**Operators & Assignments**

Topics

|  |  |
| --- | --- |
| **Module 2: Operators & Assignments** | |
| 1. | Increment & Decrement op |
| 2. | Arithmetic op |
| 3. | String concatenation op |
| 4. | Relational op (<, >, <=, =<) |
| 5. | Equality op |
| 6. | Instanceof op, isInstance() |
| 7. | Bitwise op |
| 8. | Short circuit op / logical |
| 9. | Typecast op |
| 10. | Assignments op |
| 11. | Conditional op / Ternary |
| 12. | new, newInstance(), [] operator |
| 13. | ClassNotFoundException Vs NoClassDefFoundError |
| 14. | Operator precedence & Evaluation order of operands |

**1. Increment & Decrement Operators**

**Increment**

Pre - Increment Post - Increment

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

**Decrement**

Pre - Decrement Post – Decrement

y = --x; y = x--;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Expression** | **Initial value of x** | **Value of y** | **Final value of x** |
| 1. | y = ++x | 10 | 11 | 11 |
| 2. | y = x++ | 10 | 10 | 11 |
| 3. | y = --x | 10 | 9 | 9 |
| 4. | y = x-- | 10 | 10 | 9 |

**Note**:

* Increment & Decrement operators can be applied only to the variables not to the constants.

e.g. int x = 10; int x = 10;

int y = ++x; //correct int y = ++10; // wrong

* Nesting of increment & decrement operator is not allowed.

e.g. int x = 10;

int y = ++(++x); // wrong

* We can apply increment & decrement operators to every primitive type except to Boolean.
* In case of increment & decrement operators, internal typecasting will be performed automatically.

i.e. b++ -> b = (type of b) (b + 1);

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

* If we apply any arithmetic operator between 2 variable a & b, the result type is always

**Max (int, type of a, type of b)**

i.e. byte + byte = int; byte + short = int; byte + long = long; char + double = double;

char + char = int;

* **Infinity:** In Integral arithmetic (byte, short, int, long), there is no way to represent infinity, hence if infinity comes, we will get Arithmetic Exception.
* But In floating point Arithmetic (float, double), there is a way to represent infinity. For this, Float & Double classes contain the following 2 constants: POSITIVE\_INFINITY; NEGATIVE\_INFINITY
* Hence, even though result is infinity, we won’t get any Arithmetic Exception in floating point arithmetic.

e.g. System.out.println(10/0.0); // Infinity

System.out.println(-10/0.0); // -Infinity

* **NaN:** In Integral arithmetic (byte, short, int, long), there is no way to represent undefined results, hence if undefined comes, we will get Arithmetic Exception.
* But In floating pint Arithmetic (float, double), there is a way to represent undefined. For this, Float & Double classes contain NaN constant.
* Hence, even though result is undefined, we won’t get any Arithmetic Exception in floating point arithmetic.

e.g. System.out.println(0/0.0); // NaN

* **Arithmetic Exception**
* Runtime Exception
* Possible only in Integral Arithmetic.
* The only operators which cause Arithmetic Exception are / and %

**3. String concatenation Operator (+)**

* The only overloaded operator in Java is “+” operator. Sometimes it acts as Arithmetic addition operator & sometimes it acts as String concatenation operator.
* If at least one argument is String type, “+” operator acts as concatenation operator & if both arguments are numeric type, then “+” operator acts as Arithmetic addition operator.

**4. Relational Operators (<, <=, >, >=)**

* We can apply relational operators for every primitive types except Boolean & Object types.
* Nesting of relational operators is not allowed otherwise we will get compile time error.

**5. Equality Operators (==, !=)**

* We can apply equality operators for every primitive type including Boolean type also.
* We can apply equality operators for Object types also but it’s reference comparison.
* 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)

Q. Difference between == operator & .equals () method ?

Ans: In general, we use == operator for reference comparation & .equals () method for content comparison.

**6. instanceof Operator**

* We can use instanceof operator to check whether the given object is of particular type or not.
* > Syntax: **r instanceof x**

where r is object reference & x is class name or interface name.

* To use instanceof operator, compulsory there should be some relation between argument types (Either child to parent or parent to child or same type).
* For any class or interface X, null **instanceof** X is always **false.**

|  |  |  |
| --- | --- | --- |
| **No** | **instanceof** | **isInstance ()** |
| 1. | Instanceof is an operator in Java. | isInstance () is a method present in java.lang.Class |
| 2. | We can use instanceof to check whether the given object is of particular type or not & we know the type at the beginning. | We can use isInstance () method to check whether the given object is of particular type or not & we don’t know the type at the beginning & it is available dynamically at runtime. |
| 3. | **E.g.**  Thread t = new Thread ();  t instanceof Runnable // true  t instanceof Object // true | **E.g.**  Thread t = new Thread ();  args [0] = String  Class.forName(args [0]). isInstance(t) //false |

**7. Bitwise Operators (&, |, ^)**

**&** (AND) – Returns true if both arguments are true.

**|**  (OR) – Returns true if at least one argument is true.

**^** (XOR) – Returns true if both arguments are different.

**e.g.** true & false = false true | false = true true ^ false = true

* We can apply these operators to boolean types as well as integral types (byte, short, int, long)

e.g. 4 & 5 = 4; 4 | 5 = 5; 4 ^ 5 = 1;

* **Bitwise Complement Operator (~): -** We can apply this operator only for integral types but not for boolean types.

e.g. ~4 = -5 [MSB of ~4 is 1 so negative no. will be represented in 2’s complement]

* **Boolean Complement Operator (!): -** We can apply this operator only for boolean types.

e.g. ! true = false;

**8. Short – Circuit Operator (&&, ||)**

* These are exactly same as Bitwise operator (&, |) except the following difference

|  |  |  |
| --- | --- | --- |
| No. | **&, |** | **&&, ||** |
| 1. | Both arguments should be evaluated always. | Second argument evaluation is optional. |
| 2. | Relatively performance is low. | Relatively performance is high. |
| 3. | Applicable for both boolean & integral types. | Applicable only for boolean types not for integral types. |

**Case 1:** (X **&&** Y) – Y will be evaluated if X is true i.e. if X is false then Y won’t be evaluated.

**Case 2:** (X **||** Y) – Y will be evaluated if X is false i.e. if X is true then Y won’t be evaluated.

**E.g.**

|  |  |  |
| --- | --- | --- |
| **op** | **x** | **y** |
| & | 11 | 17 |
| && | 11 | 16 |
| | | 12 | 16 |
| || | 12 | 16 |

int x = 10, y = 15;

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

x++;

} else {

y++;

}

**9. typecase Operator**

* There are 2 types of typecasting

a) Implicit typecasting

b) Explicit typecasting

**a) Implicit typecasting (widening / upcasting)**

* Compiler is responsible to perform implicit typecasting.
* Whenever we’re assigning smaller datatype value to bigger datatype variable, implicit typecasting will be performed.
* There is no loss of information in this typecasting.
* The following are various possible conversions where implicit typecasting will be performed.

byte short

int long float double

char

E.g. int x = ‘a’; System.out.println(x); // 97

double d = 10; System.out.println(d); // 10.0

**b) Explicit typecasting (narrowing / downcasting)**

* Programmer is responsible to perform explicit typecasting.
* Whenever we’re assigning bigger datatype value to smaller datatype variable then explicit typecasting is required.
* There may be a chance of loss of information in this typecasting.
* The following are various possible conversions where explicit typecasting will be performed.

byte short

int long float double

char

E.g. int x = 130; byte b = (byte) x; System.out.println(b); // -126

* Whenever we’re assigning bigger datatype value to smaller datatype variable by explicit typecasting, the MSB (Most significant bit) will be lost, we have to consider LSB.
* If we assign floating point values to the integral types by explicit typecasting, the digits after the decimal points will be loosed.

**10. Assignment Operators**

* There are 3 types of Assignment operators:

1. Simple Assignment: e.g. int x = 10;
2. Chained: e.g. int a, b, c, d; a = b = c = d = 20;
3. Compound Assignment operators:

* The following are all possible compound assignment operators in Java

|  |  |  |
| --- | --- | --- |
| +=  -=  \*=  /=  %= | &=  |=  ^= | >>= (right shift)  >>>= (unsigned right shift)  <<= (left shift) |

* In case of Compound Assignment, internal typecasting will be performed automatically.

E.g. byte b = 10; b += 1 means b = (byte) (b + 1)

**11. Conditional / Ternary Operators (? :)**

* The only possible ternary operator in Java is Conditional operator.
* **Syntax**: int x = (10 < 20) ? 30 : 40; // 30
* We can perform nesting of conditional operator also.

**12. new operator**

* We can use new operator to create Object.

**E.g.** Test t = new Test ();

* After creating an object, constructor will be executed to perform initialization of Object, hence constructor is not for creation of object & it is for initialization of an object.
* In Java, we have only new keyword but not delete keyword because destruction of useless objects is the responsibility of Garbage Collector.

**[] operator**

* We can use this operator to declare & create arrays.

E.g. int [] x = new int [10];

**new vs newInstance ()**

|  |  |  |
| --- | --- | --- |
| **No** | **new** | **newInstance ()** |
| 1. | It is operator in Java | It is a method in java.lang.Class |
| 2. | We can use new operator to create an object if we know class name at the beginning. | We can use newInstance () method to create object if we don’t know class name at the beginning & it is available dynamically at runtime. |
| 3. | To use new operator class not required to contain no-arg constructor. | To use newInstance () compulsory class should contain no-arg constructor otherwise we will get Runtime Exception saying **InstantiationException** |
| 4. | At runtime if .class file not available then we will get Runtime Exception saying **NoClassDefFoundError**. | At runtime if .class file not available then we will get Runtime Exception saying **ClassNotFoundException**. |
| 5. | **E.g.**  Test t = new Test (); | **E.g.**  Object o = Class.forName(args [0]).newInstance ()) |

**13. NoClassDefFoundError Vs ClassNotFoundException**

|  |  |  |
| --- | --- | --- |
| **No** | **NoClassDefFoundError** | **ClassNotFoundException** |
| 1. | For hardcoded class names, at runtime if the corresponding .class file is not available then we will get runtime exception saying NoClassDefFoundError. | For Dynamically provided class names, at runtime if the corresponding .class file is not available then we will get runtime exception saying ClassNotFoundException. |
| 2. | It is unchecked exception | It is checked exception. |
| 3. | Test t = new Test (); | Object o = Class.forName(args [0]). newInstance () |

**14. Java Operator precedence and Associativity (Evaluation order of Operands)**

