Operators & Assignments in Java

Operators define **how operands are evaluated and combined**.
In Java, operators are **strictly defined**, strongly typed, and **heavily tested** in certification exams—especially edge cases involving **type promotion, evaluation order, and side effects**.

---

1. Increment & Decrement Operators (`++`, `--`)

Types

- **Pre-increment**: `++x`
- **Post-increment**: `x++`
- **Pre-decrement**: `--x`
- **Post-decrement**: `x--`

Core Rule

- **Pre** → modify first, then use
- **Post** → use first, then modify

```
int x = 10;
int y = ++x; // y=11, x=11
int z = x++; // z=11, x=12
```

Restrictions (Exam Traps)

❌ Cannot apply to:

- Constants
- Literal values

- `final` variables
- `boolean`

```
++4;          // CE
final int a=5;
a++;          // CE
boolean b=true;
b++;          // CE
```

❌ **Nesting is not allowed**

```
++(++x);      // CE
```

Internal Type Casting (Important)

```
byte b = 10;
b++;          // valid
b = b + 1;    // CE
```

✔ `++` performs **implicit casting**, arithmetic operators do not.

---

2. Arithmetic Operators (`+ - * / %`)

Type Promotion Rule (VERY IMPORTANT)

> **Result type = max(int, type of operand1, type of operand2)**

Examples:

```
byte + byte     → int
char + char     → int
int + double    → double
```

```
System.out.println('a' + 'b'); // 195
System.out.println('a' + 1);   // 98
```

Division by Zero

Integral Types

```
System.out.println(10 / 0); // ArithmeticException
```

Floating-Point Types

```
System.out.println(10 / 0.0);  // Infinity
System.out.println(0.0 / 0.0); // NaN
```

✔ `Infinity` and `NaN` exist **only for float/double**

NaN Rules (Frequently Tested)

```
Float.NaN == Float.NaN     // false
Float.NaN != Float.NaN     // true
```

✔ Any comparison with NaN → `false` (except `!=`)

---

3. String Concatenation Operator (+)

Key Rule

- If **at least one operand is String**, + performs **concatenation**
- Evaluation is **left to right**

```
System.out.println("Java"+10+20); // Java1020
System.out.println(10+20+"Java"); // 30Java
```

Assignment Pitfalls

```
String s;
s = 10 + 20;       // CE
s = "" + 10 + 20; // valid
```

✔ Once concatenation starts, **everything becomes String**

4. Relational Operators (`< <= > >=`)

Allowed

- All **primitive types except boolean**

```
10 < 10.5        // true
'b' > 'a'        // true
```

❌ Not allowed for:

- `boolean`
- Objects (`String`, `Thread`, etc.)

```
"abc" > "xyz"; // CE
```

❌ **No nesting**

```
10 > 20 > 30; // CE
```

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

Primitive Types

✔ Compare **values**

```
10 == 10.0  // true
```

Object Types (Exam Favorite)

✔ `==` → **reference comparison**
✔ `.equals()` → **content comparison**

```
String s1 = new String("Java");
String s2 = new String("Java");

s1 == s2          // false
s1.equals(s2)     // true
```

Compatibility Rule

To compare objects using `==`, **types must be related**.

```
Thread t = new Thread();
Object o = new Object();

t == o   // valid
t == "x" // CE
```

`null` Rules

```
null == null   // true
obj == null    // false
null instanceof Object // false
```

---

6. `instanceof` Operator

Purpose

Checks **runtime type**, not compile-time type.

```
Object o = new String("Java");
o instanceof String // true
o instanceof Object // true
```

Rules

✔ Works with **classes & interfaces**
✔ Returns `false` for `null`
❌ Compile-time error if types are unrelated

```
Thread t = new Thread();
t instanceof String // CE
```

7. Bitwise Operators (& | ^ ~)

Boolean Usage

```
true & false  // false
true | false  // true
true ^ false  // true
```

Integral Usage (Bit-Level)

```
4 & 5  // 4
4 | 5  // 5
4 ^ 5  // 1
~4     // -5
```

✔ ~ works **only on integral types**

8. Logical NOT (!)

✔ Works **only on boolean**

```
!true  // false
!false // true
```

❌ Not allowed on numbers

9. Short-Circuit Operators (&&, ||)

Difference from & and |

Operator	Evaluates RHS?	Applicable To

&, `          `	Always
&&, `	`

```
if(false && (10/0>1)) // no exception
```

✔ Saves performance
✔ Prevents runtime errors

---

10. Type Casting Operators

Widening (Implicit)

```
int x = 'a';    // 97
double d = 10; // 10.0
```

✔ Safe, no data loss

Narrowing (Explicit)

```
int x = 130;
byte b = (byte)x; // -126
```

✔ Possible data loss
✔ Decimal part is truncated

---

11. Assignment Operators

Simple Assignment

```
int x = 10;
```

Chained Assignment

```
int a,b,c;
a = b = c = 20;
```

❌ Cannot chain at declaration

```
int a = b = c = 20; // CE
```

Compound Assignment (High-Weight Topic)

```
byte b = 10;
b += 1;  // valid
b = b+1; // CE
```

✔ Compiler performs **implicit casting**

---

12. Conditional (Ternary) Operator (`?:`)

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

✔ Nesting allowed
✔ Must return compatible types

---

13. `new` Operator & Arrays (`[]`)

- `new` creates objects
- No `delete` operator (GC handles cleanup)

```
int[] a = new int[5];
```

---

14. Operator Precedence (Exam-Relevant)

High → Low:

1. Unary (`++ -- ! ~ new`)
2. Arithmetic (`* / %`)

3. Relational
4. Equality
5. Logical
6. Ternary
7. Assignment

---

15. Operand Evaluation Order

✔ **Left to right** (always)

```
m1(1)+m1(2)*m1(3)
```

Operands evaluated in order: `1 → 2 → 3`

---

16. `new` vs `Class.newInstance()` (Modern Note)

`new`	Reflection
Compile-time type	Runtime type
No exception	Checked exceptions
Any constructor	Needs no-arg constructor

⚠️ `Class.newInstance()` is **deprecated**

✔ Use:

```
Class.forName("Test")
      .getDeclaredConstructor()
      .newInstance();
```

---

17. `instanceof` vs `Class.isInstance()`

- `instanceof` → compile-time type known
- `isInstance()` → runtime-dynamic type

Certification Takeaways (Must Remember)

- `++` performs implicit casting
- Arithmetic promotion always occurs
- `==` compares **references**, not content
- `NaN` breaks normal comparison rules
- `&&` prevents runtime errors
- Compound assignment hides casting
- Evaluation order ≠ precedence
- Reflection behaves differently from `new`