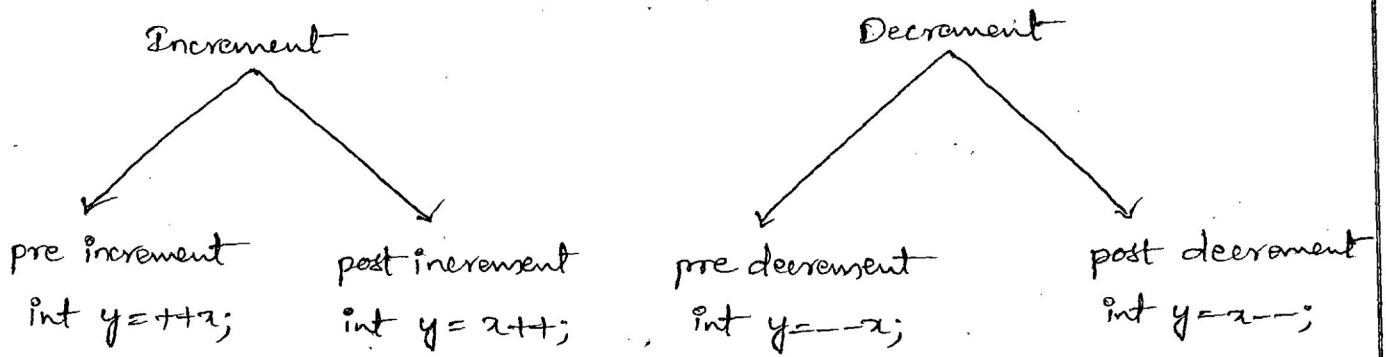
2. Operators & Assignments

1. Increment & Decrement Operators
2. Arithmetic operators
3. String concatenation operator
4. Relational operators
5. Equality operators
6. instanceof operator
7. Bitwise Operators
8. Short circuit operators
9. Type cast operator
10. Assignment Operators
11. Conditional Operator
12. new operator
13. [] operator
14. Operator precedence
15. Evaluation order of operands
16. new Vs newInstance()
17. instanceof Vs isInstance()
18. ClassNotFoundException Vs NoClassDefFoundError

1. Increment & Decrement Operators :-



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

→ We can apply increment & decrement operators only for variables but not for constant values o.w, we will get compile time error.

Ex:-
```
int x=10;
int y=++x;
S.o.p(y);
```
O/P : 11 ✓

```
int x=10;
int y=++10;
S.o.p(y);
```
CE: unexpected type
required : variable
found : value

→ Nesting of increment & decrement operators not applicable.

Ex:-
```
int x=10;
int y=++(++x);
S.o.p(y);
```
CE: unexpected type
required : variable
found : value

→ We can't apply increment & decrement operators for final variable.

Ex:
```
final int x=10;
x++;
S.o.p(x);
```
→ x=x+1;

CE: cannot assign a value to final variable x

```
final int x=4;
x=5
```

→ We can apply increment & decrement operators for every primitive type except boolean.

Ex:
```
int x=10;
x++;
S.o.p(x);
```
O/P : 11

```
char ch='a';
ch++;
S.o.p(ch);
```
O/P : b

```
double d=10.5
d++;
S.o.p(d);
```
O/P : 11.5

```
boolean b=true;
b++;
S.o.p(b);
```
CE: operator ++ cannot be applied to boolean

→ If we apply any arithmetic operator blw 2 variables a and b then the result type is always max(int, type of a, type of b).

byte → short
char → int → long → float → double

ex: byte a=10;
byte b=20;
byte c=a+b;
S.o.p(c);

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

byte c=(byte)(a+b); max(int, byte, byte) = int

→ If we perform explicit type casting then we won't get any CE.

ex: byte b=10;
b++;
S.o.p(b)
O/P: 11

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

max(int, byte, int) = int

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

b=(byte)(b+1);

→ In the increment & decrement operators, internal type casting will be performed automatically by the compiler.

b++;
⇓
b=(type of b)(b++);

ex: byte b=10;
b++;
S.o.p(b);
O/P: 11

b=(byte)(b++);

2. Arithmetic Operators (+, −, *, /, %) :—

→ If we apply any arithmetic operator blw 2 variables a and b then the result type is always max(int, type of a, type of b).

Ex: int + int = int
byte + byte = int
byte + short = int
short + short = int
int + long = long
long + float = float
float + double = double
char + char = int ⇒ S.o.p('a'+'b'); // O/P: 195 ✓
char + int = int ⇒ S.o.p('a'+1); // O/P: 98 ✓
char + double = double ⇒ S.o.p('a'+1.1); // O/P: 98.1 ✓

Infinity :—

→ In the Integral arithmetic (byte, short, int, long), there is no way to represent Infinity. Hence if infinity is the result we will get ArithmeticException in integral arithmetic.

Ex: S.o.p(10/0); // RE: ArithmeticException: / by zero.

→ But, in floating point arithmetic (float, double) there is a way to represent Infinity. For this, Float & Double classes contains the following 2 constants.

POSITIVE_INFINITY
NEGATIVE_INFINITY

→ Hence if the result is infinity then we won't get any AE in floating point arithmetic.

Ex: S.o.p(10/0.0) ⇒ O/P: Infinity
S.o.p(-10.0/0) ⇒ O/P: -Infinity

NaN (Not a Number):-

→ In integral arithmetic, there is no way to represent undefined results. Hence if the result is undefined then we will get AE in integral arithmetic.

Ex: S.o.p(0/0); ⇒ RE : AE : / by zero.

→ But in floating point arithmetic, there is a way to represent undefined results.

→ For this, Float & Double classes contain a constant NaN. Hence if the result is undefined then we won't get any AE in floating point arithmetic.

Ex: S.o.p(0/0.0); ⇒ o/p : NaN

S.o.p(-0.0/0); ⇒ o/p : NaN

→ For any x value, including NaN the following expressions return false.

x > NaN
x >= NaN
x < NaN ⇒ false
x <= NaN
x == NaN

→ For any x value, including NaN the following expression returns true

x != NaN ⇒ true

Ex: S.o.p(10 > Float.NaN); ⇒ o/p : false

S.o.p(10 >= Float.NaN); ⇒ o/p : false

S.o.p(10 < Float.NaN); ⇒ o/p : false

S.o.p(10 <= Float.NaN); ⇒ o/p : false

S.o.p(10 == Float.NaN); ⇒ o/p : false

S.o.p(Float.NaN == Float.NaN); ⇒ o/p : false

S.o.p(10 != Float.NaN); ⇒ o/p : true

S.o.p(Float.NaN != Float.NaN); ⇒ o/p : true

ArithmeticException
- It is RuntimeException, but not compile time error.
- It is possible only in integral arithmetic but not in floating point arithmetic.
- The only operators which cause AE are / (division) and % (modulus)

3. String concatenation operator (+):—

→ The only overloaded operator in Java is + operator. Sometimes it acts as arithmetic addition operator and some times it acts as string concatenation operator.

→ If atleast one argument is String type then + operator acts as concatenation and if both arguments are number type then + operator acts as arithmetic addition operator.

Ex: ① String a = "durga";
int b=10, c=20, d=30;
S.o.p (a+b+c+d); ⇒ o/p : durga102030
S.o.p (b+c+d+a); ⇒ o/p : 60durga
S.o.p (b+a+c+d); ⇒ o/p : 10durga2030
S.o.p (b+c+a+d); ⇒ o/p : 30durga30

(a+b)+c+d
"durga10"+c)+d
"durga1020"+d
durga102030 ✓

Ex ②: String a = "durga";
int b=10, c=20, d=30;

✗ a = b+c+d; → CE: incompatible types
found: int
required: java.lang.String

✓ a = a+b+c;

✗ b = a+c+d; → CE: incompatible types
found: j.l.String
required: int

✓ d = b+c+d;

4. Relational operators (<, <=, >, >=):—

→ We can apply relational operators for every primitive type except boolean.

Ex: S.o.p(10 > 20); ⇒ o/p : true

S.o.p(10 < 'a'); ⇒ o/p : true

S.o.p(10 < 10.5); ⇒ o/p : true

S.o.p('d' > 10.6); ⇒ o/p : true

S.o.p(true > false); → CE: operator > cannot be applied to boolean, boolean.

→ We cannot apply relational operators for object types.

Ex: S.o.p("durga" < "durga123");

→ CE: operator < cannot be applied to j.l.String, j.l.String.

→ Nesting of relational operators is not allowed.

Ex: S.o.p(10 < 20 < 30);

true < 30 → CE: operator < cannot be applied to boolean, int.

5. Equality operators (==, !=):—

→ We can apply equality operators for every primitive type including boolean also.

Ex: S.o.p(10 == 20); ⇒ o/p : false

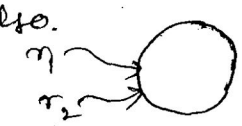
S.o.p('a' == 97.0); ⇒ o/p : true

S.o.p(10.5 == 10); ⇒ o/p : false

S.o.p(false == false); ⇒ o/p : true.

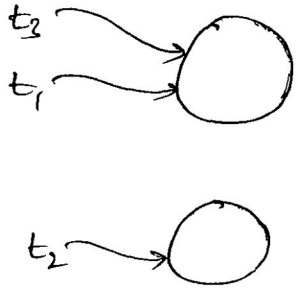
→ We can apply equality operators for object types also.

*** → For object references r_1 and r_2,



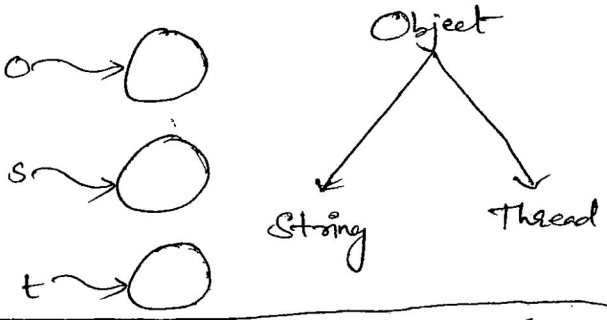
$r_1 == r_2$ returns true iff both are pointing to same object (reference/address comparision).

Ex: Thread t_1 = new Thread();
Thread t_2 = new Thread();
Thread t_1 = t_3;
S.o.p (t_1 == t_2); ⇒ o/p : false
S.o.p (t_1 == t_3); ⇒ o/p : true.



→ To use equality operators b/w object types compulsory there should be some relation b/w argument types (either child to parent or parent to child or same type) o.w. we will get compile time error saying incomparable types.

Ex: Object o = new Object();
String s = new String();
Thread t = new Thread();
S.o.p (o == s);
S.o.p (o == t);
S.o.p (s == t);
→ CE: incomparable types : j.l.String and j.l.Thread



*** Note:- For any object reference, r == null is always false.
But, null == null is always true.

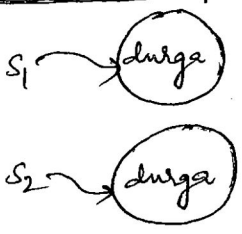
Ex: String s = new String("durga");
S.o.p (s == null); ⇒ o/p : false
S.o.p (null == null); ⇒ o/p : true

String s = null;
S.o.p (s == null); ⇒ o/p : true

*** Difference b/w == operator and .equals() method :-

→ In general, we can use == operator for reference comparision (address comparision) and .equals() method meant for content comparision.

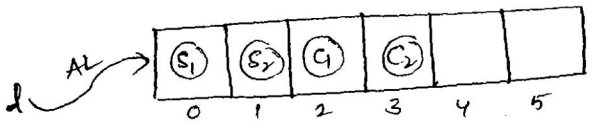
Ex: String s_1 = new String("durga");
String s_2 = new String("durga");
S.o.p (s_1 == s_2); ⇒ o/p : false
S.o.p (s_1.equals(s_2)); ⇒ o/p : true



6. instanceof operator :—

→ we can use instanceof operator to check whether the given object is a particular type or not.

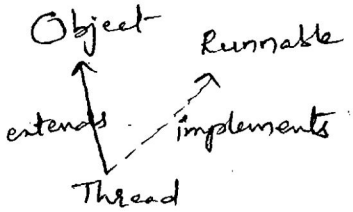
Ex:
```
Object o = l.get(0);
if(o instanceof Student)
{
    Student s = (Student)o;
    // perform student specific f'ality
}
else if(o instanceof Customer)
{
    Customer c = (Customer)o;
    // perform customer specific f'ality
}
```



Syntax:- `r instanceof X`

r → Object reference; X → class / interface

Ex:
```
Thread t = new Thread();
S.o.p(t instanceof Thread); ⇒ O/P : true
S.o.p(t instanceof Object); ⇒ O/P : true
S.o.p(t instanceof Runnable); ⇒ O/P : true
```



→ To use instanceof operator compulsory there should be some relation b/w argument types (either parent to child or child to parent or same type) o.w. we will get compile time error saying inconvertible types.

Ex:
```
Thread t = new Thread();
S.o.p(t instanceof String); → CE: inconvertible types
                              found: j.l.Thread
                              required: j.l.String
```

→ If we are checking parent object is of child type or not by using instanceof operator then we will get false as output.

Ex:① Object o=new Object();
S.o.p(o instanceof String); ⇒ o/p: false.

Ex:② Object o=new String("durga");
S.o.p(o instanceof String); ⇒ o/p: true

*** → For any class or interface,

null instanceof X is always false.

Ex: S.o.p(null instanceof String); ⇒ o/p: false
S.o.p(null instanceof Object); ⇒ o/p: false.

7. Bitwise Operators :—

& → AND ⇒ if both arguments are true then only result is true.
| → OR ⇒ if atleast one argument is true then only result is true.
^ → X-OR ⇒ if both arguments are different then only result is true.

Ex: S.o.p(true & false); ⇒ o/p: false
S.o.p(true | false); ⇒ o/p: true
S.o.p(true ^ false); ⇒ o/p: true

→ We can apply these operators even for integral datatypes also.

Ex: S.o.p(4 & 5); ⇒ o/p: 4
S.o.p(4|5); ⇒ o/p: 5
S.o.p(4^5); ⇒ o/p: 1

4 = 1 0 0	1 0 0	1 0 0
5 = 1 0 1	1 0 1	1 0 1
(&) 1 0 0	(1) 1 0 1	(^) 0 0 1
= 4	= 5	= 1

Bitwise Complement Operator (~):—

→ We can apply this operator only for integral types, but not for boolean type.

Ex: ① S.o.p (~true); ⇒ CE: operator ~ cannot be applied to boolean.

Ex ②: S.o.p (~4); ⇒ O/P: −5

4 = 0 0 0 0 ..... 0 1 0 0 (32-bit format)

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

(1 → −ve)

1's complement ⇒ 0 0 0 0 ..... 0 1 0 0

2's complement ⇒ 1

0 0 0 0 ..... 0 1 0 1 = 5

Note:- The most significant bit (MSB) acts as sign bit.

0 means +ve number

1 means −ve number

→ Positive numbers will be represented in the memory where as negative numbers will be represented in 2's complement form.

Boolean Complement Operator (!):—

→ We can apply this operator only for boolean types but not for integral types.

Ex:- S.o.p (~4); → CE: operator ! cannot be applied to int

S.o.p (~true); ⇒ O/P: false

(& | ^) → applicable for both boolean & integral types.

~ → applicable for integral types but not for boolean types.

! → applicable only for boolean types but not for integral types.

8. Short Circuit Operators (&&, ||):—

→ These are exactly same as Bitwise operators (&, |) except for the following differences.

&, |	&&, ||
1. Both arguments should be evaluated always.	1. second argument evaluation is optional.
2. Relatively performance is low.	2. Relatively performance is high..
3. Applicable for both boolean & integral types.	3. Applicable only for boolean but not for integral types.

x&&y ⇒ y will be evaluated iff x is true
i.e., if x is false then y won't be evaluated.

x||y ⇒ y will be evaluated iff x is false
i.e., if x is true then y won't be evaluated.

Ex①:
```
int x=10, y=15;
if (++x>10 & ++y<15)
{
  ++x;
}
else {
  ++y;
}
S.o.p(x+" "+y);
```
(| / && / ||)

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

Ex②:
```
int x=10
if (++x<10 && (x/0>10))
{
  S.o.p("Hello");
}
```

```
else {
    }
      S.o.p("Hi");
```

① AE: / by zero ② CE ③ Hello ✓④ Hi

Note:- If we replace && with & then we will get runtime exception saying AE: / by zero.

9. Type cast operator:—

→ There are 2 types of primitive type casting.

1. Implicit type casting
2. Explicit type casting

1. Implicit Type casting:—

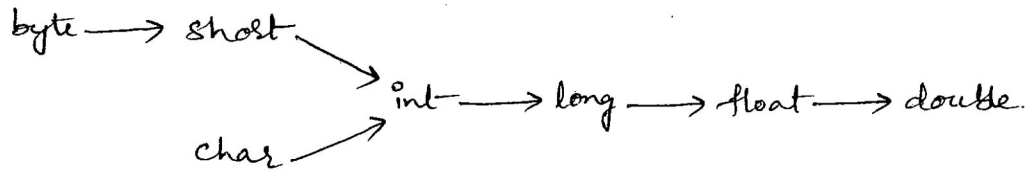
→ Compiler is responsible for this type casting.

→ It is required whenever we are assigning smaller data type value to the bigger data type variable.

→ It is also known as widening or upcasting.

→ There is no loss of information in this type casting.

→ The following are various possible places where implicit type casting is required.



Ex①: int x='a';
S.o.p(x); ⇒ o/p: 97

Ex②: double d=10;
S.o.p(d); ⇒ o/p: 10.0

2. Explicit Type casting:—

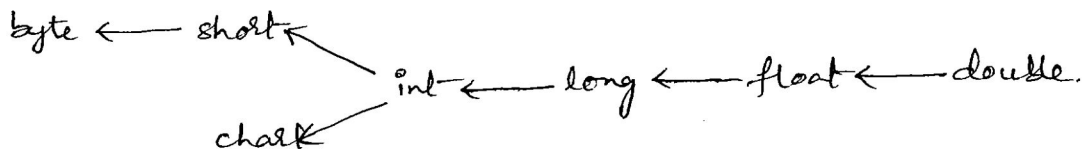
→ Programmer is responsible for this type casting.

→ It is required whenever we are assigning bigger data type value to the smaller data type variable.

→ It is also known as Narrowing or Down casting.

→ There may be a chance of loss of information in this type casting.

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



Ex: int x=130;

byte b=x; ⇒ CE: PLP
found: int
required: byte

byte b=(byte)x;

S.o.p(b); ⇒ O/P : -126.

x = 0 0 0 ------- 0 1 0 0 0 0 0 1 0 (32-bit format)

b = 1 0 0 0 0 0 1 0

-ve

1 1 1 1 1 0 1 (1's complement)

1 (2's complement)

1 1 1 1 1 1 0

$= 1\times2^6+1\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0 = 126$

∴ The final value is −126

Note:— Whenever we are assigning bigger data type value to the smaller data type variable by explicit type casting the most significant bits will be lost.

→ whenever we are assigning floating point values to the integral types by explicit type casting the digits after the decimal point will be lost.

Ex: double d = 130.456;
int a = (int) d;
S.o.p(a); ⇒ o/p: 130 ✓
byte b = (byte) d;
S.o.p(b); ⇒ o/p: -126

10. Assignment Operators :—

→ There are 3 types of assignment operators.

1. simple assignment
2. chained assignment
3. compound assignment

1. Simple assignment :—

Ex: int x = 10;

2. Chained assignment :—

Ex①: int a, b, c, d;
a = b = c = d = 20;
S.o.p(a+" "+b+" "+c+" "+d);
o/p: 20 20 20 20

Ex②: int a = b = c = d = 20; → CE: cannot find symbol
symbol: variable b
location: class Test

***
→ we can't use chained assignment at the time of declaration.

Ex①: int a = b = c = d = 20; // Invalid.

Ex②: int b, c, d;
int a = b = c = d = 20; } ✓

3. Compound assignment :—

→ Sometimes we can mix assignment operator with some another operator to form compound assignment.

Ex: int x=10;
x+=20;

→ The following is the list of all possible compound assignment operators in Java.

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

→ In case of compound assignment operators, implicit type casting will be performed automatically by the compiler.

Ex:
```
byte b=10;
b=b+1;
S.o.p(b);
```
CE: PLP
found: int
required: byte

```
byte b=10;
b++;
S.o.p(b);
```
b=(byte)(b+1);
O/P : 11

```
byte b=10;
b+=1;
S.o.p(b);
```
b=(byte)(b+1);
O/P : 11

```
byte b=127;
b+=3;
S.o.p(b);
```
O/P : -126

11. Conditional operator (?:) :—

→ The only ternary operator in Java is Conditional operator.

Ex: int x=(10>20)?30:40;
S.o.p(x); ⇒ O/P : 40

→ We can also perform nesting of conditional operator.

Ex: int x=(10>20)?30:((40>50)?60:70);
S.o.p(x) ⇒ O/P : 70.

12. new operator :—

→ We can use new operator to create objects.

Note :— There is no delete operator in Java because Garbage Collector is responsible to destroy useless objects.

13. [] operator :—

→ We can use this operator to declare and create Arrays.

14. Java Operator Precedence :—

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

15. Evaluation order of Java operands :—

→ There is no precedence for Java operands. Before applying any operator all operands will be evaluated from left to right.

Ex①:
```
class Test
{
  p s v m(-)
  {
    S.o.p(m1(1)+m1(2) * m1(3) / m1(4) * m1(5) + m1(6));
  }
  public static int m1(int i)
  {
    S.o.p(i);
    return i;
  }
}
```

O/P: 1 2 3 4 5 6 12

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

Ex②:
```
int a, b, c, d;
a = b = c = d = 20;        ①
a += b -= c *= d /= 2;
S.o.p(a+" "+b+" "+c+" "+d);
```
-160 -180 200 10

*

Ex③:
```
int i = 10;
i += ++i + i++ + ++i + i++;
S.o.p(i);
```
O/P: 58

i = i + ++i + i++ + ++i + i++;
= 10 + 11 + 11 + 13 + 13
= 58

Ex④:

```
byte b = 10;
b = ++b;
S.o.p(b);
```
O/P: 11

```
byte b = 10;
b = b++
S.o.p(b);
```
O/P: 10

***

16. new Vs newInstance():—

→ We can use new operator to create objects if we know the class name at the beginning.

Ex:
```
Test t = new Test();
Student s = new Student();
```

→ newInstance() is a method present in "Class" class which can be used to create object if we don't know the class name at the beginning and it is available dynamically at runtime.

new operator	newInstance() method
1. new is an operator.	1. newInstance() is method present in "Class" class.
2. It can be used to create an object if we know the class name at the beginning. Ex: Test t=new Test();	2. It can be used to create an object if we don't know the class name at the beginning & is available dynamically at runtime. Ex: Object o = Class.forName(args[0]).newInstance();
3. If the corresponding .class file is not available at runtime then we will get runtime Exception saying NoClassDefFoundError, which is unchecked.	3. At runtime if the corresponding .class file is not available then we will get runtime Exception saying ClassNotFoundException, which is checked.
4. To use new operator class is not required to contain no argument constructor.	4. To use newInstance() method compulsory class should contain no argument constructor o.w, we will get runtime Exception saying InstantiationException.

*** 17. ClassNotFoundException Vs NoClassDefFoundError :—

→ If hard-coded class name is not available at runtime then we will get runtime Exception saying NoClassDefFoundError, which is unchecked.

Ex: Test t=new Test();

At runtime, if Test.class file is not available then we will get runtime Exception saying NoClassDefFoundError.

→ If dynamically provided class name is not available at runtime then we will get runtime Exception saying ClassNotFoundException.

Ex: Object o = Class.forName("Test").newInstance();

If Test.class file is not available at runtime then we will get runtime Exception saying ClassNotFoundException.

*** 18. instanceof Vs isInstance() :—

→ We can use instanceof operator to check whether the given object is of particular type or not & the type is specified at beginning.

Ex: Thread t=new Thread();
S.o.p(t instanceof Runnable); ⇒ o/p : true.

→ We can use isInstance() method to check whether the given object is of particular type or not & we don't know the type at beginning & is specified dynamically at runtime.

Ex: Thread t=new Thread();
S.o.p(Class.forName(args[0]).isInstance(t));
C:>java Test Runnable ⇒ true
C:>java Test j.l.String ⇒ false

Note: - newInstance() is method equivalent of new operator.
isInstance() is method equivalent of instanceof operator.