

JAVA

First code in Java:

Class FirstCode

```
{
    for visibility           to receive command line arguments,
    public static void main (String a[])
    {
        System.out.print ("Hello World");
    }
}
```

Two things to do to run java program.

1. javac <file name>
2. java <class name>

- * When you compile the code internally it will create a file which is ".class" file.
- * When you make the class as public then you need to name the Classname same as file name.
- * In one file you can create multiple classes, and when you compile it multiple ".class" files are created i.e. for every class a .class file is created.

.jar - Java Archive : It's a file format based on the popular ZIP file format and is used for aggregating many files into one.

Main Method in Java :

In Java the main() method is the starting point from where compiler starts program execution.

Four things the method will have:

1. name
2. parameter
3. body
4. return type.

Difference between war file and .class file?

- Class file: Source code that is converted by a compiler that is a .class file
- war file: Collection of many .class files + .html files which is compressed we call as .war file
- Jar file: Collection of many .class files which is compressed we call as .jar file.

OOPS (Basic Introduction)

- * It Stands for Object Orientation Principles
- * Object: real time instance like Car, Student, Employee etc.
- * Every Object in real time will have two parts:
 - (a) what it does
 - (b) what it has

Java Code:

Class Car

{

// Has part of an object is represented as a "variable".

String brandName;

int noOfWheels;

// Does - part of an object is represented through "methods"

public void move()

{ // logic of moving a vehicle

}

public void accelerate()

{ // logic of acceleration

?

}

Identifier: * it is a name in java program

* it can be a class name, method name, variable name and label name.

Ex 1.

Class Test

```
{
    public static void main(String [args])
        { int a = 10; }
}
```

Total 5 identifiers.

Ex 2

Class Test

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

Total there are 7 identifiers.

Ex 3

Class Demo

```
{
    public static void main(String [args])
        { String name = "Sachin";
          String result = name.toUpperCase();
          System.out.println(result);
        }
}
```

Rules for writing an identifier:

1. The only allowed characters in java identifiers are:
a to z, A to Z, 0 to 9, -(underline), \$
2. If we use any other characters it would result in error.

3. Identifiers are not allowed to start with digits.

int telusko1 = 100; (valid)

int 1telusko = 100; (invalid)

4. Java identifiers are case sensitive (number != Number)

5. There is no length limit on java identifiers, but still it is a good practice to keep the length not more than 15 characters.

6. We can't use reserved words as identifiers

eg: int if = 10; (compiler error)

* String - inbuilt class name

* Runnable - inbuilt interface name.

* Student - user defined class name

7. Predefined class names can be used as identifiers

eg: * String Runnable = "Sachin";

System.out.println (Runnable); || Sachin.

* int String = 10;

System.out.println (String); || 10

Even though predefined class names can be used as identifiers, it is not a good practice to keep-

* int If = 10; if != IF.

System.out.print (If); valid ✓

* int-int = 10;

System.out.print (int); (E.

Note:

Literal: Any constant value which can be assigned to a variable is called literal.

int data = 10; Literal → 10

data → Variable name, int → datatype

Note: for boolean datatypes the only values allowed for variable is "true" and "false", other than this if we try to keep the values (any) it would result in "Compile Time Error".

Q1. Which of the following list contains only reserve word?

1. final, finally, finalize

Ans: finalize is not a reserve word, it is a method in Object class.

2. break, Continue, exit, return

Ans: exit is not a reserve word, it is a method in System class.

3. byte, Short, Integer, long

Ans: Integer is not a reserved word, it is a predefined class.

4. throw, throws, throws

Ans: throws is not a reserved word, it is a user-defined variable.

Datatypes:

Every variable has a type, every expression has a type and all types are strictly typed / defined in java, because java is strictly type / statically typed language.

Compiler role → Compiler will check the value stored can be handled by datatype or not. This checking which is done by compiler is called "Type checking / Strictly type checking"

Primitive Datatypes: data which is commonly used and supported by any language to store directly.

a) Numeric values: to store number

- whole number, real number

b) Character values: to store character type of data

c) Boolean values: to store logical values.

Number data:

To store whole numbers we have 4 datatypes

- a) byte
- b) short
- c) int
- d) long

Data-type information like:

- a) Size of datatype (how much memory is allocated on the ram for that datatype by Jvm)
- b) min value what it can keep
- c) max value it can keep.

Note:

- * `System.out.println ("Size of byte is :: "+Byte.SIZE);`
- * `System.out.println ("minvalue of byte is :: "+Byte.MIN_VALUE);`
- * `System.out.println ("Maxvalue of byte :: "+Byte.MAX_VALUE);`

a. byte:

Output: Size → 8 bits

min value → -128

max value → 127

eg:

`byte mark = 35 // valid`

`byte mark = 135 // CE: possible loss of precision`

`byte a = "a"; // CE: incompatible types.`

Q2 When to use byte datatype?

It is commonly used when we handle the data which is coming from Stream, network.

Stream → java.io package

" " → means String

' ' → char data

b.

Short :

Size : 16 bits (2 byte)

min value : -32768

max value : +32768

Note: this data is not at all used in java and this datatype is best suited only if you have old processors.

c.

int :

Size : 32 bits (4 byte)

min value : -2147483648

max value : +2147483647.

Note: the most commonly used datatype for storing whole number is "int" and by default if we specify any literal of number, compiler will keep it as "int".

d.

long :

Size : 64 bits (8 bytes)

min value : -2^{64-1} max value : $+2^{64-1}$

Note: when we work with large file, data would come to java programs in terms of GBs, int is not enough so we use long.

* If the data goes beyond the range of int, then keep the data inside the long data type we need to explicitly suffix the data with 'L' or 'l' otherwise it would result in "Compile Time Error".

Real Number

a.

float

Size : 32 bits (4 byte)

min value : 1.4E-45

max value : 3.4028235E38

Note: by default if you specify any real number / dec number compiler will treat it as "double", to specify to treat as float, we need to suffix with "F" or "f".

eg: float a = 10.5 - C.E

float b = 10.5f; { valid ✓ }

float c = 25.5F;

b) double

Size: 64 bits (8 bytes)

min value: 4.9E-324

max value: 1.79...E308

eg: double d = 23.567;

double d = 1.79...57E308;

Note: Datatypes are actually represented to the compiler and
 just using reserve words. reserve words normally is
 "lower case".

To map primitive data as Object in java from Jdk 1.5 concept
 of "Wrapper Class" were introduced

- | | |
|------------------|--------------------|
| 1. byte → Byte | 4. long → Long |
| 2. Short → Short | 5. double → Double |
| 3. int → Integer | 6. float → Float |

Char Data type:

Size: 16 bit (2 Byte)

Java was developed for multilingual languages it adopted
 the Unicode System i.e there are 65,535 characters
 whereas in 'C' char has just 8 bit (1 byte) and it adopted
ASCII System (0 to 128)

eg:

char a = 'A'; **valid**char a = "A"; **invalid** - *Single quote onlychar a = "AB"; **invalid** - no two chars allowed, *only one!

* Char → Character (class)

Type Casting: Changing one type of data to another type.

→ (implicitly) → without any efforts or automatically
 is called "implicit type casting".

eg: int a = 25;

int b = 2;

also called "Numeric type promotion"

float c = a/b; O/p = 12.0 (int → float converted)

* Implicit Typecasting happens in

byte → short → int → long → float → double

left to right is possible but reverse is not possible implicitly.

Explicit type casting : Changing type of data to another type with extra effort is called "explicit typecasting".

Eg: 1. double a = 45.5;

byte b;

b = (byte) a;

System.out.println(b); Output: 45 (Data loss may occur)

2. byte ab = 10;

byte ac = 20;

byte res = ab * ac;

* result in error

Solution:

* (byte) ab * ac;

= * int res = ab * ac;

because resultant will be

an integer by default.

Incrementation

* Pre increment = ++a;

* Post increment → a++;

Eg: int a = 5; int b;

b = a++;

System.out.print(a); - 6

System.out.print(b); - 5

If: b = ++a;

System.out.print(b); - 6

Decrementation

* Pre decrement → --a;

* Post decrement → a--;

* In pre increment or pre decrement the value of variable is first changed then assigned to a variable if assignment operator is there.

Example: int a = 5;
 int b;
 $b = a++ + a++ + ++a; \rightarrow 5 + 6 + 8$
 $\text{System.out.print}(a); \rightarrow 8$
 $\text{System.out.print}(b); \rightarrow 19$

Code Snippets:

1. For the code below what should be the name of java file?

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

Ans: **HelloWorld.java** (Identifiers are case sensitive)

2. public class myfile

```
{ public static void main(String [ ] args)
{
  String arg1 = args[1];
  String arg2 = args[2];
  String arg3 = args[3];
  System.out.println("Arg is " + arg3);
}
```

Which command line arguments should you pass to program
 to obtain the following output: Arg is 2

Ans: **java myfile 1 3 2 2.** \downarrow **arg[3]**, so 4th index = 2.

3. public class Test

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

Does the code compile successfully?
 \downarrow
 Ans: Yes

multiple Semicolons won't do anything.

4. What is the signature of special main method?

Ans: public static void main (String [] a)

5. What will be result of Compiling and executing Test class?

java Test good morning everyone

Private Class Test

```
{ public static void main (String [ ] args)
{ System.out.println (args[1]); }
```

Ans: Compilation error (because of private class)

6. For the Class Test, which options, if used to replace /* INSERT */,
will print "Hurrah I passed...." on to the console?

Public Class Test

```
{ /* INSERT */
{ System.out.println ("Hurrah I passed..."); }
```

Ans: public static void main (String [] args)

static public void main (String [] a)

7. Suppose you have created a java file , "myClass.java".

Which of the following commands will compile java file?

Ans: javac MyClass.java

'for compilation and java for execution'

Operators in Java

1. Arithmetic Operators (+, -, *, /, %)
2. Increment and decremental operators (++ and --)
3. Logical operators (&&, ||, !)
4. Assignment operators (=)
5. Conditional operators (if, else, ternary operator)
6. Relational Operators (==, !=, >, <, >=, <=)
7. Bitwise and shift operators

* Unary Operator: Unary operators require only one operand.
 operations such as : incrementing / decrementing, negating an expression or inverting a boolean.

* Binary Operators: Operators which require atleast two operands to perform operations are called binary op.

* Ternary Operator: Operator which require 3 operands and it is the only conditional operator that takes three operands.

eg: int a=s, b=10;
 int res = (a>b)? a: b; (Syntax (exp1)? a:b)
 System.out.print(res); → 10

Switch Statement : The switch statement execute one statement from multiple conditions like if - else if ladder Statement.

Syntax: Switch (expression) {

Case Value 1 : Code to be executed ; break;

Case Value 2 : Code to be executed ; break;

default : Code to be executed if no case matching }

* break - optional.

Code Snippets:

1. `public class Test {
 public static void main(String[] args)
 { args[1] = "Day!";
 System.out.print(args[0] + " " + args[1]);
 }
}`

→ And the commands : `javac Test.java`

~~`java Test Good`~~

Result?

Ans: Jvm would create a problem during execution as `args[1]` there was no memory, memory was till `args[0]` because of one argument "Good" so .exception!

2. File name: Test.java

`public class Test {
 public static void main(String[] args)
 { System.out.println("Welcome" + args[0] + "!");
 }
}`

→ And the commands are : `javac Test.java`

~~`java Test "James Gosling" "Bill Joy"`~~
What is the result?
~~`args[0]`~~

Ans: Welcome James Gosling! (we use " " to ignore the space).

3. `public class Test {`

`public static void main(String[] args)
 { boolean b1=0;
 boolean b2=1;
 System.out.print(b1+b2);
 }
}`

Ans: Compilation Error.

4. Given:

35. String #name = "Jane Doe";

36. int \$age = 24;

37. Double -height = 123.5;

38. double ~temp = 37.5;

Which two statements are true?

Ans: Line 35 and 38 will not compile.

5. public class Test {

public static void main (String [] args) : public static void main

{ byte b1 = (byte) 127 + 21; } it's correct.

System.out.print(b1); } ↴ 148 can't be stored in byte

}

Ans: -108

so: Jvm will do minrange + (rec - max - 1)

= -128 + (148 - 127 - 1)

= -108 (formula for every datatype)

6. public class Test {

public static void main (String [] args)

{ char c1 = 'a'; // ASCII value of 'a' is 97 }

int ii = c1; Char → int (implicit typecasting)

System.out.print(ii); }

}

Ans: 97

7. public class Test {

public static void main (String [] args)

{ byte b1 = 10;

int ii = b1; byte → int (implicit)

byte b2 = ii; int → byte (not possible, need to tell explicitly)

System.out.print(b1 + ii + b2); }

}

Ans: Line 3 causes compilation error.

8. What will be the output?

```
int x = 100;
```

```
int a = x++; → a=100, x=101
```

```
int b = ++x; → b=102, x=102
```

```
int c = x++; → C=102, x=103.
```

```
int d = (a < b) ? (a < c) ? a : (b < c) ? b : c : x;  
(100 < 102) ? (100 < 102) : 100
```

```
System.out.print(d);
```

Ans: The output will be 100.

9. Class Test {

```
public static void main (String [ ] args)  
{ int a = 100;  
    System.out.print (-a++); }
```

Ans: -100.

Loops in Java: Looping in programming language is a feature which facilitated the execution of set of instructions repeatedly while some condition is true.

Types of loops are: ① for()

② while()

③ do-while()

④ for-each (enhanced for loop)

1. for



Initialization



Condition check



(Yes)



Body of loop



Update

No

→ for (init; cond^{!=}; update)



Body

e.g. for (int i=0; i<n; i++)

{ System.out.print ("*"); }

2. while (boolean condition)
 { loop statements... }

While loop starts with checking boolean condition. If true executed otherwise not. It is also called as "Entry control loop".

3. do while

do {

 Statements }

while (condition);

do while loop is similar to while loop with only difference that is it checks for condition after executing statements, and therefore also called as "Exit control loop".

Ex. 1. int n=4

for (int i=0; i<n; i++)

{ for (int j=0; j<n-1; j++)

{

 if (i==0 || i==3 || j==0 || j==3)

 System.out.print ("*");

 else

 System.out.print (" ");

Output:

 j=0 1 2 3,

 i=0 * * * *

 1 * * *

 2 * * *

 3 * * * *

System.out.println();

}

Ex. 2. int n=10;

for (int i=0; i<n; i++)

{ for (int j=0; j<n; j++)

{ if ((i==0 && j<n-1) || j==0 || (i==n-1 && j<n-1) ||
 (j==n-1 && i>0 && i<n-1))

 System.out.print ("*");

O/P: * * * * * "D".

 else System.out.print (" ");

}

5. `int x = 4;`

`int y = ++(++x);` → Line 2 : Compile Time Error

`System.out.print(x);` (Because incrementation done only on variables not on direct literals).

6. `boolean b = true;`

`b++;` Ans: Line 2 : Compile time error

`System.out.println(b);` because increment/decrement not applicable on boolean type.

7. `int b, c, d;`

`int a = b = c = d = 10;` Ans: Yes, the code will compile.
Will the code compile?

8. `int a = b = c = d = 20;`

Ans: No, because b, c, d not initialized.

`System.out.println(a);`

9. `byte c = (10 > 20) ? 30 : 40;` int

`byte d = (10 < 20) ? 30 : 40;` Ans: 40

`System.out.println(c);` 30

`System.out.println(d);`

10. `int a = 10;` int `b = 20;` - type checking is valid

`byte c = (a > b) ? 30 : 40;` - literals are not involved in operation,

`byte d = (a < b) ? 30 : 40;` So compiler would just check typechecking of result

Ans: Compile Time Error.

11.

`int x = 5 --- 0;`

How many statements are legal?

`int y = --- 50; ^`

'Should not be in beginning/end, okay in b/w.

`int z = 50 ---; ^`

Ans: three statements only

`float f = 123.76 - 86f;`

`double d = 1 - 2 - 3 - 4;`

13. `int x = 10;`

`if (++x < 10 && (x/10) > 10) {`

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

`else`

`System.out.println("Hi");`

Ans: "Hi" is printed.

13.

`int i = 10;`

`int j = 20;`

Ans: 10:30:6

`int k = (j += i) / 5;`

`System.out.println(i + ":" + j + ":" + k);`

14.

`int x = 10;`

`if (x) System.out.println("Hello");`

`if (boolean exp.) = Syntax`

`else System.out.println("Hiee");`

`in this code int is found
so error.`

Ans: Compile Time Error.

15.

`int x = 10;`

`if (x = 20) System.out.println("Hello");`

`else System.out.println("Hiee");`

Ans: Compile time error because of assignment operator.

16.

`boolean b = false;`

`if (b = true) System.out.println("Hello");`

`else System.out.println("Hiee");`

Ans: "Hello" because assignment operator is evaluated on boolean type.

17.

`if (true)`

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

Ans: "Hello" is printed

18. Public class Test {
 public static void main (String [] args) {
 if (true); }
 }
 Ans: No output. (";" is a valid java Statement)

19. Public class Test {
 public static void main (String [] args) {
 if (true)
 int x=10; }
 }
 Ans: Compile time error.

Note: if there is only one statement which needs to be a part of "if" then {} is optional, but statement should not be a declarative statement.

20. if (true){}
 System.out.println ("hello");
 System.out.println ("bye");
 Ans: 1 statement.

OOPS

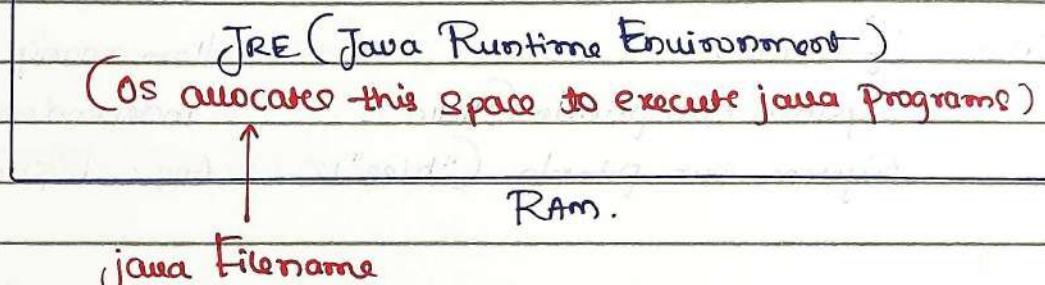
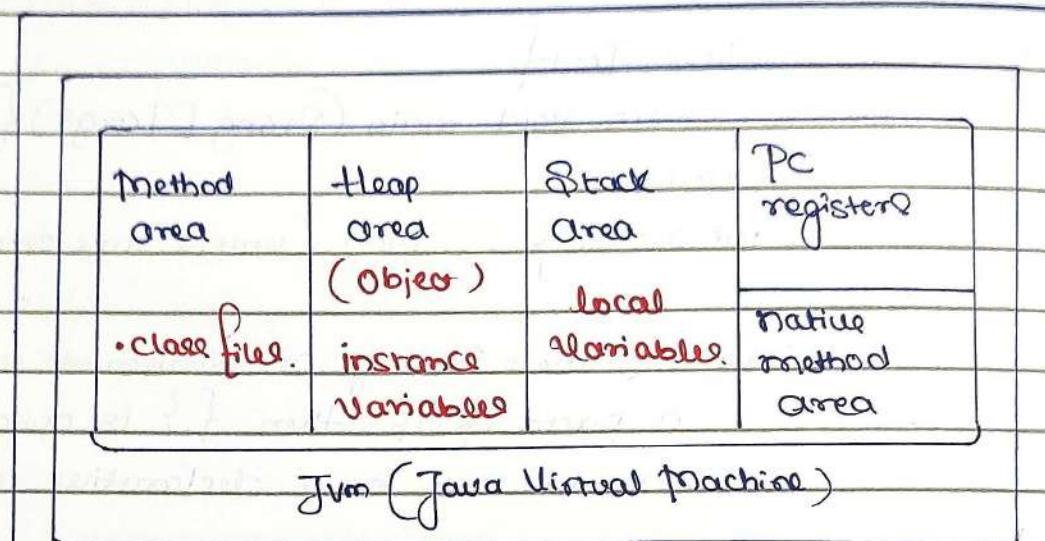
It is actually theory concept, which is implemented by many programming language like C++, java, python.... Any real time problem can be solved if we solve oops principle

To OOPS while solving the problem

1. We need to first mark the objects
2. Every object we mark should have 2 parts:
 - a. HAS-part / attributes (store info. as variables)
 - b. DOES-part / behaviours (represent them as methods)
3. To represent an object, first we need to have a blueprint of an object.
4. We use "new" keyword / reserve word to create an object for blueprint.

5. Every object should always be in constant interaction
 6. Useless object doesn't exist.

JVM architecture:



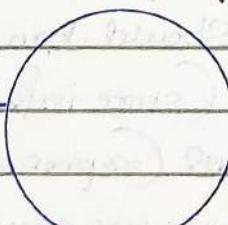
PC registers: address of next instruction which needs to be executed

Native method area: code of other languages which is required for java would be available here.

Student std = new Student();

↓
123457ACEF (HashCode of the Object)

Std



Student

(Heap Area)

1. What is Object?

Physical existence of any element we say as object.

e.g.: book, car, computer...

2. What is "Has-part" and "Does-part" of an object represents?

Has part : indicate what it can hold.

Does part : indicate what it can do

3. What is Blueprint in Java and how to represent it?

In Java to represent a blueprint we have a reserve word called "class".

Conventions followed by Java developer while writing a class is :

a.) Classname should be in "Pascal Convention".

e.g.: InputStream, FileReader...

b.) Variables are represented in "CamelCase".

e.g.: javaFullStack.

c.) methods are represented in "camel case".

e.g.: // Blueprint of Student object

Class Student

{ // HAS-part

 int Sid;

 String name;

 int age;

 char gender;

// DOES-part

 void play() { }

 void study() { }

* To create an object in java
we use "new" keyword.

Syntax:

Classname Variable = new

 classname();

new:

it is a signal to jvm to
Create some space for object in
heap area.

new: It is a signal to Jvm to create some space for the object in the heap area.

- * Tell the className, JVM informs the className, JVM creates the object and sends the "hashcode" to the user.
- * User should collect the hashcode through "ref. variable"

Types of Variables:

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

1. Primitive Variable: primitive Variable used to represent primitive values. eg: int x=10;
2. Reference Variable: reference Variable can be used to refer objects.
eg: Student s = new Student();

Instance Variables: if the variable is declared inside the class, but outside the methods such variables are called as "instance variables".

or

If the value of the variable changes from object to object then such variables are called as "instance variables".

When will the memory for instance variable be given?

Ans: Only when the object is created JVM will create a memory and by default JVM will also assign the default value based on the datatype of variable.

eg: int - 0, float - 0.0f, boolean - false, char - ' ', String - null ...

Note: Scope of the instance variable would be available only when we have reference pointing to the object, if the object reference becomes null, then we can't access "instance variable".

eg. public class Test {

 int i=10;

 public static void main(String[] args)

 { System.out.println(i); } // CE: because instance variable can't be accessed directly in static context.

Test t = new Test(); // Object stored in heap area.

System.out.println(t.i);

t.methodOne(); }

public void methodOne()

{ // Inside instance method instance variable can be directly accessed.

System.out.println(i); } // 10 because it is an instance variable

}

}

Key points about instance variables:

- * If the value of a variable is varied from object to object such type of variable are called instance variable.
- * For every object a separate copy of instance variable 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 same as scope of objects.
- * Instance Variables will be stored on heap as a part of object.
- * Instance Variables will be declared within the class directly but outside of any method or block or constructor.

- * Instance variables can be accessed directly from instance area but cannot be directly accessed from static area.
- * But by object reference we can access instance variables from static area.

Local Variables:

1. Variables which are created inside the stack area are called local variables.
2. During the execution of the method the memory for local variables will be given, and after the execution of the method the memory for variables will be taken out from the stack.
3. Local variables default value will not be given by the JVM, programmer should give the default value.
4. If the programmer doesn't give default value and if he uses the variable inside the method then program would result in "CE".

eg: public class Test

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

```
{     int i=0;
```

```
    for (int j=0; j<3; j++) {
```

```
        i = i+j;
```

```
}
```

```
    System.out.println(j); } } || CE: 'j' not declared
```

eg. 2. Class Test

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

```
{     int a:
```

```
?     System.out.print ("hello"); } } || 'hello' because 'a' not used.
```

eg 3 Class Test

```
{
    public static void main (String [large])
    {
        int x;
        System.out.println(x); } // CE: 'x' not initialised
    }
```

eg 4. Class Test

```
{
    public static void main (String [large])
    {
        int x;
        if (large.length > 0) x=10;
        System.out.println(x); }
    }
```

Keypoints of Local Variables:

- * Sometimes to meet temporary requirements of the programmer we can declare variable inside a method or block or constructor. Such type of variable are called local variables or automatic variables / temporary variable / stack variable.
- * 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 the block execution completes. Hence scope of local variables is exactly same as scope of the block in which we declared.
- * It is highly recommended to perform initialization for the local variable at the time of declaration atleast with default values.

Code Snippets:

1. public class Test

{ public static void main(String args[])

{ int x=10;

switch (x)

{ System.out.println("hello"); }

{ } } ↓ Statement is not part of case label so CTE.

{ }

2. public class Test

{ public static void main(String args[])

{ int x=10;

int y=20;

switch (x)

{ Case 10: System.out.println("hello"); }

break;

Case y: System.out.println("hiee");

break; }

{ }

Ans: Compile time error because case label is
not constant* Label in switch should be "compile time constants",
meaning the value should be known to compiler.

3. In the above code ↑

int x=10;

Ans: "hello".

final int y=20;

switch (x)

{ Case 10: System.out.println("hello"); }

break;

Case y: System.out.println("hiee"); ??

Note: "final" means Compiler will get to know the value and compiler treats it as "Compile Time constant".

4. { int x = 10;
 Switch (x+1)
 { Case 10:
 Case 10+20;
 Case 10+20+30: } }

Ans: No output.

5. byte x = 10;
 Switch (x)
 { Case 10: System.out.println ("Hello"); **Ans: Compile time error because**
 break;
 Case 100: System.out.println ("Free"); **core value outside the range**
 break;
 Case 1000: System.out.println ("Bye");
 break; }

Note: Label value should be within the range of `byte` type
 otherwise it would result in "CE".

6. byte x = 10;
 Switch (x+1) || `byte + int → int`, so `Switch(int)`
 { Case 10: System.out.println ("Hi");
 break;
 Case 100: System.out.println ("Hi"); **Ans: No output.**
 break;
 Case 1000: System.out.println ("No");
 break; }

7.

```

int x = 97;
switch(x)
{
    case 97 : Print("hi"); break;
    case 'a' : Print("100"); break;
}
    
```

' int x='a' x=97'

And case label value can't be duplicated so "CE".

Methods in Java:

A method in Java is a block of code that when called performs specific actions mentioned in it.

Methods have 4 things:

- ① name
- ② input (parameters)
- ③ Body
- ④ return type.

In my program if I have any task, I will write inside method.

Task is present inside a method and if the task has to be executed it has to be brought into stack area. What will be there in Stack area only that part will be executed.

Class Calculator

```

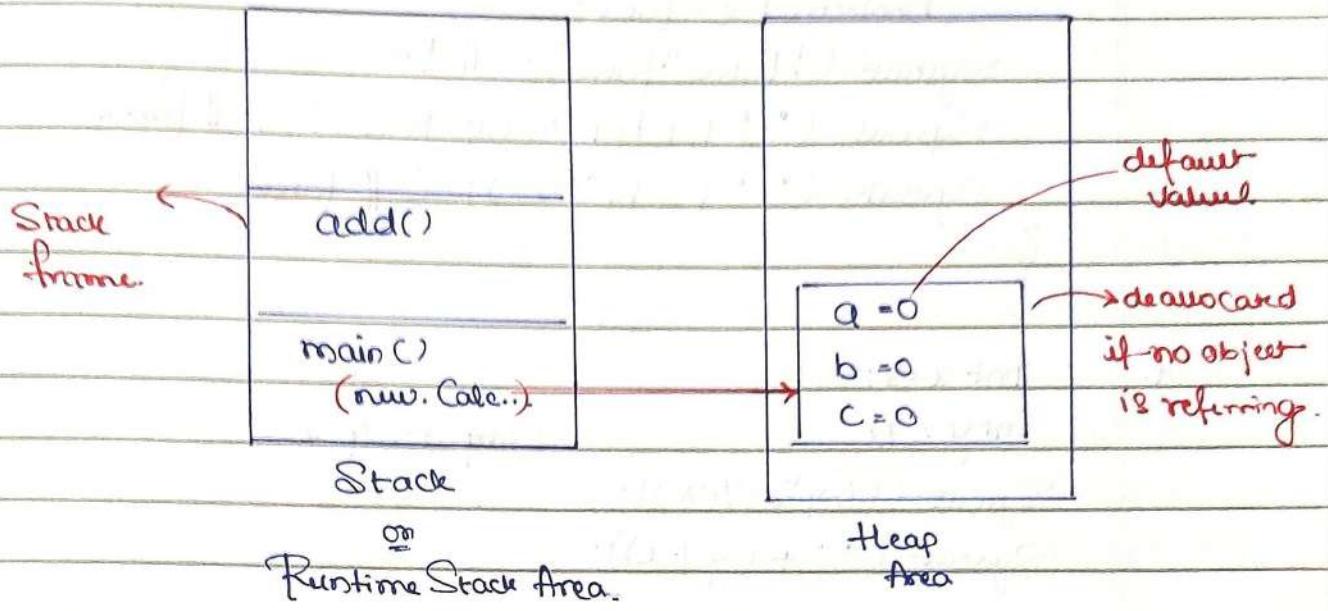
{
    int a,b,c; // Instance var.
    void add()
    {
        a=10;
        b=20;
        c=a+b;
        System.out.println(c);
    }
}
    
```

Public Class Launch

```

{
    public static void main (String []
    {
        Calculator calc = new Calculator();
        calc.add();
    }
}
    
```

Calling method.



Code Snippets:

1. int x = ?

Note: replace x with 0, 1, 2, 3.

Switch(x)

```
{
    default : Print("default");
    Case 0 : Print("0"); break;
    Case 1 : Print("1");
    Case 2 : Print("2");
}
```

x=0, out: 0

x=1 out: $\frac{1}{2}$

x=2 out: 2

x=3 out: default

.

2. Boolean b1=true; //Wrapper Class.

boolean b2=false;

boolean b3=true;

if ((b1&b2) | (b2 & b3) & b3)

Ans: No output is produced

System.out.print("alpha");

if ((b1=false) | (b1&b3) | (b1|b2))

System.out.print("beta");

3. class Maybe

```
{
    public static void main (String [ ] args)
    {
        boolean b1 = true;
    }
}
```

```

boolean b2 = false;
System.out (!false ^ false); // true
System.out (" " + (!b1 & (b2=true))); // false
System.out. (" " + (b1 ^ b2)); } // false
}

```

4.

```

int x=5;
int y=7;
System.out (((y+2) % x));
System.out (" " + (y % x));

```

Output: 4 2

5.

```

Integer i = 42;
String s = (i<40) ? "life" : (i>50) ? "universe" : "everything";
System.out (s);

```

Ans: "everything".

6.

```

{ Integer x=0;
  Integer y=0;
  for (Short z=0; z<5; z++)
    if ((++x>2) || (++y>2)) Ans: 82
      x++;
  System.out (x + " " + y); }

```

Method Overloading

Method Overloading in Java is a feature that allows a class to have more than one method with the same name, but with different parameters.

Java supports method overloading through two mechanisms:

- ① By changing the number of parameters
- ② By changing the datatypes of parameters.

eg: Class Calculator

```
{  
    int add (int a, int b)  
    { return a+b; }
```

```
int add (int a, int b, int c)  
{ return a+b+c; }
```

```
float add (int a, float b)  
{ return a+b; }
```

void main ()

```
{  
    Calculator calc
```

```
= new Calculator();
```

```
int a=10, b=20, c=30;
```

```
float m=5.5;
```

```
Sysout (calc.add (10, 20));
```

```
Sysout (calc.add (10, 20, 30));
```

```
Sysout (calc.add (20, a, m));
```

1. Can we overload main method in Java?

Ans Yes, we can overload main method however Jvm will call such a main method which accept "String [] args" as parameters.

eg: public class LaunchMain

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

```
public static void main (int [] args)  
{ System.out.println ("accepting args"); }
```

Output: "Yes".

Arrays in Java:

* Arrays are index based data structure to store large volume of data using single name (variable name)

* Arrays store homogeneous type of data

* Arrays are treated as objects and allocated on heap.

Syntax :

`int [] a = new int[10];` → 1D Array

'a' is an array of '10' integers.

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

`int [][] arr = new int[3][4];` → 2D Array

* Jagged Array : Array where data is irregular.

eg:	Classes.	Students	{	int [][] arr = new int[3][];
	0	5		arr[0] = new int[5];
	1	3		arr[1] = new int[3];
	2	4.		arr[2] = new int[4]; ?

Code Snippets :

1. `public static void main (String [] args)`
`{ while (true); }`

Ans: Infinite loop with no output

2. `{ while (true)`
`int x = 10; }` Ans: CE because declarative statement
 are not allowed.

3. `while (true)`

`{ int x = 10; }` Ans: Memory for x will be given 4 bytes
 during execution.

4. `while (true) { Sysow ("Hello"); }`

`Sysow ("Hi");`

Ans: Unreachable code
 (Compile time error).

5. while (false)

```
{ System.out("hello"); }
```

```
System.out("hi");
```

Ans: Compile Time Error at line 1.
(unreachable code).

6. { int a=10, b=20;

```
while (a < b)
```

```
{ System.out("hello"); }
```

```
System.out("hi"); }
```

Ans: "Hello" will be printed infinite times

Note: Whenever variables are marked as final, compiler does not do know the value of those variables and it will use the value in the expression to get the result.

7. { final int a=10, b=20;

```
while (a < b)
```

Ans: Compile time error at line n2

```
{ System.out("hello"); }
```

```
System.out("hi"); }
```

8. { do

```
System.out("Hello");
```

Ans: Hello infinite times.

```
while (true); }
```

9. { do

```
while (true); }
```

Ans: Compile Time Error (there should be atleast one statement in loop)

10. { do;

```
while (true); }
```

Ans: No output, Semicolon is an empty statement.

11. { do while (true)

```
System.out("Hello");
```

Ans: Hello infinite times.

```
while (true); }
```

Taking input from Console:

```
Scanner Scan = new Scanner (System.in);
int a = Scan.nextInt();
(To take integer input)
```

Taking input in an array:

```
int[] ar = new int[s];
Scanner Scan = new Scanner (System.in);
for (int i=0; i<n; i++)
{ ar[i] = Scan.nextInt(); }
```

```
for (int i=0; i<s; i++)
{ System.out.println (ar[i]); }
```

Making an array of objects

```
Class Fan
{
    int cost;
    String brand;
    int noOfWings;
```

```
public class Launch
{
    public static void main (String [args])
    {
        Fan [1] f = new Fan[3];
        f[0] = new Fan();
        f[1] = new Fan();
        f[0].brand = "Viwell"; }
```

Disadvantages of Array:

- * It can store only homogeneous type of data.
- * Memory of array is fixed in size, it cannot grow or shrink.
- * Array demands contiguous memory locations.

Enhanced / Advanced "for Loop"

eg. `int arr[] = { 10, 20, 30, 40 };`

```
for (int x : arr)
    { System.out.println(x); }
```

Output: 10
20
30
40

Code Snippets:

1. `do {
 Sysout("hello"); } while (true);` Ans: Compile time error at line 2

```
while (true);  
Sysout("hi");
```

2. `do {
 Sysout("hello"); } while (false);
Sysout("hi");`

Ans: hello
hi

3. `int i=0, j=0;
int i=0, Boolean b= true;
int i=0, int j=0;` How many statements are valid?

Ans: Only 1 statement is valid
(first).

4. `int i=0;
for (Sysout("hello"), i<3; i++) {
 Sysout("hi"); }` Ans: hello
hi (x3)

Note: Syntax of for loop

```
for (Stmt1; Stmt2; Stmt3)
    { Stmt4; }
```

Stmt 1: Can be any statement, but suggested for initialization.

Stmt 2: Compulsorily should be a Boolean statement only.

Stmt 3: Can be any statement, but suggested for inc/dec.

Stmt 4: Can be any statement, suggested for repetitive logic.

5. `int i=0;`

```
for (Sysout("Hello"); i<3; Sysout("hi"))
    { i++; }
```

Ans: hello

hi x3.

6. `for (; ;)`

```
{ Sysout("Hello"); }
```

Ans: "Hello" infinite times

Boolean value / statement will be evaluated as true if empty.

7. `for (int i=0; true; i++)`

```
{ Sysout("Hello"); } Ans: Compile time error at line 2
```

`Sysout("Second");`

8. `for (int i=0; false; i++)`

```
{ Sysout("Hello"); }
```

Ans: Compile time error at line 1.
(code unreachable)

Note: * If a variable marked as final, then those values are known to compiler so we say them as "Compile-time constants".

* If a variable is marked as final, then the value for those variables should never be changed in the program, if we try to do will result in "Compile time error".

* In Java memory for a variable is given by JVM as per its datatype specification and value also will be assigned by JVM only. Compiler will not allocate memory for the variable and it will not initialize the value for the variable.

e.g:

```
final int a=10;
```

```
a++; // CE: Value can't be reassigned.
```

```
System.out(a);
```

Enhanced for loop for 2D arrays.

e.g.

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

```
for (int ar[] : a)
{
    for (int elem : ar)
    {
        System.out.print(elem + " ");
    }
}
```

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

Output: 10 20

30 40 50

60 70 80 90

Limitations of Enhanced for loop:

1. We cannot access a particular element at index and modify it.
2. Traverse iterates only in forward direction.
3. We cannot traverse alternative index or skip certain index.

Syntaxes for arrays:

`int []x; ✓`

`int []x; ✓`

`int [6]ar; ✗`

`x = new int[4]; ✓`

`int x[]; ✓`

`int []ar[]; ✓`

`int []aa, []bb; ✗`

`int []aa, bb[][]; ✓ aa=10, bb=30`

Note :Size can be :

`int []a = new int [s];` → byte, short, int, char ✓
 ↓
 size. → long, float, double, boolean ✗

- * Range in an array is upto the range of "int" not beyond it.
- * There are many methods in Array class like sort, fill, binary-search etc.
- * All these methods are static methods which means we do not need to create objects to call methods, we can call the methods directly.

eg: `Arrays.sort();`

`int arr [] = {70, 80, 90};`

`Arrays.sort(arr);` // Sorts the array.

Code Snippets

1. `int x=0;`

`switch(x){`

`Case 0 : System.out.println("hello"); break; Ans: hello`

`Case 1 : System.out.println("hi"); }`

2. `for (int i=0; i<10; i++)`

`{ if (i==5) break; Ans: 0 1 2 3 4 }`

`System.out.println(i); }`

`}`

3. `int x=10;`

`for { System.out.println("begin");`

`if (x==10) break for;`

`System.out.println("end"); }`

`System.out.println("hello");`

`Ans: begin`

`hello`

4. `int x=10;
if (x==10) break;
System.out("hello");` Ans: Compile time error because break is used outside switch/loop.

5. `int x=2;
for (int i=0; i<10; i++)
{ if (i*x==0) continue;
System.out(i); }` Ans: 1 3 5 7 9

6. `int x=10;
if (x==10) continue;
System.out("hello");` Ans: Compile time error.

7. `int x=0;
switch (x)
{ case 0: System.out("Hello"); continue;
case 1: System.out("hi"); }` Ans: Compile time error
- continue can be only used in loops and labelled blocks.

8. `if(true) System.out("Hello");
else System.out("hi");` Ans: Hello.
Concept of unreachability holds good only for loops.

Binary Search

Binary Search is one of searching techniques applied when input is sorted

public class BinarySearch

```
public static void main(String[] args)
{ int arr[] = {10, 20, 30, 40, 50};
```

```
Scanner Scan = new Scanner(System.in);
System.out.println("Enter the key");
```

```

int key = Scan.nextInt();
int low = 0;
int high = arr.length - 1;
while (low <= high)
{
    int mid = (low + high) / 2;
    if (key == arr[mid])
    {
        System.out.println("Key found");
        break;
    }
    else if (key < arr[mid]) high = mid - 1;
    else if (key > arr[mid]) low = mid + 1;
}
if (low > high) System.out.print("Key found");
    
```

Array Class.

public class AC

```

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

Arrays.fill (a, 5); *// Fills '5' in every index*

Arrays.fill (a, 2, 5, 9); *// Fills '9' from index 2-4*

Arrays.sort (a); *// Sorts the array*

int res = Arrays.binarySearch (arr, 10);

// Searched for '10' in the sorted array & returns index, if not found returns negative index.

}

Code Snippets

1. int $x=0$;

def

$++x$;

Ans: 1 4 8 6 10

Sysout(x);

if ($++x < 5$) continue;

$++x$;

Sysout(x); }

while ($++x < 10$):

2. for (int $i=0$; $i++ < 5$; $i++$) *How many times hello printed?

{ Sysout ("hello");

$i = i++$; }

Ans: 3 times

assignment and increment to same variable, increment
does not have any scope, so increment doesn't happen.

3. int $i=5$;

Sysout ($i++$ " ");

Ans: 5 6 7 16

Sysout (i " ");

Sysout ($++i$ " ");

Sysout ($++i + i++$ " ");

4. int $[7]a = \{0, 2, 4, 1, 3\}$;

for (int $i=0$; $i < a.length$; $i++$)

Ans: 2

$a[i] = a[(a[i]+3)/a.length]$;

Sysout ($a[i]$);

5. int $i=0$; $j=5$;

for (; ($i < 3$) and ($j > 10$) ;)

Ans: 0 6 1 7 2 8 3 8

{ Sysout (" " + i + " " + j); $i++$; }

Sysout (i + " " + j);

Guesser Game Assignment.

```
import java.util.*;  
class Guesser  
{ int guessNum;  
    int guessNum()  
    { Scanner Scan = new Scanner (System.in);  
        System.out.println ("Guess the numbers");  
        guessNum = Scan.nextInt();  
        return guessNum; }  
}
```

Class Player

```
{ int guessNum;  
    int guessNum()  
    { Scanner Scan = new Scanner (System.in);  
        System.out.println ("Player's Guess");  
        guessNum = Scan.nextInt();  
        return guessNum; }  
}
```

Class Vampire

```
{ int numFromGuesser;  
    int numFromPlayer1;  
    int numFromPlayer2;
```

void CollectNumFromGuesser()

```
{ Guesser g = new Guesser();  
    numFromGuesser = g.guessNum(); }
```

void CollectNumFromPlayers()

```
{ Player p1 = new Player();
```

```
Player p2 = new Player();  
numFromPlayer1 = p1.guessNum();  
numFromPlayer2 = p2.guessNum(); }
```

Void compare()

```
{ if (numFromGuesser == numFromPlayer1)  
{ if (numFromGuesser == numFromPlayer1 && 2)  
    System.out.println ("All players won");  
  
else if (numFromGuesser == numFromPlayer2)  
    System.out.println ("Player 1 and 2 won");  
  
else if (numFromGuesser == numFromPlayer3)  
    System.out.println ("Player 1 and 3 won");  
  
else  
    System.out.println ("Player 1 Won"); }  
  
else if (numFromGuesser == numFromPlayer2)  
{ if (numFromGuesser == numFromPlayer3)  
    System.out.println ("Player 2 and 3 won");  
  
else  
    System.out.println ("Player 2 Won"); }  
  
else if (numFromGuesser == numFromPlayer3)  
{ System.out.println ("Player 3 Won"); }}
```

else

```
{
    System.out.println("Game Lost, Try Again!"); }
```

Public Class LaunchGame

{

```
public static void main (String args[ ])
```

{

```
Vampire v = new Vampire();
```

```
v. CollectNumFromGuesser();
```

```
v. CollectNumFromPlayers();
```

```
v. Compare(); }
```

{

String in Java

- * It is a class name, String is basically an immutable class for which object can be created.
- * String refers to collection of characters.

Eg: Syntax:

```
String s1 = "Sachin";
```

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

* String class and
without "new".

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

```

graph TD
    String --> Immutable
    String --> Mutable
    
```

Immutable
(no change)

Mutable (change)

1. StringBuffer

2. StringBuilder (1.5v)

1. String (c)

* In Java String object is by default immutable, meaning once the object is created we cannot change the value of the object, if we try to change then those changes will be reflected on the new object not on the existing object.

Case 1: `String s = "Sachin";`

`s.concat("tendulkar");` || new object of string got created in heap area "Sachintendulkar" so immutable, the new object gets collected by garbage collector.

`System.out.println(s);`

Output: Sachin

VS

`StringBuilder sb = new StringBuilder("Sachin");`

`sb.append("tendulkar");` || on same object modification so mutable.

`System.out.println(sb);`

Output: Sachintendulkar

Case 2:

`String s1 = "Sachin";`

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

`System.out.println(s1 == s2);` || False because they are two different objects.

`System.out.println(s1.equals(s2));` || true.

String class.equals method will compare the content of the object, if returns true if same else false.

V/S

String Builder s1 = new StringBuilder("sachin");

String Builder s2 = new StringBuilder ("sachin");

System.out.println(s1==s2); // False

System.out.println(s1.equals(s2)); // False

String Builder class equals() compare the reference
(Address of object) not the Content of String Builder

Case 3:

String s1 = new String("sachin");

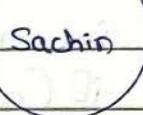
In this case two objects
will be created one in heap
and another one in String
Constant Pool, the reference
will always point to heap.

v/s

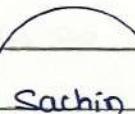
Stack

s1 →

Heap Area



String ConstantPool (SCP)



Garbage
Object
but can't
be collected.

String s="Sachin"

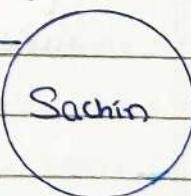
In this case only one
object will be created in
the String Constant Pool and
it will be referred by our
reference.

Stack

s1 →

Heap Area

String ConstantPool (SCP)



Code Snippets

1. int a=7;

boolean res = a++ == 7 && ++a == 9 || a++ == 9;

Sysout ("a = " + a);

Sysout ("res = " + res); Ans: a=9, res=True.

2. if (args.length == 1 | args[1].equals("test"))

Sysout ("test case");

else

Sysout ("production" + args[0]);

Arguments: java Fork 1me2

Ans: Exception thrown at runtime (array index out of bounds).

3. int aVar = 9;

if (aVar++ < 10)

System.out.print (aVar + "Hello World!"); Ans: 10HelloWorld!

else

System.out.print (aVar + "Hello Universe!");

4. int []a = {1, 2, 3, 4, 5};

for (x)

{ System.out.println (a[x]); }

→ What can u replace with x to print 135?

Ans: int e=0; e<5; e+=2;

5. int num=5;

do

{ System.out.println (num-- + ""); }

while (num==0);

Ans: 5

6. `int ii = 0;
int jj = 7;
for (int ii = 0; ii < jj - 1; ii = ii + 2;) { Sysout(ii); }`

Ans: 0,2,4.

7. `arr[0] = 10;
arr[1] = 20;
Sysout(arr[0] + ":" + arr[1]);`

→ Which code fragment should be kept first to print 10:20?
**Ans: int arr;
arr = new int[2];**

8. `boolean opt = true;
switch(opt) { case true: Sysout("tree");
break;
default: Sysout(" ***"); }
Sysout("Done");`

Note: Switch (args.)

**args can be : byte, short, int,
char, String, enum.**

9. `int intArr[] = {15, 30, 45, 60, 75};
int Arr[2] = intArr[4];
int Arr[4] = 90;`

What are values of each element in the array?
Ans: 15 30 75 60 90

10. `int x;
Sysout("Hello");`

Ans: Hello.

11. `int x;
Sysout(x);`

Ans: Compile time error (x is not initialised).

12. `int x;
if (args.length > 0) x = 10;
Sysout(x);`

Ans: Compile time error, because in case if condition fails then what would be the value of x? so JVM will give error.

Strings (Continued)

eg1.

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

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

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

```
String s3 = "dhoni";
```

```
String s4 = "dhoni";
```

```
System.out.println(s3 == s4); // True
```

Note: * Duplicate allowed in Heap Area
and not in SCP

* Memory will be cleaned at
the time of JVM shutdown.

Stack

s1 →

Heap Area

dhoni

s2 →

dhoni

SCP

s3 →

dhoni

s4 →

dhoni

eg2

```
String s = new String("sachin");
```

```
s.concat("tendulkar");
```

```
s = s.concat("IND");
```

```
s = "Sachintendulkar";
```

```
System.out.println(s); // Sachintendulkar
```

Stack

s →

Heap Area

Sachin

X Sachintendulkar

s →

SCP

Sachin IND

Note: Direct Literals are always placed in SCP, because of runtime operation if object is required to create compulsorily that Object should be placed on heap, but not on SCP.

Stack

s →

Heap Area

Sachintendulkar

IND

eg.3
 String s1 = new String ("Sachin");
 s1.concat ("tendulkar");
 s1 += "IND";

Stack.

Heap Area

s1

~~Sachin~~

String s2 = s1.concat ("MI");

System.out.println (s1); // SachinIND

System.out.println (s2); // SachinINDMI

s2 → SachinINDMI
 (Stack) (Heap A)

Note: \rightarrow String + "IND"

Addition if one operand is string
 then it causes Concatenation.

* Total 8 objects created and 2 are eligible
 for garbage collection.

s1

s1

→

SCP

Q1.

String s1 = new String ("you cannot change");

String s2 = new String ("you cannot change");

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

String s3 = "you Cannot Change";

System.out.println (s1 == s3); // false

String s4 = "you Cannot Change";

System.out.println (s3 == s4); // true

final String s8 = "you Cannot";

String s9 = s8 + "change";

System.out.println (s3 == s9);
// true

String s5 = "you Cannot" + "change";

System.out.println (s3 == s5); // true

System.out.println (s6 == s8);

// true

String s6 = "you Cannot";

String s7 = s6 + "change";

System.out.println (s3 == s7); // false

eg.4.

```
String s1 = new String("Sachin");
String s2 = s1.intern();
System.out.println(s1==s2);
```

Stack

 $s_1 \rightarrow$

Heap Area

Sachin

String s3 = "sachin";

System.out.println(s2==s3);

Scp

 $s_2 \rightarrow$ $s_3 \rightarrow$

Sachin

Interning: Using heap object reference, if we want to get corresponding Scp object, then we need to use intern() method.

eg.5

String s1 = new String("Sachin");

String s2 = s1.concat("tendulkar");

Stack

 $s_1 \rightarrow$

Heap Area

Sachin

String s3 = s2.intern();

String s4 = "Sachintendulkar";

System.out.println(s3==s4); // true

 $s_2 \rightarrow$

Sachin

tendulkar

Scp

Note:

Using heap object reference, if we want to get the corresponding Scp object and if object does not exists, their intern() will create a new object in Scp & return it.

 $s_3 \rightarrow$ $s_4 \rightarrow$

Sachin

tendulkar

Sachin

tendulkar

Importance of Scp:

1. In our programs if any String object is required to use repeatedly then it is not recommended to create multiple object with same content it reduces performance of the system and affects memory utilization.

- 2) We can create only one copy and we can reuse the same object for every requirement. This approach improves performance and memory utilization, we can achieve this by using "scp".
- 3) In SCP several references pointing to same object, the main disadvantage in this approach is by using one reference if we are performing any change the remaining references will be impacted. To overcome this problem immutability concept for String Objects were introduced.
- 4) According to this once we create a String Object we can't perform any changes in the existing object if we are trying to perform any change a new String object will be created hence immutability is the main disadvantage of SCP.

String Class Constructor (Commonly used)

String s = new String() → Create an empty String object

String s = new String(String Literals) → Create an object with String Literals on heap

String s = new String(StringBuffer sb) → Create an equivalent String object for string buffer.

String s = new String(char ch[]) → Create an equivalent String object for character array

String s = new String(byte [] b) → Create an equivalent String object for byte array

eg: char [] ch = { 'j', 'a', 'v', 'a' };
 String s1 = new String (ch);
 System.out.println (s1);
 // java

byte [] b = { 65, 66, 67, 68 } ;
 String s2 = new String (b);
 System.out.println (s2);
 // ABCD

Important methods of String

1. `char charAt(int index)`
2. `String concat(String str)`
3. `boolean equals(Object o)`
4. `boolean equalsIgnoreCase(String s)`
5. `String subString(int begin)`
6. `String subString(int begin, int end)`
7. `int length()`
8. `String replace(char old, char new)`
9. `String toLowerCase()`
10. `String toUpperCase()`
11. `String trim()` — removes first and last blank space.
12. `int indexOf(char ch)`
13. `int lastIndexOf(char ch)`

eg: `String s = new String("Sachin");`

`System.out.print(s[3]);` //CE because in java string is

an object and accessing prohibited

`System.out.print(s.charAtIndex(3));` //h

`System.out.print(s.charAtIndex(1));` // String index out-of
bound exception.

Note:

Package <code>java.lang;</code>	Integer Array Class.	<code>s.length()</code>
Class <u>String</u>	<u>Class [I]</u>	<u>method of String class</u>

```
{ public int length()
    {
        { }
    }
}
```

<code>{ int length;</code>	<code>}</code>
----------------------------	----------------

<code>array.length.</code>	<u>property of array class.</u>
----------------------------	---------------------------------

Code Snippets

1. `int n[7] = {{1,3},{2,4}};`
`for (int i = n.length-1; i >= 0; i--)`
`{ for (int y : n[i])` Ans: 2 4 1 3
`System.out(y); }`

2. `int nume1[7] = {1,2,3};`
`int nume2[7] = {1,2,3,4,5};`
`nume2 = nume1;` // Compiler for array assignment Compiler
`for (x : nume2)` will check only the type not length.
`System.out(x + "");` Ans: 1 2 3

3. `int data[] = {2010, 2013, 2014, 2015, 2014};`
`int key = 2014; int count = 0;`
`for (int e : data)`
`{ if (e != key)` Ans: 0 found ✗
`{ continue; count++; }`
`}` Compile time error because
`System.out(count + " found");` "Count++" is unreachable code.

4. `int numbers[];`
`numbers = new int[2];`
`numbers[0] = 10; numbers[1] = 20;`
`numbers = new int[4];` // new object created
`numbers[2] = 30; numbers[3] = 40;` Ans: 0 0 30 40
`for (int x : numbers)`
`System.out(" " + x);`

5. `String days[] = {"sun", "mon", "wed", "sat"};` int wd = 0;
`for (String s : days)`
`{`

Switch (s)

```
{
    Case "sat":
        Case "Sun": wd = 1;
        break;
    Case "mon": wd++;
    Case "wed": wd++ } } Ans : 3.
```

6. String [] str = {"A", "B"}
 int idx = 0;
 for (String s: str)
 { str[idx].concat("element" + idx);
 idx++; }
 for (idx = 0; idx < str.length(); idx++)
 { System.out.length(str[idx]); } Ans : AB.

Strings (Continued)

Q1. String s1 = "sachin"; s1, s3 → "Sachin" (SCP)
 String s2 = s1.toUpperCase(); || s2 → "SACHIN" (Heap Area)
 String s3 = s1.toLowerCase();

System.out.println(s1 == s2); || false

System.out.println(s1 == s3); || true.

Q2. String str = "";
 str.trim();
 System.out.println(str.equals("") + " " + str.isEmpty());
 || false false (because immutable)

Note: Whenever we print any reference, by default JVM will call "toString()" on the reference.

eg: Class Student

```
{ String name = "Sachin";
  int id = 10; }
```

In main {

```
Student std = new Student();
```

```
System.out.println(std); }
```

Out: Student@hexValue.

eg6 final StringBuffer sb = new StringBuffer("Sachin");
 sb.append(" tendulkar");

```
System.out.println(sb); // Sachin-tendulkar
```

Content of StringBuffer can be changed because
it is mutable even with final keyword.

sb = new StringBuffer("Kohli");

↑ sb if final can't be reused, sb can't point to new object,
because it's final

final vs immutability

- * final is a modifier applicable for variables, whereas immutability is only applicable for objects.
- * If reference variable is declared as final, it means we cannot perform re-assignment for the reference variable, it does not mean we cannot change/perform any change in that object.
- * By declaring the reference variable as final, we won't get immutability nature.
- * final and immutability are different concepts.

Note :

final variable ✓	immutable Variable ✗
final object ✗	immutable object ✓

- * String Builder, StringBuffer and all wrapper classes (Byte, Short, Long, Integer, Float, Double, Boolean, Character) are by default mutable.

Important methods of StringBuffer / String Builder

1. int length()
2. int capacity()
3. char charAt(int index)
4. void setCharAt(int index, char ch)
5. StringBuffer append(String s)
6. StringBuffer append(int i)
7. StringBuffer append (long l), [boolean, double, float]
8. StringBuffer append (int index, Object o)
(overloaded)
9. StringBuffer insert (int index, String s) [int, long, double, bool, float]
overloaded
10. StringBuffer delete (int begin, int end);

Constructors of StringBuffer

StringBuffer sb = new StringBuffer();

→ Create an empty StringBuffer object with default initial capacity of 16. Once StringBuffer reaches its maximum capacity a new StringBuffer object will be created
 New Capacity = (Current Capacity + 1) * 2;

Eg:

```
StringBuffer sb = new StringBuffer(19);
```

```
System.out.println(sb.length()); // 10
```

```
System.out.println(sb.capacity()); // 19
```

`StringBuffer sb = new StringBuffer("strings");`

It creates a String buffer object for the given string
with the capacity = s.length() + 16.

Code Snippets:

1. `String []arr = {"A", "B", "C", "D"};`
`for (int i=0; i<arr.length; i++)` Output: A Workdone
`{ System.out.print(arr[i] + " ");`
`if (arr[i].equals("C"))`
 `Continue;`
`System.out.println("Workdone");`
`break;`

2. `String []str = new String[2];`
`int idz = 0;`
`for (String s: str)` Output: Null Pointer exception
`{ str[idz].Concat(" element." + idz);`
 `idz++; }`
`for (idz = 0; idz < str.length; idz++)`
`System.out.print(str[idz]);`

3. `StringBuffer sb = new StringBuffer("java");`
`String s = "java";`
`if (sb.toString().equals(s.toString()));` Output: Match!
`System.out.println("matched");`

```
else if (sb.equals(s))
    System.out("match 2");
```

```
else
```

```
    System.out("No-match");
```

4. `int[] a = new int[] ?` What is the array size?

Ans: Compile time error

5. `int[] a = new int[0]`

Ans: Code compiles fine.

6. `int[] a = new int[-5];` Size of the array?

Ans: Array index out of the bound exception is occurred

7. `long x = 42L, y = 44L;`

`System.out(" " + x + 2 + " ");` || 72 { return "foo"; }

`System.out(foo + x + 5 + " ");` || foo425 (because '+' with String is

`System.out(x + y + foo);` always concatenation)

result in addition and then concatenation $\rightarrow 86\text{foo}$.

String (continued)

* void setLength(int length)

→ It is used to consider only the specified no. of characters and remove all the remaining characters.

eg: `StringBuffer sb = new StringBuffer("sachinramesh");`

`sb.setLength(6);`

`System.out.println(sb);` || sachin

* void trimToSize()

→ this method is used to deallocate the extra allocated free memory such that capacity and size are equal.

eg:

StringBuffer sb = new StringBuffer(1000);

sb.capacity(); // 1000 sb.append("sachin"); // 1000 ~~is~~

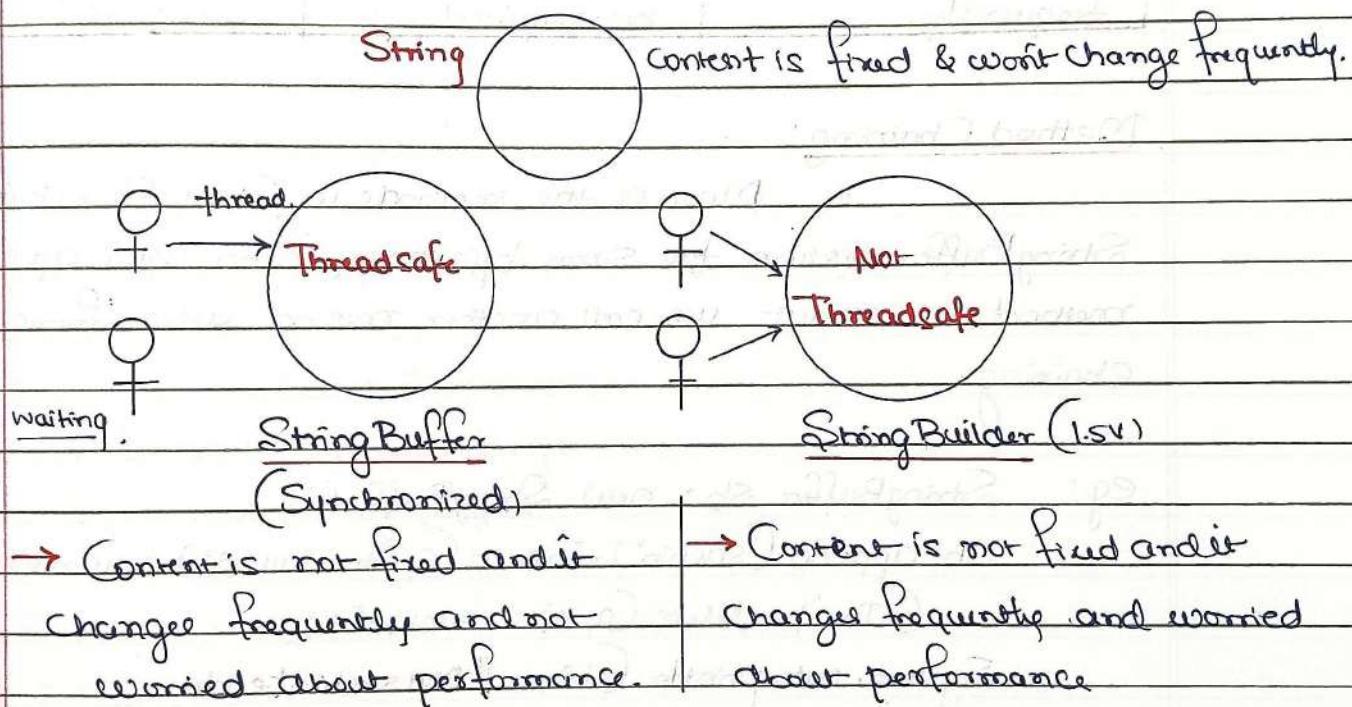
sb.trimToSize();

System.out.print(sb.capacity()); // 6

~~capacity~~,

* void ensureCapacity(int capacity)

→ it is used to increase the capacity dynamically based on our requirement



Every method present in StringBuffer is synchronized, so at a time only one thread are allowed to operate on StringBuffer object, it would create performance problem, to overcome this problem we should go for StringBuilder.

StringBuilder (1.5v): Same as StringBuffer (1.0v) with few differences.

StringBuilder:

- * No methods are Synchronized
- * At one time more than one thread can operate so it is not threadsafe.
- * Threads are not required to wait so performance is high.
- * Introduced in jdk 1.5 version

String	StringBuffer	StringBuilder
We opt if the content is fixed and it won't change frequently.	We opt if content changes frequently but thread safety is not required.	We opt if content changes frequently but threadsafety is not required.

Method Chaining:

Most of the methods in String, StringBuilder, StringBuffer return the same type only, hence after applying method on result we call another method which forms method chaining.

eg: `StringBuffer sb = new StringBuffer();
sb.append("Sachin").insert(6, "tendulkar").reverse().append("IND").delete(0, 4).reverse();
System.out.println(sb); // INDsachintendu`

Program: Copy one String to another.

```
String s1 = "iNeuron", s2 = "";  
for (int i=0; i < s1.length(); i++)  
{ s2 = s2 + charAt(i); }
```

`System.out.println(s2); //iNeuron`

Program : Uppercase a String without inbuilt methods.

```
String s1 = "iNeuron";
String s2 = "";
for (int i=0; i<s1.length(); i++)
{
    s2 = s2 + (char) (s1.charAt(i)+32);
}
System.out.println(s2);
```

Program : Reverse a String without inbuilt methods.

```
String s1 = "iNeuron";
String s2 = "";
for (int i = s1.length(); i>0; i--)
{
    s2 = s2 + s1.charAt(i);
}
System.out.println(s2);
```

Code Snippets:

1. String s = "SACHIN TENDULKAR";
int length = s.trim().length();
System.out.println(length); // 16

2. String s = "HelloWorld";
s.trim();
int i = s.indexOf(" ");
System.out.println(i); Ans: 5

3. String s1 = "Java";
String s2 = new String ("Java");
System.out.equals();

To print equal which code fragment should be added?

Ans : s1.equals(s2) { A }

if (s1.equals(s2))

- if (s1.equalsIgnoreCase(s2)) { B }

else System.out ("not equal");

4. `System.out.println(" " == "");` - true

`Sysout.println(" ");` -

`System.out.println("A" == "A");` - true

`System.out.println("a" == "A");` - $a == A$ (because both in double quotes)

5. `String str = "Java Rocks!";`

`Sysout(str.length() + ":" + str.charAt(0));`

Ans: 11 : !

6. `String s1 = "OCA";`

`String s2 = "oCa";`

`Sysout(s1.equals(s2));` Ans: false

7. `String fname = "James", lname = "Gosling";`

`Sysout(fname = lname);`

Ans: "Gosling" because it is assigned (no E).

8. `String str = "Good";`

`change(str);`

`Sysout(str);`

Ans: Good.

public static void change(String s)

{ s.concat("-morning"); }

9. `StringBuilder sb = new StringBuilder("Good");`

`change(sb);`

`Sysout(sb);`

Ans: Good-morning

void change(String s)

{ s.append("-morning"); }

10. `String str1 = new String("Core");`

`String str2 = new String("CoRe");`

`Sysout(str1 == str2);`

Ans: Core.

11. Public class Test

```
{ public String toString()
{ return "TEST"; }
}
```

Public static void main (String [] args)

```
{ Test obj = new Test();
System.out (obj); }
```

}

Ans: TEST

12. String s1 = "OCAJP";

String s2 = "OCAJP";

System.out (s1 == s2); Ans: true

13. final String fname = "James";

String lname = "Gosling";

String name1 = fname + lname, name2 = fname + "Gosling";

String name3 = "James" + "Gosling";

System.out (name1 == name2); System.out (name2 == name3);

'false'

'true'

String Programming:

Program: Check if a string is a paliindrome or not.

String s1 = "NITIN", s2 = "";

for (int i = s1.length() - 1; i >= 0; i--)

{ s2 = s2 + s1.charAt(i); }

if (s1.equals(s2))

System.out.println("Yes");

else

System.out.println("No");

Program : Check for Anagram between two String

```

String s1 = "Race", s2 = "carE";
s1 = s1.toLowerCase();
s2 = s2.toLowerCase();
char [] ch1 = s1.toCharArray();
char [] ch2 = s2.toCharArray();

Arrays.sort(ch1); Arrays.sort(ch2);
if (Arrays.equals(ch1, ch2))
    System.out.println ("Its Anagram");
else
    System.out.println ("Its Not");

```

Program : Check for Pangram

```

String s1 = "The quick brown fox jumps over a the lazy dog";
s1 = s1.replace (" ", "");
char [] ch = s1.toCharArray();
int [] ar = new int [26];
boolean flag = false;

```

```

for (int i=0; i<ch.length-1; i++)
{
    int index = ch[i]-65;
    ar[index]++;
}

```

```

for (int i=0; i<ar.length; i++)
{
    if (ar[i] == 0)
    {
        System.out.println ("Not pangram");
        flag = true; break;
    }
}

```

```

if (flag == false) System.out.println ("Its pangram");

```

Encapsulation in Java:

- * Data hiding
- * Data binding
- * Providing Security
- * Providing Controlled access to data members.

Code. Class Student

{

```
private int age;           // instance variable / data members
private String name;      // private members can be only accessed
private String City;      inside the class.
```

```
void SetAge (int age)    // Setter
{ this.age = age; }
```

```
int getAge ()            // getter
{ return age; }
```

Note:

→ If a method is doing the activity of setting a value or data to its class variable (receiving data from outside) then it is called "Setter".

```
void SetName (String name)
{ this.name = name; }
```

→ Such a method whose sole purpose is to return its value to whoever calls the method, we called it as "getters".

```
void SetCity (String City)
{ this.City = City; }
```

```
String getCity ()
{ return City; }
```

{}

Public Class LaunchEncap

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

```
{ Student st = new Student();
```

```
st.age = 28; // Compiler error because age is private.
```

```
st.setAge(28); // valid
```

```
int age = st.getAge();
```

```
System.out.println (age); // 28
```

```
st.setName ("Hyder");
```

```
System.out.println (st.getName()); // Hyder }
```

```
}
```

Code Snippets :

```
1. int mask=0, count=0;
if ((5<7) || (7+count<10)) | mask++<10) mask = mask+1;
if ((6>8)^ false) mask = mask+10;
if (! (mask>1)&& 7+count>1) mask = mask+100;
System.out (mask + " " + count);
```

Ans : 2 + 0

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

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

`System.out.print(a); System.out(a[0]); System.out(a[0][0]);`
 ↪ [[I@... ↪ [I@... ↪ _0

4. `int[1][2] a = new int[2][1];`

`System.out(a); System.out(a[0]); System.out(a[0][0]);`
 ↪ [[I@ ↪ null ↪ Null Pointer Exception

5. `int[1][2] a = new int[3][2];` How many objects are created and

`a[0] = new int[3];` how many are eligible for garbage

`a[1] = new int[4];`

`a = new int[4][3];`

collection?

Object created: 11, GC-6

6. `int[1] a;`

`main()`

{ `Test t1 = new Test();`

`System.out(t1.a);` // Null

`System.out(t1.a[0]);` } // Null pointer exception.

7. `int[1] a = new int[3];` // declared at instance level

`System.out.print(Obj.a);` // null

`System.out(Obj.a[0]);` // array index out of bound exception.

8. `int[1] a;` // can't be used without initialized (local variable).

`System.out(a); System.out(a[0]);`

Ans: Compile time error

Note: A class in which all data members are private then
 that class is called "bean"

Encapsulation (Continued)

"this" Keyword: Within an instance method or a constructor, this is a reference to the current object - the object whose method or constructor is being called. You can refer to the any member of current object from within an instance method or a constructor by using this.

Class Student

```
{ private String name;
  private int age;
  private String city;
```

Public Student (String name, int age, String city) // Constructor

```
{ this.name = name;           // Constructor doesn't have return type.
  this.age = age;             // Constructor called when object created
  this.city = city; }         // name is same as class name.
```

```
public String getName()
{ return name; }
```

Note: * Constructor will be there by default, if we create one, Jvm will not create the default one.

```
public int getAge()
{ return age; }
```

* Constructors can have parameters or can be empty, and same arguments should be passed while creating object.

```
public String getCity()
{ return city; }
```

main()

```
{ Student1 std = new Student1 ("Rohit", 17,
  "Hyderabad"); }
```

// Object Created and initialized.

Constructors:

- * Constructors have same name as class name
- * Parameters of the Constructors are similar like method parameters.
- * Constructor is called when object is created or instantiated.
- * It will not have a return type and return statements.
- * Inside Constructors the first statement will be either Super() or this()

Super(): Calls → parent constructor } Constructor

this(): Calls → Constructor of same class } Chaining

- * We can write Super() / this() in first statement only.
- * Constructors can be overloaded → "Constructor overloading"

Purpose: If some statement has to be executed the moment we create an object → Constructor.

this() Method: Inside one constructor if we have a requirement to call another constructor we use "this()". If the constructor we are calling through this() is not available results in error.

"this"	"this()"
<ul style="list-style-type: none"> * It is a keyword * It refers to the current object 	<ul style="list-style-type: none"> It is a method It will call same class constructor.
method	Constructor
<ul style="list-style-type: none"> * Call explicitly by calling name * has return type and Statement 	<ul style="list-style-type: none"> it is called when object is created no explicit return type & Stmt.

Code Snippets.

1. `int [][] a = {{1,2,3}, {3,4}};`
`int [] b = (int []) a[1];`
`Object o1 = a;`
`int [] [] a2 = (int [] []) o1;`
`Sysout (b[1]);` Ans : 4

2. `String [] x;`
`int [] a [] = {{1,2},{1,4}};`
`Object c = new long [4];`
`Object d = x;` || Compilation Succeed.

3. `int x = 5;`

```
main()
{ final Fizz f1 = new Fizz();
  Fizz f2 = new Fizz();
  Fizz f3 = FizzSwitch(f1,f2);
  Sysout (f1==f3 + " " + (f1.x == f3.x));
  Output: true true.
```

Static Fizz FizzSwitch (f1,f2,
f2.x)

```
{ final Fizz z=x;
  z.x=6;
  return z; }
```

4. `int value = 0;`
`boolean setting = true;`
`String title = "Hello";`
`if (value || (setting && title == "Hello")) { return 1; }`
`if (value == 1 & title.equals("Hello")) { return 2; }`

Class A a = new Class A(); Ans : Compilation error because
value is not boolean type

a.getValue();

5. `int a=188, b=15, c=4;`
`Sysout (2 * ((a*5) * (4+(b-3)/(c+2))));` Output : 36

6. `Sysout (23|2.0) || 11.5`
`Sysout (23*2.0) || 1.0`

7. `Sysout ("Hello" + 1 + 2 + 3 + 4);` Output: Hello1234

8. `Sysout (1+2+3+4 + "Hello");` Output: 10Hello

9. `Sysout ("Output is :" + 10 != 5);` Output: Compilation Error

On String objects only '+' operator is allowed other operators would result in compile time error.

10. `Sysout ("Output is :" + (10 != 5));` Ans: Output is : true

11. `int grade = 75;`

`if (Grade >= 60) Sysout ("Congrats");`

`Sysout ("You Passed");`

Output: Compilation error

`else`

`Sysout ("You Failed");`

because an inner class is not

before if and else.

12. `int grade = 60;`

`if (grade = 60)`

Output: Compilation Error because

`Sysout ("you passed");`

of assignment operator.

13. `String arr[] = { ".f.", " *** ", "!!!", "@@@@", "#####", "?" };`

`for (String str : arr)`

{ for (String s : str)

{ Sysout (s);

if (s.length() == 4) break; }

Output: *

break; }

Static Keyword:

Static Variable:

- * Static keyword is used before a variable
- * Memory is allocated during the class loading on heap area.
- * Memory is allocated once in heap
- * One copy of static variable used by all the objects and memory is not allocated all the time
- * Static variables can be called using class name
- * It is called as class variable because one copy is shared by all the objects.
- * They are object independent
- * They can be accessed inside static and non static elements.

Sequence of execution

1. Static Variables
2. Static block
3. Static method
4. Instance Variables
5. Non-static block
6. Constructors
7. methods (if called)

Execution flow of code (default)

1. Static variable → heap → during 'class loading'
2. Static block → to initialise → static variables (during class loading).
3. Static method → main method → other methods (if called)

Note : { 1. Static variables } executed by default in the sequence, Other methods are need to call.
 { 2. Static blocks }
 { 3. main method }

When Object Created: Demo d = new Demo();

- memory allocated for inst. variables
- java block → constructor

eg: **Class Demo**

```
{ Static int a;
  Static int b;
```

Static

```
{ System.out.println ("Static block"); } || Static block
  a=10; b=20; }
```

Static void disp()

```
{ System.out.println ("Static method");
  System.out.println (a);
  System.out.println (b); } || Static method
```

int x,y;

|| instance variables

```
{ x=10; y=20;
  System.out.println ("Non Static Java Block"); }
```

Demo()

```
{ System.out.println ("Constructor"); } || Constructor
```

Void disp1()

```
{ System.out.println ("Non static method"); } || method
  System.out.println (x);
  System.out.println (y); }
```

Output:

Static block

Static method

10 20

Non Static block

Constructor

Non static method

10

20

main()

```
{ Demo.disp();
  Demo d= new Demo();
  d.disp(); d.disp1(); }
```

eg

Class Demo2{
 Static int a;
 Static
 { a=10; }
}

Static void disp()

{ System.out.println("Static Disp "+a); }

}

Public Class Launch{
 Static void disp2()
 { System.out.println("Disp 2"); }
}public static void main (String [] args){ System.out.println("main method");
 disp2(); // directly calling (same class)

Demo2.disp(); // calling without object creation

Demo2 d = new Demo2();

d.disp(); } // calling after creating object.

}

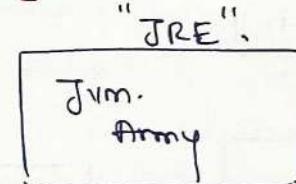
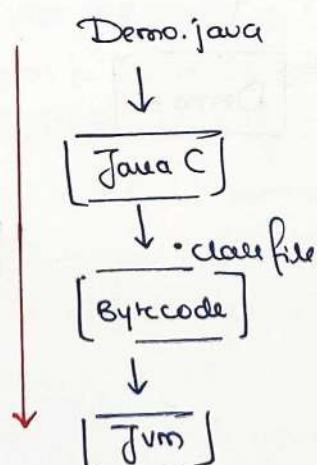
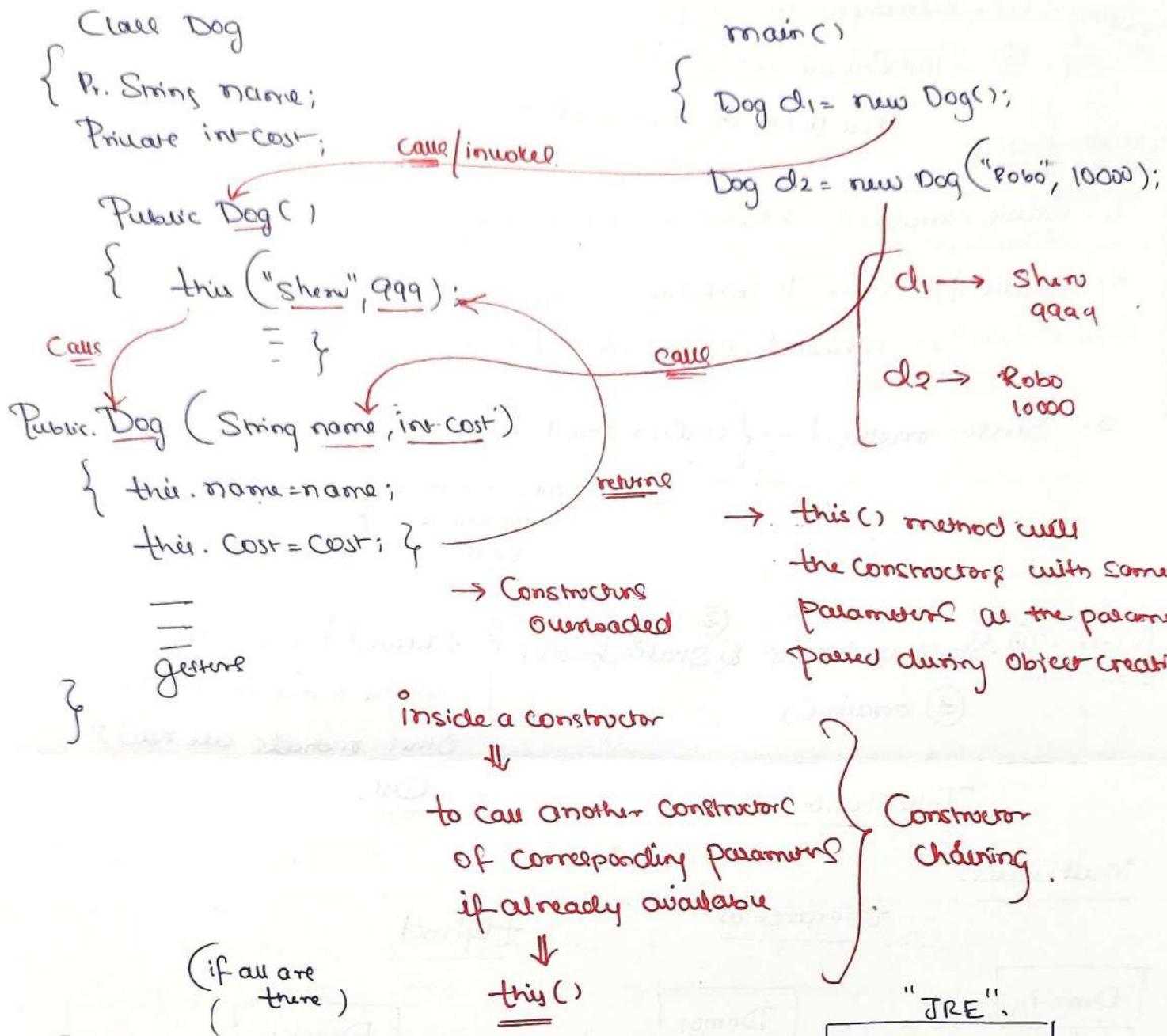
Output: main method

Disp 2

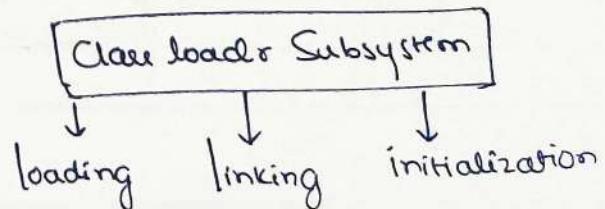
Static Disp 10

Static Disp 10

"this" method example:



- Jvm → ① class loader subsystem
- ② JIT compiler
- ③ execution interpreter, JIT compiler ..



eg : Demo d = new Demo();

Execution { ① - memory for inst. variables
 ② → java block → constructor

Execution flow: ↓ first to execute, then constructor.

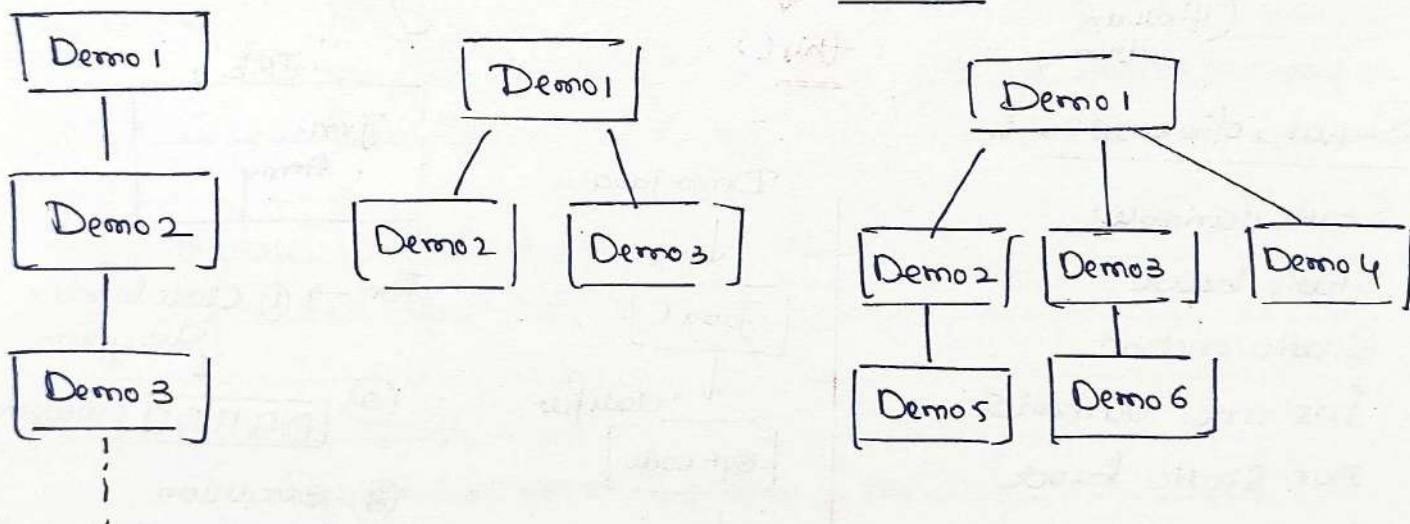
1. Static variables → Heap → class loading
 2. Static block → To initialize → Stat. variable?
executed during Class loading
 3. Static method — { main method (first)
= { Other methods
 you should

Note : ① Static variables & static blocks } ② executed by default -
in the same sequence,
Other methods are need to
③ main ()

Inheritance: Call

Multilevel.

Hierarchical, planar, hybrid



Static methodnon Static method

- * Static method can be called using class name (Generic)
- * Can be called using object reference also.
- * Object independent, need not have to create object.

- * Object creation is mandatory for using non static methods (Specific)
- * Can be called using object reference only
- * Object dependent, need to create object for calling method.

Simple interest Calculator Application:

```
import java.
```

```
class Farmer
```

```
{ private float pa, td, si;  
private float static ri;
```

```
Static
```

```
{ ri = 2.5f; }
```

```
public void acceptInput()
```

```
{ Scanner Scan = new Scanner(System.in);
```

```
System.out.println("Dear, enter loan amount");
```

```
pa = Scan.nextFloat();
```

```
System.out.println("Dear, enter the time needed to repay");
```

```
td = Scan.nextFloat(); }
```

```
public void compute()
```

```
{ si = (pa * td * ri) / 100; }
```

```
public void disp()
```

```
{ System.out.println("Si is " + si); }
```

```
}
```

```

public class LaunchFarmer
{
    public static void main (String [] args)
    {
        Farmer f1 = new Farmer();
        Farmer f2 = new Farmer();

        f1.acceptInput(); f1.compute(); f1.disp();
        f2.acceptInput(); f2.compute(); f2.disp();
    }
}

```

Q1. When to use Static Variables?

Ans Whenever a common copy of data has to be shared among all the objects of a class and are not specific to any object then the data has to be used inside static

Code Snippets :

1. String fruit = new String (new char [] { 'm', 'a', 'n', 'g', 'o' });
switch (fruit)
{
 default: System.out ("Any fruit you do");
 Case "Apple": System.out ("Apple");
 Case "mango": System.out ("mango"); Output: Mango
 Case "Banana": System.out ("Banana"); Output: Banana
 break;
}

2. boolean flag = ! true;
System.out (! flag ? args [0] : args [1]); Output: Am.

3. int a = 3;
System.out (a++ == 3 || --a == 3 || a < 0 || a == 3); // true
System.out (a); // 4

4. `int a = 3;
m(++a, a++);
Sysout(a);` static void m (int i, int j)
 { i++; j--; }
 Output: 5.

5. `boolean flag = false;
Sysout ((flag = true) | (flag = false) || (flag = true));
Sysout (flag);` Output: true
 false

6. `boolean status = true;
Sysout (status = false || status = true | status = false);
Sysout (status);` Ans: Compilation error (because of assign operator).

7. `String msg = "Hello";
boolean [] flag = new boolean [7];
if (flag [6]) msg = "Welcome";
Sysout (msg);` Output: Hello.

8. `boolean flag1 = true, flag2 = false, flag3 = true; flag4 = false;
Sysout (!flag1 == flag2 != flag3 == !flag4); // false
Sysout (flag1 = flag2 != flag3 == !flag4); // true.`

9. `int score = 30; char grade = 'F';
if (50 <= score < 60) grade = 'D'; // Compilation error
- nesting of relational operator is not possible.`

Inheritance in Java

Code example

Class Demo1

```
{ String name = "Hyder";
  int age = 28;
```

```
void disp()
```

```
{ System.out.println ("Demo1 "+age);
}
```

Output:

Demo1 28

(because of inheritance)

Class Demo2 extends Demo1

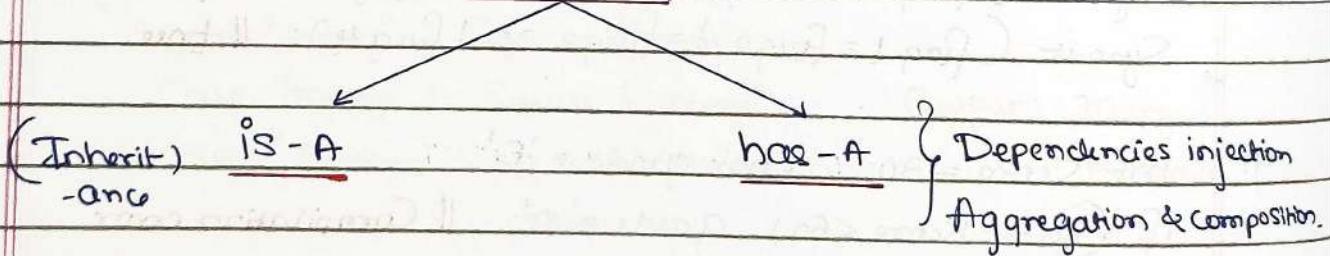
```
{ }
```

Demo2 is child/derived
Demo1 is parent/base
and it extends
"Object class".

Public Class Launch

```
{ public static void main (String [] args)
{
  Demo2 demo = new Demo2 ();
  demo.disp();
}
```

Relationship



Also called:

- ① Parent-child relationship
- ② base class - Sub class
- ③ existing class - derived class.

Unrelated classes → We developed relation (code reusability)
→ through "extend" keyword.

Inheritance Key points:

- * Inheritance is a relationship.
- * Single inheritance is allowed (One class can extend another class).
- * Object class is parent of all class.
- * Multilevel inheritance is allowed.
- * One parent/base class can have any no. of child/subclass. (Hierarchical inheritance allowed).
- * Hybrid inheritance (Hierarchical + multilevel) allowed.
- * Multiple inheritance is not allowed (One class can have only one parent).
- * Cyclic inheritance is not allowed.
- * Private members will not participate in inheritance because to preserve the property of encapsulation.
- * Constructor will not participate in inheritance however parent class constructor will be called because of "super()" method call present inside the child class.

eg:

Class Parent

```
{
    private String name;
    Parent()
    {
        System.out.println("Parent Constructor");
    }
}
```

void disp()

```
{
    System.out.println("Parent");
}
```

}

Class Child

```
{
    // Child()
    {
        // { Super();
    }
}
```

void disp2()

```
{
    // name = hyder;
}
} // private member
// can't access.
```

main()

```
{
    Child c = new Child();
    c.disp();
}
```

Output: Parent Constructor

Parent

eg

Class Parent

```
{ int a, b;
  Parent() { constructor
  {
    a=10;
    b=20;
    Sysout ("Parent Const"); }
```

Parent (int a, int b) { Para

```
  this.a=a;           constr.
  this.b=b;
  Sysout (" Parent para Const"); }
```

main()

```
{ Child ch = new Child();
  ch.disp(); }
```

Class Child extends Parent.

```
{ int x,y;
  Child() { Constructor
  {
    Super(); { adding this will call
    x=100;      parent class. Const.
    y=200; ? }}
```

Child (int x, int y) { param.

```
  this.x=x;
  this.y=y; }
```

void disp()

```
{ Sysout (a); Sysout (b);
  Sysout (x); Sysout (y); }
```

Output: Parent Const

10 20 100 200

Another Variation:

Child (int x, int y)

```
{ Super(x,y); { adding parameters to Super will call the
  this.x=x;   parameterized parent constructor
  this.y=y; } { atleast one constructor (in child class) must have
                Super method else not allowed. }
```

main()

```
{ Child ch = new Child (1000,2000);
  ch.disp(); }
```

Output: Parent para const

1000 2000 1000 2000

Access Specifiers : access specifiers help to restrict the scope of a class, constructor, variable, method, or data member.

There are four access specifiers:

1. public
2. protected
3. default
4. private (strongest)

(visibility)

eg
void display()
{
}

default

	Within a class	Outside class within package	outside package (is-A relationship)	outside package (no is-A relation)
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

eg

class Plane

code {
 public void takeOff()
 {
 System.out.println("Plane taking off");
 }
}

public void fly()
{
 System.out.println("Plane is flying");
}

public void landing()
{
 System.out.println("Plane is landing");
}

Class CargoPlane extends Plane

{
 public void fly()
 {
 System.out.println("Cargo plane flies");
 }
}

public void carryGoods()

{
 System.out.println("Carry goods");
}

Class PassengerPlane extends Plane

{
 public void fly()
 {
 System.out.println("Passenger plane flies");
 }
}

public void carryPassenger()

{
 System.out.println("Passenger plane
 carries passenger");
}

main (String [1 args])

```
{
    CargoPlane cp = new CargoPlane();
    PassengerPlane pp = new PassengerPlane();
    cp.takeOff(); // Inherited method
    cp.carryGoods(); // Specialised method
    cp.fly(); // Overridden method
    cp.landing(); // Inherited method
}
```

```

pp.takeOff();
pp.carryPassengers();
pp.fly();
pp.landing(); }
```

Output:

Plane is taking off.

Cargo plane carries goods
cargo plane flies

Plane is landing

Plane is taking off
Passenger plane carries passengers
Passenger plane flies
plane is landing

Rules to override methods:

1. We Cannot reduce the visibility of overridden method but we can increase it.
2. Return type of the overridden method must be same as that of overriding method in parent.
3. Return type of overridden method in child class can be different than that of parent class if it is co-variant return type (return type: is-A relationship)
4. Parameters of overridden method must be same as that of parent else it will be considered as specialized method
Considering method Overloading concept.

"Super ()" method:

- * Super method will be there inside constructor.
- * It will call the parent class.

Super Keyword: Super keyword is used in Child class to get the values of parent class.

eg: Class Demo1

```
{ int age = 28; }
```

Class Demo2 extends Demo1

```
{ int age = 27; }
```

void disp()

```
{ System.out.println(age); } // 27
```

System.out.println(super.age); } // 28 (parent class value)

}

}

Final Keyword: the final keyword is a non access modifier used for classes, variables and methods which makes them non changeable.

- * Final Variables acts as Constants we cannot change the value.
- * Classes declared as final don't participate in inheritance.
- * Methods declared as final can be inherited but cannot be overridden.

Code Snippets

1. Class Counter

```
{ int count;
private static void inc(Counter counter)
{ counter.count++; }
```

```
Counter C1 = new Counter();
```

```
Counter C2 = C1;
```

```
Counter C3 = null; C2.count = 1000;
inc(C2); }
```

Ans: No. of objects created: 1

2. String res = "";
 loop: for (int i=1; i<=5; i++)
 { switch (i)
 { Case 1: res += "UP"; break;
 Case 2: res += "TO"; break;
 Case 3: break;
 Case 4: res += "DATE"; break loop; }
 }

Sysout (res); // UP TO DATE

Sysout (String.join ("-", res.split (""))); // Up-To-DATE

3. String res = "";
 String [] arr = {"Dog", null, "friendly"};
 for (String s: arr)
 { res += String.join ("-", s); }
 Sysout (res); // Dog- null- friendly.

4. String [][] chs = new String [2] [];
 chs [0] = new String [2];
 chs [1] = new String [5]; int i=97;
 for (int a=0; a<chs.length; a++)
 { for (int b=0; b<chs[a].length; b++)
 { chs [a] [b] = "" + i; i++; }
 }
 for (String [] ca : chs) Output: 97 98
 { for (String c: ca) 99 100 null null null
 { Sysout (c + ""); }
 System.out.println(); }

5. String ta = "A";	ta = ta.concat (tb);	Sysout (ta);
ta = ta.concat ("B");	ta.replace ('c', 'd');	Out: A B C C
String tb = "C";	ta = ta.concat (tb);	

6. String Builder sb = new StringBuilder("B");
 sb.append(sb.append('A'));
 System.out.println(sb); Output: BABAB.

7. Class A

```
{ public String toString()
{ return null; }
}
```

main()

```
{ String text = null;      Output: (null null)
  text = text + new A();
  System.out.println(text); }
```

8.

8. Class SpecialString

```
{ String str;
  SpecialString(String str)
  { this.str = str; }
}
```

for (int i=1; i<=3; i++)

{ switch(i):

{ Case 1: arr[i] = new String("Java");
 break;

Case 2: arr[i] = new String("Java");
 break;

Case 3: arr[i] = new SpecialString("Java");
 break;

for (Object obj: arr)

{ System.out.println(obj); }

main()

```
{ Object[] arr = new Object[4];
  (continued next).
```

Output: null

Java

Java

Special String @ hash code

9. Class ToyStringClass extends String

{ String name; }

Output: Compile time error.

10. String name = "Sachin tendulkar".substring(4); // intendukar

11. String s = "I".repeat(5);
 System.out.println(s); // IIIIII

12. `Sysout ("1".concat("2").repeat(5).charAt(7));` || 2.

13. To which of following class, you can create object `new`?

- a. String
- b. String Builder
- c. String Buffer

14. `String String = "String".replace('i', 'O');`
`Sysout (String.substring(2,5));` Output: rOn.

15. `Sysout ("Java" == new String("Java"));` || False

16. `if ("String".toUpperCase() == "STRING")`
`Sysout (true);`
`else` Output: false.
`Sysout (false);`

17. String, StringBuffer and String Builder - are final classes?
 Ans: Yes

18. `String str1 = "1", str2 = "2"; Str3 = "333";`
`Sysout (str1.concat(str2).concat(str3).repeat(3));`
 Output: 122333122333122333

19. `Sysout ("Java" + 1000 + 2000);` Output: Java10002000

20. `Sysout (1000 + 2000 + "Java");` Output: 3000Java

21. `Sysout (7.7 + 3.3 + "Java" + 3.3);` Output: 11.0 Java 3.3.

22. `String s1 = " "; Sysout (s1.isBlank());` || true (one blank space available)
`Sysout (s1.isEmpty());` || false

Class Parent

```
{ public static void disp()
    {
        System.out.println("Hello parent");
    }
}
```

Class Child {

```
public static void disp()
{
    System.out.println("Hello child");
}
```

- * Static methods participate in inheritance.
- * If you override static methods, it will be treated as specialized methods.

Ques.

```
main()
{
    Parent p = new Child();
    p.disp(); // Hello parent, because child class disp() is a specialized
              // function so you get parent class method
}
```

Child c1 = new Child();

```
c1.disp(); // Hello child, because it is specialized method
}
```

Polymorphism in Java

Class Parents

```
{ public void cry()
  { System.out ("Parents Crying"); }
```

assume
this is
not
there

```
/public void eat()
{ System.out ("Parents eating"); }
}
```

Class Child1 Extends Parents

```
{ public void cry()
  { System.out ("Child1 Crying"); }
```

```
public void eat()
{ System.out ("Child1 eats less"); }
}
```

Output: Child1 crying

Child1 eats less.

Child2 crying

Child2 eats more

Class Child2 Extends Parents

```
{ public void cry()
  { System.out ("Child Crying"); }
```

```
public void eat()
{ System.out ("Child2 eats more"); }
}
```

main() (valid)

```
{ Parents p = new Child1();
  p.cry(); // Child1 crying
```

// p.eat(); // directly using parent

type you cannot call Child class

Specialized methods

```
((Child1)p).eat(); // Child1 eating
```

// downcasting

```
Parents p2 = new Child2();

```

p2.cry(); // Child2 crying

```
((Child2)p2).eat();
```

// Child2 eats more.

Polymorphism:

Polymorphism in java is a concept that allows objects of different classes to be treated as common class. It enables objects to behave differently based on their specific class type.

- * Increases code reusability by allowing objects of different classes to be treated as objects of common class.
- * Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.
- * Improves readability and maintainability of the code.

Upcasting: Upcasting is the type casting of a child object to a parent object.

Downcasting: downcasting means the type casting of a parent object to a child object.

eg: Parent p = new Child();
 ((Child) p).eat();
 ('downcasting' 'calling specialized method')

eg2 Class Plane

```
{ public void takeOff()  

  { Sysout ("Plane taking off"); }
```

```
public void fly()  

{ Sysout ("Plane is flying"); }
```

```
public void landing()  

{ Sysout ("Plane Landing"); }
```

```
Class CargoPlane extends Plane  

{ public void fly()  

  { Sysout ("Cargo Plane flied"); }}
```

```
Class PassengerPlane ext. Plane  

{ public void fly()  

  { Sysout ("Passenger Plane  

  flied"); }}
```

Class FighterPlane extends Plane

```
{ public void fly()  

  { Sysout ("Fighter plane flied"); }}
```

Class Airport

```
{ public void permit //take  

  (Plane plane) plane  

  { plane.takeOff(); cargo and  

  plane.landing(); close all  

  plane.fly(); } work }
```

main()

```
{ CargoPlane cp = new CargoPlane();  

  FighterPlane fp = new FighterPlane();  

  PassengerPlane pp = new PassPlane();  

  Airport a = new Airport();  

  a.permit(cp);  

  a.permit(fp);  

  a.permit(pp); }
```

Abstraction in Java:

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Abstract Keyword:

- Abstract keyword cannot be applied to a variable
- Abstract keyword can be applied to a method and the method should not have a body.
- Abstract keyword is added to a class when there is atleast one abstract method.

eg

Abstract Class Loan

```
{
    abstract public void dispInt();
    public void welcomeNote()
    {
        System.out.println("Welcome to xyz Bank");
    }
}
```

Class HomeLoan extends Loan

```
{
    public void dispInt()
    {
        System.out.println("RI is 12.1");
    }
}
```

Class EducationLoan extends Loan

```
{
    public void dispInt()
    {
        System.out.println("RI is 10.1");
    }
}
```

main()

```
{
    // Loan l = new Loan(); // we Cannot Create object of
    // Abstract class.
```

Loan l1 = new HomeLoan(); // we can create reference of abstract class.

l1.dispInt(); // RI is 12%.

l1.welcomeInt(); // Welcome to xyz Bank

Loan l2 = new EducationLoan();

l2.dispInt(); // RI is 10%.

l2.welcomeNote(); // Welcome to xyz Bank

}

Abstract Key points:

- * We Cannot Create Abstract Class object.
- * Abstract Class Can have all methods abstract. (100% abstraction).
- * Abstract Class can have both abstract and concrete methods.
- * Abstract Class can have all methods concrete.
- * The subclass / child class if extending abstract class then either have to implement abstract method (or) declare class abstract.
- * Constructor cannot be made abstract.
- * Abstract → incomplete Class
 - Child Class will implement } inheritance
- * We Cannot make Abstract Class as final, doing such is illegal.
- * We Cannot make Abstract method as final → illegal.
- * Variable Cannot be made abstract.
- * We can have Constructor in abstract class.

Area Calculator Program

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

Abstract Class Shapes

{ float area;

Abstract public void input();

Abstract public void compute();

```
public void disp()
{
    System.out.length ("The area is " + area);
}
```

Class Rectangle extends Shape

```
{ float length, breadth;
public void input()
{
    Scanner Scan = new Scanner (System.in);
    System.out.println ("Enter length of rectangle");
    length = Scan.nextInt();
}
```

```
System.out.println ("Enter breadth of rectangle");
breadth = Scan.nextFloat(); }
```

public void Compute()

```
{ area = length * breadth; }
}
```

Class Square extends Shape

```
{ float length;
public void input()
{
    Scanner Scan = new Scanner (System.in);
    System.out.println ("Enter length of square");
    length = Scan.nextFloat(); }
```

public void Compute()

```
{ area = length * length; }
}
```

Class Circle extends Shape

```
{ float radius;
final float pi = 3.14f;
```

```
public void input()
{
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter radius of circle");
    radius = scan.nextDouble();
}
```

```
public void Compute()
{
    area = pi * radius * radius;
}
```

Class Geometry

```
{ void Permit(Shapes s)
{
    s.input();
    s.compute();
    s.disp();
}}
```

public Class Launch Project

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

```
    Rectangle r = new Rectangle();
    Square ss = new Square();
    Circle c = new Circle();
```

```
    Geometry g = new Geometry();
```

```
    g.Permit(r);
    g.Permit(ss);
    g.Permit(c);
}
```

Code Snippets

1. `for (int i=5; i>=1; i--)
{ System.out.print("*".repeat(i)); }`

**
*

2. `boolean opt = true;
switch(opt)
{ case true: System.out.println("True"); break;
default: System.out.println("False");
System.out.println("Done"); }`

What modification should be enabled to print "True Done"?
: Replace first line with
String s="True", case with
"True";

3. `StringBuilder sb = new StringBuilder(s);
String s = "";
if (sb.equals(s)) { System.out.println("match1"); }
else if (sb.toString().equals(s.toString())) { System.out.println("match2"); }
else System.out.println("match3"); Output: Match2`

4. `String text = "RISE";
text = text + (text = "ABOVE"); Output: RISE ABOVE
System.out.println(text);`

5. `StringBuilder sb = new StringBuilder("Java");
String s1 = sb.toString();
String s2 = "Java";
System.out.println(s1 == s2); Output: False`

6. `StringBuilder sb = new StringBuilder("Java");
String s1 = sb.toString();
String s2 = sb.toString();
System.out.println(s1 == s2); Output: False`

7. String str = "Java";
 String Builder sb = new String Builder ("Java");
 Sysout (str.equals (sb) + ":" + sb.equals (str));
 Output: false false

8. String Builder sb = new String Builder ();
 Sysout (sb.append (null).length()); Compilation Error

9. String Builder sb = new String Builder (s);
 sb.append ("0123456789");
 sb.delete (8, 1000); Sysout (sb); Output: 01234567

10. String Builder sb = new String Builder ("Hurray!");
 sb.delete (0, 1000); Sysout (sb); Output: o

Dependency Injection:

The process of injecting dependant object into target object is called as "Dependency Injection".

We can achieve dependency injection in 2 ways

- a) Constructor dependency injection
- b) Setter dependency injection

Constructor dependency injection: Injecting dependant object into target through a constructor.

Setter dependency injection: injecting dependent object into target object through a Setter.

eg Class Address || Dependency Object

{ }
Class Student || Target object
{ }
Address address; }
}
}

Relationships in Java

As part of Java application development, we have to use entities as per application requirements. In java application development, if we want to provide optimizations over memory utilization, code reusability, execution time, sharability then we have to define relationships between entities.

There are three types of relationships between entities

1. HAS-A relationship (extensively used in projects)
2. IS-A relationship
3. USE-A relationship

Difference between HAS-A and IS-A relationship:

Has-A relationship will define associations between entities in java applications, here associations between entities will improve communication between entities and data navigation between entities.

IS-A relationship is able to define inheritance between entity classes, it will improve "Code Reusability" in java applications.

Associations in Java

1. One to One Association (1:1)
2. One to many association (1:m)
3. Many to one association (m:1)
4. Many to many association (m:m)

To achieve associations between entities, we have to declare either single reference or array of reference variables of an entity class in another entity class.

eg

Class Address {

.....

}

Class Account {

.....

}

Class Employee {

.....

Address [] addr; // it will establish 1:m association

Account account; // 1:1 association

}

1. One to one association: it is a relationship between entities, where one instance of an entity should be mapped with exactly one instance of another entity.

eg: Every employee should have only one account

Account.java:

package in.inuron.bean;

// Dependent object

```
public class Account
{
    String accNo;
    String accName;
    String accType;
}
```

```
public Account (String accNo, String accName, String accType)
{
    super();
    this.accNo = accNo;
    this.accName = accName;
    this.accType = accType;
}
```

Employee.java:

Package: in.inuron.bean;

|| Target Object

```
public class Employee
{
    private String eid;
    private String ename;
    private String eaddr;
```

Account account; || has-a relationship

public Employee (String eid, String ename, String eaddr,
Account account) || Constructor injection

```
{
    Super();
    this.eid = eid;
    this.ename = ename;
    this.eaddr = eaddr;
    this.account = account;
}
```

```

public void getEmployeeDetails()
{
    System.out ("Employee details are:");
    System.out ("Empid : " + eid);
    System.out ("Empname : " + ename);
    System.out ("Emp address : " + eaddr);
    System.out.println();
    System.out ("Account details are");
    System.out ("Account no : " + account.accNo);
    System.out ("Account name : " + account.accName);
    System.out ("Account type : " + account.accType);
}

```

TestApp.java :

```

package in.inuron.main;
import in.inuron.bean.Account;
import in.inuron.bean.Employee;

```

```
public class TestApp {
```

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

```
        Account account = new Account ("ABC123", "Sachin", "Savings");
```

// Constructor injection

```
        Employee employee = new Employee ("Ino 10", "Sachin", "MI",
                                         account);
```

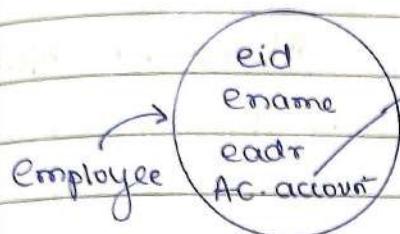
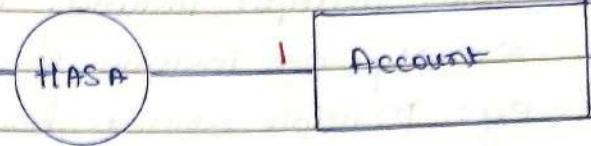
```
        employee.getEmployeeDetails(); }
```

```
}
```

Target Object

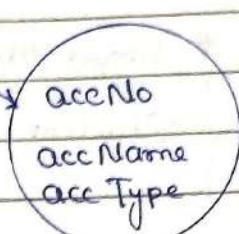


Dependent object



Dependency injection

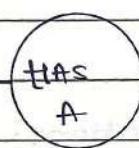
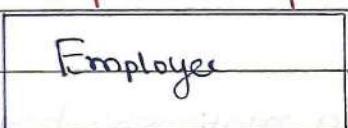
1. Constructor
2. getter and setter.



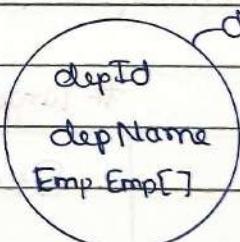
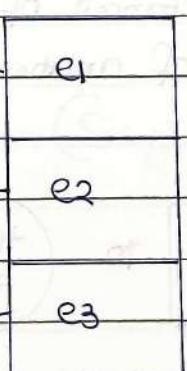
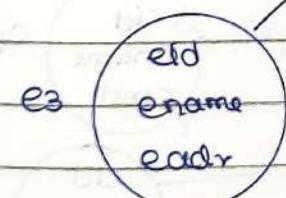
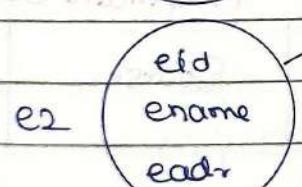
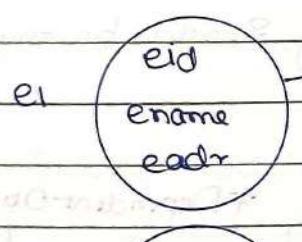
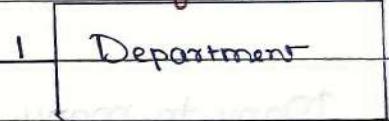
② One to many association: it is a relationship between entity classes, where one instance of an entity should be mapped with multiple instances of another entity.

eg: Single department has multiple employee.

Dependent Object



Target Object

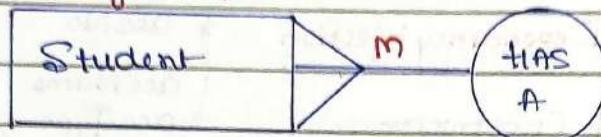


3.

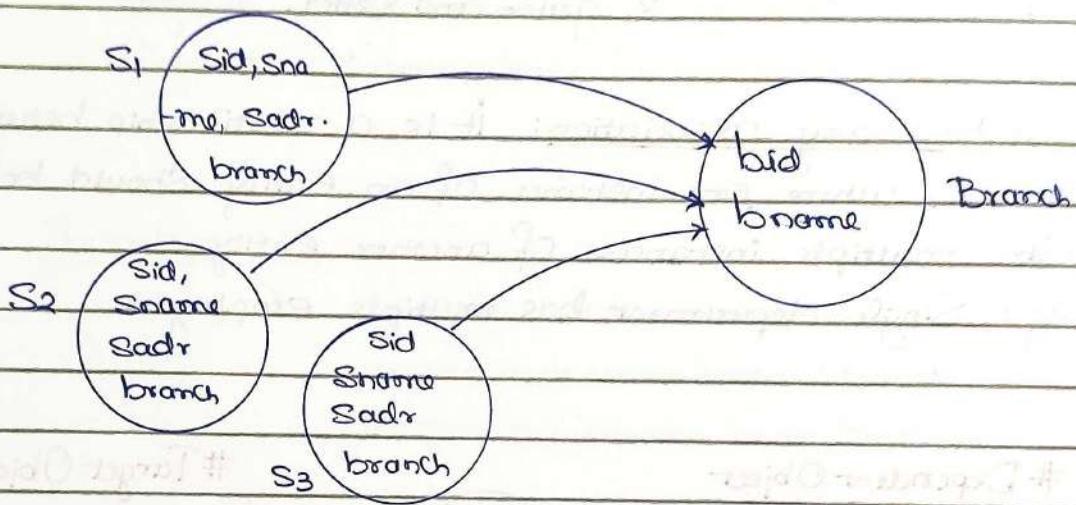
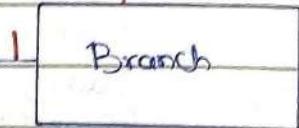
Many to One Association: It is a relationship between entities, where multiple instances of an entity should be mapped with exactly one instance of another entity.

e.g.: Multiple Students have joined with a single branch

Target Object



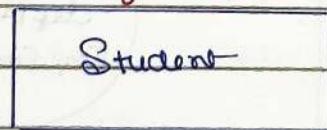
Dependent Object



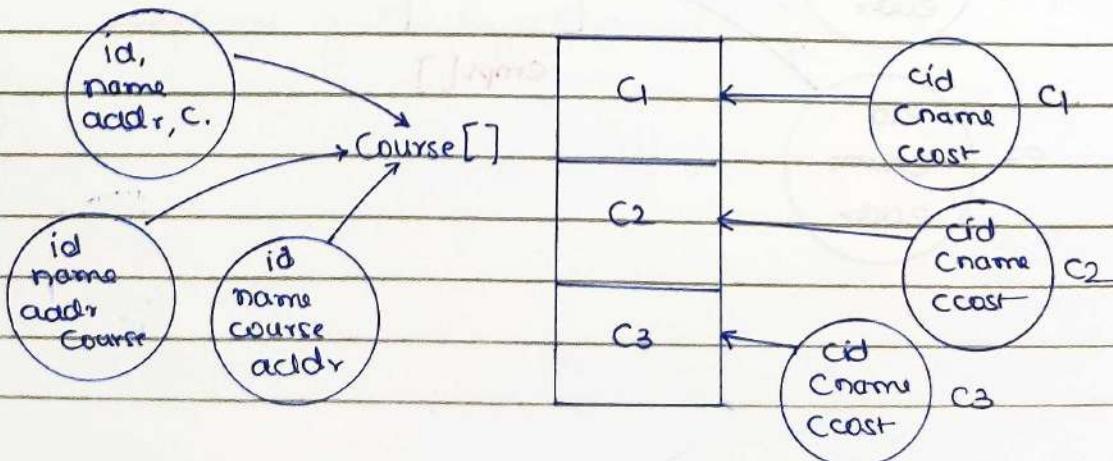
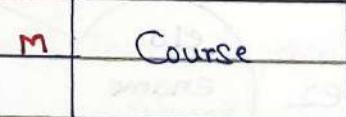
4.

Many to many associations: It is a relationship between entities, where multiple instances of an entity should be mapped with multiple instances of another entity.

Target Object



Dependent Object



Inner Class in Java

In java you can define a class within another class. Such a class is called nested class / inner class. Inner classes are divided into two categories : static and non-static.

Nested classes that are declared static nested are called static inner class and only inner class can be declared static.

Code: Class A

```
{ public void show()
    { System.out.println ("in Show"); }
```

Class B

|| inner class

|| object of inner class can be created in A.

```
{ public void Config()
    { System.out.println ("in Config"); }
```

}

public class FirstCode

```
{ public static void main (String[] args)
    {
        A obj = new A();
        obj.show(); // in Show
```

A.B obj1; // Creating object of inner class

obj1 = new A.B(); // Creating object of inner class (syntax only for static class)

obj1 = Obj.new B(); // Creating inner class object from A obj
obj1.config(); } // in config

}

Code 2.Class Computer

```
{
    public void config()
    {
        System.out.println("In Computer config");
    }
}
```

public class SecondCode

```
{
    public static void main (String [] args)
    {
        Computer obj = new Computer();

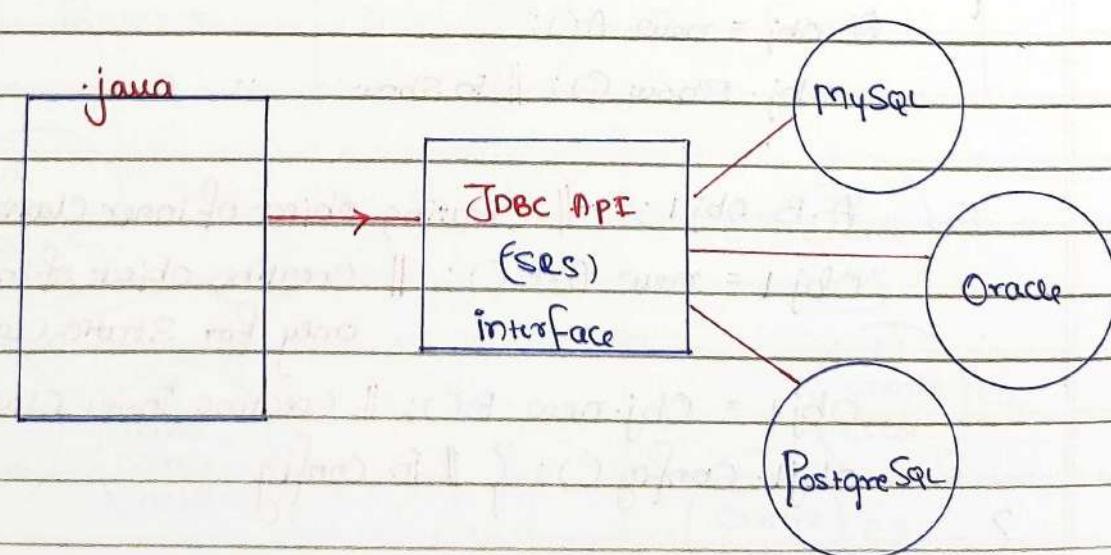
        inner class {
            {
                public void config() // Overriding Computer method
                {
                    System.out.println ("Something new");
                }
            };
        }

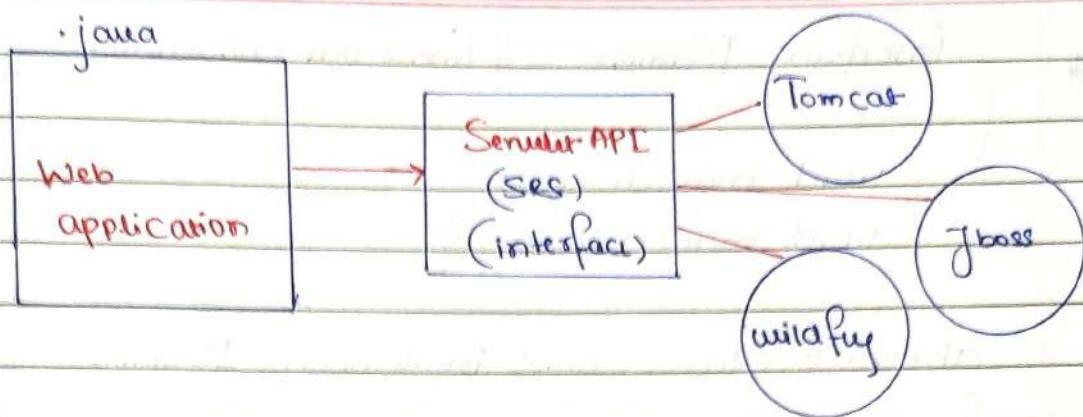
        obj.config(); // Something new.
    }
}
```

Note: There three classes are available and three .class files are generated.

Interface in Java

Def 1: Any service requirement specification (SRS) is called interface.



Note:

* Inside interface every method is always abstract whether we are declaring or not, hence interface is considered as 100% pure abstract class.

eg: interface Account

{

// 100% Abstract Class,

// methods are abstract and public by default

void withdraw();

void deposit(); }

Summary: Interface corresponds to Service requirement specification or Contract b/w Client and Service provider or 100% Abstract Class.

Declaration and Implementation of Interface.

- ① Whenever we are implementing an interface compulsorily for every method of that interface we should provide implementation. Otherwise we have to declare the class as Abstract Class in that case Child class is responsible to provide implementation for remaining methods.
- ② While implementing an interface method, it should be declared public. Otherwise results in Compile Time Error.

eg interface ISample // follow this convention

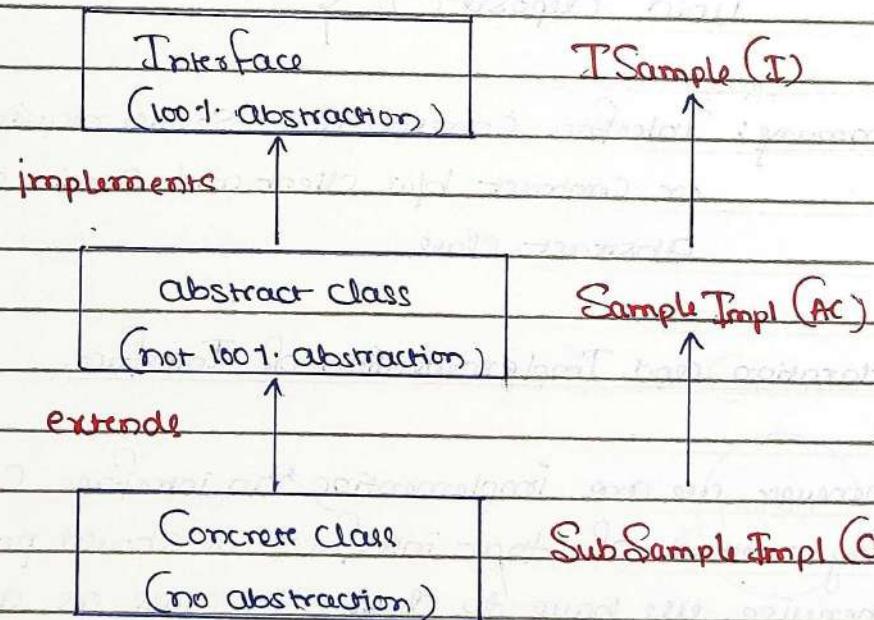
```
{
    void methodOne();
    void methodTwo();
}
```

abstract class Sample implements ISample

```
{
    public void methodOne() // method defined, but 2nd not defined
    { ... }
}
So class is abstract.
```

Class SubServiceProvider extends Sample

```
{
    @Override // indication to Compiler (methods are overridden)
    public void methodTwo() { ... } // method 2 defined
}
```



SampleImpl ref = new SubSampleImpl(); not good approach
 TSample ref = new SubSampleImpl(); good approach

Code Snippets

1. String Str = " ";

Sysout (Str.isBlank()); || true

Sysout (str.isEmpty()); || false

2 Which class of "String" have delete() and reverse() method?

Ans StringBuilder and StringBuffer.

3 java.lang.String class has append() method? Ans: No

4 String Str1 = "Java";

Sysout (Str2 == Str3); || false

String Str2 = Str1.intern();

Sysout (Str3 == Str1); || false

String Str3 = new String ("Java");

Sysout (Str1 == Str2) || true

5 String Str1 = "Java";

(Str1 == Str2); || true

String Str2 = Str1.intern();

Sysout (Str1.equals(Str2)); || true

6 StringBuffer Sb1 = new StringBuffer ("1111");

StringBuffer Sb2 = Sb1.append (2222).append.reverse();

Sysout (Sb1 == Sb2); || false

7. StringBuilder and StringBuffer class has intern() method? No

8. Sysout (String.join (",", "1", "2", "3").concat (",").repeat (3).lastIndexOf (",",));

17

9. StringBuffer sb = new StringBuffer ("1111");

Sysout (sb.insert (3, false).insert (5, 33).insert (7, "one"));

Output: 111fa3. One31se1.

Interfaces (continued)

Difference between extends and implements

extends: One class can extend only one class at a time

eg: Class One { }

Class Two extends One { }

implements: One class can implement any number of interfaces at a time.

eg: Case 1

interface IOne

{ public void methodOne(); }

interface ITwo

{ public void methodTwo(); }

Class Demo implements IOne, ITwo

{ public void methodOne() { }; }

public void methodTwo() { }; }

Case 2: Class can extend as well as can implement an interface

1. reusability : → extends

2. interface : → implementation

Class Sample

{ public void m1() { } }

interface IDemo

{ void m2(); }

Class DemoImpl extends Sample implements IDemo

{ public void m2() { } }

Case 3 : An interface can extend any no. of interfaces at a time.

e.g.:

interface One
{ public void m1(); }

interface Two
{ public void m2(); }

interface Three extends One, Two
{ public void m1();
public void m2();
public void m3(); }

Q1. Which of the following is true?

- A class can extend any no. of class at a time.
- An interface can extend only one interface at a time.
- A class can implement only One " " " "
- A class can extend a class and implement an interface but not simultaneously.
- An interface can implements any no of interface at a time
- None of the above.

Q2. Consider the expression X extends Y which of the possibility of X and Y expression is true?

- Both X and Y should be classed
" " " " interface
- " " " Can be classes or interfaces
- No restriction

Interface Methods:

Every method present inside the interface is public and abstract.

Valid declarations are:

1. void m1();
2. public void m1();
3. abstract void m1();
4. public abstract void m1();

Why?

Public : → to make the method available for every implementation class.

Abstract : → implementation class is responsible for providing the implementation.

eg: JDBC API (java.sql.*)

Implementation given by : → MySQL, Oracle, Postgre

* Method allowed in interface is : public, abstract (2)

we can't use : static, private, protected, strictfp, synchronized, native, final.

Interface Variables:

* Inside the interface we can define variable.

* Inside the interface variables define requirement level constants.

* Every variable inside interface is "public static final" by default.

Why?

Public : → to make it avail. for implementation class object

Static : → to access it without using implementation class name

Final : → implementation class can access value without any modification.

- Note:
- * Since variable defined inside interface is public, static and final we cannot use modifiers like private, protected, transient, volatile.
 - * Since the variable is static and final, compulsorily it should be initialized at the time of compilation/declaration. Otherwise results in CE.

Interface 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.

eg:

```
interface Left { public void m1(); }
interface Right { public void m1(); }
```

Class Test implements Left, Right {

 @Override

```
    public void m1() { ... }
```

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 act as overload methods.

eg:

```
interface Left { public void m1(); }
interface Right { public void m1(int i); }
```

Class Test implements Left, Right

{ @Override

```
    public void m1() { ... }
```

```
    public void m1(int i) { ... } }
```

Q1 Can a java class implements two interface Simultaneously?

Ans. Yes, possible but in both the interface method signature should be same, but not different return types

Case 3 : If two interface contain a method with same signature but different return types then it is not possible to implement both interface simultaneously.

eg:

interface Left { public void m1(); }

interface Right { public int m1(); }

Class Test implements Left, Right {

@ Override

public void m1() { ... }

@ Override

public int m1() { ... } || Invalid, So can't implement both simultaneously,

}

Q2 Can java class implements two interface simultaneously?

Ans Yes possible, except if two interface contain a method with same signature but different return types.

Variable Naming Conflicts :

Two interface can contains a variable with same name and there may be a chance of variable naming conflicts but we can resolve variable naming conflict by using interface names.

eg: interface Left { int x=10; }

interface Right { int x=20; }

Output: 10 20

public class Test implements Left, Right {

public static void main (String [] args)

{ System.out (Left.x); System.out (Right.x); }

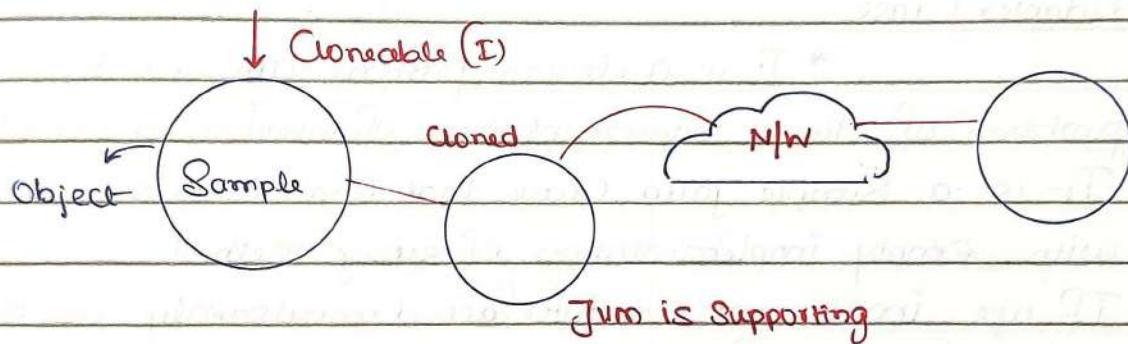
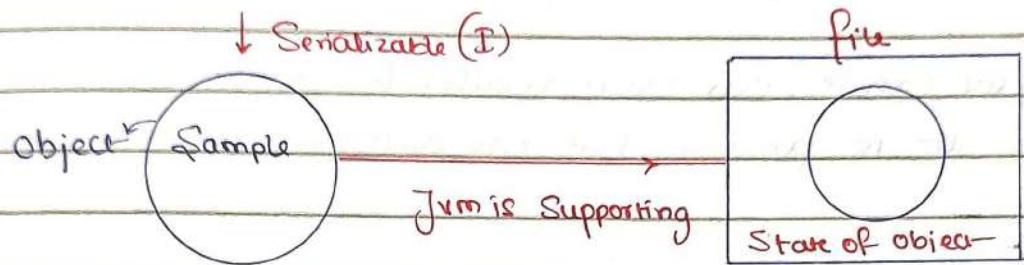
Note: We can also write an interface without any variable or abstract methods.

eg * `interface Serializable { }`

Class Sample implements Serializable { ... }

* `interface Cloneable { }`

Class Sample implements Cloneable { ... }



Marker Interface:

If an interface does not contain any method and by implementing that interface if our Object will get some ability such type of interface are called "Marker Interface" / "Tag Interface" / "Ability Interface".

eg:

1. Serializable: by implementing Serializable interface we can send that object across the network and we can save state of an object into the file

2. SingleThreadModel: by implementing it interface Servlet can process only one client request at a time so we can get "Thread Safety".

3. Cloneable: by implementing it our object is in a position to provide exactly duplicated clone object.

Q3 Without any method in marker interface how objects will get ability?

Ans Jvm is responsible to provide required ability.

Q4 Why Jvm is providing the required ability?

Ans To reduce the complexity of programming.

Q5 Can we create our own marker interface?

Ans Yes, it is possible but we need to customize Jvm.

Adapter Class:

- * It is a design pattern allowed to solve the problem of direct implementation of interface methods.
- * It is a simple java class that implements an interface only with empty implementation of every method.
- * If we implement an interface compulsorily we should give the body for all the methods whether it is required or not. This approach increases length of code and reduces readability.

eg: interface X

```
{ void m1();  
void m2();  
void m3(); }
```

Abstract class Sample Adapter X implements X

```
{ public void m1(){ } // giving dummy implementation  
public void m2(){ }}
```

```
public void m3() { }
```

Class Test extends Adapter

```
{ public void m3() { } } giving implementation only for required
{ System.out.print("I am from m3"); } method.
```

eg : Server (I)

↓ implements

Generic Server (Ac)

↓ extends

Http Server (Ac)

↓ extends

My Server (Class)

Interface

Abstract Class

- * If we don't know anything about implementation and just have requirement specification, we should go for interface.

- * If we are talking about implementation but not completely then we should go for abstract class.

- * Every method present inside the interface is always public and abstract (default).

- * Every method present inside abstract class need not to be public and abstract.

- * We can declare methods as private, protected, final, static, synchronized, native, static.

- * No restriction on abstract class method modifiers.

- * Every variable : public, static, final.

- * Need not to be public static final.

* Can't declare Variable as private, protected, transient, volatile.	No restriction on access modifier.
* Every Variable Should be initialized at time of declaration.	No need to initialize Variables at time of declaration.
* We can't write static and instance block.	Static and instance block allowed.
* Can't write Constructor.	We can write Constructor.

Note:

Static block: - class file loading happens and used to initialize static variables.

instance block: during the creation of an object, just before the constructor call used for initialization of instance variables.

Constructor: during the creation of an object, used for initialization of instance variables.

Q6 What is the need of abstract class? Can we create an object class and does it contains constructor?

Ans Abstract class object cannot be created because it is abstract. But constructor is needed for constructor to initialize the object.

Q7 Why abstract class can contain constructor whereas interface cannot?

Abstract Class:

- * it is used to perform initialization of object
- * to provide value for the instance variable
- * to contain instance variable which are req for child object to perform initialization for those instance variable.

interface:

- * Every variable is always public, static and final so there is no chance of instance variable inside interface.
- So we should initialize at time of declaration
- * So constructor is not required

eg: abstract class Person

```
{
    String name;
    int age, height, weight;
```

Person (String name, int age, int height, int weight)

```
{
    this.name = name;
    this.age = age;
    this.height = height;
    this.weight = weight;
}
```

Class Student extends Person {

int rollNo;

int marks;

Student (String name, int age, int height, int weight, int rollNo, int marks)

```
{
    Super (name, age, height, weight, rollNo);
    this.rollNo = rollNo;
    this.marks = marks;
}
```

}

Q8. Can reference be created for abstract class?

Person p = new Student ("Sachin", 49, 5.6f, 71, 10, 100);

Q9. Can reference be created for interface?

ISample Sample = null;

Note: Every method present inside interface is abstract, but in abstract class also we take only abstract methods then what is need of interface concept?

Abstract Class Sample {

 public abstract void m1();

 public abstract void m2(); }

interface ISample {

 void m1();

 void m2(); }

→ We can replace interface concept with abstract class, but it is not a good programming practise.

eg: 1.

interface X

{ ... }

Class Test implements X

{ ... }

Test t = new Test();

eg: 2

abstract X

{ ... }

Class Test extends X

{ ... }

Test t = new Test();

* performance is high

* while implementing X we can extend one more class, through which we can bring reusability

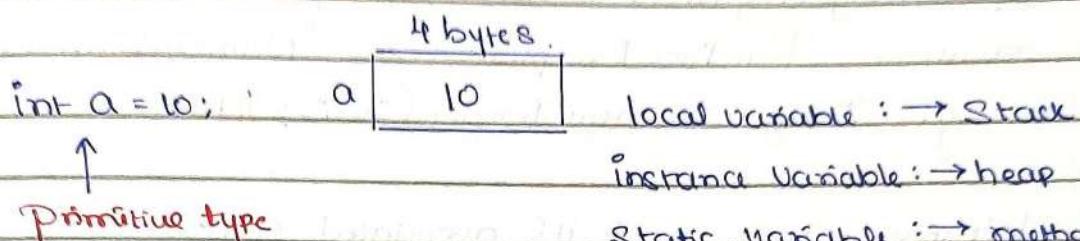
* performance is low

* while extending X we can't extend any other classes so reusability is not brought.

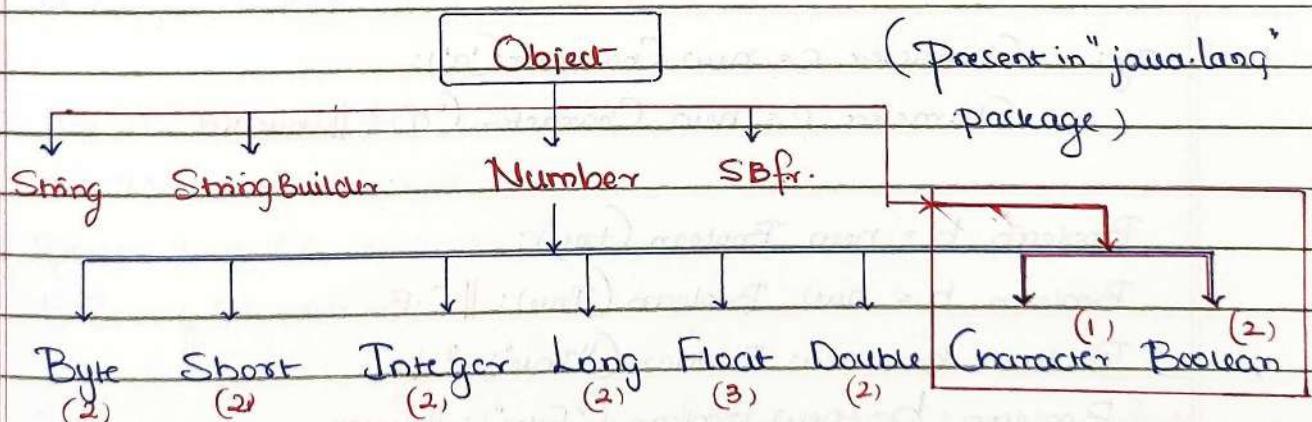
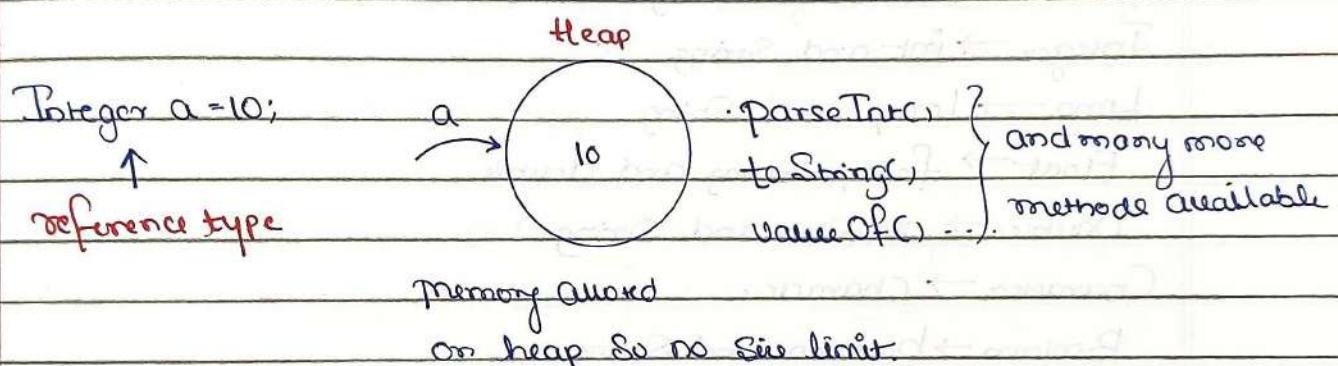
Note: if everything is abstract then it is recommended to go for interface.

Wrapper Class

Purpose : (i) To wrap primitive into object form so that we can handle primitive also just like object.
(ii) To define several utility functions which are required for the primitives.



JDK 1.5 v Wrapper Class are introduced



- * Object → `toString()` : returns the hashCode value of the object.
- * `toString()` : → Overridden to print the data present in the object.
- * `equals()` : → Overridden to compare the content present in the object.

```

cg: Integer i1 = new Integer(10);
      System.out.println(i1); // 10 (toString())
Integer i2 = new Integer("10");
      System.out.println(i2); // 10 (toString())
  
```

- * If String Argument is not properly defined then it would result in RunTime Exception called "NumberFormat-Exception".
- eg: Integer i = new Integer("tio"); // RE

Wrapper class and its associated constructor

Byte → byte and String

Short → short and String

Integer → int and String

Long → long and String

Float → float, String and double

Double → double and String

Character → Character

Boolean → boolean and String

eg: Character c = new Character('a');

Character c = new Character("a"); // invalid

Boolean b = new Boolean(true);

Boolean b = new Boolean(True); // C.E

Boolean b = new Boolean("True"); // true

Boolean b = new Boolean("False"); // false

Boolean b = new Boolean("nitin"); // false

- * If we are using Boolean, boolean value will be treated as true w.r.t insensitive part of "true", for all others it would be treated as "false".

- Note:
- * If we are passing String argument then case is not important and content is not important.
 - * If the content is Case Sensitive String of true then it is treated as true in all other cases it is treated as false.
 - * In Case of Wrapper Class, `toString()` is overridden to print the data.
 - * In Case of Wrapper Class, `equals()` is overridden to check content.
 - * Just like String Class Wrapper class are also treated as "Immutable Class".
- Immutable Class:
- If we create an object and if we try to make a change, with that change new object will be created and those changes will not reflect in old copy.

Q10 Can we make our undefined class as Immutable?

Ans Yes, by making the class final.

Wrapper Class Utility methods

1. `valueOf()` method.
2. `XXXValue()` method.
3. `ParseXXX()` method
4. `toString()` method.

(i) `public static wrapper valueOf (String data, int radix);` } throw
 (ii) `public static wrapper valueOf (String data);` } number
 (iii) `public static wrapper valueOf (int data);` } format
 exception.

1 `valueOf()` method:

- * To create a wrapper object from primitive type or

String we use `valueOf()`.

- * It is alternative to constructor of wrapper class, not suggested to use.
- * Every wrapper class, except character class contain static `valueOf()` to create a wrapper object.

(i)

Eg: `Integer i1 = Integer.valueOf("1111");`

`Sysout(i1); //1111`

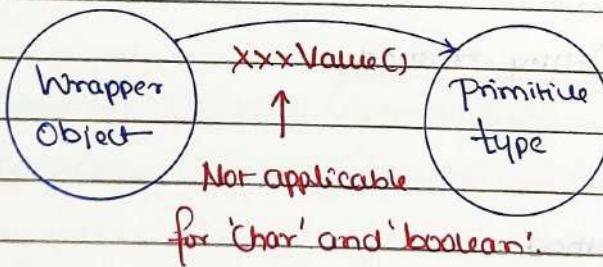
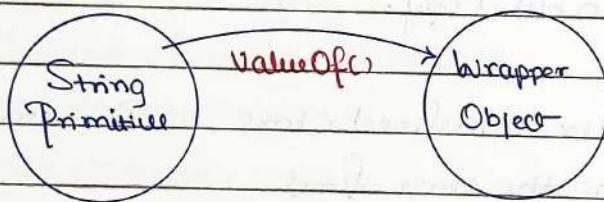
`Integer i2 = Integer.valueOf("1111", 2);`

`Sysout(i2); //15`

`Integer i3 = Integer.valueOf("ten"); //RE: NumberFormatException`

Eg (ii) `Integer i1 = Integer.valueOf(10);`

`Double d1 = Double.valueOf(10.5);`



2. xxxValue()

We can use `xxxValue()` to get primitive type for the given wrapper object. These methods are part of every Number type Object. All these classes have 6 methods.

Eg:

`Integer i = new Integer(130);`

`Sysout(i.byteValue()); // -126`

`Sysout(i.shortValue()); // 130`

`Sysout (i.intValue()); //130`

`Sysout (i.longValue()); //130`

`Sysout (i.floatValue()); //130.0`

`Sysout (i.doubleValue()); //130.0`

3. CharValue()

Character class contains CharValue() to get Char primitive for the given Character object.

eg: Character c = new Character ('c');

Char ch = c.charValue();

4. Boolean Value()

Boolean class contains booleanValue() to get boolean primitive for the given Boolean object.

eg: Boolean b = new Boolean ("initial");

boolean b1 = b.booleanValue();

Sysout (b1); // false

In total xxxValue() are 36 in number

xxxValue() → Convert Wrapper Object → Primitive

5. Parse xxxx()

We use parsexxxx() to convert String object into primitive type.

eg: int i = Integer.parseInt("10");

double d1 = Double.parseDouble ("10.0");

boolean b = Boolean.parseBoolean ("true");

* Every wrapper class, except Character class has parsexxxx() to convert String into primitive type.

eg 1.2. WAP to take inputs from Command line and perform arithmetic operations.

```
int i1 = Integer.parseInt(args[0]);
int i2 = Integer.parseInt(args[1]);
System.out.println((i1 * i2) + (i1 - i2));
```

form-2: public static primitive parseXXX (String, int radix)
Radix → range from 2 to 36

* Every integral type wrapper class (Byte, Short, Int, Long)
Contains the following parseXXX() to convert specified radix
String to primitive type

eg: int i = Integer.parseInt("111", 2); //15.

6. toString()

: To convert wrapper object or primitive to String.
Every wrapper class contains toString().

Form 1: public String toString()

1. Every wrapper class (including character) contains above method to convert wrapper object to String.
 2. It is the overriding version of the Object class toString() method.
 3. Whenever we are trying to print wrapper object reference internally this "toString()" method only executed.
- eg: Integer i = Integer.valueOf("10");

Form 2: public static String toString(primitive type)

Every wrapper class contains a static toString() method to convert primitive to String.

eg: String s = Integer.toString(10);

String s = Character.toString('a');

String s = Boolean.toString(true);

form 3: Public static String toString (primitive p, int radix)

Integer and long class contain the following static toString() method to convert the primitive to specified radix string form.

eg: String s = Integer.toString(15, 2);
System.out(s); // 1111

form 4:

Integer and Long class contain the following toXXXString() methods:

Public static String toBinaryString(primitive p);

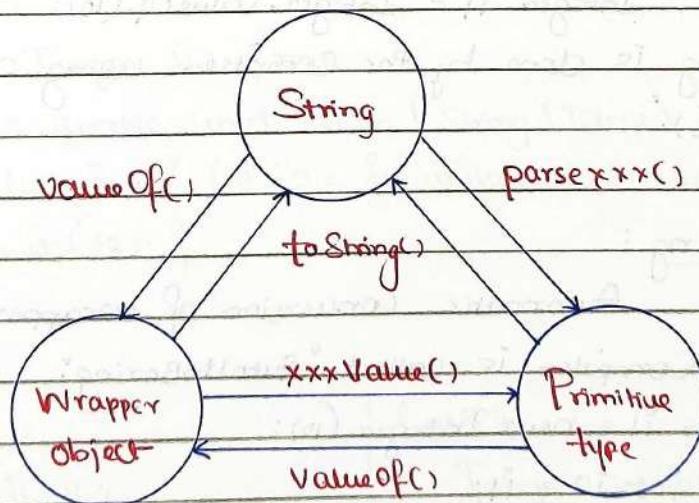
Public static String toOctalString(primitive p);

Public static String toHexString(primitive p);

eg: String s1 = Integer.toBinaryString(7); // 111

String s2 = Integer.toOctalString(10); // 12

String s3 = Integer.toHexString(10); // a



Note: String data = String.valueOf('a'); // static factory method

String data = "Sachin".toUpperCase(); // instance factory method,

AutoBoxing and Auto Unboxing

Until 1.4 version, we can't provide wrapper class objects in place of primitive and primitive in place of wrapper object, all the required conversions should be done by the program. But from Jdk 1.5 version onwards, we can provide primitive in place of wrapper and in place of wrapper we can keep primitives also.

All the required conversions will be done by the compiler automatically, this mechanism is called "AutoBoxing" and "AutoUnboxing".

eg: `ArrayList al = new ArrayList();
al.add(10);`

AutoBoxing:

Automatic conversion of primitive type to wrapper object by the compiler is called "AutoBoxing".

`Integer ii = 10;`

↓ After compilation the code would be

`Integer ii = Integer.valueOf(10);`

AutoBoxing is done by the compiler using a method called "valueOf()".

AutoUnboxing:

Automatic conversion of wrapper object to primitive type by Compiler is called "AutoUnBoxing".

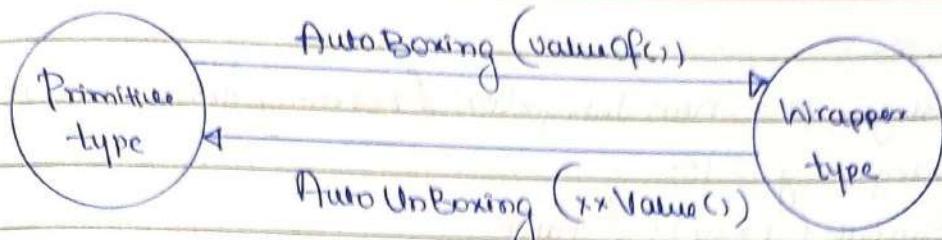
`Integer ii = new Integer(10);`

`int i2 = ii;`

↓ Compiler converts Integer to int

`int i2 = ii.intValue();`

AutoUnBoxing is done by compiler using a method called "xxxValue()";



Compiler will automatically do the conversions from Jav 1.5 version

eg Class Test

```
{ static Integer i1 = 10; // AutoBoxing
```

```
public static void main (String [] args)
{
    int i2 = i1; // AutoUnBoxing
    m1(i2); }
```

```
public static void m1 (Integer i2) // AutoBoxing
{
    int k = i2; // AutoUnBoxing
    System.out.println (k); } // 10
}
```

eg 2 Class Test

```
{ static Integer ii; // null
public static void main (String [] args)
{
    int i2 = ii; // int i2 = ii.intValue() :: Null Pointer Exception
    System.out.println (i2); }
}
```

Case 3:

```
Integer i1 = 10;
Integer i2 = i1; i1++;
System.out.println (i1 + " " + i2); // 10
System.out.println (i1 == i2); // false
```

Case 4:

```
Integer x = new Integer (10);
Integer y = new Integer (10);
System.out.println (x == y); // false
```

Case 5:

Integer x = new Integer(10); // memory from heap area

Integer y = 10;

System.out(x==y); // false

Case 6:

Integer x = new Integer(10);

Integer y = x; // reference is used So pointing to same object

System.out(x==y); // true

Case 7:

Integer x = 10;

Integer y = 10;

System.out(x==y); // true

Integer i = 1000;

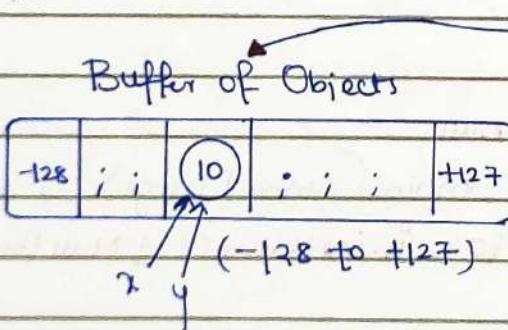
Integer j = 1000;

System.out(i==j); // false

* Compiler uses "valueOf()" for AutoBoxing



Implemented in intelligent way in wrapper class



At the time of loading the class

JVM will buffer with already the objects which are to be used during AutoBoxing range (-128 to 127)

Note:

1. To implement auto boxing concept in wrapper class a buffer of object will be created at the time of class loading.
2. During auto boxing if an object has to be created first JVM will check whether the object is already available inside buffer or not.
3. If available JVM will reuse buffer object instead of creating one.

4. If the object is not available inside buffer, Jvm will create a new object in the heap area, this approach improves the performance and memory utilization.

But this buffer concept is applicable for few cases

1. Byte, Short, Integer, Long : -128 to +127

2. Character : 0 to 127

3. Boolean : true, false

In remaining Cases new object will be created

eg : Integer a=128;

Integer y=128;

Systout (x==y); // false

Integer a=127;

Integer b=127;

Systout (a==b); // true

Boolean b1=true;

Boolean b2=true;

Systout (b1==b2); // true

Double d1 = 10.0;

Double d2 = 10.0;

Systout (d1==d2); // false

Code Snippets

1. final int i1=1;

final Integer i2=1; // memory will be decided at runtime because it is

final String s1=":ONE";

wrapper class object.

String Str1 = i1+s1, Str2 = i2+s1;

Systout (Str1 == "1:ONE"); // true

Systout (Str2 == "1:ONE"); // false

2. String javaworld = "JavaWorld", java="Java", world="World"; // Sc

java=world; // Jvm → JavaWorld (HeapArea)

Systout (java == javaworld); // false

3. `StringBuilder sb = new StringBuilder("SpaceStation");
sb.delete(5,6).insert(5,'$').toString().toUpperCase();
System.out.println(sb); // Space Station`

4. `String s1 = "OCP", s2 = "ocp";
System.out.println(s1.equalsIgnoreCase(s2)); What should be added to print true?
Ans: s1.equalsIgnoreCase(s2), s1.length == s2.length()`

5. `String str = "PANIC"; StringBuilder sb = new StringBuilder("THE").
System.out.println(str.replace("N", sb)); // PATHETIC`

6. `boolean f1 = "Java" == "Java".replace("J", "J"); // true
boolean f2 = "Java" == "Java".replace("J", "J"); // true
System.out.println(f1 & f2); // true.`

7. `StringBuilder sb = new StringBuilder("TOMATO");
System.out.println(sb.reverse().replace("O", "A")); // Compilation error`

8. `String[] strings = "iNeuron-Techology-Private-known-for-Java".
split("-", 3);
for (String string : strings) System.out.println(string);`
Output: iNeuron
Technology
Private-known-for-Java
Specified index for an array.

Output: iNeuron

Technology

Private-known-for-Java

{ Because 3 indices was

specified in split.

9. `String s1 = "null"+null+1; System.out.println(s1); // nullnull1`

10. `System.out.println(1+null+"null"); // Compile time error`

11. `String[] strings = {"Java", "Hibernate", "Spring"};
String l = String.join("-", strings); // Java-Hibernate-Spring`

Var arg method

It Stands for Variable argument methods. In Java if we have variable number of arguments, then compulsory new method has to be written till Jdk 1.4 version.

But in Jdk 1.5 version, we can write single method which can handle variable no. of arguments but all of them should be of same type.

Syntax: methodOne (dataType ... variable name)

... → It stands for "ellipse".

eg: public void int add (int ... x)

(...) JVM internally uses array representation to hold values of x

Object.add(); → new int[] {}

Object.add(10); → new int[] {10}

Object.add(10, 20); → new int[] {10, 20}

eg) Class Demo

```
{ public void add (int ... x)
  { int total = 0;
    for (int i=0; i<x.length; i++) total += x[i];
    System.out.print (total);
  }
}
```

public static void main (String [] args)

```
{ Demo d = new Demo();
```

d.add(); // 0

d.add(10); // 10

d.add(10, 20, 30); // 60 ?

}

Valid Signatures

1. void add (int ... x)
2. void add (int ... x)
3. void add (int ... x)

Case 2 : We can mix normal argument with var argument.

public void methodOne (int x, int... y) ✓

public void methodOne (String s, int... x) ✓

void methodOne (int... x, String s) // Invalid

Case 3 : While mixing Var arg with normal argument var arg should always be last.

public void add (int... a, String b) Invalid.

Case 4 : In an argument list there should be only one var argument.

public void add (int... x, int... y) Invalid

Case 5 : We can overload var arg method, but var arg method will get a call only if none of matches are found (just like "default" of switch case).

eg: Class Test

```
{ public void m1 (int... x)
  { System.out ("var arg method"); }
```

```
public void m1 (int x)
{ System.out ("int method"); }
```

```
public static void main (String [] args)
{
  Test t = new Test ();
  t.m1(); // var arg method
  t.m1 (10, 20); // var arg method
  t.m1 (10); } // int method
```

}

Case 6: public void m1(int... x) → it can't be replaced as int[1x]
but vice versa.

Case 7:

public void m1(int... x)

public void m1(int[1x]) //CE.

We cannot have two methods with same signature

Single dimension array v/s Var Arg method

1. Whenever Single dimension array is present we can replace it with Var Arg.

eg: public static void main(String[1 args]) → String... args.

2. Whenever Var Arg is present we cannot replace it with Single dimension array.

eg: public void m1(String... args) → String[1args] (invalid)

Note:

m1(int... x) → We can call to this method by passing group of int values and it will become 1D array.

m1(int[1x]) → We can call to this method by passing 1D array only

eg: 1. public void m1(int... x) { ... }

Test t = new Test();

t.m1(10, 20, 30); // Treated as 1D array

2. public void m1(int[1...2])

int[1] a = {10, 20, 30}; int[1] b = {40, 50};

Test t = new Test();

t.m1(a, b); // Treated as 2D array

Wrapper Class

1. AutoBoxing
2. Widening (implicit type casting done by compiler for both primitive and wrapper class).
3. Var-args.

Case 1: Widening v/s AutoBoxing

Class Test

```
{
    public static void m1 (long l)
    {
        System.out ("widening");
    }
}
```

```
public static void m1 (Integer i)
{
    System.out ("autoboxing");
}
```

```
public static void main (String [] args)
```

```
{
    int x = 10;
    m1 (10);
}
```

// int → implicit type casting → long

Output: widening

Case 2: Widening v/s Var-arg method

```
public static void m1 (long l)
{
    System.out ("widening");
}
```

```
public static void m1 (int ... i)
{
    System.out ("var-arg method");
}
```

```
public static void main (String [] args)
```

```
{
    int x = 10;
    methodm1 (10);
}
```

// int → typecasting → long

Output: widening

Case 3: Autoboxing v/s var arg method

```

public static void m1(Integer i)
{
    System.out("autoboxing");
}

public static void m1(int... i)
{
    System.out("var-arg method");
}

public static void main(String[] args)
{
    int x=10; m1(x); } // int → type casting → long, float, double
// int → autoboxing → Integer

```

Output: autoboxing.

Case 4:

```

public static void m1(Long l)
{
    System.out("long");
}

public static void main(String[] args)
{
    int x=10; m1(x); } // CE: Can't find the method.

```

- * Widening followed by autoboxing is not allowed in Java, but autoboxing followed by widening is allowed.

Case 5:

```

public static void m1(Object o)
{
    System.out("Object");
}

public static void main(String[] args)
{
    int x=10; m1(x); } // Autoboxing → int → Integer
// Widening → Integer → Number, Object

```

Output: Object

Valid declarations and invalid declarations:

1. int i=10; ✓

2. Integer I = 10; ✓ Autoboxing (ValueOf())
3. int i = 10L; ✗ long → int
4. long l = 10L; ✓ Autoboxing
5. long l = 10; ✗ Autoboxing → Integer, Number so invalid.
6. long l = 10; ✓ int → long
7. Object o = 10; ✓ Autoboxing → Integer → Object
8. double d = 10; ✓ int → Double
9. Double d = 10; ✗ Autoboxing → Integer → Number, Object
10. Number n = 10; ✓ Autoboxing → Integer → Number

New v/s newInstance

1. new is an operator to Create objects, if we know class name at the beginning then we can create an object by using new operator
2. newInstance() is a method presenting Class "Class", which can be used to create object.
3. If we don't know the class name at the beginning and it's available dynamically Runtime then we should go for newInstance() method.

eg: 1. public class Test

```
{ public static void main(String[] args) throws Exception
{ // take input of class name for which object has to be
  created at the runtime.
```

String className = args[0];

// load class file explicitly

Class c = Class.forName(className);

// for the loaded class object is created using zero param constructor
Object obj = c.newInstance();

// perform type casting to get Student object

```
Student std = (Student) obj;
```

```
System.out.println(std); }
```

{

Args (cmd) : java Test Student.

Class Student

```
{ static {
```

```
System.out.println("Student class file is loading"); }
```

```
public Student()
```

```
{ System.out.println("Student constructor is called"); }
```

{}

Output: Student. class file is loading

Student constructor is called

Student@....

Note:

- * If dynamically provide class name is not available then we will get the Runtime Exception saying "ClassNotFoundException".
- * To use newInstance() method Compulsory Corresponding Class Should Contains no Argument Constructor, otherwise we will get the Runtime Exception saying "InstantiationException".
- * If the Argument Constructor is private then it would result in "IllegalAccessException".
- * During typecasting if there is no relationship b/w two classes as parent to child then it would result in "ClassCastException".

Difference between new and newInstance()

new : * new is an operator, which can be used to create an Object.

- * We can use new operator if we know the class name in the beginning
- * If the corresponding class file is not available at the

Runtime then we will get Runtime Exception saying "NoClassDefFoundError". It is unchecked.

To used new Operator the corresponding Class not required to contain no argument constructor.

newInstance() : * newInstance() is a method, present in class "Class" used to create an object

- * We can use the newInstance() method, if we don't write Class name at the beginning and available dynamically Runtime.

- * Object o = Class.forName(args[0]).newInstance();

- * If the Corresponding .class file not available at runtime then we will get runtime exception saying "ClassNotFoundException", it is checked.

- * To used newInstance() method the corresponding class should Compulsory Contain no Runtime Exception saying InstantiationException.

Difference between ClassNotFoundException and NoClassDefFound Error :

1. For hard coded class name at Runtime in the corresponding .class files not available we will get NoClassDefFoundError, which is unchecked

Test t = new Test();

In Runtime Test.class file is not available then we will get NoClassDefFoundError

2. For dynamically provided class name at Runtime, if the corresponding .class files is not available then we will get the Runtime Exception saying "ClassNotFoundException".

Ex: Object o = Class.forName("Test").newInstance();
At runtime if .class file not found we get ClassNotFoundException

Note:

New will Create memory on the heap area

Student: → Jvm will search for .class file in current working directory if found load the file into Method Area.

During the loading of .class file

- Static variable will get memory set with default value
- Static block gets executed

In the heap area, for the required object memory for instance variables is given jvm will set the default values to it

- Execute the instance block if available
- Call the constructor to set the meaningful value to the instance variable

Jvm will give the address of the Object to hashing algorithm which generates the hashCode for the Object and that hashCode will be returned as the reference to the programmer.

Import Statements:Class Test

```
{ public static void main (String args[])
    { ArrayList l = new ArrayList();
}
```

Output: Compile time error (Cannot find symbol)

* We can resolve this problem by using fully qualified name "java.util.ArrayList" l = new java.util.ArrayList(); But problem is we need to use this name everytime and increase length of code and reduce readability.

* We can resolve this problem by using Import Statements

```

Example: import java.util.ArrayList;
class Test{
    public static void main(String[] args)
    { ArrayList t = new ArrayList(); }
}

```

- * Hence whenever we are using import statements it is not required to use fully qualified names, we can use short names directly.
- * This approach decreases length of code and improves readability.

Case 1: Types of import statements

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

Explicit Class import:

eg: import java.util.ArrayList;

- * This type of import is highly recommended because it improves the readability of the code.
- * Best suitable for developers where readability is important.

Implicit Class import:

eg: import java.util.*;

- * It is not recommended to use because it reduces the readability of the code.
- * Best suited for students where typing is important.

Case 2 Which of statements are meaningful: ?

- a. import java.util;
- b. import java.util.ArrayList.*;
- c. import java.util.*;
- d. import java.util.ArrayList;

Implicit Import

```
import java.util.*;
```

```
ArrayList
```

Compilation Time ↑

Explicit Import

```
import java.util.ArrayList;
```

```
ArrayList
```

Compilation Time ↓

Case 3 Class MyArrayList extends java.util.ArrayList

- * 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

```
import java.util.*;
import java.sql.*;
Class Test{
    public static void main (String [] args)
    {
        Date d = new Date();
    }
}
```

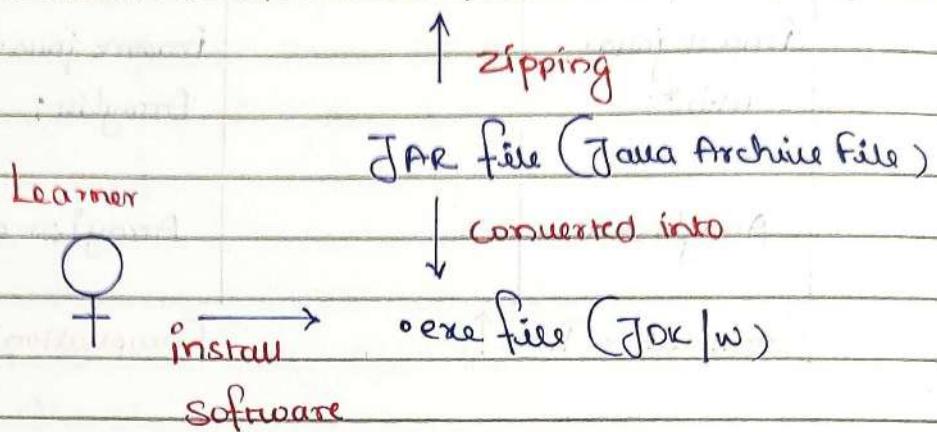
Output: Compile time error

Reference to Date is ambiguous.

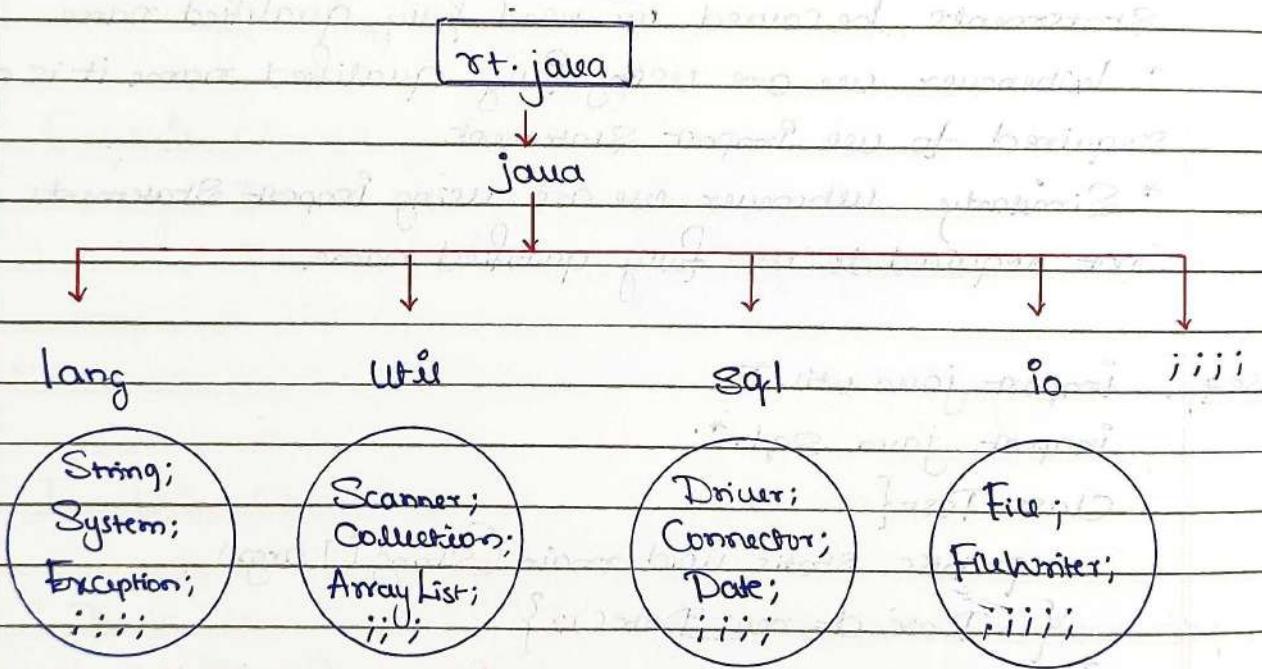
Error in list case also we may get the same ambiguity problem because it is available in both util and sql package.

* JDK software is an API code

API : Collection of .class files (class/interface/enum)



* Sooot .class files are present, for maintenance and easy use they created folder and keep the .class files, these folders are called as "packages" in java.



→ Need to inform compiler and jump about these classes through "import"

Cases

While resolving class name compiler will always give the importance in the following order.

1. Explicit class import

2. Classes present in the current working directory

3. Implicit Class import

eg:

```
import java.util.Date;
import java.sql.*;

class Test{
    public static void main (String args[])
    {
        Date d= new Date();
    }
}
```

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

Case 6

Whenever we are importing an / a package all classes and interfaces present in that package are by default available but not Sub Package Classes.

java

 |→ util

 |→ Scanner class, ArrayList class, LinkedList class

 |→ regex

 |→ Pattern class

To use Pattern class in our program directly which import statement is required?

Ans

import java.util.regex.*; (implicit import)

import java.util.regex.Pattern; (explicit import)

Case 7

In any java program the following 2 packages are not required to import because there 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 compilation but "no change in execution time."

Difference between C language #include and java language import?

	C/C++		Java
C Compiler	#include <stdio.h> 3000+ functions are available	Java Compiler	import java. lang.*; 300+ classes
Static inclusion	Compiler will bring all 3000 functions to our code	Jvm will load System class only when it is needed so we call it as "Dynamic inclusion"	System String StringBuffer
(memory is wasted.)			(Memory is effectively utilised)

#include

1. It can be used in C and C++.
2. At compile time only Compiler copy the code from std library and placed in curr. program.
3. It is static inclusion.
4. wastage of memory.

import

1. It can be used in Java.
2. At runtime Jvm will execute the corresponding std library and use its result in curr. program.
3. It is dynamic inclusion.
4. No wastage of memory.

Note: In the case of C language #include : all the header file will be loaded at the time of include statement hence it follows static loading.

- * But in java import statement number of ".class" will be loaded at the time of import statements in the next lines of code whenever we are using a particular class then only corresponding ".class" file will be loaded.
- Hence it follows "dynamic inclusion" or "load on demand" or "load on fly".

JDK 1.5 Version's new features:

- | | |
|--------------------------------|----------------------------|
| 1. For - Each | 6. Co-varient return types |
| 2. Var-Arg | 7. Annotations |
| 3. Queue | 8. Enums |
| 4. Generics | 9. Static import |
| 5. Autoboxing and AutoUnboxing | 10. String Builder |

Static import:

If was introduced in JDK 1.5 versions . According to Sun microsystems : Static import improves readability of code but according to world wide programming experts Static import creates confusion and reduce readability of the code . Hence if there is no specific requirement it is not recommended to use static import .

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

Without Static Import : Class Test {

```
public static void main (String [ ] args)
{
    System.out (math.sqrt(4)); // 2
    System.out (math.max (10,20)); } // 20
}
```

With Static import:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;

class Test{
    public static void main(String[] args)
    {
        System.out.println(sqrt(4)); // 2.0
        System.out.println(max(10, 20)); // 20
    }
}
```

eg: class Test{

```
    static String name = "Sachin";
    Test.name.length(); // 6
```

eg

System.out.println()

It is a method of PrintStream Class.
It is a reference of PrintStream Class
It is a class

eg:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
```

Class Test

```
{ public static void main(String[] args)
    {
        System.out.println(Integer.MAX_VALUE);
    }
}
```

Output: Compile-time error

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 interface 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.

eg: `import static java.lang.Integer.MAX_VALUE;`

`import static java.lang.*;`

`Class Test {`

`static int MAX-VALUE = 999;`

`public static void main(String []args)`

`{ System.out.println(MAX-VALUE); }`

`}`

Which of the following statements are valid?

`import java.lang.Math.*; // invalid`

`import static java.lang.Math.*; // valid`

`import java.lang.Math; // valid`

`import static java.lang.Math; // invalid`

`import static java.lang.Math.sqrt.*; // invalid`

`import java.lang.Math.Sqrt; // invalid`

`import static java.lang.Math.Sqrt(); // invalid`

`import static java.lang.Math.Sqrt; // valid`

Usage of static import reduce readability and create confusion

Hence if there is no specific requirement never recommend to use static import.

Difference between general import and static import.

Normal import:

* We can use normal imports to import classes and interface of package.

- * Whenever we are using ^{normal} static import we can access class and interface directly by their short name, it is not required to use fully qualified names.

Static Import:

- * We can use static import to import static members of a particular class
- * Whenever we are using static import it is not required to use class name, we can access static members directly.

Functional Interface:

An interface having only one abstract method is a functional interface.

e.g.:

```
@FunctionalInterface // indication to compiler (annotation)
interface Demo
{
    void disp(); }
```

Note: Lambda expressions and Functional Interface, they work together.

Lambda Expression:

In Java 8 a new operator was introduced



— arrow operator / lambda operator to write lambda expression.

e.g. interface Demo
{ void disp(); }

Public class Launch {

 public static void main(St[ta])

<pre>Demo d = () -> System.out.println("Hello");</pre>	<pre>d.disp(); } } </pre>
	bracket for multi. stmts.
	Output: Hello

eg. interface Demo

```
{ public void disp(); }
```

public class Launch {

```
    public static void main (String [] args)
```

```
{
```

```
    // Demo d = new Demo(); // not valid
```

```
    Demo d = new Demo() { // implementation of functional
```

// Anonymous

method approach.

```
    void disp()
```

```
    { System.out ("Hello"); }
```

```
}
```

```
d.disp();
```

interface without "implements"

// multiple methods can be

implemented like this from a

normal interface.

```
}
```

// you cannot write more than one method in func. interf.

eg. @FunctionalInterface

interface Add

```
{ void add (int a, int b); }
```

@FunctionalInterface interface Sub

```
{ void sub (int num1); }
```

Public Class LaunchLambda

```
Add add = (a, b) → { // implementing Add interface
```

int res = a + b; // writing data type of

System.out (res); }; parameters is optional

```
add.add (10, 20); // 20
```

Sub sub = num1 → { // implementing Sub interface

int res = num1 - 5; // no need of parenthesis

return res; }; for single parameter

/Sub Sub = num1 → { return num1 - 5; }^{Curly brackets req.}

// valid lambda expression, no need to specify
the data type of parameters,

/Sub Sub = num1 → num1 - 5; // bracket not req. cos no return

// no need to write return statement, compiler will
automatically detect the return type and return.

Sub. Sub(10); // 5; }

}

Key points about lambda expressions:

1. To write lambda expression we use lambda operator (\rightarrow).
2. Lambda operator divided into two parts to write lambda exp:
left side of lambda operator we write parameters required.
right side we write body or implementation.
3. Left side for parameters declaring/writing data type is optional.
4. Right side if implementation/body has one statement then {} is optional.
5. Left side if parameter is single the () is optional.
6. Right side in body if its single line implementation then return statement is also optional.
7. {} is mandatory if there are more than one statement and also if there is return statement explicitly used by the developer.

WAP to compute length of String:

@ Functional Interface

interface CLS

{ int getLength(String); }

```
public class Launch {  
    public static void main (String [] str)  
    {  
        CLS sl = str → str.length(); // lambda expression  
        System.out.println (sl.getLength ("Nuke"));  
    }  
}
```

Output: 5

Exception Handling

Exception handling in java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception handling is a mechanism to handle runtime errors.

Exception is an unwanted or unexpected event which occurs during execution of a program at runtime that disrupts the normal flow of program instructions. Exceptions can be caught and handled by the program. (Default Exception Handler).

Major reasons why an exception occurs:

- * invalid user input
- * device failure
- * loss of network connection
- * Physical limitations
- * Code errors
- * Opening an unavailable file.

When an exception occurs within a method, it creates an object. This object is called exception object. It contains information about the exception, such as the name and description of the exception and state of the program when the exception occurred.

If the exception is not handled by the developer, then the object will go to the "Default Exception Handler" else if try and catch blocks are given by the developer then Jvm will not go to "Default Exception Handler".

Example 1

```
import java.lang.util.*;
```

```
Sysout ("Connection Established");
try {
```

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

```
int num1 = Scan.nextInt();
```

```
int num2 = Scan.nextInt();
```

```
int res = num1 / num2;
```

```
Sysout ("The result is " + res); }
```

Input: 10 5

Output: The result
is 2

```
Catch (Exception e) // General exception
```

```
{ Sysout ("You are dividing by zero"); }
```

Input: 10 0

Output: You are
dividing by zero

Examp. 2

```
Sysout ("Connection established");
```

```
try {
```

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

```
Sysout ("Enter 1st number");
```

```
int num1 = sc.nextInt(); // num1 input
```

```
Sysout ("Enter 2nd number");
```

```
int num2 = sc.nextInt(); // num2 input
```

```
int res = num1 / num2;
```

```
Sysout ("Result is : " + res); // Executes if exception not  
occur
```

```
Sysout ("Enter array size");
```

```
int size = Scan.nextInt(); // size input for array
```

```
int [] a = new int [size]; // array initialization
```

```
Sysout ("Enter element to be inserted");
```

```
int ele = sc.nextInt();
```

```
Sysout ("Enter position"); // position should be from
```

```
int pos = sc.nextInt(); 0-(size-1),
```

$a[pos] = ele;$

Sysout ("Element is inserted"); }

Catch (ArithmeticException ae)

{ Sysout ("Dividing by zero"); }

// Specific catch blocks are executed, rest are not executed

catch (NegativeArraySizeException nae)

{ Sysout ("Please be positive"); }

// "Exception" is the parent of all exceptions is kept

Catch (ArrayIndexOutOfBoundsException a) at last.

{ Sysout ("Be in your limits"); }

Sysout ("connection terminated");

Input:

Connection Established

Enter 1st number 100

Enter 2nd number 0

Dividing by zero

Connection terminated

Test Case 2:

Connection Established

Enter 1st number 50

Enter 2nd number 5

The result is 10

Enter array size 5

Enter element to insert 500

Enter position 5

Be in your limits

Connection terminated

Test Case 3:

Connection established

Enter 1st number 70

Enter 2nd number 7

The result is 10

Enter array size -5

Please be positive

Connection terminated.

Note: * Single try block and multiple catch blocks are allowed.

* Corresponding catch block will be called and rest will not be executed.

* As a developer you need to give generic exception "Exception" for default.

Key points:

- * In java to handle exception we use try and catch blocks
- * Catch block will be executed only if there is an exception in try block otherwise it will not.
- * If there is an exception, then the statements below the statement where the exception has occurred will not be executed.
- * One try block can have multiple catch blocks
- * When we are writing catch blocks specific to every kind of exception, it is recommended to use generic catch block
- * The generic catch block must be at the end

Whenever there is an Exception:

1. Handle exception (try and catch block)
2. Duck the exception (throw)
3. Re-throwing an exception (throw, throws, try, catch, finally)

Exceptions can be Categorized in two ways:

1. Built-in Exceptions: Built in exceptions are the exceptions that are available in Java libraries.
 - (a) Checked exceptions: Checked exceptions are called Compile time exceptions because the exceptions are checked at compile time by the compiler.
 - (b) Unchecked exceptions: The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time.
2. User defined exceptions: Sometimes, the built in exceptions in Java are not able to describe a situation. In such cases users can also create exceptions, which are called 'User-defined exceptions'.

Example

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

Class Alpha

```
{
    void alpha() throws ArithmeticException
    {
        System.out.println("Connection established"); // throws Keyword throws
        Scanner Scan = Scan.nextInt(); // the exception to whom
        System.out.println("Enter first number");
        int num1 = Scan.nextInt();
        System.out.println("Enter second number"); // caught it and handled
        int num2 = Scan.nextInt();
        int res = num1 / num2; // where catch block is
        System.out.println("The res is " + res);
        System.out.println("Connection is terminated");
    }
}
```

// if catch block is within the method, then exception is handled and main catch is not executed.

Class Beta

```
{
    void beta() throws ArithmeticException
    {
        Alpha a = new Alpha();
        a.alpha();
    }
}
```

public class Launch

```
{
    public static void main(String[] args)
    {
        try
        {
            Beta b = new Beta();
            b.beta();
        }
    }
}
```

Catch (ArithmeticException e)

```
{
    System.out.println(e.getMessage());
    System.out.println("Exception finally handled
        in main");
}
```

Input: 100 0Output:

"Exception finally
handled in main."

Example

```

import java.util.Scanner;
// Re-throwing an exception
class Alpha1
{
    void alpha() throws ArithmeticException
    {
        System.out.print("Connection Established");
        try
        {
            Scanner sc = new Scanner(System.in);
            System.out("Enter first number");
            int num1 = sc.nextInt();
            System.out("Enter second number");
            int num2 = sc.nextInt();
            int res = num1 / num2;
            System.out("The result is " + res);
        }
    }
}

```

Catch (Arithmetic Exception e)

```

{
    System.out.println("Exception handled in Alpha");
    throw e; // rethrowing the exception to the method where
              // Alpha1 was created
}

```

finally

```

{
    System.out.println("Connection is terminated");
}
// finally will execute after the exception is handled
}

```

public class Launch

```

{
    public static void main(String[] args)
    {
        try
        {
            System.out("Main Method");
            Alpha1 a = new Alpha1();
            a.alpha();
        }
    }
}

```

Catch (Arithmetic Exception e)

```

{
    System.out("Exception handled in main");
}

```

Output : Main method

Connection established

Enter first number 100

Enter Second number 0

Exception handled in Alpha

Connection is terminated.

Exception handled in main

throw

throws

- | | |
|---|---|
| * Written inside method, usually in Catch block | Written in method signature |
| * It is used to rethrow an exception | It is used to catch an exception. |
| * Statements below the throw keyword are not executed | Statements below the 'throws' keyword are executed in the method. |

Catch : → if there is no exception, block will not execute
 → Only executes if there is a matching exception

finally : finally block statements are executed if there are exceptions, no exception, throw keyword, hidden stack.

Purpose of:

try and catch → to handle exception

throws → catch and method signature

throw → Catch → rethrow

finally → Close resource

The Exception Object Contains : 1. Name of the exception

2. Description of the exception

3. Stack trace of the exception

Methods to print exception information:

1. getMessage() : prints the description of the exception

Example: / by zero

2. toString() : prints the name and description of exception

Example: ArithmeticException: / by zero

3. PrintStackTrace() : prints the name and the description of the exception along with stack trace.

Example: ArithmeticException: / by zero at Demo.alpha();

Example

Class Demo

```

{ int disp()
  {
    try
    {
      System.out.println("Method Started");
      return 10; } //finally block cannot be written without
    finally
    {
      System.out.println("Method Ending"); }
  } //finally block will be executed even after return
  statement (finally will dominate "return").
```

public class Launch{

```
  public static void main(String[] args)
```

```
  {
    Demo d = new Demo();
  }
```

```
  d.disp(); }
```

```
}
```

Output: Method Started

Method Ending.

Note: `System.exit(0)` will dominate finally and return statement.

In above example replacing 'return' with `System.exit(0)` then the program will stop after the first statement and the program is terminated, no more running.

Code Snippets

1. Class Atom

{ Atom() { Sysout("Atom"); } }

Class Rock extends Atom

{ Rock (String type) { Sysout(type); } }

public class Mountain extends Rock {

Mountain()

Super ("granite");

new Rock ("granite"); }

public static void main (String [1a]) { new Mountain(); }

}

Output: atom granite atom granite

2. Interface TestA { String toString(); }

public class Test {

public static void main (String [] args)

{ Sysout (new TestA) { public String toString() { return "test"; } } }

{ }

3. Class Building { }

Public class Barn extends Building { }

public static void main (String [] args)

{ Building b1 = new Building(); Barn b1 = new Barn(); }

Barn b2 = (Barn) b1; Object ob1 = (Object) b1;

String str1 = (String) b1; Building b2 = (Building) b1; }

{ ↳ RE: Class Cast Exception (this should be removed to compile).

4. interface Foo { }

Class Alpha implements Foo { }

Class Beta extends Alpha { }

Class Delta extends Beta { }

{ public static void main (String [] args) { }

Beta x = new Beta();

16. // insert code here! }

? What line should be inserted to get ClassCastException?

Ans: Foo f = (Delta)x;

5. Public class Batman { int square = 81;

Public static void main (String [] args) { new Batman().go(); }

void go () {

incr (++square); System.out (square); }

void incr (int square) { square += 10; }

? Output: 82

6. Class Pass { public static void main (String [] args) { }

{ int x = 5; Pass p = new Pass();

p.doStuff(); System.out ("main x = " + x); }

void doStuff (int x) { }

{ System.out ("do stuff x = " + x); }

Output: do stuff x = 5

main x = 5

7. String [] elements = { "for", "tea", "too" };

String first = (elements.length > 0) ? elements[0] : null;

8. Class Foo { public int a = 3; // instance variable

public void addFive() { a += 5; System.out ("f"); }

? Class Bar extends Foo { public int a = 8; }

public void addFive() { this.a += 5; System.out ("b"); }

?

Exception Handling in Java

try {

Statement 1;

Statement 2;

Statement 3;

try {

Statement 4;

Statement 5;

Statement 6; }

Catch (xxx e)

{ Statement 7; }

finally { Statement 8; }

Statement 9;

}

Catch (yyy e)

{ Statement 10; }

finally { Statement 11; }

}

Statement 12;

Case 6: Exception at 5, and both catch blocks not matched.

Statements 8, 4, 1, 2, 3, 11 will be executed (abnormal termination)

Case 1: If no exception occurs

Statements 1, 2, 3, 4, 5, 6, 8, 9, 11, 12 will be executed.

Case 2: If exception occurs

at Stmt 2 and corresponding catch block is matched.

Statements 1, 10, 11, 12 will be executed

Case 3: If exception occurs

at Stmt. 2 and corresponding catch block is not matched.

Statements 1, 11 will be executed resulting in abnormal termination.

Case 4: If exception occurs

at 5, and corresponding inner block is matched.

Statements 1, 2, 3, 4, 7, 8, 9, 11, 12 will be executed (normal)

Case 5: Exception at 5, and inner block not matched but outer catch block matched

Statements 1, 2, 3, 4, 8, 10, 11, 12 will be executed (normal)

Valid Syntax in Exception Handling

1. Only try

`try { ... }` Not Allowed

2. Only Catch

`Catch (xxx e)` Not allowed
`{ ... }`

3. Only finally

`finally { ... }` Not allowed

4. try-Catch

`try { ... }`
`Catch (xxx e)` Allowed
`{ ... }`

5. Reverse order

`Catch (xxx e)`
`{ ... }` Not allowed
`try { }`

6. Multiple try

`try { ... }`
`try { ... }` Not valid
`Catch (xxx e)`
`{ ... }`

7. Multiple try and catch

<code>try { ... }</code>	<code>try { ... }</code>
<code>Catch (xxx e)</code>	<code>Catch (xxx e)</code>
<code>{ ... }</code>	<code>{ ... }</code>

Valid

8. Multiple try

`try { ... }``Catch (xxx e) Not allowed``{ ... }``try { ... }`

9. Multiple Catch

`try { ... }``Catch (xxx e) { ... } Allowed``Catch (yyy e) { ... }`

10. Multiple Catch

`try { ... }``Catch (xxx e) { ... } Not allowed``Catch (xxx e) { ... }`

11. Multiple Catch

`try { ... }``Catch (xxx | yyy e) { ... } Allowed`

12. try-finally

`try { ... } Allowed``finally { ... }`

13. Catch-finally

`Catch (xxx e)``{ ... } Not allowed.``finally { ... }`

14. Unordered

`try { ... }``finally { ... }``Not allowed``Catch (xxx e) { ... }`

15. try - multiple catch - finally

```
try{...}
catch (xxx e){...} Allowed
catch (yyy e){...}
finally {...}
```

16. Multiple finally

```
try{...}
catch (xxx e){...} Not allowed
finally {...}
finally {...}
```

17. Statements in between try - Catch - finally

```
try{...}
System.out ("");
catch (yyy e){...}
System.out ("");
finally {...}
```

18. Only try with resource

```
try (R) Allowed
{...}
```

Synchronous Exceptions :- Occur at specific program statement.

Class Launch

```
{ public static void main (String [] args)
{
    String Str= null;
    Str.toUpperCase();
```

```
}
```

→ Null Pointer Exception

A Synchronous Exception: occurs anywhere in the program.

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

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

```
    System.out ("Enter name");
```

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

```
    System.out ("Enter your grades");
```

```
    String grade = Scan.next(); }
```

Output: Enter your name: Sachin

Enter your grades: **CTRL + C** (Keyboard interrupt)

Exception in thread "main" Terminate Batch Job (Y/N)?

User Defined Exception

Code

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

```
class InvalidCustomerException extends Exception // User defined
{ public InvalidCustomerException (String msg) exception
    { super (msg); }}
```

Class Atm

```
{ int userId = 1212; // data for verification
```

```
int password = 1111;
```

```
int pw; int uid;
```

```
public void input()
```

```
{ Scanner Scan = new Scanner (System.in);
```

```
System.out ("Kindly enter user id");
```

```
uid = Scan.nextInt();
```

```
Sysout ("Enter the password");
pw = Scan.nextInt(); }
```

```
{ Public void verify () throws InvalidCustomerException
{ if (Cuserid == uid) && (password == pw)
{ Sysout ("Take your cash"); }
else
{ InvalidCustomerException ice = new InvalidCustomerException ("Are you sure?"); // passing args to
Sysout (ice.getMessage()); // Constructor
throw ice; } // re-throwing
}
}
```

Class Bank

```
{ public void initiate ()
{ Atm a = new Atm();
try {
a.input();
a.verify();
}
Catch (InvalidCustomerException e1) // first catch block
{
try { a.input(); // re-enter the data for verification
a.verify(); }
}
}
```

```
Catch (InvalidCustomerException e2) // second catch block
```

```
{ try {
a.input(); // re-confirmation
a.verify(); }
}
// final time verification
```

```

    Catch (InvalidCustomerException e) // final catch block
    {
        System.out.println("We caught you! Card is blocked");
        System.exit(0); // terminate the program.
    }
}
}
}
}

```

Public Class LaunchCE

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

```
{
    Bank b = new Bank();
    b.initiate(); }
```

detailMessage is
used by these
methods to print
exception info.

Throwable
String detailMessage;
Throwable (String msg)
{ detailMessage = msg; }
getmessage()
toString()
PrintStackTrace()

Exception
Exception (String msg)
{ Super (msg); }

InvalidCustomerExcep.
InvalidCustomerException (String msg) { Super (msg); }

new InvalidCustomerException
("Invalid Input");

A simple RTO program to implement exception handling:

```
import java.util.Scanner;  
class UnderAgeException extends Exception // user defined  
{  
    public UnderAgeException (String msg) Class 1  
    {  
        super (msg); }  
}
```

```
Class OverageException extends Exception // user defined  
{  
    public OverageException (String msg) Class 2  
    {  
        super (msg); }  
}
```

```
Class Applicant  
{  
    int age;
```

```
    public void input()  
    {  
        Scanner Scan = new Scanner (System.in);  
        System.out ("Enter your age");  
        age = Scan.nextInt(); }  
}
```

```
void verify() throws UnderAgeException, OverageException  
{  
    if (age < 18)  
    {  
        UnderAgeException ue = new UnderAgeException  
        ("Wait till your age!");  
        System.out (ue.getMessage());  
        throw ue; }  
}
```

```
else if (age > 60)  
{  
    OverageException oe = new OverageException  
    ("your time is near");  
}
```

```
Sysout (oae.getmessage(1));
-throw oae; }
```

```
else { Sysout (" You are eligible"); }
}
```

Class RTO

```
{ public void initiate()
{ Applicant a = new Applicant();
try{ a.input();
a.verify(); } }
```

Catch (UnderAge Exception | Overage Exception e) // handle one

```
{ try{ a.input();
a.verify(); }
```

Catch (UnderAge Exception | Overage Exception e1)

```
{ Sysout ("Don't ever try again");
System.exit(0); } // terminate the program.
```

public class LaunchRTO {

```
public static void main (String [] args)
{ RTO r= new RTO();
r.initiate(); }
```

JDK 1.7 version enhancements

1. -try with resource
2. -try with multicatch block

Until jdk 1.6, it is compulsory required to write finally block to close all the resources which are open as a part of try block.

```
eg:: BufferedReader br = null;  
try{  
    br = new BufferedReader(new FileReader("abc.txt"));  
}  
catch (IOException ie)  
{  
    ie.printStackTrace();  
}  
finally{  
    try{ if(br != null)  
        br.close(); }  
    catch (IOException ie)  
    { ie.printStackTrace(); }  
}
```

Problems in this approach:

1. Compulsorily the programmer is required to close all opened resources which increases the complexity of the program.
2. Compulsorily we should write finally block explicitly, which increases the length of the code and reduces readability.

To overcome this problem SUN MicroSystem introduced try with resources in "1.7" version of Jdk.

try with resource

In this approach, the resources which are opened as a part of try block will be closed automatically once the control reaches to the end of the code try block normally or abnormally, so it is not required to close explicitly and complexity of the program would be reduced.

It is not required to write finally block explicitly, so length of code would be reduced and readability is improved.

eg:: try (BufferedReader br = new BufferedReader (new FileReader ("abc.txt")))

{ // use br and perform the necessary operation
 // Once Control reaches to the end of try block automatically
 // br will be closed }

Catch (IOException ie)

{ // handling code }

Rules of using try with resource

1. We can declare any no. of resource, but all these resources should be separated with ;

eg:: try (R₁; R₂; R₃;) {
 // Use the resource }

2. All resources are said to be AutoClosable resources if the class implements an interface called as "java.lang.AutoCloseable". either directly or indirectly.

eg:: java.io package classes, java.sql package classes.
 public interface java.lang.AutoCloseable {
 public abstract void close() throws java.lang.Exception; }

Note: Which ever class has implemented this interface those class objects are referred as "resources".

3. All resource reference by default are treated as implicitly final and hence we can't perform assignment within try block.

```
try (BufferedReader br = new BufferedReader (new FileReader
("abc.txt")))
{
    br = new BufferedReader (new FileReader ("abc.txt"));
}
```

Output :: CE Can't reassign a value.

4. Until 1.6 version try should compulsorily be followed by either catch or finally, but from 1.7 version we can take only "try with resource" without catch or finally.

```
try (R) {
    /valid }
```

5. Advantage of try with resource concept is finally block will become dummy because it is not required to close resource explicitly.

6. try with resource nesting is also possible

```
try (R1) {
    try (R2) {
        try (R3) {
            / }
        }
    }
}
```

Note: A code which should repeat multiple times in our project with small change or no change, such code we call it as "boilerplate" code.

Multi Catch Block

Till jdk 1.6, even though we have multiple exception having same handling code we have to write a separate catch block for every exception, it increases length of code and reduces readability.

eg:: try{...
...}

Catch (ArithmeticException ae){
ae.printStackTrace(); }

Catch (NullPointerException ne){
ne.printStackTrace(); }

To overcome this problem Sun has introduced "multi catch block" concept in 1.7 version

eg:: try{...
...}

Catch (ArithmeticException | NullPointerException ae)
{ e.printStackTrace(); }

Catch (ClassCastException | IOException e)
{ e.printStackTrace(); }

In multi catch block, there should not be any relation b/w exception types (either Child to parent or parent to child or same type) it would result in compile time error.

eg:: try{..}

Catch (ArithmeticException | IOException)
{ e.printStackTrace(); }

Output: CE

Rules of overriding when exception is involved.

While overriding if the child class method throws any checked exception compulsorily the parent class method should throw the same checked exception or its parent. Otherwise we will get compile time error.

There are no restrictions on unchecked exceptions.
eg::

```
Class Parent { public void m1(); }

Class Child extends Parent
{ public void m1() throws Exception {} }
```

Error: m1() in Child cannot override m1() in Parent
public void m1() throws Exception {}
Overridden method does not throw Exception.

Rules w.r.t Overriding

1. Parent: public void m1() throws Exception {}

Child: public void m1()

Valid

2. Parent: public void m1() {}

Child: public void m1() throws Exception {}

Invalid

3. Parent: public void m1() throws Exception {}

Child: public void m1() throws Exception {}

Valid

4. Parent: public void m1() throws IOException {}

Child: public void m1() throws IOException {} **Valid**

5. Parent: public void mi() throws IOException {}
 Child: public void mi() throws FileNotFoundException,
 EOFException {} **Valid**

6. Parent: public void mi() throws IOException {}
 Child: public void mi() throws FileNotFoundException,
 InterruptException {} **Invalid**

7. Parent: public void mi()
 Child : public void mi() throws ArithmeticException,
 NullPointerException, RuntimeException {} **Valid**

8. Parent: public void mi() throws IOException
 Child : public void mi() throws Exception {} **Invalid**

9. Parent: public void mi() throws Throwable {}
 Child : public void mi() throws IOException {} **Valid**

* instanceof : We can use the instanceof operator to check whether the given an object is particular type or not

eg:: $\text{reference} \ \text{instanceof} \ \text{Class/interface name}$

eg:: ArrayList al = new ArrayList(); // inbuilt object where we can keep any type of other objects
 al.add(new Student()); // 0th position
 al.add(new Cricketer()); // 1st position
 al.add(new Customer()); // 2nd position

Object o = l.get(0); // l is an ArrayList object.

if (o instanceof Student) { Student s = (Student)o;
 // perform Student Specific opn. }

```

        elseif (o instanceof Customer) {
            Customer c = (Customer)o;
            // perform customer specific operations
        }
    
```

eg:: Thread t = new Thread();
 Sysout (t instanceof Thread); // true
 Sysout (t instanceof Object); // true
 Sysout (t instanceof Runnable); // true

eg:: Public class Thread extends Object implements Runnable { }

* To use instanceof operator compulsorily there should be some relation between argument type (either child to parent or parent to child or same type) otherwise we will get Compile time error saying incompatible types.

eg:: String s = new String ("Sachin");
 Sysout (s instanceof Thread); // CE

* Whenever we are checking the parent object is child type or not by using instanceof operator that we get false.

```

Object o = new Object();
Sysout (o instanceof String); // false
Object o = new String ("abck");
Sysout (o instanceof String); // true
    
```

* for any class or interface x null instanceof x is always returns false Sysout (null instanceof x); // false

eg:: Object t = new Thread();
 Sysout (t instanceof Object); // true
 Sysout (t instanceof Thread); // true
 Sysout (t instanceof Runnable); // true
 Sysout (t instanceof String); // false
 Sysout (new instanceof Object); // false

- * isInstance(): The isInstance() method is used to check if the specified object is compatible to be assigned to the instance of this class.

Difference between instanceof and isInstance():

instanceof:

- * instanceof is an operator which can be used to check whether the given object is a particular type or not.
- * eg:: String s = new String("Sachin");
 Sysout (s instanceof Object); // true

isInstance():

- * isInstance() is a method, present in class "Class", we can use it to check whether the given object is particular type or not. We don't know the type at the beginning, it is available dynamically at runtime.

* eg:: class Test{

```
public static void main (String [ ] args)
{ Test t = new Test();
```

```
Sysout (Class.forName (args[0]).isInstance (t)); }
```

}

java Test Test // true
 java Test String // false

Code Snippets

1.

```
interface Foo{}  
Class Alpha implements Foo{}  
Class Beta extends Alpha{}  
Class Delta extends Beta{}  
public static void main(String [] args)  
{  
    Beta x = new Beta();  
    // insert code here  
}
```

What code will cause java.lang.ClassCastException?

Foo f = (Delta)x; (Because Foo is not related to Delta)

2.

```
public class Batman{  
    int square=81;  
    public static void main (String [] args)  
    {  
        new Batman().go();  
    }  
    void go(){  
        incr (++square);  
        System.out.println(square);  
    }  
    void incr (int square){ square+=10; }  
}
```

Output: 82

3.

```
public class Pass{  
    public static void main (String [] args)  
    {  
        int x=5;  
        Pass p = new Pass();  
        p.doStuff(x); System.out.println("main x" + x);  
        void doStuff (int x) // main x=5  
        {  
            System.out.println("doStuff x=" + x++);  
        } // doStuff x=5.  
    }
```

4. Class Thingy { Meter m = new Meter(); }
 Class Component { void go() { System.out("C"); } }
 Class Meter extends Component { void go() { System.out("m"); } }
 Class DeluxeThingy extends Thingy {
 public static void main (String [] args)
 { DeluxeThingy dt = new DeluxeThingy();
 dt.m.go();
 Thingy t = new DeluxeThingy();
 t.m.go();
 } } Output: mm

5. public static void main (String [] args)
 { Integer i = new Integer (1) + new Integer (2);
 switch (i) {
 case 3: System.out("three"); break;
 default: System.out("other"); break; } }
 Output: three

6. Class Mammal {}
 Class Raccoon extends Mammal {} // Raccoon IS-A mammal
 Mammal m = new Mammal(); } Raccoon HAS-A mammal
 Class BabyRaccoon extends Mammal {}
 / BabyRaccoon IS-A mammal, BabyRaccoon HAS-A mammal
 does not have.

7. Public class Hello
 { String title;
 int value;
 public Hello()
 { title += "World"; } }
 public Hello (int value)
 { this.value = value;
 this.title = "Hello";
 Hello(); } }
 Hello c = new Hello();
 System.out(c.title);
 Constructor can't be called explicitly
 Compile Time Error