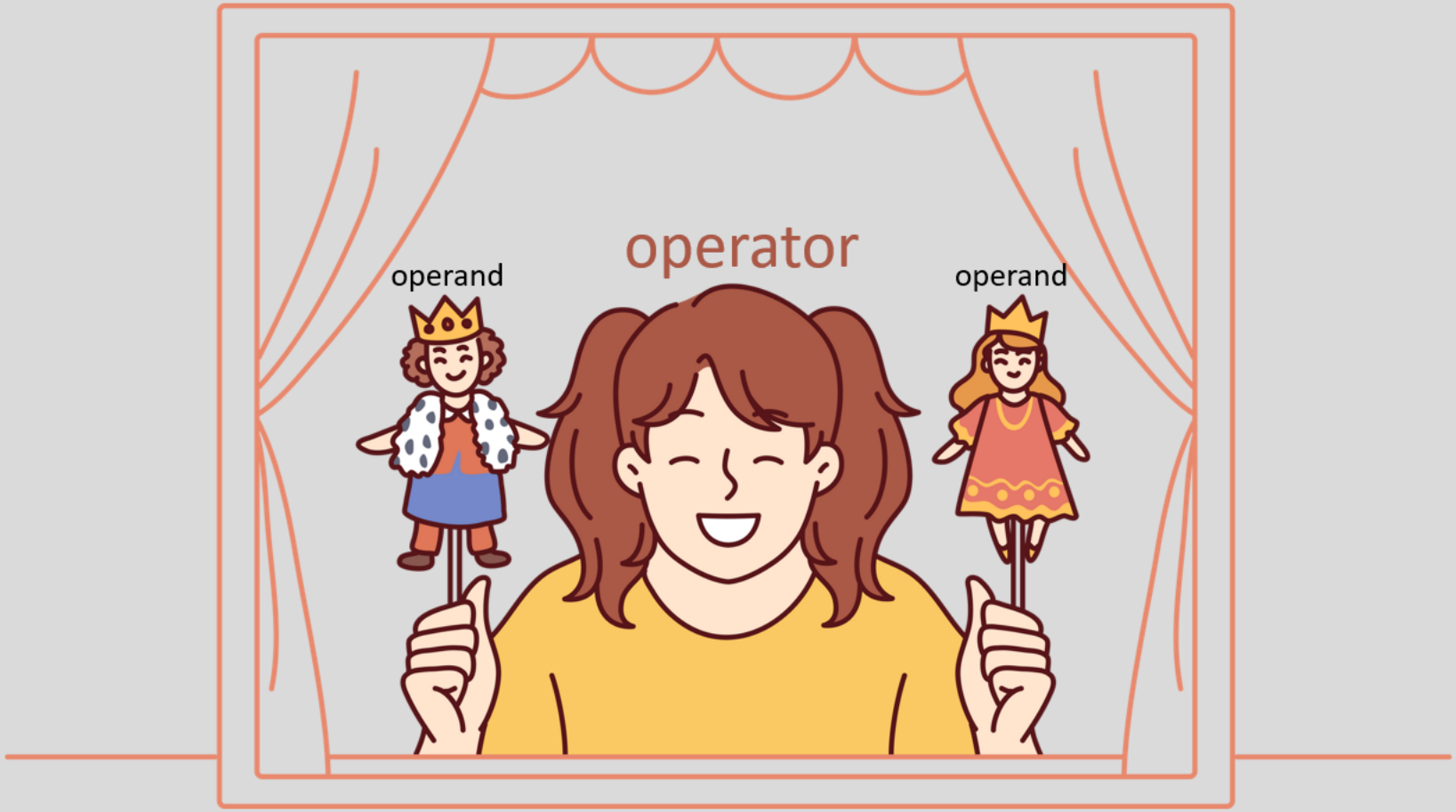
Operators & Operands in Java

eazy bytes

Inside puppet shows, we have an operator who operates puppets (operands). Just like puppet shows, inside Java also we have operators and operands.



Operators & Operands in Java

eazy bytes

An operator in Java is like a symbol that carries out a specific action on one, two, or three operands, generating a result. The nature of the operator, along with the types of its operands, determines the operation performed and the resulting data type.

An operand is a value or variable that is manipulated by an operator.

For example, in the below expression, the + and = are the operators, and the operands are 2 and 3. Variables used instead of literals can also be operands.

```
int result  = 2 + 3;
```

Variables used instead of literals can also be operands like shown below.

```
int num1 = 10;
int num2 = 20;
int result  = 160+ (num1 * num2);
```

Java operators fall into two categories:

- Based on the number of operands they work with.
- Based on the type of operation they execute on these operands.

Operators categorization based on the number of operands

eazy bytes

There are three types of operators in terms of the number of operands they handle: unary operators, which involve one operand; binary operators, which involve two operands; and ternary operators, which involve three operands.

unary operator

Below are few examples of using a unary operator:

num++
5++
num--
+66

Syntax

<operand> <postfix-unary-operator>

<prefix-unary-operator> <operand>

binary operator

Below are few examples of using a binary operator:

num1 + num2
10 - 5
5 * 3
a + b

Syntax

<first-operand> <infix-binary-operator> <second-operand>

ternary operator

Below is a example of using a ternary operator:

isEven ? even : odd

Syntax

<first-operand> <operator1> <second-operand> <operator2> <third-operand>

Operators categorization based on the type of operation

eazy bytes

Operators in Java can be classified into specific categories based on the type of operation they execute. These categories include arithmetic operators, relational operators, logical operators, and bitwise operators.

arithmetic operators

Used to perform basic mathematical operations on numerical values.

Example operators include + (addition), - (subtraction), * (multiplication), / (division), % (modulo).

relational operators

Used to compare values and produce a boolean result.

Example operators include == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to).

logical operators

Used to perform logical operations and combine boolean values.

Example operators include && (logical AND), || (logical OR), ! (logical NOT).

bitwise operators

Used for bit-level operations on integral types.

Example operators include & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (left shift), >> (right shift).

Expressions in Java

eazy bytes

In Java, an expression is a combination of one or more values, operators, and method invocations that produces a result. Expressions are the building blocks of Java programs and are used to perform a wide range of operations, from simple arithmetic calculations to complex data manipulations.

The Java interpreter evaluates an expression to compute its value. The very simplest expressions are called primary expressions and consist of literals and variables.

```
false //literal
3.4 //literal
isValid //variable
piValue //variable
```

Primary expressions are simple & not very interesting. More complex expressions are made by using operators to combine primary expressions. For example, in the below expressions we used the assignment operator (=) to combine two primary expressions—a variable and a literal.

```
boolean isValid = false;
float piValue = 3.4f;
```

The variety of expressions that can be written in a programming language is determined solely by the operators that are at one's disposal. Fortunately, Java offers a vast array of operators, which we are going to explore and learn.

Assignment Operator (=)

eazy bytes

The assignment operator (=) in Java, classified as a binary operator, serves the purpose of assigning a value to a variable. This operator, requiring two operands, involves assigning the value of the right-hand operand to the left-hand operand, with the condition that the left-hand operand must be a variable.

Below are few examples. In the first example, 98 is the right-hand operand, and num is the left-hand operand, which is a variable.

```
int num = 98;
boolean isValid = true;
String name = ”John”;
```

Assignment operator is a right-associative operator. Because it is evaluated from right to left. Associativity of operators in Java refers to the order in which operators of the same precedence are evaluated in an expression.

For example, consider the below chained assignment operator expression. Since assignment operator is a right-associative, first it is going to assign 98 to num2 followed by expression num2 = 98 is replaced by a value 98, which changes the main expression num1 = num2 = 98 to num1 = 98.

```
int num1, num2;
num1 = num2 = 98;
```

Assignment Operator (=)

eazy bytes

We can have any number of variables in a chained assignment expression like shown below and the expression is always going to be evaluated from right to left

```
num1 = num2 = num3 = num4 = num5 = 98;
```

Consider below example where we are trying to assign a value of 3.14 and 4.48 initially to num1 and num2. When you execute num1 = num2 in Java, the value stored in num2 is copied to num1. Both num1 and num2 then maintain their independent copies of the same value, such as 4.48. If, at a later point, you execute num2 = 6.58, only the value of num2 changes to 6.58. However, the value of num1 remains unaffected and continues to hold its original value, 4.48. This behavior illustrates that each variable retains its own distinct copy of the assigned value, allowing them to evolve independently in subsequent assignments.

```
double num1 = 3.14;
double num2 = 4.48;
num1 = num2;
num2 = 6.58;
```

Now consider two reference variables, objRef1 and objRef2, pointing to distinct objects of the same class in Java. When the assignment objRef1 = objRef2; is executed, both objRef1 and objRef2 become references to the same object in memory.

Subsequently, both objRef1 and objRef2 possess equal capabilities to manipulate this shared object. Any modifications made to the object in memory through objRef1 will be reflected in objRef2, and vice versa, establishing a mutual influence between the two reference variables.

Assignment Operator (=)

eazy bytes

What is Declaration, Initialization and Assignment ?

Declaration is the process of introducing a variable to the Java compiler. It involves specifying the variable's data type and name.

```
int age;  // Declaring an integer variable named 'age'
```

Initialization is the process of assigning a value to a variable at the time of declaration itself

```
int age = 25;  // Declaration and Initialization
```

Assignment is the process of assigning a value to a variable after it has been declared

```
int age;  // Declaring an integer variable named 'age'
age = 25;  // Assignment
```

Multiple variables of the same type can be declared in a single statement by separating each variable's name with a comma.

```
int x, y, z; // Declaring three integer variables: x, y, and z
```

Assignment Operator (=)

eazy bytes

In Java, it's permissible to declare and initialize multiple variables within a single statement, allowing for the option to set initial values for either some or all of them.

```
int x = 56, y, z = 86; // Declaration of variables x, y and z but initialization of x and z only
int x = 73, y = 93, z = 13; // Declaration and initialization of variables x, y and z
```

Java guarantees that the right-hand operand of the assignment operator must be assignment-compatible with the data type of the left-hand operand. If this compatibility is not met, a compile-time error will occur.

Arithmetic Operators

eazy bytes

Arithmetic operators in Java are symbols that work with numeric values, like addition and subtraction, to produce a new numeric result. It's important to note that these operators exclusively operate with numeric data types, specifically byte, short, char, int, long, float, and double. They do not support boolean primitive types or reference types as operands.

The table below outlines all arithmetic operators in Java.

Operator	Description	Example	Result	Type
+	Addition	1 + 6	7	Binary
-	Subtraction	6 – 1	5	Binary
*	Multiplication	3 * 6	18	Binary
/	Division	14/2 14/3 25.0/2.0 40.0/8.0	7 4 12.5 5.0	Binary
%	Modulus	15%4 24%8	3 0	Binary
+	Unary plus	+9	Postive 9	Unary
-	Unary minus	-9	Negative 9	Unary

Arithmetic Operators

eazy bytes

Operator	Description	Example	Result	Type
++	Increment	num++	Evaluates the value of num & increase it's value by 1.	Unary
--	Decrement	num--	Evaluates the value of num & decrease it's value by 1.	Unary
+=	Arithmetic compound assignment	num += 8	Same as num = num + 8	Binary
-=		num -= 8	Same as num = num - 8	Binary
*=		num *= 8	Same as num = num * 8	Binary
/=		num /= 8	Same as num = num / 8	Binary
%=		num %= 8	Same as num = num % 8	Binary

eazy bytes

Arithmetic Operators

The arithmetic operators can operate on integers, floating-point numbers, and even characters, except for boolean types. When one of the operands is a floating-point number, the calculation will use floating-point arithmetic. On the other hand, if neither of the operands is a floating-point number, the calculation will use integer arithmetic.

Addition (+)

When the + operator is applied to numbers, it performs addition. However, it's worth noting that the same operator can also be used to concatenate strings, which we discussed previously. If one of the operands is a string, the other operand is automatically converted to a string as well.

```
int total = 8 + 8 ; // 16

String output = ”Hello" + 1 + 6 ; // Hello16
```

The + operator can also be used in unary form to express a positive number like +16

Subtraction (-)

As a binary operator, the - symbol subtracts the second operand from the first operand. For instance, 12 minus 8 evaluates to 4.

```
int num1 = 12 - 8 ; // 4
int num2 = 4 - 8; //-4
```

Unary minus (-)

If the - operator appears before a single operand, it functions as a unary operator, specifically unary negation. Its role is to convert a positive value to an equally negative value, and vice versa.

```
int negativeNum = -8;
int positiveNum = -(-8);
```

Multiplication (*)

The * operator multiplies its two operands. For example, 8*2 evaluates to 16.

```
int total = 8 * 2 ; // 16
```

Addition Operator

eazy bytes

The addition operator in Java adds two numbers together. For instance, 4 + 7 equals 11. You can use this operator with numeric values, variables, expressions, or method calls. When you use the addition operator, the result's data type follows these rules. These rules carry significant consequences.

1. If one operand is double, the other is converted to double, and the result is of type double.
2. If one operand is float, the other is converted to float, and the result is of type float.
3. If one operand is long, the other is converted to long, and the result is of type long.
4. If none of the above applies, all operands are converted to int (if not already), and the result is of type int.

Consider below code example, where we are trying to assign a value 8 to a byte variable and it works perfectly with out any issues.

```
byte num1;
num1 = 8;
```

Consider below code example, where we are getting a compilation error when we try to assign the same 8 value with the + operator. So what's going on ?

```
byte num1;
byte num2 = 5;
byte num3 = 3;
num1 = num2 + num3; // A compile-time error. Trying to assign 8 to num1
```

In the case of the second assignment, num1 = num2 + num3, the fourth rule for determining the data type of an arithmetic expression comes into play. Since both operands (num2 and num3) are of the byte type, they are initially converted to int. Consequently, the expression num2 + num3 becomes of the int type. As the data type of num1 is byte, smaller than the int type of the expression num2 + num3, attempting to assign int to byte causes an incompatibility issue, leading to the observed error.

Addition Operator

eazy bytes

We can try to fix the previous compilation error by doing a type casting with the following code examples,

```
num1 = (byte)(num2 + num3); // Compilation will be success
num1 = (byte) num2 + num3; // Compilation will fail
```

The expressions (byte)(num2 + num3) and (byte)num2 + num3 differ in their outcomes. In the (byte)(num2 + num3) expression, both num2 and num3 are first promoted to int, followed by an addition resulting in the int value 8. Subsequently, this int value is cast to byte and assigned to num1.

On the other hand, in the (byte)num2 + num3 expression, the initial cast of num2 to byte is of no use as num2 is already of type byte. However, both num2 and num3 are then promoted to int, making the entire expression (byte)num2 + num3 of type int. Due to the restriction on assigning int to byte, this expression would fail to compile.

Why didn't Java calculate num2 + num3 first in (byte)num2 + num3 and then apply (byte) to the result? The reason lies in the precedence order of operators in Java. Operators with higher precedence are evaluated before those with lower precedence. Since the cast operator has higher precedence than the addition operator, Java prioritizes casting num2 to byte first and then performing the addition. This decision is not arbitrary; it follows the predefined order of operator precedence. However, you can alter this order by using parentheses. In the expression (byte)(num2 + num3), we explicitly changed the precedence, ensuring that the addition is evaluated first before the cast.

Addition Operator

eazy bytes

Now let's see another typical scenario using addition operator. Consider the below example,

```
byte num1;
num1 = 5 + 3; // Do you think this line of code compile successfully ?
```

According to the fourth rule for determining the data type, the expression num1 = 5 + 3 should not compile, as 5 and 3 are int literals, resulting in an int type for the expression. Since int is not directly assignable to byte, an error would be expected. However, contrary to this assumption, the expression num1 = 5+ 3 compiles successfully due to a unique scenario.

In this case, the operands 5 and 3 are constants with known values at compile time. The compiler optimizes by computing the result of 5 + 3 during compilation, replacing the expression with its result, which is 8. Consequently, the expression num1 = 5 + 3 is effectively transformed into num1 = 8 by the compiler.

This behavior is permissible because the int literal 8 falls within the range of -128 to 127, making the assignment num1 = 8 valid according to the rule allowing assignment of an int literal to a byte variable. However, if attempting an expression like num1 = 127 + 1, which results in 128, it would not compile due to exceeding the range for a byte data type.

The rules we covered for the numeric data conversion of operands and the determination of the data type in expressions using the addition operator also apply to expressions involving the subtraction, multiplication, division, modulus operators.

String Concatenation Operator (+)

eazy bytes

The + operator in Java is overloaded, meaning it is used to perform more than one function. While it is commonly known as an arithmetic addition operator for adding two numbers, it can also be employed to concatenate two strings. For instance, when applied to two strings like ”hello" and ”world," the + operator concatenates them, resulting in a new string ”helloworld"

Let’s try to solve the below puzzle,

```
4 + 2 + " goals" // What will be the result ? Will it be 42 goals or 6 goals
```

In the expression 4 + 2 + " goals", the addition is executed from left to right because + operator is left-associative:

1. The first + operator adds 4 and 2, resulting in 6
2. The expression now becomes 6 + " goals". Since the right-hand operand is a string, the + operator performs string concatenation, resulting in the final string ”6 goals".

The same output will be evaluated for the below code as well,

```
int num1 = 4;
int num2 = 2;
String str1 = " goals";
String str2;
str2 = num1 + num2 + str1; // “6 goals” will be the value stored inside str2
```

String Concatenation Operator (+)

eazy bytes

Let's try to solve the below puzzle,

```
int num1 = 4;
int num2 = 2;
String str1 = " goals";
String str2;
str2 = num1 + (num2 + str1);
```

The expression inside the parentheses, (num2 + str1), is evaluated first. If num2 is a numeric type and str1 is a string, the + operator performs string concatenation. The result of this expression is a string.

Then, this string is concatenated with num1 using the outer + operator, resulting in a final string "42 goals"

The same output will be evaluated for the below code statements as well.

```
str2 = "" + num1 + num2 + str1; // "42 goals" will be the value stored inside str2
```

```
str2 = num1 + "" + num2 + str1; // "42 goals" will be the value stored inside str2
```

In the above example codes, the empty string makes + operator to act as a String concatenation operator while evaluating the statement from left to right.

String Concatenation Operator (+)

eazy bytes

Let's try to solve the below puzzle,

```
boolean b = false;
int num = 6;
String str1 = "goals";
String str2 = b + num + str1;
```

The last statement compilation fails. Do you know why ?

When Java try to evaluate the expression from left to right, it will find a + Operator which can't be used as either String concatenation operator or normal addition operator. Because the first operand which is b is neither a String nor a int. So compilation fails.

We can fix the above compilation error with any of the below approaches,

```
str2 = b + (num + str1);  // "false6goals" will be the value stored inside str2

str2 = "" + b + num + str1; // "false6goals" will be the value stored inside str2

str2 = b + "" + num + str1;  // OK. Assigns "false6goals" to str2
```

If a String variable holds a null reference, the concatenation operator will use the string "null". Where as if you try concatenate the null literal to a int literal the compilation fails.

```
String str1 = null;
String str2 = 6 + str1; // "6null"
String str3 = "goals" + null; // "goalsnull"
String str4 = 6 + null; // Compilation fails
```

Arithmetic Operators

eazy bytes

Quotient

Division (/)

When the / operator is used, the first operand is divided by the second operand. If both operands are integers, the result will be an integer, and any remainder will be discarded. However, if either operand is a floating-point value, the outcome will be a floating-point value. If you divide two integers, dividing by zero will throw an ArithmeticException. On the other hand, for floating-point calculations, dividing by zero will result in an infinite result or NaN.

```
8/3             // Output will be 2
8/3.0f          // Output will be 2.6666667
8/0             // Throws an ArithmeticException
8/0.0           // Output will be infinity
0.0/0.0         // Output will be NaN
```

Remainder

Modulus (%)

The % operator computes the first operand modulo the second operand (i.e., it returns the remainder when the first operand is divided by the second operand an integral number of times). For example, 8%3 is 2. The sign of the result is the same as the sign of the first operand. While the modulo operator is typically used with integer operands, it also works for floating-point values. When you are operating with integers, trying to compute a value modulo zero causes an ArithmeticException. When you are working with floating-point values, anything modulo 0.0 evaluates to NaN, as does infinity modulo anything.

```
8%3             // Output will be 2
8%3.0f          // Output will be 2.0
8%0             // Throws an ArithmeticException
8%0.0           // Output will be NaN
0.0%0.0         // Output will be NaN
```

Division Operator (/)

eazy bytes

There are two types of division:

- Integer division: When both operands of the division operator are integers (byte, short, char, int, or long), the division operation is performed in the usual way, and the result is truncated towards zero to represent an integer. For instance, if you have the expression 9/2, the division results in 4.5, but the fractional part (0.5) is disregarded, and the final result is 4. This type of division, where the fractional part is ignored, is known as integer division.

- Floating point division: When either or both operands of the division operator are float or double, a floating point division is carried out, and the result retains its decimal precision without truncation. For example, 27/2.0f will give ouput as 13.5

Performing integer division by zero in a Java program results in a runtime error. For example, if you have the expression 9/0, it compiles without issues but leads to an ArithmeticException during runtime.

However, if either operand of the division operator is a floating-point number, dividing by zero in a floating-point division does not result in an error. Instead, the result becomes positive infinity or negative infinity.

0.0F/0.0F will result in Float.NaN and 0.0/0.0 will result in Double.NaN

Modulus Operator (%)

eazy bytes

Special rules apply when determining the result of a modulus operation since it involves a division operation internally.

When both operands of the modulus operator are integers, the following rules are applied to calculate the result

Rule 1: If the right operand is zero, it will result in run time exception ArithmeticException

Rule 2: If the right-hand operand is not zero, the sign of the result aligns with the sign of the left-hand operand. For example:

```
int num;
num = 21 % 2;   // Result is 1
num = -21 % 2;  // Result is -1
num = 21 % -2;  // Result is 1
num = -21 % -2; // Result is -1
```

Modulus Operator (%)

eazy bytes

When either operand of the modulus operator is a floating-point number, the result is determined according to the following rules.

Rule 1: If the right-hand operand is 0, 0.0 or 0.0F, the result will be NaN. For example, the output of 2.0F % 0.0F is NaN

Rule 2: If either operand is NaN, the result of the operation will be NaN. For example, the output of Float.NaN % 1.5F is NaN

Rule 3: If the left-hand operand is infinity, the result will be NaN. For example, the output of Float.POSITIVE_INFINITY % 6.3F is NaN

Rule 4: If the right-hand operand is infinity, the result will be same as left operand. For example, the output of 6.3%Float.POSITIVE_INFINITY is 6.3

Rule 5: If none of the previous rules apply, the modulus operator returns the remainder of the division between the left-hand operand and the right-hand operand. The sign of the result aligns with the sign of the left-hand operand. Below are few examples,

```
13.5%2.5 // 1.0
7.5%20.5 // 7.5
4.3f % 4.4f // 4.3f
```

Unary Plus Operator (+)

eazy bytes

The operand for the unary plus operator must be a primitive numeric type. If the operand is a byte, short, or char, it gets promoted to an int. Otherwise, using this operator has no effect. For instance, if there's an int variable 'num' with a value of 9, applying +num still results in the same value of 9.

Let's see below example of code and try to understand why it fails compilation,

```
byte num1 = 9;
byte num2 = 3;
num1 = num2;  // Compilation will be successful
num1 = +num2; // Compilation fails
```

The compilation fails due to the following reason,

num2 is of type byte. But, with the use of unary plus operator on num2 will promote its type to int. So, +num2 is of type int. Assign the int num2 to a byte num1 is not allowed. So the compilation fails.

We can fix the above compilation error by using the below code,

```
num1 = (byte) +num2; // Compilation will be successful
```

Unary Minus Operator (-)

eazy bytes

The operand for the unary minus operator must be a primitive numeric type. If the operand is a byte, short, or char, it gets promoted to an int. This operator arithmetically negates the value of its operand. For instance, if there's an int variable 'num' with a value of 9, applying -num results in the value of -9.

Let's see below example of code and try to understand why it fails compilation,

```
byte num1 = 9;
byte num2 = 3;
num1 = num2;  // Compilation will be successful
num1 = -num2; // Compilation fails
```

The compilation fails due to the following reason,

num2 is of type byte. But, with the use of unary minus operator on num2 will promote its type to int. So, -num2 is of type int. Assign the int num2 to a byte num1 is not allowed. So the compilation fails.

We can fix the above compilation error by using the below code,

```
num1 = (byte) -num2; // Compilation will be successful
```

Applying the minus operator 2 times, will result in the same positive number. Below is an example,

```
num = - (-9); // num will be assigned with 9
```

Compound Arithmetic Assignment Operators

eazy bytes

Each of the five fundamental arithmetic operators (+, -, *, /, and %) has an associated compound arithmetic assignment operator.

The compound arithmetic assignment operator is employed in the below syntax. Here, 'op' represents one of the arithmetic operators +, -, *, /, and %. Both operand1 and operand2 are of primitive numeric data types, where operand1 must be a variable.

```
operand1 op= operand2
```

The above expression is equal to the following expression:

```
operand1 = (Type of operand1) (operand1 op operand2)
```

For example, consider below code.

```
int num = 42;
num += 3.3; // num will have value 45 populated by following the code num = (int) num + 3.3
```

The compound assignment operator += is applicable to String variables as well. In such instances, operand1 must be of type String, and operand2 can be of any type, including boolean.

```
String str = "Hello";
str= str + 9;    // Assigns "Hello9" to str
str += 9; // Instead of above statement, we can use this as well
```

The += operator is the only one that can be utilized with a String as the left-hand operand.

Compound Arithmetic Assignment Operators

eazy bytes

There are two advantages associated with the use of compound arithmetic assignment operators:

Performance

Operand1 is evaluated only once.

For instance, in num+=3.3, the variable num is evaluated only once. In contrast, in num=(int)(num+3.3), the variable num is evaluated twice.

Automatic casting

The result is automatically cast to the type of operand1 before assignment. This cast may result in either a narrowing conversion or an identity conversion.

Increment (++) and Decrement (--) Operators

eazy bytes

The increment operator (++) is used with a variable of a numeric data type to increase the variable's value by 1, while the decrement operator (--) is utilized to decrease the value by 1. In this section, we specifically focus on the increment operator. The same principles apply to the decrement operator, with the sole distinction being that it decreases the value by 1 instead of increasing it by 1.

While using this operators, the operand must be a variable, an element of an array, or a field of an object. For example, below is invalid,as they operate on literals directly,

```
2++ //invalid
4--  //invalid
```

There are two kinds of increment/decrement operators:

1. Postfix increment/decrement operator, for example, num++ / num--

2. Prefix increment/decrement operator, for example, ++num / --num

When used before the variable name (called pre increment/decrement), the value of the variable is modified before the rest of the expression is evaluated. When used after the variable name (called post-increment/decrement), the value of the variable is modified after the expression is evaluated.

Increment (++) and Decrement (--) Operators

eazy bytes

Let's consider below code and expressions with a postfix increment operator,

```
int num1 = 45;
int num2 = 4;
num2 = num1++ + 5;  // Assigns 50 to num2 and num1 becomes 46
```

The evaluation of the expression `num2 = num1++ + 5;` unfolds as follows:

1. The current value of num1 is assessed, and the right-hand expression transforms into 45+5.
2. The value of num1 in memory is then incremented by 1. Consequently, the value of the variable num1 in memory becomes 46
3. The expression 45+5 is computed, and the resulting value of 50 is assigned to num2

Let's consider below code and expressions with a prefix increment operator,

```
int num1 = 45;
int num2 = 4;
num2 = ++num1 + 5;  // Assigns 51 to num2 and num1 becomes 46
```

The evaluation of the expression `num2 = ++num1 + 5;` unfolds as follows:

1. Since ++num1 uses the prefix increment operator, first the value of num1 is incremented in memory by 1. So num1 value becomes 46
2. The current value of num1 which is 46 used in the expression. So the expression becomes 46+5
3. The expression 46+5 is computed, and the resulting value of 51 is assigned to num2

Increment (++) and Decrement (--) Operators

eazy bytes

Let’s try to solve the below puzzle,

```
int num = 9;
num = num++; // What will be the value stored inside num after executing this statement ?
```

Here's the explanation of how the expression is evaluated:

1. Since num++ uses a postfix increment operator, the current value of num (which is 9) is used in the expression.
2. The expression becomes num=9.
3. The value of num is then incremented by 1 in memory as the second effect of num++. At this point, the value of num is 10 in memory.
4. The expression num=9 is evaluated, and the value 9 is assigned to num. The final value of the variable num in memory is 9. In fact, although num momentarily had a value of 10 in a previous step, this step overwrote that value with 9.

Therefore, the ultimate value of the variable num after num=num++ is executed will be 9, not 10.

Increment (++) and Decrement (--) Operators

eazy bytes

Let’s try to consider the below code and understand the output values,

```
int num = 9;
num = num++ + num;  // Assigns 19 to num
```

```
num = 9;
num = ++num + num++; // Assigns 20 to num
```

Let's break down the expression num = num++ + num; step by step:

1. The current value of num is 9.
2. The postfix increment operator num++ uses the current value of num (9) in the expression and then increments num by 1. So, the expression becomes num = 9 + num.
3. Now, the value of num is incremented to 10.
4. The expression becomes num = 9 + 10.
5. The addition is performed, and the result, 19, is assigned to num.

Let's break down the expression num = ++num + num++; step by step:

1. The current value of num is 9
2. The prefix increment operator ++num increments the current value of num by 1 first, and then the incremented value is used in the expression. So, the expression becomes num = 10 + num++
3. The expression becomes num = 10 + 10 followed by incrementing the num to 11 inside memory
4. The addition is performed, and the result, 20, is assigned to num.

Relational Operators

eazy bytes

A relational operator is used to compare the values of its operands. Examples of such comparisons include equality, inequality, greater than, less than, and more. Java features seven relational operators, with six of them listed in below table. The seventh operator, instanceof, will be discussed in the coming lectures.

Operator	Description	Example	Result	Type
==	Equal To	9==9 6 == 9	true false	Binary
!=	Not Equal to	9 != 9 6 != 9	true false	Binary
>	Greater Than	6 > 9	false	Binary
>=	Greater Than or Equal To	6 >= 9	false	Binary
<	Less Than	6 < 9	true	Binary
<=	Less Than or Equal To	6 <= 9	true	Binary

Relational operators in Java are binary operators, meaning they operate on two operands. The outcome of a relational operator is always a boolean value, resulting in either true or false.

Equality Operator (==)

eazy bytes

Below is the syntax of using the equality operator (==) :

operand1 == operand2

The equality operator is used to assess the equality of two operands, following these rules:

1. The equality operator requires both operands to be of the same type, either primitive or reference. Combining different types of operands is not permitted.
2. For primitive operands, it evaluates to true if both operands represent the same value; otherwise, it results in false. Both operands must be either numeric or boolean; a combination of numeric and boolean types is not allowed.
3. For reference operands, the operator returns true if both operands refer to the same object in memory; otherwise, it returns false.
4. We should not use the == operator to test two strings for equality, instead we need to use equals() method

Below are few examples of using equality operator,

```
int num1 = 9;
int num2 = 9;
boolean isSame = num1 == num2; // true
boolean isSimilar = num1 == 9; // true
boolean isEqual = 9 == 9; // true
```

```
boolean a = true;
boolean b = true;
boolean isSame = a == b; // true
boolean isSimilar = a == true; // true
boolean isEqual = true == true; // true
```

Equality Operator (==)

eazy bytes

Let’s try to consider the below code and understand the output values,

```
int num1;
int num2;
int num3;
boolean a;
num1 = num2 = num3 = 9;   // Assign 9 to num1, num2 and num3
a = (num1 == num2 == num3); // A compile-time error
```

The expression a = (num1 == num2 == num3); will result in a compilation error. The reason is that the equality operator (==) associates from left to right, and it compares the values of its operands.

Let's break down the expression and try to understand why it fails compilation:

1. num1 == num2 is evaluated first, resulting in a boolean value (true or false).
2. The next comparison becomes either true == num3 or false == num3, which is an invalid comparison between a boolean and an integer. We cannot mix Boolean and numeric type operands with the equality operator.

To fix this, you need to compare each pair of operands separately or use logical operators like shown below. Below code ensures that you compare the integers separately and then combine the results using the logical AND (&&) operator.

```
a = (num1 == num2) && (num2 == num3);
```

Equality Operator (==)

eazy bytes

When dealing with floating-point types, the equality operator in Java follows these rules:

Rule 1: NaN (Not a Number): If either operand is NaN, the equality operator returns false. Even if both operands are NaN, the result is false.

```
double num1 = Double.NaN;
double num2 = 3.14;
boolean a = (num1 == num2); // false
```

```
num1 = Double.NaN;
num2 = Double.NaN;
b = (num1 == num2); // false
```

If NaN and equality operator always returns false, then how to test if a float or double variable holds NaN ? For the same, Float and Double classes have an isNaN() method, which accepts a float and a double argument, respectively. It returns true if the argument is NaN; otherwise, it returns false.

```
double num1 = Double.NaN;
boolean isNan = Double.isNaN(num1); // true
```

Rule 2: Positive and Negative Zero: Positive zero (+0.0) is considered equal to negative zero (-0.0), so the equality operator returns true.

```
double num1 = 0.0;
double num2 = -0.0;
boolean isSame = (num1 == num2); // true
```

Equality Operator (==)

eazy bytes

Rule 3: Infinity: If both operands are positive infinity or both are negative infinity, the equality operator returns true. If one operand is positive infinity and the other is negative infinity, the result is false.

```
double num1 = Double.POSITIVE_INFINITY;
double num2 = Double.NEGATIVE_INFINITY;
boolean isSame = (num1 == num2); // false
```

Rule 4: Finite Values: For finite values, the equality operator checks if the values are numerically equal. If both operands have the same numerical value, the result is true. If the values are different, the result is false.

```
double num1 = 3.14;
double num2 = 3.14;
double num3 = 3.15;
boolean isSame = (num1 == num2); // true
 boolean isSame = (num1 == num3); // false
```

Inequality Operator (!=)

eazy bytes

Below is the syntax of using the inequality operator (!=) :

operand1 != operand2

The inequality operator (!=) returns true if operand1 and operand2 are not equal; otherwise, it returns false. Here are several examples of using the inequality operator:

```
int num1 = 5;
int num2 = 10;
boolean result = (num1 != num2); // true, because 5 is not equal to 10

boolean bool1 = true;
boolean bool2 = false;
boolean result = (bool1 != bool2); // true, because true is not equal to false

double num1 = 3.14;
double num2 = 2.71;
boolean result = (num1 != num2); // true, because 3.14 is not equal to 2.71
```

Inequality Operator (!=)

eazy bytes

When dealing with floating-point types, the inequality operator in Java follows these rules:

Rule 1: NaN (Not a Number): If either operand is NaN, the inequality operator returns true. Even if both operands are NaN, the result is true.

```
double num1 = Double.NaN;
double num2 = 3.14;
boolean a = (num1 == num2); // true
```

```
num1 = Double.NaN;
num2 = Double.NaN;
b = (num1 == num2); // true
```

Rule 2: Positive and Negative Zero: Positive zero (+0.0) is considered equal to negative zero (-0.0), so the inequality operator returns false.

```
double num1 = 0.0;
double num2 = -0.0;
boolean isSame = (num1 != num2); // false
```

Inequality Operator (!=)

eazy bytes

Rule 3: Infinity: If both operands are positive infinity or both are negative infinity, the inequality operator returns false. If one operand is positive infinity and the other is negative infinity, the result is true.

```
double num1 = Double.POSITIVE_INFINITY;
double num2 = Double.NEGATIVE_INFINITY;
boolean isSame = (num1 != num2); // true
```

Rule 4: Finite Values: For finite values, the inequality operator checks if the values are numerically not equal. If both operands have the same numerical value, the result is false. If the values are different, the result is true.

```
double num1 = 3.14;
double num2 = 3.14;
double num3 = 3.15;
boolean isSame = (num1 != num2); // false
 boolean isSame = (num1 != num3); // true
```

Greater Than Operator (>)

eazy bytes

Below is the syntax of using the “greater than” operator,

operand1 > operand2

The "greater than" operator evaluates to true if the value of operand1 is greater than the value of operand2; otherwise, it yields false. This operator is applicable exclusively to primitive numeric data types. If either operand is NaN (floating-point or double), the result is false.

```
int num1 = 9;
int num2 = 6;
double num3 = Double.NaN;
boolean isGreater;
isGreater = (num1 > num2); // true
isGreater = (num3 > Double.NaN); // false
boolean isValid = (true > false); // Compilation fails as > can be used only with primitive numeric data types
String str1 = "Hello";
String str2 = "World";
isValid = (str1 > str2); // Compilation fails as > can’t be used with reference data types
```

Greater Than or Equal to Operator (>=)

eazy bytes

Below is the syntax of using the “greater than or equal to” operator,

operand1 >= operand2

The "greater than or equal to" operator evaluates to true if the value of operand1 is greater than or equal to the value of operand2; otherwise, it yields false. This operator is exclusively applicable to primitive numeric data types. If either operand is NaN (floating-point or double), the result of the greater than or equal to operator is false.

```
int num1 = 9;
int num2 = 6;
double num3 = Double.NaN;
boolean isGreaterOrEqual;
isGreaterOrEqual = (num1 >= num2); // true
isGreaterOrEqual = (num3 >= Double.NaN); // false
boolean isValid = (true >= false); // Compilation fails as >= can be used only with primitive numeric data types
String str1 = "Hello";
String str2 = "World";
isValid = (str1 >= str2); // Compilation fails as >= can’t be used with reference data types
```

Less Than Operator (<)

eazy bytes

Below is the syntax of using the “less than” operator,

operand1 < operand2

The ”less than" operator evaluates to true if the value of operand1 is less than the value of operand2; otherwise, it yields false. This operator is applicable exclusively to primitive numeric data types. If either operand is NaN (floating-point or double), the result is false.

```
int num1 = 9;
int num2 = 6;
double num3 = Double.NaN;
boolean isLess;
isLess = (num1 < num2); // false
isLess = (num3 < Double.NaN); // false
boolean isValid = (true < false); // Compilation fails as < can be used only with primitive numeric data types
String str1 = "Hello";
String str2 = "World";
isValid = (str1 < str2); // Compilation fails as < can’t be used with reference data types
```

Less Than or Equal to Operator (<=)

eazy bytes

Below is the syntax of using the “less than or equal to” operator,

operand1 <= operand2

The ”less than or equal to" operator evaluates to true if the value of operand1 is less than or equal to the value of operand2; otherwise, it yields false. This operator is exclusively applicable to primitive numeric data types. If either operand is NaN (floating-point or double), the result of the less than or equal to operator is false.

```
int num1 = 9;
int num2 = 6;
double num3 = Double.NaN;
boolean isLessOrEqual;
isLessOrEqual = (num1 <= num2); // false
isLessOrEqual = (num3 <= Double.NaN); // false
boolean isValid = (true <= false); // Compilation fails as <= can be used only with primitive numeric data types
String str1 = "Hello";
String str2 = "World";
isValid = (str1 <= str2); // Compilation fails as <= can’t be used with reference data types
```

Logical Operators

eazy bytes

Logical operators in Java take boolean operands, apply boolean logic to them, and generate a boolean value. Below table enumerates the boolean logical operators available in Java. It's important to note that all boolean logical operators can only be used with boolean operands.

Operator	Description	Example	Result	Type
!	Logical NOT	!false	true	Unary
&&	Short-circuit AND	true && true	true	Binary
&	Logical AND	true && true	true	Binary
||	Short-circuit OR	true || false	true	Binary
|	Logical OR	true | false	true	Binary
^	Logical XOR (Exclusive OR)	true ^ true	false	Binary
&=	AND assignment	a &= b	a = a & b	Binary
|=	OR assignment	a |= b	a = a | b	Binary
^=	XOR assignment	a ^= b	a = a ^ b	Binary

Logical NOT operator (!)

eazy bytes

Below is the syntax of using the “logical NOT” operator,

```
!operand
```

The Logical NOT Operator (!) in Java is a unary operator that negates the boolean value of its operand. It's used to reverse the logical state of a boolean expression. Below are few examples,

```
boolean isValid;
isValid = !true;   // false
isValid = !false;  // true
int num1 = 9;
int num2 = 6;
isValid = !(num1 > num2); // false because num1 > num2 returns true
```

The operand of the logical NOT operator must be of type boolean.

It's a unary operator, meaning it operates on only one operand.

Logical Short-Circuit AND Operator (&&)

eazy bytes

Below is the syntax of using the “logical short-circuit AND” operator,

operand1 && operand2

The && operator in Java returns true only if both operands are true. If any operand is false, it results in false. It's referred to as a short-circuit AND operator because if the left-hand operand (operand1) evaluates to false, the operator immediately returns false without evaluating the right-hand operand (operand2).

operand1	operand2	Result of && operator
true	true	true
true	false	false
false	true	false
false	false	false

Not evaluated as first operand is false

Logical Short-Circuit AND Operator (&&)

eazy bytes

Consider below example. The evaluation begins with num1 > 5, which returns true. Since the left-hand operand is true, the right-hand operand, num2 < 10, is then evaluated, and it also returns true. Consequently, the expression is simplified to true && true. As both operands are true, the ultimate result is true.

```
int num1 = 9;
int num2 = 6;
boolean isTrue = num1 > 5 && num2 < 10; // true
```

Consider another example. The expression num1 > 10 yields false. Consequently, the expression is transformed into false && num2 < 10. As the left-hand operand is false, the right-hand operand num2 < 10 is not assessed, and the && operator immediately returns false. However, that in this example, there is no conclusive evidence that the right-hand operand (num2 < 10) was indeed not evaluated. To illustrate this point, let's examine another example.

```
int num1 = 9;
int num2 = 6;
boolean isTrue = num1 > 10 && num2 < 10; // false
```

Logical Short-Circuit AND Operator (&&)

eazy bytes

In the below example, since num1>5 evaluates to true, the second expression ((num2 = 16) > 15)) will be executed. In the same process, the num2 value is changed to 16. That's why we are able to see 16 when we print num2 value

```
int num1 = 9;
int num2 = 6;
boolean b = (num1 > 5 && ((num2 = 16) > 15));
System.out.println("b = " + b); // true
System.out.println("num1 = " + num1); // 9
System.out.println("num2 = " + num2); // 16
```

In the below example, since num1>10 evaluates to false, the second expression ((num2 = 16) > 15)) will never executed. That's why num2 value never changed from the initial value which is 6. So when we print num2 value, we get 6 as an output.

```
int num1 = 9;
int num2 = 6;
boolean b = (num1 > 10 && ((num2 = 16) > 15));
System.out.println("b = " + b); // false
System.out.println("num1 = " + num1); // 9
System.out.println("num2 = " + num2); // 6
```

Logical AND Operator (&)

eazy bytes

Below is the syntax of using the “logical AND” operator,

operand1 & operand2

The logical AND operator returns true only if both operands are true; otherwise, it returns false. The behavior of the logical AND operator (&) is similar to the logical short-circuit AND operator (&&), with one key difference — the logical AND operator (&) evaluates its right-hand operand even if the left-hand operand evaluates to false.

operand1	operand2	Result of & operator
true	true	true
true	false	false
false	true	false
false	false	false

Evaluated even if first operand is false

Logical AND Operator (&)

eazy bytes

Consider below examples.

```
int num1 = 9;
int num2 = 6;
boolean isTrue = (num1 > 5 & num2 < 10); // true
isTrue = (num1 > 10 & num2 < 10); // false
```

In the below example, even num1>10 evaluates to false, the second expression ((num2 = 16) > 15)) will be executed. That's why num2 value changed from the initial value 6 to 16. So when we print num2 value, we get 16 as an output.

```
int num1 = 9;
int num2 = 6;
boolean b = (num1 > 10 & ((num2 = 16) > 15));
System.out.println("b = " + b); // false
System.out.println("num1 = " + num1); // 9
System.out.println("num2 = " + num2); // 16
```

Logical Short-Circuit OR Operator (||)

eazy bytes

Below is the syntax of using the “logical short-circuit OR” operator,

operand1 || operand2

The logical short-circuit OR operator returns true if at least one operand is true; otherwise, it returns false. It is termed a short-circuit OR operator because if the left-hand operand (operand1) evaluates to true, the operator immediately returns true without assessing the right-hand operand (operand2).

operand1	operand2	Result of || operator
true	true	true
true	false	true
false	true	true
false	false	false

Not evaluated as first operand is true

Logical Short-Circuit OR Operator (||)

eazy bytes

Consider below example. The evaluation begins with num1 < 5, which returns false. Since the left-hand operand is false, the right-hand operand, num2 < 10, is then evaluated, which returns true. Consequently, the expression is simplified to false && true, which will result into true as final value

```
int num1 = 9;
int num2 = 6;
boolean isTrue = (num1 < 5 || num2 < 10); // true
```

Consider another example. The expression num1 > 5 yields true. Since the left-hand operand is true, with out executing the right-hand operand expression num2<10, the final value true will be assigned to isTrue variable

```
int num1 = 9;
int num2 = 6;
boolean isTrue = (num1 > 5 || num2 < 10); // true
```

Logical OR Operator (|)

eazy bytes

Below is the syntax of using the “logical OR” operator,

operand1 | operand2

The logical OR operator returns true if either operand is true and false if both operands are false. While the logical short-circuit OR operator behaves similarly, there is a crucial difference — the logical OR operator (|) evaluates its right-hand operand even if the left-hand operand evaluates to true.

operand1	operand2	Result of | operator
true	true	true
true	false	true
false	true	true
false	false	false

Evaluated even if first operand is true

Logical OR Operator (|)

eazy bytes

Consider below examples.

```
int num1 = 9;
int num2 = 6;
boolean isTrue = (num1 > 5 | num2 < 10); // true
isTrue = (num1 > 10 | num2 > 10); // false
```

In the below example, even num1<10 evaluates to true, the second expression ((num2 = 16) > 15)) will be executed. That's why num2 value changed from the initial value which is 6 to 16. So when we print num2 value, we get 16 as an output.

```
int num1 = 9;
int num2 = 6;
boolean b = (num1 < 10 | ((num2 = 16) > 15));
System.out.println("b = " + b); // true
System.out.println("num1 = " + num1); // 9
System.out.println("num2 = " + num2); // 16
```

Logical XOR Operator (^)

eazy bytes

Below is the syntax of using the "logical XOR" operator,

```
operand1 ^ operand2
```

The logical XOR operator returns true when operand1 and operand2 have different values. In other words, it yields true if one of the operands is true, but not both. If both operands are the same, it results in false.

operand1	operand2	Result of ^ operator
true	true	false
true	false	true
false	true	true
false	false	false

Below is the example of XOR operator,

```
int num1 = 9;
int num2 = 6;
boolean isTrue = (num1 > 5 ^ num2 < 10); // false
isTrue = (num1 > 5 ^ num2 > 10); // true
isTrue = (num1 < 5 ^ num2 < 10); // true
isTrue = (num1 < 5 ^ num2 > 10); // false
```

Compound Logical Assignment Operators

eazy bytes

There are three compound logical assignment operators in Java. It's important to note that Java does not support operators &&= and ||= . Below is the syntax of compound boolean logical assignment operators,

```
operand1 op= operand2
```

operand1 must be a boolean variable, and op may be &, |, or ^. The above syntax is equivalent to below,

```
operand1 = operand1 op operand2
```

Below table shows compound logical assignment operators and their equivalent expressions,

Compund expression	Equivalent expression
operand1 &= operand2	operand1 = operand1 & operand2
operand1 |= operand2	operand1 = operand1 | operand2
operand1 ^= operand2	operand1 = operand1 ^ operand2

Compound Logical Assignment Operators

eazy bytes

Below are few examples on using compound logical assignment operators and their outputs,

```
boolean b1 = true;
b1 &= true;  // true
b1 &= false; // false

boolean b2 = true;
b2 |= true;  // true
b2 |= false; // true

boolean b3 = true;
b3 ^= true;  // false
b3 ^= false; // true
```

Bitwise Operators

eazy bytes

In Java, bitwise operators are used to perform operations at the bit level. Java supports below bitwise operators, and they can be applied to integer types (byte, short, int, long) and char.

Operator	Description	Example	Result	Type
&	Bitwise AND	10 & 6	2	Binary
|	Bitwise OR	9 | 6	15	Binary
^	Bitwise XOR	9 ^ 6	15	Binary
~	Bitwise NOT (1's complement)	~ 9	-10	Unary
<<	Left Shift	9 << 6	576	Binary
>>	Signed right shift	9 >> 6	0	Binary
>>>	Unsigned right shift	9 >>> 6	0	Binary
&=, !=, ^=, <<=, >>=, >>>=	Compound assignment bitwise operators	x &= y x |= y x ^= y x <<= y x >>= y x >>>= y	x = x & y x = x | y x = x ^ y x = x << y x = x >> y x = x >>> y	Binary

The bitwise operators are not commonly used in modern Java except for low-level work (e.g., network programming).

Bitwise NOT Operator/ 1's complement (~)

eazy bytes

The bitwise complement (~) operator is a unary operator that changes each bit of its operand to its opposite value (0 becomes 1 and 1 becomes 0). The bitwise complement operator works by performing a logical NOT operation on each bit of the operand.

Bitwise NOT operator	
Input bit	Output bit
1	0
0	1

Below is an example code on how to use Bitwise NOT operator,

```
int x = 9;
int y = ~x; // -10
```

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x = 9	0	0	0	0	1	0	0	1
~x	1	1	1	1	0	1	1	0

int y = ~x; performs a bitwise NOT operation on the value of x.
The bitwise NOT operator (~) flips each bit of the operand. It changes each '0' to '1' and each '1' to '0'.

After the bitwise NOT operation, the binary representation of y is 11110110.
In decimal, this binary representation is equal to -10 (assuming two's complement representation).

Steps to represent a negative number in binary

eazy bytes

To represent a negative number in binary, you can use the Two's complement notation. The Two's complement is a way of representing signed integers in binary. Here's how you can represent the decimal number -10 in binary using Two's complement:

1) Start with the binary representation of the positive counterpart of the number. The binary representation of 10 is 0000 1010 (8 bits for simplicity).

2) Invert all the bits (change 0s to 1s and 1s to 0s). Inverting 0000 1010 gives 1111 0101.

3) Add 1 to the inverted binary number. Adding 1 to 1111 0101 gives 1111 0110.

So, the binary representation of -10 in an 8-bit Two's complement format is 1111 0110.

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
STEP 1 -> 10 in binary	0	0	0	0	1	0	1	0
STEP 2 -> INVERT	1	1	1	1	0	1	0	1
STEP 3 -> ADD 1								1
-10 in binary	1	1	1	1	0	1	1	0

Bitwise AND Operator (&)

eazy bytes

The bitwise AND operator (&) is a binary operator that performs a bitwise AND operation on the binary representation of its operands. It compares each bit of the two operands, and if both bits are 1, then the resulting bit is also 1; otherwise, the resulting bit is 0.

Bitwise AND operator		
operand1 bit	operand2 bit	output
1	1	1
1	0	0
0	1	0
0	0	0

Below is an example code on how to use Bitwise AND operator,

```
int x = 10;
int y = 6;
int z = x & y; // 2
```

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x = 10	0	0	0	0	1	0	1	0
Y = 6	0	0	0	0	0	1	1	0
z = x & y	0	0	0	0	0	0	1	0

After the bitwise AND operation, the binary representation of z is 00000010. In decimal, this binary representation is equal to 2.

Bitwise OR Operator (|)

eazy bytes

The bitwise OR operator (|) is a binary operator that performs a bitwise OR operation on the binary representation of its operands. It compares each bit of the two operands, and if either bit is 1, then the resulting bit is also 1; otherwise, the resulting bit is 0.

Below is an example code on how to use Bitwise OR operator,

```
int x = 10;
int y = 6;
int z = x | y; // 14
```

Bitwise OR operator		
operand1 bit	operand2 bit	output
1	1	1
1	0	1
0	1	1
0	0	0

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x = 10	0	0	0	0	1	0	1	0
Y = 6	0	0	0	0	0	1	1	0
z = x | y	0	0	0	0	1	1	1	0

After the bitwise AND operation, the binary representation of z is 00001110. In decimal, this binary representation is equal to 14.

Bitwise XOR Operator (^)

eazy bytes

The bitwise XOR operator (^) is a binary operator that performs a bitwise exclusive OR operation on the binary representation of its operands. It compares each bit of the two operands, and if the bits are different, then the resulting bit is 1; otherwise, the resulting bit is 0.

Bitwise XOR operator		
operand1 bit	operand2 bit	output
1	1	0
1	0	1
0	1	1
0	0	0

Below is an example code on how to use Bitwise XOR operator,

```
int x = 10;
int y = 6;
int z = x ^ y; // 12
```

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x = 10	0	0	0	0	1	0	1	0
Y = 6	0	0	0	0	0	1	1	0
z = x ^ y	0	0	0	0	1	1	0	0

After the bitwise AND operation, the binary representation of z is 00001100. In decimal, this binary representation is equal to 12.

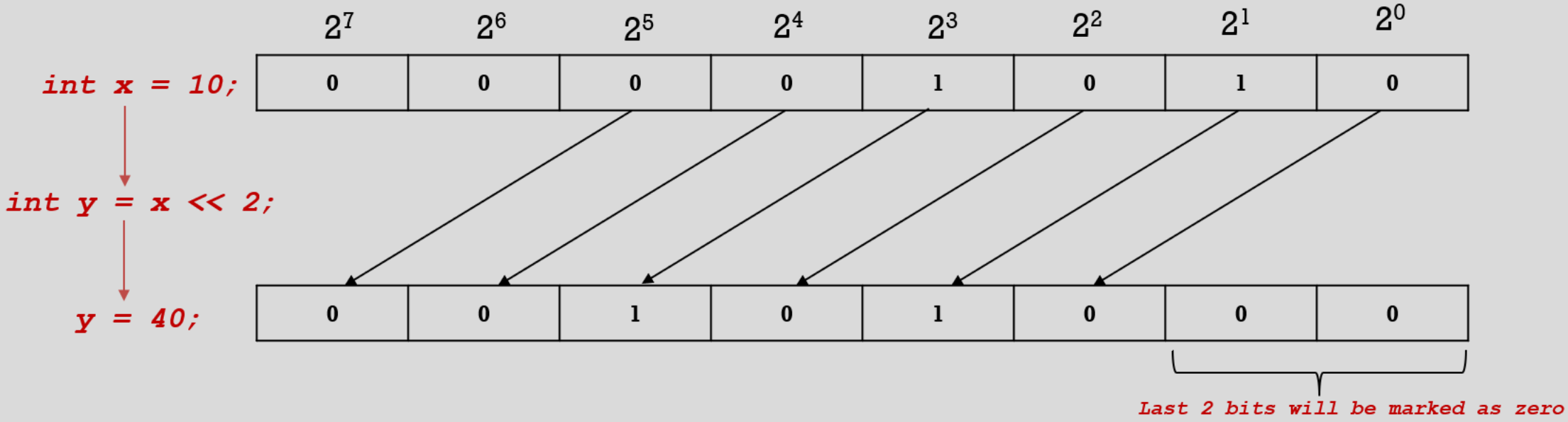
Left shift Operator (<<)

eazy bytes

The left shift operator (<<) is a binary operator that performs a bitwise left shift operation on the binary representation of its left-hand operand by the number of bits specified by its right-hand operand. High-order bits of the left operand are lost, and zero bits are shifted in from the right. Shifting an integer left by n places is equivalent to multiplying that number by 2^n.

Below is an example code on how to use Left Shift operator,

```
int x = 10;
int y = x << 2; // 40
```



In this example, the variable x is assigned the value 10, which is represented in binary as 00001010. The left shift operator is then applied to x with a shift amount of 2, resulting in the value of y being 40 (represented in binary as 00101000). It's important to note that the left shift operator may cause the sign bit to change for signed integer types.

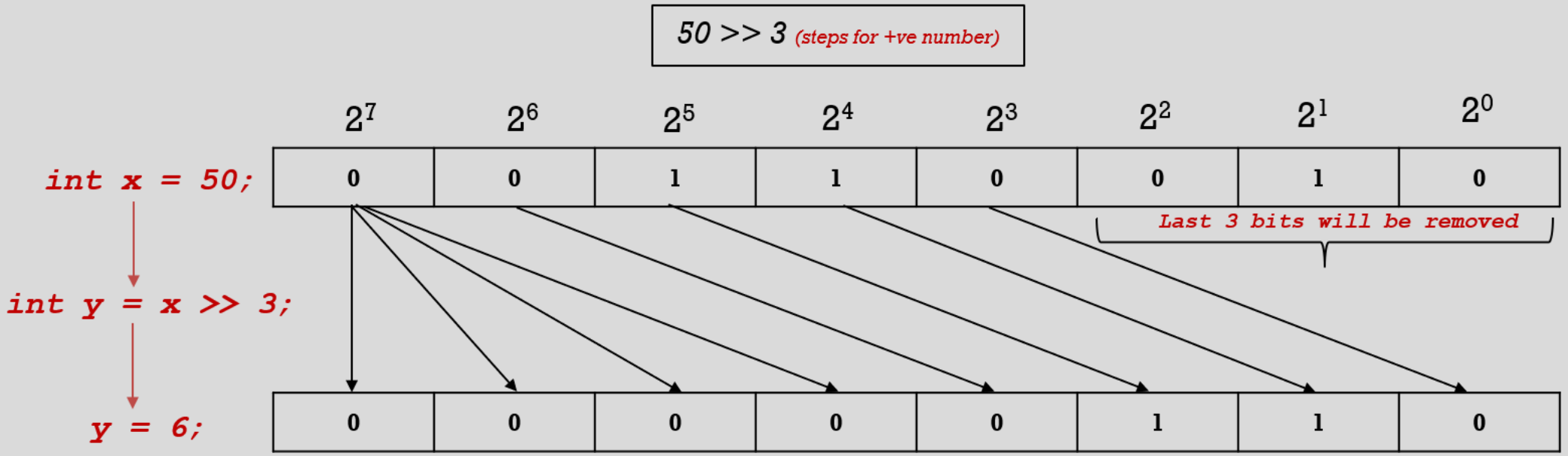
Signed right shift Operator (>>)

eazy bytes

The signed right shift operator (>>) is a binary operator that performs a bitwise right shift operation on the binary representation of its left-hand operand by the number of bits specified by its right-hand operand. Unlike the unsigned right shift operator (>>>), the signed right shift operator preserves the sign of the left-hand operand when shifting.

Below is an example code on how to use signed right shift operator,

```
int x = 50;
int y = x >> 3; // 6
```

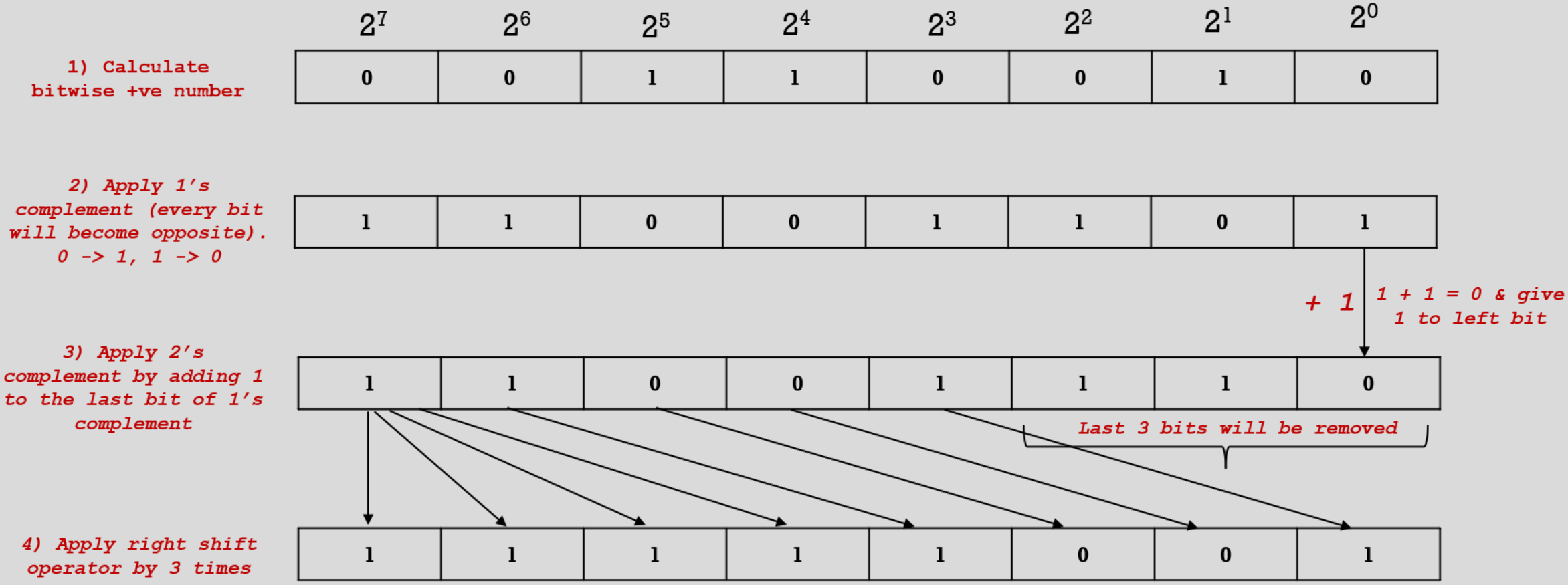


If the left operand is positive and the right operand is n, the >> operator is the same as integer division by 2^n.
In the above example, $50/2^3 = 50/8 = 6$

Signed right shift Operator (>>)

eazy bytes

-50 >> 3 (steps for -ve number)



Output is -7

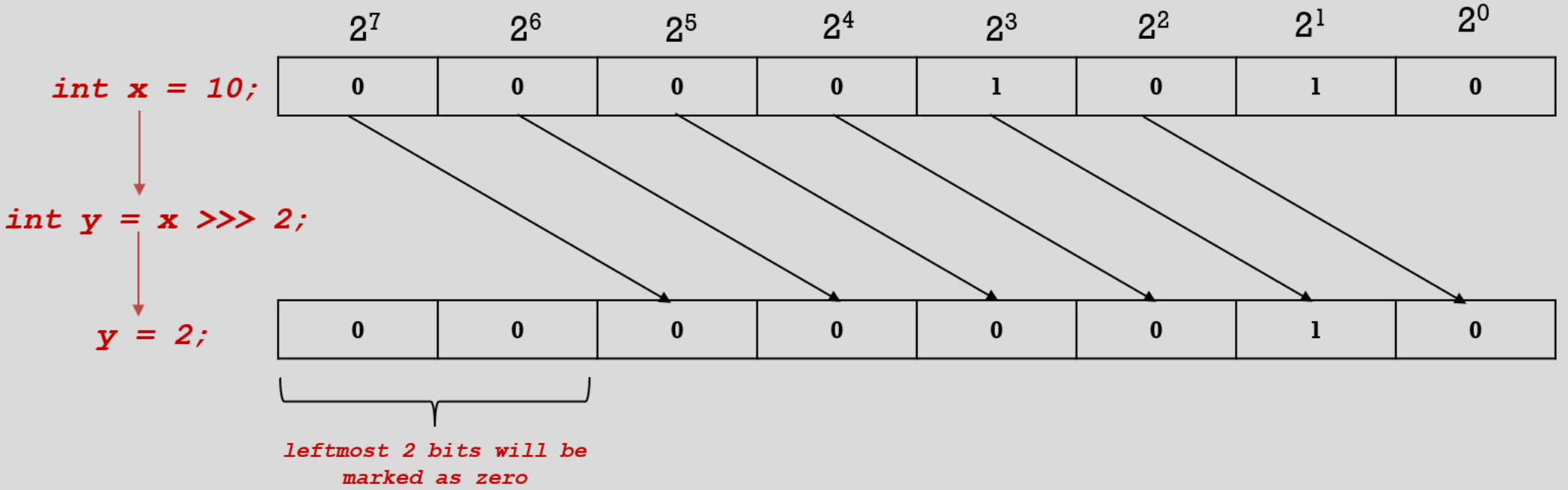
Unsigned right shift Operator (>>>)

eazy bytes

In Java, the unsigned right shift operator (>>>) is a bitwise operator that shifts the bits of its first operand to the right by the number of positions specified by its second operand, and fills the leftmost bits with zeros.

Below is an example code on how to use unsigned right shift operator,

```
int x = 10;
int y = x >>> 2; // 2
```



The >>> operator behaves similarly to the >> operator, but with one key difference: it always fills the high-order bits of the result with zeros, regardless of the sign of the value being shifted. This approach is known as "zero extension", and is commonly used when the left operand is being treated as an unsigned value.

Compound assignment bitwise operators

eazy bytes

Below is the syntax that needs to be followed while using compound bitwise assignment operator,

operand1 op= operand2

In the provided context, where "op" represents one of the bitwise operators such as &, |, ^, <<, >>, and >>>, and "operand1" and "operand2" are primitive integral data types with "operand1" being a variable, the preceding expression can be expressed equivalently as follows:

operand1 = (Type of operand1) (operand1 op operand2)

Operator	Equivalent expression
x &= y	x = (Type of x) (x & y)
x |= y	x = (Type of x) (x | y)
x ^= y	x = (Type of x) (x ^ y)
x <<= y	x = (Type of x) (x << y)
x >>= y	x = (Type of x) (x >> y)
x >>>= y	x = (Type of x) (x >>> y)

Ternary / Conditional operator

eazy bytes

The conditional operator in Java is represented by the symbol "?:". It is also known as the ternary operator because it takes three operands: a boolean expression followed by a question mark (?), then an expression to be evaluated if the boolean expression is true, followed by a colon (:) and finally another expression to be evaluated if the boolean expression is false. The syntax of the conditional operator is as follows:

boolean-expression ? true-expression : false-expression

If the boolean expression is true, the value of expression1 is returned; otherwise, the value of expression2 is returned.
For example, consider the following code snippet:

```
int x = 10;
int y = 20;
int max = (x > y) ? x : y;
System.out.println(max);
```

In this example, the boolean expression x > y is evaluated, which is false because x is not greater than y. Therefore, the value of y (which is 20) is assigned to the variable max. Finally, the value of max is printed, which is 20.

The conditional operator is a concise way of writing an if-else statement and is often used in place of an if-else statement for simple conditional expressions. However, it can make the code harder to read if it is overused or used unnecessarily.

Java operators Precedence & Associativity

eazy bytes

Java Operator Precedence is the order in which Java evaluates operators in an expression. When an expression contains more than one operator, the order of evaluation is determined by the precedence of the operators.

For example, consider the below Java expression,

```
int calNum = 16 - 8 * 2;
```

The value of calNum is uncertain. Will it be 16, as in (16 - 8)*2? Or will it be 0, as in 16 - (8 * 2)? When two operators share a common operand, 8 in this case, the * operator has a higher precedence than the - operator in Java. As a result, multiplication is performed before subtraction, leading to a calNum value of 0. If needed, the precedence can be changed using (). For example, we can change the above expression like shown below,

```
int calNum = (16 – 8) * 2;
```

Let's consider another example,

```
int calNum = 16 * 5 / 2;
```

The expression 16 * 5 / 2 involves two operators, namely a multiplication operator and a division operator, both having equal precedence. The evaluation of the expression proceeds from left to right. Initially, 16 * 5 is evaluated, followed by the evaluation of 80 / 2. Consequently, the entire expression results in 40. If the intention is to prioritize the division operation, parentheses must be employed. Parentheses, possessing the highest precedence, ensure that the expression within them is evaluated first.

```
int calNum =  16 * ( 5 / 2 ); // 32
```

Java operators Precedence & Associativity

eazy bytes

Associativity of operators in Java refers to the order in which operators of the same precedence are evaluated in an expression. There are two types of associativity:

Left-associative: Operators with left-associativity are evaluated from left to right. For example, the addition operator (+) is left-associative. In an expression like a + b + c, the addition is performed first on a and b, and then the result is added to c.

Right-associative: Operators with right-associativity are evaluated from right to left. The assignment operator (=) is an example of a right-associative operator. In an expression like a = b = c, the assignment is performed first on c and b, and then the result is assigned to a.

Java operators Precedence & Associativity

eazy bytes

Precedence & Associativity	operator	Operand type	Operation type
16 & Left - Right	.	object, member	object, member access
	[ ]	array, int	Array element access
	(args)	method, arglist	Method invocation
	++, --	variable	Post-increment, post-decrement
15 & Right - left	++, --	variable	Pre-increment, pre-decrement
	+, -	number	Unary plus, unary minus
	~	integer	Bitwise complement
	!	boolean	Boolean NOT

Java operators Precedence & Associativity

eazy bytes

Precedence & Associativity	operator	Operand type	Operation type
14 & Right - left	new	class, arglist	Object creation
	(type)	type, any	Cast (type conversion)
13 & Left - Right	*, /, %	number, number	Multiplication, division, remainder
12 & Left - Right	+, -	number, number	Addition, subtraction
	+	string, any	String concatenation
11 & Left - Right	<<	integer, integer	Left shift
	>>	integer, integer	Right shift with sign extension
	>>>	integer, integer	Right shift with zero extension

Java operators Precedence & Associativity

eazy bytes

Precedence & Associativity	operator	Operand type	Operation type
10 & Left - Right	<, <=	number, number	Less than, less than or equal
	>, >=	number, number	Greater than, greater than or equal
	instanceof	reference, type	Type comparison
9 & Left - Right	==	primitive, primitive	Equal (contains identical values ?)
	!=	primitive, primitive	Not Equal (contains non identical values ?)
	==	reference, reference	Equal (refer to same object)
	!=	reference, reference	Not equal (refer to different objects)

Java operators Precedence & Associativity

eazy bytes

Precedence & Associativity	operator	Operand type	Operation type
8 & Left - Right	&	integer, integer	Bitwise AND
	&	boolean, boolean	Boolean AND
7 & Left - Right	^	integer, integer	Bitwise XOR
	^	boolean, boolean	Boolean XOR
6 & Left - Right	|	integer, integer	Bitwise OR
	|	boolean, boolean	Boolean OR
5 & Left - Right	&&	boolean, boolean	Conditional AND
4 & Left - Right	||	boolean, boolean	Conditional OR

Java operators Precedence & Associativity

eazy bytes

Precedence & Associativity	operator	Operand type	Operation type
3 & Right - left	? :	boolean, any	Conditional (ternary) operator
2 & Right - left	=	variable, any	Assignment
	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	variable, any	Assignment with operation
1 & Right - left	→	arglist, method body	lambda expression