E:incompatible types(invalid)
```

```
char ch='ab';//C.E:unclosed character literal(invalid)
```



2) We can specify a char literal as integral literal which represents Unicode of that character. We can specify that integral literal either in decimal or octal or hexadecimal form but allowed values range is 0 to 65535.

Example:

```
char ch=97; (valid)
char ch=0xFace; (valid)
System.out.println(ch); //?
char ch=65536; //C.E: possible loss of precision(invalid)
3) We can represent a char literal by Unicode representation which is nothing but '\uxxxx' (4 digit hexa-decimal number) .
```

Example:

```
char ch='\\ubeef';
char ch1='\\u0061';
System.out.println(ch1); //a
char ch2='\\u0062'; //C.E:cannot find symbol
char ch3='\\iface'; //C.E:illegal escape character
Every escape character in java acts as a char literal.
```

Example:

```
1) char ch='\\n'; //(valid)
2) char ch='\\l'; //C.E:illegal escape character(invalid)
```

Escape Character	Description
\\n	New line
\\t	Horizontal tab
\\r	Carriage return
\\f	Form feed
\\b	Back space character
\\'	Single quote
\\"	Double quote
\\\\	Back slash



Which of the following char declarations are valid?

char ch=a; //C.E:cannot find symbol(invalid)
char ch='ab'; //C.E:unclosed character literal(invalid)
char ch=65536; //C.E:possible loss of precision(invalid)
char ch=\uface; //C.E:illegal character: \64206(invalid)
char ch='/'n'; //C.E:unclosed character literal(invalid)
none of the above. (valid)

String literals:

Any sequence of characters with in double quotes is treated as String literal.

Example:

String s="Durga"; (valid)

1.7 Version enhancements with respect to Literals:

The following 2 are enhancements

- Binary Literals
- Usage of '_' in Numeric Literals

Binary Literals:

For the integral data types until 1.6v we can specified literal value in the following ways

Decimal

Octal

Hexa decimal

But from 1.7v onwards we can specify literal value in binary form also.

The allowed digits are 0 to 1.

Literal value should be prefixed with 0b or OB .

int x = 0b111;

System.out.println(x); // 7

Usage of _ symbol in numeric literals:

From 1.7v onwards we can use underscore(_) symbol in numeric literals.

double d = 123456.789; //valid

double d = 1_23_456.7_8_9; //valid

double d = 123_456.7_8_9; //valid



The main advantage of this approach is readability of the code will be improved At the time of compilation ' _ ' symbols will be removed automatically , hence after compilation the above lines will become double d = 123456.789

We can use more than one underscore symbol also between the digits.

Ex : double d = 1_23_456.789;

We should use underscore symbol only between the digits

double d=_1_23_456.7_8_9; //invalid

double d=1_23_456.7_8_9_; //invalid

double d=1_23_456_.7_8_9; //invalid

PRIMITIVE TYPE CASTING

There are 2 types of type-casting

Implicit Type Casting

Explicit Type Casting

IMPLICIT TYPE CASTING :

```
int x='a';
System.out.println(x); //97
```

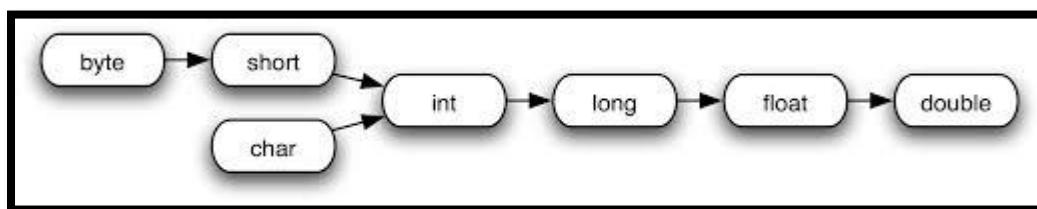
The compiler is responsible to perform this type casting.

Whenever we are assigning lower datatype value to higher datatype variable then implicit type cast will be performed .

It is also known as Widening or Upcasting.

There is no lose of information in this type casting.

The following are various possible implicit type casting.



Example 1:

```
int x='a';
System.out.println(x); //97
```

Note: Compiler converts char to int type automatically by implicit type casting.



Example 2:

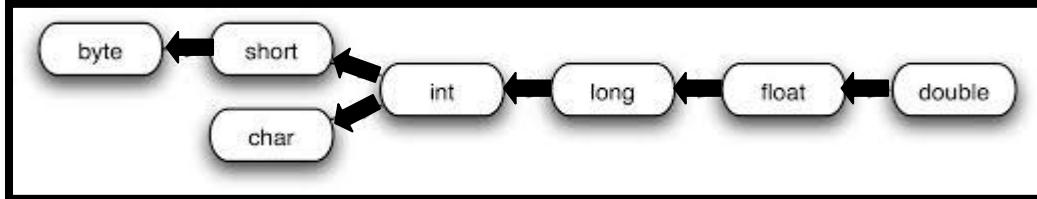
```
double d=10;  
System.out.println(d);//10.0
```

Note: Compiler converts int to double type automatically by implicit type casting.

EXPLICIT TYPE CASTING:

- 1) Programmer is responsible for this type casting.
- 2) Whenever we are assigning bigger data type value to the smaller data type variable then explicit type casting is required.
- 3) Also known as Narrowing or down casting.
- 4) There may be a chance of lose of information in this type casting.

The following are various possible conversions where explicit type casting is required.



Left → Right → Implicit Type Casting
Right → Left → Explicit Type Casting

```
int x=130;  
byte b=x;
```

```
E:\scjp>javac OperatorsDemo.java  
OperatorsDemo.java:6: possible loss of precision  
  found  : int  
  required: byte  
        byte b=x;
```

Example:

```
int x=130;  
byte b=(byte)x;  
System.out.println(b); // -126
```



$x(130) \equiv 0000000010000010$
byte $b = (byte)x \equiv 10000010$

$$\begin{array}{r} 1111101 \\ 11 \\ \hline 1111110 \end{array}$$

signbit

0 — +ve
1 — -ve

$$0*2^0 + 1*2^1 + 1*2^2 + 1*2^3 + 1*2^4 + 1*2^5 + 1*2^6$$

$$0*1 + 1*2 + 1*4 + 1*8 + 1*16 + 1*32 + 1*64$$

$$0 + 2 + 4 + 8 + 16 + 32 + 64$$

$$-126$$

2	130	
2	65	0
2	32	1
2	16	0
2	8	0
2	4	0
2	2	0
		1

Example 2:

```
int x=130;
byte b=x;
System.out.println(b); //CE : possible loss of precision
```

Whenever we are assigning higher datatype value to lower datatype value variable by explicit type casting ,the most significant bits will be lost i.e., we have considered least significant bits .

Example 3 :

```
int x=150;
short s=(short)x;
byte b=(byte)x;
System.out.println(s); //150
System.out.println(b); // -106
```

Whenever we are assigning floating point value to the integral types by explicit type casting, the digits of after decimal point will be lost .



Example 4:

```
double d=130.456 ;  
  
int x=(int)d ;  
System.out.println(x); //130  
  
byte b=(byte)d ;  
System.out.println(b); //-206
```

float x=150.1234f; int i=(int)x; System.out.println(i); //150	double d=130.456; int i=(int)d; System.out.println(i); //130
--	---

Q1. Which of the following conversions will be performed automatically in Java?

- A) int to byte
- B) byte to int
- C) float to double
- D) double to float
- E) None of the above

Answer: B, C

Q2. In which of the following cases explicit Type casting is required ?

- A) int to byte
- B) byte to int
- C) float to double
- D) double to float
- E) None of the above

Answer: A, D

Q3. Consider the code

```
int i =100;  
float f = 100.100f;  
double d = 123;
```



Which of the following assignments won't compile?

- A) i=f;
- B) f=i;
- C) d=f;
- D) f=d;
- E) d=i;
- F) i=d;

Answer: A,D,F

Q4. In which of the following cases we will get Compile time error?

- A) float f =100F;
- B) float f =(float)1_11.00;
- C) float f =100;
- D) double d = 203.22;
 float f = d;
- E) int i =100;
 float f=(float)i;

Answer: D

Q5. Consider the code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         int a=10;
6)         float b=10.25f;
7)         double c=100;
8)         a = b;
9)         b = a;
10)        c = b;
11)        c = a;
12)    }
13) }
```

Which change will enable the code fragment to compile successfully?

- A) Replace a = b; with a=(int)b;
- B) Replace b = a; with b=(float)a;
- C) Replace c = b; with c=(double)b;
- D) Replace c = a; with c=(double)a;

Answer: A



Arrays

- 1) Introduction
- 2) Array declaration
- 3) Array construction
- 4) Array initialization
- 5) Array declaration, construction, initialization in a single line
- 6) length variable Vs length() method
- 7) Anonymous arrays
- 8) Array element assignments
- 9) Array variable assignments

1) Introduction

An array is an indexed collection of fixed number of homogeneous data elements. The main advantage of arrays is we can represent multiple values with the same name so that readability of the code will be improved.

But the main disadvantage of arrays is:

Fixed in size, that is once we created an array there is no chance of increasing or decreasing the size based on our requirement that is to use arrays concept compulsory we should know the size in advance which may not possible always.

We can resolve this problem by using collections.

2) Array declarations:

Single dimensional array declaration:

Example:

```
int[] a;//recommended to use because name is clearly separated from the type
```

```
int []a;
```

```
int a[];
```

At the time of declaration we can't specify the size otherwise we will get compile time error.

Example:

```
int[] a;//valid
```

```
int[5] a;//invalid
```



Two dimensional array declaration:

Example:

```
int[][] a;  
int [][]a;  
int a[][]; All are valid.(6 ways)  
int[] []a;  
int[] a[];  
int []a[];
```

Three dimensional array declaration:

Example:

```
int[][][] a;  
int [][][]a;  
int a[][][];  
int[] [][]a;  
int[] a[][]; All are valid.(10 ways)  
int[] []a[];  
int[][] []a;  
int[][] a[];  
int []a[][];  
int [][]a[];
```

Which of the following declarations are valid?

- 1) int[] a1,b1; //a-1,b-1 (valid)
- 2) int[] a2[],b2; //a-2,b-1 (valid)
- 3) int[] []a3,b3; //a-2,b-2 (valid)
- 4) int[] a,[]b; //C.E: expected (invalid)

Note:

If we want to specify the dimension before the variable, then it is allowed only for the first variable. By mistake if we are trying to declare for any other variable in the same declaration then we will get compile time error.

Example:

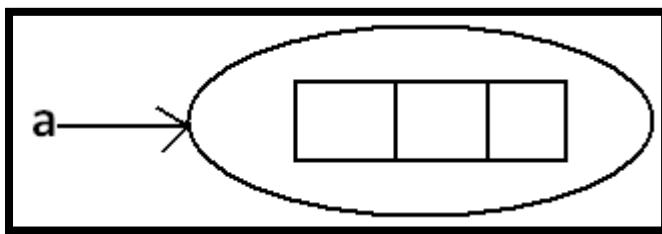
int[] []a,[]b;
invalid
valid



3) Array construction:

Every array in java is an object hence we can create by using new operator.

Example: `int[] a=new int[3];`



For every array type corresponding classes are available but these classes are part of java language and not available to the programmer level.

Array Type	corresponding class name
<code>int[]</code>	<code>[I</code>
<code>int[][]</code>	<code>[[I</code>
<code>double[]</code>	<code>[D</code>

Rule 1:

At the time of array creation compulsory we should specify the size otherwise we will get compile time error.

Example:

```
int[] a=new int[3];  
int[] a=new int[];//C.E:array dimension missing
```

Rule 2:

It is legal to have an array with size zero in java.

Example:

```
int[] a=new int[0];  
System.out.println(a.length);//0
```

Rule 3:

If we are taking array size with -ve int value then we will get runtime exception saying `NegativeArraySizeException`.

Example:

```
int[] a=new int[-3];//R.E:NegativeArraySizeException
```



Rule 4:

The allowed data types to specify array size are byte, short, char, int.
By mistake if we are using any other type we will get compile time error.

Example:

```
int[] a=new int['a'];//(valid)
byte b=10;
int[] a=new int[b];//(valid)
short s=20;
int[] a=new int[s];//(valid)
int[] a=new int[10l];//C.E:possible loss of precision//(invalid)
int[] a=new int[10.5];//C.E:possible loss of precision//(invalid)
```

Rule 5:

The maximum allowed array size in java is maximum value of int size [2147483647].

Example:

```
int[] a1=new int[2147483647];(valid)
int[] a2=new int[2147483648];
//C.E:integer number too large: 2147483648(invalid)
```

In the first case we may get RE : OutOfMemoryError.

Q. Consider the code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         /* Line-1: insert code here */
6)         x[0]=10;
7)         x[1]=20;
8)         System.out.println(x[0]+":"+x[1]);
9)     }
10) }
```

Which code fragment required to insert at Line-1 to produce output 10:20

- A) int[] x = new int[2];
- B) int[] x;
x = int[2];
- C) int x = new int[2];
- D) int x[2];

Answer: A

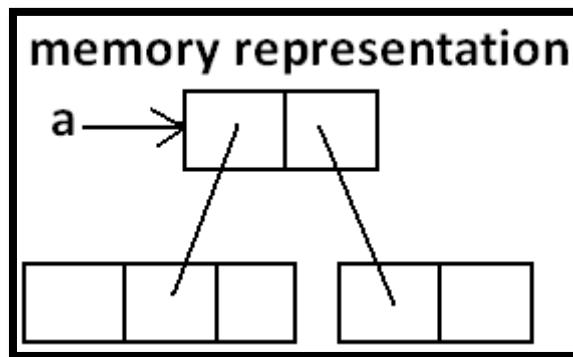


Multi dimensional array creation:

In java multidimensional arrays are implemented as array of arrays approach but not matrix form. The main advantage of this approach is to improve memory utilization.

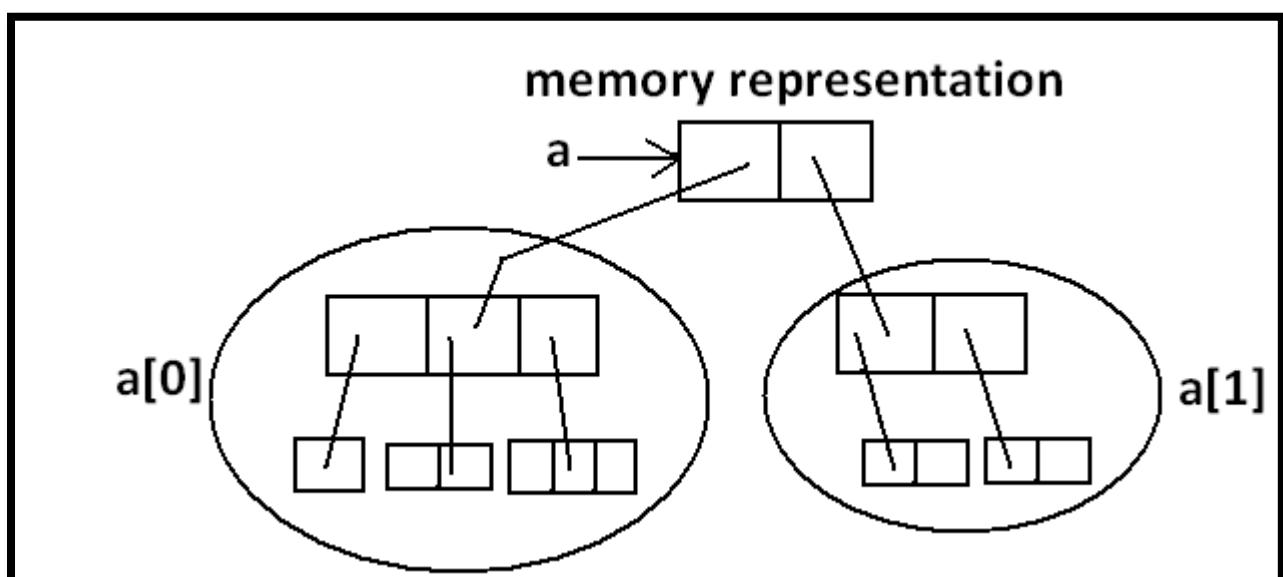
Example 1:

```
int[][] a=new int[2][];
a[0]=new int[3];
a[1]=new int[2];
```



Example 2:

```
int[][][] a=new int[2][][];
a[0]=new int[3][];
a[0][0]=new int[1];
a[0][1]=new int[2];
a[0][2]=new int[3];
a[1]=new int[2][2];
```





Which of the following declarations are valid?

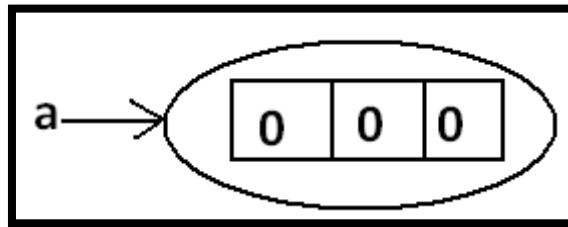
- 1) int[] a=new int[];//C.E: array dimension missing(invalid)
- 2) int[][] a=new int[3][4];//(valid)
- 3) int[][] a=new int[3][];//(valid)
- 4) int[][] a=new int[][],//C.E:']' expected(invalid)
- 5) int[][][] a=new int[3][4][5];//(valid)
- 6) int[][][] a=new int[3][4]);//(valid)
- 7) int[][][] a=new int[3][][];//C.E:']' expected(invalid)

4) Array Initialization:

Whenever we are creating an array every element is initialized with default value automatically.

Example 1:

```
int[] a=new int[3];
System.out.println(a);//[I@3e25a5
System.out.println(a[0]);//0
```

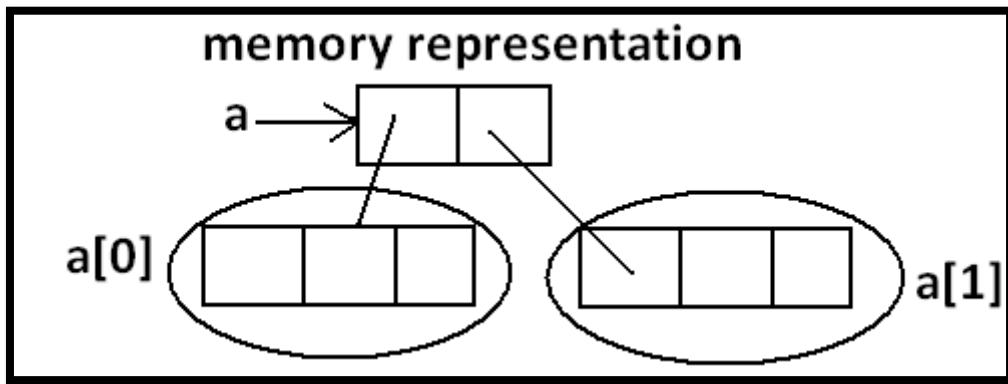


Note: Whenever we are trying to print any object reference internally `toString()` method will be executed which is implemented by default to return the following.
classname@hexadecimalstringrepresentationofhashcode.

Example 2:

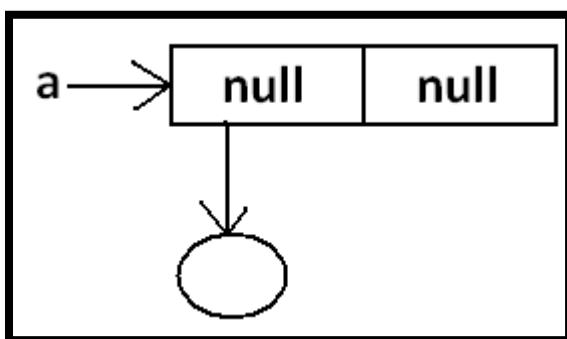
int[][] a=new int[2][3]; base size

```
System.out.println(a);//[[I@3e25a5
System.out.println(a[0]);//[I@19821f
System.out.println(a[0][0]);//0
```



Example 3:

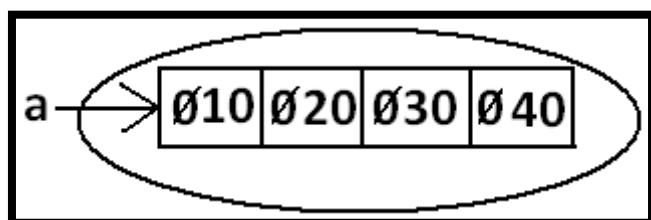
```
int[][] a=new int[2][];
System.out.println(a); //[[I@3e25a5
System.out.println(a[0]); //null
System.out.println(a[0][0]); //R.E:NullPointerException
```



Once we created an array all its elements by default initialized with default values. If we are not satisfied with those default values then we can replace with our customized values.

Example:

```
int[] a=new int[4];
a[0]=10;
a[1]=20;
a[2]=30;
a[3]=40;
a[4]=50; //R.E:ArrayIndexOutOfBoundsException: 4
a[-4]=60; //R.E:ArrayIndexOutOfBoundsException: -4
```





Note: if we are trying to access array element with out of range index we will get Runtime Exception saying `ArrayIndexOutOfBoundsException`.

Q. Consider the code

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String[] s = new String[2];
6)         int i=0;
7)         for(String s1 : s)
8)         {
9)             s[i].concat("Element "+i);
10)            i++;
11)        }
12)        for(i=0;i<s.length;i++)
13)        {
14)            System.out.println(s[i]);
15)        }
16)    }
17) }
```

What is the result?

- A) Element 0
Element 1
- B) null Element 0
null Element 1
- C) null
null
- D) `NullPointerException`

Answer: D

Note: On null,if we are trying to apply any operation then we will get `NullPointerException`

Q. Consider the code

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         int[][] n = new int[1][3];
6)         for(int i=0; i<n.length; i++)
7)     }
```



```
8)     for (int j=0;j>n[i].length ;j++ )  
9)     {  
10)         num[i][j]=10;  
11)     }  
12)     }  
13) }  
14} }
```

Which option represents the state of the array after successful completion of outer for loop?

A) n[0][0]=10;
n[0][1]=10;
n[0][2]=10;

B) n[0][0]=10;
n[1][0]=10;
n[2][0]=10;

C) n[0][0]=10;
n[0][1]=0;
n[0][2]=0;

D) n[0][0]=10;
n[0][1]=10;
n[0][2]=10;
n[1][0]=0;
n[1][1]=0;
n[1][2]=0;
n[1][3]=0;

Answer: A

Q. Consider the code

```
1) class Student  
2) {  
3)     String name;  
4)     public Student(String name)  
5)     {  
6)         this.name=name;  
7)     }  
8) }  
9) public class Test  
10) {  
11)     public static void main(String[] args)  
12)     {  
13)         Student[] students = new Student[3];  
14)         students[1]= new Student("Durga");
```



```
15)  students[2]= new Student("Ravi");
16)  for(Student s : students)
17)  {
18)      System.out.println(s.name);
19)  }
20) }
21) }
```

What is the result?

- A) Durga
Ravi
- B) null
Durga
Ravi
- C) Compilation Fails
- D) ArrayIndexOutOfBoundsException
- E) NullPointerException

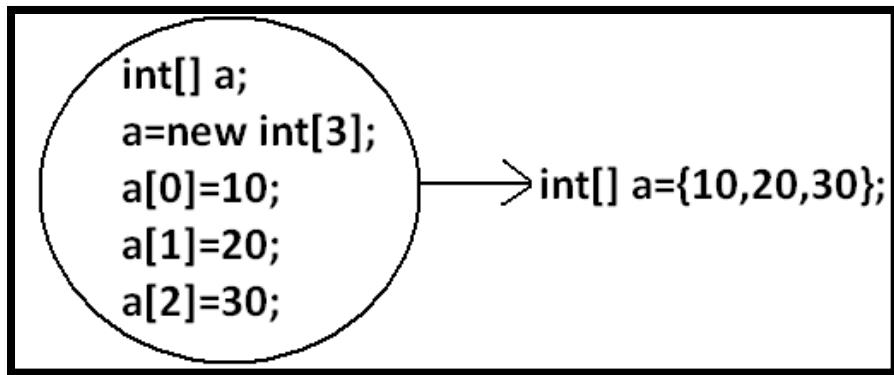
Answer: E



5) Declaration, construction and initialization of an array in a single line:

We can perform declaration, construction and initialization of an array in a single line.

Example:



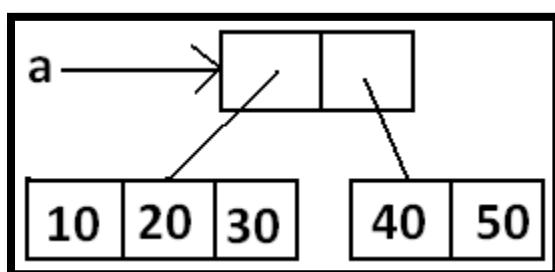
char[] ch={'a','e','i','o','u'};(valid)

String[] s={"balayya","venki","nag","chiru"};(valid)

We can extend this short cut even for multi dimensional arrays also.

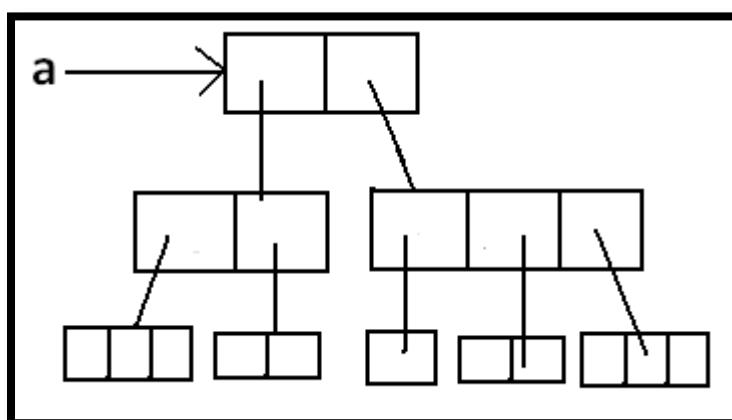
Example:

int[][] a={{10,20,30},{40,50}};



Example:

int[][][] a={{{{10,20,30},{40,50}},{{60},{70,80}},{{90,100,110}}};



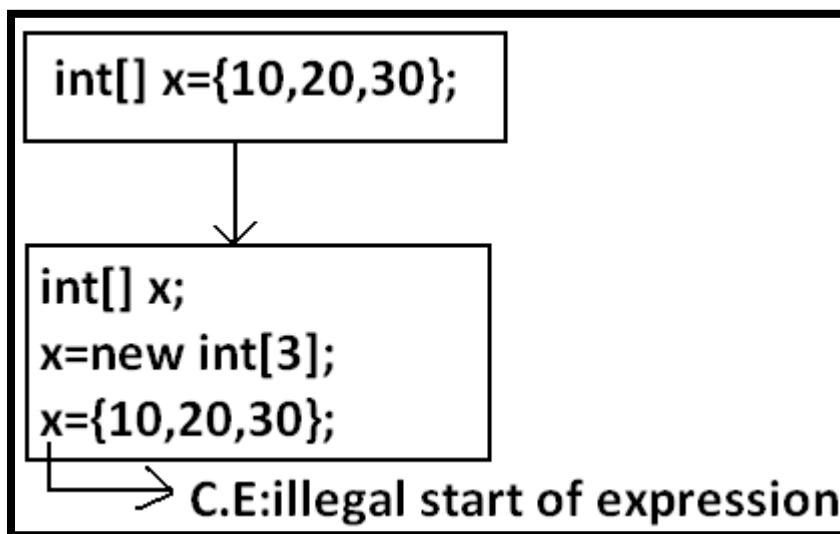


```
int[][][] a={{ {10,20,30},{40,50}},{ {60},{70,80},{90,100,110}}};  
System.out.println(a[0][1][1]);//50(valid)  
System.out.println(a[1][0][2]);//R.E:ArrayListOutOfBoundsException: 2(invalid)  
System.out.println(a[1][2][1]);//100(valid)  
System.out.println(a[1][2][2]);//110(valid)  
System.out.println(a[2][1][0]);//R.E:ArrayListOutOfBoundsException: 2(invalid)  
System.out.println(a[1][1][1]);//80(valid)
```

If we want to use this short cut compulsory we should perform declaration, construction and initialization in a single line.

If we are trying to divide into multiple lines then we will get compile time error.

Example:



Q. Given the following code

```
int[] x= {10,20,30,40,50};  
x[2]=x[4];  
x[4]=60;
```

After executing this code Array elements are

- A) 10,20,30,40,50
- B) 10,20,50,40,50
- C) 10,20,50,40,60
- D) 10,20,30,40,50

Answer: C



Q. Given Student class as

```
1) class Student
2) {
3)     int rollno;
4)     String name;
5)     public Student(int rollno, String name)
6)     {
7)         this.rollno=rollno;
8)         this.name=name;
9)     }
10) }
11) public class Test
12) {
13)     public static void main(String[] args)
14)     {
15)         Student[] students ={
16)             new Student(101, "Durga"),
17)             new Student(102, "Ravi"),
18)             new Student(103, "Shiva"),
19)             new Student(104, "Pavan")
20)         };
21)         System.out.println(students);
22)         System.out.println(students[2]);
23)         System.out.println(students[2].rollno);
24)     }
25) }
```

What is the output?

- A) students
Shiva
103
- B) [LStudent;@61baa894
Shiva
103
- C) [LStudent;@61baa894
Student@66133adc
103
- D) [LStudent;@61baa894
Pavan
103

Answer: C



6) length Vs length():

length:

It is the final variable applicable only for arrays.
It represents the size of the array.

Example:

```
int[] x=new int[3];
System.out.println(x.length());//C.E: cannot find symbol
System.out.println(x.length());//3
```

length() method:

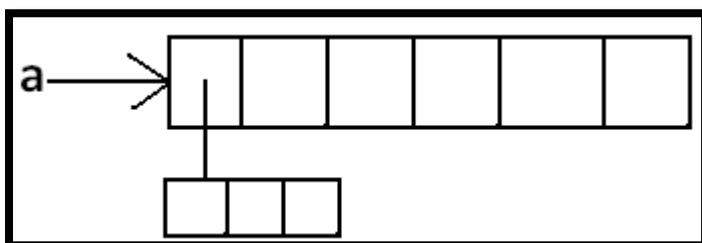
It is a final method applicable for String objects.
It returns the no of characters present in the String.

Example:

```
String s="durga";
System.out.println(s.length());//C.E:cannot find symbol
System.out.println(s.length());//5
In multidimensional arrays length variable represents only base size but not total size.
```

Example:

```
int[][] a=new int[6][3];
System.out.println(a.length);//6
System.out.println(a[0].length);//3
```



length variable applicable only for arrays where as length() method is applicable for String objects.
There is no direct way to find total size of multi dimensional array but indirectly we can find as follows
 $x[0].length + x[1].length + x[2].length + \dots$



Q. Consider the following code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String[] courses={"Java","Python","Testing","SAP"};
6)         System.out.println(courses.length);
7)         System.out.println(courses[1].length());
8)
9)     }
10) }
```

What is the output?

- A) 4
6
- B) 4
4
- C) 6
4
- D) 6
6

Answer: A



7) Anonymous Arrays:

Sometimes we can create an array without name such type of nameless arrays are called anonymous arrays.

The main objective of anonymous arrays is "just for instant use".

We can create anonymous array as follows.

```
new int[]{10,20,30,40};(valid)
new int[][]{{10,20},{30,40}};(valid)
```

At the time of anonymous array creation we can't specify the size otherwise we will get compile time error.

Example:

```
new int[3]{10,20,30,40};//C.E:';' expected(invalid)
new int[]{10,20,30,40};(valid)
```

Based on our programming requirement we can give the name for anonymous array then it is no longer anonymous.

Example:

```
int[] a=new int[]{10,20,30,40};(valid)
```

Example:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         System.out.println(sum(new int[]{10,20,30,40}));//100
6)     }
7)     public static int sum(int[] x)
8)     {
9)         int total=0;
10)        for(int x1:x)
11)        {
12)            total=total+x1;
13)        }
14)        return total;
15)    }
16) }
```

In the above program just to call sum() , we required an array but after completing sum() method. For this instant use we can use anonymous arrays.



8) Array element assignments:

Case 1:

In the case of primitive array as array element any type is allowed which can be promoted to declared type.

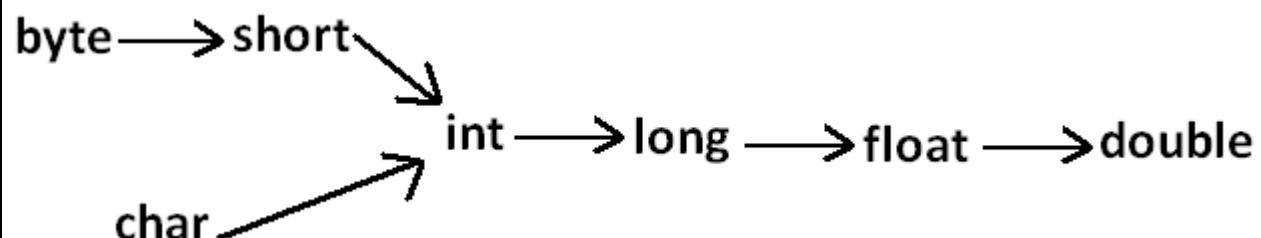
Example 1:

For the int type arrays the allowed array element types are byte, short, char, int.

```
int[] a=new int[10];
a[0]=97;//(valid)
a[1]='a';//(valid)
byte b=10;
a[2]=b;//(valid)
short s=20;
a[3]=s;//(valid)
a[4]=10l;//C.E:possible loss of precision
```

Example 2:

For float type arrays the allowed element types are byte, short, char, int, long, float.



Case 2:

In the case of Object type arrays as array elements we can provide either declared type objects or its child class objects.

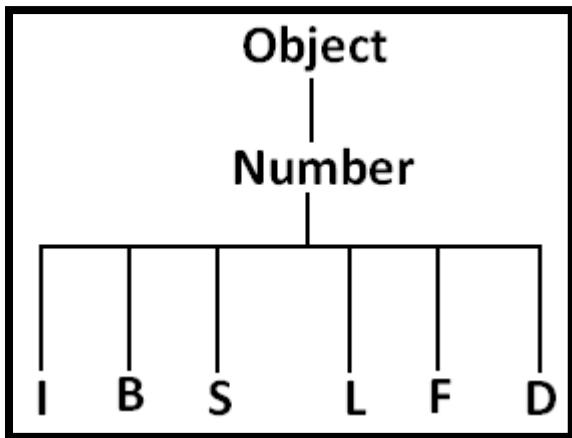
Example 1:

```
Object[] a=new Object[10];
a[0]=new Integer(10);//(valid)
a[1]=new Object();//(valid)
a[2]=new String("durga");//(valid)
```



Example 2:

```
Number[] n=new Number[10];
n[0]=new Integer(10);//(valid)
n[1]=new Double(10.5);//(valid)
n[2]=new String("durga");//C.E:incompatible types//(invalid)
```



Case 3:

In the case of interface type arrays as array elements we can provide its implemented class objects.

Example:

```
Runnable[] r=new Runnable[10];
r[0]=new Thread();
r[1]=new String("bhaskar");//C.E: incompatible types
```

Array Type	Allowed Element Type
1) Primitive arrays.	1) Any type which can be promoted to declared type.
2) Object type arrays.	2) Either declared type or its child class objects allowed.
3) Interface type arrays.	3) Its implemented class objects allowed.
4) Abstract class type arrays.	4) Its child class objects are allowed.



Q. The following grid shows the state of a 2D array:

Diagram

This grid is created with the following code

```
char[][] grid= new char[3][3];
grid[0][0]='Y';
grid[0][1]='Y';
grid[1][1]='X';
grid[1][2]='Y';
grid[2][1]='X';
grid[2][2]='X';
// Line-1
```

Which line inserted at Line-1 so that grid contains 3 consecutive X's?

- A) grid[3][1]='X';
- B) grid[0][2]='X';
- C) grid[1][3]='X';
- D) grid[2][0]='X';
- E) grid[1][2]='X';

Answer: D

9) Array variable assignments:

Case 1:

Element level promotions are not applicable at array object level.

Ex: A char value can be promoted to int type but char array cannot be promoted to int array.

Example:

```
int[] a={10,20,30};
char[] ch={'a','b','c'};
int[] b=a;//(valid)
int[] c=ch;//C.E:incompatible types(invalid)
```

Which of the following promotions are valid?



- | | |
|-------------|-------------------|
| 1) char | int (valid) |
| 2) char[] | int[] (invalid) |
| 3) int | long (valid) |
| 4) int[] | long[] (invalid) |
| 5) double | float (invalid) |
| 6) double[] | float[] (invalid) |
| 7) String | Object (valid) |
| 8) String[] | Object[] (valid) |

Note: In the case of object type arrays child type array can be assign to parent type array variable.

Example:

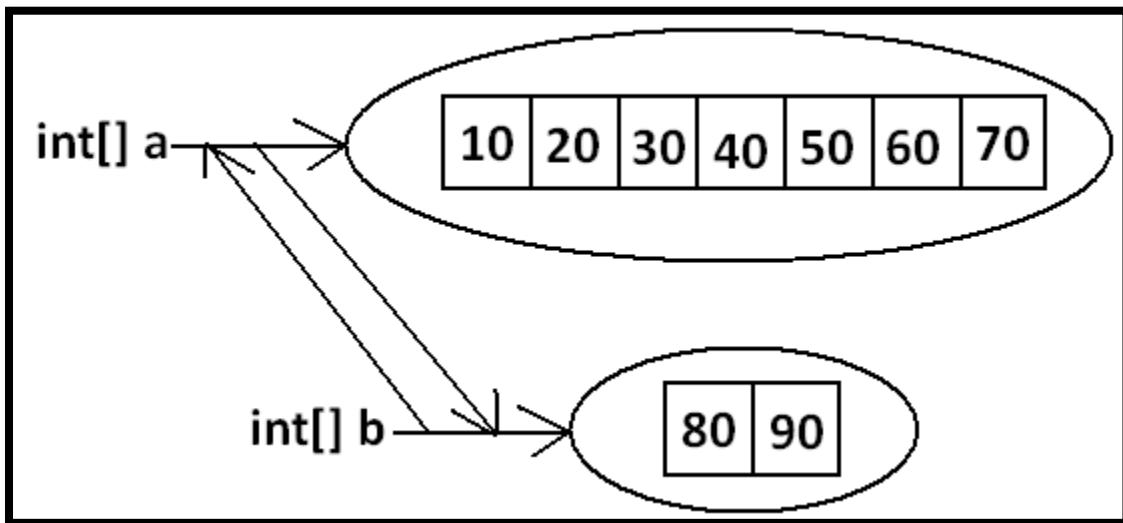
```
String[] s={"A","B"};
Object[] o=s;
```

Case 2:

Whenever we are assigning one array to another array internal elements won't be copied, just reference variables will be reassigned .Hence sizes are not required to be equal, but types must be matched.

Example:

```
int[] a={10,20,30,40,50,60,70};
int[] b={80,90};
a=b;//(valid)
b=a;//(valid)
```



Case 3:

Whenever we are assigning one array to another array dimensions must be matched that is in the place of one dimensional array we should provide the same type only otherwise we will get compile time error.

Example:

```
int[][] a=new int[3][];
a[0]=new int[4][5];//C.E:incompatible types(invalid)
a[0]=10;//C.E:incompatible types(invalid)
a[0]=new int[4];//(valid)
```

Note: Whenever we are performing array assignments the types and dimensions must be matched but sizes are not important.

Q. Consider the code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         int[] n1= new int[3];
6)         int[] n2={10,20,30,40,50};
7)         n1=n2;
8)         for(int x : n1)
9)         {
10)             System.out.print(x+":");
11)         }
12)
13)     }
14) }
```



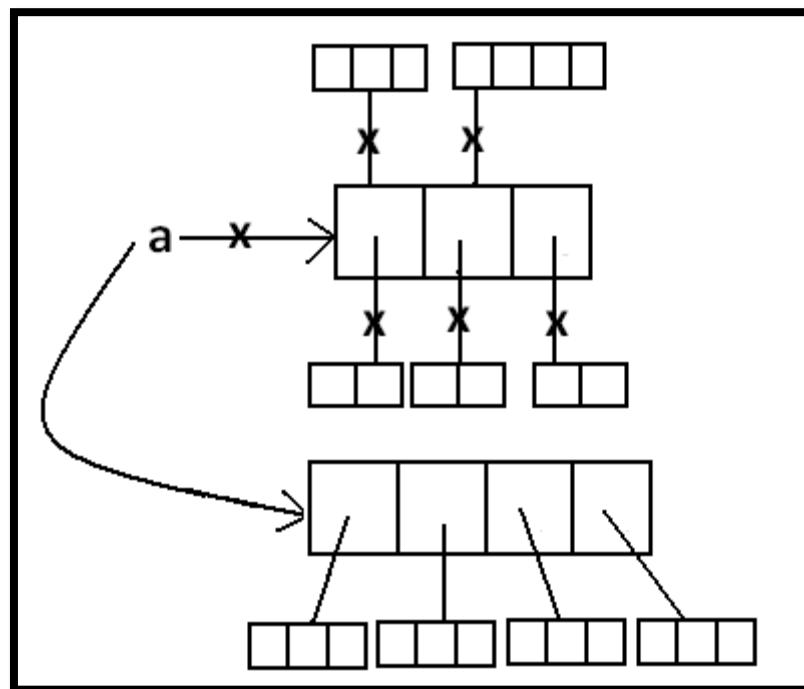
What is the output?

- A) 10:20:30:40:50:
- B) 10:20:30:
- C) Compilation fails
- D) ArrayIndexOutOfBoundsException at runtime

Answer: A

Example 1:

```
int[][] a=new int[3][2];
a[0]=new int[3];
a[1]=new int[4];
a=new int[4][3];
```



Total how many objects created?

Ans: 11

How many objects eligible for GC: 6



Example 2:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String[] argh={"A","B"};
6)         args=argh;
7)         System.out.println(args.length); //2
8)         for(int i=0;i<args.length;i++)
9)         {
10)             System.out.println(args[i]);
11)         }
12)     }
13) }
```

Output:

```
java Test x y
R.E: ArrayIndexOutOfBoundsException: 2
java Test x
R.E: ArrayIndexOutOfBoundsException: 2
java Test
R.E: ArrayIndexOutOfBoundsException: 2
Note: Replace with i<args.length
```

Example 3:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String[] argh={"A","B"};
6)         args=argh;
7)         System.out.println(args.length); //2
8)         for(int i=0;i<args.length;i++)
9)         {
10)             System.out.println(args[i]);
11)         }
12)     }
13) }
```

Output:

```
2
A
B
```



Example 4:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String[] argh={"A","B"};
6)         args=argh;
7)
8)         for(String s : args) {
9)             System.out.println(s);
10)    }
11) }
```

Output:

A
B



Types of Variables

Division 1: Based on the type of value represented by a variable all variables are divided into 2 types. They are:

- Primitive variables
- Reference variables

Primitive variables:

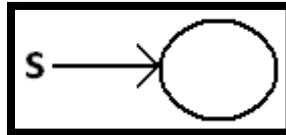
Primitive variables can be used to represent primitive values.

Example: int x=10;

Reference variables:

Reference variables can be used to refer objects.

Example: Student s=new Student();



Division 2: Based on the behavior and position of declaration all variables are divided into the following 3 types.

- Instance variables
- Static variables
- Local variables

Instance variables:

- If the value of a variable is varied from object to object such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.
- Instance variables will be created at the time of object creation and destroyed at the time of object destruction hence the scope of instance variables is exactly same as scope of objects.
- Instance variables will be stored on the heap as the part of object.
- Instance variables should be declared with in the class directly but outside of any method or block or constructor.
- Instance variables can be accessed directly from Instance area. But cannot be accessed directly from static area.
- But by using object reference we can access instance variables from static area.



Example:

```
1) class Test
2) {
3)     int i=10;
4)     public static void main(String[] args)
5)     {
6)         //System.out.println(i);
7)     //C.E:non-static variable i cannot be referenced from a static context(invalid)
8)     Test t=new Test();
9)     System.out.println(t.i); //10(valid)
10)    t.methodOne();
11) }
12) public void methodOne()
13) {
14)     System.out.println(i); //10(valid)
15) }
16) }
```

- For the instance variables it is not required to perform initialization JVM will always provide default values.

Example:

```
1) class Test
2) {
3)     boolean b;
4)     public static void main(String[] args)
5)     {
6)         Test t=new Test();
7)         System.out.println(t.b); //false
8)     }
9) }
```

- Instance variables also known as object level variables or attributes.

Static variables:

- If the value of a variable is not varied from object to object such type of variables is not recommended to declare as instance variables. We have to declare such type of variables at class level by using static modifier.
- In the case of instance variables for every object a separate copy will be created but in the case of static variables for entire class only one copy will be created and shared by every object of that class.
- Static variables will be created at the time of class loading and destroyed at the time of class unloading hence the scope of the static variable is exactly same as the scope of the .class file.



- Static variables will be stored in method area. Static variables should be declared with in the class directly but outside of any method or block or constructor.
- Static variables can be accessed from both instance and static areas directly.
- We can access static variables either by class name or by object reference but usage of class name is recommended.
- But within the same class it is not required to use class name we can access directly.

- java Test
- Start JVM.
- Create and start Main Thread by JVM
- Locate(find) Test.class by main Thread.
- Load Test.class by main Thread // static variable creation
- Execution of main() method.
- Unload Test.class // static variable destruction
- Terminate main Thread.
- Shutdown JVM.

Example:

```
1) class Test
2) {
3)     static int i=10;
4)     public static void main(String[] args)
5)     {
6)         Test t=new Test();
7)         System.out.println(t.i); //10
8)         System.out.println(Test.i); //10
9)         System.out.println(i); //10
10)    }
11) }
```

For the static variables it is not required to perform initialization explicitly, JVM will always provide default values.

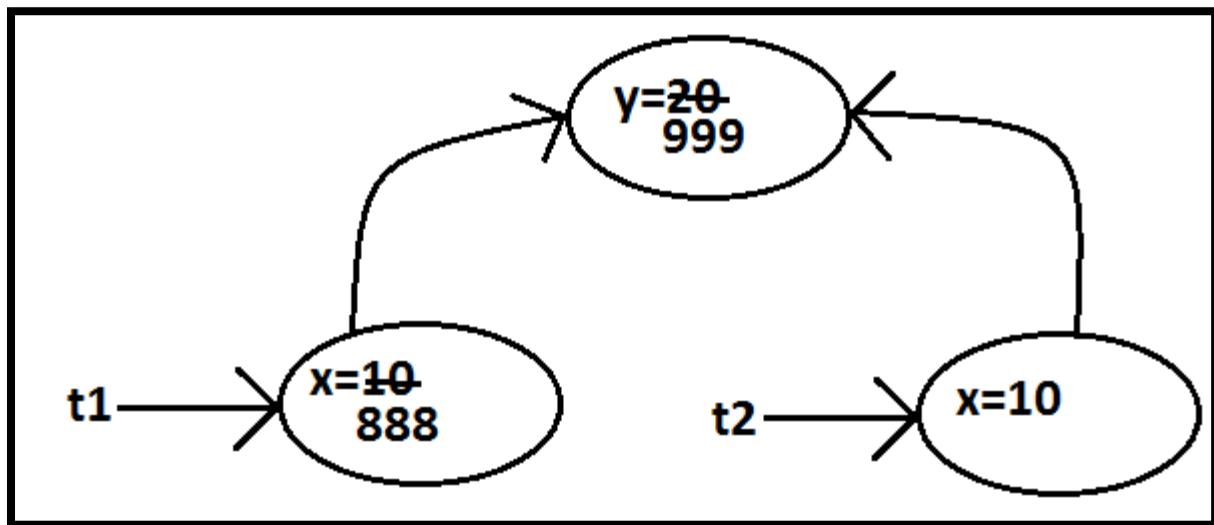
Example:

```
1) class Test
2) {
3)     static String s;
4)     public static void main(String[] args)
5)     {
6)         System.out.println(s); //null
7)     }
8) }
```



Example:

```
1) class Test
2) {
3)     int x=10;
4)     static int y=20;
5)     public static void main(String[] args)
6)     {
7)         Test t1=new Test();
8)         t1.x=888;
9)         t1.y=999;
10)        Test t2=new Test();
11)        System.out.println(t2.x+"----"+t2.y);//10----999
12)    }
13) }
```



Static variables also known as class level variables or fields.

Local variables:

- Some times to meet temporary requirements of the programmer we can declare variables inside a method or block or constructors such type of variables are called local variables or automatic variables or temporary variables or stack variables.
- Local variables will be stored inside stack.
- The local variables will be created as part of the block execution in which it is declared and destroyed once that block execution completes. Hence the scope of the local variables is exactly same as scope of the block in which we declared.



Example 1:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         int i=0;
6)         for(int j=0;j<3;j++)
7)         {
8)             i=i+j;
9)         }
10)    }
11) }
```

System.out.println(i+"----"+j);

C.E →

```
javac Test.java
Test.java:10: cannot find symbol
  symbol : variable j
  location: class Test
```

Example 2:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         try
6)         {
7)             int i=Integer.parseInt("ten");
8)         }
9)         catch(NullPointerException e)
10)        {
11)        }
12)    }
13) }
```



`System.out.println(i);`

C.E

```
javac Test.java
Test.java:11: cannot find symbol
  symbol : variable i
  location: class Test
```

- The local variables will be stored on the stack.
- For the local variables JVM won't provide any default values compulsory we should perform initialization explicitly before using that variable.

Example:

<code>class Test { public static void main(String[] args) { int x; System.out.println("hello");//hello } }</code>	<code>class Test { public static void main(String[] args) { int x; System.out.println(x);//C.E:variable x might not have been initialized } }</code>
---	--

Example:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         int x;
6)         if(args.length>0)
7)         {
8)             x=10;
9)         }
10)        System.out.println(x);
11)        //C.E:variable x might not have been initialized
12)    }
13) }
```



Example:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         int x;
6)         if(args.length>0)
7)         {
8)             x=10;
9)         }
10)        else
11)        {
12)            x=20;
13)        }
14)        System.out.println(x);
15)    }
16) }
```

Output:

```
java Test x
10
```

```
java Test x y
10
```

```
java Test
20
```

- It is never recommended to perform initialization for the local variables inside logical blocks because there is no guarantee of executing that block always at runtime.
- It is highly recommended to perform initialization for the local variables at the time of declaration at least with default values.

Q. Consider the code

```
1) public class Triangle
2) {
3)     static double area;
4)     int b=30,h=40;
5)     public static void main(String[] args)
6)     {
7)         double p,b,h;// Line-1
8)         if(area ==0)
9)         {
10)             b=3;
11)             h=4;
```



```
12)      p=0.5;
13)  }
14)  area=p*b*h;// Line-2
15)  System.out.println(area);
16)  }
17) }
```

What is the result?

- A) 6.0
- B) 3.0
- C) Compilation fails at Line-1
- D) Compilation fails at Line-2

Answer: D

Note: The only applicable modifier for local variables is final. If we are using any other modifier we will get compile time error.

Example:

```
1) class Test
2) {
3)  public static void main(String[] args)
4)  {
5)
6)  public int x=10; //(invalid)
7)  private int x=10; //(invalid)
8)  protected int x=10; //(invalid) C.E: illegal start of expression
9)  static int x=10; //(invalid)
10) volatile int x=10; //(invalid)
11) transient int x=10; //(invalid)
12) final int x=10; //(valid)
13) }
14) }
```

Conclusions:

- For the static and instance variables it is not required to perform initialization explicitly JVM will provide default values. But for the local variables JVM won't provide any default values compulsory we should perform initialization explicitly before using that variable.
- For every object a separate copy of instance variable will be created whereas for entire class a single copy of static variable will be created. For every Thread a separate copy of local variable will be created.
- Instance and static variables can be accessed by multiple Threads simultaneously and hence these are not Thread safe but local variables can be accessed by only one Thread at a time and hence local variables are Thread safe.



- If we are not declaring any modifier explicitly then it means default modifier but this rule is applicable only for static and instance variables but not local variable.

Example:

```
1) class Test
2) {
3)     int[] a;
4)     public static void main(String[] args)
5)     {
6)         Test t1=new Test();
7)         System.out.println(t1.a); //null
8)         System.out.println(t1.a[0]); //R.E:NullPointerException
9)     }
10) }
```

Instance level:

Example 1:

```
1) int[] a;
2) System.out.println(obj.a); //null
3) System.out.println(obj.a[0]); //R.E:NullPointerException
```

Example 2:

```
1) int[] a=new int[3];
2) System.out.println(obj.a); //[@3e25a5
3) System.out.println(obj.a[0]); //0
```

Static level:

Example 1:

```
1) static int[] a;
2) System.out.println(a); //null
3) System.out.println(a[0]); //R.E:NullPointerException
```

Example 2:

```
1) static int[] a=new int[3];
2) System.out.println(a); //[@3e25a5
3) System.out.println(a[0]); //0
```



Local level:

Example 1:

- 1) `int[] a;`
- 2) `System.out.println(a); //C.E: variable a might not have been initialized`
- 3) `System.out.println(a[0]);`

Example 2:

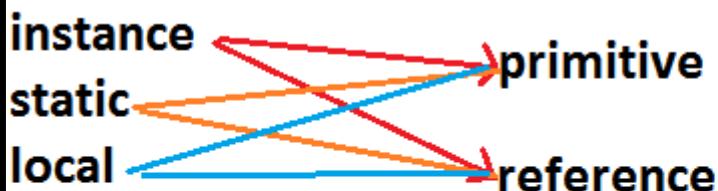
- 1) `int[] a=new int[3];`
- 2) `System.out.println(a);//[I@3e25a5`
- 3) `System.out.println(a[0]);//0`

Note:

Once we create an array every element is always initialized with default values irrespective of whether it is static or instance or local array.

Note:

- Every variable in Java should be either instance or static or local.
- Every variable in Java should be either primitive or reference.
- Hence the following are the various possible combinations for variables



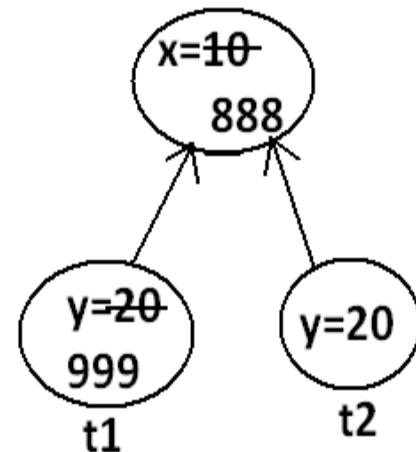
```
class Test {  
    int[] a=new int[3];    // instance-reference  
    static int x=20;        //static-primitive  
    public static void main(String[] args) {  
        String s="xyz";      //local-reference  
    }  
}
```



- Static is the modifier applicable for methods, variables and blocks.
- We can't declare a class with static but inner classes can be declared as the static.
- In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by all objects of that class.

Example:

```
class Test{  
    static int x=10;  
    int y=20;  
    public static void main(String args[]){  
        Test t1=new Test();  
        t1.x=888;  
        t1.y=999;  
        Test t2=new Test();  
        System.out.println(t2.x+"...."+t2.y);  
    }  
}
```



Output:

```
D:\Java>javac Test.java  
D:\Java>java Test  
888....20
```

- Instance variables can be accessed only from instance area directly and we can't access from static area directly.
- But static variables can be accessed from both instance and static areas directly.

Q. Consider the following 4 declarations

- 1) int x=10;
- 2) static int x=10;
- 3) public void m1() {
 System.out.println(x);
 }



```
4) public static void m1() {  
    System.out.println(x);  
}
```

Which of the following declarations are allowed within the same class simultaneously ?

- a) 1 and 3
- b) 1 and 4
- c) 2 and 3
- d) 2 and 4
- f) 1 and 2
- f) 3 and 4

Answer: a,c,d

Example:

```
1) class Test {  
2)     int x = 10;  
3)     public void methodOne() {  
4)         System.out.println(x);  
5)     }  
6) }
```

Output:

Compile successfully.

Example:

```
1) class Test {  
2)     int x = 10;  
3)     public static void methodOne() {  
4)         System.out.println(x);  
5)     }  
6) }
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: non-static variable x cannot be referenced from a static context

System.out.println(x);

Example:

```
1) class Test {  
2)     static int x = 10;  
3)     public void methodOne() {  
4)         System.out.println(x);
```



```
5)    }
6) }
```

Output:

Compile successfully.

Example:

```
1) class Test {
2)     static int x = 10;
3)     public static void methodOne() {
4)         System.out.println(x);
5)     }
6) }
```

Output:

Compile successfully.

e) 1 and 2

Example:

```
1) class Test {
2)     int x = 10;
3)     static int x = 10;
4) }
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:4: x is already defined in Test

static int x=10;

Example:

```
1) class Test {
2)     public void methodOne() {
3)         System.out.println(x);
4)     }
5)     public static void methodOne() {
6)         System.out.println(x);
7)     }
8) }
```

Output:

Compile time error.

D:\Java>javac Test.java



Test.java:5: methodOne() is already defined in Test

```
public static void methodOne(){
```

For static methods implementation should be available but for abstract methods implementation is not available hence static abstract combination is illegal for methods.

Overloading concept is applicable for static method including main method also. But JVM will always call String[] args main method .

The other overloaded method we have to call explicitly then it will be executed just like a normal method call .

Example:

```
class Test{  
    public static void main(String args[]){  
        System.out.println("String() method is called");  
    }  
    public static void main(int args[]){  
        System.out.println("int() method is called");  
    }  
}
```

This method we have to call explicitly.

Output :

String() method is called

Inheritance concept is applicable for static methods including main() method hence while executing child class, if the child doesn't contain main() method then the parent class main method will be executed.

Example:

```
1) class Parent {  
2)     public static void main(String args[]){  
3)         System.out.println("parent main() method called");  
4)     }  
5) }  
6) class child extends Parent {}
```



Output:

```
javac Parent.java
                ↓           ↓
Parent.class    Child.class
                ↓
java Parent ←
D:\Java>java Parent
parent main() method called
D:\Java>java child
parent main() method called
```

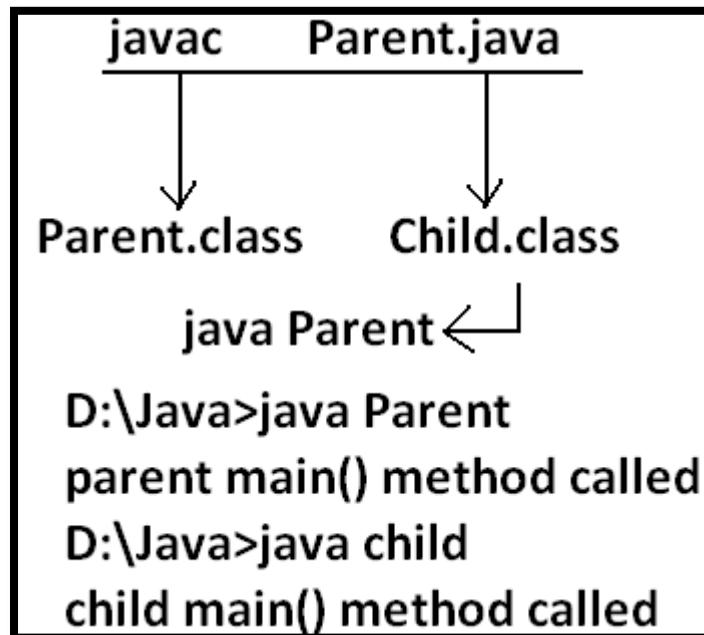
Example:

```
class Parent{
    public static void main(String args[]){
        System.out.println("parent main() method called");
    }
}
class child extends Parent{
    public static void main(String args[]){
        System.out.println("child main() method called");
    }
}
```

it is not overriding but method hiding.



Output:



It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

Q. Consider the following code

```
1) public class Test
2) {
3)     String myStr="7007";
4)     public void doStuff(String s)
5)     {
6)         int myNum=0;
7)         try
8)         {
9)             String myStr=s;
10)            myNum=Integer.parseInt(myStr);
11)        }
12)        catch(NumberFormatException e)
13)        {
14)            System.err.println("Error");
15)        }
16)        System.out.println("myStr: "+myStr+"," +myNum);
17)    }
18)    public static void main(String[] args)
19)    {
20)        Test t = new Test();
21)        t.doStuff("9009");
22)    }
23) }
```



- A) myStr: 9009 ,myNum: 9009
- B) myStr: 7007 ,myNum: 7007
- C) myStr: 7007 ,myNum: 9009
- D) Compilation Fails

Answer: C

Explanation: In the above example, the variable myStr, which is declared inside try block is not available outside of try block. Hence while printing, instance variable myStr will be considered.

Q. Consider the following code

```
1) public class Test
2) {
3)     public void doStuff(String s)
4)     {
5)         int myNum=0;
6)         try
7)         {
8)             String myStr=s;
9)             myNum=Integer.parseInt(myStr);
10)        }
11)        catch(NumberFormatException e)
12)        {
13)            System.err.println("Error");
14)        }
15)        System.out.println("myStr: "+myStr+" ,myNum: "+myNum);
16)    }
17)    public static void main(String[] args)
18)    {
19)        Test t = new Test();
20)        t.doStuff("9009");
21)    }
22) }
```

- A) myStr: 9009 ,myNum: 9009
- B) myStr: 7007 ,myNum: 7007
- C) myStr: 7007 ,myNum: 9009
- D) Compilation Fails

Answer: D

Explanation: myStr is local variable of try block and we cannot access outside of try block.



Q. Consider the code

```
1) class Test
2) {
3)     int x =10;
4)     static int y = 20;
5)     public static void main(String[] args)
6)     {
7)         Test t1 =new Test();
8)         Test t2 =new Test();
9)         t1.x=100;
10)        t1.y=200;
11)        t2.x=300;
12)        t2.y=400;
13)        System.out.println(t1.x+..+t1.y+..+t2.x+..+t2.y);
14)    }
15) }
```

- A) 100..400..300..400
- B) 100..400..100..400
- C) 200..400..300..400
- D) 100..200..300..400

Answer: A

Q. Consider the following code

```
1) public class Test
2) {
3)     static int x;
4)     int y;
5)     public static void main(String[] args)
6)     {
7)         Test t1 = new Test();
8)         Test t2 = new Test();
9)         t1.x=3;
10)        t1.y=4;
11)        t2.x=5;
12)        t2.y=6;
13)        System.out.println(t1.x+":"+t1.y+":"+t2.x+":"+t2.y);
14)    }
15) }
```

What is the result?

- A) 3:4:5:6
- B) 3:4:3:6
- C) 5:4:5:6
- D) 3:6:4:6

Answer: C



Q. Consider the code

```
1) public class Test
2) {
3)     static int count=0;
4)     int i=0;
5)     public void modify()
6)     {
7)         while(i<5)
8)         {
9)             i++;
10)            count++;
11)        }
12)    }
13)    public static void main(String[] args)
14)    {
15)        Test t1 = new Test();
16)        Test t2 = new Test();
17)        t1.modify();
18)        t2.modify();
19)        System.out.println(t1.count+".."+t2.count);
20)    }
21)
22} }
```

What is the result?

- A) 10..10
- B) 5..5
- C) 5..10
- D) Compilation Fails

Answer: A

Q. Consider the code

```
1) class Test
2) {
3)     int count;
4)     public static void display()
5)     {
6)         count++;//Line-1
7)         System.out.println("Welcome Visit Count:"+count);//Line-2
8)     }
9)     public static void main(String[] args)
10)    {
11)        Test.display();//Line-3
12)        Test.display();//Line-4
13)    }
```



14) }

What is the result?

- A) Welcome Visit Count: 1
Welcome Visit Count: 2
- A) Welcome Visit Count: 1
Welcome Visit Count: 1
- C) Compilation Fails at Line-1 and Line-2
- D) Compilation Fails at Line-3 and Line-4

Answer: C

Q. Consider the code

```
1) public class Test
2) {
3)     public static int x=100;
4)     public int y = 200;
5)     public String toString()
6)     {
7)         return y+":"+x;
8)     }
9)     public static void main(String[] args)
10)    {
11)        Test t1 = new Test();
12)        t1.y=300;
13)        System.out.println(t1);
14)        Test t2 = new Test();
15)        t2.x=300;
16)        System.out.println(t2);
17)    }
18)
19) }
```

What is the result?

- A) 300:100
200:300
- B) 200:300
200:300
- C) 300:0
0:300
- D) 300:300
200:300

Answer: A



Q. Consider the code

```
1) class Demo
2) {
3)     int ns;
4)     static int s;
5)     Demo(int ns)
6)     {
7)         if(s<ns)
8)         {
9)             s=ns;
10)            this.ns=ns;
11)        }
12)    }
13)    void display()
14)    {
15)        System.out.println("ns = "+ns+ " s = "+s);
16)    }
17) }
18) public class Test
19) {
20)     static int x;
21)     int y;
22)     public static void main(String[] args)
23)     {
24)         Demo d1= new Demo(50);
25)         Demo d2= new Demo(125);
26)         Demo d3= new Demo(100);
27)         d1.display();
28)         d2.display();
29)         d3.display();
30)     }
31) }
```

- A. ns = 50 s = 125
ns = 125 s = 125
ns = 0 s = 125
- B. ns = 50 s = 125
ns = 125 s = 125
ns = 100 s = 125
- C. ns = 50 s = 50
ns = 125 s = 125
ns = 0 s = 0
- D. ns = 50 s = 125
ns = 125 s = 125



ns = 100 s = 100

Answer: A

Q. Consider the code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         int x =200;
6)         System.out.print(m1(x));
7)         System.out.print(" "+x);
8)     }
9)     public static int m1(int x)
10)    {
11)        x=x*2;
12)        return x;
13)    }
14) }
```

What is the result?

- A) 400 200
- B) 200 200
- C) 400 400
- D) Compilation Fails

Answer: A

Q. Consider the code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         try
6)         {
7)             int n=10;
8)             int d=0;
9)             int ans=n/d;
10)        }
11)        catch (ArithmaticException e)
12)        {
13)            ans=0;//Line-1
14)        }
15)        catch(Exception e)
16)        {
```



```
17)     System.out.println("Invalid Calculation");
18) }
19)     System.out.println("Answer="+ans);//Line-2
20) }
21) }
```

What is the result?

- A) Answer=0
- B) Invalid Calculation
- C) Compilation Fails at Line-1
- D) Compilation Fails at Line-2
- E) Compilation Fails at Line-1 and Line-2

Answer: E

Q. Consider the code

```
1) public class Test
2) {
3)     char c;
4)     boolean b;
5)     float f;
6)     public void print()
7)     {
8)         System.out.println("c = "+c);
9)         System.out.println("b = "+b);
10)        System.out.println("f = "+f);
11)    }
12)    public static void main(String[] args)
13)    {
14)        Test t = new Test();
15)        t.print();
16)    }
17) }
```

What is the result?

- A. c =
b = false
f = 0.0
- B. c = null
b = false
f = 0.0
- C. c = 0
b = false



f = 0.0

D. c =
b = true
f = 0.0

Answer: A

Passing Arguments:

If we pass primitive type to a method and within that method if we perform any changes then those changes won't be reflected to the caller. In this case a separate local copy will be created for the primitive variable in that method.

```
1) public class Test
2) {
3)     public void m1(int i, int j)
4)     {
5)         i=i+10;
6)         j=j+10;
7)         System.out.println("Inside Method:"+i+".."+j);
8)     }
9)     public static void main(String[] args)
10)    {
11)        int x=100;
12)        int y =200;
13)        Test t = new Test();
14)        t.m1(x,y);
15)        System.out.println("After Completing Method:"+x+".."+y);
16)    }
17) }
```

Output

Inside Method:110..210
After Completing Method:100..200

Q. Consider the code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         int x=200;
6)         System.out.print(m1(x));
7)         System.out.print(" "+x);
8)     }
9)     public static int m1(int x)
```



```
10) {
11)     x= x*2;
12)     return x;
13) }
14) }
```

What is the result?

- A) 400 200
- B) 200 200
- C) 400 400
- D) Compilation Fails

Answer: A

Q. Consider the following code

```
1) public class Test
2) {
3)     int i,j;
4)     public Test(int i,int j)
5)     {
6)         initialize(i,j);
7)     }
8)     public void initialize(int i,int j)
9)     {
10)        this.i = i*i;
11)        this.j=j*j;
12)    }
13)    public static void main(String[] args)
14)    {
15)        int i =3, j= 5;
16)        Test t = new Test(i,j);
17)        System.out.println(i+"..."+j);
18)    }
19) }
```

What is the result?

- A) 9...25
- B) 0...0
- C) 3...5
- D) Compilation Fails

Answer: C



If we pass object reference to a method and within the method if we perform any changes to the object state, then those changes will be reflected to the caller also. In this case just duplicate reference variable will be created and new object won't be created.

Eg 2:

```
1) class Demo
2) {
3)     int x;
4)     int y;
5) };
6) public class Test
7) {
8)     public void m1(Demo d)
9)     {
10)         d.x=888;
11)         d.y=999;
12)     }
13)     public static void main(String[] args)
14)     {
15)         Demo d1 = new Demo();
16)         d1.x=10;
17)         d1.y=20;
18)         Test t = new Test();
19)         t.m1(d1);
20)         System.out.println(d1.x+"..."+d1.y);
21)     }
22) }
```

Output: 888...999

In the above example we are passing Demo object reference as argument to method m1(). Inside method m1(), we are changing the state of the object. These changes will be reflected to the caller.

Practice Question:

```
1) class Product
2) {
3)     double price;
4) }
5) public class Test
6) {
7)     public void updatePrice(Product p,double price)
8)     {
9)         price=price*2;
10)        p.price=p.price+price;
11)    }
12)    public static void main(String[] args)
```



```
13) {
14)     Product prt = new Product();
15)     prt.price=200;
16)     double newPrice=100;
17)
18)     Test t = new Test();
19)     t.updatePrice(prt,newPrice);
20)     System.out.println(prt.price+"...."+newPrice);
21)
22} }
```

What is the result?

- A) 200.0....100.0
- B) 400.0....400.0
- C) 400.0....100.0
- D) Compilation Fails

Answer: C

Explanation:

In the above example, we are passing Product object reference as argument to updatePrice() method and within the method we are changing the state of object. These changes will be reflected to the caller.

Q. Consider the code

```
1) public class Test
2) {
3)     int x;
4)     int y;
5)     public void doStuff(int x,int y)
6)     {
7)         this.x=x;
8)         y=this.y;
9)     }
10)    public void print()
11)    {
12)        System.out.print(x+":"+y+":");
13)    }
14)    public static void main(String[] args)
15)    {
16)        Test t1=new Test();
17)        t1.x=100;
18)        t1.y=200;
19)
20)        Test t2 = new Test();
21)        t2.doStuff(t1.x,t1.y);
22)        t1.print();
```



```
23)    t2.print();
24) }
25) }
```

What is the result?

- A) 100:200:100:0:
- B) 100:0:100:0:
- C) 100:200:100:200:
- D) 100:0:200:0:

Answer: A

Q. Consider the code

```
1) public class Vowel
2) {
3)     private char ch;
4)     public static void main(String[] args)
5)     {
6)         char ch1='a';
7)         char ch2=ch1;
8)         ch2='e';
9)
10)        Vowel obj1= new Vowel();
11)        Vowel obj2=obj1;
12)        obj1.ch='i';
13)        obj2.ch='o';
14)
15)        System.out.println(ch1+":"+ch2);
16)        System.out.println(obj1.ch+":"+obj2.ch);
17)    }
18) }
```

What is the result?

- A. a:e
o:o
- B. e:e
i:o
- C. a:e
i:o
- D. e:e
o:o

Answer: A



Main Method And Command Line Arguments

Whether the class contains main() method or not, and whether it is properly declared or not, these checking's are not responsibilities of the compiler, at runtime JVM is responsible for these. If JVM unable to find the required main() method then we will get runtime exception saying NoSuchMethodError: main.

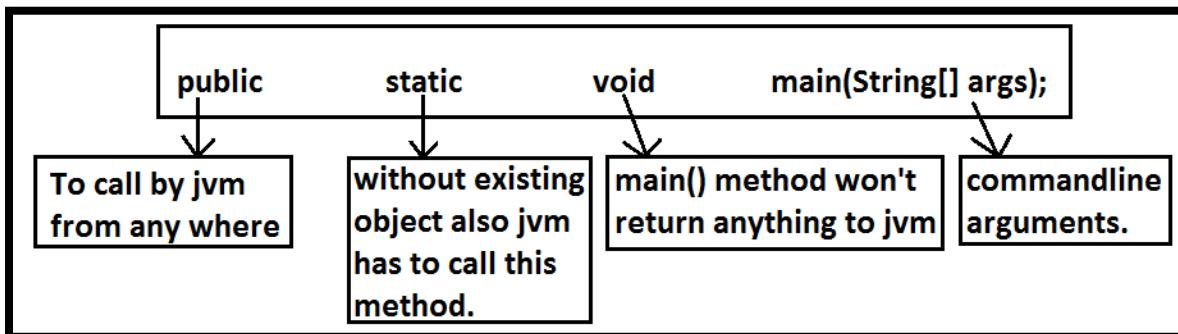
Example:

```
class Test
{}
```

Output:

```
javac Test.java
java Test R.E: NoSuchMethodError: main
```

At runtime JVM always searches for the main() method with the following prototype.



If we are performing any changes to the above syntax then the code won't run and will get Runtime exception saying NoSuchMethodError.

Even though above syntax is very strict but the following changes are acceptable to main() method.

The order of modifiers is not important that is instead of public static we can take static public.

We can declare string[] in any acceptable form

String[] args

String []args

String args[]

Instead of args we can use any valid java identifier.

We can replace string[] with var-arg parameter.



Example: main(String... args)

main() method can be declared with the following modifiers.

final, synchronized, strictfp.

```
1) public class Test
2) {
3)     static final synchronized strictfp public void main(String... durga)
4)     {
5)         System.out.println("Valid Main Method");
6)     }
7) }
```

Output: Valid Main Method

Q. Which of the following main() method declarations are valid ?

```
public static void main(String args){} (invalid)
public synchronized final strictfp void main(String[] args){} (invalid)
public static void Main(String... args){} (invalid)
public static int main(String[] args){} //int return type we can't take // (invalid)
public static synchronized final strictfp void main(String... args){} (valid)
public static void main(String... args){} (valid)
public void main(String[] args){} (invalid)
```

In which of the above cases we will get compile time error ?

No case, in all the cases we will get runtime exception.

Case 1 :

Overloading of the main() method is possible but JVM always calls string[] argument main() method only.

Example:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         System.out.println("String[] array main method"); //overloaded methods
6)     }
7)     public static void main(int[] args)
8)     {
9)         System.out.println("int[] array main method");
10)    }
11) }
```



Output:

String[] array main method

The other overloaded method we have to call explicitly then only it will be executed.

Case 2:

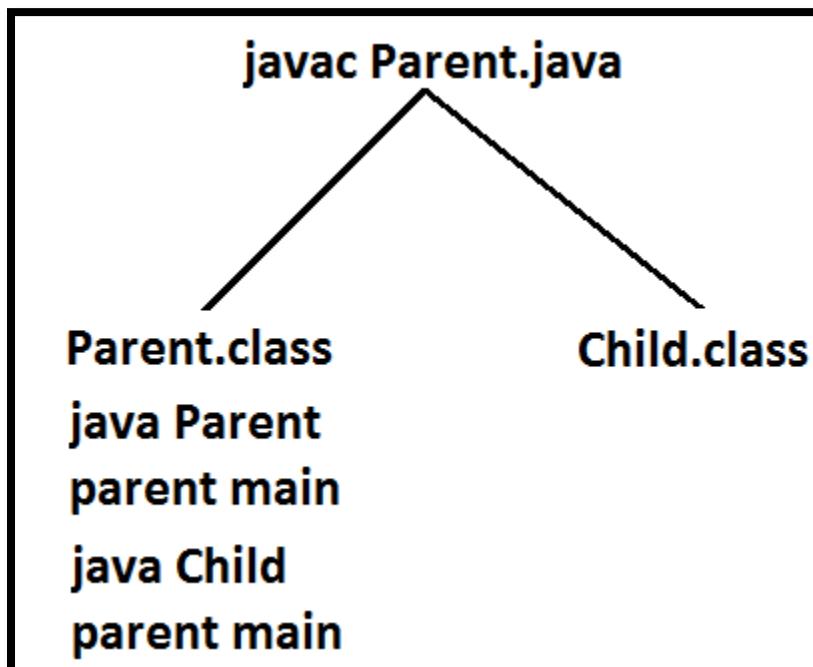
Inheritance concept is applicable for static methods including main() method hence while executing child class if the child class doesn't contain main() method then the parent class main() method will be executed.

Example 1:

Parent.java

```
1) class Parent
2) {
3)     public static void main(String[] args)
4)     {
5)         System.out.println("parent main");
6)     }
7) class Child extends Parent
8) {
9) }
```

Analysis:

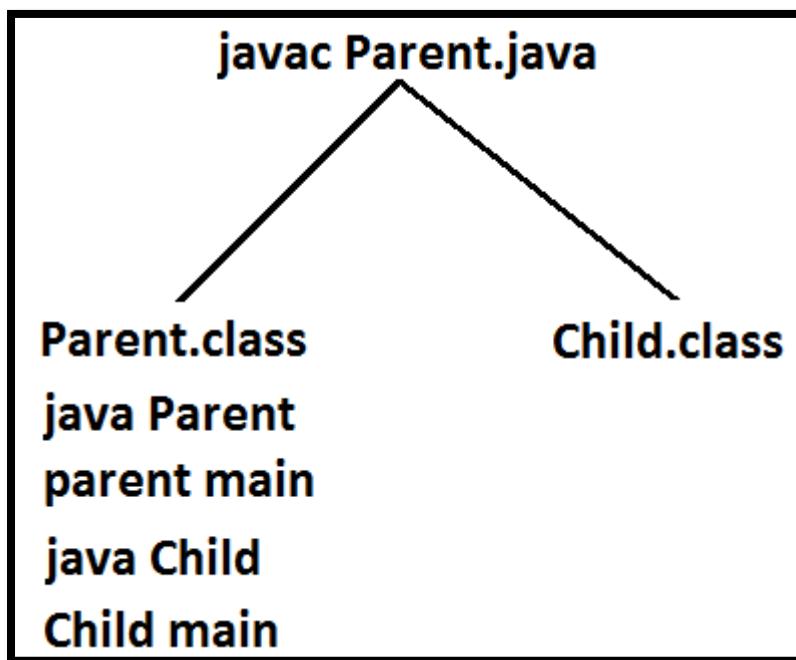




Example 2:

```
1) class Parent
2) {
3)     public static void main(String[] args)
4)     {
5)         System.out.println("parent main");    // Parent.java
6)     }
7) }
8) class Child extends Parent
9) {
10)    public static void main(String[] args)
11)    {
12)        System.out.println("Child main");
13)    }
14) }
```

Analysis:



It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.



1.7 Version Enhancements with respect to main():

Case 1 :

Until 1.6v if our class doesn't contain main() method then at runtime we will get Runtime

Exception saying NoSuchMethodError:main

But from 1.7 version onwards instead of NoSuchMethodError we will get more meaningful description

```
class Test {  
}
```

1.6 version:

```
javac Test.java
```

```
java Test
```

```
RE: NoSuchMethodError:main
```

1.7 version:

```
javac Test.java
```

```
java Test
```

```
Error: main method not found in class Test, please define the main method as  
public static void main(String[] args)
```

Case 2:

From 1.7 version onwards to start program execution compulsory main method should be required, hence even though the class contains static block if main method not available then won't be executed

```
1) class Test {  
2) static {  
3) System.out.println("static block");  
4) }  
5) }
```

1.6 version:

```
javac Test.java
```

```
java Test
```

```
output :
```

```
static block
```

```
RE: NoSuchMethodError:main
```



1.7 version:

```
javac Test.java
```

```
java Test
```

Error: main method not found in class Test, please define the main method as
public static void main(String[] args)

Case 3:

```
1) class Test {  
2) static {  
3) System.out.println("static block");  
4) System.exit(0);  
5) }  
6) }
```

1.6 version:

```
javac Test.java
```

```
java Test
```

output :

static block

1.7 version:

```
javac Test.java
```

```
java Test
```

Error: main method not found in class Test, please define the main method as
public static void main(String[] args)

Case 4:

```
1) class Test {  
2) static {  
3) System.out.println("static block");  
4) }  
5) public static void main(String[] args) {  
6) System.out.println("main method");  
7) }  
8) }
```

1.6 version:

```
javac Test.java
```

```
java Test
```

output :

static block

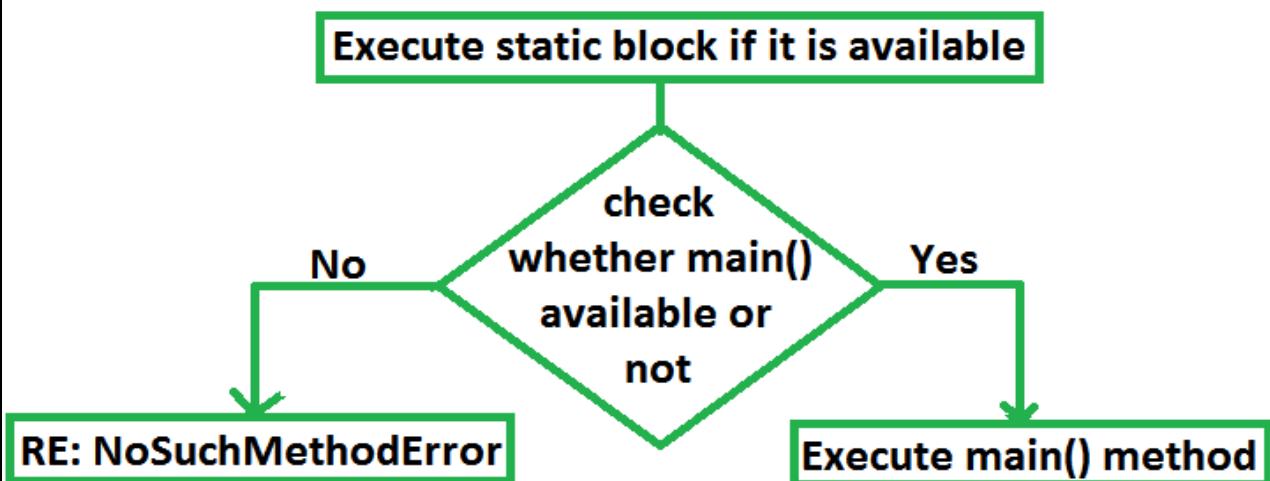
main method



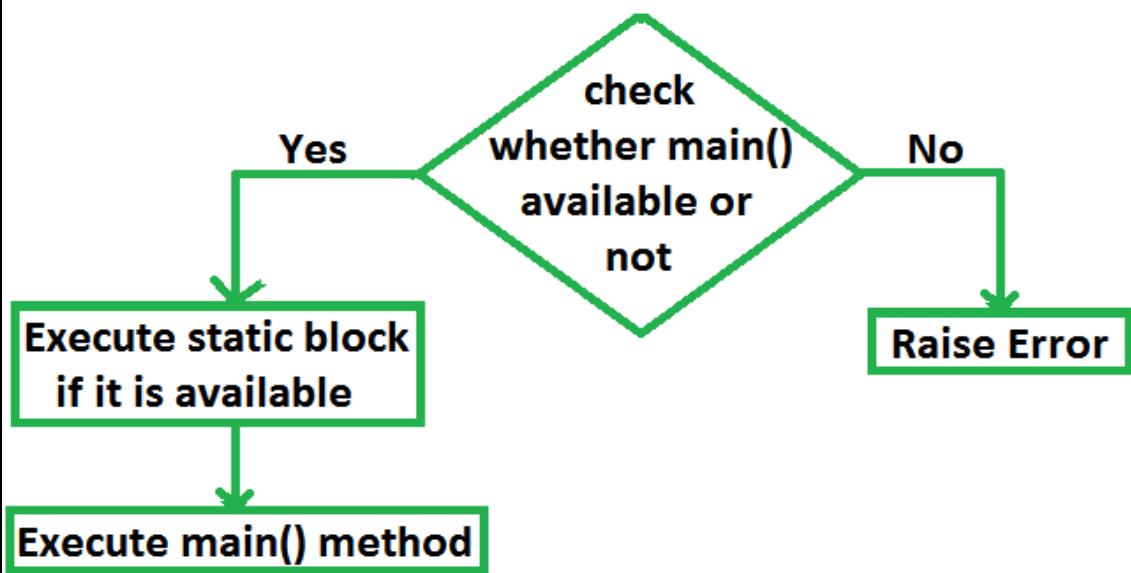
1.7 version:

```
javac Test.java  
java Test  
output :  
static block  
main method
```

1.6 version :



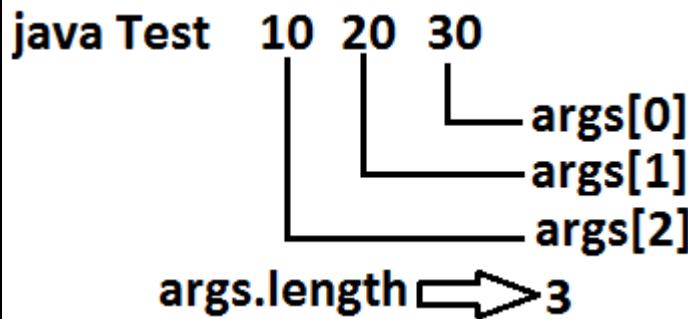
1.7 version :





Command Line Arguments

The arguments which are passing from command prompt are called command line arguments. The main objective of command line arguments are we can customize the behavior of the main() method.



Example 1:

```
1) class Test
2) {
3)   public static void main(String[] args)
4)   {
5)     for(int i=0;i<=args.length;i++)
6)     {
7)       System.out.println(args[i]);
8)     }
9)   }
10) }
```

Output:

java Test x y z
ArrayIndexOutOfBoundsException: 3
Replace $i \leq \text{args.length}$ with $i < \text{args.length}$ then it will run successfully.

Example 2 :

```
1) class Test
2) {
3)   public static void main(String[] args)
4)   {
5)     String[] argh={"X","Y","Z"};
6)     args=argh;
7)     for(String s : args)
8)     {
```



```
9)     System.out.println(s);
10)    }
11)   }
12) }
```

Output:

java Test A B C

X
Y
Z

java Test A B

X
Y
Z

java Test

X
Y
Z

Within the main() method command line arguments are available in the form of String hence "+" operator acts as string concatenation but not arithmetic addition.

Example 3 :

```
1) class Test
2) {
3)   public static void main(String[] args)
4)   {
5)     System.out.println(args[0]+args[1]);
6)   }
7) }
```

Output:

E:\SCJP>javac Test.java

E:\SCJP>java Test 10 20

1020

Space is the separator between 2 command line arguments and if our command line argument itself contains space then we should enclose with in double quotes.

Example 4 :

```
1) class Test
2) {
3)   public static void main(String[] args)
4)   {
5)     System.out.println(args[0]);
```



6) }
7) }

Output:

E:\SCJP>javac Test.java
E:\SCJP>java Test "Sai Charan"
Sai Charan

Q. Which one of the following code examples uses valid java syntax?

A)

1) **public class Bunny**
2) {
3) **public static void main(String[] args)**
4) {
5) **System.out.println("Bunny");**
6) }
7) }

B)

1) **public class Chinny**
2) {
3) **public static void main(String[])**
4) {
5) **System.out.println("Chinny");**
6) }
7) }

C)

1) **public class Sunny**
2) {
3) **public void main(String[] args)**
4) {
5) **System.out.println("Sunny");**
6) }
7) }

D)

1) **public class Vinny**
2) {
3) **public static void main(String() args)**
4) {
5) **System.out.println("Vinny");**
6) }



7) }

Answer: A

Q. Given the code from the Test.java file:

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         System.out.println("Hello "+args[0]);
6)     }
7) }
```

Which set of commands prints Hello Durga in the console?

- A) javac Test
java Test Durga
- B) javac Test.java Durga
java Test
- C) javac Test.java
java Test Durga
- D) javac Test.java
java Test.class Durga

Answer: C

Q. Consider the code Test.java:

```
1) public class Test
2) {
3)     public static void main(int[] args)
4)     {
5)         System.out.println("int[] main: "+args[0]);
6)     }
7)     public static void main(Object[] args)
8)     {
9)         System.out.println("Object[] main: "+args[0]);
10}
11    public static void main(String[] args)
12    {
13        System.out.println("String[] main: "+args[0]);
14    }
15}
```



and the commands

`javac Test.java`
`java Test 1 2 3`

What is the result?

- A) `int[] main 1`
- B) `Object[] main 1`
- C) `String[] main 1`
- D) **Compilation Fails**
- E) `An Exception raises at runtime`

Answer: C

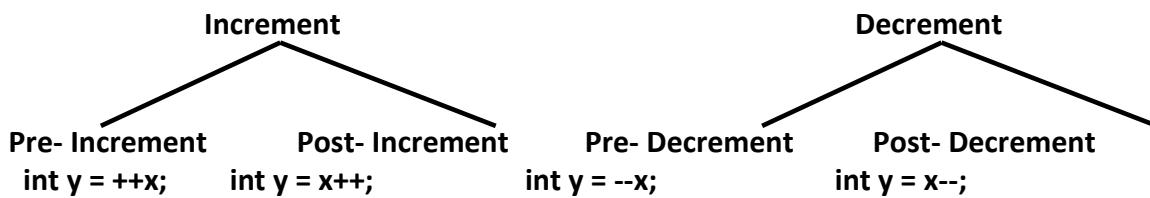


Operators & Assignments

- 2.1) Increment and Decrement Operators
- 2.2) Arithmetic Operators
- 2.3) String Concatenation Operator
- 2.4) Relational Operators
- 2.5) Equality Operators
- 2.6) Bitwise Operators
 - Bitwise Compliment Operator (~)
 - Boolean Complement Operator (!)
- 2.7) Short Circuit Operators
- 2.8) instanceof Operator
- 2.9) Type Cast Operator
- 2.10) Assignment Operators
 - Simple Assignment
 - Chained Assignment
 - Compound Assignment
- 2.11) Conditional Operator
- 2.12) new Operator
- 2.13) [] Operator
- 2.14) Operator Precedence
- 2.15) Evaluation Order of Operands
- 2.16) new Vs newInstance()
- 2.17) instanceof Vs isInstance()
- 2.18) ClassNotFoundException Vs NoClassDefFoundError



Increment and Decrement Operators



Expression	Initial Value of x	Value of y	Final Value of x
y = ++x;	10	11	11
y = x++;	10	10	11
y = --x;	10	9	9
y = x--;	10	10	9

We can Apply Increment and Decrement Operators only for Variables but Not for Constant Values Otherwise we will get CE.

```
int x = 10;  
int y = ++x;  
System.out.println(y); //11
```

```
int x = 10;  
int y = ++10;  
System.out.println(y);
```

CE: unexpected type
required: variable
found: value

Nesting of Increment and Decrement Operators is Not allowed Otherwise we will get Compile Time Error.

```
int x = 10;  
int y = ++(++x);  
System.out.println(y);
```

CE: unexpected type
required: variable
found: value



We can't Apply Increment and Decrement Operators for final Variables.

```
final int x = 10;  
x++;  
System.out.println(x);
```

CE: cannot assign a value to final variable x e

```
final int x = 10;  
x++;  
System.out.println(x);
```

We can Apply Increment and Decrement Operators for every Primitive Data Type except boolean.

```
int x = 10;  
x++;  
System.out.println(x); //11
```

```
double d = 10.5;  
d++;  
System.out.println(d); //11.5
```

```
char ch = 'a';  
ch++;  
System.out.println(ch); //b
```

```
boolean b = true;  
b++; //bad operand type boolean for unary operator '++'  
System.out.println(b);
```

Differences between b++ and b = b+1:

If we Apply any Arithmetic Operations between 2 Variables a and b the Result Type is Always *Max(int, Type of a, Type of b)*.

```
byte b1 = 10;  
byte b2 = 20;  
byte b3 = b1 + b2;  
System.out.println(b3);
```

CE: possible loss of precision
required: byte
found: int

```
byte b1 = 10;  
byte b2 = 20;  
byte b3 = (byte)(b1 + b2);  
System.out.println(b3); //30
```

Max(int, Type of a, Type of b)
Max(int, byte, byte)
int

If we Perform Explicit Type Casting then we won't get any CE.

```
byte b = 10;  
b++;  
System.out.println(b); //11
```

```
byte b = 10;  
b = (byte)(b+1);  
System.out.println(b); //11
```

```
byte b = 10;  
b = b+1;  
System.out.println(b);
```

CE: possible loss of precision
required: byte
found: int



In the Case of Increment and Decrement Operators Internal Type Casting Automatically performed by the Compiler.

```
byte b = 10;  
b++; ➔ b = (Type of b)(b++);  
System.out.println(b); //11
```

Arithmetic Operators

The Arithmetic Operators are (+, -, *, /, %).

If we are applying any Arithmetic Operation between 2 Variables a and b the Result Type is Always *Max(int, Type of a, Type of b)*.

- | | |
|------------------------|--|
| 1) byte + byte = int | 6) long + float = float |
| 2) byte + short = int | 7) byte + double = double |
| 3) short + short = int | 8) char + char = int ➔ System.out.println('a' + 'b'); //195 |
| 4) int + int = int | 9) char + int = int ➔ System.out.println('a' + 1); //98 |
| 5) int + long = long | 10) char + double = double ➔ System.out.println('a' + 1.1); //98.1 |

Infinity:

- In Integral Arithmetic (int, short, long, byte), there is No way to Represent infinity. Hence, if the Infinity is the Result we will Always get RE: ArithmeticException: / by zero) in Integral Arithmetic.

Eg: System.out.println(10/0); //RE: java.lang.ArithmaticException: / by zero

- But in floating Point Arithmetic (float & double), there is a Way to Represent Infinity.
- For this float & double Classes contains the following 2 Constants.
 - POSITIVE_INFINITY;
 - NEGATIVE_INFINITY;
- Hence Even though the Result is Infinity we won't get any Arithmetic Exception in floating Point Arithmetic.

```
System.out.println(10/0.0); //RE: Infinity  
System.out.println(-10/0.0); //RE: -Infinity
```



NaN: (Not a Number)

- In Integral Arithmetic there is No Way to Represent Undefined results. Hence, if the Result is Undefined we will get RE: ArithmeticException: / by zero

Eg: `System.out.println(0/0); //RE: java.lang.ArithmaticException: / by zero`

- But in floating Point Arithmetic there is a way to Represent Undefined Results, for this Float & Double Classes a Constant NaN. Hence, Even though the Result is Undefined we won't get any Runtime Exception in floating Point Arithmetic.

```
System.out.println(0/0.0); //NaN  
System.out.println(0.0/0); //NaN  
System.out.println(-0/0.0); //NaN
```

- For any x Value including NaN the Below Expressions Always Returns false.

```
System.out.println(10 > Float.NaN);  
System.out.println(10 >= Float.NaN);  
System.out.println(10 < Float.NaN);  
System.out.println(10 <= Float.NaN);  
System.out.println(10 == Float.NaN);  
System.out.println(Float.NaN == Float.NaN);
```

f
a
l
s
e

x > Nan
x >= Nan
x < Nan
x <= Nan
x == Nan

f
a
l
s
e

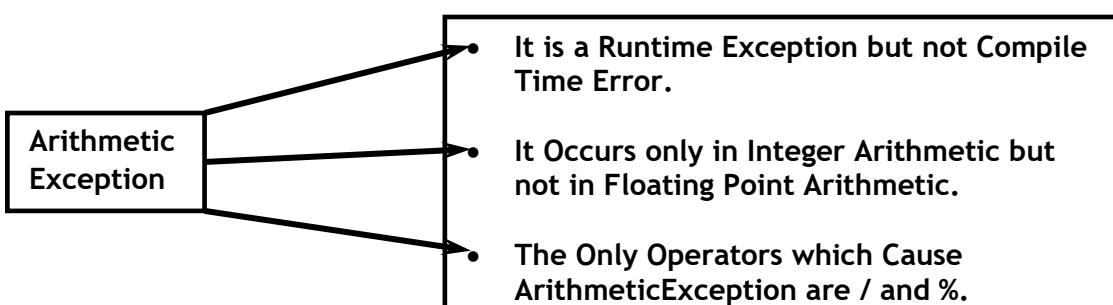
- For any x Value including NaN the Below Expressions Always Returns true.

```
System.out.println(10 != Float.NaN);  
System.out.println(Float.NaN != Float.NaN);
```

x != Nan True

True

Conclusion About Arithmetic Exception:





String Concatenation Operator (+)

- The Only Overloaded Operator in Java is ‘+’ Operator.
- Sometimes it Acts as Arithmetic Addition and Sometimes Acts as String Concatenation Operator.
- If at Least One Operand is String Type then ‘+’ Operator Acts as Concatenation, Whereas if Both Operands are Number Type then ‘+’ Operator Acts as Arithmetic Addition Operator.

```
int a = 10, b = 20, c = 30;  
String d = "Durga";  
  
System.out.print(a+b+c+d); //60Durga  
System.out.print(a+b+d+c); //30Durga30  
System.out.print(d+a+b+c); //Durga102030  
System.out.print(a+d+b+c); //10Durga2030
```

Evolution

a+b+c+d
"Durga10"+c+d
"Durga1020"+d
Durga102030

Here System.out.print() is evaluated from Left to Right.

Consider the following Declarations

```
String a = "Durga";  
int b = 10, c = 20, d = 30;
```

Which of the following Expressions are Valid?

1) a = b+c+d;

CE: incompatible types
required: String
found: int

2) b = a+c+d;

CE: incompatible types
required: String
found: int

3) a = a+b+c;

System.out.println(a); //Durga1020

4) b = b+c+d;

System.out.println(a); //Durga



Relational Operators

- The Relational Operators are (`>`, `>=`, `<`, `<=`).
- We can Apply Relational Operators to Every Primitive Data Type except boolean.

<code>System.out.println(10 > 20);</code>	false
<code>System.out.println('a' < 'b');</code>	true
<code>System.out.println(10 >= 10.0);</code>	true
<code>System.out.println('a' < 125);</code>	true
<code>System.out.println(true <= true);</code>	CE: bad operand types for binary operator '<=' first type: boolean second type: boolean
<code>System.out.println(true < false);</code>	CE: bad operand types for binary operator '<' first type: boolean second type: boolean

- We can't Apply Relational Operators for Object Types. Otherwise we will get CE.

`System.out.println("Durga" < "Durga");`
CE: bad operand types for binary operator '<'
first type: String
second type: String

`System.out.println("Durga" < "Durga123");`
CE: bad operand types for binary operator '<'
first type: String
second type: String

- Nesting of Relational Operators are Not allowed.

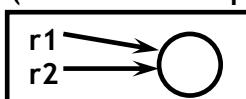
`System.out.println(10 < 20 < 30);`
CE: bad operand types for binary operator '<'
first type: boolean
second type: int

Equality Operators (==, !=)

- We can Apply Equality Operators for Every Primitive Type including boolean Also.

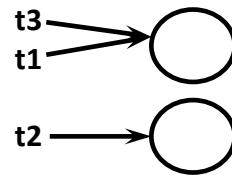
<code>System.out.println(10 == 20);</code>	false
<code>System.out.println('a' == 97.0);</code>	true
<code>System.out.println(10.5 == 10);</code>	false
<code>System.out.println(false == false);</code>	true

- We can Apply Equality Operators for Object Types Also.
- For Object References `r1` and `r2`, `r1 == r2` Returns true if and only if Both are pointing to the Same Object (Reference Comparison OR Address Comparison).





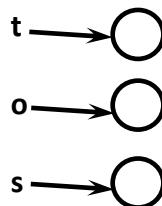
```
Thread t1 = new Thread();
Thread t2 = new Thread();
Thread t3 = t1;
System.out.println(t1 == t2) //false
System.out.println(t1 == t3) //true
```



- To Apply Equality Operators between Object Types Compulsory there should be Some Relation between Argument Types (Either Child to Parent OR Parent to Child OR Same Type) Otherwise we will get Compile Time Error Saying incomparable types.

```
Object o = new Object();
String s = new String("Durga");
Thread t = new Thread();

System.out.println(o == s); //false
System.out.println(t == o); //false
System.out.println(s == t); //CE: incomparable types: String and Thread
```



- For any Object Reference r, `r == null` is Always null. But `null == null` is Always true.

```
String s = new String("Durga");
System.out.println(s == null); //false
```

```
String s = null;
System.out.println(s == null); //true
System.out.println(null == null); //true
```

Difference between == Operator and .equals():

`==` Operator Meant for Reference Comparison (Address Comparison) whereas `.equals()` Meant for Content Comparison.

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

System.out.println(s1 == s2); //false
System.out.println(s1.equals(s2)); //true
```



Bitwise Operators

& → AND	If Both Arguments are true then Only Result is true.
 → OR	If at Least One Argument is true then Result is true.
^ → X-OR	If Both Arguments are Different then Only Result is true.

```
System.out.println(true & false); //false  
System.out.println(true | false); //true  
System.out.println(true ^ false); //true
```

- We can Apply these Operators even for Integral Types Also.

```
System.out.println(4 & 5); //4  
System.out.println(4 | 5); //5  
System.out.println(4 ^ 5); //1
```

Bitwise Compliment Operator (~)

- We can Apply this Operators Only for Integral Data Types but Not for boolean Types.

```
System.out.println(~true); //CE: operator ~ cannot be applied to boolean
```

System.out.println(~4); // -5

4 = 0 0 0 . . . 0 1 0 0

-4 = 1 1 . . . 1 0 1 1

-ve → 0 0 . . . 0 1 0 0

0 0 . . . 0 1 0 1 → 5

- The Most Significant Bit Acts as Sign Bit.
- 0 Means Positive Number and 1 Means Negative Number
- Positive Numbers will be Represented Directly in the Memory where as Negative Numbers will be Represented in 2's Compliment Form.



boolean Compliment Operator (!):

This Operator Applicable Only for boolean Types but Not for Integral Types.

```
System.out.println(!4); //CE: bad operand type int for unary operator '!'
```

```
System.out.println(!true); //false
```

- ❖ &, |, and ^ are Applicable for Both boolean and Integral Types.
- ❖ ~ Applicable Only for Integral Types but Not for boolean Types.
- ❖ ! Applicable Only for boolean type but Not for Integral Types.

Short Circuit Operators (&&, ||)

These are Exactly Same as Bitwise Operators (&, |), Except the following Differences.

&,	&&,
Both Arguments should be evaluated Always.	2 nd Argument Evaluation is Optional.
Relatively Performance is Low.	Relatively Performance is High
Applicable for Both boolean and Integral Types.	Applicable Only for boolean but Not for Integral Types.

Note:

- x && y → y will be evaluated if and Only if x is true i.e. if x is false then y won't be evaluated.
- x || y → y will be evaluated if and if x is false i.e. if x is true then y won't be evaluated.

```
int x = 10, y = 15;
if(++x < 10 & ++y > 15) {
    ++x;
}
else {
    ++y;
}
System.out.println(x+"....."+y);
```

	x	y
&	11	17
	12	16
&&	11	16
	12	16



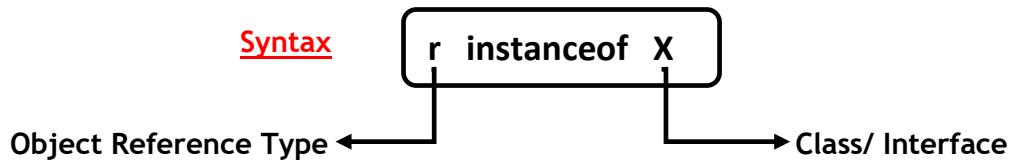
```
int x = 10;
if((++x < 10) && (x/0 >10)) {
    System.out.println("Hello");
}
else {
    System.out.println("Hi");
}
```

1) RE 2) CE 3) Hello 4) Hi ✓

Note: If we Replace && with & then we will get
RE: java.lang.ArithmaticException: / by zero

instanceof Operator

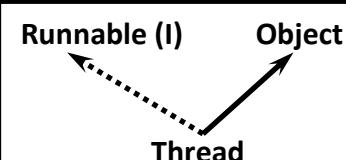
We can Use instanceof Operator to Check whether the given Object is the Particular Type OR Not.



```
Object o = l.get(0);
if(o instanceof Student) {
    Student s = (Student)o;
    //Perform Student Specific Functionality
}
else if(o instanceof Customer) {
    Customer c = (Customer)o;
    //Perform Customer Specific Functionality
}
```

It Returns 1st Element of ArrayList and it can be any Type.

```
Thread t = new Thread();
System.out.println(t instanceof Thread); //true
System.out.println(t instanceof Object); //true
System.out.println(t instanceof Runnable); //true
```



- To Use instanceof Operator Compulsory there should be Some Relation between Argument Types. Otherwise we will get Compile Time Error Saying *inconvertible types*.
- The Relation can be Either Child to Parent OR Parent to Child OR Same Type.



```
Thread t = new Thread();
System.out.println(t instanceof String);
CE: inconvertible types
 required: String
 found: Thread
```

- Whenever we are checking Parent Object is of Child Type OR Not by using instanceof Operator then we will get false as Output.

```
Object o = new Object();
System.out.println(o instanceof Thread); //false
```

```
Object o = new Thread();
System.out.println(o instanceof Thread); //true
```

```
Object o = new String("Durga");
System.out.println(o instanceof String); //true
```

- For any Class OR Interface as *null interface X* is Always false.

```
System.out.println(null instanceof String); //false
System.out.println(null instanceof Object); //false
System.out.println(null instanceof Runnable); //false
```

Type Cast Operator

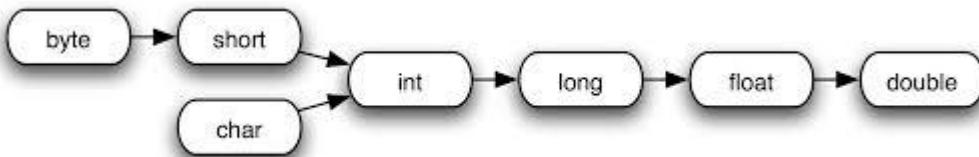
There are 2 Types of Type Casting

- 1) Implicit Type Casting
- 2) Explicit Type Casting

Implicit Type Casting:

- Compiler is Responsible to Perform Implicit Type Casting.
- Whenever we are assigning Lower Data Type Value to Higher Data Type Variable
- Implicit Type Casting will be Performed.
- It is also Known as *Widening OR Up Casting*.
- There is No Loss of Information in this Type Casting.

The following are Various Possible Conversions where Implicit Type Casting will be Performed.



Examples:

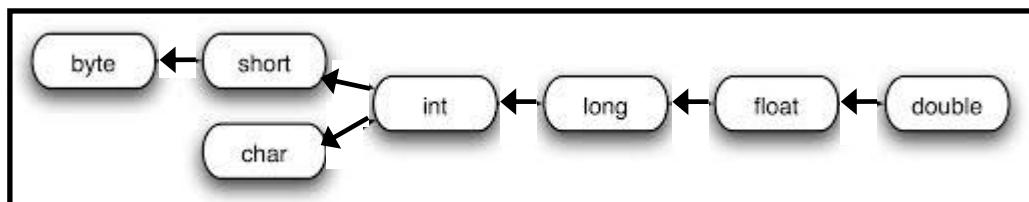
1) `int x = 'a';` [Compiler Converts char → int Automatically by Implicit Type Casting]
`System.out.println(x); //97`

2) `double d =10;` [Compiler Converts int → double Automatically by Implicit Type Casting]
`System.out.println(d); //10.0`

Explicit Type Casting:

- Programmer is Responsible to Perform Explicit Type Casting.
- Whenever we are assigning Higher Data Type Value to Lower Data Type Variable then Explicit Type Casting is Required.
- It is Also Known as *Narrowing OR Down Casting*.
- There May be Chance of Loss of Information in this Type Casting.

The following are Various Possible Conversions where Explicit Type Casting will be Performed.



Note:

- Left → Right → Implicit Type Casting
- Right → Left → Explicit Type Casting

Examples:

```
int x = 130;
byte b = x;
CE: possible loss of precision
    required: byte
    found:   int

byte b = (byte) x;
System.out.println(b); //-126
```

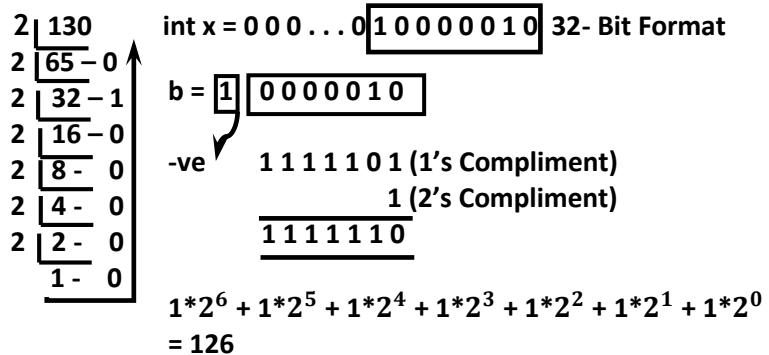


Explanation:

int x = 130; \rightarrow 000.....010000010

int Range: 4 Bytes (32 Bits)

byte b = (byte)x; \rightarrow 10000010



- Whenever we are assigning Higher Data Type Value to Lower Data Type Variable by Explicit Type Casting then the Most Significant Bits will be Lost. We have to Consider Only Least Significant Bits.

Example:

```
int x = 150;  $\rightarrow$  000.....0110010110
short s = (short)x;  $\rightarrow$  00..10010110
System.out.println(s); //150
```

```
byte b = (byte)x;  $\rightarrow$  10010110
System.out.println(s); // -106
```

- Whenever we are assigning floating Point Data Type Values to the Integral Data Types by Explicit Type Casting then the Digits after the Decimal Point will be Lost.

```
double d = 130.456;
int x = (int)d;
System.out.println(x); //130
byte b = (byte)d;
System.out.println(b); // -126
```



Assignment Operators

There are 3 Types of Assignment Operators.

- Simple Assignment
- Chained Assignment
- Compound Assignment

Simple Assignment: `int x = 10;`

Chained Assignment:

```
int a,b,c,d;
a = b = c = d = 20;
System.out.println(a+".."+"b+".."+"c+".."+"d); //20..20..20..20
```

We can't Perform Chained Assignment Directly at the Time of Declaration.

```
int a = b = c = d = 20;
#javac Test.java
Test.java:3: error: cannot find symbol
    int a = b = c = d = 20;
               ^
  symbol:  variable b
  location: class Test
Test.java:3: error: cannot find symbol
    int a = b = c = d = 20;
               ^
  symbol:  variable c
  location: class Test
Test.java:3: error: cannot find symbol
    int a = b = c = d = 20;
               ^
  symbol:  variable d
  location: class Test
3 errors
```

```
int b,c,d;
int a = b = c = d = 20; ✓
System.out.println(b+".."+"c+".."+"d); //20..20..20
```

Compound Assignment:

- Some Time we can Mix Assignment Operator with Some Other Operator to form
- Compound Assignment Operator.

```
int x = 10;
x += 20;
System.out.println(x); //30
```

- The following with the List of All Possible Compound Assignment Operators in Java.



<code>+=</code>	<code>>>=</code>	<code>&=</code>
<code>-=</code>	<code>>>>=</code>	<code> =</code>
<code>*=</code>	<code><<=</code>	<code>^=</code>
<code>/=</code>		
<code>%=</code>		

- In the Component aAssinment Implicit Type Casting will be performed Automatically by the Compiler.

```
byte b = 10;  
b = b+1;  
System.out.println(b);
```

CE: possible loss of precision
required: byte
found: int

```
byte b = 10;  
b++;  
System.out.println(b); //11
```

```
byte b = 10;  
b +=1;  
System.out.println(b); //11
```

$b = (byte)(b+1);$

```
byte b = 127;  
b += 3;  
System.out.println(b); -126
```

Mixing with Chained and Compound Assignemts

```
int a, b, c, d;  
a = b = c = d = 20;  
a += b -= c *= d /= 2;  
System.out.println(a +"...."+ b +"...."+ c +"...."+ d); // -160....-180....200....10
```

Conditional Operator (?:)

- The Only Possible Ternary Operator in Java is Conditional Operator.

```
int x = (10 > 20) ? 30:40;  
System.out.println(x); //30
```

If Conditiona is false then Result is 40.

- Nesting of Conditional Operator is Always Possible.

```
int x = (10>20) ? 30 : ( (40>50) ? 60:70 );  
System.out.println(x); //70
```



new Operator

- We can Use new Operator to Create Objects in Java.
- But there is No delete Operator in Java because Garbage Collector is Responsible to Destroy Useless Objects.

[] Operator

We can Use this Operator to Declare and Create Arrays.

Operators Presidency

- 1) **Unary Operators:**
 - [], x++, x--
 - ++x, --x, ~, !
 - new, <type>
- 2) **Arithmetic Operators:** *, /, %, +, -
- 3) **Shift Operators:** >>, >>>, <<
- 4) **Comparison Operators:** <, <=, >, >=, instanceof
- 5) **Equality Operators:** ==, !=
- 6) **Bitwise Operators:** &, ^ , |
- 7) **Short Circuit Operators:** &&, ||
- 8) **Conditional Operators:** ?:
- 9) **Assignment Operators:** =, +=, -=, *=.....

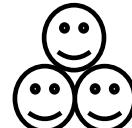


Evaluation Order of Operands

- There is No Operand Precedence and we have Only Operator Precedence.
- Before applying any Operator all Operands will be evaluated from Left to Right.

```
class Test {  
    public static void main(String[] args) {  
        System.out.println( m1(1)+m1(2)*m1(3) / m1(4)*m1(5)+m1(6));  
    }  
    public static int m1(int i) {  
        System.out.println(i);  
        return i;  
    }  
}
```

1	1+2* 3 / 4-5+6
2	1+6 / 4-5+6
3	1+1-5+6
4	2-5+6
5	-3+6
6	3
12	



new Vs newInstance()

We can Use new Operator to Create Objects if we Know the Class Name at the Begining.

- 1) Test t = new Test();
- 2) Student s = new Student();
- 3) Customer c = new Customer();
- 4) String s = new String();

newInstance() is a Method Present in 'Class' class which can be used to Create Object if we don't know the Class Name at the beginning and it is Available Dynamically at Runtime.

```
class Test {  
    public static void main(String[] args) throws Exception {  
        Object o = Class.forName(args[0]).newInstance();  
        System.out.println("Class Name: "+o.getClass().getName());  
    }  
}  
  
>java Test java.lang.String  
Class Name: java.lang.String  
  
>java Test Test  
Class Name: Test
```



new Operator	newInstance() method
<p>new is a Key Word which can be used to Create an Object if we Know the Class Name at the beginning.</p> <p>Eg: Test t = new Test();</p>	<p>newInstance() is a Method which can be used to Create an Object if we don't Know the Class Name at the beginning and it is Available Dynamically at Runtime.</p> <p>Eg: Object o = Class.forName(args[0]).newInstance();</p>
<p>To Use new Operator Class Not required to contain No Argument Constructor. Based on Our Requirement we can Invoke any Constructor.</p>	<p>To Use newInstance() Method Compulsory Class should contain No- Argument Constructor. It may be explicitly provided by the JVM OR Default Constructor generated by the Compiler. Otherwise we will get Runtime Exception Saying java.lang.InstantiationException.</p>
<p>At Runtime if the Corresponding .class File Not Available then we will get Runtime Exception Saying NoClassDefFoundError. Which is Unchecked Exception</p>	<p>At Runtime if the Corresponding .class File is Not Available then we will get Runtime Exception Saying ClassNotFoundException. Which is Checked Exception.</p>

NoClassDefFoundError Vs ClassNotFoundException

NoClassDefFoundError:

- If Hard Coded Class Name is Not Available at Runtime then we will get Runtime Exception Saying NoClassDefFoundError. Which Unchecked Exception.
- Eg: Test t = new Test();
- At Runtime if Test.class is Not Available then we will get Runtime Exception Saying NoClassDefFoundError: Test
- NoClassDefFoundError is an Unchecked Exception and Hence Compiler won't Check whether we are handling OR Not.

ClassNotFoundException:

- If Dynamically provided Class Name is Not Available at Runtime then we will get Runtime Exception Saying ClassNotFoundException. Which is Checked Exception.
- Eg: Object o = Class.forName(args[0]).newInstance();
➤ java Test Test123
- If Test123.class File is Not Available at Runtime then we will get Runtime Exception Saying ClassNotFoundException: Test123.
- ClassNotFoundException is an Checked Exception and Hence Compiler will Check whether we are handling OR Not either by try- catch OR by throws Key Word.



instanceof Vs isInstance()

instanceof

- We can Use instanceof Operator to Check whether the given Object is the Particular Type OR Not and the Type is specified at the beginning.

Eg: `Thread t = new Thread();
System.out.println(t instanceof Runnable); true`

isInstance():

- isInstance() is a Method Present in class Class which can be used to Check whether the given Object is Particular Type OR Not and we don't Know the Type at the beginning and it is Available Dynamically at Runtime.

Eg: `Thread t = new Thread();
System.out.println(Class.forName(args[0]).isInstance(t));`

➤ java Test java.lang.Runnable true
➤ java Test java.lang.String false

Note:

- ★ newInstance() is Equivalent of new Operator.
- ★ isInstance() is Equivalent of instanceof Operator.



Flow Control

Flow Control

Selection Statements

- if - else
- switch

Iterative Statements

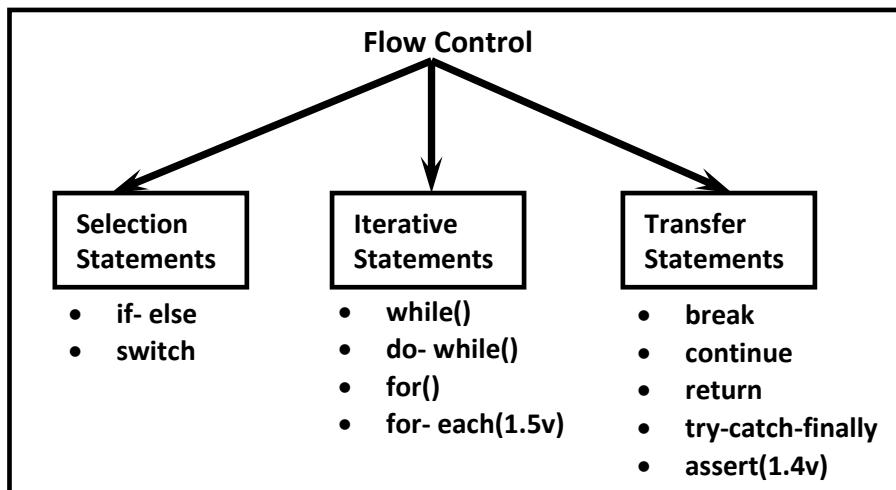
- while()
- do - while()
- for()
- for - each Loop (1.5)

Transfer Statements

- break
- continue
- return
- try - catch - finally
- assert (1.4)



Flow Control describes the Order in which the Statements will be executed at Runtime.



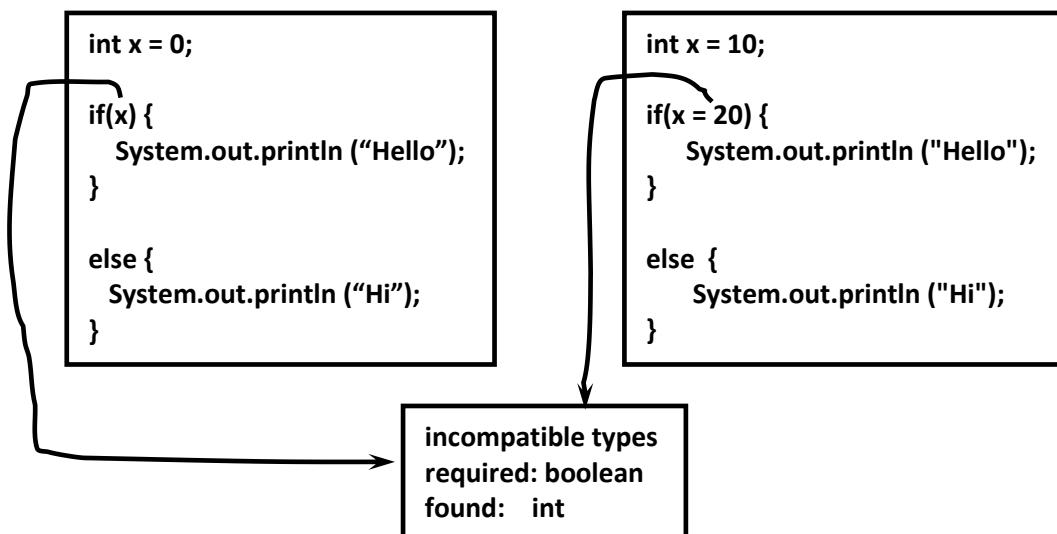
Selection Statements

if - else:

Syntax:

```
if(b) {  
    Action if b is true  
}  
else {  
    Action if b is false  
}
```

- The Argument to the if Statement should be boolean Type.
- By Mistake if we are providing any Other Type we will get Compile Time Error.





<pre>int x = 10; if (x == 20) { System.out.println ("Hello"); } else { System.out.println ("Hai"); }</pre> <p><u>Output:</u> Hai</p>	<pre>boolean b = true; if (b = false) { System.out.println ("Hello"); } else { System.out.println ("Hi"); }</pre> <p><u>Output:</u> Hai</p>	<pre>boolean b = false; if (b == false) { System.out.println ("Hello"); } else { System.out.println ("Hi"); }</pre> <p><u>Output:</u> Hello</p>
--	---	---

Both else Part and Curly Braces are Optional.

Without Curly Braces we can Take Only One Statement Under if, which should Not be Declarative Statement.

if (true)
SOP ("Hello"); ✓

if (true); ✓

if (true)
int x = 10; ✗

if (true) {
int x = 10;
} ✓

Note: Semi Colon (;) is a Valid Java Statement which is Also Known as Empty Statement.

switch Statement

- If Several Options are Available, then it is Not Recommended to Take Nested if-else. Because it Reduces Readability of the Code.
- In this Case we should go for switch Statement.
- The Advantage of switch Statement is Readability will be Improved.

Syntax:

```
switch(x)
{
    case 1:
        Action 1;
        break;

    case 2:
        Action 1;
        break;
        .
        .
        .

    case n:
        Action 1;
        break;

    default:
        default Action;
}
```



- Until 1.4 Version The allowed Argument Types for the switch Statement are byte, short, char and int.
- From 1.5 Version onwards the Corresponding Wrapper Classes and enum Type Also allowed.
- From 1.7 Version onwards even String Type Also allowed.

1.4	1.5	1.7
byte	Byte	
short	Short	String
char	Character	
int	Integer + enum	

- Curly Braces are Mandatory (switch is the Only Place where Curly Braces are Mandatory. Except this Every Where Curly Braces are Optional).
- Inside switch Both *case* and *default* are Optional.

```
switch (x)
{
}
```

- Within the switch Every Statement should be Under Some *case* OR *default*. That is Independent Statements are Not allowed within the switch.

```
int x = 10;
switch(x) {
    System.out.println("Hai"); //CE: case, default, or '}' expected
}
```

- Every case Label should be Constant Expression Otherwise we will get Compile Time Error.

```
class Test {
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        switch(x) {
            case 10:
                System.out.println(10);
                break;
            case y:
                System.out.println(20);
        }
    } CE: constant expression required
}
```



Note: If we Declare y as final then we won't get any Compile Time Error.
final int y = 20;

- For Both switch Arguments and case Label we can Take Expressions. But case Label should be Constant Expression.

```
int x = 10;
switch(x + 1) {
    case 10:
        System.out.println(10);
        break;
    case 10 + 20:
        System.out.println(20);
}
```

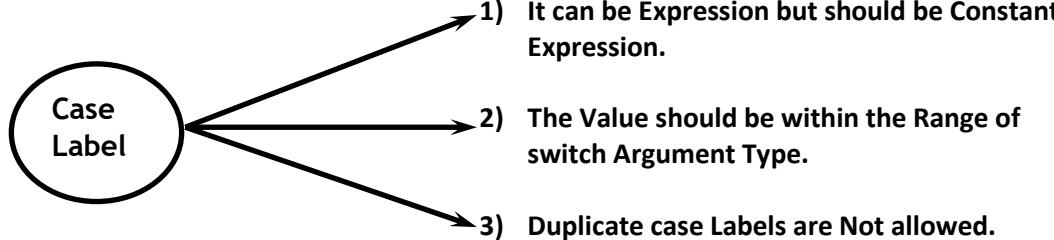
- Every case Label should be within the Range of switch Argument Type.

```
byte b = 10;
switch(b) {
    case 10:
        SOP(10);
        break;
    case 100:
        SO(100);
        break;
    case 1000:
        SOP(1000);
}
CE: possible loss of precision
required: byte
found: int
```

```
byte b = 10;
switch(b + 1) int
{
    case 10:
        SOP(10);
        break;
    case 100:
        SO(100);
        break;
    case 1000:
        SOP(1000);
}
```

- Duplicate case Labels are Not allowed.

```
switch(x) {
    case 97:
        System.out.println(97);
        break;
    case 98:
        System.out.println(98);
        break;
    case 'a': //CE: duplicate case label
        System.out.println('a');
        break;
}
```



Fall-through inside switch:

- Within the switch if any *case* OR *default* Matched, from that case onwards all Statements will be executed until *break* OR End of the switch. This is called fall-through Inside switch.
- The Main Advantage of Fall-through Inside switch is we can define Common Action for Multiple Cases. (Code- Reusability)

```
switch(x) {  
    case 0:  
        System.out.println("0");  
    case 1:  
        System.out.println("1");  
        break;  
    case 2:  
        System.out.println("2");  
    default:  
        System.out.println("default");  
}
```

<u>x = 0</u>	<u>x = 1</u>
0	1
1	
<u>x = 2</u>	<u>x = 3</u>
2	
default	default

```
switch(x) {  
    case 1:  
    case 2:  
    case 3:  
        System.out.println("Q-4");  
        break;  
    case 4:  
    case 5:  
    case 6:  
        System.out.println("Q-5");  
        break;  
    .  
    .  
}
```



default case:

- Within the switch we can take default case at-most Once.
- If No Other case Matched then Only default case will be executed.
- We can Take default case anywhere within the switch. But it is Convention to Take Last case.

```
switch (x) {  
    default:  
        System.out.println("default");  
    case 0:  
        System.out.println("0");  
        break;  
    case 1:  
        System.out.println("1");  
    case 2:  
        System.out.println("2");  
}
```

<u>x = 0</u>	<u>x = 1</u>
0	1
	2
<u>x = 2</u>	<u>x = 3</u>
2	default
	0

while Loop:

- If we don't Know the Number of Iteration in Advance then the Best Suitable Loop is while Loop.

Examples:

```
while (rs.next())  
{  
    .....  
    .....  
}
```

```
while (itr.hasNext())  
{  
    .....  
    .....  
}
```

```
while (e.hasMoreElements())  
{  
    .....  
    .....  
}
```

Syntax:

```
while (b)  
{  
    Action  
}
```

- The Argument to the while should be boolean Type. By Mistake if we are Provide any Other Type we will get Compile Time Error.

```
while(1){  
    System.out.println("Hello");  
}  
CE: incompatible types  
    required: boolean  
    found: int
```



- Curly Braces are Optional and without Curly Braces we can Take Only One Statement Under while. Which should Not be Declarative Statement.

```
while(true)  
    System.out.println("Hello");
```

```
while(true);  
; is Valid Java Statement
```

```
while (true)  
    int x = 10; //Declarative Statement X
```

```
while (true) {  
    int x = 10;  
} V
```

```
while(true)  
{  
    System.out.println("Hello");  
}  
System.out.println("Hi");
```

```
while(false)  
{  
    System.out.println("Hello");  
}  
System.out.println("Hi");
```

CE: unreachable statement

```
int a = 10, b = 20;  
while(a < b) {  
    System.out.println("Hello");  
}  
System.out.println("Hi");
```

Output: Hello
Hello
.....
.....

```
int a = 10, b = 20;  
while(a > b) {  
    System.out.println("Hello");  
}  
System.out.println("Hi");
```

Output: Hi

```
final int a = 10, b = 20;  
while(a < b) {  
    System.out.println("Hello");  
}  
System.out.println("Hi");  
//CE: unreachable statement
```

```
final int a = 10, b = 20;  
while(a > b)  
{ // CE: unreachable statement  
    System.out.println("Hello");  
}  
System.out.println("Hi");
```

Note:

- final Variables are Replaced by Value at Compile Time Only.
- Compiler is Responsible to Identify Unreachable Statement and JVM is Not Responsible for this.

```
final int x = 10;  
int y = 20;  
System.out.println(x); } System.out.println(10);  
System.out.println(y); } System.out.println(y);
```



- If Every Variable is final Variable then that Operation will be performed at Compile Time Only.
- If at least One Argument is Normal Variable then that Operation should be performed at Runtime Only.

```
final int a = 10, b = 20;  
SOP (a < b); → SOP (10 < 20); → SOP (true);
```

```
int a = 10, b = 20;  
SOP (a < b); → SOP (a < b);
```

```
final int a = 10;  
int b = 20;  
SOP (a < b); → SOP (10 < b); → This Operation should be performed at Runtime
```

do-while

If we want to execute Loop Body at least Once then we should go for do-while.

Syntax

```
do {  
    Body  
}while(b);
```

Here Semicolon is Mandatory and Argument should be boolean Type.

Curly Braces are Optional and without Curly Braces we can Take Only One Statement which should Not be Declarative Statement.

```
do  
    SOP("Hello");  
while(true);
```

```
do;  
while(true);
```

```
do  
while(true);  
X
```

```
do  
    int x = 10;  
while(true); X
```

```
do {  
    int x = 10;  
} while(true); ✓
```

```
do while(true);  
SOP("Hello");  
while(false)✓
```

do	while(true)	<u>Output</u>
	SOP("Hello");	Hello
	while(false); ✓	Hello
		:::::::



```
do {  
    System.out.println ("Hello");  
} while(true);  
    System.out.println("Hai");  
//CE: unreachable statement
```

```
do {  
    System.out.println("Hello");  
}while(false);  
    System.out.println("Hai");
```

Output: Hello
Hai

```
int a = 10, b = 20;  
do {  
    System.out.println("Hello");  
} while(a < b);  
    System.out.println("Hai");
```

Output: Hello
Hello
:::::::

```
int a = 10, b = 20;  
do {  
    System.out.println("Hello");  
} while(a > b);  
    System.out.println("Hai");
```

Output: Hello
Hai

```
final int a = 10, b = 20;  
do {  
    System.out.println("Hello");  
} while(a < b);  
    System.out.println("Hai");  
//CE: unreachable statement
```

```
final int a = 10, b = 20;  
do {  
    System.out.println("Hello");  
} while(a > b);  
    System.out.println("Hai");
```

Output: Hello
Hai

for loop

- The Most Commonly Used Loop in Java is for Loop.
- for Loop is the Best Choice If we Know the Number of Iterations in Advance.

```
① ② ⑤ ③ ⑧ ←  
for (Initialization _Section; Conditional _Check; Increment/Decrement _Section)  
{  
    Body ③ ⑥ ⑨  
}
```

- Curly Braces are Optional and without Curly Braces we can Take Only One Statement in the Loop, which should Not be Declarative Statement.



Initialization Section

- This Part will be executed Only Once.
- Here we can Declare and Initialize Local Variables of for Loop.
- Here we can Declare Multiple Variables but should be of the Same Type i.e. we can't Declare different Data Types Variables.

- 1) int i = 0; ✓
- 2) int i = 0, j = 0; ✓
- 3) int i = 0, String s = "Durga"; X
- 4) int i = 0, int j = 0; X

- In Initialization Section we can Take any Valid Java Statement including `System.out.println()` Statement Also.

<pre>int i = 0; for(System.out.println("Hello U R Sleeping "); i<3; i++) { System.out.println("No Boss U Only Sleeping"); }</pre>	<u>Output</u> Hello U R Sleeping No Boss U Only Sleeping No Boss U Only Sleeping No Boss U Only Sleeping
--	--

Conditional Section

- Here we can Take any Valid Java Expression but should be boolean Type.
- This Part is Optional and if we are Not writing anything, then Complier Always Place true.

<pre>int i = 0; for (System.out.println("Hello U R Sleeping"); i++) { System.out.println("U Only Sleeping"); }</pre>	<u>Output</u> Hello U R Sleeping U Only Sleeping U Only Sleeping
---	--

Increment and Decrement Section

- Here we can Take any Valid Java Statement including `System.out.println()` Statement Also.

<pre>int i = 0; for(System.out.println("Hello"); i<3; System.out.println("Hi")) { i++; }</pre>	<u>Output</u> Hello Hi Hi Hi
---	--

Note: All 3 Parts of for Loop are Independent to Each Other and Optional.



Infinite Loops

<code>for (; ;) { Body; }</code>	<code>for (; ;);</code>
--	---------------------------

```
for (int i = 0; true; i++) {  
    System.out.println("Hello");  
}  
System.out.println("Hai");  
//CE: unreachable statement
```

```
for (int i = 0; false; i++) {  
    //CE: unreachable statement  
    System.out.println("Hello");  
}  
System.out.println("Hai");
```

```
for (int i = 0; ; i++) {  
    System.out.println("Hello");  
}  
System.out.println("Hai");  
//CE: unreachable statement
```

```
int a = 10, b = 20;  
for (int i = 0; a < b; i++) {  
    System.out.println("Hello");  
}  
System.out.println("Hai");  
  
Output: Hello  
    Hello  
    :::::::
```

```
int a = 10, b = 20;  
for (int i = 0; a > b; i++) {  
    System.out.println("Hello");  
}  
System.out.println("Hai");  
  
Output: Hai
```

```
final int a = 10, b = 20;  
for (int i = 0; a < b; i++) {  
    System.out.println("Hello");  
}  
System.out.println("Hai");  
CE: unreachable statement
```

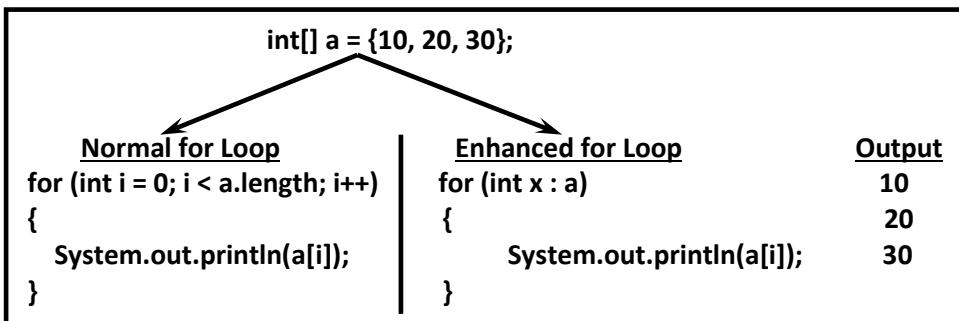
```
final int a = 10, b = 20;  
for (int i = 0; a > b; i++) {  
    // CE: unreachable statement  
    System.out.println("Hello");  
}  
System.out.println("Hai");
```



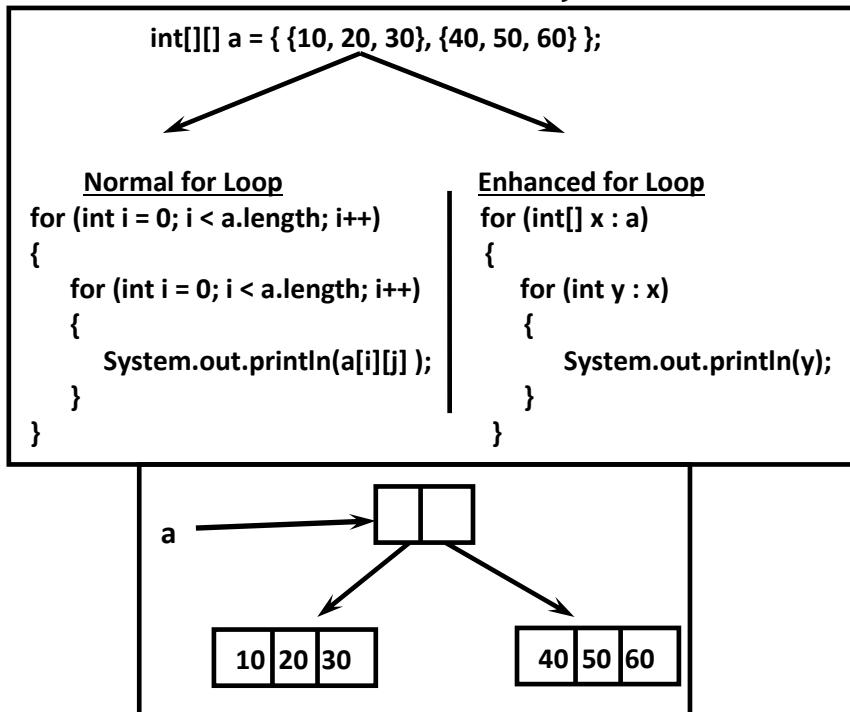
for-each Loop OR Enhanced for Loop (1.5)

It is the Specially designed Loop to retrieve Elements of Arrays and Collections.

Eg1: To Print Elements of One Dimensional int[]



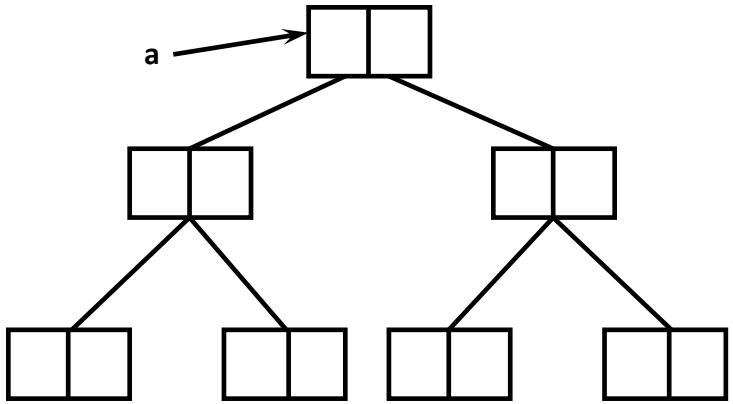
Eg2: To Print Elements of 2 Dimensional Array.





For 3 Dimensional Array

```
for (int[][][] x : a )  
{  
    for(int[] y : x)  
    {  
        for (int z : y)  
        {  
            System.out.println(z);  
        }  
    }  
}
```



Collections but its Limitation is it is Applicable only for *Arrays* and *Collections* and it is Not a General Purpose Loop.

```
for(int i =0; i<10; i++) {  
    System.out.println("Hello"); //We can't write Equivalent for-each Loop Directly  
}
```

- By using Normal for Loop we can Print an Elements of an Array in Reverse Order. But by using for-each Loop it is Not Possible.

Iterable(I) Vs Iterator(I)

Iterable(I):

- The Target Element in for-each Loop should be *Iterable* Object.
- An Object is Said to be *Iterable* if and Only if the Corresponding Class implements *java.lang.Iterable* Interface.
- *Iterable* Interface introduced in 1.5 Version and it contains Only One Method. *public Iterator iterator()*
- All Array Classes and Collection Classes Already implements Iterable Interface. Being a Programmer we are Not required to do anything.

```
for(Each Item x: Target)  
{  
    -----  
    -----  
}
```

Array/ Collection
It should be Iterable Object



Iterable (I)	Iterator (I)
It is related to for-each Loop	It is related to Collections
The Target Element in for-each Loop should be Iterable Object	We can Use Iterator Object to get Objects One by One from the Collection.
Introduced in 1.5 Version	Introduced in 1.2 Version
Present in <code>java.lang</code> Package	Present in <code>java.util</code> Package
Iterable Interface defines Only One Method <code>iterator()</code>	Iterator Interface defines 3 Methods - <code>hasNext()</code> , <code>next()</code> and <code>remove()</code>

Transfer Statements

break:

We can Use break Statement in the following Places

- ❖ Inside switch: Inside the switch we can Use break to Stop Fall-through.

```
int x = 0;
switch(x) {
    case 0:
        System.out.println(0);
    case 1:
        System.out.println(1);
        break;
    case 2:
        System.out.println(2);
    default:
        System.out.println("Default");
}
```

Output:
0
1

- ❖ Inside Loops: Inside Loops to break Loop Execution based on Some Condition.

```
for(int i = 0; i<10; i++) {
    if(i == 5)
        break;
    System.out.println(i);
}
```

Output:
0
1
2
3
4



- ❖ **Inside Labeled Block:** Inside Labeled Block to break Block Execution based on Some Condition.

```
class Test {  
    public static void main(String[] args) {  
        int x = 10;  
        l1: {  
            System.out.println("Begin");  
            if(x == 10)  
                break l1;  
            System.out.println("End");  
        }  
        System.out.println("Hello");  
    }  
}
```

Output:
Begin
Hello

These are the Only Places where we can Use break Statement. If we are using anywhere Else, we will get Compile Time Error.

```
class Test {  
    public static void main(String[] args) {  
        int x = 10;  
        if(x == 10)  
            break; //CE: break outside switch or loop  
        System.out.println("Hello");  
    }  
}
```

continue:

We can Use continue Statement to Skip Current Iteration and continue for the Next Iteration.

```
class Test {  
    public static void main(String[] args) {  
        for(int i = 0; i < 10; i++) {  
            if(i%2 == 0)   
                continue;  
            System.out.println(i);  
        }  
    }  
}
```

Output:
1
3
5
7
9

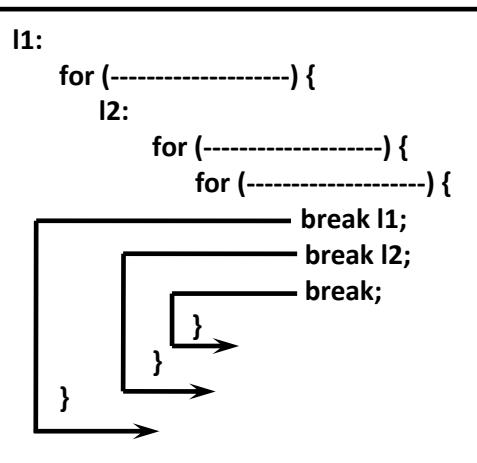
We can Use continue Statement Only Inside Loops. If we are Use anywhere Else we will get Compile Time Error.



```
class Test {  
    public static void main(String[] args) {  
        int x = 10;  
        if(x == 10)  
            continue; //CE: continue outside of loop  
        System.out.println("Hello");  
    }  
}
```

Labeled break and continue Statement:

We can Use Labeled break and continue Statements to break OR continue a Particular Loop.



```
class Test {  
    public static void main(String[] args) {  
        l1:  
        for (int i = 0; i < 3; i++ ) {  
            for (int j = 0; j < 3; j++ ) {  
                if (i == j)  
                    break;  
                //break l1;  
                // continue;  
                //continue l1;  
                System.out.println(i+"-----"+j);  
            }  
        }  
    }  
}
```

<u>break</u>	<u>break l1</u>
1-----0	No Output
2-----0	
2-----1	
<u>continue</u>	<u>continue l1</u>
0-----1	1-----0
0-----2	2-----0
1-----0	2-----1
1-----2	
2-----0	
2-----1	



do-while Vs continue (The Most Dangerous Combination)

```
class Test {  
    public static void main(String[] args) {  
        int x = 0;  
        do {  
            x++;   
            System.out.println(x);  
            if (++x < 5)  
                continue;   
            x++;  
            System.out.println(x);  
        } while (++x < 10);  
    }  
}
```

x =	Output
0	
1	1
2 < 5 ✓	4
3	6
4	8
5 < 5 X	
6	
7 < 10 ✓	
8	
9 < 5	
10	
11 < 10 X	



Declaration and Access Modifiers

Agenda

1. Java source file structure
 - o Import statement
 - o Types of Import Statements
 - Explicit class import
 - Implicit class import
 - o Difference between C language #include and java language import ?
 - o 1.5 versions new features
 - o Static import
 - Without static import
 - With static import
 - o Explain about System.out.println statement ?
 - o What is the difference between general import and static import ?
 - o Package statement
 - How to compile package Program
 - How to execute package Program
 - o Java source file structure

2. Class Modifiers
 - o Only applicable modifiers for Top Level classes
 - o What is the difference between access specifier and access modifier ?
 - o Public Classes
 - o Default Classes
 - o Final Modifier
 - Final Methods
 - Final Class
 - o Abstract Modifier
 - Abstract Methods
 - Abstract class
 - o The following are the various illegal combinations for methods
 - o What is the difference between abstract class and abstract method ?
 - o What is the difference between final and abstract ?
 - o Strictfp
 - o What is the difference between abstract and strictfp ?



3. Member modifiers

- **Public members**
- **Default member**
- **Private members**
- **Protected members**
- **Compression of private, default, protected and public**
- **Final variables**
 - **Final instance variables**
 - **At the time of declaration**
 - **Inside instance block**
 - **Inside constructor**
 - **Final static variables**
 - **At the time of declaration**
 - **Inside static block**
 - **Final local variables**
- **Formal parameters**
- **Static modifier**
- **Native modifier**
 - **Pseudo code**
- **Synchronized**
- **Transient modifier**
- **Volatile modifier**
- **Summary of modifier**

4. Interfaces

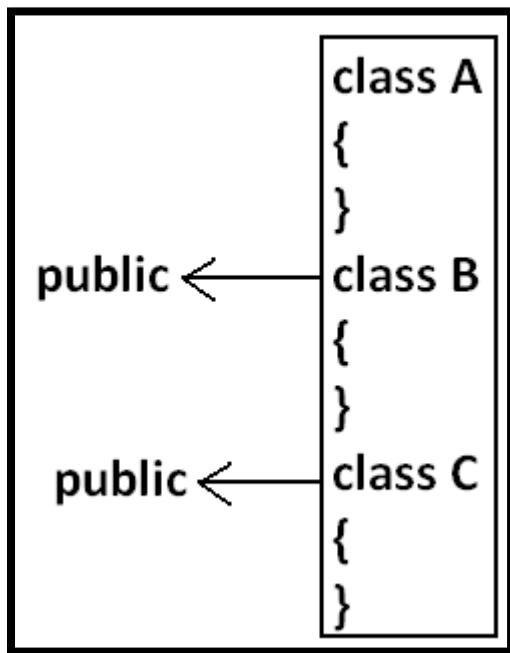
- **Interface declarations and implementations**
- **Extends vs implements**
- **Interface methods**
- **Interface variables**
- **Interface naming conflicts**
 - **Method naming conflicts**
 - **Variable naming conflicts**
- **Marker interface**
- **Adapter class**
- **Interface vs abstract class vs concrete class**
- **Difference between interface and abstract class?**
- **Conclusions**



Java source file structure:

- A Java Program can contain any no of classes but at most one class can be declared as public. "If there is a public class the name of the Program and name of the public class must be matched otherwise we will get compile time error".
- If there is no public class then any name we gives for Java source file.

Example:



Case 1:

If there is no public class then we can use any name for java source file there are no restrictions.

Example:

A.java
B.java
C.java
Ashok.java

Case 2:

If class B declared as public then the name of the Program should be B.java otherwise we will get compile time error saying "class B is public, should be declared in a file named B.java".



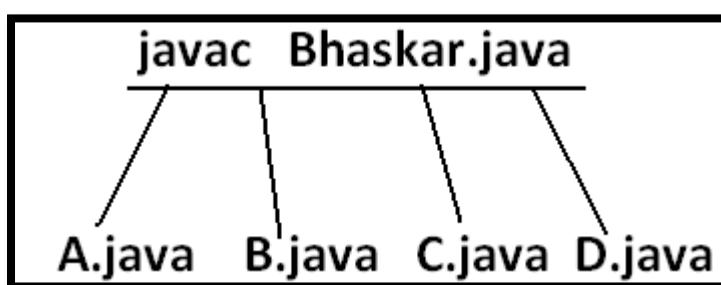
Case 3:

- If both B and C classes are declared as public and name of the file is B.java then we will get compile time error saying "class C is public, should be declared in a file named C.java".
- It is highly recommended to take only one class for source file and name of the Program (file) must be same as class name. This approach improves readability and understandability of the code.

Example:

```
1) class A {  
2)     public static void main(String args[]) {  
3)         System.out.println("A class main method is executed");  
4)     }  
5) }  
6)  
7) class B {  
8)     public static void main(String args[]) {  
9)         System.out.println("B class main method is executed");  
10)    }  
11) }  
12)  
13) class C {  
14)     public static void main(String args[]) {  
15)         System.out.println("C class main method is executed");  
16)     }  
17) }  
18)  
19) class D {  
20) }
```

Output:



D:\Java>java A
A class main method is executed
D:\Java>java B
B class main method is executed



D:\Java>java C
C class main method is executed
D:\Java>java D
Exception in thread "main" java.lang.NoSuchMethodError: main
D:\Java>java Ashok
Exception in thread "main" java.lang.NoClassDefFoundError: Ashok

- We can compile a java Program but not java class in that Program for every class one dot class file will be created.
- We can run a java class but not java source file whenever we are trying to run a class the corresponding class main method will be executed.
- If the class won't contain main method then we will get runtime exception saying "NoSuchMethodError: main".
- If we are trying to execute a java class and if the corresponding .class file is not available then we will get runtime execution saying "NoClassDefFoundError: Ashok".

Import statement:

```
1) class Test {  
2)     public static void main(String args[]) {  
3)         ArrayList l = new ArrayList();  
4)     }  
5) }
```

Output:

Compile time error.
D:\Java>javac Test.java
Test.java:3: cannot find symbol
symbol : class ArrayList
location: class Test

ArrayList l = new ArrayList();

- We can resolve this problem by using fully qualified name
`java.util.ArrayList l = new java.util.ArrayList();`
- But problem with using fully qualified name every time is it increases length of the code and reduces readability.
- We can resolve this problem by using import statements.

Example:

```
1) import java.util.ArrayList;  
2) class Test {  
3)     public static void main(String args[]) {  
4)         ArrayList l = new ArrayList();  
5)     }
```



6) }

Output:

D:\Java>javac Test.java

Hence whenever we are using import statement it is not require to use fully qualified names we can use short names directly. This approach decreases length of the code and improves readability.

Case 1: Types of Import Statements:

There are 2 types of import statements.

- 1) Explicit class import
- 2) Implicit class import.

Explicit class import:

Example: Import java.util.ArrayList

- This type of import is highly recommended to use because it improves readability of the code.
- Best suitable for Hi-Tech city where readability is important.

Implicit class import:

Example: import java.util.*;

- It is never recommended to use because it reduces readability of the code.
- Best suitable for Ameerpet where typing is important.

Case 2: Which of the following import statements are meaningful?

import java.util; X
import java.util.ArrayList.*; X
import java.util.*; ✓
import java.util.ArrayList; ✓

Case 3: Consider the following code.

- 1) **class MyArrayList extends java.util.ArrayList**
- 2) {
- 3) }



- The code compiles fine even though we are not using import statements because we used fully qualified name.
- Whenever we are using fully qualified name it is not required to use import statement. Similarly whenever we are using import statements it is not required to use fully qualified name.

Case 4:

Example:

```
1) import java.util.*;
2) import java.sql.*;
3) class Test {
4)     public static void main(String args[]) {
5)         Date d = new Date();
6)     }
7) }
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:7: reference to Date is ambiguous;
both class java.sql.Date in java.sql and class java.util.Date in java.util match

`Date d = new Date();`

Note: Even in the List case also we may get the same ambiguity problem because it is available in both util and awt packages.

Case 5:

While resolving class names compiler will always gives the importance in the following order.

1. Explicit class import
2. Classes present in current working directory.
3. Implicit class import.

Example:

```
1) import java.util.Date;
2) import java.sql.*;
3) class Test {
4)     public static void main(String args[]) {
5)         Date d = new Date();
6)     }
7) }
```

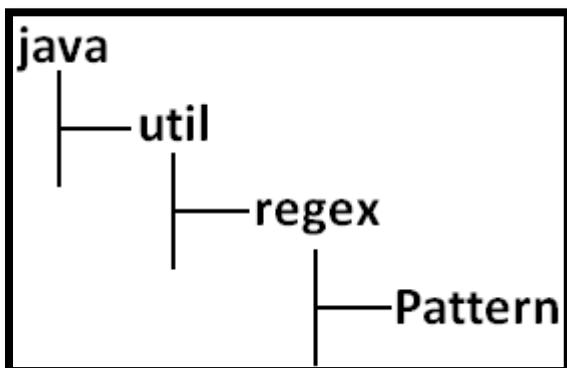


The code compiles fine and in this case util package Date will be considered.

Case 6:

Whenever we are importing a package all classes and interfaces present in that package are by default available but not sub package classes.

Example:



To use pattern class in our Program directly which import statement is required?

1. `import java.*;` X
2. `import java.util.*;` X
3. `import java.util.regex.*;` ✓
4. `import java.util.regex.Pattern;` ✓

Case 7:

In any java Program the following 2 packages are not require to import because these are available by default to every Java Program.

1. `java.lang` package
2. default package (current working directory)

Case 8:

"Import statement is totally compile time concept" if more no of imports are there then more will be the compile time but there is "no change in execution time".



Difference between C language #include and java language import ?

#include	import
It can be used in C & C++	It can be used in Java
At compile time only compiler copy the code from standard library and placed in current program.	At runtime JVM will execute the corresponding standard library and use it's result in current program.
It is static inclusion	It is dynamic inclusion
wastage of memory	No wastage of memory
Ex : <jsp:@ file="">	Ex : <jsp:include >

- In the case of C language #include all the header files will be loaded at the time of include statement hence it follows static loading.
- But in java import statement no ".class" will be loaded at the time of import statements in the next lines of the code whenever we are using a particular class then only corresponding ".class" file will be loaded. Hence it follows "dynamic loading" or "load-on-demand" or "load-on-fly".

1.5 versions new features :

- 1) For-Each
- 2) Var-arg
- 3) Queue
- 4) Generics
- 5) Auto boxing and Auto unboxing
- 6) Co-variant return types
- 7) Annotations
- 8) Enum
- 9) Static import
- 10) String builder

Static import:

This concept introduced in 1.5 versions. According to sun static import improves readability of the code but according to worldwide Programming exports (like us) static imports creates confusion and reduces readability of the code. Hence if there is no specific requirement never recommended to use a static import.

Usually we can access static members by using class name but whenever we are using static import it is not require to use class name we can access directly.



Without static import:

```
1) class Test {  
2)     public static void main(String args[]) {  
3)         System.out.println(Math.sqrt(4));  
4)         System.out.println(Math.max(10,20));  
5)         System.out.println(Math.random());  
6)     }  
7) }
```

Output:

D:\Java>javac Test.java

D:\Java>java Test

2.0

20

0.841306154315576

With static import:

```
1) import static java.lang.Math.sqrt;  
2) import static java.lang.Math.*;  
3) class Test {  
4)     public static void main(String args[]) {  
5)         System.out.println(sqrt(4));  
6)         System.out.println(max(10,20));  
7)         System.out.println(random());  
8)     }  
9) }
```

Output:

D:\Java>javac Test.java

D:\Java>java Test

2.0

20

0.4302853847363891



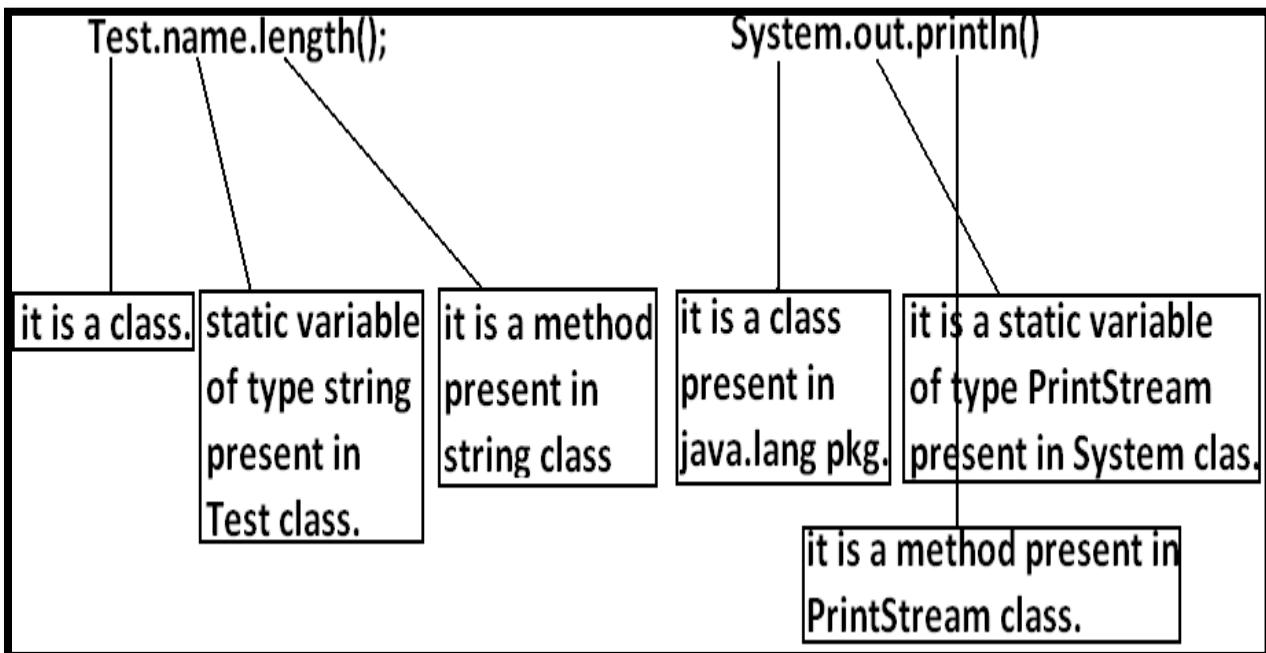
Explain about System.out.println statement?

Example 1

```
1)
class Test
{
    static String name="bhaskar";
}
```

Example 2

```
2)
import java.io.*;
class System
{
    static PrintStream out;
}
```



Example 3:

```
1) import static java.lang.System.out;
2) class Test {
3)     public static void main(String args[]) {
4)         out.println("hello");
5)         out.println("hi");
6)     }
}
```



7) }

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
hello
hi
```

Example 4:

```
1) import static java.lang.Integer.*;
2) import static java.lang.Byte.*;
3) class Test {
4)     public static void main(String args[]) {
5)         System.out.println(MAX_VALUE);
6)     }
7) }
```

Output:

```
Compile time error.
D:\Java>javac Test.java
Test.java:6: reference to MAX_VALUE is ambiguous,
 both variable MAX_VALUE in java.lang.Integer and variable MAX_VALUE in java.lang.Byte match
System.out.println(MAX_VALUE);
```

Note: Two packages contain a class or interface with the same name is very rare hence ambiguity problem is very rare in normal import.

But 2 classes or interfaces can contain a method or variable with the same name is very common hence ambiguity problem is also very common in static import.

While resolving static members compiler will give the precedence in the following order.

1. Current class static members
2. Explicit static import
3. Implicit static import.



Example:

```
//import static java.lang.Integer.MAX_VALUE;→line2
import static java.lang.Byte.*;
class Test
{
//static int MAX_VALUE=999;→line1
public static void main(String args[])throws Exception{
System.out.println(MAX_VALUE);
}
}
```

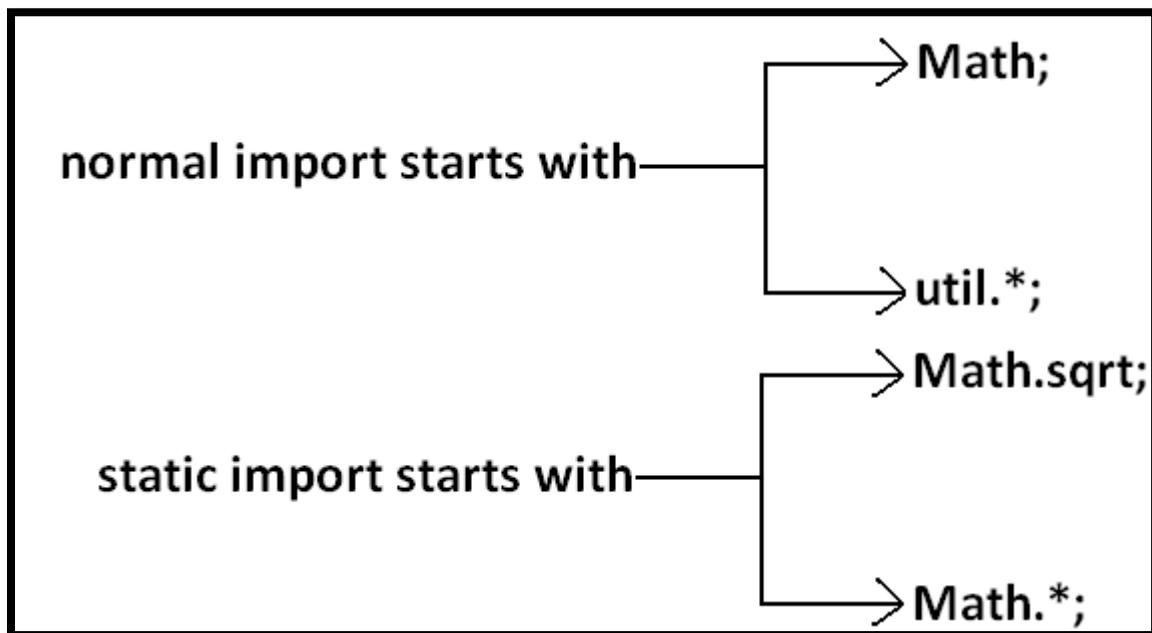
- If we comment line one then we will get Integer class MAX_VALUE 2147483647.
- If we comment lines one and two then Byte class MAX_VALUE will be considered 127.

Which of the following import statements are valid ?

1. import java.lang.Math.*; ×
2. import static java.lang.Math.*; ✓
3. import java.lang.Math; ✓
4. import static java.lang.Math; ×
5. import static java.lang.Math.sqrt.*; ×
6. import java.lang.Math.sqrt; ×
7. import static java.lang.Math.sqrt(); ×
8. import static java.lang.Math.sqrt; ✓



Diagram



Usage of static import reduces readability and creates confusion hence if there is no specific requirement never recommended to use static import.

What is the difference between general import and static import?

We can use normal imports to import classes and interfaces of a package. whenever we are using normal import we can access class and interfaces directly by their short name it is not require to use fully qualified names.

We can use static import to import static members of a particular class. whenever we are using static import it is not require to use class name we can access static members directly.

Package statement:

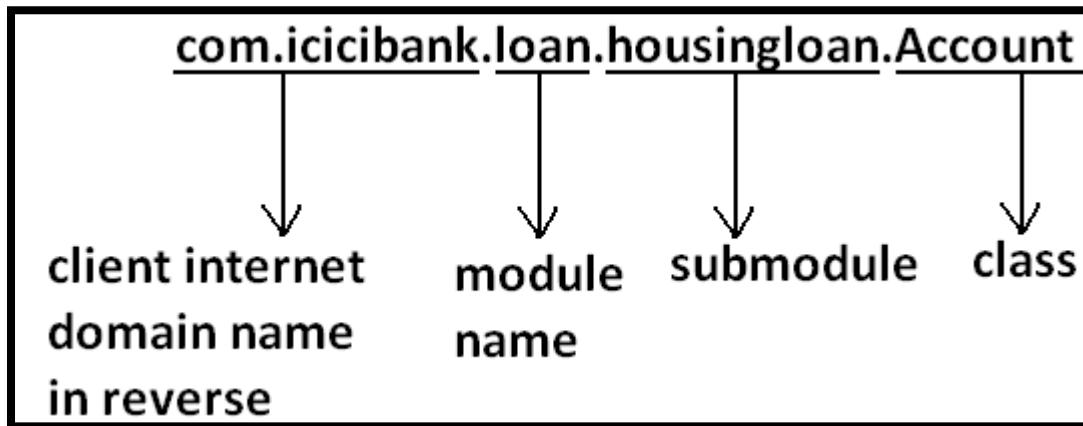
It is an encapsulation mechanism to group related classes and interfaces into a single module.

The main objectives of packages are:

- To resolve name conflicts.
- To improve modularity of the application.
- To provide security.
- There is one universally accepted naming convention for packages that is to use internet domain name in reverse.



Example:



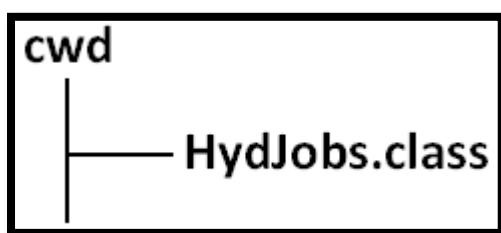
How to compile package Program:

Example:

```
1) package com.durgajobs.itjobs;
2) class HydJobs {
3)     public static void main(String args[]) {
4)         System.out.println("package demo");
5)     }
6) }
```

Javac HydJobs.java generated class files will be placed in current working directory.

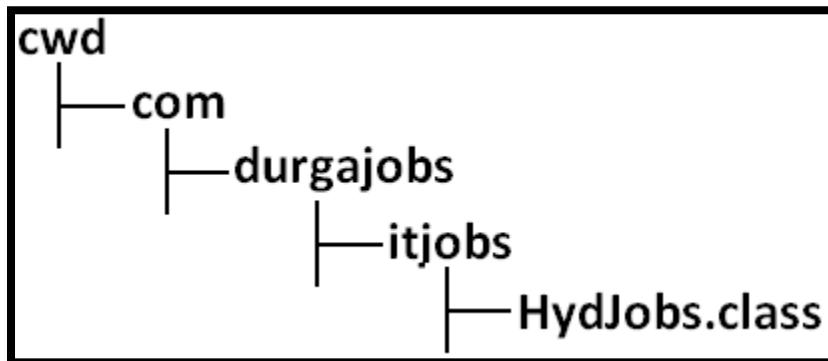
Diagram:



- Javac -d . HydJobs.java
- -d means destination to place generated class files "." means current working directory.
- Generated class file will be placed into corresponding package structure.



Diagram:

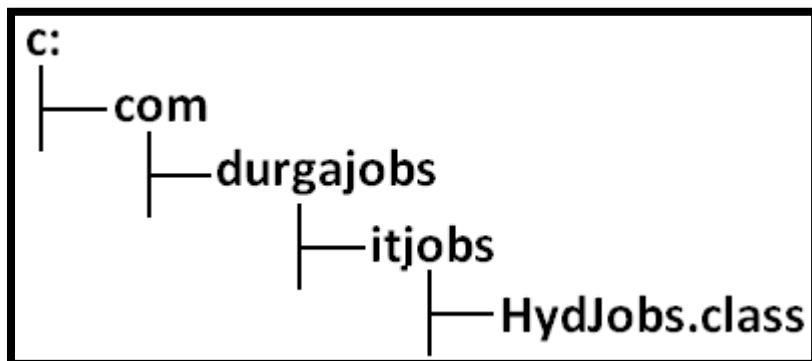


- If the specified package structure is not already available then this command itself will create the required package structure.
- As the destination we can use any valid directory.
- If the specified destination is not available then we will get compile time error.

Example:

D:\Java>javac -d c: HydJobs.java

Diagram:



If the specified destination is not available then we will get compile time error.

Example:

D:\Java>javac -d z: HydJobs.java

If Z: is not available then we will get compile time error.



How to execute package Program:

D:\Java>java com.durgajobs.itjobs.HydJobs

At the time of execution compulsory we should provide fully qualified name.

Conclusion 1:

In any java Program there should be at most one package statement that is if we are taking more than one package statement we will get compile time error.

Example:

```
1) package pack1;
2) package pack2;
3) class A
4) {
5) }
```

Output:

Compile time error.

D:\Java>javac A.java

A.java:2: class, interface, or enum expected
package pack2;

Conclusion 2:

In any java Program the 1st non comment statement should be package statement [if it is available] otherwise we will get compile time error.

Example:

```
1) import java.util.*;
2) package pack1;
3) class A
4) {
5) }
```

Output:

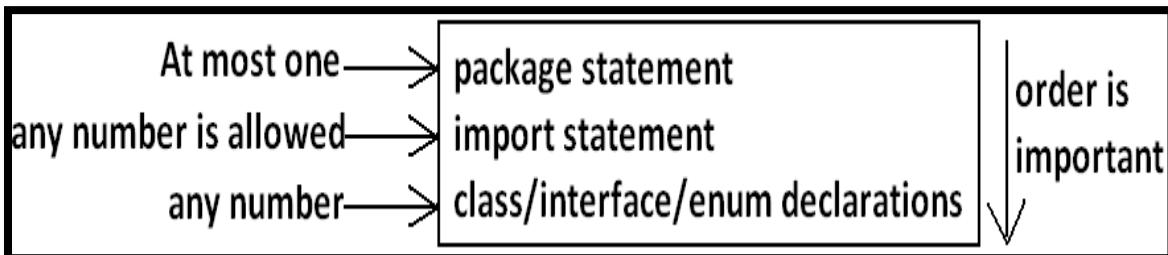
Compile time error.

D:\Java>javac A.java

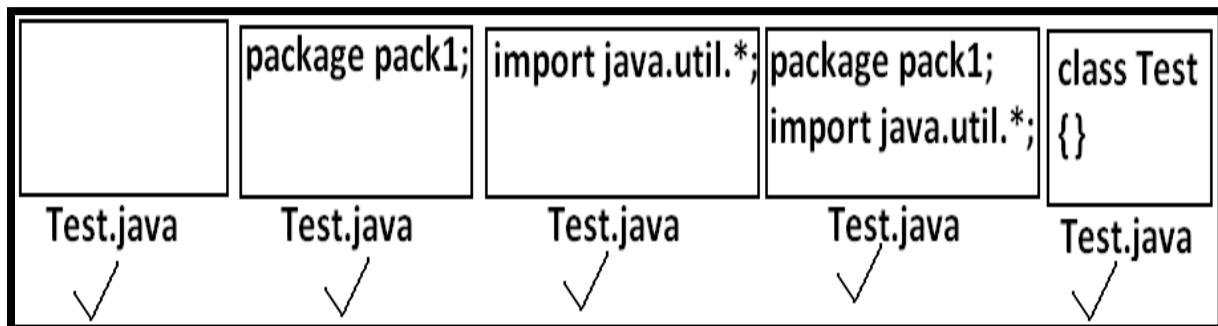
A.java:2: class, interface, or enum expected
package pack1;



Java source file structure



All the following are valid java Programs.



Note: An empty source file is a valid Java Program.



Class Modifiers

Whenever we are writing our own classes compulsory we have to provide some information about our class to the JVM.

Like

- 1) Whether this class can be accessible from anywhere or not.
- 2) Whether child class creation is possible or not.
- 3) Whether object creation is possible or not etc.

We can specify this information by using the corresponding modifiers.

The only applicable modifiers for Top Level classes are:

- 1) Public
- 2) Default
- 3) Final
- 4) Abstract
- 5) Strictfp

If we are using any other modifier we will get compile time error.

Example:

```
1) private class Test {  
2)     public static void main(String args[]) {  
3)         int i = 0;  
4)         for(int j=0; j<3; j++) {  
5)             i = i+j;  
6)         }  
7)         System.out.println(i);  
8)     }  
9) }
```

OUTPUT:

Compile time error.

D:\Java>javac Test.java

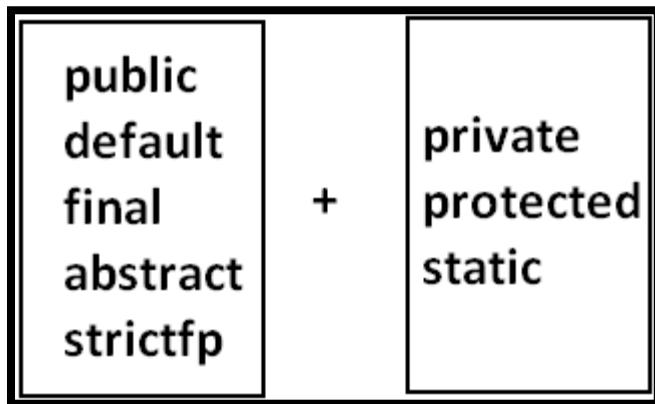
Test.java:1: modifier private not allowed here

private class Test

But For the inner classes the following modifiers are allowed.



Diagram:



What is the difference between access specifier and access modifier?

In old languages 'C' (or) 'C++' public, private, protected, default are considered as access specifiers and all the remaining are considered as access modifiers.

But in java there is no such type of division all are considered as access modifiers.

Public Classes:

If a class declared as public then we can access that class from anywhere. Within the package or outside the package.

Example:

Program1:

```
1) package pack1;
2) public class Test {
3)     public void methodOne() {
4)         System.out.println("test class methodone is executed");
5)     }
6) }
```

Compile the above Program:

D:\Java>javac -d . Test.java

Program2:

```
1) package pack2;
2) import pack1.Test;
3) class Test1 {
4)     public static void main(String args[]) {
5)         Test t = new Test();
```



```
6)         t.methodOne();  
7)     }  
8) }
```

Output:

```
D:\Java>javac -d . Test1.java  
D:\Java>java pack2.Test1  
Test class methodone is executed.
```

If class Test is not public then while compiling Test1 class we will get compile time error saying pack1.Test is not public in pack1; cannot be accessed from outside package.

Default Classes:

If a class declared as the default then we can access that class only within the current package hence default access is also known as "package level access".

Example:

Program 1:

```
1) package pack1;  
2) class Test {  
3)     public void methodOne() {  
4)         System.out.println("test class methodone is executed");  
5)     }  
6) }
```

Program 2:

```
1) package pack1;  
2) import pack1.Test;  
3) class Test1 {  
4)     public static void main(String args[]) {  
5)         Test t = new Test();  
6)         t.methodOne();  
7)     }  
8) }
```

Output:

```
D:\Java>javac -d . Test.java  
D:\Java>javac -d . Test1.java  
D:\Java>java pack1.Test1  
Test class methodone is executed
```



Final Modifier:

Final is the modifier applicable for classes, methods and variables.

Final Methods:

- Whatever the methods parent has by default available to the child.
- If the child is not allowed to override any method, that method we have to declare with final in parent class. That is final methods cannot be overridden.

Example:

Program 1:

```
1) class Parent {  
2)     public void property() {  
3)         System.out.println("cash+gold+land");  
4)     }  
5)     public final void marriage() {  
6)         System.out.println("subbalakshmi");  
7)     }  
8) }
```

Program 2:

```
1) class child extends Parent {  
2)     public void marriage() {  
3)         System.out.println("Thamanna");  
4)     }  
5) }
```

OUTPUT:

Compile time error.

D:\Java>javac Parent.java

D:\Java>javac child.java

child.java:3: marriage() in child cannot override marriage() in Parent;
overridden method is final

public void marriage() {



Final Class:

If a class declared as the final then we can't create the child class that is inheritance concept is not applicable for final classes.

Example:

Program 1:

```
1) final class Parent  
2) {  
3) }
```

Program 2:

```
1) class child extends Parent  
2) {  
3) }
```

OUTPUT:

Compile time error.

D:\Java>javac Parent.java

D:\Java>javac child.java

child.java:1: cannot inherit from final Parent

class child extends Parent

Note: Every method present inside a final class is always final by default whether we are declaring or not. But every variable present inside a final class need not be final.

Example:

```
1) final class parent {  
2)     static int x = 10;  
3)     static {  
4)         x = 999;  
5)     }  
6) }
```

The main advantage of final keyword is we can achieve security.

Whereas the main disadvantage is we are missing the key benefits of oops:

polymorphism (because of final methods), inheritance (because of final classes) hence if there is no specific requirement never recommended to use final keyword.



Abstract Modifier:

Abstract is the modifier applicable only for methods and classes but not for variables.

Abstract Methods:

Even though we don't have implementation still we can declare a method with abstract modifier. That is abstract methods have only declaration but not implementation. Hence abstract method declaration should compulsory ends with semicolon.

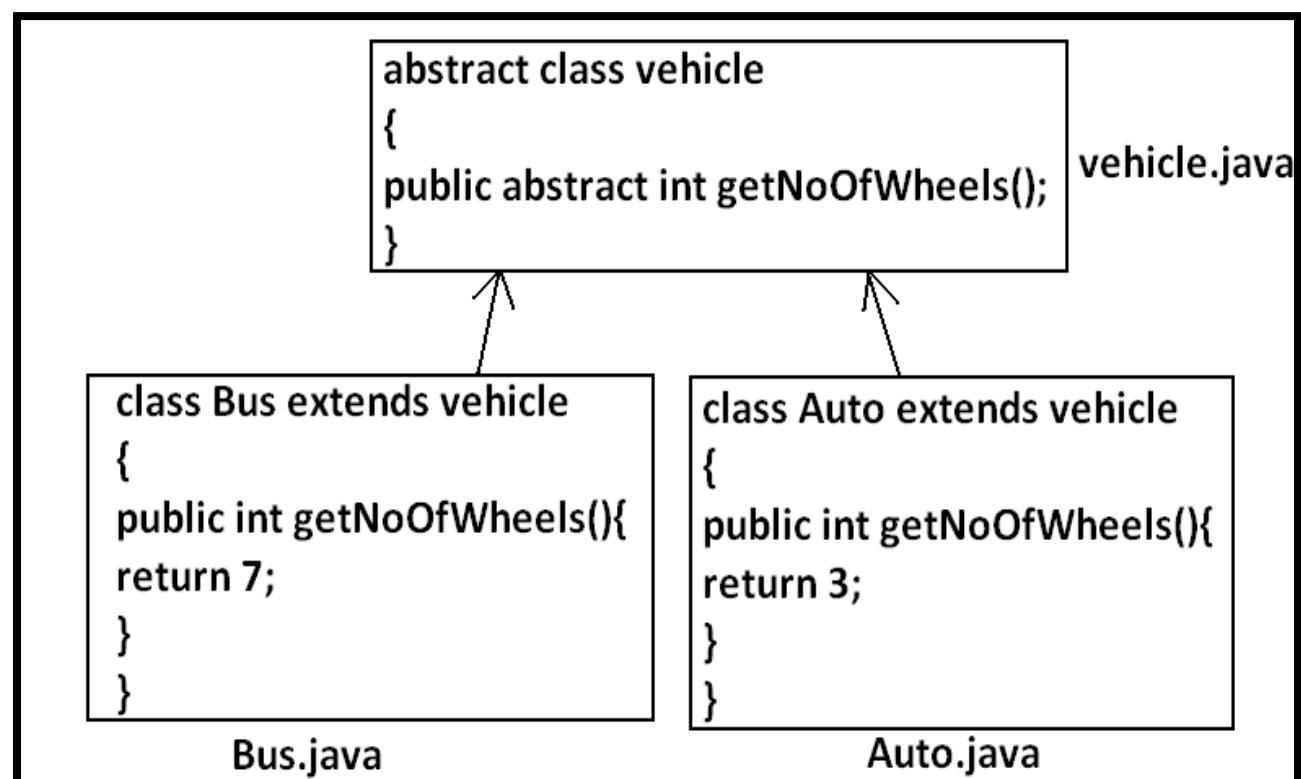
Example:

```
public abstract void methodOne(); → valid
public abstract void methodOne(){ } → invalid
```

Child classes are responsible to provide implementation for parent class abstract methods.

Example:

Program:

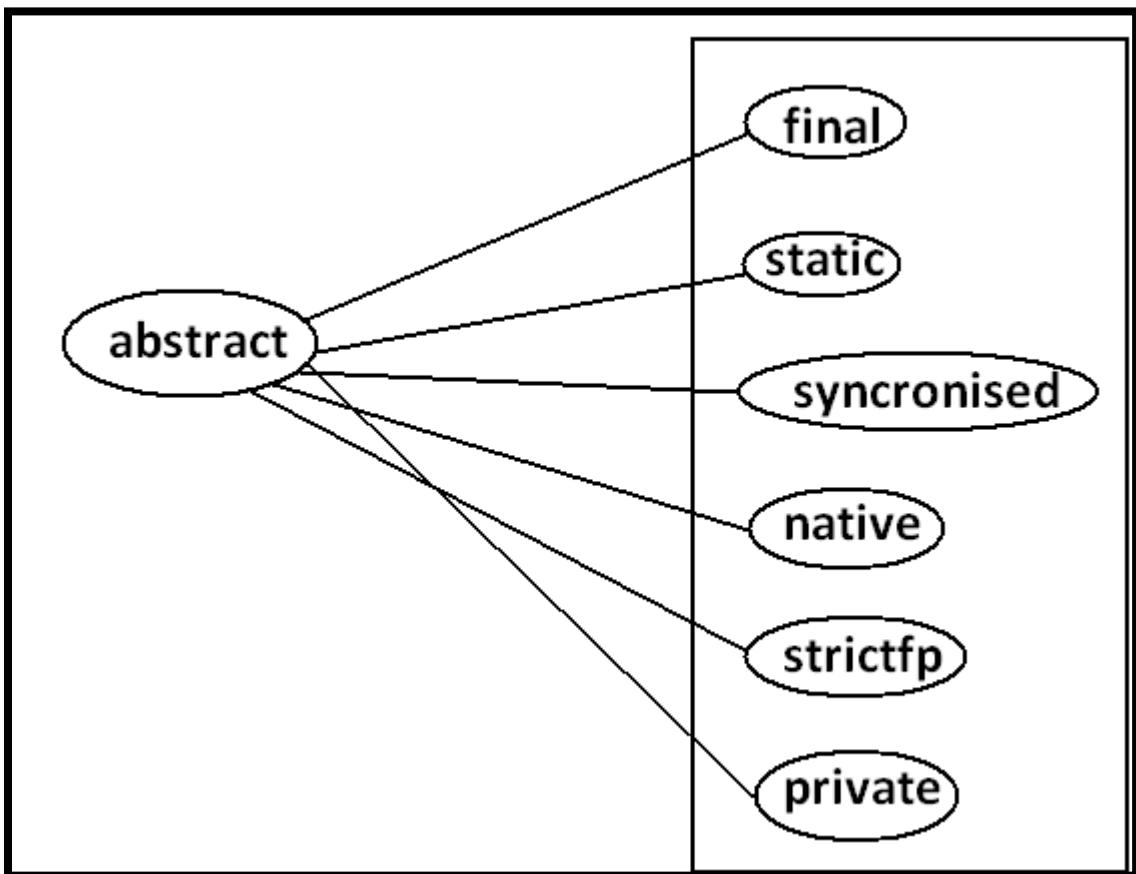




- The main advantage of abstract methods is , by declaring abstract method in parent class we can provide guide lines to the child class such that which methods they should compulsory implement.
- Abstract method never talks about implementation whereas if any modifier talks about implementation then the modifier will be enemy to abstract and that is always illegal combination for methods.

The following are the various illegal combinations for methods.

Diagram:



All the 6 combinations are illegal.



Abstract class:

For any java class if we are not allow to create an object such type of class we have to declare with abstract modifier that is for abstract class instantiation is not possible.

Example:

```
1) abstract class Test {  
2)     public static void main(String args[]) {  
3)         Test t = new Test();  
4)     }  
5) }
```

Output:

Compile time error.

```
D:\Java>javac Test.java  
Test.java:4: Test is abstract; cannot be instantiated  
Test t=new Test();
```

What is the difference between abstract class and abstract method?

- If a class contain at least one abstract method then compulsory the corresponding class should be declare with abstract modifier. Because implementation is not complete and hence we can't create object of that class.
- Even though class doesn't contain any abstract methods still we can declare the class as abstract that is an abstract class can contain zero no of abstract methods also.

Example1: HttpServlet class is abstract but it doesn't contain any abstract method.

Example2: Every adapter class is abstract but it doesn't contain any abstract method.

Example1:

```
1) class Parent {  
2)     public void methodOne();  
3) }
```

Output:

Compile time error.

```
D:\Java>javac Parent.java  
Parent.java:3: missing method body, or declare abstract  
public void methodOne();
```



Example2:

```
1) class Parent {  
2)     public abstract void methodOne() {}  
3) }
```

Output:

Compile time error.
Parent.java:3: abstract methods cannot have a body
public abstract void methodOne(){}

Example3:

```
1) class Parent {  
2)     public abstract void methodOne();  
3) }
```

Output:

```
Compile time error.  
D:\Java>javac Parent.java  
Parent.java:1: Parent is not abstract and does not override abstract method methodOne() in  
Parent class Parent
```

If a class extends any abstract class then compulsory we should provide implementation for every abstract method of the parent class otherwise we have to declare child class as abstract.

Example:

```
1) abstract class Parent {  
2)     public abstract void methodOne();  
3)     public abstract void methodTwo();  
4) }  
5) class child extends Parent {  
6)     public void methodOne() {}  
7) }
```

Output:

Compile time error.
D:\Java>javac Parent.java
Parent.java:6: child is not abstract and does not override abstract method methodTwo() in Parent
class child extends Parent
If we declare class child as abstract then the code compiles fine but child of child is responsible to
provide implementation for methodTwo().



What is the difference between final and abstract?

- For abstract methods compulsory we should override in the child class to provide implementation. Whereas for final methods we can't override hence abstract final combination is illegal for methods.
- For abstract classes we should compulsory create child class to provide implementation whereas for final class we can't create child class. Hence final abstract combination is illegal for classes.
- Final class cannot contain abstract methods whereas abstract class can contain final method.

Example:

final class A

{

public abstract void
 methodOne();

}

invalid

abstract class A

{

public final void methodOne();

}

valid

Note: Usage of abstract methods, abstract classes and interfaces is always good Programming practice.

Strictfp:

- strictfp is the modifier applicable for methods and classes but not for variables.
- Strictfp modifier introduced in 1.2 versions.
- Usually the result of floating point of arithmetic is varying from platform to platform , to overcome this problem we should use strictfp modifier.
- If a method declare as the Strictfp then all the floating point calculations in that method has to follow IEEE754 standard, So that we will get platform independent results.

Example:

System.out.println(10.0/3);

P4

3.33333333333333

P3

3.333333

IEEE754

3.333



If a class declares as the Strictfp then every concrete method(which has body) of that class has to follow IEEE754 standard for floating point arithmetic, so we will get platform independent results.

What is the difference between abstract and strictfp?

Strictfp method talks about implementation where as abstract method never talks about implementation hence abstract, strictfp combination is illegal for methods.
But we can declare a class with abstract and strictfp modifier simultaneously. That is abstract strictfp combination is legal for classes but illegal for methods.

Example:

```
public abstract strictfp void methodOne(); (invalid)
abstract strictfp class Test (valid)
{
```



Member modifiers

Public members:

If a member declared as the public then we can access that member from anywhere "but the corresponding class must be visible" hence before checking member visibility we have to check class visibility.

Example:

Program 1:

```
1) package pack1;
2) class A {
3)     public void methodOne() {
4)         System.out.println("a class method");
5)     }
6) }
```

D:\Java>javac -d . A.java

Program 2:

```
1) package pack2;
2) import pack1.A;
3) class B {
4)     public static void main(String args[]) {
5)         A a = new A();
6)         a.methodOne();
7)     }
8) }
```

Output:

Compile time error.

D:\Java>javac -d . B.java

B.java:2: pack1.A is not public in pack1;
cannot be accessed from outside package
import pack1.A;

In the above Program even though methodOne() method is public we can't access from class B because the corresponding class A is not public that is both classes and methods are public then only we can access.



Default member:

If a member declared as the default then we can access that member only within the current package hence default member is also known as package level access.

Example 1:

Program 1:

```
1) package pack1;
2) class A {
3)     void methodOne() {
4)         System.out.println("methodOne is executed");
5)     }
6) }
```

Program 2:

```
1) package pack1;
2) import pack1.A;
3) class B {
4)     public static void main(String args[]) {
5)         A a = new A();
6)         a.methodOne();
7)     }
8) }
```

Output:

D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
D:\Java>java pack1.B
methodOne is executed

Example 2:

Program 1:

```
1) package pack1;
2) class A {
3)     void methodOne() {
4)         System.out.println("methodOne is executed");
5)     }
6) }
```



Program 2:

```
1) package pack2;
2) import pack1.A;
3) class B {
4)     public static void main(String args[]) {
5)         A a = new A();
6)         a.methodOne();
7)     }
8) }
```

Output:

Compile time error.

D:\Java>javac -d . A.java

D:\Java>javac -d . B.java

B.java:2: pack1.A is not public in pack1; cannot be accessed from outside package

import pack1.A;

Private members:

- If a member declared as the private then we can access that member only with in the current class.
- Private methods are not visible in child classes where as abstract methods should be visible in child classes to provide implementation hence private, abstract combination is illegal for methods.

Protected members:

- If a member declared as the protected then we can access that member within the current package anywhere but outside package only in child classes.
Protected = default+kids.
- We can access protected members within the current package anywhere either by child reference or by parent reference
- But from outside package we can access protected members only in child classes and should be by child reference only that is we can't use parent reference to call protected members from outside package.

Example:

Program 1:

```
1) package pack1;
2) public class A {
3)     protected void methodOne() {
4)         System.out.println("methodOne is executed");
5)     }
}
```



6) }

Program 2:

```
1) package pack1;
2) class B extends A {
3)     public static void main(String args[]) {
4)         A a = new A();
5)         a.methodOne();
6)         B b = new B();
7)         b.methodOne();
8)         A a1 = new B();
9)         a1.methodOne();
10)    }
11) }
```

Output:

```
D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
D:\Java>java pack1.B
methodOne is executed
methodOne is executed
methodOne is executed
```

Example 2:

```
package pack2;
import pack1.A;
public class C extends A
{
    public static void main(String args[]){
        A a=new A();
        a.methodOne(); X
        C c=new C();
        c.methodOne(); ✓
        A a1=new B();
        a1.methodOne(); X
    }
}
```

output:
compile time error.
D:\Java>javac -d . C.java
C.java:7: methodOne() has protected access in
pack1.A
a.methodOne();



Compression of private, default, protected and public:

visibility	private	default	protected	public
1)With in the same class	✓	✓	✓	✓
2)From child class of same package	✗	✓	✓	✓
3)From non-child class of same package	✗	✓	✓	✓
4)From child class of outside package	✗	✗	✓ but we should use child reference only	✓
5)From non-child class of outside package	✗	✗	✗	✓

- The least accessible modifier is private.
- The most accessible modifier is public.

Private < default < protected < public

Recommended modifier for variables is private where as recommended modifier for methods is public.

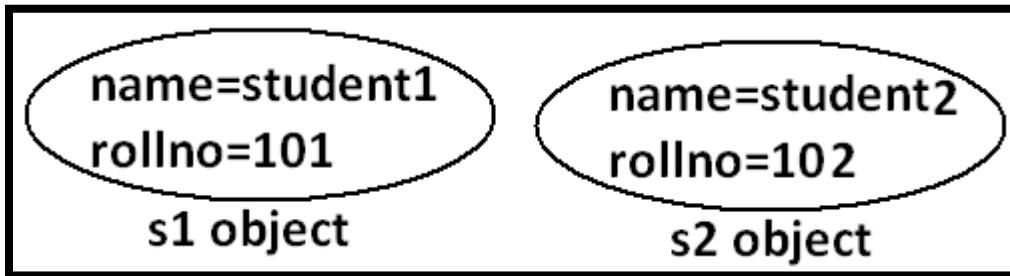


Final variables:

Final instance variables:

- If the value of a variable is varied from object to object such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.

DIAGRAM:



For the instance variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

```
1) class Test {  
2)     int i;  
3)     public static void main(String args[]) {  
4)         Test t = new Test();  
5)         System.out.println(t.i);  
6)     }  
7) }
```

Output:

```
D:\Java>javac Test.java  
D:\Java>java Test  
0
```

If the instance variable declared as the final compulsory we should perform initialization explicitly and JVM won't provide any default values.

Whether we are using or not otherwise we will get compile time error.



Example:

Program 1:

```
1) class Test {  
2)     int i;  
3) }
```

Output:

D:\Java>javac Test.java
D:\Java>

Program 2:

```
1) class Test {  
2)     final int i;  
3) }
```

Output:

Compile time error.
D:\Java>javac Test.java
Test.java:1: variable i might not have been initialized
class Test

Rule:

For the final instance variables we should perform initialization before constructor completion.
That is the following are various possible places for this.

1) At the time of declaration:

Example:

```
1) class Test {  
2)     final int i = 10;  
3) }
```

Output:

D:\Java>javac Test.java
D:\Java>



2) Inside instance block:

Example:

```
1) class Test {  
2)     final int i;  
3)     {  
4)         i = 10;  
5)     }  
6) }
```

Output:

D:\Java>javac Test.java
D:\Java>

3) Inside constructor:

Example:

```
1) class Test {  
2)     final int i;  
3)     Test() {  
4)         i = 10;  
5)     }  
6) }
```

Output:

D:\Java>javac Test.java
D:\Java>

If we are performing initialization anywhere else we will get compile time error.

Example:

```
1) class Test {  
2)     final int i;  
3)     public void methodOne() {  
4)         i = 10;  
5)     }  
6) }
```

Output:

Compile time error.
D:\Java>javac Test.java



Test.java:5: cannot assign a value to final variable i
i=10;

Final static variables:

- If the value of a variable is not varied from object to object such type of variables is not recommended to declare as the instance variables. We have to declare those variables at class level by using static modifier.
- In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by every object of that class.
- For the static variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

```
1) class Test {  
2)     static int i;  
3)     public static void main(String args[]) {  
4)         System.out.println("value of i is :" +i);  
5)     }  
6) }
```

Output:

```
D:\Java>javac Test.java  
D:\Java>java Test  
Value of i is: 0
```

If the static variable declare as final then compulsory we should perform initialization explicitly whether we are using or not otherwise we will get compile time error.
(The JVM won't provide any default values)



Example:

```
class Test
{
    static int i;
```

valid

```
class Test
{
    final static int i;
}
```

invalid

output:

```
D:\Java>javac Test.java
Test.java:1: variable i might not have been initialized
class Test
```

Rule:

For the final static variables we should perform initialization before class loading completion otherwise we will get compile time error. That is the following are possible places.

1) At the time of declaration:

Example:

```
1) class Test {
2)     final static int i = 10;
3) }
```

Output:

```
D:\Java>javac Test.java
D:\Java>
```

2) Inside static block:

Example:

```
1) class Test {
2)     final static int i;
3)     static {
4)         i = 10;
5)     }
```



6) }

Output: Compile successfully.

If we are performing initialization anywhere else we will get compile time error.

Example:

```
1) class Test {  
2)     final static int i;  
3)     public static void main(String args[]) {  
4)         i = 10;  
5)     }  
6) }
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: cannot assign a value to final variable i

i=10;

Final local variables:

- To meet temporary requirement of the Programmer sometime we can declare the variable inside a method or block or constructor such type of variables are called local variables.
- For the local variables jvm won't provide any default value compulsory we should perform initialization explicitly before using that variable.

Example:

```
1) class Test {  
2)     public static void main(String args[]) {  
3)         int i;  
4)         System.out.println("Hello");  
5)     }  
6) }
```

Output:

D:\Java>javac Test.java

D:\Java>java Test

Hello



Example:

```
1) class Test {  
2)     public static void main(String args[]) {  
3)         int i;  
4)         System.out.println(i);  
5)     }  
6) }
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: variable i might not have been initialized

System.out.println(i);

Even though local variable declared as the final before using only we should perform initialization.

Example:

```
1) class Test {  
2)     public static void main(String args[]) {  
3)         final int i;  
4)         System.out.println("hello");  
5)     }  
6) }
```

Output:

D:\Java>javac Test.java

D:\Java>java Test

hello

Note: The only applicable modifier for local variables is final if we are using any other modifier we will get compile time error.



Example:

```
class Test
{
    public static void main(String args[])
    {
        private int x=10; ----- (invalid)
        public int x=10; ----- (invalid)
        volatile int x=10; ----- (invalid)
        transient int x=10; ----- (invalid)
        final int x=10; ----- (valid)
    }
}
```

Output:

Compile time error.
D:\Java>javac Test.java
Test.java:5: illegal start of expression
private int x=10;



Formal parameters:

- The formal parameters of a method are simply acts as local variables of that method hence it is possible to declare formal parameters as final.
- If we declare formal parameters as final then we can't change its value within the method.

Example:

```
class Test{
    public static void main(String args[]){
        methodOne(10,20);
    }
    public static void methodOne(final int x,int y){
        //x=100; _____ → Formal parameters
        y=200;
        System.out.println(x+"...."+y);
    }
}
```

output:
compile time error.
D:\Java>javac Test.java
Test.java:6: final parameter x may not be assigned
x=100;

- For instance and static variables JVM will provide default values but if instance and static declared as final JVM won't provide default value compulsory we should perform initialization whether we are using or not .
- For the local variables JVM won't provide any default values we have to perform explicitly before using that variables , this rule is same whether local variable final or not.

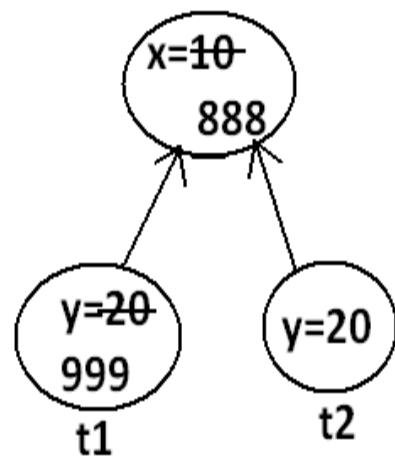
Static modifier:

- Static is the modifier applicable for methods, variables and blocks.
- We can't declare a class with static but inner classes can be declaring as the static.
- In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by all objects of that class.



Example:

```
class Test{
static int x=10;
int y=20;
public static void main(String args[]){
Test t1=new Test();
t1.x=888;
t1.y=999;
Test t2=new Test();
System.out.println(t2.x+"...."+t2.y);
}
}
```



Output:

D:\Java>javac Test.java

D:\Java>java Test

888....20

- Instance variables can be accessed only from instance area directly and we can't access from static area directly.
- But static variables can be accessed from both instance and static areas directly.

```
1) int x = 10;
2) static int x = 10;
3)
4) public void methodOne() {
5)     System.out.println(x);
6) }
7)
8) public static void methodOne() {
9)     System.out.println(x);
10) }
```



Which are the following declarations are allow within the same class simultaneously?

- a) 1 and 3

Example:

```
1) class Test {  
2)     int x = 10;  
3)     public void methodOne() {  
4)         System.out.println(x);  
5)     }  
6) }
```

Output: Compile successfully.

- b) 1 and 4

Example:

```
1) class Test {  
2)     int x = 10;  
3)     public static void methodOne() {  
4)         System.out.println(x);  
5)     }  
6) }
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: non-static variable x cannot be referenced from a static context

System.out.println(x);

- c) 2 and 3

Example:

```
1) class Test {  
2)     static int x = 10;  
3)     public void methodOne() {  
4)         System.out.println(x);  
5)     }  
6) }
```

Output: Compile successfully.

- d) 2 and 4



Example:

```
1) class Test {  
2)     static int x = 10;  
3)     public static void methodOne() {  
4)         System.out.println(x);  
5)     }  
6) }
```

Output: Compile successfully.

e) 1 and 2

Example:

```
1) class Test {  
2)     int x = 10;  
3)     static int x = 10;  
4) }
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:4: x is already defined in Test

static int x=10;

f) 3 and 4

Example:

```
1) class Test {  
2)     public void methodOne() {  
3)         System.out.println(x);  
4)     }  
5)     public static void methodOne() {  
6)         System.out.println(x);  
7)     }  
8) }
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: methodOne() is already defined in Test

public static void methodOne(){



For static methods implementation should be available but for abstract methods implementation is not available hence static abstract combination is illegal for methods.

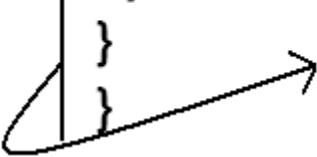
Case 1:

Overloading concept is applicable for static method including main method also. But JVM will always call String[] args main method .

The other overloaded method we have to call explicitly then it will be executed just like a normal method call .

Example:

```
class Test{
    public static void main(String args[]){
        System.out.println("String() method is called");
    }
    public static void main(int args[]){
        System.out.println("int() method is called");
    }
}
```

 This method we have to call explicitly.

Output: String() method is called

Case 2:

Inheritance concept is applicable for static methods including main() method hence while executing child class, if the child doesn't contain main() method then the parent class main method will be executed.

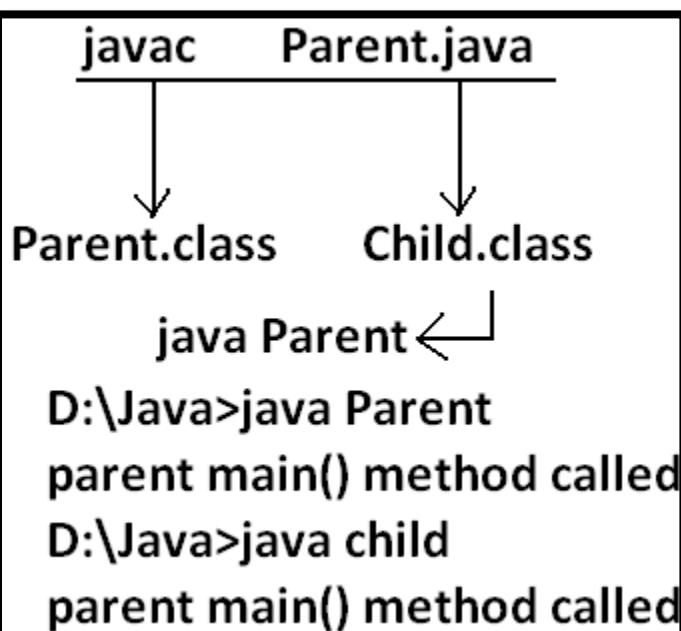
Example:

```
1) class Parent {
2)     public static void main(String args[]) {
3)         System.out.println("parent main() method called");
4)     }
5) }
6) class child extends Parent {
```



7) }

Output:



Example:

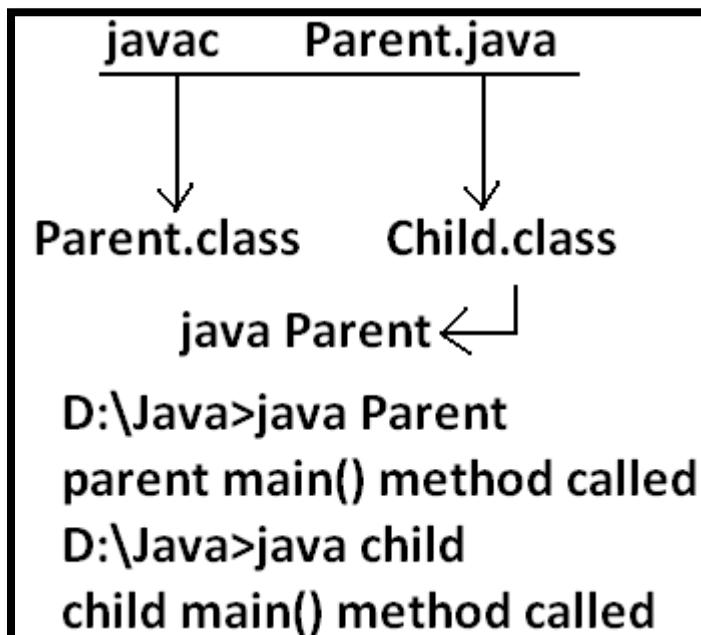
```
class Parent{
    public static void main(String args[]){
        System.out.println("parent main() method called");
    }
}
class child extends Parent{
    public static void main(String args[]){
        System.out.println("child main() method called");
    }
}
```

it is not overriding but method hiding.

A code example showing two classes, Parent and child. The Parent class has a main method that prints "parent main() method called". The child class also has a main method that prints "child main() method called". A bracket on the left side of the code groups the Parent class definition, and an arrow points from the end of the Parent class definition to a box containing the text "it is not overriding but method hiding.".



Output:



It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

Native modifier:

- Native is a modifier applicable only for methods but not for variables and classes.
- The methods which are implemented in non java are called native methods or foreign methods.

The main objectives of native keyword are:

- To improve performance of the system.
- To use already existing legacy non-java code.
- To achieve machine level communication(memory level - address)
- Pseudo code to use native keyword in java.



To use native keyword:

Pseudo code:

```
class Native{
    static
    {
        1)load native library.
        System.loadLibrary("Native library");
    }

    2)native method declaration.←public native void methodOne();
    }

    class Client
    {
        public static void main(String args[]){
            Native n=new Native();

            3) invoke a native method.←n.methodOne();
        }
    }
}
```

- For native methods implementation is already available and we are not responsible to provide implementation hence native method declaration should compulsory ends with semicolon.
 - public native void methodOne();---invalid
 - public native void methodOne();---valid
- For native methods implementation is already available where as for abstract methods implementation should not be available child class is responsible to provide that, hence abstract native combination is illegal for methods.
- We can't declare a native method as strictfp because there is no guaranty whether the old language supports IEEE754 standard or not. That is native strictfp combination is illegal for methods.
- For native methods inheritance, overriding and overloading concepts are applicable.
- The main advantage of native keyword is performance will be improves.
- The main disadvantage of native keyword is usage of native keyword in Java breaks platform independent nature of Java language.



Synchronized:

1. Synchronized is the modifier applicable for methods and blocks but not for variables and classes.
2. If a method or block declared with synchronized keyword then at a time only one thread is allowed to execute that method or block on the given object.
3. The main advantage of synchronized keyword is we can resolve data inconsistency problems.
4. But the main disadvantage is it increases waiting time of the threads and affects performance of the system. Hence if there is no specific requirement never recommended to use synchronized keyword.

For synchronized methods compulsory implementation should be available, but for abstract methods implementation won't be available, Hence abstract - synchronized combination is illegal for methods.

Transient modifier:

- 1) Transient is the modifier applicable only for variables but not for methods and classes.
- 2) At the time of serialization if we don't want to serialize the value of a particular variable to meet the security constraints then we should declare that variable with transient modifier.
- 3) At the time of serialization jvm ignores the original value of the transient variable and save default value that is transient means "not to serialize".
- 4) Static variables are not part of object state hence serialization concept is not applicable for static variables duo to this declaring a static variable as transient there is no use.
- 5) Final variables will be participated into serialization directly by their values due to this declaring a final variable as transient there is no impact.

Volatile modifier:

- 1) Volatile is the modifier applicable only for variables but not for classes and methods.
- 2) If the value of variable keeps on changing such type of variables we have to declare with volatile modifier.
- 3) If a variable declared as volatile then for every thread a separate local copy will be created by the jvm, all intermediate modifications performed by the thread will take place in the local copy instead of master copy.
- 4) Once the value got finalized before terminating the thread that final value will be updated in master copy.
- 5) The main advantage of volatile modifier is we can resolve data inconsistency problems, but creating and maintaining a separate copy for every thread increases complexity of the Programming and affects performance of the system. Hence if there is no specific requirement never recommended to use volatile modifier and it's almost outdated.
- 6) Volatile means the value keep on changing whereas final means the value never changes hence final volatile combination is illegal for variables.



Summary of Modifier:

Modifier	Inner Classes	Outer Classes	Methods	Variables	Blocks	Interfaces	Enum	Constructors
Public	✓	✓	✓	✓	✗	✓	✓	✓
Private	✗	✓	✓	✓	✗	✗	✗	✓
Protected	✗	✓	✓	✓	✗	✗	✗	✓
Default	✓	✓	✓	✓	✗	✓	✓	✓
Final	✓	✓	✓	✓	✗	✗	✗	✗
Abstract	✓	✓	✓	✗	✗	✓	✗	✗
Strictfp	✓	✓	✓	✗	✗	✓	✓	✗
Static	✗	✓	✓	✓	✓	✗	✗	✗
Synchronized	✗	✗	✓	✗	✓	✗	✗	✗
Native	✗	✗	✓	✗	✗	✗	✗	✗
Transient	✗	✗	✗	✓	✗	✗	✗	✗
Volatile	✗	✗	✗	✓	✗	✗	✗	✗

Note:

The modifiers which are applicable for inner classes but not for outer classes are private, protected, static.

The modifiers which are applicable only for methods native.

The modifiers which are applicable only for variables transient and volatile.

The modifiers which are applicable for constructor public, private, protected, default.

The only applicable modifier for local variables is final.

The modifiers which are applicable for classes but not for enums are final , abstract.

The modifiers which are applicable for classes but not for interface are final.

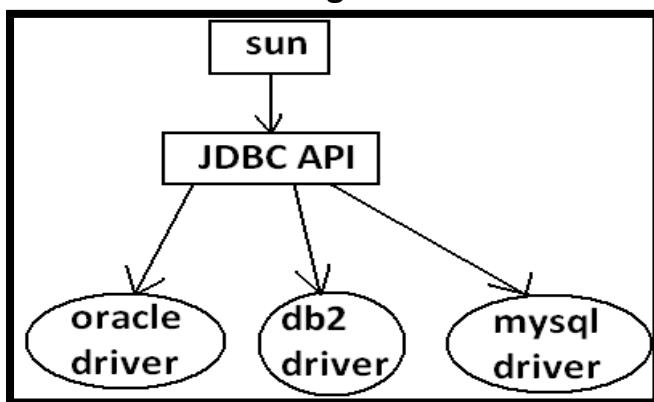


Interfaces

Def1: Any service requirement specification (srs) is called an interface.

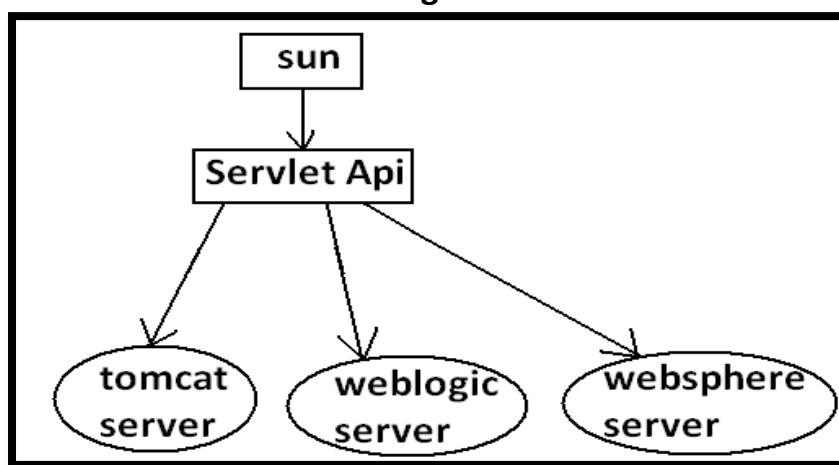
Example1: Sun people responsible to define JDBC API and database vendor will provide implementation for that.

Diagram



Example2: Sun people define Servlet API to develop web applications web server vendor is responsible to provide implementation.

Diagram



Def 2: From the client point of view an interface define the set of services what is expecting. From the service provider point of view an interface defines the set of services what is offering. Hence an interface is considered as a contract between client and service provider.



Example: ATM GUI screen describes the set of services what bank people offering, at the same time the same GUI screen the set of services what customer is expecting hence this GUI screen acts as a contract between bank and customer.

Def3: Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Summery def: Any service requirement specification (SRS) or any contract between client and service provider or 100% pure abstract classes is considered as an interface.

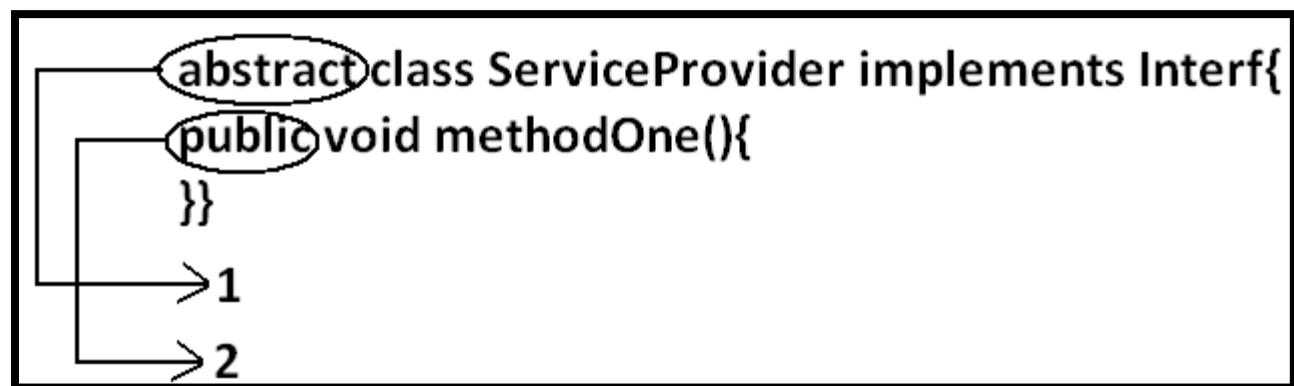
Declaration and implementation of an interface:

Note 1: Whenever we are implementing an interface compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract in that case child class is responsible to provide implementation for remaining methods.

Note 2: Whenever we are implementing an interface method compulsory it should be declared as public otherwise we will get compile time error.

Example:

```
1) interface Interf {  
2)     void methodOne();  
3)     void methodTwo();  
4) }
```



```
1) class SubServiceProvider extends ServiceProvider {  
2) }
```

Output:

Compile time error.

D:\Java>javac SubServiceProvider.java

SubServiceProvider.java:1:



SubServiceProvider is not abstract and does not override abstract method methodTwo() in Interf
class SubServiceProvider extends ServiceProvider

Extends vs implements:

A class can extends only one class at a time.

Example:

```
1) class One {  
2)     public void methodOne() {  
3)     }  
4) }  
5) class Two extends One {  
6) }
```

A class can implements any no. of interfaces at a time.

Example:

```
1) interface One {  
2)     public void methodOne();  
3) }  
4) interface Two {  
5)     public void methodTwo();  
6) }  
7) class Three implements One,Two {  
8)     public void methodOne() {  
9)     }  
10)    public void methodTwo() {  
11)    }  
12) }
```

A class can extend a class and can implement any no. of interfaces simultaneously.

```
1) interface One {  
2)     void methodOne();  
3) }  
4) class Two {  
5)     public void methodTwo() {  
6)     }  
7) }  
8) class Three extends Two implements One {  
9)     public void methodOne() {  
10)    }
```



11) }

An interface can extend any no. of interfaces at a time.

Example:

```
1) interface One {  
2)     void methodOne();  
3) }  
4) interface Two {  
5)     void methodTwo();  
6) }  
7) interface Three extends One, Two {  
8) }
```

Which of the following is true?

1. A class can extend any no. of classes at a time.
2. An interface can extend only one interface at a time.
3. A class can implement only one interface at a time.
4. A class can extend a class and can implement an interface but not both simultaneously.
5. An interface can implement any no. of interfaces at a time.
6. None of the above.

Ans: 6

Consider the expression X extends Y for which of the possibility of X and Y this expression is true?

1. Both x and y should be classes.
2. Both x and y should be interfaces.
3. Both x and y can be classes or can be interfaces.
4. No restriction.

Ans: 3

X extends Y, Z ?

X, Y, Z should be interfaces.

X extends Y implements Z ?

X, Y should be classes.

Z should be interface.

X implements Y, Z ?

X should be class.

Y, Z should be interfaces.



X implements Y extend Z ?

Example:

```
1) interface One {  
2) }  
3) class Two {  
4) }  
5) class Three implements One extends Two {  
6) }
```

Output:

Compile time error.

D:\Java>javac Three.java

Three.java:5: '{' expected

class Three implements One extends Two{

Interface methods:

Every method present inside interface is always public and abstract whether we are declaring or not. Hence inside interface the following method declarations are equal.

```
void methodOne();  
public Void methodOne();  
abstract Void methodOne();           Equal  
public abstract Void methodOne();
```

public: To make this method available for every implementation class.

abstract: Implementation class is responsible to provide implementation .

As every interface method is always public and abstract we can't use the following modifiers for interface methods.

Private, protected, final, static, synchronized, native, strictfp.

Inside interface which method declarations are valid?

1. public void methodOne(){}
 2. private void methodOne();
 3. public final void methodOne();
 4. public static void methodOne();
 5. public abstract void methodOne();

Ans: 5



Interface variables:

- An interface can contain variables
- The main purpose of interface variables is to define requirement level constants.
- Every interface variable is always public static and final whether we are declaring or not.

Example:

```
1) interface interf {  
2)     int x = 10;  
3) }
```

public: To make it available for every implementation class.

static: Without existing object also we have to access this variable.

final: Implementation class can access this value but cannot modify.

Hence inside interface the following declarations are equal.

```
int x=10;  
public int x=10;  
static int x=10;  
final int x=10;           Equal  
public static int x=10;  
public final int x=10;  
static final int x=10;  
public static final int x=10;
```

- As every interface variable by default public static final we can't declare with the following modifiers.
 - Private
 - Protected
 - Transient
 - Volatile
- For the interface variables compulsory we should perform initialization at the time of declaration only otherwise we will get compile time error.

Example:

```
1) interface Interf {  
2)     int x;  
3) }
```



Output:

Compile time error.

D:\Java>javac Interf.java
Interf.java:3: = expected
int x;

Which of the following declarations are valid inside interface?

- 1) int x;
- 2) private int x=10;
- 3) public volatile int x=10;
- 4) public transient int x=10;
- 5) public static final int x=10;

Ans: 5

Interface variables can be access from implementation class but cannot be modified.

Example:

```
1) interface Interf {  
2)     int x = 10;  
3) }
```

Example 1:

```
class Test implements Interf  
{  
    public static void main(String args[]){
```

x=20;

System.out.println("value of x"+x);

}

}

output:

compile time error.

D:\Java>javac Test.java

Test.java:4: cannot assign a value to final variable x
x=20;



Example 2:

```
1) class Test implements Interf {  
2)     public static void main(String args[]) {  
3)         int x = 20;  
4)         //Here we declaring the variable x.  
5)         System.out.println(x);  
6)     }  
7) }
```

Output:

```
D:\Java>javac Test.java  
D:\Java>java Test  
20
```

Interface naming conflicts

Method naming conflicts:

Case 1:

If two interfaces contain a method with same signature and same return type in the implementation class only one method implementation is enough.

Example 1:

```
1) interface Left {  
2)     public void methodOne();  
3) }
```

Example 2:

```
1) interface Right {  
2)     public void methodOne();  
3) }
```

Example 3:

```
1) class Test implements Left, Right {  
2)     public void methodOne() {  
3)     }  
4) }
```



Output:

D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java

Case 2:

If two interfaces contain a method with same name but different arguments in the implementation class we have to provide implementation for both methods and these methods acts as a overloaded methods

Example 1:

```
1) interface Left {  
2)     public void methodOne();  
3) }
```

Example 2:

```
1) interface Right {  
2)     public void methodOne(int i);  
3) }
```

Example 3:

```
1) class Test implements Left, Right {  
2)     public void methodOne() {  
3)     }  
4)     public void methodOne(int i) {  
5)     }  
6) }
```

Output:

D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java

Case 3:

If two interfaces contain a method with same signature but different return types then it is not possible to implement both interfaces simultaneously.

Example 1:

```
1) interface Left {  
2)     public void methodOne();
```



3) }

Example 2:

```
1) interface Right {  
2)     public int methodOne(int i);  
3) }
```

We can't write any java class that implements both interfaces simultaneously.

Is a Java class can implement any no. of interfaces simultaneously?

Yes, except if two interfaces contains a method with same signature but different return types.

Variable naming conflicts:

Two interfaces can contain a variable with the same name and there may be a chance variable naming conflicts but we can resolve variable naming conflicts by using interface names.

Example 1:

```
1) interface Left {  
2)     int x = 888;  
3) }
```

Example 2:

```
1) interface Right {  
2)     int x = 999;  
3) }
```

Example 3:

```
1) class Test implements Left, Right {  
2)     public static void main(String args[]) {  
3)         //System.out.println(x);  
4)         System.out.println(Left.x);  
5)         System.out.println(Right.x);  
6)     }  
7) }
```

Output:

D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java



D:\Java>java Test
888
999

Marker interface:

If an interface doesn't contain any methods and by implementing that interface if our objects will get some ability such type of interfaces are called Marker interface (or) Tag interface (or) Ability interface.

Example:

Serializable
Cloneable
RandomAccess These are marked for some ability
SingleThreadModel
.
.
.

Example 1:

By implementing Serializable interface we can send that object across the network and we can save state of an object into a file.

Example 2:

By implementing SingleThreadModel interface Servlet can process only one client request at a time so that we can get "Thread Safety".

Example 3:

By implementing Cloneable interface our object is in a position to provide exactly duplicate cloned object.

Without having any methods in marker interface how objects will get ability?
Internally JVM is responsible to provide required ability.

Why JVM is providing the required ability in marker interfaces?

To reduce complexity of the programming.

Is it possible to create our own marker interface?

Yes, but customization of JVM must be required.

Eg: Sleepable , Jumpable ,



Adapter class:

- Adapter class is a simple java class that implements an interface only with empty implementation for every method.
- If we implement an interface directly for each and every method compulsory we should provide implementation whether it is required or not. This approach increases length of the code and reduces readability.

Example 1:

```
1) interface X {  
2)     void m1();  
3)     void m2();  
4)     void m3();  
5)     void m4();  
6)     //.  
7)     //.  
8)     //.  
9)     //.  
10)    void m5();  
11) }
```

Example 2:

```
1) class Test implements X {  
2)     public void m3() {  
3)         System.out.println("m3() method is called");  
4)     }  
5)     public void m1() {}  
6)     public void m2() {}  
7)     public void m4() {}  
8)     public void m5() {}  
9) }
```

- We can resolve this problem by using adapter class.
- Instead of implementing an interface if we can extend adapter class we have to provide implementation only for required methods but not for all methods of that interface.
- This approach decreases length of the code and improves readability.

Example 1:

```
1) abstract class AdapterX implements X {  
2)     public void m1() {}  
3)     public void m2() {}  
4)     public void m3() {}  
5)     public void m4() {}  
6)     //.
```

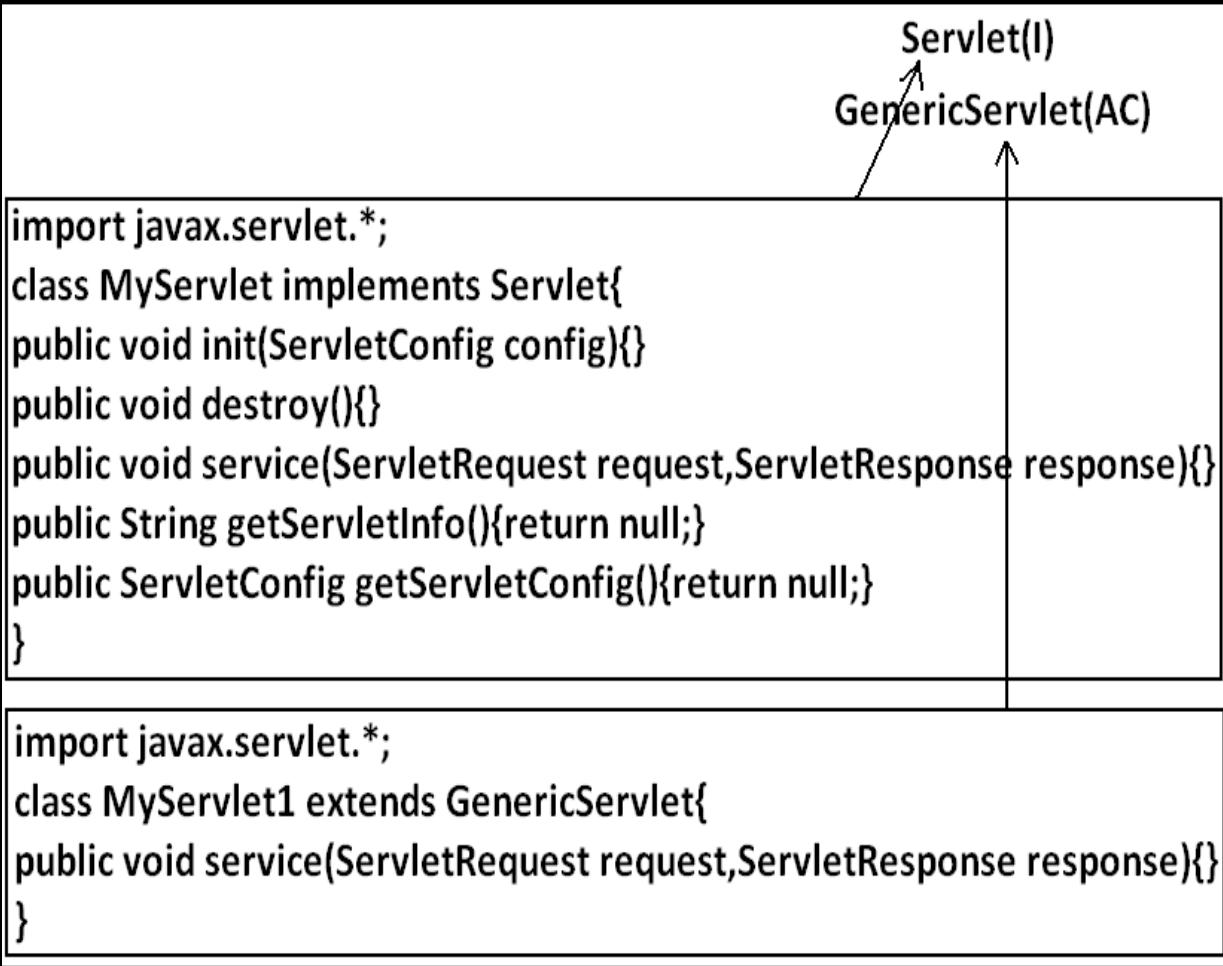


```
7)      //.
8)      //.
9)  public void m1000(){}
10) }
```

Example 2:

```
1) public class Test extend AdapterX {
2)   public void m3(){}
3) }
4) }
```

Example:



Generic Servlet simply acts as an adapter class for Servlet interface.

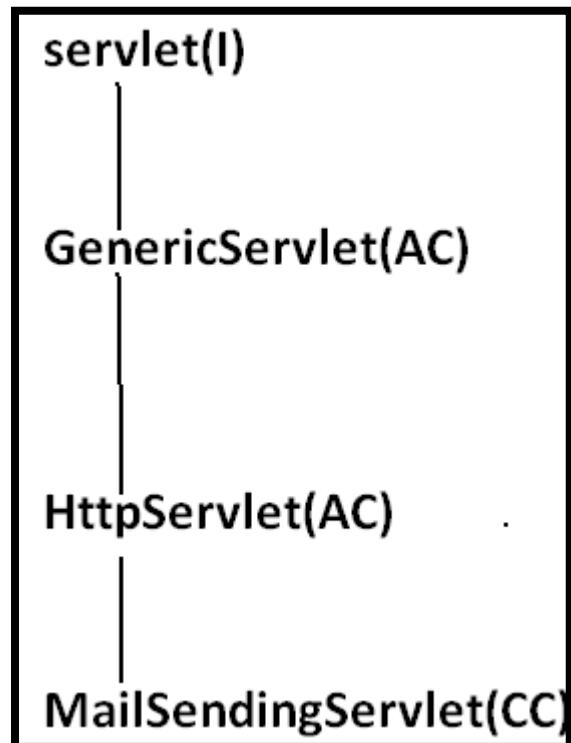
Note: Marker Interface and Adapter class are big utilities to the programmer to simplify programming.



**What is the difference between interface, abstract class and concrete class?
When we should go for interface, abstract class and concrete class?**

- If we don't know anything about implementation just we have requirement specification then we should go for interface.
- If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
- If we are talking about implementation completely and ready to provide service then we should go for concrete class.

Example:





What is the Difference between interface and abstract class ?

Interface	Abstract class
If we don't know anything about implementation just we have requirement specification then we should go for interface.	If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
Every method present inside interface is always public and abstract whether we are declaring or not.	Every method present inside abstract class need not be public and abstract.
We can't declare interface methods with the modifiers private, protected, final, static, synchronized, native, strictfp.	There are no restrictions on abstract class method modifiers.
Every interface variable is always public static final whether we are declaring or not.	Every abstract class variable need not be public static final.
Every interface variable is always public static final we can't declare with the following modifiers. Private, protected, transient, volatile.	There are no restrictions on abstract class variable modifiers.
For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	It is not require performing initialization for abstract class variables at the time of declaration.
Inside interface we can't take static and instance blocks.	Inside abstract class we can take both static and instance blocks.
Inside interface we can't take constructor.	Inside abstract class we can take constructor.

We can't create object for abstract class but abstract class can contain constructor what is the need?

abstract class constructor will be executed when ever we are creating child class object to perform initialization of child object.

Example:

```
1) class Parent {  
2)     Parent() {  
3)         System.out.println(this.hashCode());  
4)     }  
5) }  
6) class child extends Parent {  
7)     child() {  
8)         System.out.println(this.hashCode());  
9)     }  
10}
```



```
10) }
11) class Test {
12)     public static void main(String args[]) {
13)         child c = new child();
14)         System.out.println(c.hashCode());
15)     }
16) }
```

Note: We can't create object for abstract class either directly or indirectly.

Every method present inside interface is abstract but in abstract class also we can take only abstract methods then what is the need of interface concept?

We can replace interface concept with abstract class. But it is not a good programming practice. We are misusing the roll of abstract class. It may create performance problems also. (This is just like recruiting IAS officer for sweeping purpose)

Example :

<pre>interface X { ----- ----- } class Test implements X { ----- ----- }</pre>	<pre>abstract class X { ----- ----- } class Test extends X { ----- ----- }</pre>
<pre>Test t=new Test(); //takes 2 sec 1) performance is high 2) while implementing X we can extend some other classes</pre>	<pre>Test t=new Test(); //takes 20sec 1) performance is low 2) while extending X we can't extend any other classes</pre>

If every thing is abstract then it is recommended to go for interface.

Why abstract class can contain constructor where as interface doesn't contain constructor?

The main purpose of constructor is to perform initialization of an object i.e., provide values for the instance variables. Inside interface every variable is always static and there is no chance of existing instance variables. Hence constructor is not required for interface.

But abstract class can contain instance variable which are required for the child object to perform initialization for those instance variables constructor is required in the case of abstract class.



Java Source File Structure OCJA Practice Questions

Q1. Given the code fragment from 3 files

SalesMan.java:

```
package sales;  
public class SalesMan{}
```

Product.java:

```
package sales.products;  
public class Product{}
```

Market.java:

```
1) package market;  
2) //Line-1  
3) public class Market  
4) {  
5)     SalesMan sm;  
6)     Product p;  
7) }
```

Which code fragment when inserted at line 2, enables the code to compile?

- A) import sales.*;
- B) import java.sales.products.*;
- C) import sales;
import sales.products;
- D) import sales.*;
import products.*;
- E) import sales.*;
import sales.products.*;

Answer : E

Note: Whenever we are importing a package, all classes and interfaces present in that package are by default available but not sub package classes. Hence to use sub package class, compulsory we should write import statement until sub package level.



Q2. Consider the code

```
1) package pack1;
2) public class A
3) {
4)     int p;
5)     private int q;
6)     protected int r;
7)     public int s;
8) }
```

Test.java:

```
1) package pack2;
2) import p1.A;
3) public class Test extends A
4) {
5)     public static void main(String[] args)
6)     {
7)         A obj= new Test();
8)     }
9) }
```

Which statement is true?

- A) By using obj we can access p and s
- B) By using obj we can access only s
- C) By using obj we can access r and s
- D) By using obj we can access p,r and s

Answer: B

Q3. Which of the following code fragments are valid?

A)

```
1) public abstract class Test
2) {
3)     public void m1();
4)     public void m2();
5) }
```

B)

```
1) public abstract class Test
2) {
3)     public abstract void m1();
```



4) public void m2();
5) }

C)

1) public abstract class Test
2) {
3) public abstract void m1();
4) public void m2(){}
5) }

D)

1) public abstract class Test
2) {
3) public abstract void m1(){}
4) public abstract void m2(){}
5) }

Answer: C

Q4. You are asked to develop a program for a shopping application, and you are given the following information:

The application must contains the classes Book, JavaBook and PythonBook. The Book class is the super class of other 2 classes.

The int calculatePrice(Book b) method calculates the price of the Book.
The void printBook(Book b) method prints the details of the Book.

Which definition of the Book class adds a valid layer of abstraction to the class hierarchy?

A)

1) public abstract class Book
2) {
3) public abstract int calculatePrice(Book b);
4) public void printBook(Book b){}
5) }

B)

1) public abstract class Book
2) {
3) public int calculatePrice(Book b);
4) public void printBook(Book b);
5) }



C)

```
1) public abstract class Book
2) {
3)     public int calculatePrice(Book b);
4)     public final void printBook(Book b){}
5) }
```

D)

```
1) public abstract class Book
2) {
3)     public abstract int calculatePrice(Book b){}
4)     public abstract void printBook(Book b){}
5) }
```

Answer: A

Q5. Consider the code

```
1) interface Interf
2) {
3)     public void m1();
4)     public void m2();
5) }
6) class A implements Interf
7) {
8)     public void m1(){}
9) }
```

Which of the following changes individually will compile the code successfully?

- A) insert public void m2(){} inside class A
- B) declare class A as abstract
- C) insert public void m2(); inside class A
- D) No Changes are required

Answer: A and B

Q6. Consider the code

```
1) interface Writable
2) {
3)     public void writeBook();
4)     public void setBookMark();
5) }
```



```
6) abstract class Book implements Writable //Line-1
7) {
8)     public void writeBook(){}
9)     //Line-2
10) }
11) class EBook extends Book //Line-3
12) {
13)     public void writeBook(){}
14)     //Line-4
15) }
```

And given the code Fragment:

```
Book b1= new EBook();
b1.writeBook();
```

Which option enables the code to compile?

- A) Replace the code fragment at Line-3 with :
abstract class EBook extends Book
- B) Replace the code fragment at Line-1 with :
class Book implements Writable
- C) At Line-2 insert
public abstract void setBookMark();
- D) At Line-4 insert:
public void setBookMark(){}

Answer : D

Q7. Given the content of 3 files

X.java

```
1) public class X
2) {
3)     public void a(){}
4)     int a;
5) }
```

Y.java

```
1) public class Y
2) {
3)     private int doStuff()
4) {
```



```
5)     private int i =100;
6)     return i++;
7)   }
8) }
```

Z.java

```
1) import java.io.*;
2) package pack1;
3) class Z
4) {
5)   public static void main(String[] args)throws IOException
6)   {
7)   }
8) }
```

Which Statement is true?

- A) Only X.java file compiles successfully
- B) Only Y.java file compiles successfully
- C) Only Z.java file compiles successfully
- D) Only X.java and Y.java files compile successfully
- E) Only Y.java and Z.java files compile successfully
- F) Only X.java and Z.java files compile successfully

Answer: A

Q8. Given the code fragments:

A.java

```
1) package pack1;
2) public class A
3) {
4) }
```

B.java

```
1) package pack1.pack2;
2) //Line-1
3) public class B
4) {
5)   public void m1()
6)   {
7)     A a = new A();
8)   }
9) }
```



C.java

```
1) package pack3;
2) //Line-2
3) public class C
4) {
5)     public static void main(String[] args)
6)     {
7)         A a = new A();
8)         B b = new B();
9)     }
10) }
```

Which modifications enables the code to compile?

A) Replace Line-1 with:

```
import pack1.A;
```

Replace Line-2 with:

```
import pack1.A;
import pack1.pack2.B;
```

B) Replace Line-1 with:

```
import pack1;
```

Replace Line-2 with:

```
import pack1;
import pack1.pack2;
```

C) Replace Line-1 with:

```
import pack1.A;
```

Replace Line-2 with:

```
import pack1.*;
```

D) Replace Line-1 with:

```
import pack1.*;
```

Replace Line-2 with:

```
import pack1.pack2.*;
```

Answer: A



OOPs And Constructors

- 5.1) Data Hiding
- 5.2) Abstraction
- 5.3) Encapsulation
- 5.4) Tightly Encapsulated Class
- 5.5) Is- A Relationship
- 5.6) Has-A Relationship
- 5.7) Method Signature
- 5.8) Overloading
- 5.9) Overriding
- 5.10) Method Hiding
- 5.11) Polymorphism
- 5.12) Static Control Flow
- 5.13) Instance Control Flow
- 5.15) Constructors
- 5.16) Coupling
- 5.17) Cohesion
- 5.18) Object Type Casting



Data Hiding:

Our Internal Data should Not go out Directly OR Outside Person can't Access Our Internal Data Directly. This is the Concept of Data Hiding.

Eg: After Providing Proper User Name and Password Only we can able to Access Our Mail Information.

Eg: After swiping ATM Card and Providing Valid Pin Number we can able to Access Our Account Information.

By declaring Data Member as private we can achieve Data Hiding.

```
class Account {  
    private double balance;  
    .....  
    .....  
}
```

The Main Advantage of Data Hiding is Security.

Note: Recommended Modifier for Data Members is private.

Abstraction:

Hiding Internal Implementation and Highlight the Set of Services which are offering is the Concept of Abstraction.

Eg: By using Bank ATM GUI Screen Bank People are highlighting the Set of Services what they offering without highlighting Internal Implementation.

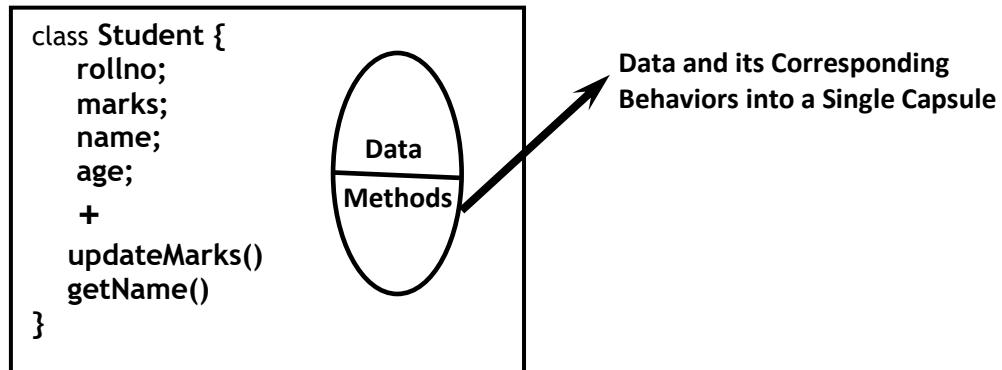
The Main Advantages of Abstraction are

- We can Achieve Security as we are not highlighting Our Internal Implementation.
- Without effecting Outside Person we can able to Perform any Type of Changes in Our Internal Design. Hence Enhancement will become Easy.
- It Improves Maintainability and Modularity of the Application.



Encapsulation:

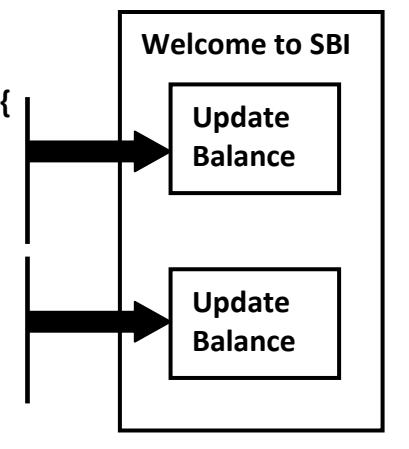
The Process of Binding Data and Corresponding Methods into a Single Unit is Called Encapsulation.



In any Component follows Data Hiding and Abstraction Such Type of Component is Called Encapsulated Component.

Encapsulation = Data Hiding + Abstraction

```
public class Account {  
    public double balance;  
    public void setBalance(double balance) {  
        //Validation Logic  
        this.balance = balance;  
    }  
    public double getBalance() {  
        //Validation Logic  
        return balance;  
    }  
    .....  
    .....  
}
```



The Main Advantages of Encapsulation are

- We can Achieve Security.
- Enhancement will become Very Easy.
- It Improves Maintainability and Modularity of the Application.

The Main Advantage of Encapsulation is we can Achieve Security and the Main Disadvantage of Encapsulation is it Increases Length of the Code and Slow Down Execution. So that Performance will be Impacted.



Tightly Encapsulated Class:

- A Class is Said to be Tightly Encapsulated if and only if Each and Every Variable of that Class declared as private.
- Whether the Class contains Getter and Setter Methods OR Not and whether these Methods are declared as public OR Not. These things are not required to Check.

```
public class Account {  
    private double balance;  
    public double getBalance() {  
        return balance;  
    }  
}
```

Which of the following Classes are Tightly Encapsulated?

```
class A {  
    private int x = 10;  
}
```

```
class B extends A {  
    int y = 10;  
}
```

```
class C extends A {  
    private int z = 20;  
}
```

Class A and Class C are Said to be Tightly Encapsulated Classes

Which of the following Classes are Tightly Encapsulated?

```
class A {  
    int x = 10;  
}
```

```
class B extends A {  
    private int y = 20;  
}
```

```
class C extends B {  
    private int z = 30;  
}
```

No Class is Tightly Encapsulated Classes.

Note: If the Parent Class is Not Tightly Encapsulated then No Child Class is Tightly Encapsulated.



IS-A Relationship (Inheritance):

- By using extends Key Word we can implement IS-A Relationship.
- The Main Advantage of Inheritance is Re-usability of the Code.

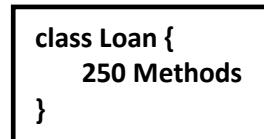
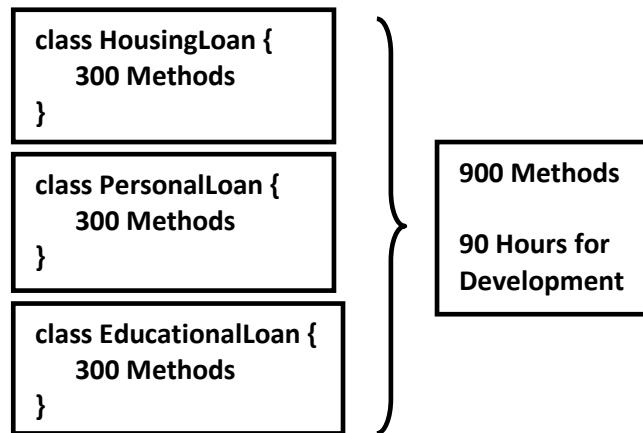
```
class P {  
    public void m1() {}  
}  
  
class C extends P {  
    public void m2() {}  
}  
  
class Test {  
    public static void main (String [] args) {  
  
        //Case 1  
        P p = new P();  
        p.m1();  
        p.m2(); cannot find symbol  
            symbol: method m2()  
            location: variable p of type P  
  
        //Case 2  
        C c = new C();  
        c.m1();  
        c.m2();  
  
        //Case 3  
        P p = new C();  
        p.m1();  
        p.m2(); cannot find symbol  
            symbol: method m2()  
            location: variable p of type P  
  
        //Case 4  
        C c = new P(); error: incompatible types  
            required: C  
            found: P  
    }  
}
```



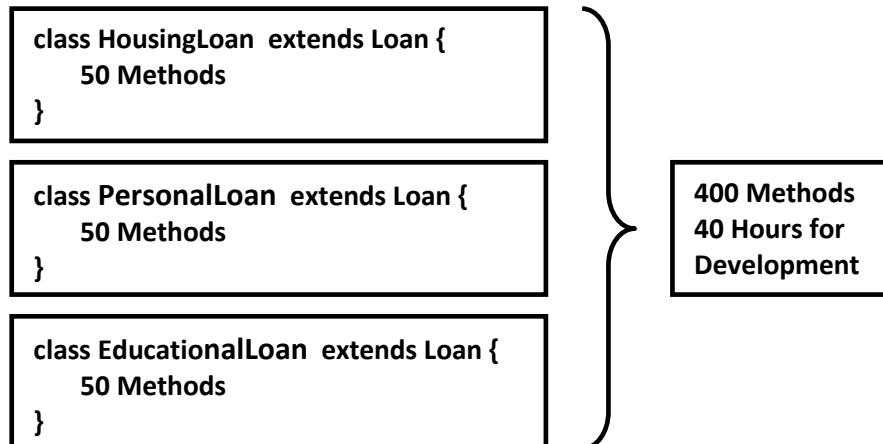
Conclusions:

- 1) Whatever Methods Parent has by Default Available to the Child. Hence on Child Class Object we can Call Both Parent and Child Class Methods.
- 2) Whatever Methods Child has by Default Not Available to the Parent and Hence on the Parent Class Reference we can't Call Child Specific Methods.
- 3) Parent Reference can be used to hold Child Object. But by using that Reference we can Call Only Methods Available in Parent Class and we can't Call Child Specific Methods.
- 4) Child Reference cannot be used to hold Parent Object. But Parent Reference can be used to hold Child Object.
- 5) Parent Class contains the Common Functionality which required for Child Class. Whereas Child Class contains Specific Functionality.

Without Inheritance

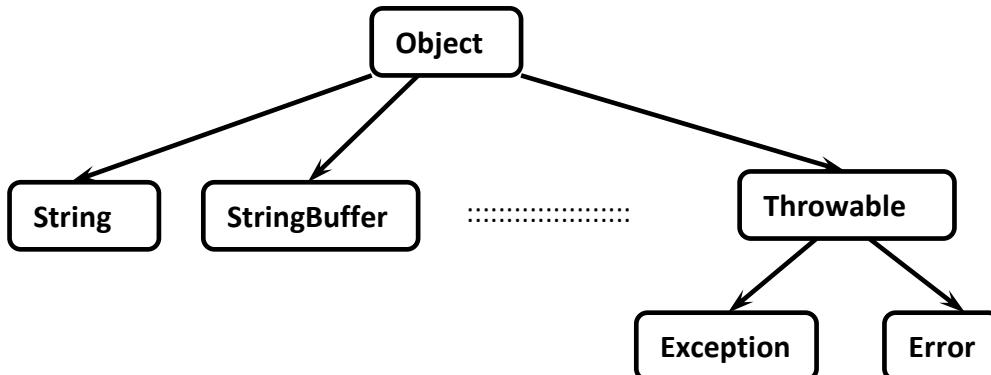


With Inheritance





Note: Total Java API is implemented by using Inheritance Concept Only.



- The Most Common Methods which are required for All Java Classes are defined Inside Object Class. Hence Object Class Acts as Root for All Java Classes.
- The Most Common Methods which are required for All Exceptions and Errors are defined in Throwable Class. Hence Throwable Class Acts as Root for Exception Hierarchy.

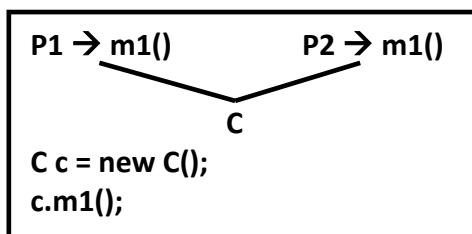
Multiple Inheritance:

A Java Class can't extend More than One Class at-a-Time. Hence Java won't Provide Support for Multiple Inheritance with Respect to Classes.

Eg: class A extends B, C { } X

Why Java won't Provide Support for Multiple Inheritance?

- There May be a Chance of Raising Ambiguity Problems.

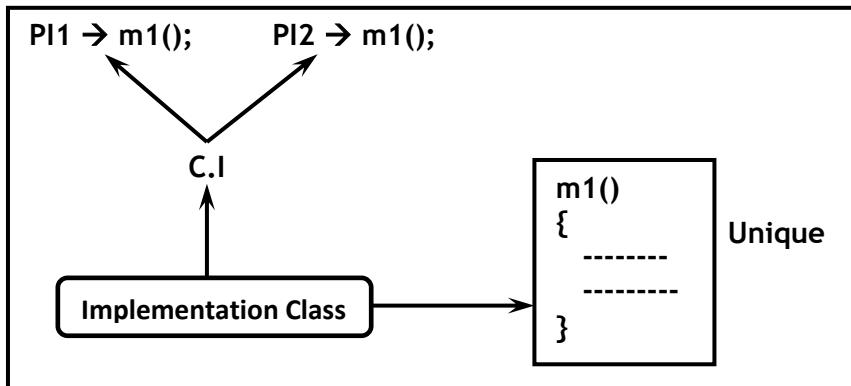


- An Interface can extend any Number of Interfaces at a Time. Hence Java Provides Support for Multiple Inheritance with Respect to Interfaces.

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

Why Ambiguity Problem won't be raised in Interfaces?

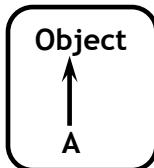
Even though Multiple Method Declarations Present, but Implementation is Unique. Hence there is No Chance of Ambiguity Problem in Interfaces.



Note:

If Our Class doesn't extends Any Other Class then Only it is the Direct Child Class of Object.

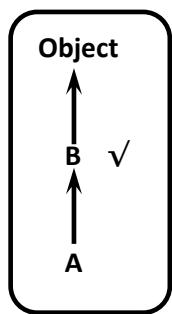
Eg: class A {}



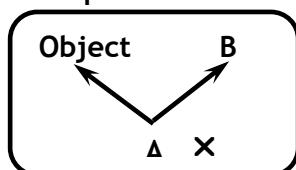
If Our Class extends Any Other Class then it is the In- Direct Child Class of Object.

Eg: class A extends B {}

Multi- Level Inheritance



Multiple Inheritance



Cyclic inheritance:

Cyclic Inheritance is Not allowed in Java.

Eg: class A extends A {}

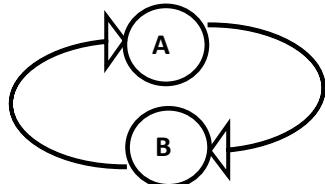
CE: cyclic inheritance involving A



Eg: class A extends B {}

class B extends A {}

CE: cyclic inheritance involving A





HAS-A Relationship:

- It is Also Known as *Composition OR Aggregation*.
- There is No Specific Key Word to implement HAS-A Relationship. But Most of the Times we are depending on *new* Key Word.
- The Main Advantage of HAS-A Relationship is Code Re- Usability.
- The Main Disadvantage is depending between the Components and Creates Maintenance Problems.

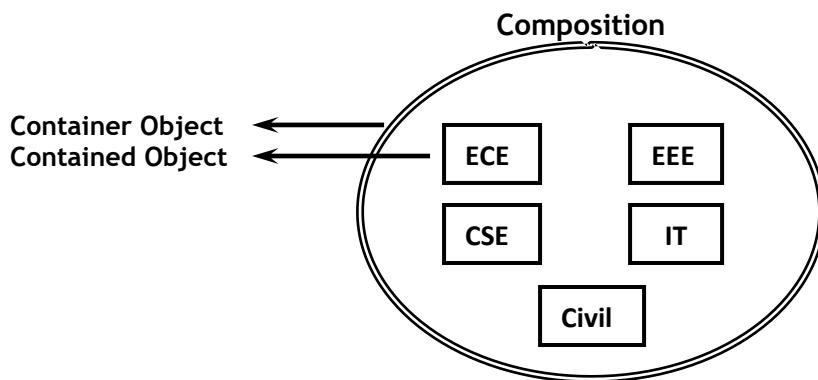
```
class Car {  
    Engine e = new Engine();  
    .  
    .  
} //Class Car has engine Reference  
  
class Engine {  
    //Engine Specific  
    Functionality.  
}
```

Composition Vs Aggregation

Composition

Without Existence of Container Object, if there is No Chance of Existence of Contained Objects then *Container* and *Contained Objects* are Said to be Strongly Associated and this *Strong Association* is known as *Composition*.

Eg: A University has Several Departments. Without Existence of University there is No Chance for Existence Departments Objects. Hence *University* and *Departments* are Strongly Associated and this Strong Association is Known as *Composition*.

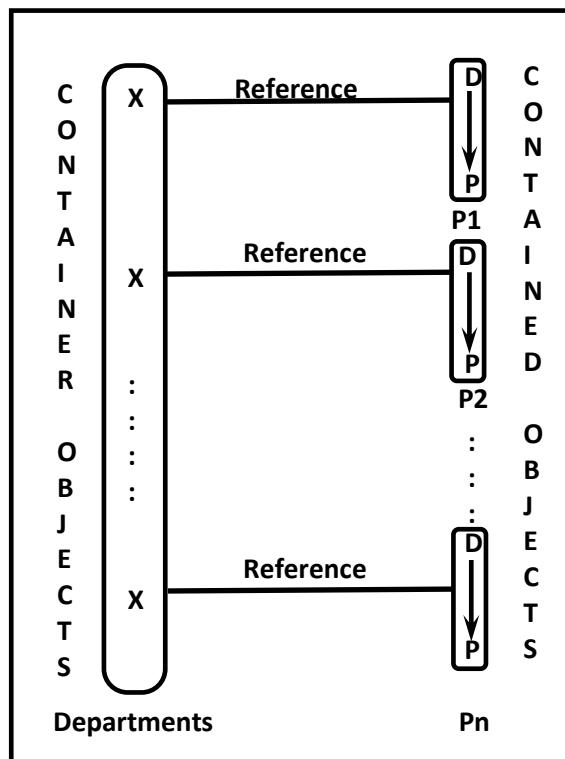


Aggregation

Without Existence of Container Object, if there is a Chance of Existence of Contained Objects then *Container* and *Contained Objects* are Said to be Weakly Associated and this *Loose Association* is known as *Aggregation*.



Eg: Within a Department there May be a Chance of working Several Professors. Without Existence of Departments Object there May be a Chance of existing Professors Object. Hence Department and Professors are Loosely Associated and this *Loose Association* is Known as Aggregation.

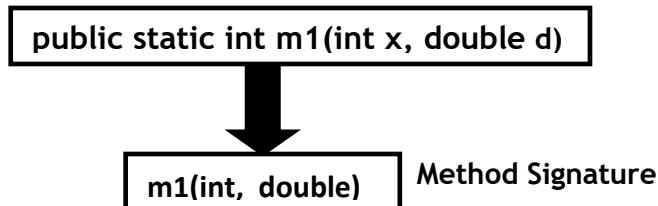


Note:

- In Composition Objects are Strongly Associated whereas in Aggregation Objects are Weakly Associated.
- In Composition Container Object holds Contained Objects Directly whereas in Aggregation Container Object Just Holds References of Contained Objects.

Method Signature:

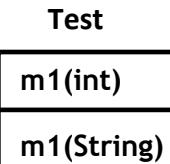
- Method Signature consists of *Method Name* and *Argument Types*.



- Return Type is Not Part of Method Signature in Java.
- Compiler will Use Method Signature while resolving Method Calls.



```
class Test {  
    public void m1(int i) {-----}  
    public void m1(String s) {-----}  
}  
Test t1 = new Test();  
t1.m1(10);  
t1.m1("Durga");  
t1.m1(10.9); //CE: cannot find symbol  
symbol: m1(double)
```



location: class Test → Method Signature

- Within a Class 2 Methods with Same Signature Not allowed. Otherwise we will get Compile Time Error Saying m1() is already defined in Test class.

```
class Test {  
    public void m1(int i) {}  
    public int m1(int x) {  
        return 10;  
    }  
}  
  
Test t1 = new Test();  
t1.m1(10);  
CE: m1(int) is already defined in Test
```

Overloading

- 2 Methods are Said to be Overloaded if and only if Both Methods having Same Name but Different Type of Arguments.
- In C Language Overloading Concept is Not there hence we can't Declare 2 Methods with the Same Name but different Type of Arguments. Hence If there is a Change in Argument Type Compulsory we should go for New Method Name.

Eg: abs(int) → abs (10);
labs(long) → labs (10l);
fabs(float) → fabs (10.5f);

- Lack of Overloading in C Increases the Complexity of the Programming.
- But in Java we can Declare Multiple Methods with the Same Name but with different Arguments Types and these Methods are Called Overloaded Methods.

Eg: abs(int),
abs(long),
abs(float)

Having Overloading Concept in Java Reduces the Complexity of the Programming.



Case 1:

The Overloading Method Resolution is the Responsibility of Compiler based on Reference Type and Method Arguments. Hence Overloading is Considered as *Compile-Time Polymorphism OR Early Binding*.

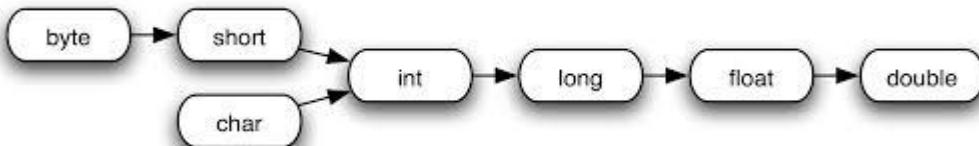
```
class Test {  
    public void m1()—  
    {  
        System.out.println ("no- args");  
    }  
    public void m1(int i)—  
    {  
        System.out.println ("int- args");  
    }  
    public void m1(double d)  
    {  
        System.out.println ("double-args");  
    }  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.m1(); → no-args  
        t.m1(10); → int-args  
        t.m1(10.5); → double-args  
    }  
}
```

Overloaded Methods

Case 2: Automatic Promotion in Overloading

- While Resolving Overloaded Methods if Exact Method with the required Argument is Not Available then the Compiler won't Raise Immediately Compile Time Error.
- First Compiler will Promote Arguments to Next Level and Check is there any Matched Method with Promoted Arguments.
- If the Matched Method is Found then it will Considered Otherwise Compiler Promotes the Argument to the Next Level.
- This Process will be continued until all Possible Promotions.
- After all Possible Promotions still the Compiler Unable to find the Matched Method then it raises Compile Time Error.

The following is the List of all Possible Automatic Promotions in Overloading.





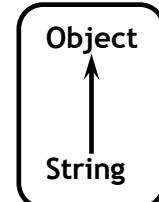
```
class Test {  
    public void m1() {  
        System.out.println ("no-args");  
    }  
  
    public void m1(int i) {  
        System.out.println ("int-args");  
    }  
  
    public void m1(float f) {  
        System.out.println ("float-args");  
    }  
  
    public static void main(String[] args) {  
        Test t = new Test();  
  
        t.m1(); //no-args  
  
        t.m1(10); //int-args  
  
        t.m1(10.9f); //float-args  
  
        t.m1('a'); //int-args  
  
        t.m1(10l); //float-args  
  
        t.m1(10.5); CE: cannot resolve symbol  
                      symbol : method m1 (double)  
                      location: class Test  
    }  
}
```

Case 3:

- In Overloading exact Match will get High Priority.
- In Overloading Child Class Argument will get More Priority than Parent Class Argument.



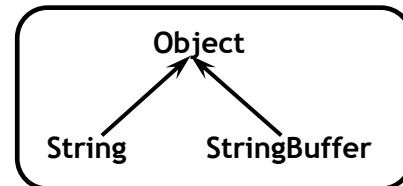
```
class Test {  
    public void m1(String s) {  
        System.out.println ("String Version");  
    }  
    public void m1(Object o) {  
        System.out.println ("Object Version");  
    }  
    public static void main(String arg[]) {  
        Test t = new Test();  
        t.m1("Durga"); //String Version  
        t.m1(new Object()); //Object Version  
        t.m1(null); //String Version  
    }  
}
```



Case 4:

In Java, Method Overloading is Not Possible by Changing the Return Type of the Method because there May Occur Ambiguity Problem.

```
class Test {  
    public void m1(String s) {  
        System.out.println ("String Version");  
    }  
  
    public void m1(StringBuffer sb) {  
        System.out.println ("StringBuffer Version");  
    }  
  
    public static void main(String arg[]) {  
        Test t = new Test();  
        t.m1("Durga"); //String Version  
        t.m1(new StringBuffer("Durga")); //StringBuffer Version  
        t.m1(null); //CE: reference to m1 is ambiguous, both method m1(String) in Test and  
                   //method m1(StringBuffer) in Test match  
    }  
}
```



Case 5: In General var-arg Method will get Least Priority i.e., if No Other Method Matched then Only var-arg Method will get the Chance. It is Exactly Same as *default* Case Inside *switch*.



```
class Test {  
    public void m1(int i) {  
        System.out.println ("General Method");  
    }  
    public void m1(int... i) {  
        System.out.println ("var-arg Method");  
    }  
    public static void main(String arg[]) {  
        Test t = new Test();  
        t.m1(); //var-arg Method  
        t.m1(10, 20); //var-arg Method  
        t.m1(10); //General Method  
    }  
}
```

Case 6:

```
class Test {  
    public void m1(int i, float f) {  
        System.out.println("int - float Version");  
    }  
    public void m1(float f, int i) {  
        System.out.println("float - int Version");  
    }  
    public static void main(String arg[]) {  
        Test t = new Test();  
        t.m1(10.5f, 10); //float - int Version  
        t.m1(10, 10.5f); //int - float Version  
        t.m1(10, 10); // C.E: reference to m1 is ambiguous, both method m1(int,float) in Test and  
                         method m1(float,int) in Test match  
        t.m1(10.5f, 10.5f); // C.E: cannot resolve symbol  
                           symbol : method m1 (float,float)  
                           location: class Test  
    }  
}
```

Case 7:

Overloading Method Resolution will always take Care by Compiler based on the Reference Type but not based on Runtime Object.



```
class Animal {}  
class Monkey extends Animal {}  
class Test {  
    public void m1(Animal a) {  
        System.out.println ("Animal Version");  
    }  
    public void m1(Monkey m) {  
        System.out.println ("Monkey Version");  
    }  
    public static void main(String arg[]) {  
        Test t = new Test();  
        Animal a = new Animal();  
        t.m1(a); //Animal Version  
  
        Monkey m = new Monkey();  
        t.m1(m); //Monkey Version  
  
        Animal a1 = new Monkey();  
        t.m1(a1); //Animal Version  
    }  
}
```

Overriding:

- Whatever the Parent has by Default Available to the Child Class through Inheritance.
- If the Child Class is Not satisfied with the Parent Class Implementation then the Child is allowed to redefine that Method in the Child Class based on its Requirement.
- This Process is Called Overriding.
- The Parent Class Method which is Overridden is called *Overridden Method* and the Child Class Method which is Overriding is called *Overriding Method*.



```
class P {
    public void property()
    {
        System.out.println ("Land + Gold + Cash");
    }
    public void mary()
    {
        System.out.println ("Subbu Lakshmi");
    }
}

class C extends P {
    public void mary()
    {
        System.out.println ("3sha/ 9tara/ 4me");
    }
}

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

        P p = new P();
        p.mary(); //Subbu Lakshmi (Parent Method)

        C c = new C();
        c.mary(); //3sha/ 9tara/ 4me (Child Method)

        P p1 = new C();
        p1.mary(); //3sha/ 9tara/ 4me (Child Method)

    }
}
```

Overridden Method

Overriding

Overriding Method

- In Overriding Method Resolution Always Take Care by JVM based on Runtime Object.
- Hence Overriding is Considered as *Runtime Polymorphism OR Dynamic Polymorphism OR Late Binding*.
- The Process of Overriding Method Resolution is Also Known as *Dynamic Method Dispatch*.

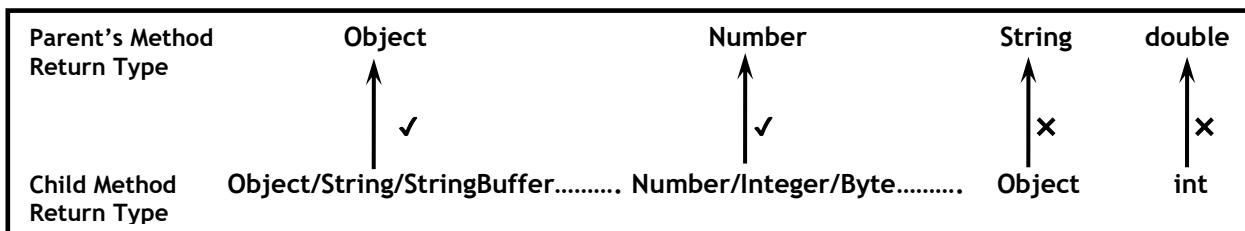
Rules for Method Overriding:

- In Overriding, Method Names and Argument Types Must be Same. i.e Method Signatures Must be Same.
- In Overriding the Return Types Must be Matched. But this Rule is Applicable Only until 1.4 Version. From 1.5 Version onwards *Co- Variant Return Types* are allowed.



According to this Child Class Method Return Types Need Not to be Same as Parent Class Method Return Type. It's Child Types Also allowed.

```
class P {  
    public Object m1() {  
        return null;  
    }  
}  
class C extends P {  
    public String m1() {  
        return null;  
    }  
}
```



Note: Co-Variant Return Type Concept Applicable Only for Object Types but Not for Primitives.

- Private Methods are Not Visible in the Child Classes. Hence Overriding Concept is Not Applicable for Private Methods. But based on Our Requirement we can Define Exactly Same Private Method in Child Class. It is Valid but Not Overriding.

```
class P {  
    private void m1() {}  
}  
class C extends P {  
    private void m1() {} } It is Valid but  
                                Not Overriding
```

- We can't Override Parent Class final Methods in Child Class i.e., Overriding Concept is Not Applicable for final Methods.

```
class P {  
    public final void m1() {}  
}  
class C extends P {  
    public void m1() {} } CE: m1() in C cannot  
                                override m1() in P;  
                                overridden method is final
```

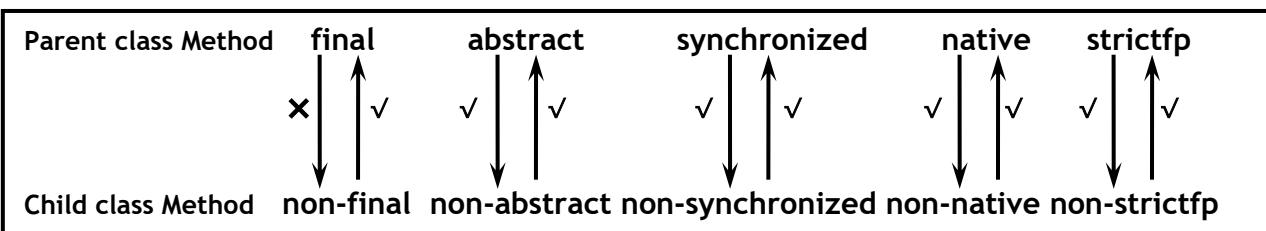
- We can Override Parent Class Abstract Method in Child Class to Provide Implementation.



We can Override Parent Class Concrete Method as Abstract in Child Class.

```
class P {  
    public void m1() {}  
}  
abstract class C extends P {  
    public abstract void m1();  
}
```

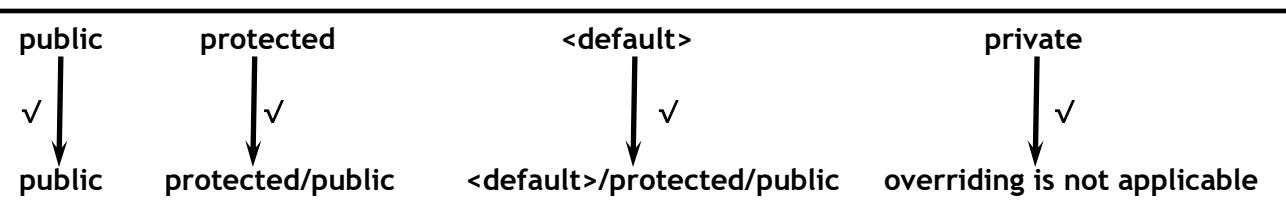
- Next Level Child is the Responsible to Provide Implementation.
- The Main Advantage of this Approach is we can Stop Availability of Parent Class Method Implementation to the Next Level Child Classes.
- The Following Modifiers won't Keep any Restrictions in Overriding.
 - synchronized,
 - native
 - strictfp
- The Following are Possible Types.



- While Overriding we can't Reduce Scope of Access Modifier. But we can Increase.

```
class P {  
    public void m1() {}  
}  
class C extends P {  
    protected void m1() {} // CE: m1() in C cannot override m1() in P  
                        // attempting to assign weaker access privileges; was public  
}
```

- The following is the List of Valid with Respect to Access Privileges.



private < default < protected < public



- If Child Class Method throws any Checked Exception Compulsory the Parent Class Method should throw the Same Checked Exception OR it's Parent. But there are No Restrictions for Un- Checked Exceptions.

```
import java.io.*;
class P {
    public void m1() throws IOException {}
}
class C extends P {
    public void m1() throws EOFException, InterruptedException{}
    // CE: m1() in C cannot override m1() in P
    // overridden method does not throw InterruptedException
}
```

Examples:

- 1) P: public void m1() throws Exception
C: public void m1 () {} ✓
- 2) P: public void m1()
C: public void m1() throws Exception

CE: m1() in C cannot override m1() in P
overridden method does not throw Exception
- 3) P: public void m1() throws Exception
C: public void m1() throws IOException ✓
- 4) P: public void m1() throws IOException
C: public void m1() throws Exception

CE: error: m1() in C cannot override m1() in P
overridden method does not throw Exception
- 5) P: public void m1() throws IOException
C: public void m1() throws EOFException, FileNotFoundException ✓
- 6) P: public void m1() throws IOException
C: public void m1() throws EOFException, InterruptedException
- 7) P: public void m1() throws IOException
C: public void m1() throws EOFException, NullPointerException ✓
- 8) P: public void m1() throws IOException
C: public void m1() throws ArithmeticException, NullPointerException, ClassCastException ✓



Overriding wrt to Static Methods

- We can't Override a Static Method as Non- Static Otherwise we will get CE.

```
class P {  
    public static void m1() {}  
}  
class C extends P {  
    public void m1() {} //CE: m1() in C cannot override m1() in P  
                      overridden method is static  
}
```

Similarly we can't Override a Non- Static Method as Static.

```
class P {  
    public void m1() {}  
}  
class C extends P {  
    public static void m1() {}  
    CE: m1() in C cannot override m1() in P  
          overriding method is static  
}
```

-

```
class P {  
    public static void m1() {}  
}  
class C extends P {  
    public static void m1() {}  
}
```

It is Method Hiding
but Not Overriding

It Seems Overriding Concept Applicable for Static Methods but it is Not Overriding. It is Method Hiding.

Method Hiding:

Method Hiding is Exactly Same as Overriding Except the following differences.

Method Hiding	Overriding
Both Parent Class and Child Class Methods should be Static.	Both Parent Class and Child Class Methods should be Non- Static.
Method Resolution always takes care by Compiler based on Reference Type.	Method Resolution always takes care by JVM based on Runtime Object.
Method Hiding is also known as <i>Compile Time Polymorphism/ Static Polymorphism/ Early Binding</i> .	Overriding is also known as <i>Runtime Polymorphism/ Dynamic Polymorphism/ Late Binding</i> .

Except these differences all Rules for Overriding and Method Hiding are Exactly Same.



```
class P {  
    public static void m1() {  
        System.out.println ("Parent Method");  
    }  
}  
class C extends P {  
    public static void m1() {  
        System.out.println ("Child Method");  
    }  
}  
class Test {  
    public static void main(String arg[]) {  
        P p = new P();  
        p.m1(); //Parent Method  
        C c = new C();  
        c.m1(); //Child Method  
        P p1 = new C();  
        p1.m1(); //Parent Method  
    }  
}
```

- If Both Parent and Child Class Methods are Not- Static, then it will become Overriding. In this Case Output is
 - Parent Method
 - Child Method
 - Parent Method

Overriding wrt var-arg Method

- We can't Override var-arg Method with General Method, if we are trying to do it will become Overloading but not Overriding.
- We should Override var-arg Method with Another var-arg Method Only.



Overloading

```
class P {  
    public void m1(int... i) {  
        System.out.println ("Parent");  
    }  
}  
class C extends P {  
    public void m1(int i) {  
        System.out.println ("Child");  
    }  
}  
class Test {  
    public static void main(String args[]) {  
        P p = new P();  
        p.m1(); //Parent  
  
        C c = new C();  
        c.m1(10); //Child  
  
        P p1 = new C();  
        p1.m1(10); //Parent  
    }  
}
```

If we Replace Child Class Method also with var-arg Method then it will become Overriding. In this Case the Output is Parent, Child, and Child.

Overriding wrt Variables

- Overriding Concept is Applicable Only for Methods but Not for Variables.
- Variables Resolution always takes Care by Compiler, based on Reference Type (but not based on Run Time Object).
- This Rule is Same whether the Variable is Static OR Non- Static.



```
class P {  
    int x = 888;  
}  
class C extends P {  
    int x = 999;  
}  
class Test {  
    public static void main(String[] args) {  
        P p = new P();  
        System.out.println(p.x); //888  
  
        C c = new C();  
        System.out.println(c.x); //999  
  
        P p1 = new C();  
        System.out.println(p1.x); //888  
    }  
}
```

Parent: non-static Child: non-static	static static	static non-static	non-static static
888	888	888	888
999	999	999	999
888	888	888	888



Comparison between Overloading and Overriding

Properties	Overloading	Overriding
Method Names	Must be Same	Must be Same
Argument Types	Must be Different (at least Order)	Must be Same (Including Order)
Method Signature	Must be Different	Must be Same
private/final/static Methods	Can be Overloaded	Can't be Overridden
Return Types	No Restriction	Must be Same until 1.4 Version. But from 1.5 Version onwards co-variant return types are allowed.
Access Modifiers	No Restriction	The Scope of Access Modifier we can't reduce but we can increase.
throws clause	No Restriction	If Child Class Method throws any Checked Exception compulsory Parent Class Method should throw the Same Checked Exception OR it's Parent but no restrictions for Un-Checked Exceptions.
Method Resolution	Always takes care by Compiler based on Reference Type and Arguments.	Always takes care by JVM based on Runtime Object.
Other Names	Static Polymorphism OR Compile time Polymorphism OR Early Binding.	Runtime Polymorphism OR Dynamic Polymorphism OR Late Binding.

Note:

- In Overloading we have to Check Only Method Names (must be Same) and Argument Types (must be different).
- The remaining things we are not required to Check.
- But in Overriding we have to Check Everything Like Method Names, Argument Types, Return Types Etc.

Consider the following Method in Parent Class

`public void m1(int i) throws IOException`

In the Child Class which of the following Methods are allowed?

- 1) `public void m1(int i)` → Overriding
- 2) `public static void m1(int i)` → Overriding
- 3) `public final void m1(int i) throws Exception` → Overriding
- 4) `private static int m1(int i) throws Exception` → Overloading



5) public static abstract void m1(double d)

CE: error: illegal combination of modifiers: abstract and static

: error: C is not abstract and does not override abstract method m1(double) in C

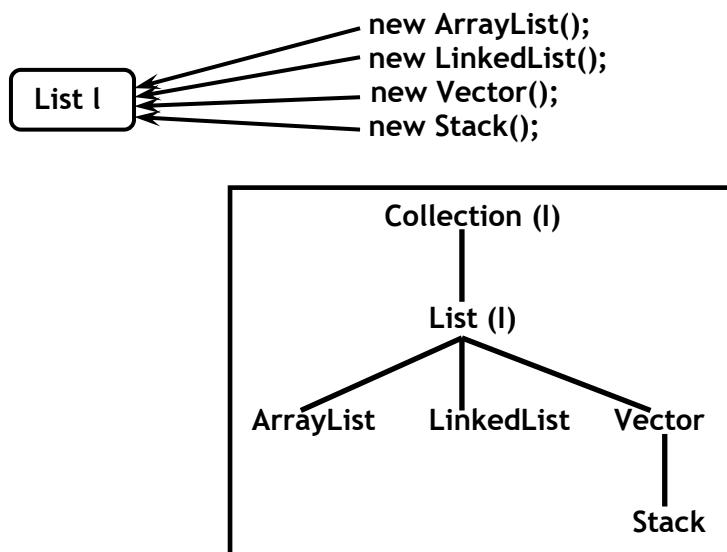
Polymorphism

Same Name but Multiple Forms is the Concept of Polymorphism.

Eg: We can Use the Same abs() for int, long and float Arguments.

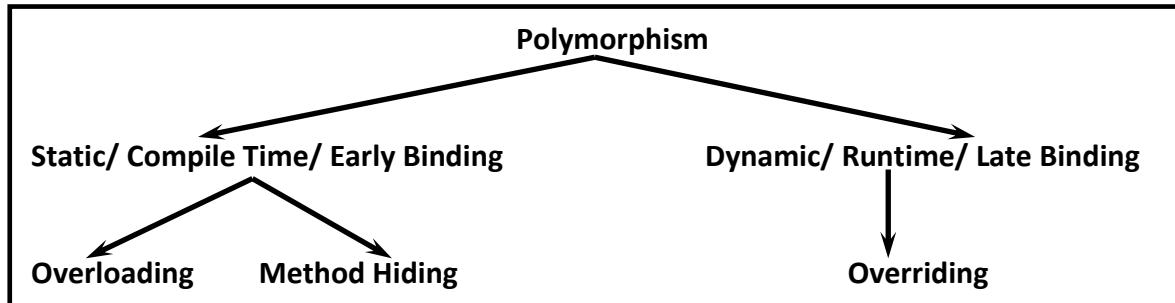
abs (int);
abs (float);
abs (long);
are all Overloaded Methods

We can Use the Same List Reference to Hold any implemented Class Object.



What is the Difference between `ArrayList al = new ArrayList();` and `List l = new ArrayList();`

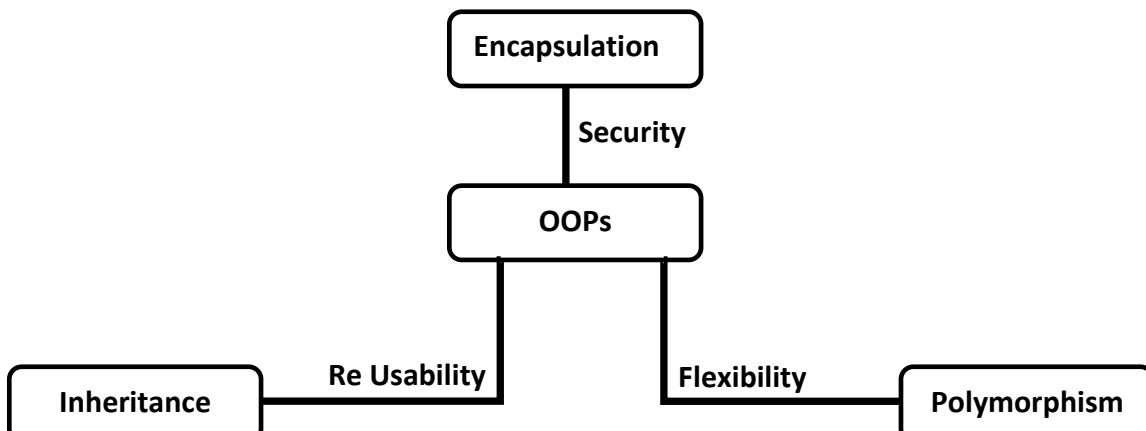
<code>ArrayList al = new ArrayList(); (C c = new C();)</code>	<code>List l = new ArrayList(); (P p = new C();)</code>
We can Use ArrayList Reference to Hold Only ArrayList Objects.	We can Use List Reference to Hold any implemented Class Objects.
If we Know the Type of Runtime Object at the beginning then we should Use this Approach.	If we don't Know the Type of Runtime Object at the beginning then we should Use this Approach.
On the Child Reference we can Call Both Parent Class and Child Class Methods.	On the Parent Reference we can Call Only Methods Available in Parent Class and Child Specific Methods we can't Call



Beautiful Definition of Polymorphism

A BOY Start LOVE with the Word FRIENDSHIP, but GIRL Ends LOVE with the Same Word FIRENDSHIP. Word is the Same but Attitude is different. This Beautiful Concept of OOPs is Nothing but Polymorphism.

Building Blocks of OOPs





Static Control Flow

```
class Base {  
    ① static int i = 10; ⑦  
  
    ② static {  
        m1(); ⑧  
        System.out.println("First Static Block"); ⑯  
    }  
    ③ public static void main(String[] args) {  
        m1(); ⑬  
        System.out.println("main()"); ⑮  
    }  
    ④ public static void m1() {  
        System.out.println(j); ⑨ ⑯  
    }  
    ⑤ static {  
        System.out.println("Second Static Block"); ⑪  
    }  
    ⑥ static int j = 20; ⑫
```

0
First Static Block
Secons Static Block
20
main()

i = 0; (RIWO)
J = 0 (RIWO)

i = 10 (R & W)
J = 20 (R & W)

Process of Static Control Flow:

Whenever we are executing a Java Class the following Sequence of Steps will be Performed Automatically.

- ① Identification of Static Members from Top to Bottom → (1-6) Steps.
i = 0 (RIWO) Read Indirect Write Only
j = 0 (RIWO)
- ② Execution of Static Variable Assignments and Static Blocks from Top to Bottom → (7-12) Steps.
i = 0 (R & W)
j = 0 (R & W)
- ③ Execution of main() → (13-15) Steps.

Read In-directly Write Only Statement:

- If a Variable is Just Identified and Assign with Default Value then the Variable is Said to be in Read in-directly write Only State.
- If we are trying to Read any Variable Inside the Static Block that Read Operation is called Direct Read.
- If we are trying to Read a Variable Inside a Method that Operation is called In-direct Read.
- If a Variable is in RIWO State then we can't Perform Read Operation Directly Otherwise we will get CE: illegal forward reference.



```
class Test {  
    ① static int x = 10;  
    ② static {  
        System.out.println(x);  
    }  
}  
Output  
10  
RE: Exception in thread "main"  
java.lang.NoSuchMethodError: main
```

x = 0 (RIWO)
x = 10

```
class Test {  
    static {  
        System.out.println (x); //CE: illegal forward reference  
    }  
    static int x = 10;  
}
```

```
class Test {  
    static {  
        m1();  
    }  
    public static void m1() {  
        System.out.println(i);  
    }  
    static int i = 10;  
}  
Output  
0  
RE: Exception in thread "main"  
java.lang.NoSuchMethodError: main
```

Static Control Flow in Parent → Child:

Whenever we are executing Parent → Child Class the following Sequence of Events will be Performed Automatically.

- Identification of Static Members from Parent → Child [1-11]
- Execution of Static Variable Assignments and Static Blocks from Parent → Child. [12-22]
- Execution of Only Child Class main(). [23-25]



```
class Base {  
    ① static int i = 10; ⑫  
    ② static {  
        m1(); ⑬  
        SOP("Base First Static Block"); ⑮  
    }  
    ③ public static void main(String[] args) {  
        m1();  
        SOP("Base main()");  
    }  
    ④ public static void m1() {  
        System.out.println(j); ⑭  
    }  
    ⑤ static int j = 20; ⑯
```

```
class Derived extends Base {  
    ⑥ static int x = 100; ⑰  
    ⑦ static {  
        m2(); ⑱  
        SOP("Derived First Static Block"); ⑳  
    }  
    ⑧ public static void main(String[] args) {  
        m2(); ⑲  
        SOP("Derived main()"); ⑳  
    }  
    ⑨ public static void m2() {  
        System.out.println(y); ⑲ ⑳  
    }  
    ⑩ static {  
        SOP("Derived Second Static Block"); ㉑  
    }  
    ⑪ static int y = 200; ㉒
```

0 Base First Static Block 0 Derived First Static Block Derived Second Static Block 200 Derived main()	1 st Step: (1 - 11) i = 0; (RIWO) j = 0; (RIWO) x = 0; (RIWO) y = 0; (RIWO)	2 nd Step: (12 - 22) i = 10; (R & W) j = 20; (R & W) x = 100; (R & W) y = 200; (R & W)	3 rd Step: (23 - 25) Execution of Derived main()
---	--	---	---

Note: Whenever we are loading Child Class Automatically Parent Class will be loaded. But whenever we are loading Parent Class Child Class won't be loaded.

Static Block:

A Class can contain Static Block, it will be executed at the Time of Class loading Automatically. Hence while loading Class if we want to Perform any Activity we have to define that Inside this Static Block Only.

Eg: After loading Driver Class Compulsory we have to Register with DriverManager. But Every Database Driver Class contains a Static Block to Perform this Activity. Hence at the Time of Driver Class loading Only registering with DriverManager will be happen Automatically and we are Responsible to Perform Explicitly.

```
class Driver {  
    static {  
        //Register this Driver with DriverManager  
    }  
}
```

Eg: At the Time of Java Class loading the Corresponding Native Libraries should be loaded. Hence we have to define this Activity Inside Static Block.



```
class Native {  
    static {  
        System.loadLibrary("Native Library Path:");  
    }  
}
```

Note: Inside a Class we can Take Any Number of Static Blocks and these Static Blocks will be executed from Top to Bottom.

Without using main() is it Possible to Print Some Statements to the Console?

Yes. By using Static Block.

```
class Test {  
    static {  
        System.out.println("Hello....I can Print ");  
        System.exit(0);  
    }  
}
```

Without using main() and Static Block is it Possible to Print Some Statements to the Console?

Yes. There are Multiple Ways.

```
class Test {  
    static int x = m1();  
    public static int m1() {  
        System.out.println("Hello I can Print");  
        System.exit(0);  
        return 10;  
    }  
}
```

```
class Test {  
    static Test t = new Test();  
    Test() {  
        System.out.println("Hello I can Print");  
        System.exit(0);  
    }  
}
```

```
class Test {  
    static Test t = new Test();  
    {  
        System.out.println("Hello I can Print");  
        System.exit(0);  
    }  
}
```

} Instance Block

Note: From 1.7 Version onwards to Run a Java Program main() is Mandatory. Even though we are writing Static Block, main() is Mandatory. Hence 1.7 Version onwards without writing main() it is not Possible to Print Some Statements to the Console.



Instance Control Flow:

- Whenever we are executing a Java Class 1st Static Control Flow will be executed.
- In the Static Control Flow whenever we are creating Object then Instance Control Flow will be executed.

```
class Parent {  
    ③ int i = 10; ⑨  
    ④ {  
        m1(); ⑩  
        System.out.println("First Instance Block"); ⑪  
    }  
    ⑤ Parent() {  
        System.out.println("Constrctor"); ⑫  
    }  
    ① public static void main(String[] args) {  
        ② Parent p = new Parent();  
        System.out.println("Parent main()");  
    }  
    ⑥ public void m1() {  
        ⑪ System.out.println(j);  
    }  
    ⑦ {  
        System.out.println("Second Instance Block"); ⑬  
    }  
    ⑧ int j = 20; ⑭
```

0
First Instance Block
Second Instance Block
Constrctor
Parent main()

1st Step (3 - 8)
i = 0; (RIWO)
j = 0; (RIWO)

2nd Step (9 - 14)
i = 10; (R & W)
j = 20; (R & W)

Whenever we are creating an Object the following Sequence of Events will be performed Automatically.

- Identification of Instance Members from Top to Bottom (3 - 8).
- Execution of Instance Variable Assignments and Instance Blocks from Top to Bottom (9 - 14).
- Execution of Constructors (15)

Note:

- Static Control Flow is One Time Activity and that will Execute at the Time of Class loading.
- But Instance Control Flow is Not One Time Activity and that It will be Execute for Every Object Creation.



Instance Control Flow in Parent → Child Relation:

```
class Parent {  
    ④ int i = 10;  ⑯  
    ⑤ {  
        m1(); ⑯  
        SOP("Parent Instance Block"); ⑯  
    }  
    ⑥ Parent() {  
        SOP("Parent Constructor"); ⑯  
    }  
    ⑦ public static void main(String[] args) {  
        Parent P = new Parent();  
        SOP("Parent main()");  
    }  
    ⑧ int j = 20;  ⑯  
}
```

```
class Child extends Parent {  
    ⑨ int x = 100;  ㉑  
    ⑩ {  
        m2(); ㉑  
        SOP("Child First Instance Block"); ㉑  
    }  
    ⑪ Child() {  
        SOP("Child Constructor"); ㉑  
    }  
    ⑫ public static void main(String[] args) {  
        ⑬ Child c = new Child();  
        SOP("Child main()"); ㉑  
    }  
    ⑯ public void m2() {  
        System.out.println(y); ㉑  
    }  
    ⑯ {  
        SOP("Child Second Instance Block"); ㉑  
    }  
    ⑯ int y = 200;  ㉑  
}
```

0
Parent Instance Block
Parent Constructor
0
Child First Instance Block
Child Second Instance Block
Child Constructor
Child main()

Whenever we are creating Child Class Objects the following Sequence of Events will be Performed Automatically.

- 1) Identification of Instance Members from Parent to Child.(4 - 14)
i = 0 (RIWO)
j = 0 (RIWO)
x = 0 (RIWO)
y = 0 (RIWO)
- 2) Execution of Instance Variable Assignments and Instance Blocks only in Parent Class.
(15 - 19)
i = 10 (R & W)
j = 20 (R & W)
- 3) Execution of Parent Class Constructor. 20 Step



4) Execution of Instance Variable Assignments and Instance Blocks in the Child Class.

(21 - 26)

x = 100

y = 200

5) Execution of Child Class Constructor. 27TH Step

```
public class Initialization {  
    private static String m1(String msg) {  
        System.out.println (msg);  
        return msg;  
    }  
    public Initialization() {  
        m = m1("1");  
    }  
    {  
        m = m1("2");  
    }  
    String m = m1("3");  
    public static void main(String args[]) {  
        Object obj = new Initialization();  
    }  
}
```

2
3
1

```
public class Initialization {  
    private static String m1(String msg) {  
        System.out.println (msg);  
        return msg;  
    }  
    static String m = m1("1");  
    {  
        m = m1("2");  
    }  
    static {  
        m = m1("3");  
    }  
    public static void main(String args[]) {  
        Object obj = new Initialization();  
    }  
}
```

1
3
2

```
class Test {  
    static {  
        System.out.println ("First Static Block");  
    }  
    {  
        System.out.println ("First Instance Block");  
    }  
    Test() {  
        System.out.println ("Constructor");  
    }  
    public static void main (String args[]) {  
        Test t = new Test();  
    }  
    static {  
        System.out.println ("Second Static Block");  
    }  
    {  
        System.out.println ("Second Instance Block");  
    }  
}
```

First Static Block
Second Static Block
First Instance Block
Second Instance Block
Constructor

From Static Area we can't Access Instance Members Directly because while executing Static Area JVM May Not Identify Instance Members.



```
class Test1 {  
    int x = 10;  
    public static void main(String[] args) {  
        System.out.println(x); }  
}
```

CE: error: non-static variable x cannot be referenced from a static context

Note: Object Creation is Most Costly Operation in Java. Hence if there is No Specific Requirement then it Never Recommended to Create Object.

In how Many Ways we can Create Object in Java? OR
How Many Ways we can get Object in Java?

We can Create Object in Java by using the following 5 Ways.

1) By using new Operator: Test t = new Test();

2) By using newInstance(); [By using Reflection API]

```
Test t = (Test)Class.forName("Test").newInstance();  
OR  
(Test)Test.class.newInstance();
```

3) By using Factory Method:

```
Runtime r = Runtime.getRuntime();  
DateFormat df = DF.getInstance();
```

4. By using clone():

```
Test t1 = new Test();  
Test t2 = (Test)t1.clone();
```

5. By using Deserialization:

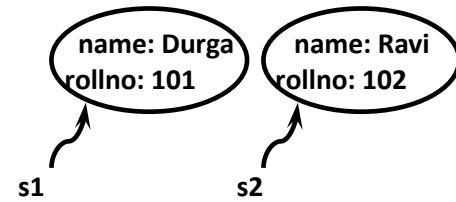
```
FIS fis = new FIS("abc.ser");  
OIS ois = new OIS(fis);  
Test t2 = (Test)ois.readObject();
```

Constructor:

- Once we Create an Object Compulsory we should Perform Initialization. Then Only the Object is in a Position to Respond Properly.
- Whenever we are creating an Object Some Piece of Code will be executed Automatically to Perform Initialization of an Object. This Piece of the Code is Nothing but Constructor.
- Hence the Main Objective of Constructor is to Perform Initialization of an Object.



```
class Student {  
    String name;  
    int rollno;  
    Student(String name, int rollno) {  
        this.name = name;  
        this.rollno = rollno;  
    }  
    public static void main(String[] args) {  
        Student s1 = new Student("Durga",101);  
        Student s2 = new Student("Ravi",102);  
        .....  
        .....  
    }  
}
```



Constructor Vs Instance Block

- The Main Purpose of Constructor is to Perform Initialization of an Object. Other than Initialization if we want to Perform any Activity for Every Object Creation then we should go for Instance Block.
- For Every Object Creation Both Constructor and Instance Block will be executed but 1st Instance Block followed by Constructor.
- Replacing Constructor with Instance Block OR Instance Block with Constructor is Not Possible. Both Concepts have their Own Purposes.

```
class Test {  
    static int count = 0;  
  
    {  
        count++;  
    }  
  
    Test() {}  
  
    Test(int i) {}  
  
    public static void main(String[] args) {  
        Test t1 = new Test();  
        Test t2 = new Test(10);  
        System.out.println ("The Number of Objects Created:" +count);  
    }  
}
```

The Number of Objects Created: 2

Rules for Constructors:

- The Name of the Constructor and the Name of the Class must be Same.
- Return Type Concept is Not applicable for Constructors Even void Also.
- By Mistake if we are trying to Declare Return Type then we won't get CE or RE. Simply it is treated as a Method.



```
class Test {  
    void Test() { } → It is a Method but Not Constructor  
}
```

- It is Legal (But Stupid) to have a Method whose Name is exactly Same as Class Name.
- The Only Applicable Modifiers for Constructor are public, private, protected and default. If we are trying to Apply any Other Modifier we will get CE.

```
class Test {  
    final Test() { } // CE: modifier final not allowed here  
}
```

Default Constructor:

- Every Class in Java contains Constructor.
- If we are Not writing any Constructor then the Compiler Always generates Default Constructor.
- If we are writing at Least One Constructor then the Compiler won't generate any Default Constructor.
- Hence Every Class contains either Programmer written Constructor OR Compiler generated Default Constructor but Not Both Simultaneously.

Prototype of Default Constructor:

- It is always no-arg Constructor.
- It contains only One Line `super();` It is a no-arg Call to Super Class Constructor.
- The Modifier of the Default Constructor is Same as Class Modifier (But this Rule is applicable only for `public` and `default`)



Programmers code	Compiler generated Code
class Test { }	class Test { Test() { super(); } }
public class Test { }	public class Test { public Test() { super(); } }
class Test { void Test() { } }	class Test { Test() { super(); } void Test() { } }
class Test { Test() { } }	class Test { Test() { super(); } }
class Test { Test(int i) { this(); } Test() { } }	class Test { Test(int i) { this(); } Test() { super(); } }
class Test { Test(int i) { super(); } }	class Test { Test(int i) { super(); } }



The 1st Line Inside Every Constructor should be either super OR this. If we are Not writing anything Compiler will always Place super().

Case1: We can use *super()* OR *this* only in 1st Line of the Constructor. If we are using anywhere else we will get CE.

```
class Test {  
    Test(){  
        System.out.println ("Constructor");  
        super(); //CE: call to super must be first statement in constructor  
    }  
}
```

Case2: Within the Constructor we can use either super OR this but not Both Simultaneously.

```
class Test {  
    Test(){  
        super();  
        this(); //CE: call to this must be first statement in constructor  
    }  
}
```

Case 3: We can Use *super()* and *this()* Only in Constructors. If we are using Outside of the Constructor we will get CE. That is we can Invoke a Constructor Directly from Another Constructor only.

```
class Test {  
    public void m1(){  
        super(); //CE: call to super must be first statement in constructor  
    }  
}
```

super();
this();

We can Use Only within Constructor
At 1st Statement Only
Only One but Not Simultaneously

Differences between *super()* - *this()* and *super - this*:

<i>super()</i> - <i>this()</i>	<i>super - this</i>
These are Constructor Calls, to Call Super Class and Current Class Constructors.	These are Key Words to Refer Super Class and Current Class Instance Members.
We can write Only in Constructor as 1 ST Statement Only.	Anywhere except Static Area.
We can write Only One but Not Both Simultaneously.	Any Number of Times.



```
class Test {  
    public static void main(String args[]) {  
        System.out.println( super.hashCode() );  
        //CE: non-static variable super cannot be referenced from a static context  
    }  
}
```

Overloaded Constructor

A Class can contain Multiple Constructors with Same Name but different Argument Types. This Type of Constructors are called *Overloaded Constructors*. Hence Overloading Concept is applicable for Constructors.

```
class Test {  
  
    Test(double d) {  
        this(10);  
        System.out.println("double-arg");  
    }  
  
    Test(int i) {  
        this();  
        System.out.println("int-arg");  
    }  
  
    Test() {  
        System.out.println("no-arg");  
    }  
  
    public static void main(String arg[]) {  
  
        Test t1 = new Test(10.8);  
        Test t2 = new Test(10);  
        Test t3 = new Test();  
        Test t4 = new Test('a');  
    }  
}
```

- Parent Class Constructors by Default won't Available to the Child and hence Inheritance Concept is Not Applicable for Constructors.
- As Inheritance Concept Not Applicable, by Default Overriding Concept Also Not Applicable.
- Every Class in Java including Abstract Class can contain Constructor. But Interface cannot have/ contain Constructor.



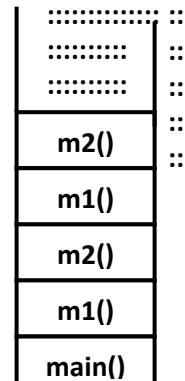
```
class Test {  
    Test() {}  
} ✓
```

```
abstract class Test {  
    Test() {}  
} ✓
```

```
interface Test {  
    Test() {}  
} ✗
```

Case 1: Recursive Method Call is always Runtime Exception saying StackOverflowError. But in Our Program if there is any Chance of Recursive Constructor Invocation we will get CE i.e., Recursive Method Invocation is the RE where as Recursive Constructor is a CE.

```
class Test {  
    public static void m1() {  
        m2();  
    }  
    public static void m2() {  
        m1();  
    }  
    public static void main(String arg[]) {  
        m1();  
        System.out.println("Hello.....Hai");  
        // RE: Exception in thread "main" java.lang.StackOverflowError  
    }  
}
```



Run Time Flow

```
class Test {  
    Test() {  
        this(10);  
    }  
    Test(int i) { //CE: recursive constructor invocation  
        this();  
    }  
    public static void main(String arg[]) {  
        System.out.println("Hello.....Hai");  
    }  
}
```

Case 2:

- If the Parent Class contains Argument Constructor then while writing Child classes we have to Take Special Care with Respect to Constructors.
- Whenever we are writing any Argument Constructor it is Highly Recommended to write no-arg Constructor Also.



```
class P {}  
class C extends P {} ✓
```

```
class P {  
    P() {}  
}  
class C extends P {} ✓
```

```
class P {  
    P(int i) {}  
}  
class C extends P {} X
```

CE: constructor P in class P cannot be applied to given types;
required: int
found: no arguments
reason: actual and formal argument lists differ in length

Case 3: If the Parent Class Constructor *throws* Some Checked Exception. Compulsory the Child Class Constructor should throw the Same Checked Exception OR its Parent Otherwise CE.

```
import java.io.*;  
class P {  
    P() throws IOException {}  
}  
class C extends P {} //CE: unreported exception IOException in default constructor
```

```
import java.io.*;  
class P {  
    P() throws IOException {}  
}  
class C extends P {  
    C() throws IOException | Exception | Throwable {  
        super();  
    }  
}
```

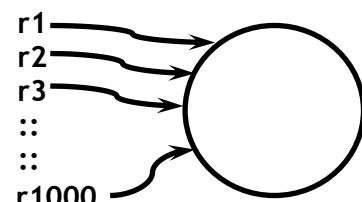
Singleton Classes: For any Java Class if we are allow to Create Only One Object Such Type of Class is called Singleton Class.

Eg: Runtime, ActionServlet, BusinessDelegate, ServiceLocator etc.

Advantage:

- Instead of creating a Separate Object for Every Requirement we can Create Only One Object and we can Reuse the Same Object for Every Similar Requirement.
- So that *Memory Utilization* and *Performance* will be Improved.
- This is the Main Advantage of Singleton Classes.

```
Runtime r1 = Runtime.getRuntime();  
Runtime r2 = Runtime.getRuntime();  
Runtime r3 = Runtime.getRuntime();  
:::  
:::  
Runtime r1000 = Runtime.getRuntime();
```





Note:

- We can't Create Singleton Class Objects by using Constructor.
- We can Create by using Factory Method.

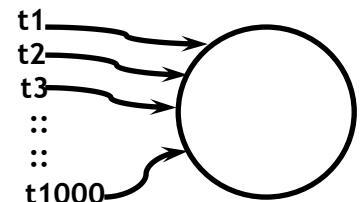
Creation of Our Own Singleton Classes:

We can Create Our Own Singleton Classes; for this we have to Use

- Private Constructor,
- Factory Method,
- One Private Static Variable

```
class Test {  
    private static Test t = null;  
  
    private Test() {}  
  
    private static Test getTest() {  
        if(t == null) {  
            t = new Test();  
        }  
        return t;  
    }  
  
    public Object clone() {  
        return this;  
    }  
}
```

```
Test t1 = Test.getTest();  
Test t2 = Test.getTest();  
Test t3 = Test.getRuntime();  
:::  
:::  
Test t1000 = Test.getTest();
```



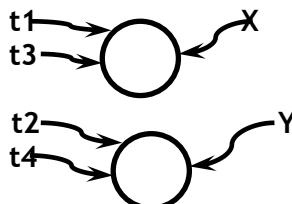
In the Above Example for Test Class we are allowed to Create Only One Object and Hence it is Singleton Class.



Creation of Our Own Doubleton Classes:

```
class Test {  
    private static Test x = null;  
  
    private static Test y = null;  
  
    private Test() {}  
  
    public static Test getTest() {  
        if(x == null) {  
            x = new Test();  
            return x;  
        }  
        else if(y == null) {  
            y = new Test();  
            return y;  
        }  
        else if(Math.random() < 0.5) return x;  
        else return y;  
    }  
}
```

```
Test t1 = Test.getTest();  
Test t2 = Test.getTest();  
:::  
:::
```



Similarly we can Create Thribleton, Tenton any Xxxton Classes.

Note: This Type of Approach we can Use while developing Connection Pools, Thread Pools etc.

Class is Not final but we are Not allowed to Create Child Class. How it is Possible?
If we Declare Every Constructor as Private then we can't Create Child Class for that.

```
class Parent {  
    private Parent() {}  
}  
class Child extends Parent {} //CE: Parent() has private access in Parent
```



Coupling: The Degree of Dependency between the Components is Called Coupling.

Eg:	class A { int i = B.j; }	class B { static int j = C.m1(); }	class C { public static int m1() { return D.k; } }	class D { static int k= 10; }
-----	--------------------------------	--	--	-------------------------------------

The Dependency between the Above Classes is More and Hence these Components are Said to be Tightly Coupled with Each Other.

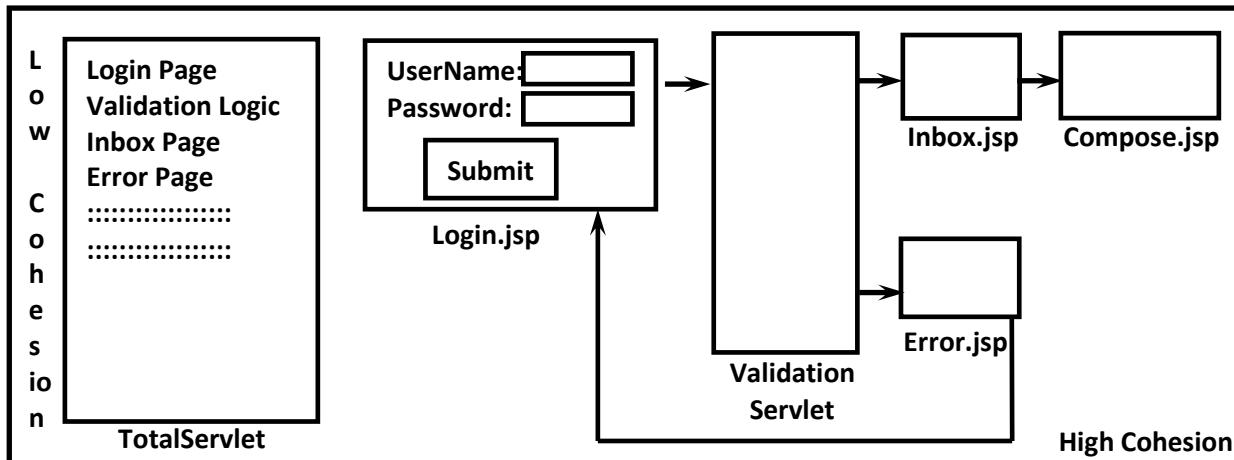
Tightly Coupling is Not a Good Programming Practice. Because it has Several Serious Disadvantages.

- Without effecting remaining Components we can't Modify any Component. Hence Enhancement will become Difficult.
- It doesn't Promote Re-usability of the Code.
- It Reduces Maintainability of the Application.

Hence we have to Maintain Dependency between the Components as Less as Possible. That is Loosely Coupling is Always Good Programming Practice.

Cohesion:

For Every Component we have to Define a Clear Well defined Functionality. Such Type of Component is Said to be follow High Cohesion.



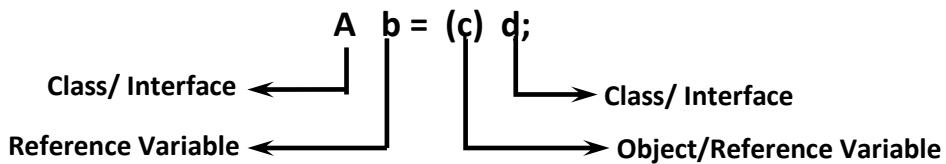
High Cohesion is Always Good Programming Practice. Because it has Several Advantages.

- Without effecting remaining Components we can Modify any Component. Hence Enhancement will become Very Easy.
- It Promotes Re- Usability of the Code (wherever Validation Logic is required we can Re- Use the Same ValidateServlet without re-writing).
- It Improves Maintainability of the Application.

Note: Loosely Coupling and High Cohesion are Always Good Programming Practices.



Type Casting



Compile Time Checking:

- ① The Type of d and c Must have Some Relation.
(Either Parent → Child OR Child → Parent OR Same Type) Otherwise we will get CE.

```
Object o = new String ("Durga");
StringBuffer sb = (StringBuffer)o; ✓
```

```
String s = new String ("Durga");
StringBuffer sb = (StringBuffer)s;
```

CE: inconvertible types
required: StringBuffer
found: String

- ② 'C' should be Either Same OR Derived Type of A. Otherwise we will get CE.

```
String s = new String ("Durga");
StringBuffer sb = (StringBuffer)s;
```

CE: inconvertible types
required: StringBuffer
found: String

Runtime Checking

The Runtime Object Type of 'd' Must be either Same OR derived Type of 'C'. Otherwise we will get Runtime Exception saying ClassCastException : 'd' type

```
Object o = new String("Durga");
StringBuffer sb = (StringBuffer)o;
```

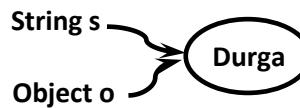
//RE: Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.StringBuffer

```
Object o = new StringBuffer("Ravi");
StringBuffer sb = (StringBuffer)o; ✓
```

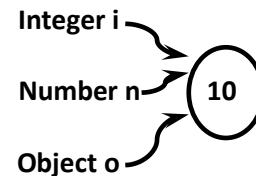


- ❖ Strictly speaking through Type Casting we are not creating any New Object for the existing Object. We are providing another Type of Reference Variable.

```
String s = new String("Durga");
Object o = (Object)s;
System.out.println(s == o); //true
```

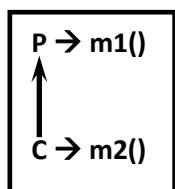


```
class Test {
    public static void main(String args[]) {
        Integer i = new Integer(10);
        Number n = (Number)i;
        Object o = (Object)n;
        System.out.println(i.hashCode()); //10
        System.out.println(n.hashCode()); //10
        System.out.println(o.hashCode()); //10
        System.out.println(i == n); //true
        System.out.println(i == o); //true
    }
}
```



Examples:

```
C c = new C();
c.m1() √
c.m2() √
( (P)c ).m1() √
( (P)c ).m2()
CE: cannot find symbol
found: method m2()
location: class P
```



```
C c = new C(c);
c.m1(); // C
( (B)c ).m1(); //C
( (A) ( (B)c ) ).m1(); //C
```



It is Overriding and Method Resolution is Always based on Runtime Object.

```
C c = new C(c);
c.m1(); // C
( (B)c ).m1(); //B
( (A) ( (B)c ) ).m1(); //A
```



It is Method Hiding and Method Resolution is Always based on Reference Type.



```
C c = new C();  
  
SopIn(c.x); //777  
  
SopIn( (B)c).x); //888  
  
sopIn( (A) ( (B)c ).x); //999
```

```
A → int x = 999;  
B → int x = 888;  
C → int x = 777;
```

Overriding Concept is Not Applicable for Variables and Variable Resolution is Always Takes Care by Compiler On Reference Type Based



OOPS Practice Questions

Exam Objectives:

1. Compare and contrast the features and components of Java such as: object orientation, encapsulation, etc.
2. Describe inheritance and its benefits
3. Create methods with arguments and return values; including overloaded methods
4. Apply encapsulation principles to a class
5. Develop code that makes use of polymorphism; develop code that overrides methods; differentiate between the type of a reference and the type of an object
6. Create and overload constructors; differentiate between default and user defined constructors
7. Use super and this to access objects and constructors
8. Determine when casting is necessary

Q1. Which three statements describe the object oriented features of the java language?

- A. Objects cannot be reused
- B. A sub class can inherit from a super class
- C. Objects can share behaviours with other objects
- D. A package must contains more than one class
- E. Object is the root class of all other objects
- F. A main method must be declared in every class

Answer: B,C,E

Q2. What is the name of the Java concept that uses access modifiers to protect variables and hide them within a class?

- A. Encapsulation
- B. Inheritance
- C. Abstraction
- D. Instantiation
- E. Polymorphism

Ans: A

Explanation: This concept is data hiding, but that option is not available and hence we can choose encapsulation

Q3. Which statement best describes encapsulation?

- A. Encapsulation ensures that classes can be designed so that only certain fields and methods of an object are accessible from other objects



- B. Encapsulation ensures that classes can be designed so that their methods are inheritable
- C. Encapsulation ensures that classes can be designed with some fields and methods declared as abstract.
- D. Encapsulation ensures that classes can be designed so that if a method has an argument X, any subclass of X can be passed to that method.

Answer: A

Q4. Given the following two classes:

```
1) public class Customer
2) {
3)     ElectricAccount acct=new ElectricAccount();
4)     public void useElectricity(double kwh)
5)     {
6)         acct.addKwh(kwh);
7)     }
8) }
9) public class ElectricAccount
10) {
11)     private double kwh;
12)     public double rate=0.09;
13)     private double bill;
14)     //Line-1
15) }
```

How should you write methods in ElectricAccount class at Line-1 so that the member variable bill is always equal to the value of the member variable kwh multiplied by the member variable rate?

Any amount of electricity used by Customer(represented by an instance of the Customer class) must contribute to the Customer's bill(represented by member variable bill) through the method useElectricity() method. An instance of the customer class should never be able to tamper with or decrease the value of the member variable bill?

A)

```
1) public void addKwh(double kwh)
2) {
3)     this.kwh+=kwh;
4)     this.bill=this.kwh*this.rate;
5) }
```



B)

```
1) public void addKwh(double kwh)
2) {
3)     if(kwh>0)
4)     {
5)         this.kwh+=kwh;
6)         this.bill=this.kwh*this.rate;
7)     }
8) }
```

C)

```
1) private void addKwh(double kwh)
2) {
3)     if(kwh>0)
4)     {
5)         this.kwh+=kwh;
6)         this.bill=this.kwh*this.rate;
7)     }
8) }
```

D)

```
1) public void addKwh(double kwh)
2) {
3)     if(kwh>0)
4)     {
5)         this.kwh+=kwh;
6)         setBill(this.kwh)
7)     }
8) }
9) public void setBill(double kwh)
10) {
11)     bill=kwh*rate;
12) }
```

Answer: C

Explanation:

Any amount of electricity used by Customer(represented by an instance of the Customer class) must contribute to the Customer's bill(represented by member variable bill) through the method useElectricity() method.

means no one is allowed to call addKwh() method directly, should be private

An instance of the customer class should never be able to tamper with or decrease the value of the member variable bill



if we pass -ve value for kwh then we can decrease the bill. To prevent this we should check $kwh > 0$

Q5. Given the following code fragment:

```
1) public class Rectangle
2) {
3)     private double length;
4)     private double height;
5)     private double area;
6)     public void setLength(double length)
7)     {
8)         this.length=length;
9)     }
10)    public void setHeight(double height)
11)    {
12)        this.height=height;
13)    }
14)    public void setArea()
15)    {
16)        area=length*height;
17)    }
18) }
```

Which two changes would encapsulation this class and ensure that the area field is always equal to length*height.whenever Rectangle class is used?

- A. Change the area field to public
- B. Change the setArea() method to private?
- C. Call the setArea() method at the beginning of the setLength() method
- D. Call the setArea() method at the end of the setLength() method
- E. Call the setArea() method at the beginning of the setHeight() method
- F. Call the setArea() method at the end of the setHeight() method

Answer: B,F

Q6. Given the following classes:

```
1) public class Employee
2) {
3)     public int salary;
4) }
5) public class Manager extends Employee
6) {
7)     public int budget;
8) }
9) public class Director extends Manager
10) {
11)     public int stockOptions;
```



12) }

And given the following main method:

```
1) public static void main(String[] args)
2) {
3)     Employee e = new Employee();
4)     Manager m = new Manager();
5)     Director d = new Director();
6)     //Line 1
7) }
```

Which two options fail to compile when placed at Line 1 of the main method?

- A. e.salary=50_000;
- B. d.salary=80_000;
- C. e.budget=2_00_000;
- D. m.budget=1_00_000;
- E. m.stockOption=500;
- F. d.stockOption=1_000;

Answer: C and E

Q7.Given the code fragment:

```
1) abstract class Parent
2) {
3)     protected void resolve()//Line-1
4)     {
5)     }
6)     abstract void rotate()//Line-2
7) }
8) class Child extends Parent
9) {
10)    void resolve()//Line-3
11)    {
12)    }
13)    protected void rotate()//Line-4
14)    {
15)    }
16) }
```

Which two modifications,made independently, enable the code to compile?

- A. Make that method at Line-1 public
- B. Make that method at Line-2 public
- C. Make that method at Line-3 public
- D. Make that method at Line-3 protected



E. Make that method at Line-4 public

Answer: C,D

Explanation: While overriding, we cannot reduce the scope of access modifier, but we can increase the scope

Q8. Given:

Base.java:

```
1) class Base
2) {
3)     public void test()
4)     {
5)         System.out.println("Base");
6)     }
7) }
```

DerivedA.java:

```
1) class DerivedA extends Base
2) {
3)     public void test()
4)     {
5)         System.out.println("DerivedA");
6)     }
7) }
```

DerivedB.java

```
1) class DerivedB extends DerivedA
2) {
3)     public void test()
4)     {
5)         System.out.println("DerivedB");
6)     }
7)     public static void main(String[] args)
8)     {
9)         Base b1= new DerivedB();
10)        Base b2= new DerivedA();
11)        Base b3= new DerivedB();
12)        b1=(Base)b3;
13)        Base b4=(DerivedA)b3;
14)        b1.test();
15)        b4.test();
16)    }
17) }
```



18) }

What is the result?

- A.
Base
DerivedA
- B.
Base
DerivedB
- C.
DerivedB
DerivedB
- D.
DerivedB
DerivedA
- E. A **ClassCastException** is thrown at runtime

Answer: C

Q9. Which two are benefits of polymorphism?

- A. Faster Code at Runtime
- B. More efficient Code at Runtime
- C. More Dynamic Code at Runtime
- D. More Flexible and Reusable Code at Runtime
- E. Code that is protected from extension by other classes

Answer: C,D

Q10. Which three statements are true about the structure of a Java class?

- A) public class should compulsory contains main method
- B) A class can have only one private constructor
- C) A method can have the same name as variable
- D) A class can have overloaded static methods
- E) The methods are mandatory components of a class
- F) The fields need not be initialized before use.

Answer: C, D,F



Q11. Given:

```
1) class A
2) {
3)     public void test()
4)     {
5)         System.out.print("A");
6)     }
7) }
8) class B extends A
9) {
10)    public void test()
11)    {
12)        System.out.print("B");
13)    }
14) }
15) public class C extends A
16) {
17)    public void test()
18)    {
19)        System.out.print("C");
20)    }
21)    public static void main(String[] args)
22)    {
23)        A a1 = new A();
24)        A a2 = new B();
25)        A a3 = new C();
26)        a1.test();
27)        a2.test();
28)        a3.test();
29)    }
30) }
```

What is the output?

- A. AAA
- B. ABC
- C. AAB
- D. ABA

Ans: B



Q12. Given:

```
1) public class Test
2) {
3)     public static void sum(Integer x,Integer y)
4)     {
5)         System.out.println("Integer sum is:"+ (x+y));
6)     }
7)     public static void sum(double x,double y)
8)     {
9)         System.out.println("double sum is:"+ (x+y));
10)    }
11)    public static void sum(float x,float y)
12)    {
13)        System.out.println("float sum is:"+ (x+y));
14)    }
15)    public static void sum(int x,int y)
16)    {
17)        System.out.println("int sum is:"+ (x+y));
18)    }
19)    public static void main(String[] args)
20)    {
21)        sum(10,20);
22)        sum(10.0,20.0);
23)    }
24) }
```

What is the result?

A.

int sum is:30
double sum is:30.0

B.

int sum is:30
float sum is:30.0

C.

Integer sum is:30
double sum is:30.0

D.

Integer sum is:30
float sum is:30.0

Answer: A



Q13. Given the code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         Short s1=200;
6)         Integer s2=400;
7)         Long s3=(long)s1+s2; //Line-1
8)         String s4=(String)(s3*s2); // Line-2
9)         System.out.println(s3);
10)    }
11) }
```

What is the result?

- A. 600
- B. Compilation Fails at Line-1
- C. Compilation Fails at Line-2
- D. A ClassCastException is thrown at Line-1
- E. A ClassCastException is thrown at Line-2

Answer: C



Practice Questions for Constructors

Q1. Given:

```
1) class Vehicle
2) {
3)     String type="4w";
4)     int maxSpeed=120;
5)     Vehicle(String type,int maxSpeed)
6)     {
7)         this.type=type;
8)         this.maxSpeed=maxSpeed;
9)     }
10) }
11) class Car extends Vehicle
12) {
13)     String trans;
14)     Car(String trans)
15)     {
16)         //Line-1
17)         this.trans=trans;
18)     }
19)     Car(String type,it maxSpeed,String trans)
20)     {
21)         super(type,maxSpeed);
22)         this(trans);//Line-2
23)     }
24) }
```

And given the code fragment:

```
1) Car c1= new Car("Auto");
2) Car c2= new Car("4w",150,"Manual");
3) System.out.println(c1.type+ .. +c1.maxSpeed+ .. +c1.trans);
4) System.out.println(c2.type+ .. +c2.maxSpeed+ .. +c2.trans);
```

What is the result?

- A. 4w 120 Auto
4w 150 Manual
- B. null 0 Auto
4w 150 Manual
- C. Compilatio Fails only at Line-1
- D. Compilatio Fails only at Line-2
- E. Compilatio Fails at both Line-1 and Line-2



Answer: E

Q2. Given:

```
1) class CD
2) {
3)     int r;
4)     CD(int r)
5)     {
6)         this.r=r;
7)     }
8) }
9) class DVD extends CD
10) {
11)     int c;
12)     DVD(int r, int c)
13)     {
14)         //line-1
15)     }
16) }
```

And Given the code Fragment:

```
DVD dvd = new DVD(10, 20);
```

Which code fragment should be inserted at Line-1 to instantiate dvd object successfully?

- A. super.r=r;
this.c=c;
- B. super(r);
this(c);
- C. super(r);
this.c=c;
- D. this.c=r;
super(c)

Answer: C

Q3. Given the code Fragment:

```
1) public class Employee
2) {
3)     String name;
4)     boolean contract;
5)     double salary;
```



```
6) Employee()
7) {
8)     //line-1
9) }
10) public String toString()
11) {
12)     return name+":"+contract+":"+salary;
13) }
14) public static void main(String[] args)
15) {
16)     Employee e = new Employee();
17)     //Line-2
18)     System.out.println(e);
19) }
20) }
```

Which 2 modifications, when made independently, enable the code to print
Durga:true:100.0

A. Replace line-2 with

```
e.name="Durga";
e.contract=true;
e.salary=100;
```

B. Replace line-2 with

```
this.name="Durga";
this.contract=true;
this.salary=100;
```

C. Replace line-1 with

```
this.name=new String("Durga");
this.contract= new Boolean(true);
this.salary= new Double(100);
```

D. Replace line-1 with

```
name="Durga";
contract=TRUE;
salary=100.0f;
```

E. Replace line-1 with:

```
this("Durga",true,100)
```

Answer: A and C



Q4. Given:

```
1) class A
2) {
3)     public A()
4)     {
5)         System.out.println("A");
6)     }
7) }
8) class B extends A
9) {
10)    public B()
11)    {
12)        //line-1
13)        System.out.println("B");
14)    }
15) }
16) class C extends B
17) {
18)    public C()
19)    {
20)        //line-2
21)        System.out.println("C");
22)    }
23)    public static void main(String[] args)
24)    {
25)        C c = new C();
26)    }
27) }
```

What is the Result?

- A. CBA
- B. C
- C. ABC
- D. Compilation Fails at line-1 and line-2

Answer: C

Q5. Given

```
1) class Vehicle
2) {
3)     int x;
4)     Vehicle()
5)     {
6)         this(10); // line-1
7)     }
```



```
8)  Vehicle(int x)
9)  {
10)   this.x=x;
11) }
12) }
13) class Car extends Vehicle
14) {
15)   int y;
16) Car()
17) {
18)   super();
19)   this(20);//line-2
20) }
21) Car(int y)
22) {
23)   this.y= y;
24) }
25) public String toString()
26) {
27)   return super.x+":"+this.y;
28) }
29) }
```

And given the code fragment:

```
Vehicle v = new Car();
System.out.println(v);
```

What is the result?

- A. 10:20
- A. 0:20
- C. Compilation Fails at Line-1
- D. Compilation Fails at Line-2

Answer: D

Q6. Given the code fragment:

```
1) public class Person
2) {
3)   String name;
4)   int age=25;
5)   public Person(String name)
6)   {
7)     this(); //line-1
8)     setName(name);
9)   }
```



```
10) public Person(String name,int age)
11) {
12)     Person(name); //Line-2
13)     setAge(age);
14) }
15) //setter and getter methods go here
16) public String show()
17) {
18)     return name+" "+age+" "+number;
19) }
20) public static void main(String[] args)
21) {
22)     Person p1= new Person("Durga");
23)     Person p2= new Person("Ravi",50);
24)     System.out.println(p1.show());
25)     System.out.println(p2.show());
26) }
27} }
```

What is the result?

- A. Durga 25
Ravi 50
- B. Compilation fails at Line-1
- C. Compilation fails at Line-2
- D. Compilation Fails at both line-1 and line-2

Answer: D

Q7. Given:

```
1) class Animal
2) {
3)     String type="Canine";
4)     int maxSpeed=60;
5)     Animal(){}
6)     Animal(String type,int maxSpeed)
7)     {
8)         this.type=type;
9)         this.maxSpeed=maxSpeed;
10)    }
11) }
12) class WildAnimal extends Animal
13) {
14)     String bounds;
15)     WildAnimal(String bounds)
16)     {
17)         //line-1
```



```
18) }
19) WildAnimal(String type,int maxSpeed,String bounds)
20) {
21) //line-2
22) }
23) }
```

And the code fragment:

```
WildAnimal wolf= new WildAnimal("Long");
WildAnimal tiger= new WildAnimal("Feline",80,"Short");
System.out.println(wolf.type+" "+wolf.maxSpeed+" "+wolf.bounds);
System.out.println(tiger.type+" "+tiger.maxSpeed+" "+tiger.bounds);
```

Which 2 modifications enable to the code to print the following output?

Canine 60 Long
Feline 80 Short

- A. Replace line-1 with
super();
this.bounds=bounds;
- B. Repalce line-1 with
this.bounds=bounds;
super();
- C. Replace line-2 with
super(type,maxSpeed);
this(bounds);
- D. Repalce line-1 with
this("Canine",60);
this.bounds=bounds;
- E. Replace line-2 with
super(type,maxSpeed);
this.bounds=bounds;

Answer: A and E

Q8.

```
1) class Employee
2) {
3) private String name;
4) private int age;
5) private int salary;
6)
```



```
7) public Employee(String name,int age)
8) {
9)     setName(name);
10)    setAge(age);
11)    setSalary(2000);
12) }
13) public Employee(String name,int age,int salary)
14) {
15)     setSalary(salary);
16)     this(name,age);
17) }
18) //getter and setter methods goes here
19) public void printDetails()
20) {
21)     System.out.println(name+":"+age+":"+salary);
22) }
23) }
```

Test.java

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         Employee e1= new Employee();
6)         Employee e2= new Employee("Durga",50);
7)         Employee e3= new Employee("Ravi",40,5000);
8)         e1.printDetails();
9)         e2.printDetails();
10)        e3.printDetails();
11)    }
12) }
```

What is the result?

- A. Compilation fails in the Employee class
- B. null:0:0
Durga:50:0
Ravi:40:5000
- C. null:0:0
Durga:50:2000
Ravi:40:5000
- D. Compilation Fails in the Test class
- E. Compilation Fails in both Test and Employee classes

Answer: E



Q9. Given the following class:

```
1) public class CheckingAccount
2) {
3)     public int amount;
4)     //line-1
5) }
6) And the given the following main method located in another class:
7)
8) public static void main(String[] args)
9) {
10)    CheckingAccount acct= new CheckingAccount();
11)    //line-2
12) }
```

Which 3 pieces of code inserted independently, set the value of amount to 100?

A. At line-2 insert:

```
amount=100;
```

B. At line-2 insert:

```
this.amount=100;
```

C. At line-2 insert:

```
acct.amount=100;
```

D. At line-1 insert:

```
1) public CheckingAccount()
2) {
3)     amount=100;
4) }
```

E. At line-1 insert:

```
1) public CheckingAccount()
2) {
3)     this.amount=100;
4) }
```

F. At line-1 insert:

```
1) public CheckingAccount()
2) {
3)     acct.amount=100;
4) }
```

Answers: C, D and E



Exception Handling

- 1) Introduction
- 2) Runtime Stack Mechanism
- 3) Default Exception Handling in Java
- 4) Exception Hierarchy
- 5) Customized Exception Handling by using try & catch
- 6) Control Flow in try & catch
- 7) Methods to Print Exception Information
- 8) try with Multiple catch Blocks
- 9) finally Block
- 10) Difference between final, finally & finalize()
- 11) Various Possible Combinations of try-catch-finally
- 12) Control Flow in try-catch-finally
- 13) Control Flow in Nested try-catch-finally
- 14) throw Key Word
- 15) throws Key Word
- 16) Exception Handling Key Words Summary
- 17) Various Possible Compile-Time Errors in Exception Handling
- 18) Customized OR User defined Exceptions
- 19) Top 10 Exception
- 20) 1.7 Version Enhancements
 - ❖ try with Resources
 - ❖ Multi- catch Block



Introduction

- An Unwanted Unexpected Event that Disturbs Normal Flow of the Program is Called Exception.
Eg: SleepingException, TyrePunchedException, FileNotFoundException Etc...
- It is Highly Recommended to Handle Exceptions.
- The Main Objective of Exception Handling is Graceful Termination of the Program. i.e. we should Not Miss anything, we should Not Block any Resource.

Exception Handling:

- Exception Handling doesn't mean repairing an Exception.
- We have to define an Alternative Way to Continue Rest of the Program Normally.
- This Way of defining Alternative is nothing but Exception Handling.
- For Example if Our Programming Requirement is to Read Data from the File locating at London.
- At Runtime if London File is Not Available then Our Program should Not be terminated Abnormally.
- We have to provide Some Local File to Continue Rest of the Program Normally. This Way of defining Alternative is nothing but Exception Handling.

```
try {
    //Read Data from Remote File locating at London
}
catch (FileNotFoundException e) {
    //Use Local File and Continue Rest of the Program Normally
}
```

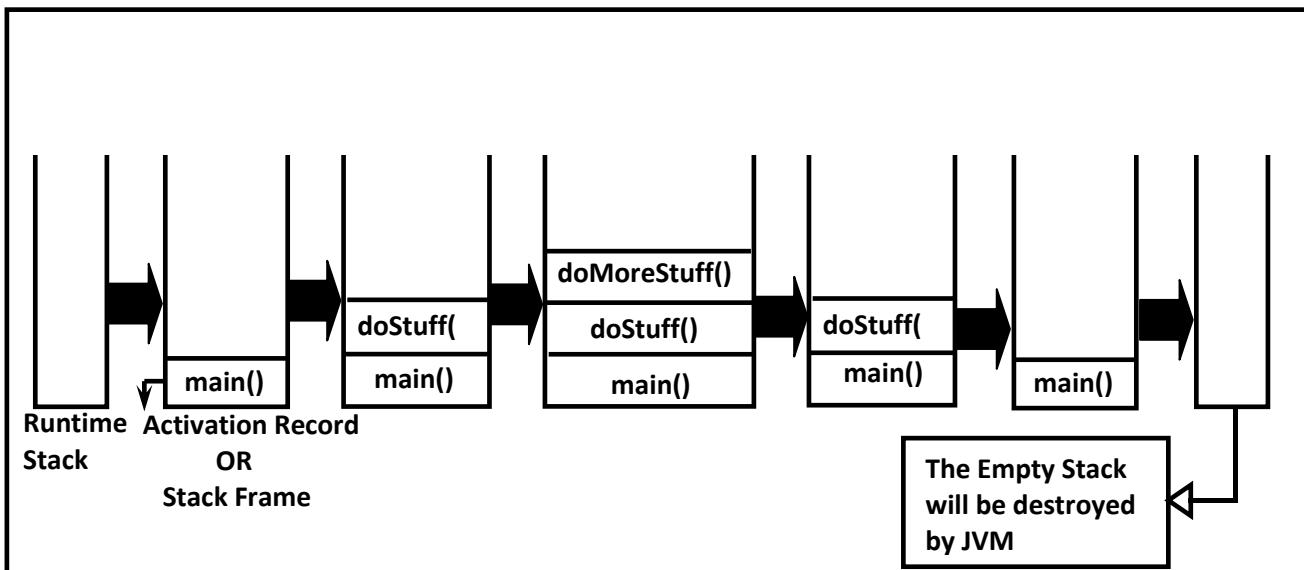
- Q) What is an Exception?
Q) What is the Purpose of Exception Handling?
Q) What is the Meaning of Exception Handling?

Runtime Stack Mechanism

- For Every Thread JVM will Create a Runtime Stack.
- Every Method Call performed by that Thread will be stored in the Corresponding Stack.
- Each Entry in the Stack is Called *Activation Record OR Stack Frame*.
- After completing Every Method Execution JVM Removes the Corresponding Entry from the Stack.
- After completing all Method Calls the Stack become Empty and that Empty Stack will be Destroyed by the JVM and the Program will be terminated Normally.



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



Default Exception Handling in Java

- In Our Java Program Inside a Method if an Exception raised, then that Method is Responsible to Create an Exception Object by including the following Information.
 - Name of Exception
 - Description of Exception
 - Location of Exception (Stack Trace)
- After creating Exception Object Method Handovers that Object to the JVM.
- JVM will Check whether Corresponding Method contain any Exception Handling Code OR Not.
- If the Method doesn't contain any Exception Handling Code then JVM Terminates that Method Abnormally and Removes Corresponding Entry from the Stack.
- JVM will Identify Caller Method and Check whether the Caller Method contain any Exception Handle Code OR Not.
- If Caller Method doesn't contain any Exception Handling Code then JVM Terminates that Caller Method and Removes Corresponding Entry from the Stack.
- This Process will be continued until `main()`.
- If the `main()` also doesn't contain Exception Handling Code then JVM Terminates `main()` Abnormally and Removes Corresponding Entry from the Stack.

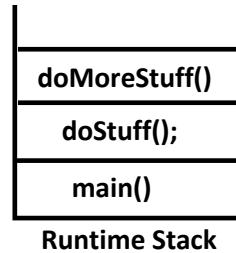


- Then JVM Handovers the Responsibility of Exception Handling to the *Default Exception Handler*, which is the Part of the JVM.
- Default Exception Handler Just Terminates the Program Abnormally and Prints Exception Information to the Console in the following Format.

Exception in thread: "Xxx" Name of the Exception: Description
Stack Trace

Examples:

```
class Test {  
    public static void main(String[] args) {  
        doStuff();  
    }  
    public static void doStuff() {  
        doMoreStuff();  
    }  
    public static void doMoreStuff() {  
        System.out.println(10/0);  
    }  
}
```



```
RE: Exception in thread "main" java.lang.ArithmetricException: / by zero  
at Test.doMoreStuff(Test.java:9)  
at Test.doStuff(Test.java:6)  
at Test.main(Test.java:3)
```

```
class Test {  
    public static void main(String[] args) {  
        doStuff();  
    }  
    public static void doStuff() {  
        doMoreStuff();  
        System.out.println(10/0);  
    }  
    public static void doMoreStuff() {  
        System.out.println("Hello");  
    }  
}
```

```
Hello  
Exception in thread "main" java.lang.ArithmetricException: / by zero  
at Test.doStuff(Test.java:7)  
at Test.main(Test.java:3)
```



```
class Test {  
    public static void main(String[] args) {  
        doStuff();  
        System.out.println(10/0);  
    }  
    public static void doStuff() {  
        doMoreStuff();  
        System.out.println("Hi");  
    }  
    public static void doMoreStuff() {  
        System.out.println("Hello");  
    }  
}
```

```
Hello  
Hi  
Exception in thread "main"  
java.lang.ArithmetricException: / by zero  
at Test.main(Test.java:4)
```

Note:

- In Our Program if all Methods Terminated Normally, then Only the Program will be Terminated Normally.
- In Our Program if at least One Method terminates Abnormally then the Program Termination is Abnormal Termination.

Exception Hierarchy

- Throwable Class Acts as Root for Exception Hierarchy.
- Throwable Class contains 2 Child Classes *Exception* and *Error*

Exception: Most of the Cases Exceptions are Caused by Our Program and these are Recoverable.

Eg:

- If Our Programming Requirement is to Read Data from the File locating at London.
- At Runtime if London File is Not Available then we get *FileNotFoundException*.
- If *FileNotFoundException* Occurs we can Provide Local File to Continue Rest of the Program Normally.
- Programmer is Responsible to Recover Exception.

Error:

- Most of the Cases Errors are Not Caused by Our Program and these are Due to Lack of System Resources.
- Errors are Non- Recoverable.

Eg:

- If *OutOfMemoryError* Occurs, being a Programmer we can't do anything and the Program will be terminated Abnormally.
- System Admin OR Server Admin is Responsible to Increase Heap Memory.



Checked Vs Unchecked Exception:

Checked Exceptions:

- The Exceptions which are Checked by the Compiler for Smooth Execution of the Program at Runtime are Called Checked Exceptions.
- Compiler Checks whether we are handling Checked Exceptions OR Not. If we are Not handling then we will get Compile Time Error.

Eg:

HallTicketMissingException, PenNotWorkingException, FileNotFoundException, Etc.

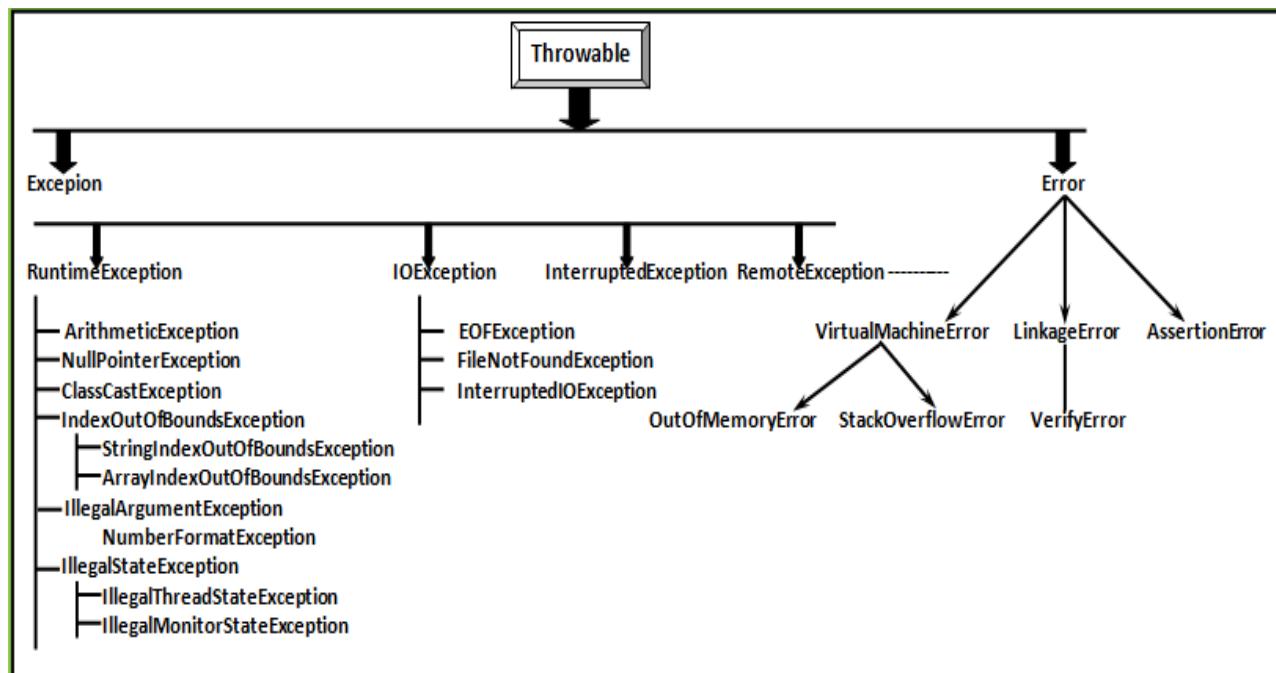
Unchecked Exceptions:

- The Exceptions which are Not Checked by the Compiler are Called Unchecked Exception.
- Compiler won't Check whether we are Handle OR Not Unchecked Exceptions.

Eg: ArithmeticException, NullPointerException, BombBlostException, Etc.

Note:

- Whether the Exception is Checked OR Unchecked Compulsory it will Occur at Runtime Only.
- There is No Chance of occurring any Exception at Compile Time.
- Runtime Exceptions and its Child Classes, Errors and its Child Classes are Unchecked Exceptions Except these all remaining are Considered as Checked Exceptions.





Fully Checked Vs Partially Checked:

A Checked Exception is Said to be Fully Checked if and Only if all its Child Classes also Checked.

Eg: IOException, InterruptedException, ServletException, Etc...

A Checked Exception is Said to be Partially Checked if and Only if Some of its Child Classes are Unchecked.

Eg: Throwable, Exception.

Note: The Only Possible Partially Checked Exceptions in Java are

- Throwable
- Exception

Describe the Behaviour of the following Exceptions

- 1) IOException → Fully Checked
- 2) RuntimeException → Unchecked
- 3) InterruptedException → Fully Checked
- 4) Error → Unchecked
- 5) Throwable → Partially Checked
- 6) ArithmeticException → Unchecked
- 7) NullPointerException → Unchecked
- 8) Exception → Partially Checked
- 9) FileNotFoundException → Fully Checked

Customized Exception Handling by using try - catch

- It is Highly Recommended to Handle Exceptions.
- The Code which May Raise an Exception is Called *Risky Code*.
- We have to Place that Risky Code Inside *try* Block and Corresponding Handle Code Inside *catch* Block.

```
try
{
    //Risky Code
}
catch (Exception e)
{
    //Handling Code
}
```



Without try - catch

```
class Test {  
    public static void main(String[] args) {  
        System.out.println("Statement 1");  
        System.out.println(10/0);  
        System.out.println("Statement 2");  
    }  
}
```

Statement 1
RE: Exception in thread "main"
java.lang.ArithmaticException: / by zero
at Test.main(Test.java:4)

Abnormal Termination

With try - catch

```
class Test {  
    public static void main(String[] args) {  
        System.out.println("Statement 1");  
        try {  
            System.out.println(10/0);  
        }  
        catch(ArithmaticException e) {  
            System.out.println(10/2);  
        }  
        System.out.println("Statement 2");  
    }  
}
```

Normal Termination

Statement 1
5
Statement 2

Control Flow in try - catch:

```
try {  
    Statement 1;  
    Statement 2;  
    Statement 3;  
}  
catch(X e) {  
    Statement 4;  
}  
Statement 5;
```

Case 1: If there is No Exception → 1, 2, 3, 5, Normal Termination.

Case 2: If an Exception raised in Statement 2 and Corresponding catch Block Matched
→ 1, 4, 5, Normal Termination.

Case 3: If an Exception raised at Statement 2 and Corresponding catch Block Not
Matched → 1 followed by Abnormal Termination.

Case 4: If an Exception raised at Statement-4 OR Statement-5 then it's Always
Abnormal Termination.

Conclusions:

- Within the try Block if any where an Exception raised then Rest of the try Block won't be executed even though we handled that Exception.
- Hence within the try Block we have to Take Only Risky Code and Hence Length of the try Block should be as Less as Possible.
- If there is any Statement which raises an Exception and it is Not Part of the try Block then it is Always Abnormal Termination.
- In Addition to try Block there May be a Chance of raising an Exception Inside catch and finally Blocks Also.



Methods to Print Exception Information

Throwable Class defines the following Methods to Print Exception Information.

Method	Printed Format
printStackTrace()	Name of the Exception: Description Stack Trace
toString()	Name of the Exception: Description
getMessage()	Description

```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println(10/ 0);  
        }  
        catch (ArithmaticException e) {  
            e.printStackTrace(); //RE: java.lang.ArithmaticException: / by zero  
                           at Test.main(Test.java:4)  
            System.out.println(e); //RE: java.lang.ArithmaticException: / by zero  
            System.out.println(e.getMessage()); /// by zero  
        }  
    }  
}
```

Note: Internally Default Exception Handler Uses *printStackTrace()* to Print Exception Information to the Console.



try with Multiple catch Blocks

The way of handling an Exception is varied from Exception to Exception. Hence for Every Exception Type we have to define a Separate catch Block. Hence try with Multiple catch Blocks is Possible and Recommended to Use.

```
try {  
    .....  
    .....  
}  
catch (Exception e) {  
    .....  
    .....  
}
```

Not Recommended

```
try {  
    .....  
    .....  
}  
catch (ArithmaticException e) {  
    //Perform these an Alternative  
    ArithmaticException  
}  
catch (NullPointerException e) {  
    //Handling Related to null  
}  
catch (FileNotFoundException) {  
    //Use Local File Instead of Remote File  
}  
catch (SQLException e) {  
    //Use MySQL DB Instead of Oracle  
}  
catch (Exception e) {  
    Default Exception Handling  
}
```

Recommended

Note:

- If try with Multiple catch Blocks Present then the Order of catch Blocks are Very Important. It should be from Child to Parent.
- By Mistake if we are trying to Take Parent to Child then we will get Compile Time Error Saying: exception XXX has already been caught

```
try {}  
catch (Exception e) {}  
catch (ArithmaticException e) {} //CE: exception ArithmaticException has already been caught
```

```
try {}  
catch (ArithmaticException e) {}  
catch (Exception e) {}
```

For any Exception if we are writing 2 Same catch Blocks we will get Compile Time Error.



```
try {}  
catch(ArithmeticException e) {}  
catch(ArithmeticException e) {} //CE: error: exception ArithmeticException has already been caught
```

finally Block

- It is Never Recommended to Define Clean-up Code Inside try Block. Because there is No Guaranty for the Execution of Every Statement Inside try Block.
- It is Never Recommended to Define Clean-up Code Inside catch Block. Because if there is No Exception then catch Block won't be executed.
- Hence we required Some Place to Maintain Clean-up Code which should be executed Always irrespective of whether Exception raised OR Not raised and whether Handled OR Not Handled. Such Type of Best Place is Nothing but finally Block.
- Hence the Main Objective of finally Block is to Maintain Clean-up Code.

```
try {  
    //Risky Code  
}  
catch(X e) {  
    //Handling Code  
}  
finally {  
    //Clean Up Code  
}
```

- The Specialty of finally Block is it will be executed Always irrespective of whether Exception raised OR Not raised and whether Exception Handled OR Not Handled.

```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println ("try");  
        }  
        catch (Exception e) {  
            System.out.println ("catch");  
        }  
        finally {  
            System.out.println ("finally");  
        }  
    }  
}
```

```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println ("try");  
            System.out.println (10/0);  
        }  
        catch (Exception e) {  
            System.out.println ("catch");  
        }  
        finally {  
            System.out.println ("finally");  
        }  
    }  
}
```



```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println ("try");  
            System.out.println (10/0);  
        }  
        catch (NullPointerException e) {  
            System.out.println ("catch");  
        }  
        finally {  
            System.out.println ("finally");  
        }  
    }  
}
```

```
try  
finally  
Exception in thread "main"  
java.lang.ArithmetricException: / by zero  
at Test.main(Test.java:5)
```

finally Vs return:

If return Statement Present Inside try OR catch Blocks 1st finally will be executed and after that Only return Statement will be Considered i.e. finally Block Dominates return Statement.

```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println("try");  
            return;  
        }  
        catch(Exception e) {  
            System.out.println("catch");  
        }  
        finally {  
            System.out.println("finally");  
        }  
    }  
}
```

```
try  
finally
```

If try-catch-finally Blocks having return Statements then finally Block return Statement will be Considered i.e. finally Block return Statement has More Priority than try and catch Block return Statements.



```
class Test {  
    public static void main(String[] args) {  
        System.out.println(m1()); //999  
    }  
    public static int m1() {  
        try {  
            return 777;  
        }  
        catch(Exception e) {  
            return 888;  
        }  
        finally {  
            return 999;  
        }  
    }  
}
```

finally Vs System.exit(0):

There is Only One Situation where the finally Block won't be executed that is whenever we are `System.exit(0)`.

Whenever we are using `System.exit(0)` then JVM itself will be Shutdown and hence finally Block won't be executed. That is `System.exit(0)` Dominates finally Block.

```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println("try");  
            System.exit(0);  
        }  
        catch(Exception e) {  
            System.out.println("catch");  
        }  
        finally {  
            System.out.println("finally");  
        }  
    }  
}
```

try

System.exit(0):

- We can Use this Method to Exit (Shut Down) the System (JVM) Programmatically.
- The Argument Represents as Status Code.
- Instead of 0 we can Pass any Valid int Value.
- 0 Means Normal Termination, Non- Zero Means Abnormal Termination.
- So this status code internally used by JVM.



- Whether it is 0 OR Non- Zero Effect is Same in Our Program but this Number Internally used by JVM.

Difference between final, finally and finalize

final:

- final is a Modifier is Applicable for Classes, Methods and Variables.
- If a Class declared as final then we can't Create Child Class. That is Inheritance is Not Possible for final Classes.
- If a Method declared as final then we can't Override that Method in Child Classes.
- If a Variable declared as final then we can't Perform Re- Assignment for that Variable.

finally:

- finally is a Block Always associated with try-catch to Maintain Clean Up Code.
- The Specialty of finally Block is it will be executed Always Irrespective of whether Exception raised OR Not and whether Handled OR Not Handled.

finalize():

- finalize() is a Method Always Called by the Garbage Collector Just before Destroying an Object to Perform Clean Up Activities.
- Once finalize() Completes Automatically Garbage Collector Destroys that Object.

Note:

- finally() is Responsible to Perform Object Level Clean-Up Activities whereas finally Block is Responsible to Perform try Block Level Clean-Up Activities i.e. whatever Resources we Opened at the Time of try Block will be Closed Inside finally Block
- It is Highly Recommended to Use finally Block than finalize() because we can't Expect Exact Behavior of Garbage Collector. It is JVM Vendor Dependent.

Control Flow in try - catch - finally:

```
try {  
    System.out.println("Statement 1");  
    System.out.println("Statement 2");  
    System.out.println("Statement 3");  
}  
catch(X e) {  
    System.out.println("Statement 4");  
}  
finally {  
    System.out.println("Statement 5");  
}  
System.out.println("Statement 6");
```

Case 1: If there is No Exception. 1, 2, 3, 5 and 6 → Normal Termination.



Case 2: If an Exception raised at Statement 2 and Corresponding catch Block Matched. then 1, 4, 5, 6 → Normal Termination.

Case 3: If an Exception raised at Statement 2 and Corresponding catch Block Matched. then 1 and 5 Abnormal Termination.

Case 4: If an Exception raised at Statement 4 then it is Always Abnormal Termination but before that finally Block will be executed.

Case 5: If an Exception raised at Statement 5 OR Statement 6 then it is Always Abnormal Termination.

Control Flow in Nested try - catch - finally:

```
try {  
    System.out.println("Statement 1");  
    System.out.println("Statement 2");  
    System.out.println("Statement 3");  
    try {  
        System.out.println("Statement 4");  
        System.out.println("Statement 5");  
        System.out.println("Statement 6");  
    }  
    catch(X e) {  
        System.out.println("Statement 7");  
    }  
    finally {  
        System.out.println("Statement 8");  
    }  
    System.out.println("Statement 9");  
}  
catch(X e) {  
    System.out.println("Statement 10");  
}  
finally {  
    System.out.println("Statement 11");  
}  
System.out.println("Statement 12");
```

Case 1: If there is No Exception then 1, 2, 3, 4, 5, 6, and 8, 9, 11, 12 Normal Termination.

Case 2: If an Exception raised at Statement 2 and Corresponding catch Block Matched 1, 10, 11, and 12 Normal Terminations.



Case 3: If an Exception raised at Statement 2 and Corresponding catch Block is Not Matched 1 and 11 Abnormal Termination.

Case 4: If an Exception raised at Statement 5 and Corresponding Inner catch Block has Matched 1, 2, 3, 4, 7, 8, 9, 11, 12 Normal Termination.

Case 5: If an Exception raised at Statement 5 and Inner catch Block has Not Matched but Outer catch Block has Matched. 1, 2, 3, 4, 8, 10, 11, and 12 Normal Termination.

Case 6: If an Exception raised at Statement 5 and Both Inner and Outer catch Blocks are Not Matched then 1, 2, 3, 4, 8, and 11 Abnormal Termination.

Case 7: If an Exception raised at Statement 7 and Corresponding catch Block Matched 1, 2, 3, , 8, 10, 11, and 12 Normal Termination.

Case 8: If an Exception raised at Statement 7 and Corresponding catch Block Not Matched 1, 2, 3, , 8, and 11 Abnormal Termination.

Case 9: If an Exception raised at Statement 8 and Corresponding catch Block has Matched 1, 2, 3, , 10, 11, and 12 Normal Termination.

Case 10: If an Exception raised at Statement 8 and Corresponding catch Block Not Matched 1, 2, 3, , 11 Abnormal Termination.

Case 11: If an Exception raised at Statement 9 and Corresponding catch Block Matched 1, 2, 3, , 8, 10, 11, and 12 Normal Termination.

Case 12: If an Exception raised at Statement 9 and Corresponding catch Block Not Matched 1, 2, 3, , 8, and 11 Abnormal Termination.

Case 13: If an Exception raised at Statement 10 is Always Abnormal Termination but before that finally Block 11 will be executed.

Case 14: If an Exception raised at Statement 11 OR 12 it is Always Abnormal Termination.

Note:

- We can Take try - catch - finally Inside try Block i.e. Nesting of try - catch - finally is Always Possible.
- More Specific Exceptions can be handled by Inner catch Block and Generalized Exceptions are handled by Outer catch Blocks.
- Once we entered into the try Block without executing finally Block the Control Never Comes Up.
- If we are Not entering into the try Block then finally Block won't be executed.



```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println(10/0);  
        }  
        catch(ArithmeticException e) {  
            System.out.println(10/0);  
        }  
        finally {  
            String s = null;  
            System.out.println(s.length());  
        }  
    }  
}
```

- 1) CE
- 2) RE: ArithmeticException
- 3) RE: NullPointerException X
- 4) RE: ArithmeticException and RE: NullPointerException

Note: Default Exception Handler can Handle Only One Exception at a Time i.e. the Most Recently raised Exception.

Various Possible Combinations of try-catch-finally

- In try-catch-finally Order is Important.
- Inside try-catch-finally we can take try-catch-finally that is nesting of try-catch-finally is always possible.
- Whenever we are taking try Compulsory we have to Take either catch OR finally Blocks i.e. try without catch OR finally is Invalid.
- Whenever we are writing catch Block Compulsory we have to write try Block i.e. catch without try is Invalid.
- Whenever we are writing finally Block Compulsory we should write try i.e. finally without try is Invalid.
- For try-catch-finally Blocks Curly Braces are Mandatory.
- We can't write 2 catch Blocks for the Same Exception Otherwise we will get CE.

```
try {}  
catch (X e) {}
```

```
try {}  
catch (X e) {}  
catch (Y e) {}
```

```
try {}  
catch (X e) {}  
catch (X e) {} // CE: exception ArithmeticException has already been caught
```

```
try {}  
finally {}
```

```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
```



```
finally {} //CE: 'finally' without 'try'
```

```
catch (X e) {} //CE: 'catch' without 'try'
```

```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
```

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

```
catch {} //CE: 'catch' without 'try'
```

```
try {}
```

```
catch (X e) {}
```

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

```
catch (Y e) {} //CE: 'catch' without 'try'
```

```
try {}
```

```
catch (X e) {}
```

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

```
finally {} //CE: 'finally' without 'try'
```

```
try {}
```

```
finally {}
```

```
catch (X e) {} //CE: 'catch' without 'try'
```

```
try {}
```

```
catch (X e) {}
```

```
try {}
```

```
finally {}
```

```
try {}
```

```
catch (X e) {}
```

```
finally {}
```

```
finally {} //CE: 'finally' without 'try'
```

```
try {}
```

```
catch (X e) {}
```

```
try {}
```

```
catch (Y e1) {}
```

```
}
```

```
try {}
```

```
catch (X e) {}
```

```
finally {}
```

```
try {}
```

```
catch (Y e1) {}
```

```
}
```

```
try {
```

```
    try {} //CE: 'try' without 'catch', 'finally' or resource declarations
```

```
}
```

```
catch (X e) {}
```



```
try //CE: '{' expected
    System.out.println("Hello");
catch (X e1) {} //CE: 'catch' without 'try'
```

```
try {}
catch (X e) //CE:{' expected
    System.out.println("Hello");
```

```
try {}
catch (NullPointerException e1) {}
finally //CE: '{' expected
    System.out.println("Hello");
```

throw Keyword:

- Sometimes we can Create Exception Object Explicitly and we can Handover Our Created Exception Object to the JVM Manually. For this we have to Use throw key Word.

Eg:

```
throw new ArithmeticException("/by zero");
```

throw Key Word Handover Our
Created Exception Objet to the JVM
Manually

Creation of Exception Object
Explicitly

- In General we can Use throw Key Word for Customized Exceptions but Not for pre-defined Exceptions.

The Result of following 2 Programs is Exactly Same.

Case – 1

```
class Test {
    public static void main(String[] args) {
        System.out.println(10/0);
    }
}

//Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Case – 2

```
class Test1 {
    public static void main(String[] args) {
        throw new ArithmeticException("/ by zero Explicitly");
    }
}

//Exception in thread "main" java.lang.ArithmeticException: / by zero Explicitly
```



In the Case – 1 `main()` is Responsible to Create Exception Object and Handover to the JVM. This Total Activity will be performed Internally.

In the Case – 2 Programmer creating Exception Object Explicitly and Handover to the JVM Manually.

Hence the Main Purpose of `throw` Key Word is to Handover Our Created Exception Object to the JVM Manually.

Case 1: `throw e;`

If 'e' Refers 'null' then we will get `NullPointerException`.

```
class Test {  
    static ArithmeticException e = new ArithmeticException();  
    public static void main(String[] args) {  
        throw e;  
    }  
}
```

RE: Exception in thread "main" java.lang.ArithmeticException

```
class Test {  
    static ArithmeticException e;  
    public static void main(String[] args) {  
        throw e;  
    }  
}
```

//RE: Exception in thread "main" java.lang.NullPointerException

Case 2: After `throw` Statement we are Not allowed to write any Statements Directly Otherwise we will get Compile Time Error Saying unreachable statement.

```
class Test {  
    public static void main(String[] args) {  
        System.out.println(10/0);  
        System.out.println("Hello");  
    }  
}
```

RE: Exception in thread "main" java.lang.ArithmeticException: / by zero

```
class Test {  
    public static void main(String[] args) {  
        throw new ArithmeticException("/by zero");  
        System.out.println("Hello"); //CE: unreachable statement  
    }  
}
```



Case 3: We can Use throw Key Word Only for Throwable Types. Otherwise we will get Compile Time Error Saying incompatible types.

```
class Test {  
    public static void main(String[] args) {  
        throw new Test();  
    }  
}
```

CE: incompatible types
required: Throwable
found: Test

```
class Test extends RuntimeException {  
    public static void main(String[] args) {  
        throw new Test();  
    }  
}
```

RE: Exception in thread "main" Test

throws Key Word

In Our Program if there is any Chance of raising Checked Exception then Compulsory we should Handled that Checked Exception Otherwise we will get Compile Time Error Saying unreported exception XXX; must be caught or declared to be thrown.

```
import java.io.PrintWriter;  
class Test {  
    public static void main(String[] args) {  
        PrintWriter out = new PrintWriter("Abc.txt");  
        out.println("Hello");  
    }  
}  
CE: unreported exception FileNotFoundException; must  
be caught or declared to be thrown
```

```
class Test {  
    public static void main(String args[]) {  
        Thread.sleep(5000);  
    }  
}  
CE: unreported exception  
InterruptedException; must be caught or  
declared to be thrown
```

We can Handle this Compile Time Error in 2 Ways.

- 1) By Using try-catch
- 2) By Using throws Key Word

1st Way: By Using try - catch Block

```
class Test {  
    public static void main(String args[]) {  
        try {  
            Thread.sleep(5000);  
        }  
        catch (InterruptedException e) {}  
    }  
}
```



2nd Way: By Using throws Key Word

- We can use throws Key Word to Delegate the Responsibility of Exception Handling to the Caller Method (It May be Another Method OR JVM). Then Caller is Responsible to Handle that Checked Exception.
- throws Key Word required Only for Checked Exceptions.
- Usage of throws Key Word for Unchecked Exceptions there is No Use.
- throws Key Word required Only to Convince Compiler and it doesn't Prevent Abnormal Termination of the Program.
- Hence Recommended to Use try- catch- finally Over throws Key Word.

```
class Test {  
    public static void main(String[] args) throws InterruptedException {  
        Thread.sleep(5000);  
    }  
}
```

Conclusions:

throws
Key Word

- 1) We can Use to Delegate the Responsibility of Exception Handling to the Caller.
- 2) It is required Only for Checked Exceptions and for Unchecked Exceptions there is No Use.
- 3) It is required Only to Convince Compiler and its Usage doesn't Prevent Abnormal Termination of the Program.

```
class Test {  
    public static void main(String[] args) throws InterruptedException {  
        doStuff();  
    }  
    public static void doStuff() throws InterruptedException {  
        doMoreStuff();  
    }  
    public static void doMoreStuff() throws InterruptedException {  
        Thread.sleep(5000);  
    }  
}
```

In the Above Program if we Remove at-least One throws Statement then the Code won't Compile. We will get CE: *unreported exception InterruptedException; must be caught or declared to be thrown*

Case 1: We can Use throws Key Word Only for Methods and Constructors but Not for Classes.



```
class Test throws Exception X{
    Test() throws Exception {} ✓
    public static void m1() throws Exception {} ✓
}
```

Case 2:

We can Use throws Key Word Only for Throwable Types but Not for Normal Java Classes. Otherwise we will get Compile Time Error Saying *incompatible types*

required: java.lang.Throwable
found: Test

```
class Test {
    public static void main(String[] args) throws Test {}
```

CE: incompatible types
required: Throwable
found: Test

```
class Test extends RuntimeException/ Exception/ Throwable {
    public static void main(String[] args) throws Test {} ✓
```

Case 3:

```
class Test {
    public static void main(String[] args) {
        throw new Exception();
    }
}
```

CE: unreported exception Exception; must be caught or declared to be thrown

Checked Exception

```
class Test {
    public static void main(String[] args) {
        throw new Error();
    }
}
```

RE: Exception in thread "main" java.lang.Error at Test.main(Test.java:3)

Unchecked Exception

Case 4:

Inside try Block, if there is No Chance of raising an Exception then we can't write catch Block for that Exception. Otherwise we will get Compile Time Error Saying CE: *exception XXX is never thrown in body of corresponding try statement*. But this Rule is Applicable Only for Fully Checked Exceptions.

```
class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Hello");
        }
        catch (ArithmaticException e) {}
    }
}
```

Unchecked Exception

Output: Hello

```
class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Hello");
        }
        catch (Exception e) {}
    }
}
```

Partially Checked Exception

Output: Hello



```
import java.io.IOException;
class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Hello");
        }
        catch (IOException e) {}
    }
}
```

Fully Checked Exception

CE: exception IOException is never thrown in body of corresponding try statement

```
class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Hello");
        }
        catch (InterruptedException e) {}
    }
}
```

Fully Checked Exception

CE: exception InterruptedException is never thrown in body of corresponding try statement

```
class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Hello");
        }
        catch (Error e) {}
    }
}
```

Unchecked Exception

Output: Hello

Summary of Exception Handling Key Words

- 1) try → To Maintain Risky Code
- 2) catch → To Maintain Handling Code
- 3) finally → To Maintain Clean Up Code
- 4) throw → To Hand-Over Our Created Exception Object to the JVM Manually
- 5) throws → To Delegate Responsibility of Exception Handling to the Caller Method

Various Possible Compile Time Errors in Exception Handling

- 1) unreported exception Exception; must be caught or declared to be thrown
- 2) Exception XXX has already been caught
- 3) Exception XXX is never thrown in body of corresponding try statement
- 4) unreachable statement
- 5) incompatible types
required: java.lang.Throwable
found: Test



-
- 6) try without catch or finally
 - 7) catch without try
 - 8) finally without try

Customised OR User defined Exceptions

Sometimes to Meet Programming Requirement we have to Create Our Own Exceptions which are Nothing but Customized Exceptions.

Eg: TooYoungException, TooOldException, InsufficientFundsException, Etc.....

```
class TooYoungException extends RuntimeException {  
    TooYoungException(String s) {  
        super(s);  
    }  
}  
class TooOldException extends RuntimeException {  
    TooOldException(String s) {  
        super(s);  
    }  
}  
class CustomizedException {  
    public static void main(String[] args) {  
        int age = Integer.parseInt(args[0]);  
        if(age > 60) {  
            throw new TooYoungException("Please Wait Some More Time U will get Best Match");  
        }  
        else if(age < 18) {  
            throw new TooOldException("Your Age Already Crossed Marriage Age No Chance of getting Match");  
        }  
        else {  
            System.out.println("U will get Match Details Soon by Email...!");  
        }  
    }  
}
```

java CustExceptionDemo 60
U will get Match Details Soon by Email...!

java CustExceptionDemo 70
Exception in thread "main" TooYoungException: Please Wait Some More Time U will get Best Match
at CustExceptionDemo.main(CustExceptionDemo.java:15)

java CustExceptionDemo 50
U will get Match Details Soon by Email...!

java CustExceptionDemo 15
Exception in thread "main" TooOldException: Your Age Already Crossed Marriage Age No Chance of getting Match
at CustExceptionDemo.main(CustExceptionDemo.java:18)



Note:

- throw Key Word is Best Suitable for Customized Exceptions but Not for Pre-defined Exceptions.
- It is Highly Recommended to define Customized Exceptions as Unchecked i.e. we have to extends *RuntimeException* but Not *Exception*.
- To Make Description Available to Parent Class, from which Default Exception Handler get these Description.

Top 10 Exceptions

Based on the Person who is raising Exception, all Exceptions are divided into 2 Types.

- 1) JVM Exceptions
- 2) Programmatic Exceptions.

JVM Exceptions: The Exceptions which are raised Automatically by the JVM whenever a Particular Event Occurs Such Type of Exceptions are Called JVM Exceptions.

Eg: *ArithmaticException*, *ArrayIndexOutOfBoundsException*, *NullPointerException* Etc.

Programmatic Exceptions:

The Exceptions which are raised Explicitly either by Programmer OR by API Developer are Called Programmatic Exceptions.

Eg: *IllegalArgumentException*, *TooYoungException*, Etc.

ArrayIndexOutOfBoundsException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Automatically by the JVM Whenever we are trying to Access Array Element with Out of Range Index.

Eg:

```
int[] a = new int[10];
System.out.println(a[0]); //0
System.out.println(a[100]); //RE: java.lang.ArrayIndexOutOfBoundsException: 100
System.out.println(a[-100]); //RE: java.lang.ArrayIndexOutOfBoundsException: -100
```

NullPointerException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Automatically by the JVM whenever we are trying to Perform any Method Call on null Reference.

Eg:

```
String s = null;
System.out.println(s.length()); //RE: Exception in thread "main" java.lang.NullPointerException
```

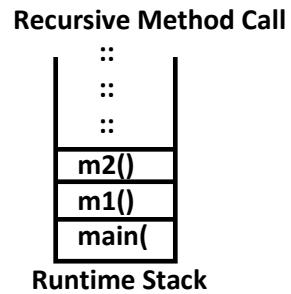
StackOverflowError:

It is the Child Class of Error and Hence it is Unchecked.

Raised Automatically by the JVM whenever we are trying to Perform Recursive Method Call.



```
class Test {  
    public static void m1() {  
        m2();  
    }  
    public static void m2() {  
        m1();  
    }  
    public static void main(String[] args) {  
        m1(); //RE: java.lang.StackOverflowError  
    }  
}
```



ClassCastException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Automatically by the JVM whenever we are trying to Type Cast Parent Object to Child Type.

Eg:

```
String s = new String("abc");  
Object o = (Object)s; //√
```

```
Object o = new Object();  
String s = (String)o;
```

RE: java.lang.ClassCastException: java.lang.Object cannot be cast to java.lang.String

```
Object o = new String("abc");  
String s = (String)o; //valid //√
```

NoClassDefFoundError:

- It is the Child Class of *Error* and Hence it is Unchecked.
- Raised Automatically by the JVM whenever JVM Unable to find required *.class* File.

Eg: java Test

If *Test.class* is Not Available then we will get *RuntimeException* Saying
java.lang.NoClassDefFoundError: Test

ExceptionInInitializerError:

- It is the Child Class of *Error* and Hence it is Unchecked.
- Raised Automatically by the JVM if any Exception Occurs while executing Static Variable Assignments and Static Blocks.

```
class Test {  
    static int i = 10/0;  
}
```

RE: Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArithmaticException: / by zero



```
class Test {  
    static {  
        String s = null;  
        System.out.println(s.length());  
    }  
}
```

Exception in thread "main"
java.lang.ExceptionInInitializerError
Caused by: java.lang.NullPointerException

IllegalArgumentException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Exprectly by the Programmer OR by API Developer to Indicate that a Method has been invoked with Illegal Argument.

```
class Test {  
    public static void main(String[] args) {  
        Thread t = new Thread();  
        t.setPriority(10);  
        t.setPriority(100);  
    }  
}
```

RE: Exception in thread "main" java.lang.IllegalArgumentException
at java.lang.Thread.setPriority(Thread.java:1125)
at Test.main(Test.java:5)

- The Valid Range of Thread Priorities is 1 to 10. If we are trying to Set the Priority with any Other Value we will get Runtime Exception Saying *IllegalArgumentException*.

NumberFormatException:

- It is the Direct Child Class of *IllegalArgumentException*, which is Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Exprectly either by Programme OR by API Developer to Indicate that we are trying to Convert String to Number but the String is Not Properly Formatted.

Eg:

```
int i = Integer.parseInt("10");  
int i = Integer.parseInt("Ten"); //RE: java.lang.NumberFormatException: For input string: "Ten"
```

IllegalStateException:

- It is the Child Class of *RuntimeException* and Hence it is Unchecked.
- Raised Exprectly either by Programmer OR by API Developer to Indicate that a Method has been invoked at Wrong Time.

Examples:

- 1) After Starting a Thread we are Not allowed to Re- Start the Same Thread Once Again. Otherwise we will get Runtime Exception Saying *IllegalThreadStateException*.

```
Thread t = new Thread();  
t.start();  
t.start(); //RE: java.lang.IllegalThreadStateException
```

- 2) Once Session Expires we are Not allow to Call any Method on that Session Object. If we are trying to Call we will get Runtime Exception Saying *IllegalStateException*.



```
HttpSession session = req.getSession();
System.out.println(session.getId()); //Valid
session.invalidate();
System.out.println(session.getId()); //RE: IllegalStateException
```

AssertionError:

It is the Child Class of Error and Hence it is Unchecked.
Raised Explicitly to Indicate that assert Statement Fails.

Eg: assert(x > 10);
if(x != 10) then we will get Runtime Exception Saying *AssertionError*.

<u>Exception/ Error</u>	<u>Raised By</u>
1) ArrayIndexOutOfBoundsException	
2) NullPointerException	
3) StackOverflowError	
4) ClassCastException	
5) NoClassDefFoundError	
6) ExceptionInInitializerError	
7) IllegalArgumentException	
8) NumberFormatException	
9) IllegalStateException	
10) AssertionError	
	Raised by JVM and Hence these are JVM Exceptions
	Raised by Explicitly either by Programmer OR by API Developer and Hence they are Programmatic Exceptions



1.7 Version Enhancements

In 1.7 Version as the Part of Exception Handling the following 2 Concepts Introduced.

- try with Resources
- Multi catch Block

try with Resources: Until 1.6 Version it is Highly Recommended to write finally Block to Close All Resources which are Opened as the Part of try Block.

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("Input.txt"));
    //Use br based on Our Requirement
}
catch (InterruptedException e) {
    //Handling Code
}
finally {
    if (br != null) {
        br.close();
    }
}
```

The Problems in this Approach are:

- Compulsory Programmer is required to Close All Opened Resources in finally Block. It Increases Length of the Code and Reduces Readability.
- It Increases the Complexity of the Programming.

To Overcome these Problems SUN People Introduce *try with Resources* in 1.7 Version.

```
try (BufferedReader br = new FileReader("Input.txt")) {      Resource
}
//Use br based on Our Requirements. br will be Closed Automatically Once the Control
//Reaches End of try either Normally OR Abnormally
}
catch (InterruptedException e) {
```

- The Main Advantage of try with Resources is the Resources which are Opened as the Part of try Block will be Closed Automatically and we are Not required to Close Explicitly. It Reduces Complexity of the Programming.
- It is Not required to write finally Block Explicitly and Hence Length of the Code will be Reduced and Readability will be Improved.



Conclusions:

- We can Declare Multiple Reasons and All these Resources should be Separated with ‘;’.

Syntax: try (R1; R2; R3) {

}

Eg: try (FileWriter fw = new FileWriter("Output.txt");
FileWriter fw = new FileWriter("Input.txt");) {

}

- All Resources should be AutoClosable Resources.
- A Resource is Said to be AutoClosable
 - Corresponding Class Implements `java.lang.AutoClosable` Interface either Directly OR In- Directly.
- This Interface introduced in 1.7 Version and it contains Only One Method
 - `public void close();`
- All Network OR Database Related OR File IO Related Resources Implements AutoClosable Interface. Being a Programmer we are Not required to do anything.
- All Resource Reference Variables are Implicitly final. Hence within the try Block we can't Perform Re- Assignment.

```
import java.io.*;  
class TryWithResources {  
    public static void main (String args[]) throws Exception {  
        try (BufferedReader br = new BufferedReader (new FileReader ("abc.txt"))){  
            br = new BufferedReader (new FileReader ("Input.txt"));  
        }  
    } //CE: auto-closeable resource br may not be assigned  
}
```

- Until 1.6 Version try should be followed by either catch OR finally but from 1.7 onwards we can Take Only try with Resources without catch and finally Blocks.
Eg: try (R) {-----}
- The Main Advantage of try with Resources is finally Block will become Dummy because we are required to Close the Resources Explicitly.

Until 1.6 Version finally Block is Hero.
But 1.7 Version onwards Zero.



Multi Catch Block (Catch Block with Multiple Exceptions)

Until 1.6 Version even though Multiple Exceptions having Same Handling Code Compulsory we have to write a Separate catch Block for Every Exception.

```
try {
    -----
}
catch (ArithmaticException e) {
    e.printStackTrace();
}
catch (NullPointerException e) {
    e.printStackTrace();
}
catch (ClassCastException e) {
    System.out.println(e.getMessage());
}
catch (IOException e) {
    System.out.println(e.getMessage());
}
```

- The Problem in this Approach is it Increases Length of the Code and Reduces Readability.
- To Overcome this Problem SUN People Introduced Multi Catch Block in 1.7 Version.
- In this Approach we can write a Single Catch Block which can Handle Multiple Exceptions of different Types.

```
try {
    -----
}
catch (ArithmaticException | NullPointerException e) {
    e.printStackTrace();
}
catch (ClassCastException | IOException e) {
    System.out.println(e.getMessage());
}
```

```
class MultiCatchBlock {
    public static void main(String[] args) {
        try {
            //System.out.println(10/ 0);
            String s = null;
            System.out.println(s.length());
        }
        catch (ArithmaticException | NullPointerException e) {
            System.out.println(e); //java.lang.NullPointerException
        }
    }
}
```



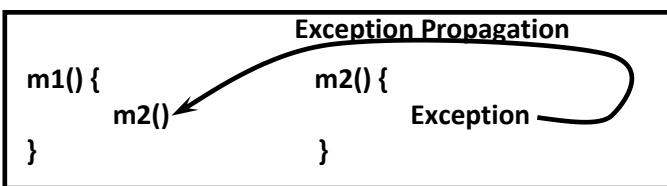
- If Multi Catch Block there should Not be any Relation between Exception Types (Like Parent to Child OR Child to Parent OR Same Type) Otherwise we will get Compile Time Error.

```
catch (ArithmaticException | NullPointerException | ClassCastException e) {}  
catch (ArithmaticException | Exception e) {}
```

CE: Alternatives in a multi-catch statement cannot be related by subclassing
Alternative ArithmaticException is a subclass of alternative Exception

Exception Propagation:

- Within a Method if an Exception raised and if we are Not Handle that Exception then that Exception Object will be propagated to Automatically to the Caller Method.
- Then Caller Method is Responsible to Handle that Exception.
- This Process is Called *Exception Propagation*.



Re-Throwing Exception

We can Use this Approach to Convert One Exception Type to Another Exception Type.

```
try {  
    System.out.println(10/ 0);  
}  
catch (ArithmaticException e) {  
    throw new NullPointerException();  
}
```



Q1. Given the code fragment:

```
1) class X
2) {
3)     public void printFileContent()
4)     {
5)         //Line-1
6)         throw new IOException(); //Line-2
7)     }
8) }
9) public class Test
10) {
11)     public static void main(String[] args) //Line-3
12)     {
13)         X x= new X();
14)         x.printFileContent(); //Line-4
15)         //Line-5
16)     }
17) }
```

Which two modifications required to compile code successfully?

- A. Replace Line-3 with `public static void main(String[] args) throws Exception`
B. Replace Line-4 with:

```
1) try
2) {
3)     x.printFileContent();
4) }
5) catch (Exception e){}
6) catch (IOException e){}
```

- C. Replace Line-3 with `public static void main(String[] args) throws IOException`

- D. Replace Line-2 with `throw IOException("Exception Raised");`

- E. At Line-5 insert `throw new IOException();`

Answer: A, C

Q2. Given the code Fragment:

```
1) public class Test
2) {
3)     void readCard(int cno) throws Exception
4)     {
5)         System.out.println("Rearding Card");
6)     }
7)     void checkCard(int cno) throws RuntimeException //Line-1
8) }
```



```
9)     System.out.println("Checking Card");
10) }
11) public static void main(String[] args)
12) {
13)     Test t = new Test();
14)     int cardNo=1234;
15)     t.checkCard(cardNo); //Line-2
16)     t.readCard(cardNo); //Line-3
17) }
18) }
```

What is the result?

- A. Checking Card
Reading Card
- B. Compilation Fails at Line-1
- C. Compilation Fails at Line-2
- D. Compilation Fails at Line-3
- E. Compilation Fails at Line-2 and Line-3

Answer: D

Q3. Given the following code for the classes MyException and Test:

```
1) public class MyException extends RuntimeException
2) {
3) }
4)
5) public class Test
6) {
7)     public static void main(String[] args)
8)     {
9)         try
10)         {
11)             m1();
12)         }
13)         catch (MyException e)
14)         {
15)             System.out.print("A");
16)         }
17)     }
18)     public static void m1()
19)     {
20)         try
21)         {
22)             throw Math.random() > 0.5 ? new Exception():new MyException();
23)         }
```



```
24)     catch (RuntimeException e)
25)     {
26)         System.out.println("B");
27)     }
28)
29) }
```

What is the result?

- A. A
- B. B
- C. Either A or B
- D. AB
- E. Compilation Fails

Answer: E

Q4. Given the code fragment:

```
1) String[] s= new String[2];
2) int i=0;
3) for(String s1: s)
4) {
5)     s[i].concat("element"+i);
6)     i++;
7) }
8) for(i=0; i<s.length;i++)
9) {
10)    System.out.println(s[i]);
11) }
```

What is the result?

- A. element 0
element 1
- B. null element 0
null element 1
- C. null
null
- D. A NullPointerException is thrown at runtime

Answer: D



Q5. Given the code fragment:

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String[] names={"Thomas","Bunny","Chinny"};
6)         String[] pwds=new String[3];
7)         int i =0;
8)         try
9)         {
10)             for (String n: names)
11)             {
12)                 pwds[i]=n.substring(2,6);
13)                 i++;
14)             }
15)         }
16)         catch (Exception e)
17)         {
18)             System.out.println("Invalid Name");
19)         }
20)         for(String p: pwds)
21)         {
22)             System.out.println(p);
23)         }
24)     }
25) }
```

What is the result?

A.
Invalid Name
omas
null
null

B.
Invalid Name

C.
Invalid Name
omas

D.
Compilation Fails

Answer: A



Q6. Given the code fragment:

```
1) import java.util.*;
2) public class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         ArrayList l = new ArrayList();
7)         String[] s;
8)         try
9)         {
10)             while(true)
11)             {
12)                 l.add("MyString");
13)             }
14)         }
15)         catch (RuntimeException e)
16)         {
17)             System.out.println("Caught a RuntimeException");
18)         }
19)         catch (Exception e)
20)         {
21)             System.out.println("Caught an Exception");
22)         }
23)         System.out.println("Ready to use");
24)     }
25) }
```

What is the result?

- A. Caught a RuntimeException printed to the console
- B. Caught an Exception printed to the console
- C. A runtime error is thrown at runtime
- D. Ready to use printed to the console
- E. The code fails to compile because a throws keyword required

Answer: C

Q7. Which three are advantages of the Java Exception Mechanism?

- A. Improves the program structure because the error handling code is separated from the normal program function.
- B. Provides a set of standard exceptions that covers all possible errors
- C. Improves the program structure because the programmer can choose where to handle exceptions



- D. Imporves the program structure because exceptions must be handled in the method in which they occurred.
- E. Allows the creation of new exceptions that are tailored to the particular program being created.

Answer: A, C, E

Q8. Which 3 statements are true about exception handling?

- A. Only unchecked exceptions can be rethrown
- B. All Subclasses of the RuntimeException are recoverable
- C. The parameter in catch block is of throwable type
- D. All subclasses of RuntimeException must be caught or declared to be thrown
- E. All Subclasses of the Exception except RuntimeException class are checked exceptions
- F. All subclasses of the Error class are checked exceptions and are recoverable

Answer: B, C, E

Q9. Which two statements are true?

- A. Error class is unextendable
- B. Error class is extendable
- C. Error is a RuntimeException
- D. Error is an Exception
- E. Error is a Throwable

Ans: B, E



FAQ's

- 1) What is An Exception?
- 2) What is The Purpose of Exception Handling?
- 3) What is The Meaning of Exception Handling?
- 4) Explain Default Exception Handling Mechanism in Java?
- 5) What is The Purpose of try?
- 6) What is The Purpose of catch Block?
- 7) Is try With Multiple catch Block Possible?
- 8) If try With Multiple catch Block Present, Is Order of catch Blocks Important in Which Order We Have To Take?
- 9) What Are Various Methods To Print Exception Information? And Differentiate Them.
- 10) If An Exception Raised Inside catch Block Then What Will Happen?
- 11) Is it Possible To Take try, catch Inside try Block?
- 12) Is it Possible To Take try, catch Inside catch Block?
- 13) Is it Possible To Take try Without catch?
- 14) What is The Purpose of Finally Block?
- 15) Is Finally Block Will Be Execute Always?
- 16) In Which Situation Finally Block Will Not Executed?
- 17) If Return Statement Present Inside try, is Finally Block Will Be Executed?
- 18) What is The Difference Between final, finally And finalize ()?
- 19) Is it Possible To Write Any Statement Between try-catch And finally?
- 20) Is it Possible To Take 2 finally Blocks For The Same try?
- 21) Is Syntax try-finally-catch is Valid?
- 22) What is The Purpose of throw?
- 23) Is it Possible To throw An Error?
- 24) Is it Possible To throw Any Java Object?



- 25) After throw is it Allow To Take Any Statement Directly?
- 26) What is The Purpose Of throws?
- 27) What is The Difference Between throw And throws?
- 28) What is The Difference Between throw And thrown?
- 29) Is it Possible To Use throws Keyword For Any Java Class?
- 30) If We Are Taking catch Block For An Exception But There is No Chance of Rising That Exception in try Then What Will Happen?
- 31) Explain Exception Handling Keywords?
- 32) Which Class Act As Root For Entire Java Exception Hierarchy?
- 33) What is The Difference Between Error And Exception?
- 34) What is Difference Between Checked Exception And Unchecked Exception?
- 35) What is Difference Between Partially Checked And Fully Checked Exception?
- 36) What is A Customized Exception?
- 37) Explain The Process of Creating The Customized Exception.
- 38) Explain Control Flow in try, catch, finally.
- 39) Can You Give The Most Common Occurred Exception in Your Previous Project?
- 40) Explain The Cases Where You Used Exception Handling in Your Previous Project?



String And StringBuilder

- 1) Introduction**
- 2) Object Class**
- 3) String Class**
- 4) StringBuffer Class**
- 5) StringBuilder Class**



Introduction

Object Class

Methods

- 1) **public String toString()**
- 2) **public int hashCode()**
 - **toString() Vs hashCode()**
- 3) **public boolean equals(Object obj)**
 - **Relationship between '==' Operator and .equals()**
 - **Difference between == Operator and .equals()**
 - **What is the Main Difference between == Operator and .equals()?**
 - **Contract between .equals() and hashCode()**
- 4) **protected Object clone() throws CloneNotSupportedException**
 - **Shallow Cloning Vs Deep Cloning**
- 5) **protected void finalize() throws Throwable**

- 6) **public final Class getClass()**

- 7) **public final void wait() throws InterruptedException**

- 8) **public final void wait(long ms) throws InterruptedException**

- 9) **public final void wait(long ms, int ns) throws InterruptedException**

- 10) **public final void notify()**

- 11) **public final void notifyAll()**

String Class

- **Interning of String Objects**
- **Importance of SCP**
- **Why SCP Concept is Available Only for String Object but Not for StringBuffer?**
- **Why String Objects are Immutable where as StringBuffer Objects are Mutable?**
- **String Class Constructors**
- **String Class Methods**
- **Creation of Our Own Immutable Class**
- **final Vs Immutability**

StringBuffer Class

- Constructors
Methods

StringBuilder Class

String Vs StringBuffer Vs StringBuilder

Method Chaining



Introduction

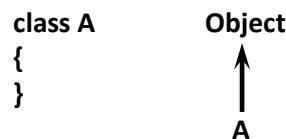
- For writing any Java Program whether it is Simple OR Complex, the Most Commonly required Classes and Interfaces are defined in Separate Package which is nothing but `java.lang` Package.
- We are Not required to Import `java.lang` Package Explicitly because by Default it is Available to Every Java Program.

Object Class

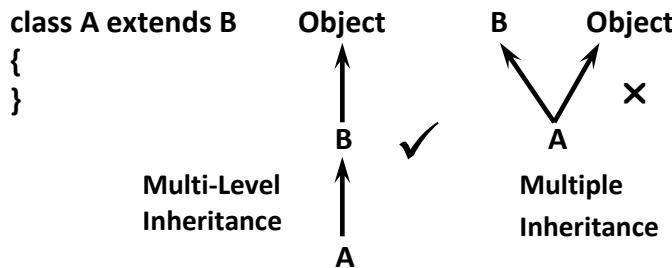
- Every Class in a Java is the Child Class of the Object Class Either Directly OR Indirectly.
- So that Object Class Methods by Default Available to Every Java Class.
- Hence Object Class Acts as Root for All Java Classes.

Note:

- If our Class won't extends any Other Class then Only it is the Direct Child Class of Object.



- If our Class extends any Other Class then it is In - Direct Child Class of Object.



Methods:

- 1) `public String toString()`
- 2) `public int hashCode()`
- 3) `public boolean equals(Object obj)`
- 4) `protected Object clone() throws CloneNotSupportedException`
- 5) `protected void finalize() throws Throwable`
- 6) `public final Class getClass()`



- 7) public final void wait() throws InterruptedException
- 8) public final void wait(long ms) throws InterruptedException
- 9) public final void wait(long ms, int ns) throws InterruptedException
- 10) public final void notify()
- 11) public final void notifyAll()

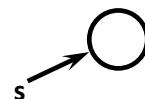
The `toString()`:

- We can Use `toString()` to get String Representation of an Object.
- Whenever we are trying to Print any Object Reference Internally `toString()` will be Called.

```
Student s = new Student();
System.out.println(s);
```



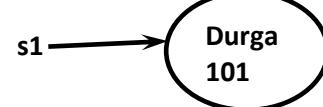
```
System.out.println(s.toString());
```



```
class Student {
    String name;
    int rollno;

    Student(String name, int rollno) {
        this.name = name;
        this.rollno = rollno;
    }

    public static void main(String arg[]) {
        Student s1 = new Student ("Durga", 101);
        Student s2 = new Student ("Ravi", 102);
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s1.toString());
    }
}
```



```
Student@1cb25f1
Student@2808b3
Student@1cb25f1
```

- In the Above Example Object Class `toString()` got executed which is implemented as follows

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

i.e. `ClassName@Hexa_Decimal_String_of_hashcode`



- To Provide Our Own String Representation we have to Override `toString()`.
- For Example whenever we are trying to Print Student Reference to Print his Name and RollNo we can Override `toString()` as follows.

```
public String toString() {  
    return name+"...."+rollno;  
    return name;  
    return "This is A Student with the Name: " +name+ " and rollno: "+rollno;  
}
```

- In All Wrapper Classes, in All Collection Classes, in `StringBuffer`, `StringBuilder` `toString()` is Overridden for Meaningful String Representation.
- It is Highly Recommended to Override `toString()` in our Classes Also.

```
import java.util.*;  
class Test {  
    public String toString() {  
        return "Test";  
    }  
    public static void main(String[] args) {  
  
        String s = new String("Durga");  
        Integer i = new Integer(10);  
        ArrayList l = new ArrayList();  
  
        l.add("A");  
        l.add("B");  
  
        HashMap m = new HashMap();  
        m.put("A", 101);  
        Test t = new Test();  
  
        System.out.println(s); //Durga  
        System.out.println(i); //10  
        System.out.println(l); // [A, B]  
        System.out.println(m); // {A=101}  
        System.out.println(t); //Test  
    }  
}
```



The hashCode():

- For Every Object JVM will generate a Unique Number which is nothing but Hash Code.
- JVM will use Hash Code while Saving Objects into Hashing related Data Structures Like HashSet, Hashtable, HashMap Etc.
- The Main Advantage of saving Objects based on Hash Code is Search Operation will become Easy.
- The Most Powerful Search Algorithm up Today is Hashing.
- If we are Not Overriding hashCode() then Object Class hashCode() will be executed. Which generates hashCode based on Address of an Object. It doesn't Mean Hash Code Represents Address of the Object (It is Impossible to find Address of an Object in Java).
- Based on our Requirement we can Override hashCode() then it is No Longer related to Address.
- Overriding hashCode() is Said to be Proper Way if and Only if for Every Object we have to Generate a Unique Number as hashCode.
- It is Improper Way of Overriding hashCode() because for all Objects we are generating Same Number as Hash Code.
- It is Proper Way of Overriding hashCode() because we are generating a Unique Number as Hash Code.

Improper Way

```
class Student {  
    public int hashCode() {  
        return 100;  
    }  
}
```

Proper Way

```
class Student {  
    public int hashCode() {  
        return rollno;  
    }  
}
```

toString() Vs hashCode()

If we are giving the Chance to Object Class `toString()` then it will Call Internally `hashCode()`.

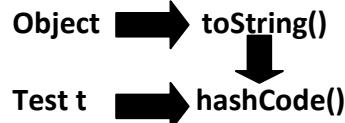
But whenever we are Overriding `toString()` Call `hashCode()`.

```
class Test {  
    int i;  
    Test(int i) {  
        this.i = i;  
    }  
    public static void main(String[] args) {  
        Test t1 = new Test(10);  
        Test t2 = new Test(100);  
        System.out.println(t1); //Test@6e3d60  
        System.out.println(t2); //Test@17fa65e  
    }  
}
```





```
class Test {  
    int i;  
    Test(int i) { this.i = i; }  
    public int hashCode() { return i; }  
    public static void main(String[] args) {  
        Test t1 = new Test(10);  
        Test t2 = new Test(100);  
        System.out.println(t1); //Test@a  
        System.out.println(t2); //Test@64  
    }  
}
```



```
class Test {  
    int i;  
  
    Test(int i) { this.i = i; }  
    public int hashCode() { return i; }  
    public String toString() { return i + ""; }  
  
    public static void main(String[] args) {  
        Test t1 = new Test(10);  
        Test t2 = new Test(100);  
        System.out.println(t1); //10  
        System.out.println(t2); //100  
    }  
}
```



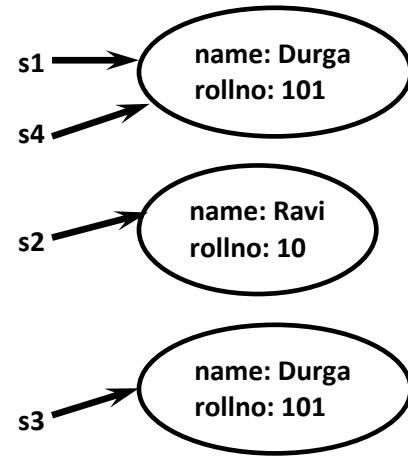
Note: The Above 3 Programs Shows Link between **toString()** and **hashCode()**.

The equals():

- We can Use **equals()** to Check Equality of 2 Objects.
- If our Class doesn't contain **equals()** then Object Class **equals()** will be executed. Which is meant for Reference Comparison (Address Comparison) i.e. `r1.equals(r2);` Returns true if and only if both `r1` and `r2` pointing to the Same Object.



```
class Student {  
    String name;  
    int rollno;  
  
    Student(String name, int rollno) {  
        this.name = name;  
        this.rollno = rollno;  
    }  
  
    public static void main(String[] args) {  
        Student s1 = new Student("Durga", 101);  
        Student s2 = new Student("Ravi", 102);  
        Student s3 = new Student("Durga", 101);  
        Student s4 = s1;  
        System.out.println(s1.equals(s2)); //false  
        System.out.println(s1.equals(s3)); //false  
        System.out.println(s1.equals(s4)); //true  
    }  
}
```



- In the Above Example Object Class equals() got executed which is meant for Reference Comparison.
- Based on our Requirement we can Override equals() in our Class for Content Comparison.

While Overriding equals() we have to Consider the following things.

- What is the Meaning of Content Comparison. For Example Whether we have to Check only Names OR only rollnos OR Both.
- If we Pass different Type of Objects our equals() should Return false but not java.lang.ClassCastException i.e Whether to Handle ClassCastException to Return false.
- If we Pass null Argument our equals() should Return false but not java.lang.NullPointerException i.e. we have to Handle NullPointerException to Return false.

The following is the Valid Way of Overriding equals() for Content Comparison in Student Class.

- If 2 Students having the Same Name and Same rollno our equals() should Return true.



```
class Student {  
    String name;  
    int rollno;  
  
    Student(String name, int rollno) {  
        this.name = name;  
        this.rollno = rollno;  
    }  
  
    public boolean equals(Object obj) {  
        try {  
            String name1 = this.name;  
            int rollno1 = this.rollno;  
            Student s2 = (Student)obj;  
            String name2 = s2.name;  
            int rollno2 = s2.rollno;  
            if(name1.equals(name2) && rollno1 == rollno2) { return true; }  
            else { return false; }  
        }  
        catch (ClassCastException c) { return false; }  
        catch (NullPointerException e) { return false; }  
    }  
    public static void main(String arg[]) {  
  
        Student s1 = new Student ("Durga", 101);  
        Student s2 = new Student ("Ravi", 102);  
        Student s3 = new Student ("Durga", 101);  
        Student s4 = s1;  
  
        System.out.println(s1.equals(s2)); //false  
        System.out.println(s1.equals(s3)); //true  
        System.out.println(s1.equals(s4)); //true  
        System.out.println(s1.equals("Durga")); //false  
        System.out.println(s1.equals(null)); //false  
    }  
}
```



Simplified Version of equals()

```
public boolean equals(Object obj) {  
    try {  
        Student s = (Student)obj;  
        if(name.equals(s.name) && rollno == s.rollno) {  
            return true;  
        }  
        else {  
            return false;  
        }  
    }  
    catch (ClassCastException c) { return false; }  
    catch (NullPointerException e) { return false; }  
}
```

More Simplified Version of equals()

```
public boolean equals(Object obj) {  
    if (obj instanceof Student) {  
        Student s = (Student)obj;  
        if(name.equals(name) && rollno == s.rollno) {  
            return true;  
        }  
        else { return false; }  
    }  
    return false;  
}
```

Note: To Make Our equals() More Efficient we have to Take the following Code at the beginning of equals().

```
if (obj == this)  
    return true;
```

i.e if 2 References pointing to the Same Object then our equals() should Return true without performing any Comaprision.

Eg: String s1 = new String("Durga");
String s2 = new String("Durga");

```
StringBuffer sb1 = new StringBuffer("Durga");  
StringBuffer sb2 = new StringBuffer("Durga");  
  
System.out.println(s1.equals(s2)); //true  
System.out.println(sb1.equals(sb2)); //false  
System.out.println(s1.equals(sb1)); //false
```



- If String Class equals() is Overridden for Content Comparison but in StringBuffer Class equals() is not Overridden for Content Comparison Hence Object Class equals() will be executed which is meant for Reference Comparison.

Note: In All Wrapper Classes, in All Collection Classes, in String Class equals() is Overridden for Content Comparison. Hence it is Highly Recommended to Override equals() in our Class Also.

Relationship between '==' Operator and .equals()

- If $r1 == r2$ Return true then $r1.equals(r2)$ is always true i.e if 2 Objects are Equal by == Operator then these Objects are always Equal by .equals() also.
- If $r1 == r2$ Returns false then $r1.equals(r2)$ then we can't conclude anything about equals(). It may Returns true OR false.
- If $r1.equals(r2)$ is true then we can't conclude anything about == Operator. It may Returns true OR false.
- If $r1.equals(r2)$ is false then $r1 == r2$ is always false.

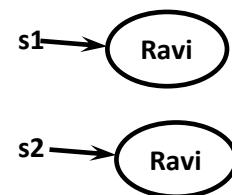
Difference between == Operator and .equals()

== Operator	.equals()
It is an Operator applicable for Primitives and Object References.	It is a Method applicable only for Object References.
In the Case of Object References == Operator always meant for Reference Comparison.	By Default equals() Present in Object Class also meant for Reference Comparison.
We can't Override == Operator for Content Comparison.	We can Override equals() for Content Comparison.
To use == Operator Compulsory there should be Some Relation between Argument Types Otherwise we will get CE: incompatible types.	If there is No Relation between Argument Type we won't get any CE/ RE. equals() Simply Returns false.
For Any Object References r, $r == null$ Returns false.	For Any Object References r, $r.equals(null)$ Returns false.

What is the Main Difference between == Operator and .equals()?

In General we can use .equals() for Content Comparison whereas == Operator for Reference Comparison.

```
String s1 = new String("Ravi");
String s2 = new String("Ravi");
System.out.println(s1.equals(s2)); //true
System.out.println(s1 == s2); //false
```





Contract between .equals() and hashCode()

2 Equivalent Objects should be placed in the Same Bucket but all Objects Present in the Same Bucket Need not be Equal.

- If 2 Objects are Equal by .equals() then Compulsory their hashCode must be Same i.e. 2 Equivalent Objects should have Same hashCode. i.e. if `r1.equals(r2)` is true then `r1.hashCode() == r2.hashCode()` should be true.
- If 2 Objects are not Equal by .equals() then No Restriction on hashCode may be Same OR may not be Same.
- If Hash Codes of 2 Objects are Equal we can't conclude anything about .equals() it may Returns true OR false.
- If Hash Codes of 2 Objects are not Equal then these Objects are always not Equal by .equals() Also.

To Specify Above Contract between equals() and hashCode() whenever we are Overriding equals() Compulsory we should Override hashCode() otherwise we won't get CE/ RE but it is Not a Good Programming Practice.

Eg: In String Class equals() is Overridden for Content Comparison hence hashCode() is also Overridden to generate Hash Code based on Content.

```
String s1 = new String("Durga");
String s2 = new String("Durga");
System.out.println(s1.equals(s2)); //true
System.out.println(s1.hashCode()); //2539842
System.out.println(s2.hashCode()); //2539842
```

Consider the following Person Class:

```
public boolean equals(Object obj)
{
    if(obj instanceof Person)
    {
        Person p = (Person)obj;
        if(name.equals(p.name) && age == p.age) { return true; }
        else {
            return false;
        }
    }
    return false;
}
```



Which of the following hashCode() is Appropriate for Person Class?

- public int hashCode() { return 100; }
- public int hashCode() { return name.length() + height; }
- public int hashCode() { return name.hashCode() + age; } ✓
- No Restriction

Note: Based on which Parameters we Override equals() use the Same Parameters by Overriding hashCode() also so that we can maintain Contract between equals() and hashCode().

The clone():

- The Process of Creating exactly Duplicate Object is known as Cloning.
- The Main Purpose of Cloning is to maintain Backup and to Preserve Initial State of an Object.
- We can cloned an Object by using clone() of Object Class.
protected native Object clone() throws CloneNotSupportedException
- To Perform Cloning Compulsory the Object should be Cloneable Object.
- An Object is said to be Cloneable if and only if the Corresponding Class implements Cloneable Interface. It is a *Marker Interface* because it doesn't contain any Methods.
- If we are trying to Clone a Non-Cloneable Object we will get a RE:
CloneNotSupportedException

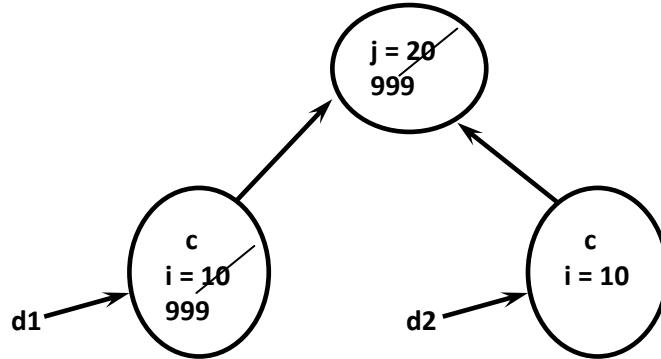
Shallow Cloning Vs Deep Cloning

Shallow Cloning

- The Process of creating Bit Wise Copy of an Object is called Shallow Cloning.
- If the Main Object contain any Primitive Variables exactly Duplicate Copy of those Variables will be created in cloned Object.
- If the Main Object contain any Referenced Variable then the Corresponding Object won't be Created Just Duplicate Reference Variable will be Created by pointing to Old contained Object.
- By using Main Object Reference if we Perform any Change to the contained Object then those Changes will be reflected to the Cloned Object.
- Object Class clone() by Default meant for Shallow Cloning.



```
class Cat {  
    int j;  
    Cat(int j) { this.j = j; }  
}  
class Dog implements Cloneable {  
    Cat c;  
    int i;  
    Dog(Cat c, int i) {  
        this.c = c;  
        this.i = i;  
    }  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}  
class ShallowCloning {  
    public static void main(String[] args) throws CloneNotSupportedException {  
        Cat c = new Cat(20);  
        Dog d1 = new Dog(c, 10);  
        System.out.println(d1.i+"...."+d1.c.j); //10....20  
        Dog d2 = (Dog)d1.clone();  
        d1.i = 888;  
        d1.c.j = 999;  
        System.out.println(d2.i+"...."+d2.c.j); //10....999  
    }  
}
```



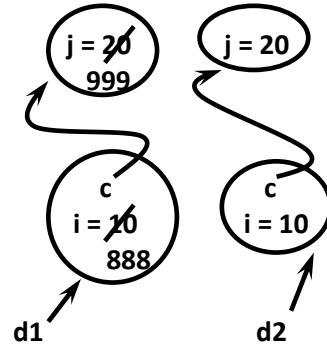
- Shallow Cloning is the Best Choice if the Object contains Only Primitive Values.
- In Shallow Cloning by using Main Object Reference if we perform any Changes to the contained Object then those Changes will be reflected to the cloned Copy.
- To overcome this Problem we should go for Deep Cloning.

Deep Cloning:

- The Process of creating exactly Independent Duplicate Object (including contained Objects also) is called Deep Cloning.
- In Deep Cloning if Main Object contain any Reference Variable then corresponding Object also will be created in the cloned Copy.
- By Default Object Class `clone()` meant for Shallow Cloning if we want Deep Cloning then Programmer is Responsible to implement that by Overriding `clone()`.



```
class Cat {  
    int j;  
    Cat(int j) { this.j = j; }  
}  
class Dog implements Cloneable {  
    Cat c;  
    int i;  
    Dog(Cat c, int i) {  
        this.c = c;  
        this.i = i;  
    }  
    public Object clone() throws CloneNotSupportedException {  
        Cat c1 = new Cat(c.j);  
        Dog d = new Dog(c1, i);  
        return d;  
    }  
}  
class DeepCloning {  
    public static void main(String[] args) throws CloneNotSupportedException {  
        Cat c = new Cat(20);  
        Dog d1 = new Dog(c, 10);  
        System.out.println(d1.i+"...."+d1.c.j); //10....20  
        Dog d2 = (Dog)d1.clone();  
        d1.i = 888;  
        d1.c.j = 999;  
        System.out.println(d2.i+"...."+d2.c.j); //10....20  
    }  
}
```



- In Deep Cloning by using Main Object Reference if we perform any Change to the contained Object then those Changes won't be reflected to the cloned Objects.

Which Cloning is the Best?

- If the Object contains Only Primitive Variables then Shallow Cloning is the Best Choice.
- If the Object contains Referenced Variables then Deep Cloning is the Best Choice.



getClass():

We can Use `getClass()` to get Runtime Class Definition of an Object.

Eg: `Object o = l.get(o);
System.out.println(o.getClass().getName());`

Class Object



We can Use this Method Very Commonly in Reflections.

finalize():

- Just before destroying an Object Garbage Collector calls this Method for Clean-Up Activities.
- Once `finalize()` completes Automatically Garbage Collector destroys that Object.

`wait()`, `notify()` and `notifyAll()`: We can use all these Methods in Multi-Threading for Inter Thread Communication.



String

Difference between String and StringBuffer Objects?

```
String s = new String("Durga");
s.concat("Software");
System.out.println(s); //Durga
```

Once we created a String Object we can't Perform any changes in the existing Object. If we are trying to Perform any changes with those changes a New Object will be created. This non-changeable Nature is nothing but Immutability.



Case 1

```
StringBuffer sb = new StringBuffer("Durga");
sb.append("Software");
System.out.println(sb); //DurgaSoftware
```

Once we created a StringBuffer Object we can Perform any Type of changes in the existing Object. This Changeable Nature is nothing but Mutability of StringBuffer Objects.



```
String s1 = new String("Durga");
String s2 = new String("Durga");
System.out.println(s1 == s2); //false
System.out.println(s1.equals(s2)); //true
```

In String Class .equals() is Overridden for Content Comparison Hence if the Content is Same .equals() Returns True Even though Objects are Different.

Case 2

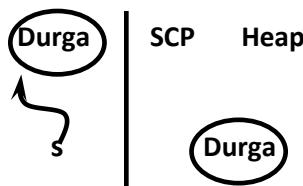
```
StringBuffer sb1 = new StringBuffer("Durga");
StringBuffer sb2 = new StringBuffer ("Durga");
System.out.println(sb1 == sb2); //false
System.out.println(sb1.equals(sb2)); //false
```

In StringBuffer Class .equals() is not Overridden for Content Comparison Hence Object Class .equals() will be executed which is meant for Reference Comparison. Due to this if Objects are different equals() Returns false Event though Content is Same.

Case 3: Difference between String s = new String("Durga"); and String s = "Durga";

❖ String s = new String("Durga");

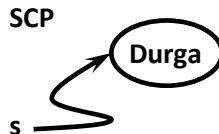
In this Case 2 Objects will be Created, One is in the Heap and the Other is in String Constant Pool (SCP) and S is always pointing to Heap Object.





❖ **String s = "Ravi";**

In this Case only one Object will be created in the SCP and S is always refers to that Object.

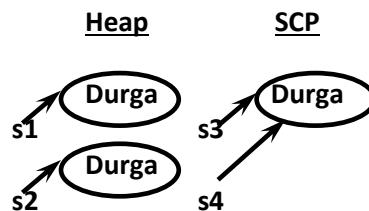


Note:

- Garbage Collector is Not allowed to Access SCP Area Hence Even though Object Not having the Reference Still it is Not Eligible for Garbage Collector if it is Present in SCP Area.
- At the Time of JVM Shutdown all SCP Objects will be destroyed automatically.
- Object Creation in SCP is always Optional. First JVM will check is any Object already Present in SCP with required Content if it is already Present then existing Object will be re-used. If it is not already Present then only a New Object will be created.

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

```
String s3 = "Durga";
String s4 = "Durga";
```



Note:

- Whenever we are using New Operator Compulsory a New Object will be created in the Heap Area.
- There may be a Chance of existing 2 Objects with Same Content on the Heap but No Chance of existing 2 Objects with the Same Content in SCP Area i.e. Duplicate Objects are Possible in Heap but there is No Chance of existing Duplicate Objects in SCP.

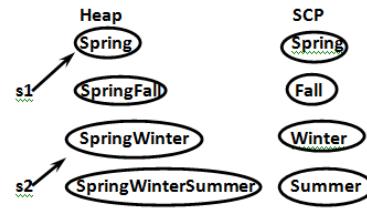
```
String s = new String("Durga");
s.concat("Software");
s = s.concat("Solutions");
```



Note:

- For every String Constant one Object will be created on the SCP.
- Because of Some Runtime Operation like Method Call if an Object is required to create that Object will be created only in Heap but not in SCP.

```
String s1 = new String("Spring");
s1.concat("Fall");
String s2 = s1.concat("Winter");
s2.concat("Summer");
System.out.println(s1); //Spring
System.out.println(s2); //SpringWinter
```



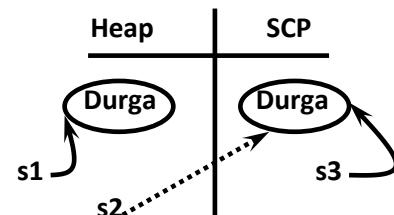


```
class StringTest {  
    public static void main(String[] args) {  
  
        String s1 = new String("You cannot Change Me");  
        String s2 = new String("You cannot Change Me");  
        System.out.println(s1 == s2); //false  
  
        String s3 = "You cannot Change Me";  
        System.out.println(s1 == s3); //false  
  
        String s4 = "You cannot Change Me";  
        System.out.println(s3 == s4); //true  
  
        String s5 = "You cannot" + " Change Me";  
        System.out.println(s4 == s5); //true  
  
        String s6 = "You cannot";  
        String s7 = s6 + " Change Me";  
        System.out.println(s4 == s7); //false  
  
        final String s8 = "You cannot";  
        String s9 = s8 + " Change Me";  
        System.out.println(s4 == s9); //true  
  
    }  
}
```

Interning of String Objects:

By Using Heap Object Reference if we want to get corresponding SCP Object Reference then we should go for intern().

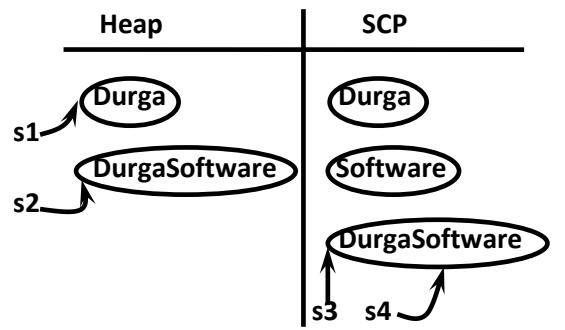
```
class StringTest {  
    public static void main(String[] args) {  
        String s1 = new String("Durga");  
        String s2 = s1.intern();  
        System.out.println(s1 == s2); //false  
        String s3 = "Durga";  
        System.out.println(s2 == s3); //true  
    }  
}
```



If the corresponding SCP Object is Not Available then intern() will Create that Object in SCP and Returns it.



```
String s1 = new String("Durga");
String s2 = s1.concat("Software");
String s3 = s2.intern();
String s4 = "DurgaSoftware";
System.out.println(s3 == s4); //true
```



Importance of SCP:

- In Our Program if any String Objects required repeatedly then it is not recommended to create a Separate Object for every Requirement because it reduces Memory Utilization and Creates Performance Problems.
- Instead of creating a Separate Object for every Requirement we can Create a Single Object and we can Reuse the Same Object for every Similar Requirement so that Memory Utilization and Performance will be improved. We can achieve this by using SCP Concept. Hence the Main Advantage of SCP is Memory Utilization and Performance will be improved.
- The Main Disadvantage of SCP is in SCP Several References pointing to the Same Object. By using One Reference if we are allowed to Change the Content then the remaining References will be impacted. To overcome this Problem SUN People implemented String Objects as Immutable. According to this once we created a String Object we can't perform any Changes in the existing Objects. If we are trying to Change with those Changes a New Object will be created.
- SCP is a specially designed Memory Area for the String Constants.

Why SCP Concept is Available Only for String Object but Not for StringBuffer?

- String Objects are Most Commonly used Objects in Java Hence SUN People provided a Special Memory Management for the String Objects.
- StringBuffer Objects are not commonly used Objects Hence SUN People won't provide any Special Memory Area for the StringBuffer Objects.

Why String Objects are Immutable where as StringBuffer Objects are Mutable?

- In the Case of String Objects Just Because of SCP A Single Object can be referred by Multiple References.
- By using One Reference if we are allowed to Change the Content then remaining References will be impacted. To overcome this Problem SUN People implemented String Objects as Immutable.
- According to this once we created a String Object we can't perform any Changes in the existing Object. If we are trying to Change with those Changes a New Object will be created.



- But in StringBuffer there is no Concept like SCP. Hence for every Requirement a New Object will be created by using One Reference. If we are changing the Content in existing Object there is No Effect on remaining References. Hence Immutability Concept not required for StringBuffer Objects.

Note: SCP is the Only Reason why String Objects are Immutable.

In Addition to String Objects any Other Objects are Immutable in Java?
All Wrapper Class Objects also Immutable.

String Class Constructors:

- 1) **String s = new String();** Creates an Empty String Object.
- 2) **String s = new String(String literal);** Creates an Equivalent String Object for the given String Literal.
- 3) **String s = new String(StringBuffer sb);** Creates an Equivalent String Object for the given StringBuffer.
- 4) **String s = new String(char [] ch);** Creates an Equivalent String Object for the given char[].
Eg:
char[] ch = {'a', 'b', 'c', 'd'};
String s = new String(ch);
System.out.println(s); //abcd
- 5) **String s = new String(byte[] b);** Creates an Equivalent String Object for the given byte[].
Eg:
byte[] b = {100, 101, 102, 103, 104};
String s = new String(b);
System.out.println(s); //defgh

Important Methods of String Class

- 1) **public char charAt(int index);** Returns the char locating at specified Index.
Eg:
String s = "Durga";
System.out.println(s.charAt(3)); //g
System.out.println(s.charAt(30)); //java.lang.StringIndexOutOfBoundsException:
String index out of range: 30
- 2) **public String concat(String s);**
The Overloaded '+' and '+=' Operators Acts as Concatenation Operation Only.

```
String s = "Durga";  
s = s.concat("Software");  
//s = s+"Software"; OR s += "Software";  
System.out.println(s); //DurgaSoftware
```



3) public boolean equals(Object o)

To Perform Content Comparison where Case is Important.
This is Overriding Version of Object Class equals().

4) public boolean equalsIgnoreCase(String s);

To Perform Content Comparison where Case is Not Important.

Eg:

```
String s = "Java";
System.out.println(s.equals("JAVA")); //false
System.out.println(s.equalsIgnoreCase("JAVA")); //true
```

Note: In General we can use *equalsIgnoreCase()* to Compare User Names where Case is Not Important, whereas we can use *equals()* to Compare Passwords where Case is Important.

5) public int length(); Return Number of Characters Present in the String.

Eg:

```
String s = "Java";
System.out.println(s.length()); //4
System.out.println(s.length); //CE: cannot find symbol
                           symbol: variable length
                           location: variable s of type String
```

Note: length is the Variable applicable for Array Objects where as length() is the Method applicable for String Objects.

6) public String replace(char old, char new);

Eg:

```
String s = "ababa";
System.out.println(s.replace('a','b')); //bbbbb
```

7) public String substring(int begin);

Returns Sub String from Begin Index to End of the String.

8) public String substring(int begin, int end);

Returns the Characters from Begin Index to End-1 Index.

Eg:

```
String s = "abcdefg";
System.out.println(s.substring(3)); //defg
System.out.println(s.substring(2, 5)); //cde
```

9) public String toLowerCase();

10) public String toUpperCase();

11) public String trim(); To Remove Blank Spaces Present at Starting and Ending of the String. But not Middle Blank Spaces.

12) public int indexOf(char ch);

Returns the Index of First Occurance of specified Character.



13) public int lastIndexOf(char ch);

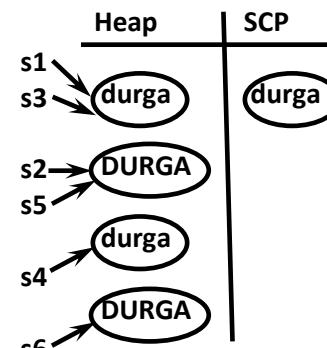
Returns the Index of Last Occurrence of specified Character.

Eg:
String s = "ababa";
System.out.println(s.indexOf('a')); //0
System.out.println(s.lastIndexOf('a')); //4

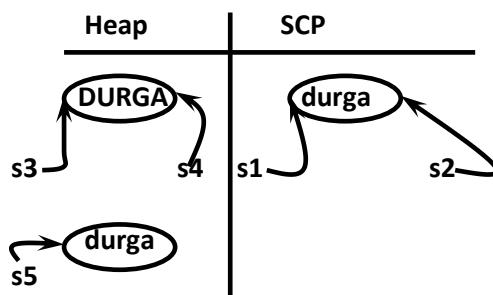
Note:

- Once We Created A String Object We Can't Perform Any Changes In The Existing Object.
- If We Are Trying To Perform Any Operation And If There Is A Change In The Content With Those Changes A New Object Will Be Created.
- With Our Operation If There Is No Change In The Content Then New Object Won't Be Created Existing Object Will Be Reused. This Rule Is Same Whether The Object Present In Heap Or SCP.

```
String s1 = new String("durga");  
  
String s2 = s1.toUpperCase();  
String s3 = s1.toLowerCase();  
  
System.out.println(s1 == s2); //false  
System.out.println(s1 == s3); //false  
  
String s4 = s2.toLowerCase();  
String s5 = s2.toUpperCase();  
String s6 = s4.toUpperCase();
```



```
String s1 = "durga";  
  
String s2 = s1.toLowerCase();  
String s3 = s1.toUpperCase();  
  
String s4 = s3.toString();  
String s5 = s3.toLowerCase();
```



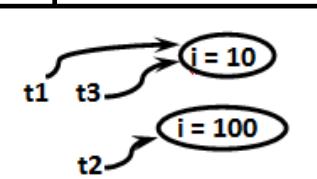
Creation of Our Own Immutable Class:

- Once we created an Object we can't perform any Changes in the existing Object.
- If we are trying to perform any Changes with those Changes a New Object will be created.
- If there is No Change in the Content then New Object won't be created and existing Object will be reused.



```
final class Test {  
    private int i;  
    Test(int i) {  
        this.i = i;  
    }  
    public Test modify(int i) {  
        if(this.i == i)  
            return this;  
        else  
            return (new Test(i));  
    }  
}
```

```
Test t1 = new Test(10);  
Test t2 = t1.modify(100);  
Test t3 = t1.modify(10);  
System.out.println(t1 == t2); //false  
System.out.println(t1 == t3); //true
```



Once we created a Test Object we can't perform any Changes in the existing Object. If we are trying to perform any Changes with those Changes a New Object will be created. Hence Test Class Objects are Immutable.

final Vs Immutability:

- final Keyword applicable for Variables but not for Objects whereas Immutability Concept is applicable for Objects but not for Variables.
- final and Immutability both are different Concepts.
- By declaring a Reference Variable as final we won't get any Immutability Nature but we can't perform re-assignment for that Reference Variable.

```
class Test {  
    public static void main(String[] args) {  
        final StringBuffer sb = new StringBuffer("Durga");  
        sb.append("Software");  
        System.out.println(sb); //true  
        sb = new StringBuffer("Solutions"); //CE: cannot assign a value to final variable sb  
    }  
}
```



Note:

- If the Content is Not Fixed and keep on changing then it is Never recommend to go for String because for every Change a New Object will be created Internally.
- To Handle this Requirement we should go for StringBuffer Class.
- The Main Advantage of StringBuffer over String is all required Changes will be performed in the existing Object only Instead of creating a New Object.



Which of the following is Meaningful?

- 1) final variable ✓
- 2) final object ✗
- 3) Immutable variable ✗
- 4) Immutable object ✓



StringBuffer

Constructors:

- 1) **StringBuffer sb = new StringBuffer();**

Creates an Empty StringBuffer Object with Default Initial Capacity 16.

```
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity()); //16
```

Eg:

```
sb.append("abcdefghijklmnp");
System.out.println(sb.capacity()); //16
```

```
sb.append("q");
System.out.println(sb.capacity()); //34
```

Once StringBuffer reaches Max Capacity then a New StringBuffer Object will be created with *new capacity = (currentcapacity + 1) * 2*

- 2) **StringBuffer sb = new StringBuffer(int initialCapacity);**

Creates an Empty StringBuffer Object with the specified Initial Capacity.

- 3) **StringBuffer sb = new StringBuffer(String s);**

Creates an Equivalent StringBuffer Object for the given String Object with *capacity = s.length() + 16*

Eg:

```
StringBuffer sb = new StringBuffer("Durga");
System.out.println(sb.capacity()); // 5 + 16 = 21
```

Methods

- 1) **public int length();**

- 2) **public int capacity();**

- 3) **public char charAt(int index);**

Eg:

```
StringBuffer sb = new StringBuffer("Durga");
System.out.println(sb.charAt(3)); //g
System.out.println(sb.charAt(30)); //RE: java.lang.StringIndexOutOfBoundsException:
String index out of range: 30
```

- 4) **public void setCharAt(int index, char ch);**

Replaces the Character Present at specified Index with the provided Character.



5) `public StringBuffer append(String s);
public StringBuffer append(int i);
public StringBuffer append(float f);
public StringBuffer append(double d);
public StringBuffer append(boolean b);
public StringBuffer append(Object o);
.....
.....`

Overloaded Methods

Eg:

```
StringBuffer sb = new StringBuffer();  
sb.append("PI Value is: ");  
sb.append(3.14);  
sb.append(" It is Exactly: ");  
sb.append(true);  
System.out.println(sb); //PI Value is: 3.14 It is Exactly: true
```

6) `public StringBuffer insert(int index, String s);
public StringBuffer insert(int index, int i);
public StringBuffer insert(int index, float f);
public StringBuffer insert(int index, double d);
public StringBuffer insert(int index, boolean b);
public StringBuffer insert(int index, Object o);
.....
.....`

Overloaded Methods

Eg:

```
StringBuffer sb = new StringBuffer("abcdefg");  
sb.insert(2, "xyz");  
System.out.println(sb); //abxyzabcdefg
```

7) **public StringBuffer delete(int begin, int end);**

To Delete Characters locating from Begin Index to End-1 Index.

8) **public StringBuffer deleteCharAt(int index);**

9) **public StringBuffer reverse();**

10) **public void setLength(int length);**

```
StringBuffer sb = new StringBuffer("AishwaryaAbhi");  
sb.setLength(8);  
System.out.println(sb); //Aishwarya
```

11) **public void ensureCapacity(int capacity);**

To Increase Capacity on Fly based on our Requirement.

```
StringBuffer sb = new StringBuffer();  
System.out.println(sb.capacity()); //16  
sb.ensureCapacity(1000);  
System.out.println(sb.capacity()); //1000
```



12)public void trimToSize();

To De-allocate Extra allocated Free Memory.

```
StringBuffer sb = new StringBuffer(1000);
sb.append("abc");
sb.trimToSize();
System.out.println(sb.capacity()); //3
```

Note:

- Every Method Present in StringBuffer is synchronized.
- Hence at a Time Only One Thread is allowed to operate on StringBuffer Object.
- It increases waiting Time of Threads and Creates Performance Problems.
- To Overcome this Problem SUN People introduced StringBuilder Class in 1.5 Version.



StringBuilder

StringBuilder is exactly Same as StringBuffer (Including Constructors and Methods Also) Except the following Differences.

StringBuffer	StringBuilder
Every Method Present In StringBuffer Is Synchronized.	No Method Present In StringBuilder Is Synchronized.
At A Time Only One Thread Is Allow To Operate On StringBuffer Object And Hence It Is Thread Safe.	At A Time Multiple Thread Are Allowed To Operate On StringBuilder Object And Hence It Is Not Thread Safe.
Threads Are Required To Wait To Operate On StringBuffer Object And Hence Relatively Performance Is Slow.	Threads Are Not Required To Wait To Operate On StringBuilder Object And Hence Relatively Performance Is High.
Introduced In 1.0 Version.	Introduced In 1.5 Version.

String Vs StringBuffer Vs StringBuilder

- If the Content is Fixed and won't changed frequently then we should go for String.
- If the Content is Not Fixed and keep on changing but Thread Safety is required then we should go for StringBuffer.
- If the Content is Not Fixed and keep on changing but Thread Safety is Not required then we should go for StringBuilder.

Method Chaining:

- For most of the Methods in String, StringBuffer and StringBuilder the Return Types are the Same Type (String, StringBuffer and StringBuilder Objects) only.
- Hence After Applying a Method on the Result we can Call Another Method which forms a Method Chaining. `sb.m1().m2().m3().m4().m5();`.....;
- All these Method Calls will execute from Left to Right.

```
StringBuffer sb = new StringBuffer();

sb.append("Durga")
.append("Software") // DurgaSoftware
.append("Solutions") // DurgaSoftwareSolutions
.insert(2, "xyz") // DuxyzrgaSoftwareSolutions
.delete(7, 15) // DuxyzrgeSolutions
.reverse() // snoitulosegrzyxuD
.append("Hyd"); //snoitulosegrzyxuDHyd
System.out.println(sb); //snoitulosegrzyxuDHyd
```



Practice Questions for String and StringBuilder

Q1. Given the code fragment:

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         StringBuilder sb=new StringBuilder(5);
6)         String s="";
7)         if(sb.equals(s))
8)         {
9)             System.out.println("Match 1");
10)        }
11)        else if(sb.toString().equals(s.toString()))
12)        {
13)            System.out.println("Match 2");
14)        }
15)        else
16)        {
17)            System.out.println("No Match");
18)        }
19)    }
20) }
```

What is the result?

- A. Match 1
- B. Match 2
- C. No Match
- D. NullPointerException is thrown at runtime

Answer: B

Q2. Given:

```
1) public class Test
2) {
3)     public static void main(String[] args) {
4)         String ta="A";
5)         ta=ta.concat("B");
6)         String tb="C";
7)         ta=ta.concat(tb);
8)         ta.replace('C','D');
9)         ta=ta.concat(tb);
10)        System.out.println(ta);
11)    }
12) }
```



What is the result?

- A. ABCD
- B. ACD
- C. ABCC
- D. ABD
- E. ABDC

Answer: C

Q3. Given the code fragment:

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         StringBuilder sb1= new StringBuilder("Durga");
6)         String str1=sb1.toString();
7)         //insert code here==>Line-1
8)         System.out.println(str1==str2);
9)     }
10) }
```

Which code fragment, when inserted at Line-1, enables the code to print true?

- A. String str2=str1;
- B. String str2=new String(str1);
- C. String str2=sb1.toString();
- D. String str2="Durga";

Answer: A

Q4. You are developing a banking module. You have developed a class named ccMask that has a maskCC method.

Given the code fragment:

```
1) class CCmask
2) {
3)     public static String maskCC(String creditCard)
4)     {
5)         String x="XXXX-XXXX-XXXX-";
6)         //Line-1
7)     }
8)     public static void main(String[] args)
9)     {
10)        System.out.println(maskCC("1234-5678-9101-1121"));
11)    }
12) }
```



You must ensure that maskCC method returns a String that hides all digits of the credit card number except last four digits(and the hyphens that separate each group of 4 digits)

Which two code fragments should you use at line 1, independently to achieve the requirement?

- A) `StringBuilder sb=new StringBuilder(creditCard);
sb.substring(15,19);
return x+sb;`
- B) `return x+creditCard.substring(15,19);`
- C) `StringBuilder sb=new StringBuilder(x);
sb.append(creditCard,15,19);
return sb.toString();`
- D) `StringBuilder sb=new StringBuilder(creditCard);
StringBuilder s=sb.insert(0,x);
return s.toString();`

Answer: B,C

Q5. Consider the following code

```
1) public class Test  
2) {  
3)     public static void main(String[] args)  
4)     {  
5)         String str=" ";  
6)         str.trim();  
7)         System.out.println(str.equals("")+" "+str.isEmpty());  
8)     }  
9) }
```

What is the result?

- A. true false
- B. true true
- C. false true
- D. false false

Answer: D



Q6. Consider the following code:

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String s1="Java";
6)         String s2= new String("java");
7)         //Line-1
8)     {
9)         System.out.println("Equal");
10)    }
11)    else
12)    {
13)        System.out.println("Not Equal");
14)    }
15) }
16} }
```

To print "Equal" which code fragment should be inserted at Line-1

- A. String s3=s2;
if(s1==s3)
- B. if(s1.equalsIgnoreCase(s2))
- C. String s3=s2;
if(s1.equals(s3))
- D. if(s1.toLowerCase()==s2.toLowerCase())

Answer: B

Q7. Given the following code:

```
1) class MyString
2) {
3)     String msg;
4)     MyString(String msg)
5)     {
6)         this.msg=msg;
7)     }
8) }
9) public class Test
10) {
11)     public static void main(String[] args)
12) { }
```



```
13) System.out.println("Hello "+ new StringBuilder("Java SE 8"));
14) System.out.println("Hello "+ new MyString("Java SE 8"));
15) }
16) }
```

What is the result?

- A) Hello Java SE 8
Hello MyString@<hashcode>
- B) Hello Java SE 8
Hello Java SE 8
- C) Hello java.lang.StringBuilder@<hashcode>
Hello MyString@<hashcode>
- D) Compilation Fails

Answer: A

Q8. Given:

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String s="Java Duke";
6)         int len=s.trim().length();
7)         System.out.println(len);
8)     }
9) }
```

What is the result?

- A. 9
- B. 8
- C. 10
- D. Compilation Fails

Answer: A

Q9. Which statement will empty the contents of a StringBuilder variable named sb?

- A. sb.deleteAll();
- B. sb.delete(0,sb.size());
- C. sb.delete(0,sb.length());
- D. sb.removeAll();

Answer: C



Q10. Given

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String s="Hello World";
6)         s.trim();
7)         int i1=s.indexOf(" ");
8)         System.out.println(i1);
9)     }
10) }
```

What is the result?

- A. An exception is thrown at runtime
- B. -1
- C. 5
- D. 0

Answer: C



Wrapper Classes

The Main Objectives of Wrapper Classes are:

- To Wrap Primitives into Object Form. So that we can handle Primitives also Just Like Objects.
- To Define Several Utility Methods for the Primitives (Like converting Primitive to the String Form).

Constructors:

- All Most All Wrapper Classes defines 2 Constructors.
- One can take corresponding Primitive as an Argument and the Other can take String as an Argument.

```
Integer I = new Integer(10);
Integer I = new Integer("10");
Double D = new Double(10.5);
Double D = new Double("10.5");
```

- If the String Argument is not representing Number then we will get RE: NumberFormatException.

Eg: Integer I = new Integer("Ten");
//RE: java.lang.NumberFormatException: For input string: "Ten"

- Float Class defines 3 Constructors with float, double and String Arguments.

```
Float F = new Float(10.5f); √
Float F = new Float("10.5f"); √
Float F = new Float(10.5); √
Float F = new Float("10.5"); √
```

- Character Class contains only one Constructor with *char* Primitive as an Argument Type.

```
Character ch = new Character('a'); √
Character ch = new Character("a"); X
```

- Boolean Class contains 2 Constructors with boolean Primitive and String Arguments.
- If we Pass *boolean* Primitive as an Argument the only allowed Values are *true* OR *false*, Where Case and Content Both are Important.

```
Boolean B = new Boolean(true); √
Boolean B = new Boolean(false); √
Boolean B = new Boolean(True); X
Boolean B = new Boolean(TRUE); X
```



- If we are passing String Argument then Case and Content Both are Not Important.
- If the Content is Case Insensitive String(true) then it is treated as true. Otherwise it is treated as false.

```
Boolean B = new Boolean("true"); → true
Boolean B = new Boolean("True"); → true

Boolean B = new Boolean("TRUE"); → true
Boolean B = new Boolean("false"); → false

Boolean B = new Boolean("Malaika"); → false
Boolean B = new Boolean("Mallika"); → false
Boolean B = new Boolean("Jareena"); → false
```

```
Boolean X = new Boolean("Yes");
Boolean Y = new Boolean("No");

System.out.println(X); //false
System.out.println(Y); //false

System.out.println(X.equals(Y)); //true
System.out.println(X == y); //false
```

Wrapper Class	Corresponding Constructor Arguments
Byte	byte OR String
Short	short OR String
Integer	int OR String
Long	long OR String
Float	float OR String OR double
Double	double OR String
Character	char
Boolean	Boolean OR String

Note:

- In every Wrapper Class `toString()` is Overridden for Meaningful String Representation.
- In every Wrapper Class `equals()` is Overridden for Content Comparision.

Utility Methods

- ❖ `valueOf()`
- ❖ `xxxValue()`
- ❖ `parseXxx()`
- ❖ `toString()`



1) **valueOf()**: We can use `valueOf()` to Create Wrapper Object as Alternative to Constructor.

- **Form 1:** All the Wrapper Classes except Character Class contains a Static `valueOf()` to Create Wrapper Object for the given String.

```
public static wrapper valueOf(String s);
```

```
Integer I = Integer.valueOf('10');  
Float F = Float.valueOf("10.5");  
Boolean B = Boolean.valueOf("Durga");
```

- **Form 2:** All Integral Wrapper Classes (Byte, Short, Integer, Long) Contains the following `valueOf()` to Create Wrapper Object for the given specified Radix String.

```
public static wrapper valueOf(String s, int radix);
```

- The allowed Range of Radix is 2 to 36.
- Because Numerics(10), Alphabets(26) Finally $10 + 26 = 36$.

```
Integer I = Integer.valueOf("100", 2);  
System.out.println(I); //4
```

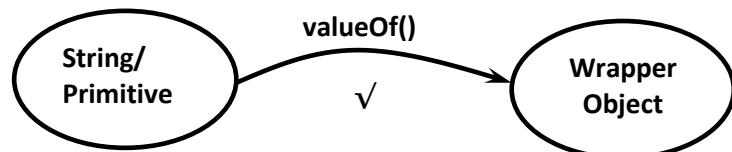
```
Integer I = Integer.valueOf("101", 4);  
System.out.println(I); //17
```

- **Form 3:** Every Wrapper Class including Character Class contains the following `valueOf()` to Convert Primitive to Wrapper Object Form.

```
public static wrapper valueOf(primitive p);
```

```
Integer I = Integer.valueOf(10);  
Character ch = Character.valueOf('a');  
Boolean B = Boolean.valueOf(true);
```

Version1, Version2 ➔ String to Wrapper Object.
Version3 ➔ Primitive to Wrapper Object.



2) **XxxValue():**

- We can use `XxxValue()` to find Primitive Value for the given Wrapper Object.
- Every Number Type Wrapper Class [Byte, Short, Integer, Long, Float and Double] contains the following `XxxValue()` to find Primitive Value for the given Wrapper Object.



Examples

```
public byte byteValue();
public short shortValue();
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();
```

```
Integer I = Integer.valueOf(130); OR Integer I = new Integer(130);
System.out.println(I.byteValue()); // -126
System.out.println(I.shortValue()); // 130
System.out.println(I.intValue()); // 130
System.out.println(I.longValue()); // 130
System.out.println(I.floatValue()); // 130.0
System.out.println(I.doubleValue()); // 130.0
```

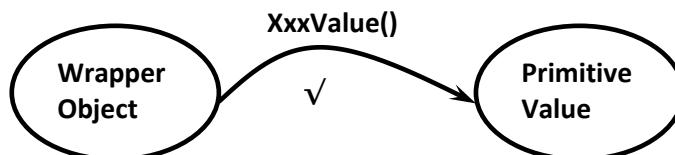
charValue(): Character Class contains `charValue()` to find char Primitive for the given kind of Character Object. `public char charValue()`

```
Character ch = new Character('a');
char ch1 = ch.charValue();
System.out.println(ch1); // a
```

booleanValue(): Boolean Class contains `booleanValue()` to Return boolean Primitive for the given Boolean Object. `public boolean booleanValue();`

```
Boolean B = Boolean.valueOf("Durga");
boolean b = B.booleanValue();
System.out.println(b); // false
```

Note: In Total there are 38 ((6 X 6) +1 + 1) `xxxValue()` Methods are Available.



3) parseXxx(): We can use `parseXxx()` to Convert String to Primitive.

- **Form 1:** Every Wrapper Class except Character Class contains the following `parseXxx()` for converting String to Primitive Type.

```
public static primitive parseXxx(String s);
```

Eg:

```
int i = Integer.parseInt("10");
double d = Double.parseDouble("10.5");
boolean b = Boolean.parseBoolean("true");
```

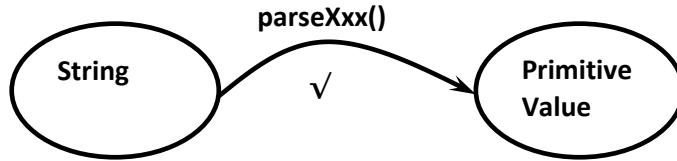
- **Form 2:** Every Integral Type Wrapper Class contains the following `parseXxx()` to Convert specified Radix String Form into Primitive.

```
public static primitive parseXxx(String s, int radix );
```

The allowed Range of Radix is 2 To 36.



Eg: `int i = Integer.parseInt("110", 2);
System.out.println(i); //6`



4) toString();

➤ **Form 1:**

- All Wrapper Classes contains an Instance Method `toString()` for converting Wrapper Object to String Type.
- This is Overriding Method of Object Class `toString()`.
- Whenever we are trying to Print Wrapper Object Reference internally this `toString()` will be Call. `public String toString();`

```
Integer I = new Integer(10);  
System.out.println(I); //10 → (I.toString());  
  
String s = I.toString();  
System.out.println(s); //10
```

➤ **Form 2:** Every Wrapper Class including Character Class defines the following Static `toString()` for converting Primitive to String Representation.

`public static String toString(primitive p);`

```
String s = Integer.toString(10);  
  
String s = Boolean.toString(true);  
  
String s3 = Character.toString('a');
```

➤ **Form 3:** Integer and Long Classes contains the following `toString()` to Convert Primitive to specified Radix String Form.

`public static String toString(primitive p, int radix);`

The allowed Range of Radix: 2 - 36

Eg: `String s = Integer.toString(7, 2);
System.out.println(s); //111`



➤ **Form 4: toXxxString()**

Integer and Long Classes contains the following toXxxString() Methods to Return specified Radix String Form.

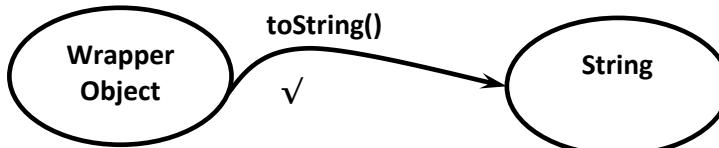
```
public static String toBinaryString(primitive p);  
  
public static String toOctalString(primitive p);  
  
public static String toHexString(primitive p);
```

```
String s = Integer.toBinaryString(10);  
System.out.println(s); //1010
```

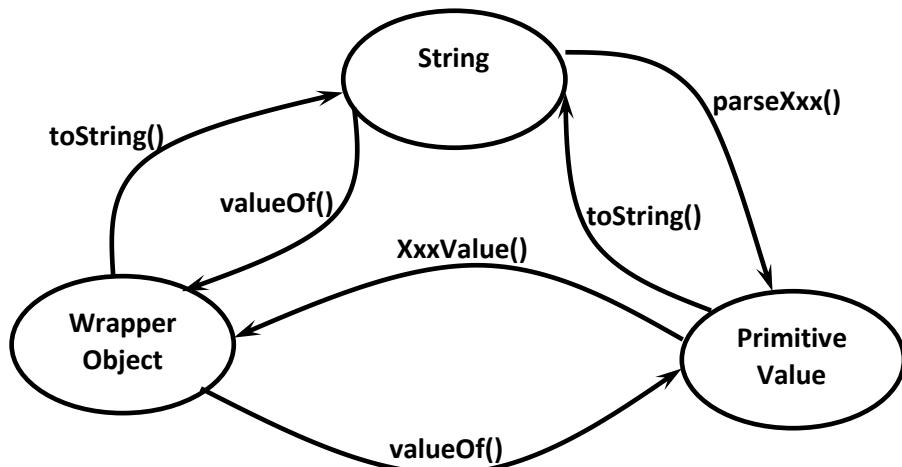
Examples:

```
String s = Integer.toOctalString(10);  
System.out.println(s); //12
```

```
String s = Integer.toHexString(10);  
System.out.println(s); //a
```



Dancing between Wrapper Object, Primitive and String





Practice Questions for Wrapper Classes

Q1. Given the code fragment:

```
1) class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         Short s1=200;
6)         Integer s2=400;
7)         Long s3=(long)s1+s2;//Line-1
8)         String s4=(String)(s3*s2);//Line-2
9)         System.out.println("Sum is:"+s4);
10)    }
11) }
```

What is the result?

- A. Sum is: 600
- B. Compilation Fails at Line-1
- C. Compilation Fails at Line-2
- D. ClassCastException is thrown at Line-1
- E. ClassCastException is thrown at Line-2

Answer: C

Q2. Consider the code:

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         Boolean[] b = new Boolean[2];
6)         b[0]=new Boolean(Boolean.parseBoolean("true"));
7)         b[1]= new Boolean(null);
8)         System.out.println(b[0].."." +b[1]);
9)     }
10) }
```

What is the result?

- A. true..false
- B. true..null
- C. Compilation Fails
- D. NullPointerException is thrown at runtime

Answer: A



Q3. Given:

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         boolean a = new Boolean(Boolean.valueOf(args[0]));
6)         boolean b = new Boolean(args[1]);
7)         System.out.println(a+".." +b);
8)     }
9) }
```

And given the commands:

```
javac Test.java
java Test TRUE null
```

What is the result?

- A. true..null
- B. true..false
- C. false..false
- D. true..true

Answer: B

Q4. Given the code

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         String s1="123";
6)         String s2="TRUE";
7)         Integer i1=Integer.parseInt(s1);
8)         Boolean b1= Boolean.parseBoolean(s2);
9)         System.out.println(i1+".." +b1);
10)
11)        int i2= Integer.valueOf(s1);
12)        boolean b2=Boolean.valueOf(s2);
13)        System.out.println(i2+".." +b2);
14)    }
15) }
```



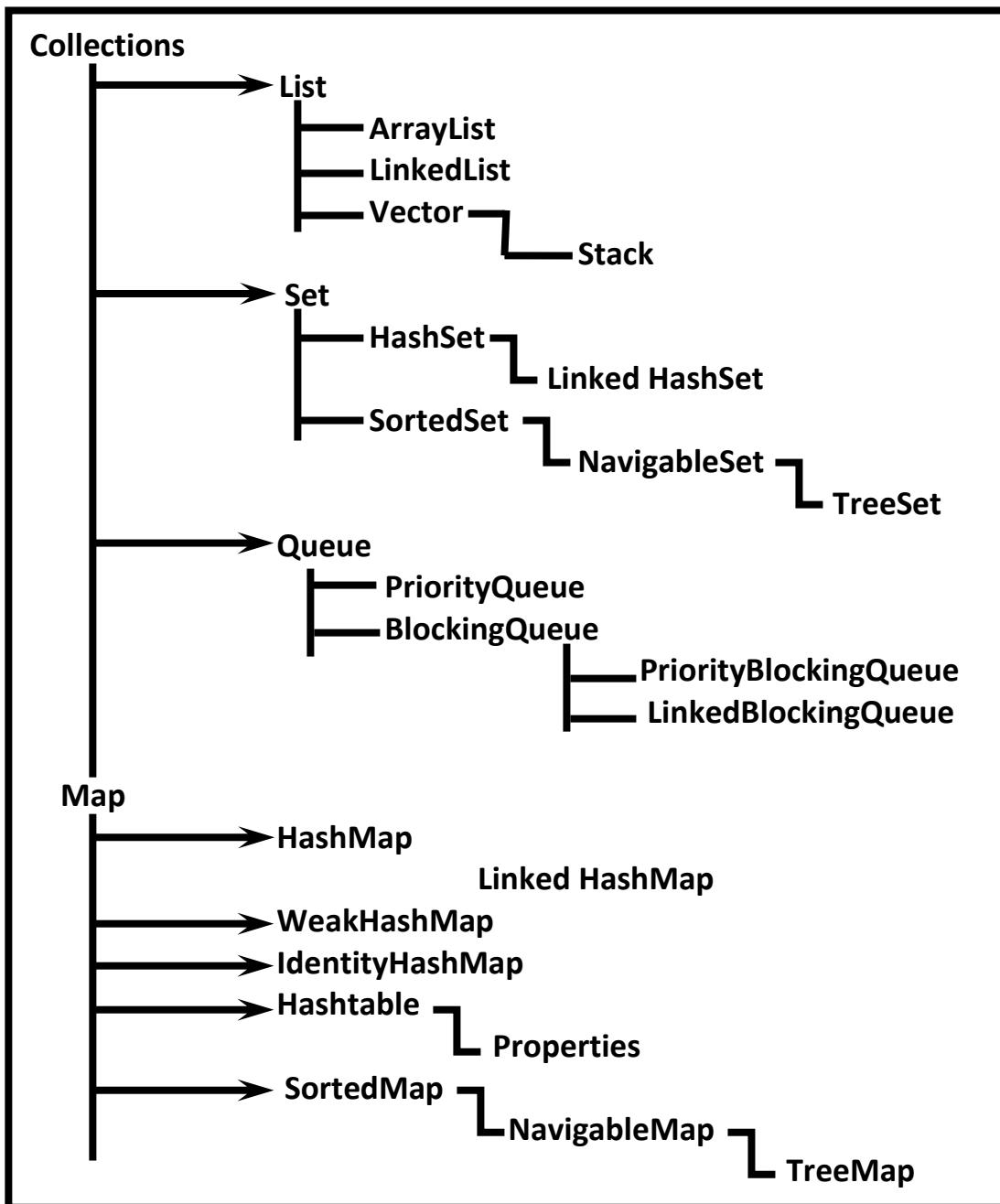
What is the result?

- A) 123..true
123..true
- B) 123..true
123..false
- C) 123..false
123..true
- D) Compilation Fails

Answer: A



Collections Framework And ArrayList



Cursors

- ❖ **Enumerations (I)**
- ❖ **Iterator (I)**
- ❖ **ListIterator (I)**

Utility Classes

- ❖ **Collections**
- ❖ **Arrays**

Sorting

- ❖ **Comparable (I)**
- ❖ **Comparator (I)**



An Array is an Indexed Collection of Fixed Number of Homogeneous Data Elements. The Main Advantage of Arrays is we can Represent Multiple Values by using Single Variable so that Readability of the Code will be Improved.

Limitations Of Object Type Arrays:

- 1) Arrays are Fixed in Size that is Once we created an Array there is No Chance of Increasing OR Decreasing Size based on Our Requirement. Hence to Use Arrays Concept Compulsory we should Know the Size in Advance which May Not be Possible Always.
- 2) Arrays can Hold Only Homogeneous Data Type Elements.

Eg:

```
Student[] s = new Student[10000];
s[0] = new Student(); ✓

s[1]=new Customer(); ✗
CE: incompatible types
  found: Costomer
  required: Student
```

We can Resolve this Problem by using Object Type Arrays.

Eg:

```
Object[] a = new Object[10000];
a[0] = new Student(); ✓
a[1] = new Customer(); ✓
```

- 3) Arrays Concept is Not implemented based on Some Standard Data Structure Hence Readymade Methods Support is Not Available. Hence for Every Requirement we have to write the Code Explicitly which Increases Complexity of the Programming.

To Overcome Above Problems of Arrays we should go for Collections.

Advantages Of Collections:

- 1) Collections are Growable in Nature. That is based on Our Requirement we can Increase OR Decrease the Size.
- 2) Collections can Hold Both Homogeneous and Heterogeneous Elements.
- 3) Every Collection Class is implemented based on Some Standard Data Structure. Hence for Every Requirement Readymade Method Support is Available. Being a Programmer we have to Use those Methods and we are Not Responsible to Provide Implementation.



Differences Between Arrays And Collections:

Arrays	Collections
Arrays are Fixed in Size.	Collections are Growable in Nature.
With Respect to Memory Arrays are Not Recommended to Use.	With Respect to Memory Collections are Recommended to Use.
With Respect to Performance Arrays are Recommended to Use.	With Respect to Performance Collections are Not Recommended to Use.
Arrays can Hold Only Homogeneous Data Elements.	Collections can Hold Both <i>Homogeneous</i> and <i>Heterogeneous</i> Elements.
Arrays can Hold Both Primitives and Objects.	Collections can Hold Only Objects but Not Primitives.
Arrays Concept is Not implemented based on Some Standard Data Structure. Hence Readymade Method Support is Not Available.	For every Collection class underlying Data Structure is Available Hence Readymade Method Support is Available for Every Requirement.

Collection:

If we want to Represent a Group of Individual Objects as a Single Entity then we should go for Collection.

Collection Frame Work:

It defines Several Classes and Interfaces which can be used to Represent a Group of Objects as a Single Entity.

JAVA	C++
Collection Collection Frame Work	Container Standard Template Library (STL)

9 Key Interfaces Of Collection Framework:

- 1) Collection (I)
- 2) List (I)
- 3) Set (I)
- 4) SortedSet (I)
- 5) NavigableSet (I)
- 6) Queue (I)
- 7) Map (I)
- 8) SortedMap (I)
- 9) NavigableMap (I)



1) Collection (I):

- If we want to Represent a Group of Individual Objects as a Single Entity then we should go for Collections.
- Collection Interface is considered as Root Interface of Collection Framework.
- Collection Interface defines the Most Common Methods which are Applicable for any Collection Object.

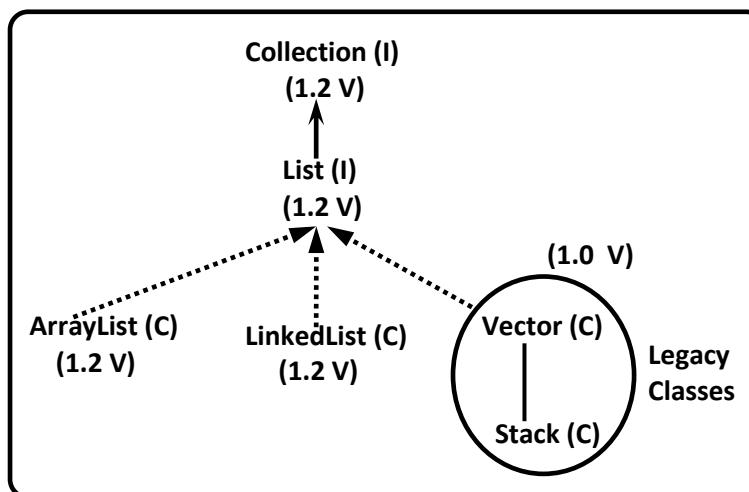
Difference Between Collection (I) and Collections (C):

- Collection is an Interface which can be used to Represent a Group of Individual Objects as a Single Entity.
- Whereas Collections is an Utility Class Present in `java.util` Package to Define Several Utility Methods for Collection Objects.

Note: There is No Concrete Class which implements Collection Interface Directly.

2) List (I):

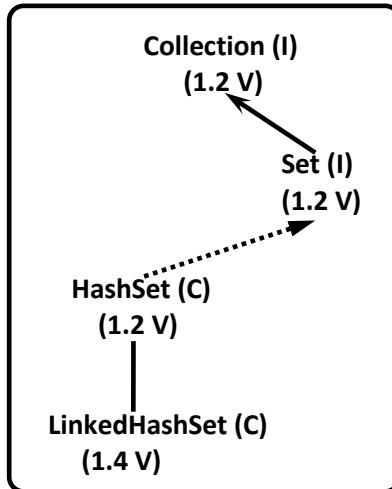
- It is the Child Interface of Collection.
- If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are allowed and Insertion Order Preserved. Then we should go for List.



Note: In 1.2 Version onwards `Vector` and `Stack` Classes are re-engineered to Implement List Interface.

3) Set (I):

- It is the Child Interface of the Collection.
- If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed and Insertion Order won't be Preserved. Then we should go for Set Interface.

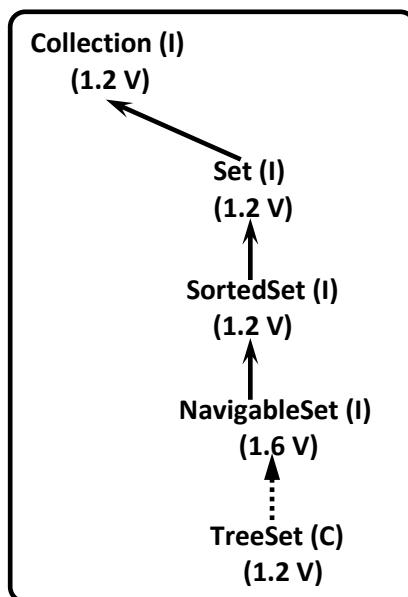


4) SortedSet (I):

- It is the Child Interface of Set.
- If we want to Represent a Group of Individual Objects Without Duplicates According to Some Sorting Order then we should go for `SortedSet`.

5) NavigableSet (I):

- It is the Child Interface of `SortedSet`.
- It defines Several Methods for Navigation Purposes.

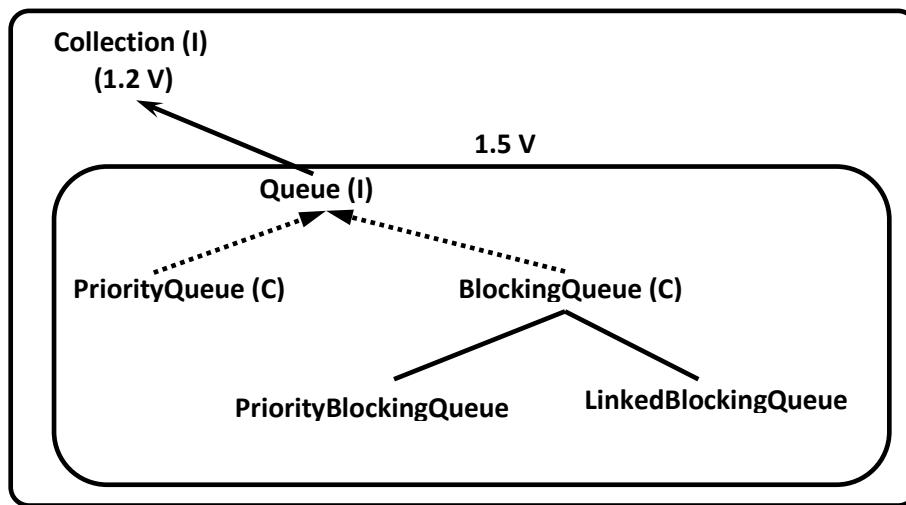


6) Queue (I):

- It is the Child Interface of Collection.
- If we want to Represent a Group of Individual Objects Prior to Processing then we should go for `Queue`.



Eg: Before sending a Mail we have to Store All MailID's in Some Data Structure and in which Order we added MailID's in the Same Order Only Mails should be delivered (FIFO). For this Requirement Queue is Best Suitable.

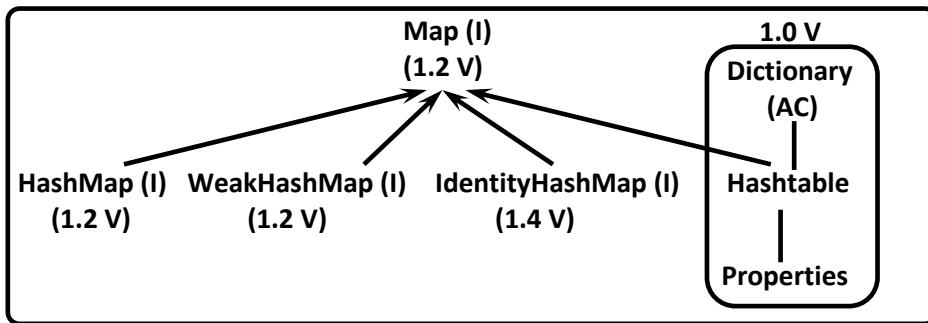


NOTE:

- All the Above Interfaces (Collection, List, Set, SortedSet, NavigableSet, and Queue) Meant for representing a Group of Individual Objects.
- If we want to Represent a Group of Key - Value Pairs then we should go for Map.

7) Map (I):

- Map is Not Child Interface of Collection.
- If we want to Represent a Group of Objects as Key - Value Pairs then we should go for Map Interface.
- Duplicate Keys are Not allowed but Values can be Duplicated.

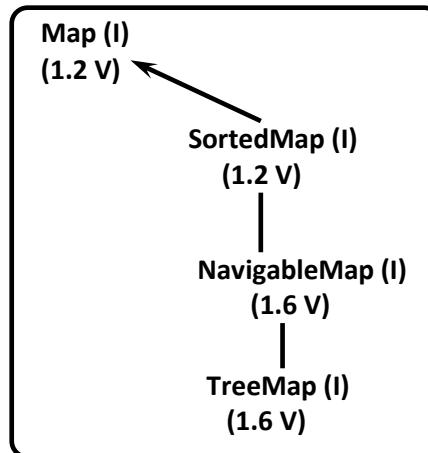


8) SortedMap (I):

- It is the Child Interface of Map.
- If we want to Represent a Group of Objects as Key - Value Pairs according to Some Sorting Order of Keys then we should go for SortedMap.
- Sorting should be Based on Key but Not Based on Value.

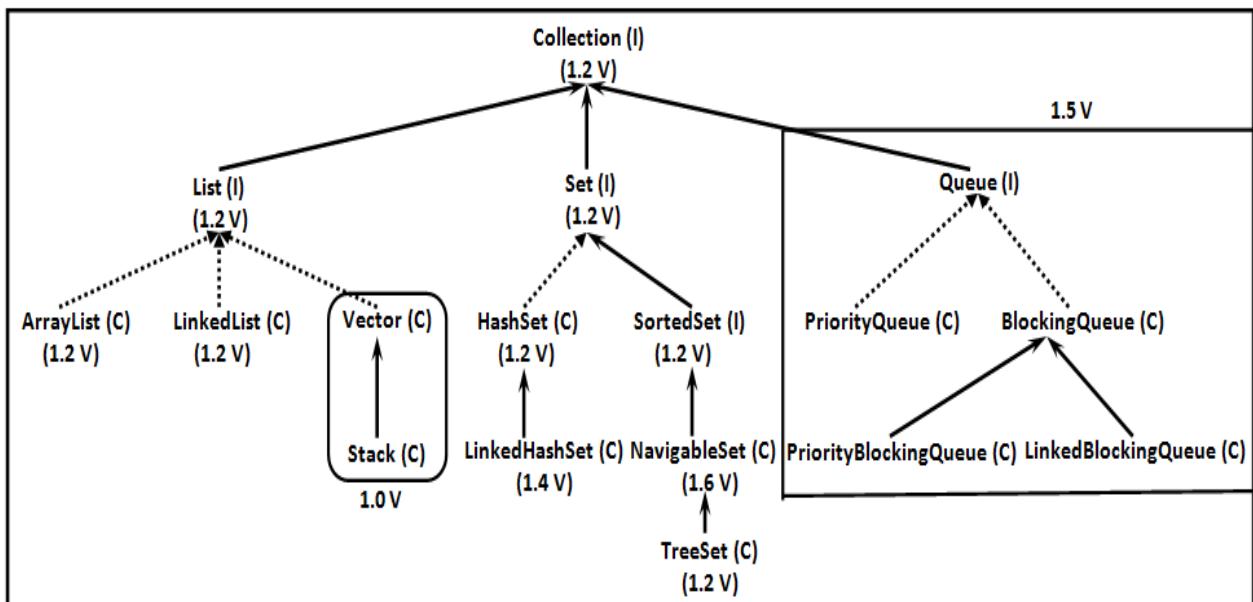
9) NavigableMap (I):

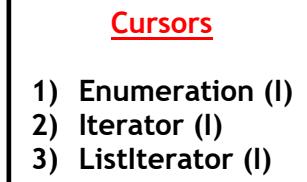
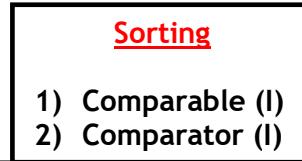
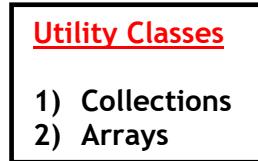
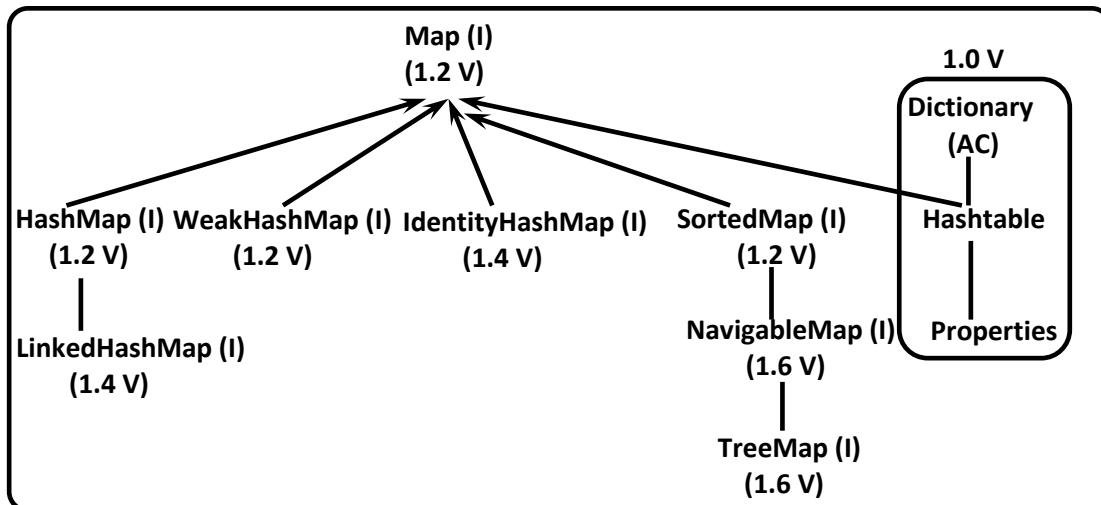
- It is the Child Interface of SortedMap.
- It Defines Several Methods for Navigation Purposes.



Note: In Collection Framework the following are Legacy Classes.

- 1) Enumeration (I)
- 2) Dictionary (Abstract Class)
- 3) Vector (Concrete Class)
- 4) Stack (Concrete Class)
- 5) Hashtable (Concrete Class)
- 6) Properties (Concrete Class)





1) Collection Interface:

If we want to Represent a Group of Individual Objects as a Single Entity then we should go for Collection Interface.

Methods:

- Collection Interface defines the Most Common Methods which are Applicable for any Collection Objects.
- The following is the List of the Methods Present Inside Collection Interface.

- 1) `boolean add(Object o)`
- 2) `boolean addAll(Collection c)`
- 3) `boolean remove(Object o)`
- 4) `boolean removeAll(Collection c)`
- 5) **boolean retainAll(Collection c):** To Remove All Objects Except those Present in c.
- 6) `void clear()`
- 7) `boolean contains(Object o)`



8) boolean containsAll(Collection c)

9) boolean isEmpty()

10) int size()

11) Object[] toArray()

12) Iterator iterator()

Note:

- There is No Concrete Class which implements Collection Interface Directly.
- There is No Direct Method in Collection Interface to get Objects.

2) List:

- It is the Child Interface of Collection.
- If we want to Represent a Group of Individual Objects where Duplicates are allowed and Insertion Order Preserved. Then we should go for List.
- We can Preserve Insertion Order and we can Differentiate Duplicate Object by using Index. Hence Index will Play Very Important Role in List.

Methods: List Interface Defines the following Specific Methods.

1) void add(int index, Object o)

2) boolean addAll(int index, Collection c)

3) Object get(int index)

4) Object remove(int index)

5) **Object set(int index, Object new):** To Replace the Element Present at specified Index with provided Object and Returns Old Object.

6) **int indexOf(Object o):** Returns Index of 1st Occurrence of 'o'

7) int lastIndexOf(Object o)

8) ListIterator listIterator();

2.1) ArrayList:

- The Underlying Data Structure for ArrayList is Resizable Array OR Growable Array.
- Duplicate Objects are allowed.
- Insertion Order is Preserved.
- Heterogeneous Objects are allowed (Except TreeSet and TreeMap Everywhere Heterogeneous Objects are allowed).
- null Insertion is Possible.



Constructors:

1) ArrayList l = new ArrayList();

- Creates an Empty ArrayList Object with Default Initial Capacity 10.
- If ArrayList Reaches its Max Capacity then a New ArrayList Object will be Created with

New Capacity = (Current Capacity * 3/2) + 1

2) ArrayList l = new ArrayList(int initialCapacity);

Creates an Empty ArrayList Object with specified Initial Capacity.

3) ArrayList l = new ArrayList(Collection c);

- Creates an Equalent ArrayList Object for the given Collection Object.
- This Constructor Meant for Inter Conversion between Collection Objects.

```
import java.util.ArrayList;
class ArrayListDemo {
    public static void main(String[] args) {

        ArrayList l = new ArrayList();

        l.add("A");
        l.add(10);
        l.add("A");
        l.add(null);
        System.out.println(l); // [A, 10, A, null]

        l.remove(2);
        System.out.println(l); // [A, 10, null]

        l.add(2,"M");
        l.add("N");
        System.out.println(l); // [A, 10, M, null, N]

    }
}
```

- Usually we can Use Collections to Hold and Transfer Data (Objects) form One Location to Another Location.
- To Provide Support for this Requirement Every Collection Class Implements *Serializable* and *Cloneable* Interfaces.
- *ArrayList* and *Vector* Classes Implements *RandomAccess* Interface. So that we can Access any Random Element with the Same Speed.
- *RandomAccess* Interface Present in *java.util* Package and it doesn't contain any Methods. Hence it is a *Marker* Interface.
- Hence *ArrayList* is Best Suitable if Our Frequent Operation is Retrieval Operation.



```
ArrayList l1 = new ArrayList();
LinkedList l2 = new LinkedList();

System.out.println(l1 instanceof Serializable); //true
System.out.println(l2 instanceof Cloneable); //true
System.out.println(l1 instanceof RandomAccess); //true
System.out.println(l2 instanceof RandomAccess); //false
```

Differences between ArrayList and Vector:

ArrayList	Vector
Every Method Present Inside ArrayList is Non – Synchronized.	Every Method Present in Vector is Synchronized.
At a Time Multiple Threads are allow to Operate on ArrayList Simultaneously and Hence ArrayList Object is Not Thread Safe.	At a Time Only One Thread is allow to Operate on Vector Object and Hence Vector Object is Always Thread Safe.
Relatively Performance is High because Threads are Not required to Wait.	Relatively Performance is Low because Threads are required to Wait.
Introduced in 1.2 Version and it is Non – Legacy.	Introduced in 1.0 Version and it is Legacy.

How to get Synchronized Version of ArrayList Object?

By Default ArrayList Object is Non - Synchronized but we can get Synchronized Version ArrayList Object by using the following Method of Collections Class.

```
public static List synchronizedList(List l)
```

Eg:

```
ArrayList al = new ArrayList ();
List l = Collections.synchronizedList(al);
```

↓ ↓
Synchronized Non - Synchronized
Version Version

Similarly we can get Synchronized Version of Set and Map Objects by using the following Methods of Collection Class.

```
public static Set synchronizedSet(Set s)
```

```
public static Map synchronizedMap(Map m)
```

- ArrayList is the Best Choice if we want to Perform Retrieval Operation Frequently.
- But ArrayList is Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle. Because it required Several Shift Operations Internally.



Practice Questions for ArrayList

Q1. Given the code fragment:

```
1) import java.util.*;
2) class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         List<String> l = new ArrayList<>();
7)         l.add("Robb");
8)         l.add("Bran");
9)         l.add("Rick");
10)        l.add("Bran");
11)        if(l.remove("Bran"))
12)        {
13)            l.remove("Jon");
14)        }
15)        System.out.println(l);
16)    }
17) }
```

What is the result?

- A. [Robb, Rick, Bran]
- B. [Robb, Rick]
- C. [Robb, Bran, Rick, Bran]
- D. An exception is thrown at runtime

Answer: A

Q2. Given the code fragment

```
1) import java.util.*;
2) class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         ArrayList l = new ArrayList();
7)         String[] s;
8)         try
9)         {
10)             while(true)
11)             {
12)                 l.add("MyString");
13)             }
14)         }
```



```
15)     catch (RuntimeException e)
16)     {
17)         System.out.println("RuntimeException caught");
18)     }
19)     catch (Exception e)
20)     {
21)         System.out.println("Exception caught");
22)     }
23)     System.out.println("Ready to use");
24) }
25) }
```

What is the result?

- A.
RuntimeException caught
Ready to use
- B.
Exception caught
Ready to use
- C. Compilation Fails
- D. A runtime error thrown in the thread main

Answer: D

Q3) Given:

```
1) import java.util.*;
2) class Patient {
3)     String name;
4)     public Patient(String name)
5)     {
6)         this.name=name;
7)     }
8) }
9) class Test
10) {
11)     public static void main(String[] args)
12)     {
13)         List l = new ArrayList();
14)         Patient p = new Patient("Mike");
15)         l.add(p);
16)         //insert code here==>Line-1
17)         if(f>=0)
18)         {
19)             System.out.println("Mike Found");
20)         }
21)     }
22) }
```



Which code inserted at Line-1 enable the code to print Mike Found.

- A. int f=l.indexOf(p);
- B. int f=l.indexOf(Patient("Mike"));
- C. int f=l.indexOf(new Patient("Mike"));
- D. Patient p1 = new Patient("Mike");
int f=l.indexOf(p1);

Answer: A

Q4. Given the code fragment:

```
1) import java.util.*;  
2) class Test  
3) {  
4)     public static void main(String[] args)  
5)     {  
6)         ArrayList<Integer> l = new ArrayList<>();  
7)         l.add(1);  
8)         l.add(2);  
9)         l.add(3);  
10)        l.add(4);  
11)        l.add(null);  
12)        l.remove(2);  
13)        l.remove(null);  
14)        System.out.println(l);  
15)    }  
16} }
```

What is the result?

- A. [1, 2, 4]
- B. NullPointerException is thrown at runtime
- C. [1, 2, 4,null]
- D. [1, 3, 4,null]
- E. [1, 3, 4]
- F. Compilation Fails

Answer: A

Q5. Given the following class declarations

```
public abstract class Animal  
public interface Hunter  
public class Cat extends Animal implements Hunter  
public class Tiger extends Cat
```



Which one fails to compile?

A.

```
ArrayList<Animal> l = new ArrayList<>();  
l.add(new Tiger());
```

B.

```
ArrayList<Hunter> l = new ArrayList<>();  
l.add(new Cat());
```

C.

```
ArrayList<Hunter> l = new ArrayList<>();  
l.add(new Tiger());
```

D.

```
ArrayList<Tiger> l = new ArrayList<>();  
l.add(new Cat());
```

E.

```
ArrayList<Animal> l = new ArrayList<>();  
l.add(new Cat());
```

Answer: D



Lambda Expression And Predicates

Java 7 – July 28th 2011
2 Years 7 Months 18 Days

Java 8 - March 18th 2014

Java 9 - September 22nd 2016

Java 10 - 2018

After Java 1.5version, Java 8 is the next major version.

Before Java 8, sun people gave importance only for objects but in 1.8version oracle people gave the importance for functional aspects of programming to bring its benefits to Java.ie it doesn't mean Java is functional oriented programming language.

Java 8 New Features:

- 1) Lambda Expression
- 2) Functional Interfaces
- 3) Default methods
- 4) Predicates
- 5) Functions
- 6) Double colon operator (::)
- 7) Stream API
- 8) Date and Time API
- Etc.....



Lambda (λ) Expression

- ✿ Lambda calculus is a big change in mathematical world which has been introduced in 1930. Because of benefits of Lambda calculus slowly this concepts started using in programming world. "LISP" is the first programming which uses Lambda Expression.
- ✿ The other languages which uses lambda expressions are:
 - C#.Net
 - C Objective
 - C
 - C++
 - Python
 - Ruby etc.and finally in Java also.
- ✿ The Main Objective of Lambda Expression is to bring benefits of functional programming into Java.

What is Lambda Expression (λ):

- Lambda Expression is just an anonymous (nameless) function. That means the function which doesn't have the name, return type and access modifiers.
- Lambda Expression also known as anonymous functions or closures.

Ex: 1

```
public void m1() { } () → {}  
    sop("hello"); } sop("hello");  
} () → { sop("hello"); }  
        () → sop("hello");
```

Ex:2

```
public void add(int a, int b) { } (int a, int b) → sop(a+b);  
    sop(a+b); }
```

- If the type of the parameter can be decided by compiler automatically based on the context then we can remove types also.
- The above Lambda expression we can rewrite as $(a,b) \rightarrow sop(a+b)$;



Ex: 3

```
public String str(String str) { } } (String str) → return str;  
    return str;                                ↓  
                                                (str) → str;
```

Conclusions:

- 1) A lambda expression can have zero or more number of parameters (arguments).

Ex:

$() \rightarrow \text{sop}(\text{"hello"})$;
 $(\text{int a}) \rightarrow \text{sop}(a)$;
 $(\text{int a, int b}) \rightarrow \text{return } a+b$;

- 2) Usually we can specify type of parameter. If the compiler expects the type based on the context then we can remove type. i.e., programmer is not required.

Ex:

$(\text{int a, int b}) \rightarrow \text{sop}(a+b)$;
↓
 $(a,b) \rightarrow \text{sop}(a+b)$;

- 3) If multiple parameters present then these parameters should be separated with comma (,).
- 4) If zero number of parameters available then we have to use empty parameter [like ()].

Ex: $() \rightarrow \text{sop}(\text{"hello"})$;

- 5) If only one parameter is available and if the compiler can expect the type then we can remove the type and parenthesis also.

Ex:

$(\text{int a}) \rightarrow \text{sop}(a)$;
↓
 $(a) \rightarrow \text{sop}(a)$;
↓
 $A \rightarrow \text{sop}(a)$;

- 6) Similar to method body lambda expression body also can contain multiple statements. If more than one statements present then we have to enclose inside within curly braces. If one statement present then curly braces are optional.
- 7) Once we write lambda expression we can call that expression just like a method, for this functional interfaces are required.



Functional Interfaces

If an interface contain only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

Ex:

- 1) Runnable → It contains only run() method
- 2) Comparable → It contains only compareTo() method
- 3) ActionListener → It contains only actionPerformed()
- 4) Callable → It contains only call() method

Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.

Ex:

```
1) interface Interf {  
2)     public abstract void m1();  
3)     default void m2() {  
4)         System.out.println ("hello");  
5)     }  
6) }
```

In Java 8, Sun Micro System introduced `@Functional Interface` annotation to specify that the interface is Functional Interface.

Ex:

```
@Functional Interface  
Interface Interf {  
    public void m1();  
}  
} ] This code compiles without any compilation errors.
```

Inside Functional Interface we can take only one abstract method, if we take more than one abstract method then compiler raise an error message that is called we will get compilation error.

Ex:

```
@Functional Interface {  
    public void m1();  
    public void m2();  
}  
} ] This code gives compilation error.
```



Inside Functional Interface we have to take exactly only one abstract method. If we are not declaring that abstract method then compiler gives an error message.

Ex:

```
@Functional Interface {  
    interface Interface { } } compilation error  
}
```

Functional Interface with respect to Inheritance:

If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also Functional Interface

Ex:

```
1) @Functional Interface  
2) interface A {  
3)     public void methodOne();  
4) }  
5) @Functional Interface  
6) Interface B extends A {  
7) }
```

In the child interface we can define exactly same parent interface abstract method.

Ex:

```
1) @Functional Interface  
2) interface A {  
3)     public void methodOne();  
4) }  
5) @Functional Interface  
6) interface B extends A {  
7)     public void methodOne(); } } No Compile Time Error  
8) }
```

In the child interface we can't define any new abstract methods otherwise child interface won't be Functional Interface and if we are trying to use @Functional Interface annotation then compiler gives an error message.



```
1) @Functional Interface {  
2) interface A {  
3)     public void methodOne();  
4) }  
5) @Functional Interface  
6) interface B extends A {  
7)     public void methodTwo();  
8) }
```

} **ComplieTime Error**

Ex:

```
@Functional Interface  
interface A {  
    public void methodOne();  
}  
interface B extends A {  
    public void methodTwo();  
}
```

} **No compile time error**

} **This's Normal interface so that code compiles without error**

In the above example in both parent & child interface we can write any number of default methods and there are no restrictions. Restrictions are applicable only for abstract methods.

Functional Interface Vs Lambda Expressions:

Once we write Lambda expressions to invoke it's functionality, then Functional Interface is required. We can use Functional Interface reference to refer Lambda Expression.

Where ever Functional Interface concept is applicable there we can use Lambda Expressions

Ex:1 Without Lambda Expression

```
1) interface Interf {  
2)     public void methodOne() {}  
3)     public class Demo implements Interface {  
4)         public void methodOne() {  
5)             System.out.println("method one execution");  
6)         }  
7)         public class Test {  
8)             public static void main(String[] args) {  
9)                 Interfi = new Demo();  
10)                 i.methodOne();  
11)             }  
12) }
```



Above code With Lambda expression

```
1) interface Interf {  
2)     public void methodOne() {}  
3)     class Test {  
4)         public static void main(String[] args) {  
5)             Interfi = () → System.out.println("MethodOne Execution");  
6)             i.methodOne();  
7)         }  
8)     }
```

Without Lambda Expression

```
1) interface Interf {  
2)     public void sum(int a, int b);  
3) }  
4) class Demo implements Interf {  
5)     public void sum(int a, int b) {  
6)         System.out.println("The sum:" + (a+b));  
7)     }  
8) }  
9) public class Test {  
10)     public static void main(String[] args) {  
11)         Interfi = new Demo();  
12)         i.sum(20,5);  
13)     }  
14) }
```

Above code With Lambda Expression

```
1) interface Interf {  
2)     public void sum(int a, int b);  
3) }  
4) class Test {  
5)     public static void main(String[] args) {  
6)         Interfi = (a,b) → System.out.println("The Sum:" +(a+b));  
7)         i.sum(5,10);  
8)     }  
9) }
```



Without Lambda Expressions

```
1) interface Interf {  
2)     public int square(int x);  
3) }  
4) class Demo implements Interf {  
5)     public int square(int x) {  
6)         return x*x; OR (int x) → x*x  
7)     }  
8) }  
9) class Test {  
10)    public static void main(String[] args) {  
11)        Interfi = new Demo();  
12)        System.out.println("The Square of 7 is: " +i.square(7));  
13)    }  
14) }
```

Above code with Lambda Expression

```
1) interface Interf {  
2)     public int square(int x);  
3) }  
4) class Test {  
5)     public static void main(String[] args) {  
6)         Interfi = x → x*x;  
7)         System.out.println("The Square of 5 is:" +i.square(5));  
8)     }  
9) }
```

Without Lambda expression

```
1) class MyRunnable implements Runnable {  
2)     public void main() {  
3)         for(int i=0; i<10; i++) {  
4)             System.out.println("Child Thread");  
5)         }  
6)     }  
7) }  
8) class ThreadDemo {  
9)     public static void main(String[] args) {  
10)         Runnable r = new myRunnable();  
11)         Thread t = new Thread(r);  
12)         t.start();  
13)         for(int i=0; i<10; i++) {  
14)             System.out.println("Main Thread")  
15)         }  
16)     }
```



17) }

With Lambda expression

```
1) class ThreadDemo {  
2)     public static void main(String[] args) {  
3)         Runnable r = () -> {  
4)             for(int i=0; i<10; i++) {  
5)                 System.out.println("Child Thread");  
6)             }  
7)         };  
8)         Thread t = new Thread(r);  
9)         t.start();  
10)        for(i=0; i<10; i++) {  
11)            System.out.println("Main Thread");  
12)        }  
13)    }  
14) }
```

Anonymous inner classes vs Lambda Expressions

Wherever we are using anonymous inner classes there may be a chance of using Lambda expression to reduce length of the code and to resolve complexity.

Ex: With anonymous inner class

```
1) class Test {  
2)     public static void main(String[] args) {  
3)         Thread t = new Thread(new Runnable() {  
4)             public void run() {  
5)                 for(int i=0; i<10; i++) {  
6)                     System.out.println("Child Thread");  
7)                 }  
8)             }  
9)         });  
10)        t.start();  
11)        for(int i=0; i<10; i++) {  
12)            System.out.println("Main thread");  
13)        }  
14)    }
```



With Lambda expression

```
1) class Test {  
2)     public static void main(String[] args) {  
3)         Thread t = new Thread(() -> {  
4)             for(int i=0; i<10; i++) {  
5)                 System.out.println("Child Thread");  
6)             }  
7)         });  
8)         t.start();  
9)         for(int i=0; i<10; i++) {  
10)             System.out.println("Main Thread");  
11)         }  
12)     }  
13) }
```

What are the advantages of Lambda expression?

- We can reduce length of the code so that readability of the code will be improved.
- We can resolve complexity of anonymous inner classes.
- We can provide Lambda expression in the place of object.
- We can pass lambda expression as argument to methods.

Note:

- Anonymous inner class can extend concrete class, can extend abstract class, can implement interface with any number of methods but
- Lambda expression can implement an interface with only single abstract method (Functional Interface).
- Hence if anonymous inner class implements Functional Interface in that particular case only we can replace with lambda expressions. Hence wherever anonymous inner class concept is there, it may not possible to replace with Lambda expressions.
- Anonymous inner class! = Lambda Expression
- Inside anonymous inner class we can declare instance variables.
- Inside anonymous inner class “this” always refers current inner class object(anonymous inner class) but not related outer class object

Ex:

- Inside lambda expression we can't declare instance variables.
- Whatever the variables declare inside lambda expression are simply acts as local variables
- Within lambda expression ‘this’ keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression)



Ex:

```
1) interface Interf {  
2)     public void m1();  
3) }  
4) class Test {  
5)     int x = 777;  
6)     public void m2() {  
7)         Interfi = ()→ {  
8)             int x = 888;  
9)             System.out.println(x); 888  
10)            System.out.println(this.x); 777  
11)        };  
12)        i.m1();  
13)    }  
14)    public static void main(String[] args) {  
15)        Test t = new Test();  
16)        t.m2();  
17)    }  
18} }
```

- From lambda expression we can access enclosing class variables and enclosing method variables directly.
- The local variables referenced from lambda expression are implicitly final and hence we can't perform re-assignment for those local variables otherwise we get compile time error

Ex:

```
1) interface Interf {  
2)     public void m1();  
3) }  
4) class Test {  
5)     int x = 10;  
6)     public void m2() {  
7)         int y = 20;  
8)         Interfi = () → {  
9)             System.out.println(x); 10  
10)            System.out.println(y); 20  
11)            x = 888;  
12)            y = 999; //CE  
13)        };  
14)        i.m1();  
15)        y = 777;  
16)    }  
17)    public static void main(String[] args) {  
18)        Test t = new Test();  
19)        t.m2();  
20} }
```



21) }

Differences between anonymous inner classes and Lambda expression

Anonymous Inner class	Lambda Expression
It's a class without name	It's a method without name (anonymous function)
Anonymous inner class can extend Abstract and concrete classes	lambda expression can't extend Abstract and concrete classes
Anonymous inner class can implement An interface that contains any number of Abstract methods	lambda expression can implement an Interface which contains single abstract method (Functional Interface)
Inside anonymous inner class we can Declare instance variables.	Inside lambda expression we can't Declare instance variables, whatever the variables declared are simply acts as local variables.
Anonymous inner classes can be Instantiated	lambda expressions can't be instantiated
Inside anonymous inner class "this" Always refers current anonymous Inner class object but not outer class Object.	Inside lambda expression "this" Always refers current outer class object. That is enclosing class object.
Anonymous inner class is the best choice If we want to handle multiple methods.	Lambda expression is the best Choice if we want to handle interface With single abstract method (Functional Interface).
In the case of anonymous inner class At the time of compilation a separate Dot class file will be generated (outerclass\$1.class)	At the time of compilation no dot Class file will be generated for Lambda expression. It simply converts in to private method outer class.
Memory allocated on demand Whenever we are creating an object	Reside in permanent memory of JVM (Method Area).



Default Methods

- Until 1.7 version onwards inside interface we can take only public abstract methods and public static final variables (every method present inside interface is always public and abstract whether we are declaring or not).
- Every variable declared inside interface is always public static final whether we are declaring or not.
- But from 1.8 version onwards in addition to these, we can declare default concrete methods also inside interface, which are also known as defender methods.
- We can declare default method with the keyword “default” as follows

```
1) default void m1(){  
2)     System.out.println ("Default Method");  
3) }
```

- Interface default methods are by-default available to all implementation classes. Based on requirement implementation class can use these default methods directly or can override.

Ex:

```
1) interface Interf {  
2)     default void m1() {  
3)         System.out.println("Default Method");  
4)     }  
5) }  
6) class Test implements Interf {  
7)     public static void main(String[] args) {  
8)         Test t = new Test();  
9)         t.m1();  
10)    }  
11) }
```

- Default methods also known as defender methods or virtual extension methods.
- The main advantage of default methods is without effecting implementation classes we can add new functionality to the interface (backward compatibility).

Note: We can't override object class methods as default methods inside interface otherwise we get compile time error.



Ex:

```
1) interface Interf {  
2)     default int hashCode() {  
3)         return 10;  
4)     }  
5) }
```

CompileTimeError

Reason: Object class methods are by-default available to every Java class hence it's not required to bring through default methods.

Default method vs multiple inheritance

Two interfaces can contain default method with same signature then there may be a chance of ambiguity problem (diamond problem) to the implementation class. To overcome this problem compulsory we should override default method in the implementation class otherwise we get compile time error.

```
1) Eg 1:  
2) interface Left {  
3)     default void m1() {  
4)         System.out.println("Left Default Method");  
5)     }  
6) }  
7)  
8) Eg 2:  
9) interface Right {  
10)    default void m1() {  
11)        System.out.println("Right Default Method");  
12)    }  
13) }  
14)  
15) Eg 3:  
16) class Test implements Left, Right {}
```



How to override default method in the implementation class?

In the implementation class we can provide complete new implementation or we can call any interface method as follows.

interfacename.super.m1();

Ex:

```
1) class Test implements Left, Right {  
2)     public void m1() {  
3)         System.out.println("Test Class Method"); OR Left.super.m1();  
4)     }  
5)     public static void main(String[] args) {  
6)         Test t = new Test();  
7)         t.m1();  
8)     }  
9) }
```

Differences between interface with default methods and abstract class

Even though we can add concrete methods in the form of default methods to the interface, it won't be equal to abstract class.

Interface with Default Methods	Abstract Class
Inside interface every variable is Always public static final and there is No chance of instance variables	Inside abstract class there may be a Chance of instance variables which Are required to the child class.
Interface never talks about state of Object.	Abstract class can talk about state of Object.
Inside interface we can't declare Constructors.	Inside abstract class we can declare Constructors.
Inside interface we can't declare Instance and static blocks.	Inside abstract class we can declare Instance and static blocks.
Functional interface with default Methods Can refer lambda expression.	Abstract class can't refer lambda Expressions.
Inside interface we can't override Object class methods.	Inside abstract class we can override Object class methods.

Interface with default method != abstract class



Static methods inside interface:

- From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions.
- Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can't call interface static methods. We should call interface static methods by using interface name.

Ex:

```
1) interface Interf {  
2)     public static void sum(int a, int b) {  
3)         System.out.println("The Sum:"+ (a+b));  
4)     }  
5) }  
6) class Test implements Interf {  
7)     public static void main(String[] args) {  
8)         Test t = new Test();  
9)         t.sum(10, 20); //CE  
10)        Test.sum(10, 20); //CE  
11)        Interf.sum(10, 20);  
12)    }  
13} }
```

- As interface static methods by default not available to the implementation class, overriding concept is not applicable.
- Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

Ex:1

```
1) interface Interf {  
2)     public static void m1() {}  
3) }  
4) class Test implements Interf {  
5)     public static void m1() {}  
6) }
```

It's valid but not overriding



Ex:2

```
1) interface Interf {  
2)     public static void m1() {}  
3) }  
4) class Test implements Interf {  
5)     public void m1() {}  
6) }
```

This's valid but not overriding

Ex3:

```
1) class P {  
2)     private void m1() {}  
3) }  
4) class C extends P {  
5)     public void m1() {}  
6) }
```

This's valid but not overriding

From 1.8 version onwards we can write main() method inside interface and hence we can run interface directly from the command prompt.

Ex:

```
1) interface Interf {  
2)     public static void main(String[] args) {  
3)         System.out.println("Interface Main Method");  
4)     }  
5) }
```

At the command prompt:

Javac Interf.java

JavalInterf



Predicates

- A predicate is a function with a single argument and returns boolean value.
- To implement predicate functions in Java, Oracle people introduced Predicate interface in 1.8 version (i.e., Predicate<T>).
- Predicate interface present in *Java.util.function* package.
- It's a functional interface and it contains only one method i.e., test()

Ex:

```
interface Predicate<T> {  
    public boolean test(T t);  
}
```

As predicate is a functional interface and hence it can refers lambda expression

Ex:1 Write a predicate to check whether the given integer is greater than 10 or not.

Ex:

```
public boolean test(Integer I) {  
    if (I > 10) {  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```



```
(Integer I) → {  
    if(I > 10)  
        return true;  
    else  
        return false;  
}
```



$I \rightarrow (I > 10);$



```
predicate<Integer> p = I →(I > 10);  
System.out.println (p.test(100)); true  
System.out.println (p.test(7)); false
```



Program:

```
1) import Java.util.function;
2) class Test {
3)     public static void main(String[] args) {
4)         predicate<Integer> p = I → (i>10);
5)         System.out.println(p.test(100));
6)         System.out.println(p.test(7));
7)         System.out.println(p.test(true)); //CE
8)     }
9) }
```

1 Write a predicate to check the length of given string is greater than 3 or not.

```
Predicate<String> p = s → (s.length() > 3);
System.out.println (p.test("rvkb")); true
System.out.println (p.test("rk")); false
```

#-2 write a predicate to check whether the given collection is empty or not.

```
Predicate<collection> p = c → c.isEmpty();
```

Predicate joining

It's possible to join predicates into a single predicate by using the following methods.

```
and()
or()
negate()
```

these are exactly same as logical AND ,OR complement operators

Ex:

```
1) import Java.util.function.*;
2) class test {
3)     public static void main(string[] args) {
4)         int[] x = {0, 5, 10, 15, 20, 25, 30};
5)         predicate<integer> p1 = i->i>10;
6)         predicate<integer> p2=i -> i%2==0;
7)         System.out.println("The Numbers Greater Than 10:");
8)         m1(p1, x);
9)         System.out.println("The Even Numbers Are:");
10)        m1(p2, x);
11)        System.out.println("The Numbers Not Greater Than 10:");
12)        m1(p1.negate(), x);
13)        System.out.println("The Numbers Greater Than 10 And Even Are:");
14)        m1(p1.and(p2), x);
15)        System.out.println("The Numbers Greater Than 10 OR Even:");
16)        m1(p1.or(p2), x);
17)    }
```



```
18)     public static void m1(predicate<integer>p, int[] x) {  
19)         for(int x1:x) {  
20)             if(p.test(x1))  
21)                 System.out.println(x1);  
22)         }  
23)     }  
24) }
```



Practice Questions on Lambda Expressions

Q1. Given the code fragment:

Person.java:

```
1) public class Person
2) {
3)     String name;
4)     int age;
5)     public Person(String n,int a)
6)     {
7)         name=n;
8)         age=a;
9)     }
10)    public String getName()
11)    {
12)        return name;
13)    }
14)    public int getAge()
15)    {
16)        return age;
17)    }
18) }
```

Test.java:

```
1) class Test
2) {
3)     public static void checkAge(List<Person> list,Predicate<Person> predicate)
4)     {
5)         for (Person p: list)
6)         {
7)             if(predicate.test(p))
8)             {
9)                 System.out.println(p.name+" ");
10)            }
11)        }
12)    }
13)    public static void main(String[] args)
14)    {
15)        List<Person> iList=Arrays.asList(new Person("Durga",45),
16)                                         new Person("Ravi",40),
17)                                         new Person("Shiva",38));
18)        //line-1
19)    }
20) }
```



Which code fragment, when inserted at line-1 enables the code to print Durga?

- A. checkAge(iList, ()>p.getAge()>40);
- B. checkAge(iList, Person p->p.getAge()>40);
- C. checkAge(iList, p->p.getAge()>40);
- D. checkAge(iList, (Person p)-> {p.getAge()>40;});

Answer: C



Date and Time API: (Joda-Time API)

Until Java 1.7 version the classes present in Java.util package to handle Date and Time (like Date, Calendar, TimeZone etc) are not up to the mark with respect to convenience and performance.

To overcome this problem in the 1.8 version oracle people introduced Joda-Time API. This API developed by joda.org and available in Java in the form of Java.time package.

program for to display System Date and time.

```
1) import Java.time.*;
2) public class DateTime {
3)     public static void main(String[] args) {
4)         LocalDate date = LocalDate.now();
5)         System.out.println(date);
6)         LocalTime time=LocalTime.now();
7)         System.out.println(time);
8)     }
9) }
```

O/p:

2015-11-23
12:39:26:587

Once we get LocalDate object we can call the following methods on that object to retrieve Day, month and year values separately.

Ex:

```
1) import Java.time.*;
2) class Test {
3)     public static void main(String[] args) {
4)         LocalDate date = LocalDate.now();
5)         System.out.println(date);
6)         int dd = date.getDayOfMonth();
7)         int mm = date.getMonthValue();
8)         int yy = date.getYear();
9)         System.out.println(dd+"..."+mm+"..."+yy);
10)        System.out.printf("\n%d-%d-%d",dd,mm,yy);
11)    }
12) }
```

Once we get LocalTime object we can call the following methods on that object.



Ex:

```
1) import java.time.*;
2) class Test {
3)     public static void main(String[] args) {
4)         LocalTime time = LocalTime.now();
5)         int h = time.getHour();
6)         int m = time.getMinute();
7)         int s = time.getSecond();
8)         int n = time.getNano();
9)         System.out.printf("\n%d:%d:%d:%d",h,m,s,n);
10)    }
11) }
```

If we want to represent both Date and Time then we should go for `LocalDateTime` object.

```
LocalDateTime dt = LocalDateTime.now();
System.out.println(dt);
```

O/p: 2015-11-23T12:57:24.531

We can represent a particular Date and Time by using `LocalDateTime` object as follows.

Ex:

```
LocalDateTime dt1 = LocalDateTime.of(1995, Month.APRIL, 28, 12, 45);
System.out.println(dt1);
```

Ex:

```
LocalDateTime dt1 = LocalDateTime.of(1995, 04, 28, 12, 45);
System.out.println(dt1);
System.out.println("After six months:" + dt1.plusMonths(6));
System.out.println("Before six months:" + dt1.minusMonths(6));
```

To Represent Zone:

`ZonedDateTime` object can be used to represent Zone.

Ex:

```
1) import java.time.*;
2) class ProgramOne {
3)     public static void main(String[] args) {
4)         ZoneId zone = ZoneId.systemDefault();
5)         System.out.println(zone);
6)     }
7) }
```



We can create ZonId for a particular zone as follows

Ex:

```
ZonId la = ZonId.of("America/Los_Angeles");
ZonedDateTimezt = ZonedDateTime.now(la);
System.out.println(zt);
```

Period Object:

Period object can be used to represent quantity of time

Ex:

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1989,06,15);
Period p = Period.between(birthday,today);
System.out.printf("age is %d year %d months %d
days",p.getYears(),p.getMonths(),p.getDays());
```

write a program to check the given year is leap year or not

```
1) import Java.time.*;
2) public class Leapyear {
3)     int n = Integer.parseInt(args[0]);
4)     Year y = Year.of(n);
5)     if(y.isLeap())
6)         System.out.printf("%d is Leap year",n);
7)     else
8)         System.out.printf("%d is not Leap year",n);
9) }
```



Practice Questions for Date and Time API

Case-1:

LocalDate date=LocalDate.of(yyyy,mm,dd);
only valid values we have to take for month,year and day

LocalDate dt=LocalDate.of(2012,01,32);=>invalid
LocalDate dt=LocalDate.of(2012,15,28);=>invalid
LocalDate dt=LocalDate.of(2012,7,28);=>valid

Q1. Given the code fragment:

```
1) import java.time.*;  
2) public class Test  
3) {  
4)     public static void main(String[] args)  
5)     {  
6)         LocalDate dt=LocalDate.of(2012,01,32);  
7)         dt.plusDays(10);  
8)         System.out.println(dt);  
9)     }  
10) }
```

What is the result?

- A. 2012-02-10
- B. 2012-02-11
- C. Compilation Fails
- D. DateTimeException thrown at runtime

Answer: D

RE:

Exception in thread "main" java.time.DateTimeException: Invalid value for DayOfMonth (valid values 1 - 28/31): 32

LocalDate class parse methods:

LocalDate class contains the following 2 parse methods.

1. public static LocalDate parse(CharSequence text)

Obtains an instance of LocalDate from a text string such as 2007-12-03.
The string must represent a valid date and is parsed using DateTimeFormatter.ISO_LOCAL_DATE.



The methods throws `DateTimeParseException` - if the text cannot be parsed

2. `public static LocalDate parse(CharSequence text, DateTimeFormatter formatter)`

Obtains an instance of `LocalDate` from a text string using a specific formatter.

The text is parsed using the formatter, returning a date.

The methods throws `DateTimeParseException` - if the text cannot be parsed

Note: `CharSequence` is an interface and its implemented classes are `String, StringBuffer, StringBuilder` etc

`LocalDate` class `format()` method:

`public String format(DateTimeFormatter formatter)`

Formats this date using the specified formatter.

This date will be passed to the formatter to produce a string.

Q2. Given the code Fragment:

```
1) import java.time.*;
2) import java.time.format.*;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         String date=LocalDate.parse("2014-05-
8)             .format(DateTimeFormatter.ISO_DATE_TIME);
9)         System.out.println(date);
10)    }
```

What is the result?

- A) May 04,2014T00:00:00.000
- B) 2014-05-04T00:00:00.000
- C) 5/4/14T00:00:00.000
- D) An exception is thrown at runtime

Answer: D

Explanation: Here we have only Date value, but we are passing `DateTimeFormatter.ISO_DATE_TIME`.

RE:

Exception in thread "main" `java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: HourOfDay at java.time.LocalDate.get0(Unknown Source)`



Eg:

```
LocalDateTime dt=LocalDateTime.parse("2014-05-04T13:45:45.000");
String s=dt.format(DateTimeFormatter.ISO_DATE_TIME);
System.out.println(s);
```

Output: 2014-05-04T13:45:45

Q3. Given the code fragment:

```
1) import java.time.*;
2) import java.time.format.*;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         LocalDate date1=LocalDate.now();
8)         LocalDate date2=LocalDate.of(2018,4,15);
9)         LocalDate date3=LocalDate.parse("2018-04-15",DateTimeFormatter.ISO_DATE);
10)        System.out.println("date-1:"+date1);
11)        System.out.println("date-2:"+date2);
12)        System.out.println("date-3:"+date3);
13)    }
14) }
```

What is the result?

A.

date-1:2018-04-15
date-2:2018-04-15
date-3:2018-04-15

B.

date-1:04/15/2018
date-2:2018-04-15
date-3:Apr 15,2018

C. Compilation Fails

D. A DateParseException is thrown at runtime

Answer: A



Q4. Given the code fragment:

```
1) import java.time.*;
2) import java.time.format.*;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         LocalDateTime dt=LocalDateTime.of(2014,7,31,1,1);
8)         dt.plusDays(30);
9)         dt.plusMonths(1);
10)        System.out.println(dt.format(DateTimeFormatter.ISO_DATE));
11)    }
12) }
```

What is the result?

- A. 2014-07-31
- B. 07-31-2014
- C. 2014-09-30
- D. An Exception is thrown at runtime

Answer: A

Explanation: LocalDateTime is an immutable date-time object that represents a date-time.

```
dt.plusDays(30);
dt.plusMonths(1);
```

With these new objects will be created and dt is always point to specified date only(2014,7,31,1,1)

Note: LocalDate,LocalTime and LocalDateTime are immutable objects.



Garbage Collections

- 1) Introduction**
- 2) The Ways to Make an Object Eligible for GC**
 - Nullifying the Reference Variable
 - Re Assigning the Reference Variables
 - Objects Created Inside a Method
 - Island of Isolation
- 3) The Ways for requesting JVM to Run GC**
 - By Using System Class
 - By Using Runtime Class
- 4) Finalization**



Introduction

- In Old Languages Like C++, Programmer is Responsible for Both *Creation* and *Destruction* of Useless Objects.
- Usually Programmer taking to Very Much Care while creating Objects and neglecting Destruction of Useless Objects.
- Due to this Negligence, at certain Point for Creation of New Object Sufficient Memory May Not be Available and entire Application will be Down with Memory Problems.
- Hence *OutOfMemoryError* is Very Common Problem in Old Languages Like C++.
- But in Java, Programmer is Responsible Only for Creation of Objects and he is Not Responsible for Destruction of Useless Objects.
- SUN People provided One Assistant which is Always Running in the Background for Destruction of Useless Objects.
- Just because of this Assistant, the Chance of failing Java Program with Memory Problems is Very Less (Robust).
- This Assistant is Nothing but Garbage Collector.
- Hence the Main Objective of Garbage Collector is to Destroy Useless Objects.

The Ways to Make an Object Eligible for GC

- Even though Programmer is Not Responsible to Destroy Useless Objects but it is Highly Recommended to Make an Object Eligible for GC if it is No Longer required.
- An Object is Said to be Eligible for GC if and Only if it doesn't contain any References.

The following are Various Possible Ways to Make an Object Eligible for GC.

1) Nullifying the Reference Variable:

If an Object is No Longer required, then Assign *null* to all its Reference Variables. Then that Object Automatically Eligible for Garbage Collection.

Eg1:

```
Student s1 = new Student();
Student s2 = new Student();
:
:
:
s1 = null; //One Object is Eligible for GC
:
:
s2 = null; //2nd Object is Also Eligible for GC
```

→ No Objects are Eligible for GC

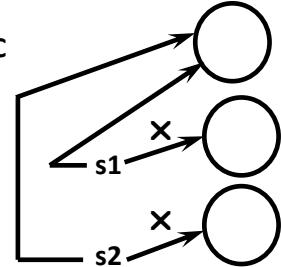


2) Re Assigning the Reference Variables:

If an Object is No Longer to required then Re Assign its Reference Variable to any New Objects, then Old Object is Automatically Eligible for GC.

```
Student s1 = new Student();
Student s2 = new Student();
:
:
:
s1 = new Student(); //One Object is Eligible for GC
:
:
s2 = s1; //2 Object are Eligible for GC
```

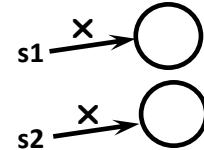
→ No Objects are Eligible for GC



3) Objects Created Inside a Method:

Objects Created Inside a Method are by Default Eligible for GC Once Method Completes (Because the Reference Variables are Local Variables of that Method).

```
class Test {
    public static void main(String[] args) {
        m1(); //After m1() 2 Objects are Eligible for GC
    }
    public static void m1() {
        Student s1 = new Student();
        Student s2 = new Student();
    }
}
```



```
class Test {
    public static void main(String[] args) {
        Student s = m1(); //After m1() Only 1 Object is Eligible for GC
    }
    public static Student m1() {
        Student s1 = new Student();
        Student s2 = new Student();
        return s1; //Here s1 Object Referenced s, s1 and s2 is Not Pointing After m1() Execution
    }
}
```



```
class Test {  
    public static void main(String[] args) {  
        m1(); //After m1() 2 Objects are Eligible for GC  
    }  
    public static Student m1() {  
        Student s1 = new Student();  
        Student s2 = new Student();  
        return s1;  
    }  
}
```

```
class Test {  
    static Student s;  
    public static void main(String[] args) {  
        m1(); //After m1() Only 1 Object is Eligible for GC  
    }  
    public static void m1() {  
        Student s1 = new Student();  
        s = new Student();  
    }  
}
```

4) Island of Isolation:

```
class Test {  
    Test i;  
    public static void main(String[] args) {  
        Test t1 = new Test();  
        Test t2 = new Test();  
        Test t3 = new Test();  
  
        t1.i = t2;  
        t2.i = t3;  
        t3.i = t1;  
  
        t1 = null;  
        t2 = null;  
        t3 = null;  
    }  
}
```

Diagram illustrating the 'Island of Isolation' concept:

On the left, the code shows three objects (t1, t2, t3) each pointing to its own copy of variable 'i'. This is labeled 'No Object is Eligible for GC'.

On the right, a diagram titled 'Island of Isolation' shows three separate circles, each labeled 'i'. Each circle has a self-loop arrow, indicating that each object 'i' is isolated from the others. The objects are labeled t1, t2, and t3.

Below the code, a box contains the text: 'Before t3 = null No Objects are Eligible for GC' and 'After t3 = null All 3 Objects are Eligible for GC'.



Note:

- 1) If an Object doesn't have any Reference then it is Always Eligible for GC.
- 2) Even though Object having Reference Still there May be a Chance of Object Eligible for GC (If All References are Internal References)

Eg: Island of Isolation

The Ways for requesting JVM to Run Garbage Collector:

- Once we Made an Object Eligible for GC if May Not Destroy Immediately by the Garbage Collector.
- Whenever JVM Runs GC then Only Object will be Destroyed. But Exactly at what Time JVM Run GC, we can't Expect. It Depends on JVM Vendor.
- Instead of waiting until JVM Runs GC, we can Request JVM to Run Garbage Collector. But there is No Guarantee whether JVM Accept Our Request OR Not. But Most of the Times JVM Accept Our Request.

The following are Various Ways for requesting JVM to Run Garbage Collector:

1) By Using System Class:

System Class contains a Static Method `gc()` for this Purpose. `System.gc();`

2) By Using Runtime Class:

- A Java Application can Communicate with JVM by using Runtime Object.
- Runtime Class Present in `java.lang` Package and it is a Singleton Class.
- We can Create a Runtime Object by using `getRuntime()`.

Eg: `Runtime r = Runtime.getRuntime();`

Once we got Runtime Object we can Call the following Methods on that Object.

- 1) **freeMemory();** Returns Number of Bytes of Free Memory Present in the Heap.
- 2) **totalMemory();** Returns Total Number of Bytes of Heap (i.e. Heap Size).
- 3) **gc();** Requesting JVM to Run Garbage Collector.



```
import java.util.Date;
class RuntimeDemo {
    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        System.out.println(r.totalMemory()); //16252928
        System.out.println(r.freeMemory()); //15956808

        for(int i=0; i<10000; i++) {
            Date d = new Date();
            d = null;
        }

        System.out.println(r.freeMemory()); //15772752
        r.gc();
        System.out.println(r.freeMemory()); //16074976
    }
}
```

Note: gc() Present in System Class is Static Method whereas gc() Present in Runtime Class is Instance Method.

Which of the following is Valid Way for requesting JVM to Run Garbage Collector?

- 1) System.gc(); ✓
- 2) Runtime.gc(); X Runtime is Static
- 3) new Runtime().gc(); X Runtime is Singleton
- 4) Runtime.getRuntime().gc(); ✓

Note:

- With Respect to Convenience it is Recommended to Use *System.gc()*.
- With Respect to Performance wise it is Recommended to Use *r.gc()*. Because Internally *System.gc()* Calls *r.gc()*.

Finalization:

- Just Before Destroying an Object Garbage Collector Calls *finalize()* to Perform Cleanup Activities.
- Once *finalize()* Completes Automatically GC Destroys that Object.
- *finalize()* Present in Object Class with the following Prototype
protected void finalize() throws Throwable
- Based on Our Requirement we can Override *finalize()* in Our Class to Perform Cleanup Activities.



Case 1:

- Just before Destroying an Object Garbage Collector Always Calls `finalze()` on that Object, then the Corresponding Class `finalize()` will be executed.
- For Example, if String Object Eligible for GC, then String Class `finalize()` will be executed. But Not Test Class `finalize()`.

```
class Test {  
    public static void main(String[] args) {  
        String s = new String("Durga");  
        s = null;  
        System.gc();  
        System.out.println("End of Main");  
  
    }  
    public void finalize() {  
        System.out.println("finalize Method Called");  
    }  
}
```

Output: End of Main

- In the Above Example String Object Eligible for GC and Hence String Class `finalize()` got executed. Which has Empty Implementation. Hence in this Case Output is End of Main.
- If we Replace String Object with Test Object, then Test Class `finalize()` will be executed. in this Case Output is

End of Main
finalize Method Called

OR

finalize Method Called
End of Main

Case 2:

- Based on Our Requirement we can Call `finalize()` Explicitly, then it will be executed Just Like a Normal Method Call and Object won't be Destroyed.
- But before destroying an Object Garbage Collector Always Calls `finalize()`.

```
class Test {  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.finalize();  
        t.finalize();  
        t = null;  
        System.gc();  
        System.out.println("End of main()");  
    }  
    public void finalize() {  
        System.out.println("finalize() Called");  
    }  
}
```

finalize() Called
finalize() Called
finalize() Called
End of main()



- In the Above Example `finalize()` got executed 3 Times.
- 2 Times explicitly by the Programmer and 1 Time by the Garbage Collector.

Note:

- Before destroying Servlet Object, Web Container Always Calls `destroy()` to Perform Cleanup Activities.
- But Based on Our Requirement we can Call `destroy()` from `init()` and `service()` Methods Explicitly, then it will be executed Just Like Normal Method Call and Servlet Object won't be Destroyed.

Case 3:

If the Programmer Calls `finalize()` Explicitly and while executing that `finalize()` if any Exception Occurs and which is Uncaught, then the Program will be terminated Abnormally.

If Garbage Collector Calls `finalize()` and by executing that `finalize()`, if any Exception raised Uncaught then JVM Run Rest of the Program will be executed Normally.

```
class Test {  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.finalize(); → 1  
        t = null;  
        System.gc();  
        System.out.println("End of main()");  
    }  
    public void finalize() {  
        System.out.println("finalize() Called");  
        System.out.println(10/0);  
    }  
}
```

finalize() Called
End of main()

If we are Not Commenting Line 1 then Programmer Calls `finalize()` and while executing that `finalize()` ArithimeticException raised which is Uncaught. Hence the Program will be terminataed Ubnormally by raising ArithimeticException. In this the Output is

finalize() Called
Exception in thread "main" java.lang.ArithmetricException: / by zero

If we Comment Line 1 then Garbage Collector Calls `finalize()` and while executing that `finalize()` ArthematicException raised which is Uncaught. JVM Ignores that Exception and Rest of the Program will be executed Normally.

Which of the following is true?

- JVM Ignores Every Exception which raised while Execution `fnialize()`.
- JVM Ignores Only Uncaught Exception which are raise by executing `finalize()` //true



Case 4:

On any Object Garbage Collector Calls `finalize()` Only Once. Even though that Object Eligible for GC Multiple Times.

```
class FinalizeDemo {  
    static FinalizeDemo s;  
    public static void main(String[] args) throws InterruptedException {  
  
        FinalizeDemo f = new FinalizeDemo();  
  
        System.out.println(f.hashCode());  
        f = null;  
        System.gc();  
        Thread.sleep(5000);  
  
        System.out.println(s.hashCode());  
        s = null;  
        System.gc();  
        Thread.sleep(10000);  
  
        System.out.println("End of main()");  
    }  
  
    public void finalize() {  
        System.out.println("finalize() Called");  
        s = this;  
    }  
}
```

30090737
finalize() Called
30090737
End of main()

In the Above Example even though Object Eligible for GC Multiple Times. But GC Calls `finalize()` Only Once.

Case 5:

We can't Expect Exact Behavior of the Garbage Collector. It is JVM Vendor Dependent. It is varied from JVM to JVM. Hence the following Questions we can't Answer Exactly.

- 1) Exactly at what Time JVM Runs Garbage Collector?
- 2) In which Order Garbage Collector Identifies Eligible Objects?
- 3) In which Order Garbage Collector Destroys the Objects?
- 4) Whether Garbage Collector Destroys All Eligible Objects OR Not?
- 5) What is the Algorithm followed by Garbage Collector. Etc.....



Note:

- Usually whenever the Program Runs with Low Memory JVM will Run Garbage Collector. But we can't Expect Exactly at what Time.
- Most of the Garbage Collectors follow *Mark and Sweep* Algorithm. But it doesn't Means Every Garbage Collector follows the Same Algorithm.

```
class Test {  
    static int count = 0;  
    public static void main(String[] args) {  
        for (int i=0; i<10; i++) {  
            Test t = new Test();  
            t = null;  
        }  
    }  
    public void finalize() {  
        System.out.println("finalize() Called: "+count);  
    }  
}
```

Case 6: Memory Leaks

- The Objects which are Not using in Our Program and which are Not Eligible for Garbage Collection, Such Type of Useless Objects are Called *Memory Leaks*.
- In Our Program if Memory Leaks Present then we will get Runtime Exception Saying OutOfMemoryError.
- To Overcome this Problem if an Object No Longer required, then it is Highly Recommended to Make that Object Eligible for GC.
- In Our Program if Memory Leaks Present, then it is Purely Programmers Mistake.
- The following are Various Memory Management Tools to Identify Memory Leaks in Application
 - HP-J-METER
 - HP-OVO
 - J-PROBE
 - HP-PATROL
 - IBM-TIVOLI



Practice Questions on Garbage Collection

Q1. Given:

```
1) public class MarkList
2) {
3)     int num;
4)     public static void graceMarks(MarkList obj4)
5)     {
6)         obj4.num+=10;
7)     }
8)     public static void main(String[] args)
9)     {
10)        MarkList obj1= new MarkList();
11)        MarkList obj2=obj1;
12)        MarkList obj3=null;
13)        obj2.num=60;
14)        graceMarks(obj2);
15)    }
16) }
```

How many MarkList instances are created in memory at runtime?

- A. 1
- B. 2
- C. 3
- D. 4

Answer: A

Q2. Given the code fragment:

```
1) class Student
2) {
3)     String name;
4)     int age;
5) }
6) And,
7) public class Test
8) {
9)     public static void main(String[] args)
10)    {
11)        Student s1= new Student();
12)        Student s2= new Student();
13)        Student s3= new Student();
14)        s1=s3;
15)        s3=s2;
```



```
16)    s2=null;-->line-1
17)    }
18) }
```

Which statement is true?

- A. After line-1, three objects eligible for garbage collection
- B. After line-1, two objects eligible for garbage collection
- C. After line-1, one object eligible for garbage collection
- D. After line-1, no object eligible for garbage collection

Answer: C