09/03/11

main()

main() :-

→ wheather the class contains main() or not & wheather the main() is properly declared or not, these checkings are not responsibilities of compiler. At runtime, JVM is responsible for these checking.

→ If the JVM unable to find required main() then we will get runtime Exception saying NoSuchMethodError: main.

Ex:-

```
class Test
{
}
```

compile Javac Test.java ✓

Run × Java Test → R.E:- NoSuchMethodError: main

→ JVM always searches for the main() with the following signature.

public static void main(String[] args)

- public → To call by JVM from any where
- static → without existing object also JVM has to call this method
- void → main method can't return anything to JVM
- main → name of method which is configured inside JVM
- String[] args → Command-line arguements.

→ If we are performining any change to the above Signature we will get Runtime Exception Saying " NoSuchMethodError: main ".

→ Any where the following changes are acceptable.

(1) we Can change the order of modifiers. i.e instead of public static weCan take static public.

(2) We Can declare String[] in any valid form

String[] args ✓

String []args ✓

String args[] ✓

(3) Instead of args we Can take any valid Java identifier.

(4) Instead of String[] we Can take Var-arg String parameter. is String...

main (String[] args) $\Longrightarrow$ main(String... args) ✓

(5) main() Can be declared with the following modifiers also

(i) final

(ii) Synchronized

(iii) Strictfp.

Ex1.

```
Class Test
{
 final static strictfp synchronized public void main(String... A)
 {
  S.o.pln(" Hai durga");
 }
}
```

Q) which of the following main() declarations are valid?

Ans:-
(i) public static int main(String[] args) ✗

(ii) static public void Main(String[] args) ✗

(iii) public synchronized strictfp final void main(String[] args) ✗

(iv) public final static void main(String. args) ✗

✓ (v) public strictfp synchronized static void main(String[] args)

Q) In which of the above cases we will get Compiletime Error.

Ans:-
No where, All cases will compile.

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

Ex:-
```
class P
{
 public static void main(String[] args)
 {
   S.opln("ILU durga s/w");
  }
}
class C extends P
{
}
```
```
javac P.java ✓
java P
o/p:- ILU durga S/w
java C
o/p:  ILU durga s/w
```

Ex 2:-
```
class P
{
 p.s.v.m(String[] args)
 {
  S.o.pln(" I Love");
  }
 }
 class C extends P
 {
  p.s.v.m(String[] args)
  {
   S.o.pln(" durga s/w");
  }
 }
```
```
Javac P.java
java P
o/p:- ILove
java C
o/p:- durga s/w.
```
→ It Seems to be overriding Concept is applicable for Static methods, but it's not overriding but it is Methodhidding.

→ Overloading concept is applicable for main() but JVM always Calls String[] ~~array~~ arguement method only. The other method we have to Call Explicitly.

ex:-
```
class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(" durga s/w");
 }
 p.s.v.m(int[] args)
 {
  S.o.pln(" is good");
 }
}
```
o/p:- durgas/w.

Q) Instead of main is it possible to Configure any other method as main method?

A) Yes, But inside JVM we have to Configure Some changes then it is possible.

Q) Explain about S.o.pln

A)

```
Class Test
{
  Static String name = "durga";
}
```

Test.Name.length()

- It is a Class-name
- Static variable of type String present in Test class
- it is a method present in String class

```
Class System
{
  Static PrintStream out;
}
```

System.out.println()

- It is a Class Name present in java.lang
- Static variable of type PrintStream present in System Class
- It is a method present in PrintStream class

10/03/11

Commandline Arguements

Commandline arguements :-

→ The arguements which are passing from Commandprompt are called Commandline arguements.

→ The main objective of Commandline arguements are we can customize the behaviour of the main().

Ex:-

$$\begin{tikzcd} \text{Java} & \text{Test} & X \arrow[dll] & Y \arrow[ddlll] & Z \arrow[dddllll] \\ \text{args[0]} \\ \text{args[1]} \\ \text{args[2]} \end{tikzcd}$$

args.length ⇒ 3 .

Ex(1) !-

```
class Test
{
 p.s.v.m(String[] args)
 {
  for(int i=0 ; i<=args.length ; i++)
  {
   S.o.pln (args[i]);
  }
 }
}
```

~~for{~~ o/p !-

```
Java Test ↵
 R.E !- AIOBE

Java Test X Y ↵
 X
 Y
R.E !- AIOBE
```

Ex(2):-

→ with in the main(), Commandline arguements are available in String form.

Ex:-
```
class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(args[0] + args[1]);
 }
}
```

```
Java Test 10 20
```

O/P:- 1020

→ Space is the Seperater B/w Commandline arguements, if the Command-Line arguments itself Contain Space then we should enclose with in doubleCodes (")

Ex:-
```
class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(args[0]);   Note Book
 }
}
```

```
Java Test "Note Book"
```

Ex(3):-
```
class Test
{
 p.s.v.m(String[] args)
 {
  String[] argh = {"A", "B"};
  args = argh;
  for(String S1 : args)
  {
                                S.o.pln(S1);
                              }
                            }
                          }
```

```
Java Test  x  y ↵
o/p  A
     B

Java Test  x  y  z ↵
o/p  A
     B

Java Test ↵
o/p  A
     B
```

<u>Note</u>: The maximum allowed no. of commandline arguments is <u>2147483647</u>, min. is '<u>0</u>'

Java Coding Standards

→ Whenever we are writing the code it is highly recommended to follow coding conventions the name of the method or class should reflect the purpose of functionality of that component.

```
Class A
{
public int m1(int x, int y)
{
  return x+y;
}
}
```

~~Ameerpet~~ Standard

```
package com.durgasoft.demo;
public class Calculator
{
 public static int sum(int number1,
                        int number2)
 {
   return number1+number2;
 }
}
```

~~Hitech~~-City

Coding Standards for Classes :-

→ Usually classnames are Nouns, should starts with uppercaseletter & if it contains multiple words every inner word should starts with uppercase letter

Ex:- Student
Customer
String
StringBuffer, } → Nouns

2) Coding Standards for Interfaces :-

→ Usually interface names are Adjectives should starts with UpperCase Letter & if it Contains multiple words every inner word should starts with UpperCase Letter.

Ex:- Runnable, Serializable, Cloneable, Movable. } Adjectives

Note :- Throwable is a Class but not interface. It acts as a root Class for all Java Exceptions & Errors.

3) Coding Standards for Methods :-

→ Usually method names are either Verbs or Verb noun Combination Should starts with Lower Case Letter & if it Contains multiple words "Every inner words should starts with Upper Case Letter". (camelCase).

Ex: run()
sleep()
eat()
init()
wait()
join()
⋮ } → Verbs

getName()
setSalary()
⋮ } Veb + noun

(4) Coding Standards for Variables :-

→ Usually the variable names are nouns should starts with LowerCase character & if it Contains multiple words, Every innerword should starts with uppercase character (CamelCase).

Ex! Name
roll no
Mobile Number
⋮

$\Big\} \longrightarrow$ nouns

⑧ Coding Standards for Constants:-

→ Usually The Constants are Nouns, Should Contain only UpperCase characters, If it Contains multiple words, These words are Seperated with "_" symbol.

→ We Can declare Constants by using Static & final modifiers.

ex:-

```
MAX_VALUE
MIN_VALUE
MAX_PRIORITY
MIN_PRIORITY
```

⑤ Java bean Coding Standards

→ A Java bean is a Simple java class with private properties & public getter & Setter methods.

ex:-

```
public class StudentBean
{
 private String name;
 public void setName(String Name)
 {
   this.name = name;
 }
 public String getName()
 {
   return name;
 }
}
```

→ Ends with Bean is not official Convention from SUN.

Syntax for setter method :-

→ The method name should be prefix with "Set". Compolsary the method should take some argument. return type should be void.

Syntax for getter method :-

→ the method name should be prefixed with "get".

→ It should be no arguement method.

→ returntype should not be void.

** Note :-

→ for the boolean property the getter method can be prefixed with either get or is. recommended to use "is"

Ex:-
```
private boolean empty;

✓ public boolean getEmpty()
  {
   return empty;
  }
✓ public boolean isEmpty()
  {
   return empty;
  }
```

① Coding Standards for Listeners:-

* To register a Listener :-

→ method name should be prefix with add,

→ after add what ever we are taking the argument should be Same.

eg:- ✓ ① public void addMyActionListener(MyActionListener l)

✗ ② public void registerMyActionListener(MyActionListener l)

✗ ③ public void add MyActionListener(Listener l)

To unregister a Listener:-

→ The rule is Same as above, Except method name Should be Prefix with remove .

eg:- ✓ ① public void removeMyActionListener(MyActionListener l)

✗ ② public void unregisterMyActionListener(MyActionListener l)

✗ ③ public void deleteMyActionListener(MyActionListener l)

✗ ④ public void removeMyActionListener(ActionListener l)

Note:-

In Javabean Coding standards & Listener Concept 1 compulsary.

Operators & Assignments

Kathy Sierra 1.6 book for SCJP.

Increment & Decrement Operators:-

$$\begin{tikzcd}
 & \text{Increment} \arrow[dl, dash] \arrow[dr, dash] & & & \text{Decrement} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{pre-increment} & & \text{post-increment} & \text{Pre-decrement} & & \text{post-decrement}
\end{tikzcd}$$

```
int x = ++y;     int x = y++;     int x = --y;     int x = y--;
```

Expression	Initial value of x	final value of x	final value of y
y = ++x;	4	5	5
y = x++;	4	5	4
y = --x;	4	3	3
y = x--;	4	3	4

i) We can apply increment and decrement only for variables but not for constant values.

```
  int x = 4;
X int y = ++4;        C.E: unexpected type
  S.o.pln(y);               found : Value ②
                            required : Variable ①
```

ii) Nesting of increment & decrement operators is not allowed otherwise we will get Compile time Error.

```
  int x = 4;
X int y = ++(++x);
  S.o.p(y);
```

after incre- it is -Constant Then

C.E: unexpected type

② found : value

① Required : variable

iii). We can't apply increment & decrement operators for the final variables.

Ex(1):- final int x = 4; X
x++;

Ex(2):- final int x = 4; X
x = 5

C.E:- Can't assign a value to final variable x.

iv). We can apply increment and Decrement operators for " Every primitive data type Except Boolean".

① ✓
```
double d = 10.5;
d++;
S.o.p(d); 11.5
```

② ✓
```
char ch = 'a';
ch++;
S.o.p(ch); // b.
```

③
```
  boolean b = true;
X       ++b;
  S.o.pln(b);
```

C.E:- operator ++ can't applied to boolean.

✓④
```
int x = 10;
x++;
S.o.pln(x); 11
```

Difference b/w b++ & b=b+1 :-

① byte b=10; ✓
b++;
S.o.pln(b); 11

② byte b=10 ✗
b=b+1;
S.o.p(b);

C.E : possible loss of precission
found : int
Required : byte

Exp:- max(int, typeof a, typeof b)
max(int, byte, int)
Res:- int

③ byte b=10 ✓
b = (byte) (b+1)
S.o.pln(b);// 11

④
byte a =10;
byte b =20;
byte c =a+b;
S.o.pln(c);

C.E: PLP
f = int
R = byte

Explanation:-

Max(int, typeof a, typeof b)

Max(int, byte, byte)

result is of type: int

∴ found is int but
required is byte

(+, -, *, %, /)

→ whenever we are performing any arithmetic operation between two variables a & b the result type is always,

$$\boxed{\text{Max (int , typeof a, typeof b)}}$$

byte b=10;
b =(byte) (b+1);
S.o.p(b);// 11

→ In the Case of Increment & decrement operators the required type casting (internal type casting) automatically performed by the Compiler.

$$\boxed{\text{byte b++ ;} \Longrightarrow \text{b = (byte) (b+1);}}$$

b++ ; ⟹ b = (type of b) (b+1);

Arithematic operators:-

→ The Arithmetic operations are (+ , - , * , / , %)

→ If we are applying any Arithematic operator b/w two variables a and b the result type is always.

$$\boxed{\text{Max (int, type of a, type of b)}}$$

byte + byte = int		
byte + short = int	S.o.pln (10 + 0.0); // 10.0	
int + long = long	S.o.pln ('a' + 'b') ; 195	
long + float = float	S.o.pln (100 + 'a') ; 197.	
double + char = double		
char + char = int		

Infinity:-

→ In the Case of integral arithematic (int, short, long, byte), there is no way to represent infinity. Hence, if the infinity is the result we will always get ArithematicException. (AE : I by zero)

Eg:- S.o.pln (10/0) ; R.E : AE: I by zero.

→ But in Case of floating point arithematic (float & double), there is always a way to represent infinity. for this float & Double Classes Contains the following two Constants.

Positive_Infinity = Infinity
Negative_Infinity = - Infinity

+ve-∞ = ∞
-ve-∞ = -∞

→ Hence, in the Case of ~~float~~ floating point Arithematic we won't get any Arithematic Exception.

Eg:- ① S.o.pln (10/0.0) ; Infinity
② S.o.pln (-10/0.0) ; -Infinity.

* NaN :- (Not a Number)

→ In integral arithematic there is no way to represent undefined results. Hence, if the result is undefined we will get A.E in Case of integral Arithematic.

Eg:- S.o.p(0/0) ; R.E: A.E: / by zero

→ But in Case of floating point Arithematic, there is a way to represent undefined results for this float & Double classes Contains NaN Constant.

→ Hence, Eventhough the result is undefined we won't get any Runtime Exception in floating point Arithematic.

Eg:- S.o.pln(0/0.0) ; NaN.

* S.o.p(0.0/0); NaN
* S.o.p(-0/0.0); NaN

Ex: * public static double sqrt(double d);

S.o.pln(math.sqrt(4)); //2.0

S.o.pln(math.sqrt(-4)); NaN.

→ for any x value including NaN the below Expressions always returns false, Except the (!=) Expression returns true.

$\boxed{X \;!=\; NaN \Rightarrow True}$

$\boxed{\text{at } x=10}$

S.o.p(10 > float.NaN); false

S.o.p(10 < float.NaN); false

S.o.p(10 == float.NaN); flase

~~floa~~ S.o.p(10 != Float.NaN); true.

S.o.p(float.NaN == float.NaN); false

S.o.p(float.NaN != float.NaN); True.

$$\left.\begin{array}{l} x > NaN \\ x >= NaN \\ x < NaN \\ x <= NaN \\ x == NaN \end{array}\right\} \text{false}$$

Conclusion about A.E (ArithmeticException):-

→ It is RuntimeException but not Compiletime Error.

→ possible only in Integral Arithematic but not floating point Arithmatic

(int, byte, short, char) (float, double)

→ The only operators which Cause A.E are / and %.

3. String Concatination Operator (+)

→ the only overloaded operator in Java is '+' operator.

→ Some times it acts as arithematic addition operator & Some time acts as String arithematic Operator (or) String Concatination operator.

Eg:- int a = 10, b = 20, c = 30;

String d = "shanth";

S.o.p(a+b+c+d); 60 shanth

S.o.p(a+b+d+c); 30shanth30

S.o.p(d+a+b+c); shanth102030

S.o.p(a+d+b+c); 10shanth2030.

$\underline{d+a}+b+c$
$\underline{shanth10}+\underline{b}+c$
$shanth1020+c$
$shanth102030$

→ If at least one operand is String type then '+' operator acts as Concatination, otherwise (if both are number type) '+' acts as arithematic operator.

Here S.o.p() is evaluated from Left to Right.

Eg:- int a=10, b=20;

String c = "shanth";

✗ $a = \underbrace{(b+c)}$ → total String C.E:- Incompatible type : found : string
Required : int

✓ $\underset{\text{String}}{c} = \underbrace{a+c}_{\text{total String}};$

✓ $\underset{\text{int}}{b} = \underbrace{a+b}_{\text{int}};$

✗ c = a+b; C.E:- InCompatible type:

found : int

Required : String.

A=65, a=97

Relational Operators

These are >, <, >=, <=

1) We can apply Relational operators for Every primitive datatype Except boolean.

Eg:-

1)	10 > 20	false	✓
2)	'a' < 'b'	True	✓
3)	10 >= 10.0	True	✓
4)	'a' < 125	True	✓

5) true <= true

6) true < false

CE:- operator <= Can't be applied to boolean, boolean

2) We Can't apply relational operators for the object types.

Eg:- 1) "shanth" < "shanth" ✗ 2) "durga" < "durga123" ✗

CE: operator < can't be applied to String, String.

3) Nesting of Relational operators we are not allowed to apply.

Eg:- ✓ S.o.p (10 < 20);

✗ S.o.p $(\underbrace{10<20}_{\text{boolean}}<30)$

CE!- operator < Can't be applied to boolean.

Eg:-

```
String s1 = new String("durga");
String s2 = new String("durga");
S.o.pln(s1==s2); false (reference)
S.o.pln(s1.equals(s2)); true (Content)
```

$$\begin{tikzcd} S_1 \arrow[r] & \boxed{\text{durga}} \\ S_2 \arrow[r] & \boxed{\text{durga}} \end{tikzcd}$$

Equality Operators (==, !=)

→ These are ==, !=

*→ We can apply Equality operators for Every primitive type including boolean types.

Eg:-

		o/p
✓ ①	10 = = 10.0	T ✓
✓ 2)	'a' = = 97	T ✓
✓ 3)	true = = false	F ✓
✓ 4)	10.5 = = 12.3	F ✓

→ We can apply Equality operators even for object reference also.

→ For the two object references r1 and r2 & r1 == r2 returns True iff both r1 & r2 are pointing to the same object.

i.e, Equality operator (==) is always ment for reference/address Comparison

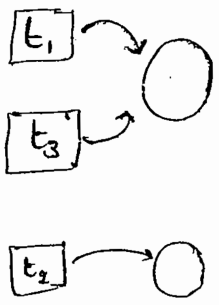
Ex①: Thread t1 = new Thread();

Thread t2 = new Thread();

Thread t3 = t1;

✗ S.o.p (t1 = = t2) ; False

✓ S.o.p (t1 = = t3) ; True



*→ To apply Equality Operators b/w the object references compulsory there should be some relationship b/w arguement types.

[either parent to child (or) child to parent (or) Same type] otherwise we will get CE: InComparable type]

Eg:-(2):- object o₁ = new Object(); because object is Super class

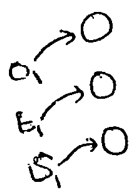
Thread t₁ = new Thread();

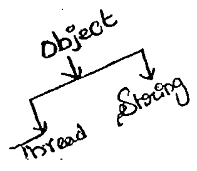
String s₁ = new String("shanth");

S.o.p(t₁ = = s₁); CE:- InComparable types Thread & java.lang.String

S.o.p(t₁ = = o₁); F

S.o.p(s₁ = = o₁); F





→ for any object reference r, if r is pointing to any object

$\boxed{r == null \text{ is always false}}$, otherwise r contains null value

→ So, $\boxed{null == null \text{ is always True.}}$

Note:-

* In General, == operator ment for reference Comparision where as .equals() method ment for Content Comparision.

Instanceof operator (instanceof) ✓

↳ By using this operator we can check, whether the given object is ~~of~~ a particular type or not.

Syn:- $\boxed{r \text{ instanceof } X}$

any reference type   class / interface.

instanceof
Hashtree
Strictfp

```
Ex:-  short s=15;
      Boolean b;
      b = (s instanceof Short)
      b = (s instanceof Number)
```

Eg:- ** 1) Thread t = new thread()

$$\begin{tikzcd} \text{object} & & \text{Runnable} \\ & \text{Thread} \arrow[ul, "\text{child class of}"] \arrow[ur, "\text{implements}"'] & \end{tikzcd}$$

✓ S.o.p(t instanceOf Thread) ; True

✓ S.o.p(t instanceOf Object) ; True

✓ S.o.p(t instanceOf Runnable) ; True

↳ To use instanceOf operator, compulsary there should be some relationship b/w argument type, otherwise we will get compile-time Error saying Inconvertable type.

Eg:- 2) Thread t = new thread();

S.o.p(t instanceof String); C.E:-
Inconvertable type
found : Thread
Required : String

↳ whenever we are checking parent object is of child type then we will get false as output.

Object o = new ~~Object()~~; Integer(10);

✓ S.o.p(o instanceOf String) ; false

↳ for any class ~~or~~ interface X, null instanceof X always returns "false".

✓ S.o.p(null instanceOF String) ; false.

```
Eg:- Iterator itr = l.iterator();
     while (itr.hasnext())
     {
         Object o = itr.next();
         if (o instanceOf Student)
         { Apply Student related function }
         else if(o instanceOf Cu)
         { Apply Customer related
```

Bit-wise Operators :-

(1) & → AND → if Both operands (or) arguements are True then Result is True

(2) | → OR → if atleast 1 operand is T " " T

(3) ^ → X-OR → if Both operands are different " " T

Eg:- S.o.pln(T&T); T

S.o.pln(T|T); T

S.o.pln(T^T); F

Ex(1):- S.o.pln(4&5); 4 →
$$\begin{array}{r} 100 \\ 101 \\ \hline 100 \\ \hline \end{array} = 4$$

S.o.pln(4|5); 5 →
$$\begin{array}{r} 100 \\ 101 \\ \hline 101 \\ \hline \end{array} = 5$$

S.o.pln(4^5); 1 →
$$\begin{array}{r} 100 \\ 101 \\ \hline 001 \\ \hline \end{array} = 1$$

→ We Can apply these Operators Even for integral data-types also.

also.

Ex:- (1) S.o.pln(4&5); 4

(2) S.o.pln(4|5); 5

(3) S.o.pln(4^5); 1

Bitwise Complement Operator ($\sim$) :- (tilde)

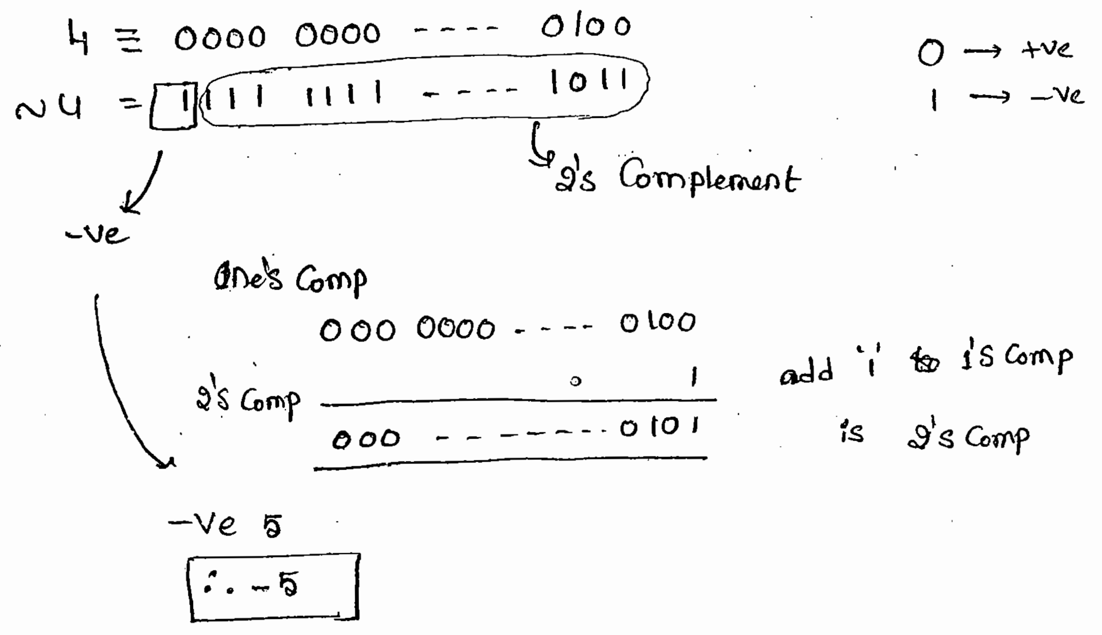
S.o.pln($\sim$T); CE: operator $\sim$ can't be applied to boolean

(1) We can apply Bitwise Complement Operator only for integral types, but not for boolean type.

Ex:- 1) S.o.pln($\sim$True);

C.E: operator $\sim$ can't be applied to boolean.

✓ 2) S.o.pln($\sim$4); -5



Note:

→ the most significant bit represents sign bit. 0 means +ve no, 1 means -ve no.

→ +ve no. will be represented directly in the memory. where as -ve no's will be represented in 2's Complement form.

Boolean Complement Operator (!) :-

→ we can apply these operator only for Boolean type but not for integral types.

Ex:- (1) S.o.p(!4);

C.E: operator ! can't be applied to int.

(2) S.o.p(! False); True

(3) S.o.p(! True); False

Summary:-

&, |, ^ ⇒ we can apply for both integral & boolean types.

~ ⇒ we can apply only for integral types but not for boolean types.

! ⇒ we can apply only for boolean types but not for integral types.

double AND
double OR

** Short-Circuit Operators (&&, ||)

1) We Can use these operators Just to improve performance of the System.

2) these are Exactly Same as normal bitwise operators &, | Except the following difference.

&, |	&&, ||
1. Both operands should be Evaluated always.	1. 2nd operand Evaluation is optional.
2. Relatively Low-performance	2. Relatively High-performance.
3. Applicable for Both Boolean & Integral types	3. Applicable only for Boolean types.

Ex:-
```
if (~~~~~  &  ~~~~~)
     e1           e2
    10min  1min  10min
       \_________/
          21 min
{
   =
}
else
{
}  =
```

1) x && y ⇒ y will be Evaluated iff x is True.

2) x || y ⇒ y will be Evaluated iff x is false.

Ex:-

```
int x=10;
int y=15;
if (++x >10 & ++y<15)
{
  ++x;
}
else
{
  ++y;
}
S.o.pln(x+ "------"+y);
```

o/p:-

	x	y
&	11	17
|	12	16
||	12	15
&&	11	17

(9)

```
int x =10;
if ((++x <10) && (x/0 >10))
{
S.o.pln ("Hello");
}
else
{
  S.o.pln ("Hi");
}
```

Ans:

a) C.E

b) R.E : Arithematic Exception : 1 by Zero.

c) Hello

d) Hi

Note:

if we Replace && with &

then Result is (b), that is R.E.

a=97
A=65

TypeCast Operators:-

→ There are 2 types of primitive type Castings.

1. Implicit type Casting
2. Explicit type Casting.

Implicit TypeCasting:-

1) Compiler is the responsible to perform this typeCasting
2) this TypeCasting is required when ever we are assigning Smaller data type value to the bigger data type variable.
3) It is also known as " widening (or) upCasting ".
4) No loss of information in this type Casting.

→ the following are various possible implicit typeCasting

$$\begin{tikzcd}
\overset{1B}{\underset{1 \text{ to } 127}{\overset{8\text{-bits}}{\text{byte}}}} \arrow[r] & \overset{2B}{\text{short}} \arrow[dr] & & & & \\
& & \overset{4B}{\text{int}} \arrow[r] & \overset{8B}{\text{long}} \arrow[r] & \overset{4B}{\text{float}} \arrow[r] & \overset{8B}{\text{double}} \\
\underset{2B}{\text{char}} \arrow[urr] & & & & &
\end{tikzcd}$$

Ex(s):-

① double d=10; [Complier Converts ~~into~~ to double automatically]

✓ S.o.pln(d); 10.0

② int x ='a'; [Compiler Converts char to int automatically]

✓ S.o.pln(x);97

a=97, b=98 ---
A = 65, B=66, C =67,

2) Explicit type Casting :-

1) programmer is responsible to perform this TypeCasting

2) It is required when ever we are assigning bigger datatype value to the Smaller datatype variable.

3) It is also Known as " Narrowing or down Casting ".

4) There may be a chance of loss of information in this Type-Casting.

→ The following are various possible Conversions where Explicit typeCasting is required.

$$\begin{tikzcd}
byte & short \arrow[l] & & & \\
& & int \arrow[ul] \arrow[dll] & long \arrow[l] & float \arrow[l] & double \arrow[l] \\
char & & & & &
\end{tikzcd}$$

Ex!.

1) X | byte b = 130

C.E : possible loss of precission
found : int
Required : byte

2)
```
byte b = (byte) 130;
S.o.p(b); -126
```

→ when ever we are assigning Bigger datatype value to the Smaller data-type variable then the most Significant bit will be lossed.

① ✗ byte b = 130 ;

✓ byte b = (byte) 130 ;

$$\begin{array}{r|l} 2 & 130 \\ 2 & 65 - 0 \\ 2 & 32 - 1 \\ 2 & 16 - 0 \\ 2 & 8 - 0 \\ 2 & 4 - 0 \\ 2 & 2 - 0 \\ 2 & 1 - 0 \end{array}$$

130 ≡ 0000 - - - - - - - - - 10000010 (32-bits)

byte b ≡ 1 0000010 (8 bit)

-ve

2's Complement

0000010

1111110

```
 1111101
      1
 -------
 1111110
```

$= 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0$

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

~~+ve~~ 126

$\boxed{\therefore -126}$

② int i = 150 ;

short s = (short) i ;

S.o.pln(s) ; 150

150 ≡ 0000 - - - - - 0 10010110 (32 bits)

short s ≡ 0000 - - - 0 0010110 → 2 Bytes = short = 16-bits

+ve

don't apply 2's Comp.

∴ s = 150

③ int x = 150 ;

byte b = (byte) x ;

short s = (short) x ;

S.o.pln (b) ; -106

S.o.pln (x) ; 150

150 ≡ 0000 - - - 0 10010110

byte ~~x~~ = 1 0010110

-ve

2's Com

1101010

```
 1101001
       1
 -------
 1101010
```

$= 2 + 8 + 32 + 64 = 106$

$\boxed{\therefore -106}$

10/2/11

→ when ever we are assigning floating point datatype values to the integral data types by Explicit type Casting the digits after the decimal point will be lossed.

Ex:-

```
double d = 130.456;
int a = (int) d;
byte b = (byte) d;
S.o.pln (a);  130
S.o.pln (b);  -126
```

Assignment Operators :-

→ there are 3 types of assignment operators

1. Simple assignment operators
2. chained assignment operator
3. Compound assignment operator.

1. Simple assignment operator :-

Ex:- `int x = 10;`

2. chained assignment operator :-

Ex:-

```
int a, b, c, d;
a = b = c = d = 20;   | ✓
```

→ We Can't perform chained assignment at the time of declaration

Ex:- int a = b = c = d = 20 ; } X C.E

C.E: Can't find Symbol

Symbol : variable b

location : class Test

int a = b = c = d = 20 ;
      ^

(Same C.E. & d)

Ex:- ② int b, c, d;

a = b = c = d = 20 } ✓

3. Compound assignment operator :-

→ Some times we Can mix assignment operator with Some other operator to form Compound assignment operator.

Ex:- int a = 10 ;

a += 30 ;

S.o.pln (a); 40

a += 30

a = a + 30

a = 10 + 30

a = 40

→ The following are various possible Compound assignment Operators in Java.

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

(11)

* In Compound assignment operators the required typecasting will be performed automatically by the Compiler.

Ex ① ✗

```
byte b =10;
 b =b+1;
S.o.pln(b);
C.E: PLP
found : int
required : byte
   b = b+1;
        ^
```

✓

```
byte b =10;
  b++;
S.o.pln(b); 11
```

```
byte b =10
 b+=1 ;
S.o.pln(b) ; 11
```

✓

```
byte b =127;
  b +=3;
S.o.pln(b) ; -126
```

Ex ②:-

```
int a, b, c, d;
 a = b = c = d = 20 ;
a += b *= c += d /= 2;
S.o.pln(a+"----"+b+"----"+c+"-----"+d);
      620       600       30        10
```

Conditional Operator (?:)

→ The only ternary operator available in Java is a Ternary Operator (or) Conditional Operator.

a+b → binary operator
++a → unary "
(a+b)? a:b ; → ternary.

Ex!-

```
int a = 10 , b=20 ;
int x = (a>b) ? 40 : 50;
S.o.pln(x) ; 50
```

$a>b$ is T then 40
$a>b$ is F then 50

→ nesting of Conditional operator is possible.

Ex:-
```
int a=10, b=20;
int x = (a>50) ? 777 : ((b>100) ? 888 : 999));
               F                  F
S.o.pln(x) ; 999
```

Ex:-
```
int a=10 , b=20 ;
✓ | byte  c= (True) ? 40 : 50;
  | byte  c = (false) ? 40 : 50;

X | byte c = (a<b) ? 40 : 50 ;
  | byte c = (a>b) ? 40 : 50;
```

✓ a<12 T

X a<b X C.E
don't Compare these variables

C.E:- PLP
found : int
required : byte.

```
final int a=10, b=20 ;

✓ | byte c = (a<b) ? 40 :50;
  | byte  c = (a>b) ? 40 : 50 ;
```

New Operator :-

→ We Can use this Operator for Creation of objects.

→ In Java there is no Delete operator . because distraction of Useless object is responsibility of Garbage Collector.

[ ] operator :-

→ We Can use these Operator for declaring & Creating arrays.

Operator precidence :-

1. Unary operators :-

   [ ] , x++ , x--

   ++x , --x , ~ , !

   new , < type > (used to type cast)

2. Arithematic Operators :-

   * , / , %

   + , -

3. shift operator :-

   >>> , >> , <<

4. Comparision operator :-

   < , <= , > , >= , instanceOf

5. Equality operator :-

   == , !=

6. Bitwise operators :-

   &

   ^

   |

7. Short - Circuit operators :-

   &&

   ||

8. Conditional operators :-

   ?:

9. Assignment operators :-

   = , += , -= , ...

Evalution order of operands :-

→ There is no precidence for operands before applying any operator all operands will be evaluated from left to right .

Ex :-

```
class EvaluationOrderDemo
{
 p.s.v.m (String[] args)
 {
  S.o.p (m,(1) + m,(2) * m,(3) + m,(4) * m,(5)/m,(6));
 }
 p.s. int m,(int i)
 {
  S.o.pln (i);
  return i;
 }
}
```

o/p:-
10

$1 + \underline{2 * 3} + 4 * 5/6$

$1 + 6 + \underline{4 * 5}/6$

$1+6+20/6$

$1+6+3$

$7+3$

$=10$

Ex(2):-

```
class Test
{
 p.s.v.m (String[] args)
 {
  int x=10;
   x = ++x;
  S.o.pln(x); 11
 }
}
```

1st increment

2nd place init into x

```
int x=10;
 x = x++;
 S.o.pln(x); 10
```

1st place x=10

∴ x = 10++

↳ x=11

but last operation is

x=10

Ex(3):-

int x=0;

$(1+2)^3$

$x = \underbrace{++x}_{1} + \underbrace{\overset{①}{x++}}_{1} + \underbrace{\overset{②}{x++}}_{2} + \underbrace{++x}_{4};$

S.o.p(x); 8

$x = \cancel{0}\,\cancel{1}\,\cancel{2}\,\cancel{3}\,4$

x++ = 1

x++ = 2

3

4

Ex 4:-

int x=0;

x += ++x + x++;

S.o.pln(x); 2

x = x+ ++x + x++;

= 0 + 1 + 1

x = 2