***-: Operators and Assignment:-***

Topics :-

1. Increment and decrement operator
2. Arithmetic Operator
3. String concatenation operator
4. Relational operator
5. Equality operator
6. instanceof operator
7. bitwise operator
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 od operands
16. new Vs newInstance( )
17. instanceof vs isInstance( )
18. ClassNotFoundException vs NoClassDefFoundError

**Increment and decrement operator(++ , --):**

We can apply increment and decrement operators only for variable but not for constant values. If we are trying to apply on constant value we will get compile time error.

Ex: int x =10; int x = 10;

int y = ++x; int y = ++10;

Sop(y); Sop(y);

O/p: 11 CE: unexpected type found: value required variable

Listing of increment and decrement operator not allowed.

Ex: int x =10;

Int y = ++(++x); // CE: unexpected type found: value required variable

Sop(y);

For final variable we can’t apply increment and decrement operators.

Ex: final int x =10;

++x; Sop(x); // CE: can’t assign a value to final variable

We can apply increment and decrement operator for every primitive type except Boolean.

Ex: public class Operator{

public static void main(String[] args){

int a = 10;

float b =12.6f;

double d = 10.5;

byte c = 23;

short x = 3;

long l = 2346765;

char ch = 'a';

++a; ++b; ++d; ++c; ++x; ++l; ++ch;

System.out.println(a+ " "+ b+ " "+d+ " "+c+ " "+x+ " "+l+ " "+ch );

}

}

o/p: 11 13.6 11.5 24 4 2346766 b

* 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)**.

Ex: byte a =10;

byte b = 20;

byte c = a + b; // CE: possible loss of precision found int required byte.

Sop(c);

byte c = (byte)(a + b); // give correct result by type casting

Ex: 2

byte b =10;

b =b + 1;

Sop(b); //CE: incompatible types: possible lossy conversion from int to byte

b =(byte)(b+1);

Here we have to do type casting explicitely. But in case of increment and decrement operators internal type casing will be performed automatically.

b++;

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

If we apply any arithmetic operator between two variable a and b then the result type is, **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

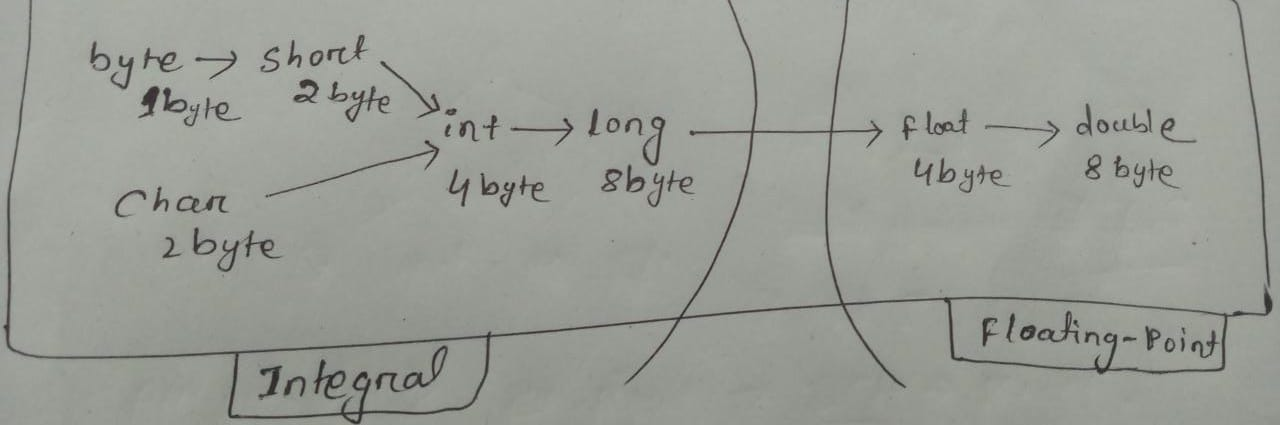
char + char = int

char + double = double

Ex:

Sop(‘a’ + ‘b’); // 97 + 98 = 195

Sop(‘a’ + .89); // 97 + .89 = 97.89



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

Ex: Sop( 10/ 0); // RE: ArithmeticException / by Zero

But in floating point arithmetic (float and double) there is a way to represent infinity. For this float and double classes following two constants.

POSITIVE\_INFINITY

NEGETIVE\_INFINITY

Hence even though result is infinity we won’t get any arithmetic exception in floating point arithmetic.

Sop(10/0.0); // O/p: Infinity

Sop(-10/0.0); // O/p: -Infinity

In integral arithmetic (byte, short, int, long) there is no way to represent undefined result. Hence if the result is undefined then we will get run time exception saying arithmetic exception.

Sop(0/0); //// RE: ArithmeticException / by Zero

But in floating point arithmetic there is a way to represent undefined results. For this float and double classes contains NaN constant. Hence the result is undefined we wont get any arithmetic exception.

Sop(0.0/0); // NaN(Not a Number)

Sop(-0.0/0); // NaN(Not a Number)

**Note :**

For any X value including NaN the following expressions returns false.

X < NaN

X > NaN

X <= NaN

X >= NaN

X == NaN

For nay X value including NaN the following expression returns true.

X != NaN

Ex:

Sop(10 < Float.NaN); //falsa

Sop(10 <= Float.NaN); //falsa

Sop(10 >= Float.NaN); //falsa

Sop(10 > Float.NaN); //falsa

Sop(10 == Float.NaN); //falsa

Sop(Float.NaN == Float.NaN); //falsa

Sop(10 != Float.NaN); // true

Sop(Float.NaN != Float.NaN); //true

**Arithmetic Exception:**

* It is Runtime exception but not compile time error.
* It is possible only in integral arithmetic but not in floating point arithmetic.
* The only operators which cause arithmetic exception are **/** and **%** .

**String concatenation Operator( + ):**

The only overloaded operator in java is + operator. Sometimes it acts as arithmetic operator and some time it acts 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.

String a = “durga”;

Int b =10,c =20,d =30;

Sop(a + b + c + d); // durga102030

Sop(b + c + d + a); // 60durga

Sop(b + c + a + d); // 30durga30

Sop(b + a + c + d); // 10durga2030

a = b + c + d; // CE : Incompatible type fount : int , required : java.lang.String

a = a + b + c; // valid

b = a + c + d; // CE : Incompatible type required : int , found : java.lang.String

b = b + c + d; // valid

**Relational Operators( < , <= , > , >= ):**

* We can relational for every primitive type except boolean.

Ex:

Sop(10 < 20); //true

Sop(‘a’ < 20); //false

Sop(‘a’ < 97.7); //true

Sop(‘a’ < ‘A’); //true

Sop(true < false); // CE: operator **>** can’t be applied boolean, boolean.

* We can’t apply relational operator for object types.

Ex: Sop(“durga123” > “durga”);

CE: operator > can’t be applied to java.lang.String , java.lang.String

* Nesting of relational operator is not allowed otherwise we will get compile time error.

Ex: Sop(10 < 20 < 30);

// 10 < 20 = true then true <20 we will get compile time error saying operator < can’t //be applied to Boolean, int

**Equality Operator( == , != ):**

* We can apply equality operator for every primitive type including Boolean also.

Ex;

Sop(10 == 20); // false

Sop(‘a’ == ‘b’); //false

Sop(‘a’ == 97.0); //true

Sop(false == false); // true

* We can apply equality operator for object type also.

For object reference r1, r2

r1 == r2 returns true if and only if both object are pointing to the same object(reference comparison or address comparison).

Ex: Thread t1 = new Thread();

Thread t2 = new Thread();

Thread t3 = t1;

Sop( t1 == t2 ); // false

Sop( t1 == t3); //true

* If we apply equality operator for object type then compulsory 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 incomparable type.

Ex: Thread t = new Thread();

Object o = new Object();

String s = new String();

Sop(t == o); // false

Sop( o == s); // false

Sop( s == t); // CE: incomparable type java.lang.String and java.lang.Thread

Difference between == operator and equals() method:-

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

Ex: String s1 = new String(“durga”);

String s2 = new String(“durga”);

Sop(s2 == s1); // false

Sop(s1.equale(s2)); // true

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

Ex: String s = new String(“Durga”);

Sop(s == null); // false

Ex; 2

String s =null;

Sop( s == null); // true

Sop( null == null); // true

**Instanceof operator:**

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

Ex: Lets say in a arraylist there are many objects we are accessing them by get() method.

Object o = l.get(0);

If(o instanceof Student){  
Student s = (Student) o;

// perform Student specific functionality.

}

else if(o instanceof Customer){

Customer c = (Customer)o;

//perform customer specific functionality

}

Syntax:

r instanceof X

r : object reference X : class or interface name

Ex;-1

Thread t = new Thread();

Sop(t instanceof Thread); // true

Sop(t instanceof Object ); // true

Sop(t instanceof Runnable ); // true

// Thread is the child of Object , Thread implements Runnable interface.

Ex:-2

To use instanceof compulsory there should be some relation between argument type(either child-parent or parent-child or same type) otherwise we will get compile time error saying inconvertible types.

Thread t = new Thread();

Sop(t instanceof String);

// CE: inconvertible type required: java.lang.String found: java.lang.Thread

**Note:** for any class or interface name X , null instanceof Thread it gives false.

Ex: Sop(null instanceof Thread); // false

**Bitwise Operators( & , | , ^ ):**

& ->AND => returns true if both arguments are true.

| -> OR => returns true if at least one argument is true.

^ -> X-OR => returns true iff both arguments are different.

Ex:

Sop(true & false); // false

Sop(true | false); //true

Sop(true ^ false); //true

We can apply these operators for integral types also.

Ex: Sop(4 & 5); //4 100 & 101 = 100

Sop(4 | 5); // 5 100 | 101 = 101

Sop(4 ^ 5); // 1 100 ^ 101 = 001

**Bitwise Complement operator ( ~ ) :**

We can apply this operator only for integral type but not for Boolean if we are trying to apply for boolean type then we will get compile time error.

Ex: Sop(~true); //CE: Operator ~ can’t be applied to Boolean

Sop(~4); // O/p: -5

**Note:** The most significant bit acts as sign bit. 0 means positive number 1 negative number. Positive numbers are directly represented in the memory. Where as negative numbers are represented indirectly in the memory in 2’s complement form.

**Boolean complement operator( ! ):**

We can apply this operator only for Boolean type not for integral type.

Ex: Sop( !4 ); // CE: Operator ! can’t be applied for int

Sop( !false ); // O/P:- true

**Short circuit operator( && , || ):**

These are exactly same as bitwise operator (& , |) except the following the following difference.

|  |  |
| --- | --- |
| & , | | && , || |
| 1. Here both arguments are should be evaluated always. 2. Relatively performance is low. 3. Applicable for both Boolean and integral type. | 1. Second argument evaluation is optional. 2. Relatively performance is high. 3. Applicable only for Boolean but not for integral 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.

Ex:

int x = 10;

if( ++x && (x/0 > 10)){

Sop(“Hello”);

}else{

Sop(“Hii”);

}

O/p: Hii

If we replace && with & then we will get RE: ArithmeticException / by 0

**Type cast Operator:**

There are two types of type casting. 1. Implicit type-casting

2. Explicit type-casting

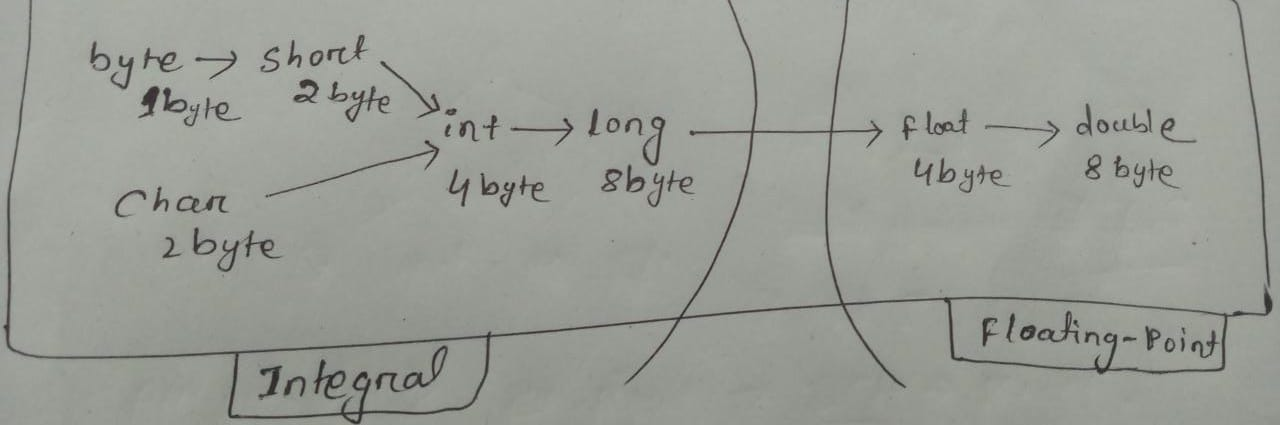
**Implicit type casting:**

Compiler s responsible for performing implicit type casting. Whenever we are assigning smaller data type value to bigger data type variable implicit type casting will be performed.

It is also known as **widening** or **upcasting.**

There is no loss of information in this type casting.

The following are various possible conversion where implicit type casing will be performed.



Ex: int x =’a’;

Sop(x); // O/p:- 97 compiler converts char to int by implicit typecasting

double d = 10;

Sop(d); // O/P: 10.0 compiler converts int to double by implicit typecasting

**Explicit Type-casting:**

* Programmer is responsible to perform explicit type-casting.
* Whenever we are assigning bigger data type value to smaller data type variable then we are required to perform explicit type conversion or type casting.
* It is also known as **narrowing** or **down casting**.
* In this type there may be a chance of loss of data.

byte 🡨 short 🡨 (char 🡨) int 🡨 long 🡨 float 🡨 double

L 🡪 R ==> Implicit type casting

R 🡪 L ==> Explicit Type casting

Ex: int x =130;

byte b = x; // CE: Possible loss of precision required : byte found : int

byte b =(byte) x;

Sop(b); // -126

Whenever we are assigning bigger datatype value to smaller datatype variable by explicit typecasting the most significant bits will be last we have to consider only least significant bits.

Explanation:

Int x = 130 🡺 0000000 ……….. 010000010 (32 bits)

byte b = (byte)x;🡺 10000010 (8 digit from right)

msb = 1 i.e. it is a negative number then 2’s complement of the left 7 bit is 1111110 which is 126 and by -ve sign the answer is -126.

Ex:2

int x = 150;

short s = (short) x;

Sop(s); // 150

byte b =(byte) x;

Sop(b); // -106

It is done in above process.

Ex:3

If we assign floating point values to the integral type by explicit type casting the digits after the decimal point will be lost.

double d = 130.567;

int x =(int)d; Sop(x); //130

byte b = (byte) d; Sop(b); // 130 in integer = -126 in byte

**Assignment operator:**

There are 3 types of assignment operator

1. Simple assignment operator:

Ex: int x =10;

1. Chained assignment operator:

Ex:

Int a,b,c,d;

a = b = c = d = 20;

Sop(a+“ ”+b + “ ”+c + “ ”+d ); // 20 20 20 20

Note: we can’t perform chained operation directly at the time of declaration.

int a = b = c = d = 20;

//CE: can’t find symbol , Symbol : variable b, Location class Test

//CE: can’t find symbol , Symbol : variable c Location class Test

//CE: can’t find symbol , Symbol : variable d Location class Test

Valid one is :

Int b,c,d;

int a = b = c = d = 20;

1. Compound assignment Operator:

Sometimes assignment operator mixed with some other operator. Such type of assignment operator are called compound assignment operator.

Ex: int a =10;

a += 20;

Sop(a); // 30

The following are all possible compound assignment operator in java

|  |  |  |
| --- | --- | --- |
| **+=**  **-=**  **\*=**  **/=**  **%=** | **&=**  **|=**  **^=** | **>>=**(right shift op)  **>>>=**(unsigned r.s op)  **<<=** |

In the case of compound assignment operator internal type casting performed automatically

Ex:.

byte b =10; byte b =10; byte b =10;

b = b + 1; b = b++ ; b += 1;

Sop(b); Sop(b); //11 Sop(b); // 11

CE: PLP f: int r: byte // b = (byte)(b+1); // b = (byte)(b+1);

Ex:

Int a, b, c, d; a = b = c = d = 20;

a += b -= c \*= d /= 2;

Sop(a +” ” +b+” ”+c+” ”+d ); //O/P:- -160 -180 200 10

**Conditional Operator( ? : ):**

The only possible ternary operator in java is conditional operator.

Syntax:

int x = (10 < 20) ? 30 : 40 ;

we can perform nesting of conditional operator also.

Int x = (10 > 20) ? 30 : ((40 > 50) ? 60: 70 ) ;

Sop(x); // 70

**new Operator:**

we can use **new**  operator to create object.

Ex: Test t = new Test();

Note: After creating an object constructor will executed to preform initialization of an object hence constructor is not for creation of object and it is for initialization of an object.

In java we have only new keyword but no delete keyword because destruction of useless object is the responsibility of garbage collector.

**[ ] Operator:**

We use this operator to declare and create arrays.

Ex: int[ ] x = new int[10];

**java Operator precedence:**

1. Unary operator
2. **[ ]** , **x++** , **x--**
3. **++x** , **--x** , **~** , **!**
4. **new** , **<type>**
5. Arithmetic Operator:
6. **\*** , **/**, **%**
7. **+** , **-**
8. Shift operator:

**>>** , **>>>** , **<<**

1. Comparison Operator:

**<** , **<=** , **>** , **>=** , **instanceof**

1. Equality operator:

**==** , **!=**

1. Bitwise Operator:
2. **&**
3. **^**
4. **|**
5. Short circuit operators:
6. **&&**
7. **||**
8. Conditional Operators:

**?:**

1. Assignment Operators:

**=** , **+=** , **-=** , **\*=** , **/=** , **%=**

**Evaluation order of java Operands:**

In java we have only operator precedence but not operand precedence. Before applying any operator all operands should be evaluated from left to right.

Ex:

public class Operator{

public static void main(String[] args){

System.out.println(m(1) + m(2) \* m(3) / m(4) + m(5) \* m(6));

}

public static int m(int i){

System.out.print(i + " ");

return i;

}

}

O/P: 1 2 3 4 5 6 32

Analysis: 1 + 2 \* 3 / 4 + 5 \* 6

= 1 + 6 / 4 + 5 \* 6

= 1 + 1 + 5 \* 6

= 1 + 1 +30

= 2 + 30 = 32

Question:

**new** vs **newInstance():**

we can us new operator to create an object if we know the class name at the bginning.

Ex:

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 runtime.

Ex:

public class Operator{

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

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

System.out.println("Object created for: " + o.getClass().getName());

}

}

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

Ex: Test t = new Test( );

Test t1 = new Test(10);

Test t2 = new Test(“Durga”);

* But newInstance() method internally calls no argument constructor. Hence to use newInstance() method corresponding class should contain no argument constructor otherwise we will get run time exception saying Instantiation exception.
* While using new operator at the run time the corresponding .class file is not available then we will runtime exception saying NoClassDefFoundError
* While using newInstance() at runtime if the corresponding .class file is not available then we will get runtime exception saying ClassNotFoundException

Ex: Object O = Class.ForName(args [0]).newInstance();

Java Test Test123

At runtime if test123.class file is not available then we will get runtime exception saying ClassNotFoundException : Test123

|  |  |
| --- | --- |
| **new** | **newInstance()** |
| It is an operator in java. | It is a method in java present in java.lang.Class |
| We can use new operator to ceate object if we know class name at the beginning. | We can use this method to create object if we don’t know class name at the beginning and it is available dynamically at runtime. |
| To use new operator class not required to contain no-arg constructor. | To use this method compulsory class should contain no-arg constructor otherwise we will get RE: InstantiationException |
| At runtime if class file not found then we will get RE: NoClassDefFoundError , which is unchecked. | At runtime if class file not found the we will get RE: ClassNotFoundException , Which is checked. |

**ClassNotFoundException vs NoClassDefFoundError :**

For hard-coded class name, at runtime if the corresponding .class file is not available then we will get runtime exception saying NoClassDefFoundError, which is unchecked.

For dynamically provided class name at the runtime if the corresponding .class file is not available then we will get runtime exception saying ClassNotFoundException, which is checked exception.

**instanceof** vs **isInstance()** :

instanceof is an operator in java we can use instance of to check whether a given object is particular type or not and we know the type at beginning.

Thread t = new Thread();

Sop(t instanceof Runnable);

Sop(t instanceof Object);

isInstance() is a method present in java.lang.Class . we can use isInstance() to check whether the given object is particular type or not and we don’t know the type at beginning and it is available dynamically at runtime.

class Test{

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

Thread t = new Thread();

Sop(Class.ForName(args[0]).isInstance(t));

}

}

Java Test Runnable

O/P: true

Java Test String

O/P: false

isInstance() is a method equivalent of instanceof operator.