JAVA: TYPE CASTING, OPERATORS, IF-ELSE, SWITCH, LOOPS, BREAK AND CONTINUE

JAVA TYPE CASTING

Type casting is used when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size byte -> short -> int -> long -> float -> double

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type double -> float -> long -> int -> short -> byte

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example Code

```
public class Main {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }
}
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example Code

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

CONVERSION FROM STRING TO INT

```
class Main {
  public static void main(String[] args) {
    // create string type variable
    String data = "10";
    System.out.println("The string value is: " + data);

    // convert string variable to int
    int num = Integer.parseInt(data);
    System.out.println("The integer value is: " + num);
  }
}
```

Output

The string value is: 10

The integer value is: 10

In the above example, notice the line

int num = Integer.parseInt(data);

Here, we have used the parseInt() method of the Java Integer class to convert a string type variable into an int variable.

Note: If the string variable cannot be converted into the integer variable then an exception named NumberFormatException occurs.

CONVERSION FROM INT TO STRING

```
class Main {

  public static void main(String[] args) {

    // create int type variable

    int num = 10;

    System.out.println("The integer value is: " + num);


    // converts int to string type

    String data = String.valueOf(num);

    System.out.println("The string value is: " + data);

  }

}
```

OUTPUT

The integer value is: 10

The string value is: 10

In the above program, notice the line

String data = String.valueOf(num);

Here, we have used the valueOf() method of the Java String class to convert the int type variable into a string.

JAVA OPERATORS

Operators are used to perform operations on variables and values.

Java divides the operators into the following groups:

1. **Arithmetic operators**
2. **Assignment operators**
3. **Comparison operators**
4. **Logical operators**

1. Java Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*****	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Example Code

```
public class OperatorsExample {
   public static void main(String[] args) {
      int a = 10;
      int b = 5;
      int additionResult = a + b;       // Addition (+)
      System.out.println("Addition result: " + additionResult);

      int subtractionResult = a - b;       // Subtraction (-)
      System.out.println("Subtraction result: " + subtractionResult);

      int multiplicationResult = a * b;       // Multiplication (*)
      System.out.println("Multiplication result: " + multiplicationResult);

      int divisionResult = a / b;       // Division (/)
      System.out.println("Division result: " + divisionResult);

      int modulusResult = a % b;       // Modulus (%)
      System.out.println("Modulus result: " + modulusResult);

      int c = 7;       // Increment (++)
      c++; // Equivalent to: c = c + 1;
      System.out.println("After increment, c is: " + c);

      int d = 9;       // Decrement (--)
      d--; // Equivalent to: d = d - 1;
      System.out.println("After decrement, d is: " + d);
   }
}
```

2. Java Assignment Operators

Assignment operators are used to assign values to variables.

A list of all assignment operators:

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

int x = 10;

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3

These operators are commonly used in Java for assigning values to variables and updating the values of variables by adding or subtracting another value.

1. Assignment (`=`):

int x = 10; // Assigns the value 10 to the variable x

2. Addition and Assignment (`+=`):

int x = 5;

x += 3; // Equivalent to: x = x + 3; (x becomes 8)

3. Subtraction and Assignment (`-=`):

int x = 10;

x -= 4; // Equivalent to: x = x - 4; (x becomes 6)

3. Java Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming because it helps us to find answers and make decisions.

The return value of a comparison is either true or false. These values are known as *Boolean values*, and you will learn more about them in If..Else chapter.

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

This code demonstrates the use of these comparison operators to compare two integer variables **a** and **b** and prints out the result of each comparison.

```
// Equality operator (==)
if (a == b) {
   System.out.println("a is equal to b");
} else {
   System.out.println("a is not equal to b");
}

// Inequality operator (!=)
if (a != b) {
   System.out.println("a is not equal to b");
} else {
   System.out.println("a is equal to b");
```

```
}

// Greater than (>)
if (a > b) {
   System.out.println("a is greater than b");
} else {
   System.out.println("a is not greater than b");
}

// Less than (<)
if (a < b) {
   System.out.println("a is less than b");
} else {
   System.out.println("a is not less than b");
}

// Greater than or equal to (>=)
if (a >= b) {
   System.out.println("a is greater than or equal to b");
} else {
   System.out.println("a is less than b");
}

// Less than or equal to (<=)
if (a <= b) {
   System.out.println("a is less than or equal to b");
} else {
   System.out.println("a is greater than b");
}
```

4. Java Logical Operators

You can also test for true or false values with logical operators.

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Logical AND (`&&`):

```
public class LogicalAndOperatorExample {
   public static void main(String[] args) {
      int a = 5;
      int b = 10;

      if (a > 0 && b < 15) {
         System.out.println("Both conditions are true");
      } else {
         System.out.println("At least one condition is false");
      }
   }
}
```

Logical OR (`||`):

```
public class LogicalOrOperatorExample {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;

        if (a > 0 || b > 15) {
            System.out.println("At least one condition is true");
        } else {
            System.out.println("Both conditions are false");
        }
    }
}
```

Logical NOT (`!`):

```
public class LogicalNotOperatorExample {
    public static void main(String[] args) {
        boolean isRainy = false;

        if (!isRainy) {
            System.out.println("It's not raining");
        } else {
            System.out.println("It's raining");
        }
    }
}
```

IF-ELSE IN JAVA

In Java, `if` and `else` are control flow statements used for decision-making. They allow you to execute certain blocks of code based on whether a specified condition evaluates to true or false.

Here's a basic syntax for `if` and `else` statements in Java:

```
if (condition) {
   // Code block to be executed if the condition is true
} else {
   // Code block to be executed if the condition is false
}
```

The `if` statement evaluates the condition inside the parentheses. If the condition is true, the code block immediately following the `if` statement is executed. If the condition is false, the code block associated with the `else` statement (if present) is executed. The `else` statement is optional.

Here's an example to illustrate the usage of `if` and `else`:

```
public class IfElseExample {
   public static void main(String[] args) {
      int number = 10;
      // Check if the number is positive or negative
      if (number > 0) {
         System.out.println("The number is positive");
      } else {
         System.out.println("The number is non-positive (zero or negative)");
      }
   }
}
```

In this example, the `if` statement checks if the variable `number` is greater than 0. If it is, the message "The number is positive" is printed. If the condition is false (i.e., if `number` is not greater than 0), the code block associated with the `else` statement is executed, and the message "The number is non-positive (zero or negative)" is printed.

You can also have multiple `if` statements (without `else`) or nested `if-else` statements to handle more complex conditions.

Nested If-Else Statements

Nested `if-else` statements in Java allow you to have multiple levels of decision-making within your code. This means that you can have an `if` statement inside another `if` statement, or an `if` statement inside an `else` block, and so on.

Here's the basic syntax for nested `if-else` statements in Java:

```
if (condition1) {
   // Code block to be executed if condition1 is true
   if (condition2) {
      // Code block to be executed if both condition1 and condition2 are true
   } else {
      // Code block to be executed if condition1 is true but condition2 is false
   }
} else {
   // Code block to be executed if condition1 is false
}
```

You can have as many levels of nesting as needed to handle the complexity of your conditions.

Here's an example demonstrating nested `if-else` statements:

```
public class NestedIfElseExample {
    public static void main(String[] args) {
        int num = 10;

        if (num > 0) {
            System.out.println("Number is positive.");

            if (num % 2 == 0) {
                System.out.println("Number is even.");
            } else {
                System.out.println("Number is odd.");
            }
        } else {
            System.out.println("Number is non-positive (zero or negative).");
        }
    }
}
```

In this example, we first check if the `num` variable is positive. If it is, we further check if it's even or odd using nested `if-else` statements. If `num` is not positive, we print that it's non-positive. This demonstrates how nested `if-else` statements can be used to handle multiple conditions in a structured manner.

Java Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands.

It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements:

Syntax

variable = (condition) ? expressionTrue : expressionFalse;

Instead of writing:

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

You can simply write:

```
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

JAVA SWITCH STATEMENTS

Instead of writing **many** if..else statements, you can use the switch statement.

The switch statement selects one of many code blocks to be executed:

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.

The example below uses the weekday number to show the weekday name:

```
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
```

```
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

The Break Keyword

When Java reaches a **break keyword**, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The Default Keyword

The **default keyword** specifies some code to run if there is no case match:

Note that if the default statement is used as the last statement in a switch block, it does not need a break.

LOOPS IN JAVA

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

JAVA FOR LOOP

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop and do-while loop and vice versa:

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

Example Explained

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

```
// Outer loop
for (int i = 1; i <= 2; i++) {
  System.out.println("Outer: " + i); // Executes 2 times

  // Inner loop
  for (int j = 1; j <= 3; j++) {
    System.out.println(" Inner: " + j); // Executes 6 times (2 * 3)
  }
}
```

Java While Loop

The while loop loops through a block of code as long as a specified condition is true:

Syntax

```
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  // code block to be executed
}
while (condition);
```

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
int i = 0;
do {
  System.out.println(i);
  i++;
}
while (i < 5);
```

JAVA BREAK AND CONTINUE

Java Break: You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch statement.

The break statement can also be used to jump out of a **loop**.

This example stops the loop when i is equal to 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  System.out.println(i);
}
```

Java Continue: The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  System.out.println(i);
}
```

Break and Continue in While Loop

You can also use break and continue in while loops:

Break Example

```
int i = 0;

while (i < 10) {

 System.out.println(i);

 i++;

 if (i == 4) {

  break;

 }

}
```

Continue Example

```
int i = 0;

while (i < 10) {

 if (i == 4) {

  i++;

  continue;

 }

 System.out.println(i);

 i++;

}
```

PRACTICE TASKS:

1. Write a Java program that demonstrates widening casting by converting a smaller data type to a larger one. Prompt the user to enter a byte value, then cast it to a short, int, long, float, and double, printing out each result.

2. Create a Java program that showcases narrowing casting by converting a larger data type to a smaller one. Ask the user to input a double value, then cast it to a float, long, int, short, and byte, displaying each converted value.

3. Code a simple grading system for a school. The grading system will take a student's score as input and output their corresponding grade based on the following criteria:

- A score of 90 or above earns an 'A' grade.
- A score of 80 to 89 earns a 'B' grade.
- A score of 70 to 79 earns a 'C' grade.
- A score of 60 to 69 earns a 'D' grade.
- Any score below 60 earns an 'F' grade.

4. Make a task that can take five student result and average them. Take input in the loop 5 times. You can use any loop (for, while or do-while).

5. You are starting a lucky draw contest and You saved a lucky number (other than 0) in your program and now you are giving turns to other players to guess a number. A number given by a player can be a right guess or a wrong guess. If a player guess wrong so print try again and give him another chance until he quits by entering 0 or win the game. Use **break** statement in your program wherever needed.

6. Identify even numbers from 1 to 10 in a loop by using **continue** statement.