**Title:** Core Java

**Author:** Bhargavi Badiginchula | JavaArena Series

**Type Conversions in Java**

**Definition:**

Type Conversion in Java refers to converting a variable from one data type to another.  
It is necessary when performing operations involving different data types or when you need to match the type expected by a method or expression.

**Types of Type Conversions:**

**1. Widening Conversion (Implicit Conversion)**

Also known as automatic type conversion, it occurs when a smaller data type is converted into a larger data type without explicit casting.

📌 Rules:

* Happens automatically.
* No data loss.

Order of Widening:  
byte → short → int → long → float → double

Example:

int a = 10;

double b = a; // int to double (automatic)

System.out.println(b); // Output: 10.0

**2. Narrowing Conversion (Explicit Conversion)**

Also called type casting, this happens when a larger data type is converted to a smaller one.  
Since it may result in data loss, it must be done manually using type cast syntax.

Syntax:

*datatype variable = (datatype) expression;*

Example:

double a = 15.8;

*int b = (int) a; // double to int (manual)*

System.out.println(b); // Output: 15

**Operators in Java — Documentation**

**📘 Introduction**

Operators are symbols that perform operations on variables and values. Java provides a wide range of operators that are used to perform different kinds of operations.

**1. Arithmetic Operators**

These are used to perform basic mathematical operations.

| **Operator** | **Name** | **Definition** |
| --- | --- | --- |
| + | Addition | Adds two operands. |
| - | Subtraction | Subtracts the right operand from the left operand. |
| \* | Multiplication | Multiplies two operands. |
| / | Division | Divides the left operand by the right operand. |
| % | Modulus | Returns the remainder of a division operation. |

**2. Relational (Comparison) Operators**

Used to compare two values or expressions.

| **Operator** | **Name** | **Definition** |
| --- | --- | --- |
| == | Equal to | Returns true if both operands are equal. |
| != | Not equal to | Returns true if operands are not equal. |
| > | Greater than | Returns true if left operand is greater than right. |
| < | Less than | Returns true if left operand is less than right. |
| >= | Greater than or equal | Returns true if left operand is greater than or equal to right. |
| <= | Less than or equal | Returns true if left operand is less than or equal to right. |

**3. Logical Operators**

Used to combine multiple boolean expressions.

| **Operator** | **Name** | **Definition** |
| --- | --- | --- |
| && | Logical AND | Returns true if **both** conditions are true. |
| ` |  | ` |
| ! | Logical NOT | Reverses the result of a boolean expression. |

**4. Assignment Operators**

Used to assign values to variables.

| **Operator** | **Name** | **Definition** |
| --- | --- | --- |
| = | Assignment | Assigns the value on the right to the left variable. |
| += | Add and assign | Adds right operand to left and assigns result to left. |
| -= | Subtract and assign | Subtracts right from left and assigns result to left. |
| \*= | Multiply and assign | Multiplies and assigns result to left. |
| /= | Divide and assign | Divides and assigns result to left. |
| %= | Modulo and assign | Calculates remainder and assigns it to left. |

**5. Unary Operators**

Work with a single operand to modify its value.

| **Operator** | **Name** | **Definition** |
| --- | --- | --- |
| + | Unary Plus | Returns the value as it is (usually redundant). |
| - | Unary Minus | Negates the value (changes sign). |
| ++ | Increment | Increases value by 1 (++a or a++). |
| -- | Decrement | Decreases value by 1 (--a or a--). |
| ! | Logical NOT | Reverses a boolean value. |

**6. Bitwise Operators**

Used to perform operations on bits.

| **Operator** | **Name** | **Definition** |
| --- | --- | --- |
| & | Bitwise AND | Performs AND on each bit of two integers. |
| ` | ` | Bitwise OR |
| ^ | Bitwise XOR | Performs XOR on each bit of two integers. |
| ~ | Bitwise Complement | Inverts all bits of an integer. |
| << | Left shift | Shifts bits to the left (adds zeros on right). |
| >> | Right shift | Shifts bits to the right (preserves sign bit). |

**7. Ternary Operator**

| **Operator** | **Name** | **Definition** |
| --- | --- | --- |
| ?: | Ternary Operator | Short form of if-else. condition ? true\_value : false\_value |

Example:

int a = 10, b = 20;

int max = (a > b) ? a : b; // max = 20

**What are Command Line Arguments in Java?**

**Command Line Arguments** are inputs passed to a Java program during execution from the terminal or command prompt. These arguments are received as **strings** and stored in the String[] args array of the main() method.

They allow users to provide **dynamic input** to a program without modifying the code.

* Accessed using args[0], args[1], etc.
* All arguments are stored as String values.
* Used for passing configuration, user data, filenames, etc.
* Commonly used in automation, testing, or when deploying programs.

Naming Conventions:

Naming conventions in Java are a set of standardized rules that developers follow to name classes, methods, variables, and other identifiers in a consistent, readable, and maintainable way.

✅ Benefits:

- Improves code readability and collaboration

- Helps in understanding code structure quickly

- Makes your code look professional and standardized

These conventions are recommended by Java’s official style guides and are widely followed in industry-level projects.

Best Practices in Java

Classes:

- PascalCase (Each word starts with capital letter)

- Example: MyFirstClass, MemoryManager

Methods & Variables:

- camelCase (First word lowercase, next capitalized)

- Example: calculateSum(), studentAge

Constants:

- ALL\_UPPERCASE with underscores

- Example: MAX\_SIZE, DEFAULT\_TIMEOUT

Packages

- all lowercase, domain-style

- Example: com.mycompany.project.module

Avoid

- Single letter names (except for loop counters)

- Long confusing names

- Starting names with \_ or $ unless needed

Extra

- Keep methods short and specific

- Write comments to improve readability

- Group related code logically

General Rules:

- Names must not start with digits

- No special characters allowed except `\_` and `$` (though discouraged)

- Keywords cannot be used as names (e.g., `class`, `int`, `public`)

**Control Structures or Control Flow Statements**

**Why are they called Control Flow Statements?**

Because they **control the flow of execution** in a program — deciding:

* **What to execute** (if, switch)
* **How many times to repeat** (for, while, do-while)
* **Where to jump/skip** (break, continue, return)

**Conditional Statements in Java**

Conditional statements allow your program to **make decisions** and execute **certain blocks of code** based on conditions (like if, else, etc.).

Types:

Simple if

If-else

If-else if (else if ladder)

Switch

**1. if Statement**

Executes a block of code only **if the condition is true**.

*Syntax:*

*if (condition) {*

*// code to execute if condition is true*

*}*

**2. if-else Statement**

Executes one block if the condition is true, **another block if false**.

*Synatx:*

*if (condition) {*

*// code if condition is true*

*} else {*

*// code if condition is false*

*}*

**3.if-else if-else Ladder**

Checks multiple conditions in **a sequence**, executing the first true block.

*Syntax:*

*if (condition1) {*

*// code if condition1 is true*

*} else if (condition2) {*

*// code if condition2 is true*

*} else {*

*// code if none of the above conditions are true*

*}*

**4.switch Statement**

A clean alternative to long if-else-if ladders, especially when comparing a variable to multiple constant values.

*Synatx:*

*switch (expression) {*

*case value1:*

*// code block for value1*

*break;*

*case value2:*

*// code block for value2*

*break;*

*// more cases...*

*default:*

*// default code block*

*}*

**LOOPING STATEMENTS:**

**What is a Loop?**

A loop is a control structure that allows you to **execute a block of code multiple times** based on a condition.

Types :

For

While

Do-While

**for loop**

**Use when you know the number of repetitions in advance.**

**Syntax:**

for (initialization; condition; update) {

// code to repeat

}

**while loop**

**Use when the number of repetitions is not known ahead of time.**

**Syntax:**

while (condition) {

// code to repeat

}

**do-while loop**

**Executes the code block at least once before checking the condition.**

**Syntax:**

do {

// code to repeat

} while (condition);

For -Each loop

The **for-each loop** is a simplified way to **iterate over elements of arrays or collections** without using index counters.

It **automatically picks each element** from a group (like an array) one by one.

**Syntax:**

for (datatype element : arrayOrCollection) {

// code to use element

}

**Limitations:**

* You **cannot modify the original array directly** (e.g., cannot change values at a specific index).
* You **can’t access the index** of the current element unless you maintain a separate counter.

**Jumping Statements in Java**

**Jumping statements** in Java are used to **transfer control** to another part of the program **abruptly**. These help manage how the code flow jumps in loops or conditional blocks.

Types:

Break

Continue

Return

**1.break Statement**

Exits from the current loop or switch block immediately.

**Syntax:**

break;

**2.Continue statement:**

Skips the current iteration jumps to the next Iteration.

**Syntax:**

continue;

**3.return Statement**

Exits from the **current method** and optionally returns a value.

**Syntax:**

return; // for void methods

return value; //