Operators & Assignments

Topics

1. Increment and Decrement Operators
2. Arithmetic Operators (+, -, *, /, %)
3. Single concatenation operator (+)
4. Relational Operators (<, <=, >, >=)
5. Equality Operator (==,!=)
6. instanceof operator
7. Bitwise operator
8. Short-Circuit Operator (&&, ||)
9. Type-cast operator
10. Assignment operator
11. Conditional Operator (? :)
12. New operator
13. [] operator
14. Java operator precedence
15. Evaluation order of operands
16. New vs newInstance ()
17. Instanceof vs isInstance()
18. ClassNotFoundException vs NoClassDefFoundError

Increment and Decrement Operators

Increment Operators

1. Pre-increment (y = ++x;)
2. Post-increment (y = x++;)

Decrement Operators

1. Pre-decrement (y = --x;)
2. Post-decrement (y = x--;)

Expression	Initial value of x	Value of y	final value of x
y = ++x	10	11	11
y = x++	10	10	11
y = --x	10	9	9
y = x--	10	10	9

Case 1: **We can apply increment and decrement operators only for variables but not for constant values else we will get compile-time error.**

`int x = 10;`
`int y = ++x;`
`Sop(y);`
`//11`
	`int x = 10;`
	`int y = ++10;`
	`Sop(y);`
	CE: Unexpected type found: Value, Required: Variable

Case 2: **Nesting of increment and decrement operator is not allowed.**

Example:

```
int x = 10;
int y = ++(++x);
Sop(y);
```
//CE: Unexpected types, found: Value, required: Variable

Case 3: **For final Variable, we can't apply increment and decrement operators.**

Example:

```
final int x = 10;
x++;
Sop(x);
```
//CE: Cannot assign a value to final Variable x

Case 4: **We can apply increment and decrement operators for every primitive type expect boolean.**

`int x = 10;`
`x++;`
`Sop(x);`
//11
	`char ch = 'a';`
	`ch++;`
	`Sop(ch);`
	`//'b'`
		`double d = 10.5;`
		`x++;`
		`Sop(x);`
		//11.5
			`boolean b = true;`
			`b++;`
			`Sop(b);`
			// CE: Operator ++ can't be apply on boolean

Case 5: **Difference between b++ and b= b+1**

If we apply any arithmetic operator between two variable a and b, the result type is always **max (int, type of a, type of b).**

Example 1:

```
byte a = 10;
byte b = 20;
byte c = a+b; → max (int, byte, byte)
Sop(c);
```
// CE: Possible loss of precision, found: int, required: byte

byte c =(byte) a+b; **//Solved**

`byte b = 10;`
`b = b+1; → max (int, byte, int)`
`Sop (b);`
`// CE: Possible loss of precision, found: int, required: byte`
`b = (byte) (b+1); → is valid`
	`byte b = 10;`
	`b++; → b= (type of b) (b+1);`
	`Sop (b);`
	`//11`

In case of increment and decrement operator internal type-casting is performed internally.

Arithmetic Operators (+, -, *, /, %)

If we apply any arithmetic operator between two variables a and b, the result type is always
`max (int, type of a, type of b).`

`byte + byte = int`
`byte + short = int`
`short + short = int`
`byte + long = long`
`long + double = double`
`float + long = float`
`char +char = int`
`char + double = double`

$$\begin{tikzcd}
\text{byte} \arrow[r] & \text{short} \arrow[r] & \text{int} \arrow[r] & \text{long} \arrow[r] & \text{float} \arrow[r] & \text{double} \\
\text{char} \arrow[urr] & & & & &
\end{tikzcd}$$

Infinity

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

Example: `Sop (10/0);` **`RE: (AE:/by zero)`**

But in floating point arithmetic float and double there is a way to represent infinity. For this Float and Double Classes contains the following two constants **POSITIVE_INFINITY** and **NEGATIVE_INFINITY**. Hence, even though result is infinity, we won’t get any ArithmeticException in floating point Arithmetic.

```
Sop (10.0/0); → Infinity
Sop (-10.0/0); → -Infinity
Sop (0/0); → RE: AE:/by zero
Sop (0.0/0); → NaN (Not a number)
```

NaN (Not a Number)

In Integral arithmetic, there is no way in undefined result. Hence, if the result is undefined we will get **RE: AE:/by zero.**

But in Floating point arithmetic float and double there is a way to represent undefined results. For this Float and Double classes contains NaN constants. Hence, if the result is undefined we won’t get any ArithmeticException in Floating point Arithmetic.

```
Sop (0.0/0); → NaN
Sop (-0/0.0); → NaN
```

Q)
```
System.out.println (10 < Float.NaN); //false
System.out.println (10 <= Float.NaN); //false
System.out.println (10 > Float.NaN); //false
System.out.println (10 >= Float.NaN); //false
System.out.println (10 == Float.NaN); //false
System.out.println (Float.NaN == Float.NaN); //false
System.out.println (10 != Float.NaN); //true
System.out.println (Float.NaN != Float.NaN); //true
```
For any x value including NaN the following expressions return false.
```
X < NaN
X <= NaN
X > NaN
X >= NaN
X == NaN
```
For any x value including NaN the following expressions return true.
```
X != NaN
NaN != NaN
```
ArithmeticException

- RunTimeException.
- Possible only in integral arithmetic.
- Only operator which cause ArithmeticException are / and %.

String concatenation operator (+)

- The only overloaded operator in java is + operator.
- It acts as arithmetic addition operator as well as String Concatenation operator.
- If at least one argument is String type then + operator acts as concatenation operator and if both arguments are number type then + operator acts as arithmetic addition operator.

Q)
```
String a = “dhruv”; int b=10, c=20, d = 30;
Sop (a + b + c + d); //“dhruv102030”
Sop (b + c + d + a); //60dhruv
Sop (b + c + a + d); //30dhruv30
Sop (b + a + c + d); //10dhruv2030
```
Q) Consider the following declaration.
```
String a =”durga”; int b=10, c=20, d =30;
```
Q) Which of the following expressions are valid?
```
a= b + c + d; //CE: incompatible types, found: int, Required: Java.lang.String
a=a + b + c; //false
b = a + c + d; //CE: incompatible types, Found: Java.lang.String, Required: int
b=b + c + d; //false
```
Relational Operators (<, <=, >, >=)

1. We can apply relational operator for every primitive type except boolean.
```
Sop (10 < 20); //true
Sop (‘a’ < 10); //false
Sop (‘a’ <97.6); //true
Sop (‘a’ > ‘A’); //true
Sop (true > false); // CE: Operator > cannot be applied to boolean.
```

2. We can’t apply relational operator for object types.
   ```
   Sop (“durga123” > “durga”); //CE: Operator > cannot applied to
   Java.lang.String, Java.lang.String.
   ```
3. Nesting of relational operators is not allowed otherwise we will get compile-time error.
   ```
   Sop(10<20<30);//CE: Operator < can’t applied to boolean, int
   ```

Equality Operator (== , !=)

1. We can apply equality operator for every primitive type including boolean type also.
   ```
   Sop (10 == 20); //false
   Sop (‘a’ == ‘b’); //false
   Sop (‘a’ == 97.0); //true
   Sop (false == false); //true
   ```
2. We can apply equality operators for object types also. For object references r1, r2
   r1==r2 →true, iff both references pointing to the same object.
   Reference comparisons are address comparison.

   Example 1:
   ```
   Thread t1 = new Thread ();
   Thread t2 = new Thread ();
   Thread t3 = t1;
   Sop (t1 == t2); //false
   Sop (t1 == t3); //true
   ```
   Example 2:
   ```
   Thread t1 = new Thread ();
   Object o = new Object ();
   String s = new String (“dhruv”);
   Sop (t == o); //false
   Sop (o == s); //false
   Sop (s == t); //CE: incompatible type: J.l.String, J.l.Thread
   ```

If we apply equality operator for object types, then compulsory there should be some relation between argument types. (Either child to parent or parent to child or same type) otherwise we will get CompileTimeError saying incomparable types.

Difference between == operator and .equals () method

In general we can use == operator for reference comparison (address comparison) and .equals () method for content comparison.

Example:
```
String s1 = new String (“dhruv”);
String s2 = new String (“dhruv”);
System.out.println (s1 == s2); //false
System.out.println (s1.equals (s2)); //true
```

Note: for any object reference r, r == null is always false. But, null == null is always true.

Q)
```
String s1 = new String (“dhruv”);
System.out.println (s1 == null); //false
String s2 = null;
System.out.println (s2 == null); //true
System.out.println (null == null); //true
```

instanceof operator

We can use instance of operator to check whether the given object is of particular type or not.

Syntax: r instanceof x r → object reference , x → class/interface name

Example 1:
```
Thread t = new Thread ();
Sop (t instanceof Thread); //true
Sop (t instanceof Object); //true
Sop (t instanceof Runnable); //true
Sop (t instanceof String); //CE: inconvertible types, found:
Java.lang.Thread, required: Java.lang.String
```

To use instanceof operator compulsory there should be some relation between argument types, either child to parent, parent to child or same type else we will get CE: inconvertible types.

Note: For any class or interface X, null instance of x is always false.
```
(null instanceof x) ; → //false
Sop (null instanceof Thread); //false
Sop (null instanceof Runnable); //false
```

Bitwise Operator (&,|,^)

& → AND → returns true if and only if both arguments are true.

| → OR → returns true if and only if at least 1 argument is true.

^ → X-OR → returns true if and only if both arguments are different.

Example:
```
Sop (true & false); //false
Sop (true | false); //true
Sop (true ^ false); //true
Sop (4 & 5); //4
Sop (4 | 5); //5
Sop (4 ^ 5); //1
```

We can apply these operators for integral types also.

Bitwise Complement Operator (~)

We can apply this operator only for integral types.
```
Sop (~ true); //CE: operator ~ cannot be applied to boolean type
Sop (~4); //-5
```

Boolean complement operator (!)

We can apply boolean complement operator only on boolean data types.
```
Sop (!4); //CE: Operator ! cannot applied to int
```

- ***(&, |, ^) → Applicable for both boolean and integral types***
- ***~ → applicable for only integral type but not for boolean types***
- ***! → applicable only for boolean types***

Short-Circuit Operator (&&, ||)

&, |	**&&, ||**
Both arguments should be evaluated.	Second Statement are optional.
Relatively performance is low.	Relatively performance is high.
Applicable for both boolean and int.	Applicable for only boolean type.

Note: **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

Example1:
```
 int x =10, =15;
if(++x <10 || ++y>15)
        X++;
else
        Y++;
Sop (x+ "…"+y);
```

	X	**y**
&	11	17
&&	11	16
|	12	16
||	12	16

Example2:
```
int x =10;
if(++x <10 && x/0>10)
        Sop ("Hello");
else
        Sop ("Hi");
```
If we replace && with & then we will get RE: AE:/by zero

Type-cast operator

types of type-casting:

1. Implicit type-casting
2. Explicit typecasting

Implicit type-casting

- Compiler is responsible to perform.
- Whenever we are assigning smaller data-types value to bigger data-types variable. Implicit type-casting will be performed.
- It is also known as widening or up-casting.
- There is no loss of information in this type-casting.
- Various possible conversation where implicit type-casting will be performed.

$$\begin{tikzcd}
\textbf{byte} \arrow[dr] & & & & & \\
& \textbf{short} \arrow[dr] & & & & \\
& & \textbf{int} \arrow[r] & \textbf{long} \arrow[r] & \textbf{float} \arrow[r] & \textbf{double} \\
\textbf{char} \arrow[urr] & & & & &
\end{tikzcd}$$

Example1: `int x ='a';`
```
Sop(x); //Compiler converts char to int automatically by implicit typecasting
```
Example2: `double d =10;`
```
Sop (d); //Compiler converts int to double automatically by implicit typecasting
```

Explicit Type casting

- Programmer is responsible to perform explicit type-casing.
- Whenever we are assigning bigger data-type variable then explicit type-casting.
- It is also known as narrowing or down-casting.
- There may be a chance no loss of information.
- Various possible conversations where implicit type-casting will be performed.

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

L → R = Implicit Typecasting

R → L = Explicit Typecasting

Example1:

```
int x = 130;
byte b = x; //CE: Possible loss of precision, found: int, Required: byte
byte b = (byte) x;
Sop (b); //-126
```

Example2:

```
int x = 150;
short s = (short) x;
Sop (s); //150
byte (b) = (byte) x;
Sop (b); //-108
```

Example3:

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

Note: If we are assigning floating point value to the integral types by explicit type-casting the digit after decimal point will be lost.

Assignment operator

Three types of Assignment operators:

1. Simple Assignment operator

   int x =10;

2. Chained Assignment

   a= b =c =d =20;

3. Compound Assignment

   int a+ = 20; → a=a+20;

Sometimes Assignment operator mixed with some other operator. Such type of Assignment operator is called Compound Assignment operator. Possible Compound Assignment operator is:

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

In compound Assignment operator internal typecasting will be performed automatically.

`byte b =10;`
`b = b+1;`
`Sop (b);`
`CE: Possible loss of Precision, found: int, required: byte`
	`byte b =10;`
	`b++;`
	`Sop (b);`
	`b = (byte) (b+1);`
	`//11`
		`byte b =10;`
		`b + = 1;`
		`Sop (b);`
		`//11`

Q)

```
int a,b,c,d;
a=b=c=d=20;
a+=b-=c*=d/=2;
Sop (a+ … +b+ … +c+ … +d); //a = -160, b = -180, c = 200, d =10;
```

Conditional Operator (? :)

1. Only possible ternary operator in Java is conditional operator.
2. **Syntax:**

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

3. Nesting of conditional operator is possible.

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

New operator

Test t = new Test ();

new → to create object

Test () → to perform initialization

Note:

1. After creating an object, constructor will be executed to perform initialization of an object. Hence, constructor is not for creation of object and it is for initialization of an object.
2. In java we have only new keyword but not delete keyword because destruction of useless object is the responsibility of garbage collector.

[] operator

To declare and create an Array:

int[] x = new int[10];

Java operator precedence

1. **Unary operator**
   [], x++, x--
   ++x, --x, ~,!
   New, <type>
2. **Arithmetic operator**
   ***, /, %**
   +, -
3. **Shift operator**
   >>, >>>, <<

4. **Comparison operator**
   <, <=, >, >=, instanceof
5. **Equality operator**
   ==, !=
6. **Bitwise operator**
   &
   ^
   |
7. **Short circuit operator**
   &&
   ||
8. **Conditional operator**
   ?:
9. **Assignment operator**
   =, +=, -+, *=,…

Evaluation order of Java operands

In java, we have only operator precedence but not operand precedence before applying any operator all operands will be evaluated from Left to Right.

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

Output:

```
1                          1+2*3/4+5*6
2                          1+6/4+5*6
3                          1+1+5*6
4                          1+1+30
5                          2+30
6                          32
32
```

new vs newInstance ()

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

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

newInstance () is a method present in Class class. We can use newInstance () to create object if we don’t know class name at the beginning and it is available dynamically at run time.

```
public class Test {
	public static void main (String [] args) throws Exception {
```

```
                Object o = Class.forName(args[0]). newInstance();
                System.out.println("Object created for +:"o.getClass().getName());
        }
}
```

Java Test Student

Object created for: Student

Java Test Customer

Object created for: Student

Java Test Java.lang.String

Object created for: Java.lang.String

In the case of new operator based on our requirement, we can invoke any constructor.

Example: Test t = new Test ();

Test t1 = new Test (10);

Test t2 = new Test (“durga”);

But newInstance () method internally calls no-arg constructor. Hence to use newInstance () method compulsory, corresponding class should contain no-arg constructor otherwise we will get **RE: InstantiationException**.

While using new operator, at run-time if the corresponding .class file is not available then we will get **RE: NoClassDefFoundError: Test**

Example: `Test t = new Test ();`

At runtime, if Test.Class file is not available then we will get **RE: NoClassDefFoundError**

While using newInstance () method, at runtime if the corresponding .class file is not available then we will get **RE: ClassNotFoundException.Test123**

Example: `Object o = Class.forName(args[0].newInstance());`

Java Test Test123

At runtime if Test123.class file is not available then we will get **RE: ClassNotFoundException: Test123**

Difference between new and newInstance ()

New	newInstance ()
Operator in Java	Method present in Java.lang class
We can use new operator to create object if we know class name at the beginning.	We can use this method to create object if we don’t use class name at the beginning and it is available dynamically at run-time.
To use new operator class not require to contain no-argument constructor.	To use newInstance () method compulsory class should contain no-argument constructor, otherwise we will get **RE: InstantaneousException**
At runtime, if .class file is not available then we will get **RE: NoClassDefFoundError**, which is unchecked	At runtime, if .class file not available then we will get **RE: ClassNotFoundException**, which is checked.

Difference between ClassNotFoundException vs NoClassDefFoundError

ClassNotFoundException	**NoClassDefFoundError**
Checked Exception	Unchecked Exception
For dynamically provided class name at runtime, if the corresponding .class file is not available then we will get **RE: ClassNotFoundException**	For hard-coded class names at run-time if the corresponding .class file is not available then we will get **RE: NoClassDefFoundError**
`Object o = Class.forName(args[0]).newInstance());` Java Test Student At runtime if Student.Class file is not available then we will get **RE: ClassNotFoundException: Student**	`Test t = new Test();` At runtime if Test.Class file is not available then we will get **RE: NoClassDefFoundError :Test**

Difference between instanceof and isInstance ()

instanceof	**isInstance ()**
operator	A method in Java.lang class
To check whether the given object is of particular type or not and we know the type in beginning.	To check whether the given object is of particular type or not and we don't know type in beginning, we know it dynamically.

`Thread t = new Thread ()`
`Sop (t instanceof Runnable);`
`//True`
	`public class Test {`
	`      public static void main (String [] args) throws Exception {`
	`            Thread t = new Thread();`
	`      System.out.println(Class.forName(args[0]).isInstance(t));`
	`      }`
	`}`
	Java Test Runnable //True
	Java Test String //False