***Operators and Assignments***

1. Increment and decrement operators
2. Arithmetic operators
3. String concatenation operators
4. Relational operators
5. Equality operator
6. Instanceof operator
7. Bitwise operators
8. Short circuit operators
9. Type cast operator
10. Assignment operators
11. Conditional operators
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
19. **Increment and decrement Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

int x = 10; int x = 10;

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

sop(y); sop(y);

Output:11 Output: Unresolved compilation problem:

Invalid argument to operation ++/--

* We can apply increment an decrement operators only for variables but not for constant values.If we are trying to apply for constants we will get compile time error.

**int** x = 10;

**int** y = ++10;

System.***out***.print(y);

Output: Unresolved compilation problem:

Invalid argument to operation ++/--

* Nesting of increment and decrement operators is not allowed

**int** x = 10;

**int** y = ++(x++);

System.***out***.print(y);

* For final variables we cannot apply increment and decrement operators.

**EG:**

**public** **class** Rough

{

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

{

**final** **int** x = 10;

x++;

System.***out***.print(x);

}

}

Output :

Unresolved compilation problem:

The final local variable x cannot be assigned. It must be blank and not using a compound assignment

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

1. **boolean** x = **true**;

x++;

System.***out***.print(x);

Output: Unresolved compilation problem:

Type mismatch: cannot convert from boolean to int

1. **char** x = **‘a’**;

x++;

System.***out***.print(x);

Output: b

1. **double** x = 10.5;

x++;

System.***out***.print(x);

Output: 11.5

1. **int** x = 10;

x++;

System.***out***.print(x);

Output: 11

* 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)**

EG 1.:

**byte** x = 10;

**byte** y = 20;

**byte** c = x + y;

System.***out***.print(c);

Output: Unresolved compilation problem:

Type mismatch: cannot convert from int to byte

EG 2.:

**byte** x = 10;

**byte** y = 20;

**byte** c = (**byte**)( x + y);

System.***out***.print(c);

Output: 30

* **Difference between b++ and b = b + 1**

EG :

**byte** x = 10;

x = x+1;

System.***out***.print(x);

Output : Unresolved compilation problem:

Type mismatch: cannot convert from int to byte

* b = (byte)b+1; ..valid
* b++ is valid as it performs internal typecasting
* In case of increment and decrement operators type casting is automatically performed.

B = (type of b)(b+1)

b++ = (byte) (b+1)

1. **Arithmetic Operators (+, -, \*, /, %)**

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

1b 2b

BYTE SHORT

4b 8b 4b 8b

INT LONG FLOAT DOUBLE

2b

CHAR

1. byte + byte = int
2. byte + short = int
3. short + short = int
4. byte + long = long
5. long + double = double
6. float + long = float
7. char + char = int
8. char + double = double

EG :

1. SOP( ‘a’ + ‘b’ ); = 97 +98 = 195
2. SOP( ‘a’ + 0.89); = 97 + 0.89 = 97.89

INFINITY:

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

EG:

SOP(10/0);

java.lang.ArithmeticException: / by zero

* But in floating point arithmetic (float , double) there is a way to represent infinity.

For this float and double classes contains the following two constants POSITIVE\_INFINITY, NEGATIVE\_INFINITY .Hence even though result is infinity we will not get any arithmetic exception in floating point arithmetic.

EG:

SOP(10/0.0);

🡪o/p = Infinity

SOP(-10.0/0)

🡪o/p = -Infinity

NaN (Not a Number)

* In integral arithmetic (byte, short, int, long) there is no way to represent undefined results, Hence if the result is undefined we will get runtime exception saying arithmetic exception

EG:

SOP(0/0)

* But in floating point arithmetic(float, double ) there is a way to represent undefined results.
* For this float and double classes contains NaN constant. Hence if the result is undefined we won’t get any arithmetic exception in floating point arithmetic.

EG:

SOP(0.0/0)

🡪o/p = NaN

SOP(-0.0/0)

🡪o/p = NaN

NOTE:

For any x value including NaN the following expressions return false

1. X < NaN
2. X <= NaN
3. X > NaN
4. X >= NaN
5. X == NaN

For any x value including NaN the following expression return true

1. X != NaN

EG:

1. SOP(10 < Float.NaN); False
2. SOP(10 <= Float.NaN); False
3. SOP(10 > Float.NaN); False
4. SOP(10 >= Float.NaN); False
5. SOP(10 == Float.NaN); False
6. SOP(Float.NaN == Float.NaN); False
7. SOP(10 != Float.NaN); **True**
8. SOP(Float.NaN != Float.NaN); **True**

ArithmeticException:

1. It is runtime exception and not compile time error.
2. It is possible only in integral arithmetic and not on floating point arithmetic
3. The only operators which cause arithmetic exception are / and % (Division and modulus)
4. **String Concatenation Operator (+)**

The only overloaded operator in java is + operator.

Sometimes it acts as arithmetic addition operator and sometimes 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.

EG: String a = “durga”;

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

1. SOP(a + b + c + d)

a + b = “durga10”

“durga10” + c = “durga1020”

“durga1020” + d = “durga102030”

1. SOP(b + c + d + a)

“60durga”

1. SOP(b + c + a + d)

“30durga30”

1. SOP(b + a + c + d)

“10durga2030”

Consider the following declarations:

String a = “durga”;

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

Which of the following expressions are valid?

1. a = b + c + d;

Invalid: Unresolved compilation problem:

Type mismatch: cannot convert from int to String

1. a = a + b + c;

Valid

1. b = a + c + d;

Invalid: Unresolved compilation problem:

Type mismatch: cannot convert from String to int

1. b = b + c + d;

Valid

1. **Relational Operator (<, <=, >, >=)**

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

1. SOP(10 < 20) true
2. SOP(‘a’ < 10) false
3. SOP(‘a’ < 97.6) true
4. SOP(‘a’ < ‘A’) true
5. SOP(true > false) Compile time error

Unresolved compilation problem:

The operator < is undefined for the argument type(s) boolean, boolean

We cannot apply relational operators for object types.

1. SOP(“durga123” > “durga”)

Unresolved compilation problem:

The operator > is undefined for the argument type(s) java.lang.String, java.lang.String

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

EG: SOP(10 < 20 < 30)

Unresolved compilation problem:

The operator < is undefined for the argument type(s) boolean, int

1. **Equality Operator (==, !=)**

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

EG:

1. SOP(10 == 20); false
2. SOP(‘a’ == ‘b’); false
3. SOP(‘a’ == 97.0); true
4. SOP(false = false) true



We can apply equality operators for object types also. For object references r1 , r2 r1 == r2 returns true if and only if both the references are pointing to same object(reference comparision or address comparision)



EG 1:



Thread t1 = new Thread();



Thread t2 = new Thread();



Thread t3 = t1;



SOP(t1 == t2); ..false



SOP(t1 == t3); ..true

EG 2:

If we apply equality operators 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 compile time error .

Thread t = **new** Thread();

Object o = **new** Object();

String s = **new** String("Durga");

System.***out***.print(t==o);

System.***out***.print(s==o);

System.***out***.print(t==s);

O/p: Unresolved compilation problem:

Incompatible operand types Thread and String

**Difference between == operator and .equals() method**

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

EG:

String s1 = **new** String("Durga");

String s2 = **new** String("Durga");

System.***out***.print(s1==s2);

System.***out***.print(s1.equals(s2));

O/p:

falsetrue

NOTE:

For any object reference r, r==null is always false.

But null == null is always true

EG:

1. String s = new String(“durga”);

SOP(s == null );

* False

1. String s = null;

SOP(s == null);

🡪true

1. SOP(null == null);

🡪true

1. **instanceof Operator**

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

Object o = p.get(0)

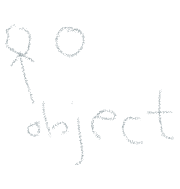
if(o instanceof student)

{



Student s = (Student)o;

//perform student specific functionality

}

Else if(o instanceof Customer)

{

Customer c = (Customer)o;

//perform Customer related functionality

}

**Syntax:**

r instanceof x

Object reference Class or interface name

EG 1:

Thread t = new Thread(); Object Runnable

SOP(t instanceof Thread); true

SOP(t instanceof Object); true is child of is implemented by

SOP(t instanceof Runnable); true Thread

EG 2:To use instanceof operator compulsorily there should be some relation between argument types(either child to parent or parent to child or same type) otherwise we will get compile time error .

Thread t = new Thread();

SOP(t instanceof String)

O/P: Unresolved compilation problem:

Incompatible conditional operand types Thread and String

NOTE:

For any class or interface x ,null instanceof x is always false.

SOP(null instanceof Thread); ..false

SOP(null instanceof String); ..false

1. **Bitwise Operator (&, |, ^)**

& - returns true iff both arguments are true.

| - returns true iff at least one argument return true.

^ - returns true iff both arguments are different.

EG:

SOP(true & false); ..false

SOP(true | false); ..true

SOP(true ^ false); ..true

We can apply these operators to integral types also.

EG:

1. SOP(4 & 5) ..o/p = 4

1 0 0

1 0 1

1 0 0 🡪4

1. SOP(4 | 5) ..o/p = 5

1 0 0

1 0 1

1 0 1 🡪5

1. SOP(4 ^ 5) ..o/p = 1

1 0 0

1 0 1

0 1 🡪1

**Bitwise complement Operator (~)**

We can apply this operator only for integral types but not for boolean type. If we try to apply for boolean type then we will get compile time error.

EG:

SOP(~4) ..o/p = -5

NOTE:

The most significant bit acts as sign bit

0 represents positive number and 1 represents negative number

Positive number is represented directly in the memory whereas negative numbers will be represented indirectly in the memory in 2’s complement form

**Boolean complement Operator (!)**

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

EG:

SOP(!4);

o/p: Unresolved compilation problem:

The operator ! is undefined for the argument type(s) int

SOP(!false)

o/p:true

|  |  |
| --- | --- |
| **&** | **Applicable for both boolean and integral types** |
| **|** |
| **^** |
| **~** | **Only applicable for integral type and not for boolean type** |
| **!** | **Only applicable for boolean type and not for integral type** |

1. **Short Circuit Operator(&& , ||)**

These are exactly same as bitwise operators(&, |) except the following differences

|  |  |
| --- | --- |
| **&, |** | **&&, ||** |
| Both arguments should be always evaluated | Second argument evaluation is optional |
| Relatively performance is low | Relatively performance is high |
| Applicable for both boolean and integral types | Applicable only for boolean types& |

NOTE:

1. X && Y:

Y will be evaluated iff X is true. If X is false then Y will not be evaluated.

1. X || Y:

Y will be evaluated iff X is false. If X is true then Y will not be evaluated.

EG 1 :

int x = 10 , y = 15;

if(++x < 10 op ++y>15)

{

x++;

}

else

{

y++;

}

SOP(x+”…….”+y);

|  |  |  |
| --- | --- | --- |
| op | x | Y |
| & | 11 | 17 |
| && | 11 | 16 |
| | | 12 | 16 |
| || | 12 | 16 |

EG 2:

int x = 10;

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

{

SOP(“Hello”);

}

else

{

SOP(“Hii”);

}

1. Compile time error
2. Runtime exception: arithmetic exception
3. Hello
4. Hii

If we replace && with & then we will get runtime exception saying : arithmetic exception / by 0

1. **Type-cast Operator**

There are two types of type casting.

1. Implicit typecasting
2. Explicit typecasting
3. Implicit type-casting
4. Compiler is responsible for implicit type-casting.
5. Whenever we are assigning smamller data type value to bigger data type variable implicit type-casting will be performed
6. It is also known as widening or upcasting
7. There is no loss of information in this type-casting
8. The following are various possible conversions where implicit type-casting will be performed

1b 2b

BYTE SHORT

4b 8b 4b 8b

INT LONG FLOAT DOUBLE

2b

CHAR

EG 1:

int x = ‘a’;

SOP(x);

🡪97 ..compiler converts char to int automatically by implicit type-casting

EG 2:

double d = 10;

SOP(d);

🡪10.0 ..compiler converts int to double automatically by implicit type-casting

1. Explicit type-casting
2. Programmer is responsible to perform explicit type-casting
3. Whenever we are assigning bigger data type value to smaller data type variable then explicit type-casting is required
4. It also known as narrowing or down-casting
5. There may be a chance of loss of information in this type casting.

The following are various possibilities where explicit type-casting is required.

1b 2b

BYTE SHORT

4b 8b 4b 8b

INT LONG FLOAT DOUBLE

2b

CHAR

1. **Assignment Operator**

There are three types of assignment operators.

1. Simple assignment

int x = 10;

1. Chained assignment

**EG 1:**

//int a, b, c, d;

int a = b = c = d = 20;

SOP(a+”…..”+b+”…..”+c+”…..”+d)

O/P: Unresolved compilation problems:

b cannot be resolved to a variable

c cannot be resolved to a variable

d cannot be resolved to a variable

We cannot perform chained assignment directly at the time of declaration.

**EG 2:**

int b, c, d;

int a = b = c = d = 20;

SOP(a+”…”+b+”…+c+”…”+d);

O/P:20…20…20…20

1. Compound assignment operators

Sometimes assignment operators mixed with some other operators such type of assignment operators are called compound assignment operators.

EG:

Int a = 10;

a+=20;

SOP(a); 🡪o/p = 30

The following are all possible compound assignment operators in java

1. += 6. &= 9. >>= ..right shift
2. -= 7. |= 10. >>>= ..unsigned right shift
3. \*= 8. ^= 11.<<= ..left shift
4. /=
5. %=

In case of compound assignment operators internal typecasting will be performed automatically.

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

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

SOP(b) SOP(b) SOP(b)

Compile time valid valid

Error

EG:

**int** b,c,d;

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

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

System.***out***.print(a+"..."+b+"...."+c+"..."+d);

O/P: -160...-180....200...10

1. **Conditional Operator**

The only possible ternary operator in java is conditional operator.

EG:

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

SOP(x);

O/p: 30

We can perform nesting of conditional operator also.

EG:

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

SOP(x);

o/p: 70

1. **new Operator**

we can use new operator to create object.

EG:

Test t = new Test()

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 object.
2. In java we have only new keyword but not delete keyword because destruction of usless objects is the responsibility of garbage collector.
3. **[] Operator**

We can use this operator to declare and create arrays .

EG:

Int[] x = new int[] 10;

***Java Operator Precedence:***

1. ***Unary operators:***

[] , x++, x- -

++x , - -x , ~ , !

new , <type> (typecast)

1. ***Arithmetic operators***

\* , / , %

***+ , -***

1. ***Shift operators***

>> , >>> , <<

1. ***Comparison operators***

< , <= , > , >= , instanceof

1. ***Equality operators***

== , !=

1. ***Bitwise operators***

&

|

^

1. ***Short Circuit operators***

&&

||

1. ***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 will be evaluated from left to right.

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

{

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

}

**public** **static** **int** m(**int** i)

{

System.***out***.println(i);

**return** i;

}

O/p:

1

2

3

4

5

6

32