CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 3)**

Prepared by Renetha J.B.

Topics

- Core Java Fundamentals**:**

✓ **Operators –**

  - ✓ Arithmetic Operators,
  - ✓ Bitwise Operators,
  - ✓ Relational Operators,
  - ✓ Boolean Logical Operators,
  - ✓ Assignment Operator,
  - ✓ Conditional (Ternary) Operator,
  - ✓ Operator Precedence.

Operators

- Operators are used for performing operations.
  - **Arithmetic Operators**

    +, -, *, /, % ++, +=, -=,*=, /=, %=, - -
  - **Bitwise Operators** ~ Bitwise unary NOT

    &,| ,^ , >> ,>>> ,<< , &=, |=, ^=, >>=, >>>=
  - **Relational Operators**

    = =, !=, >, < , <=, >=
  - **Boolean Logical Operators**

    &, |, ^, ||,&&, &=,| =, ^=, ==, !=, ?:
  - **Assignment Operator**

    =
  - **Conditional (Ternary) Operator**

    ?:

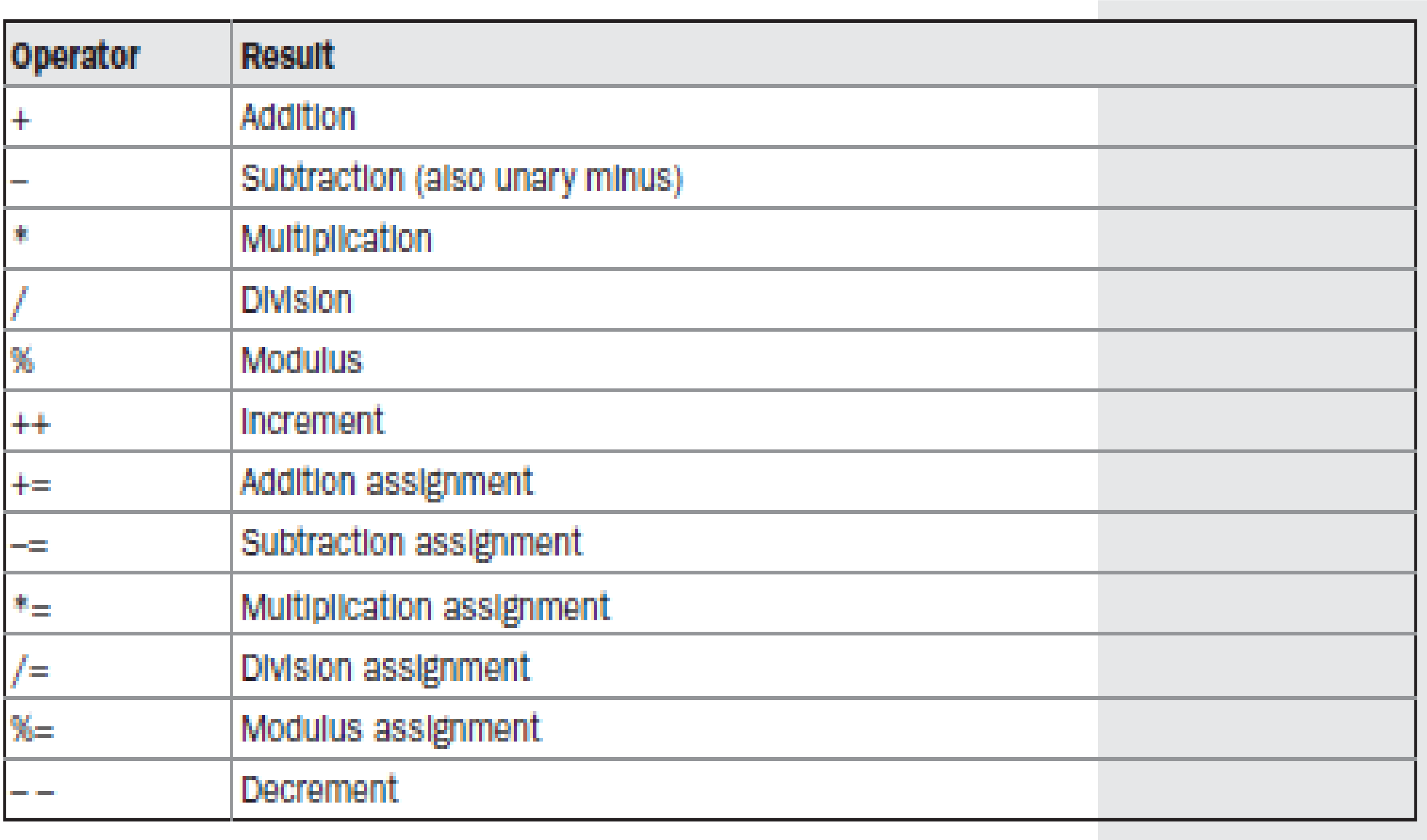
- Assignment Operator

  =

- Conditional (Ternary) Operator

  ?:

Arithmetic Operators



Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The Basic Arithmetic Operators

- The basic arithmetic operations—addition, subtraction, multiplication, and division works for all numeric types.
  - The minus operator also has a unary form that negates its single operand.
  - E.g

    ```
    int a=3;
    int b=-a;
    ```

Modulus Operator

- **The** Modulus Operator
- The modulus operator, **%, returns the remainder of a division operation. It can be applied to**
- floating-point types as well as integer types. The following example program demonstrates
- the **%:**

Arithmetic Compound Assignment Operators

- Variable operator = expression;

This is same as

Variable =Variable operator expression;

- In programming:

```
        a = a + 4;
```

can be written as

```
        a += 4;
```

E.g.

```
   int a=3;
   a+=2;     //Now value of a is 3+2=5
```

Prepared by Renetha J.B.

// Demonstrate the % operator.

```
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

When you run this program, you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Pre-Increment Post increment

- Pre increment E.g

```
x = 42;
y = ++x;
```

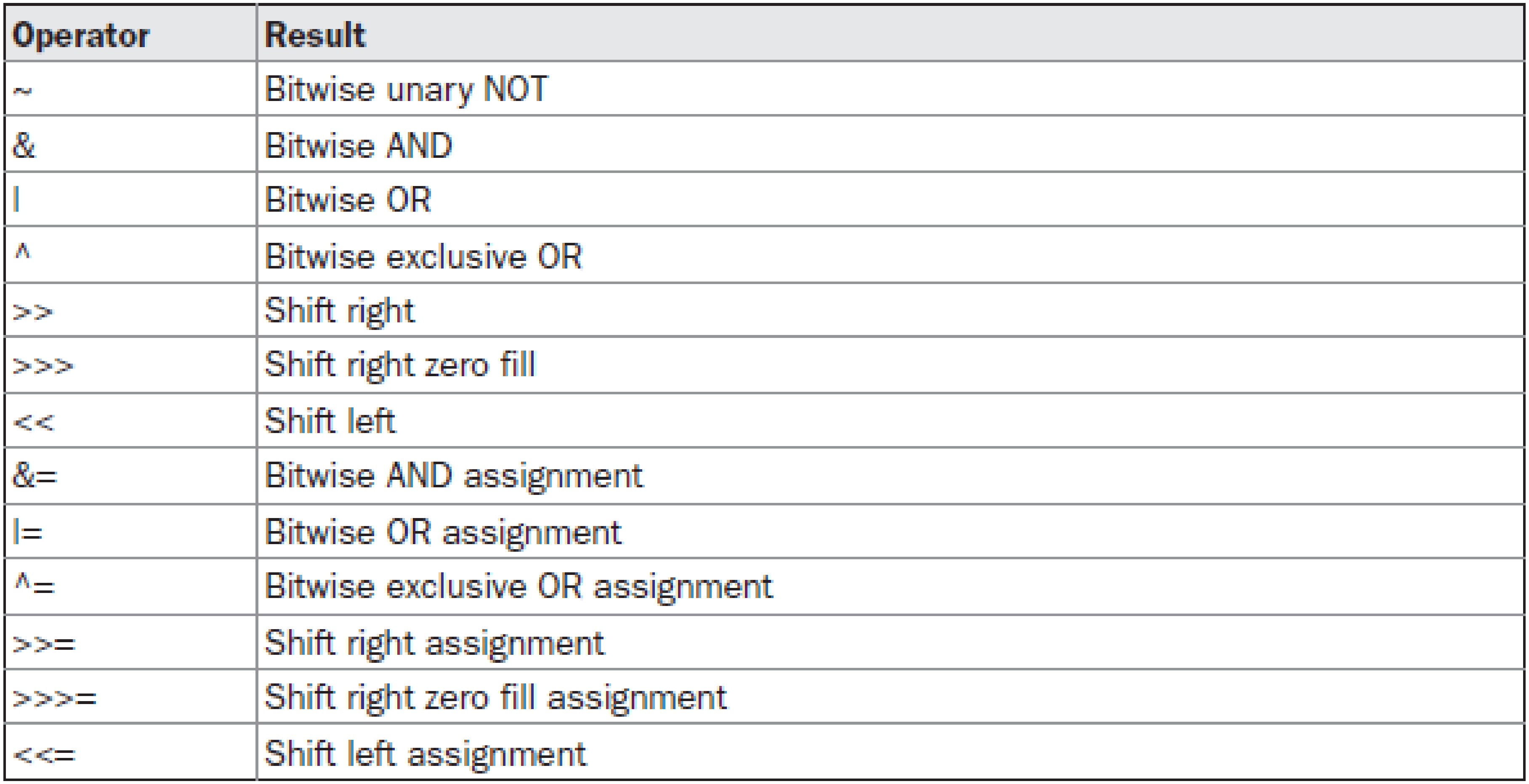
x 43
y 43

- Post increment E.g

```
x = 42;
y = x++;
```

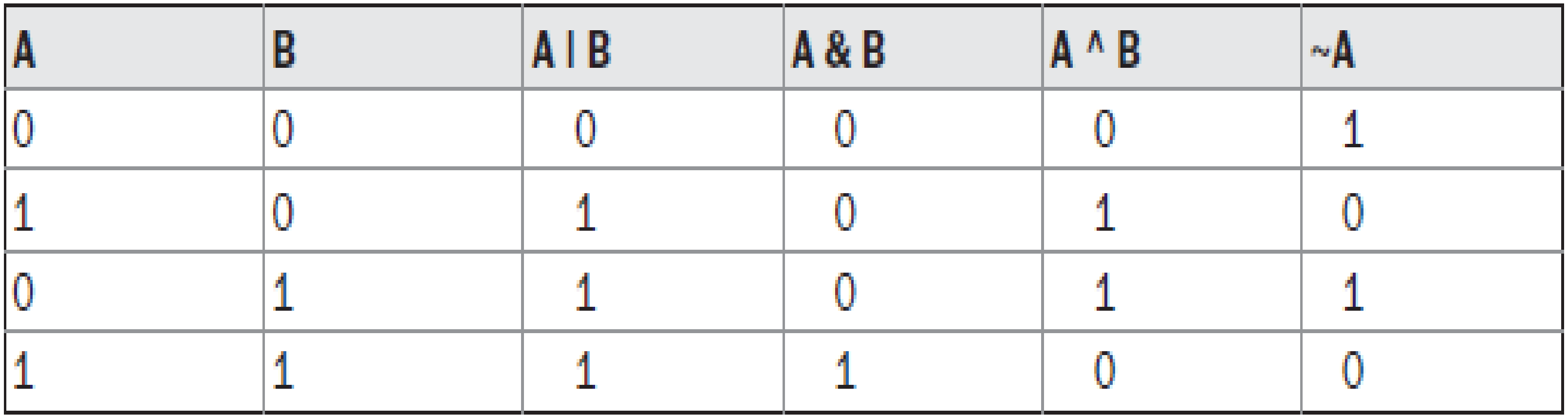
x 43
y 42

Bitwise operators



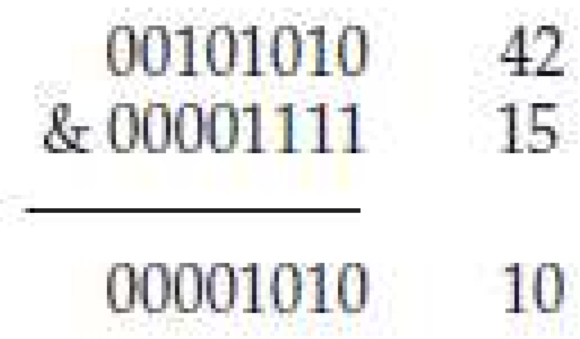
Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise logical oprators

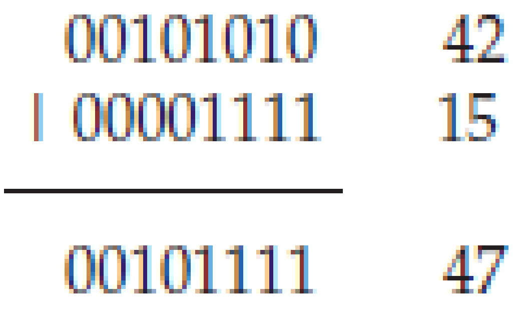


A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

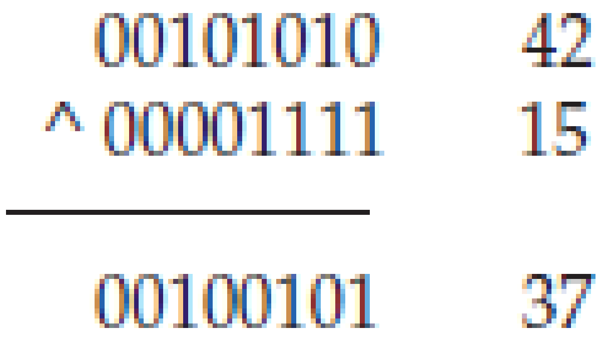
Examples



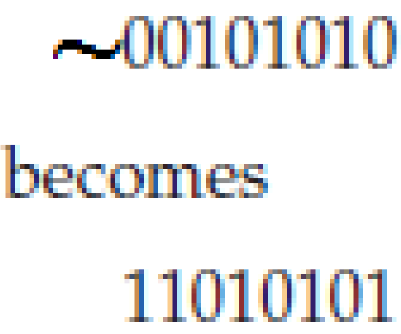
```
  00101010     42
& 00001111     15
__________
  00001010     10
```



```
  00101010     42
| 00001111     15
__________
  00101111     47
```



```
  00101010     42
^ 00001111     15
__________
  00100101     37
```



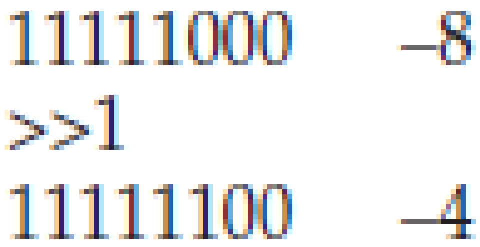
```
~00101010
becomes
 11010101
```

Prepared by Renetha J.B.

Right shift

- Each time you shift a value to the right, it divides that value by two—and discards any remainder.

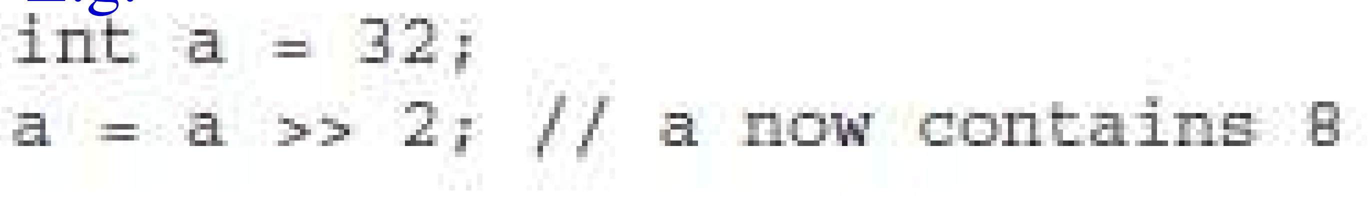
- When you are shifting right, **the top (leftmost) bits** exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension and serves to preserve* the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4



```
11111000    –8
>>1
11111100    –4
```

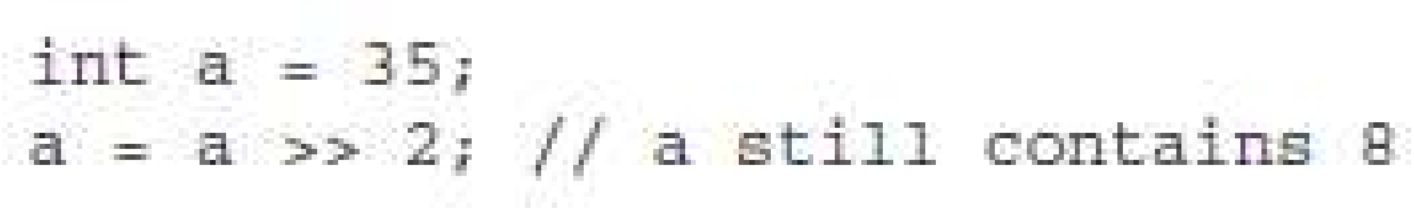
Right shift e.g

- E.g.

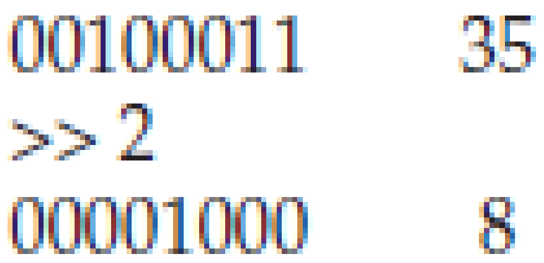


```
int a = 32;
a = a >> 2; // a now contains 8
```

- E.g.



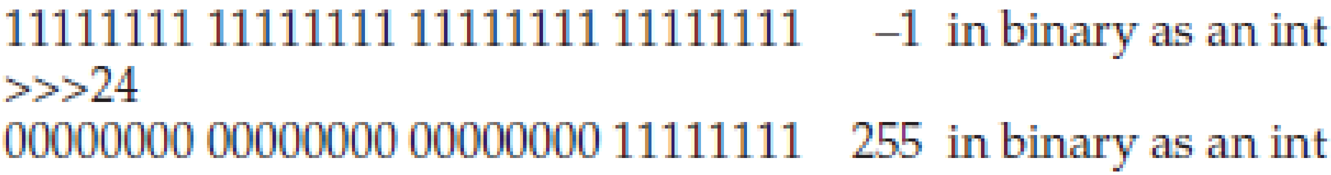
```
int a = 35;
a = a >> 2; // a still contains 8
```



```
00100011     35
>> 2
00001000      8
```

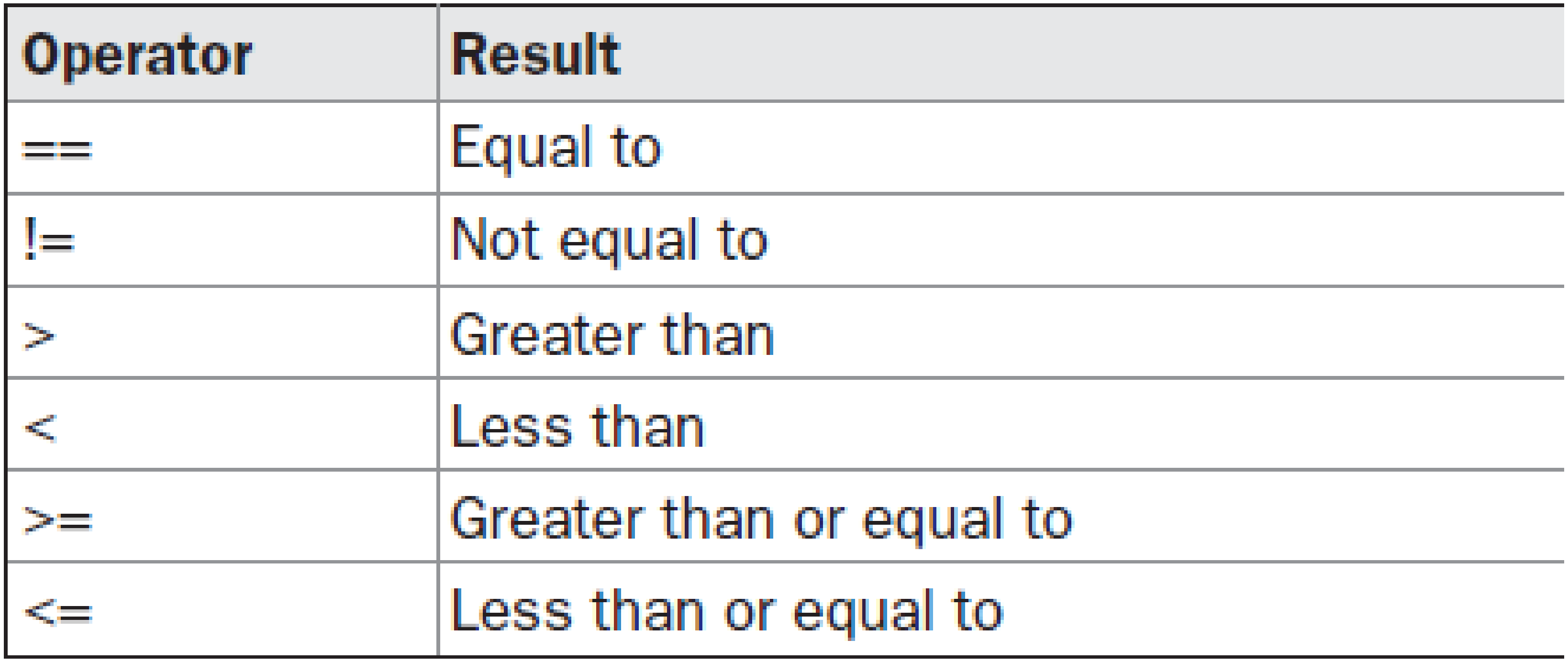
Unsigned, shift-right operator, >>>

- Shift a zero into the high-order bit(letftmost or top) no matter what its initial value was. This is known as an *unsigned shift.*
- Java's unsigned, shift-right operator, **>>>** always shifts zeros into the high-order bit.
- E.g **a is set to –1**, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.



```
11111111 11111111 11111111 11111111    –1  in binary as an int
>>>24
00000000 00000000 00000000 11111111   255  in binary as an int
```

Relational operators



Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Relational operator(contd.)

```
int a = 4;
int b = 1;
boolean c = a < b;     //c contains false. 4 is not less than 1
```

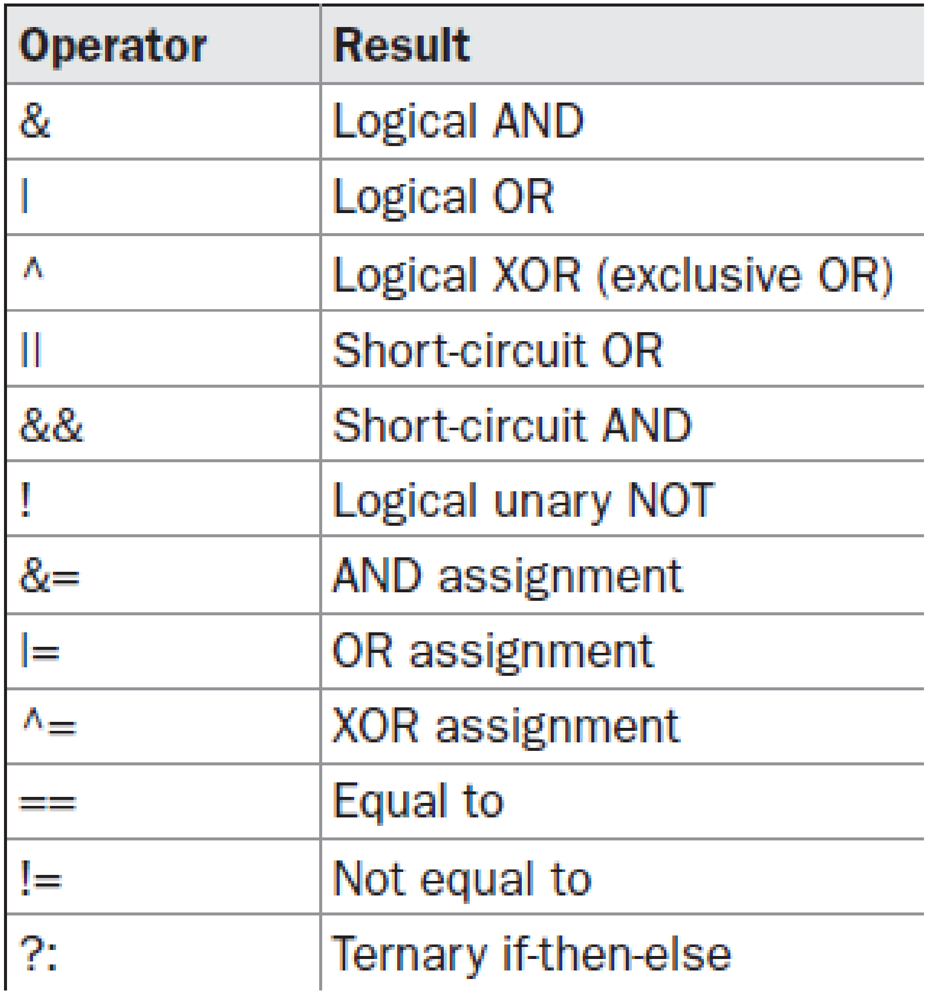
Here the result of **a<b** (which is **false**) is stored in c.

E.g.

```
int done;
// ...
if(!done) ... // Valid in C/C++
if(done) ... // but not valid  in Java.
```

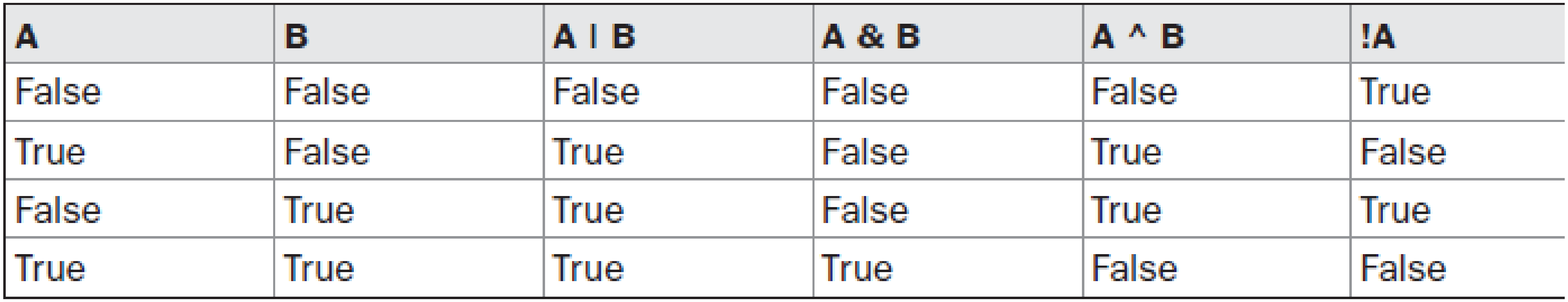
```
if(done == 0) ... // This is Java-style.
if(done != 0) ...
```

Boolean Logical Operators



Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

- The logical Boolean operators, **&, |, and ^, operate on boolean values in the same way** that they operate on the bits of an integer.



A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Short-Circuit Logical Operators

- Secondary versions of the Boolean AND and OR operators, and are known as ***short-circuit logical operators.***
- The OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is.
- If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the **outcome of the expression can be determined by the left operand alone**.

Short-Circuit Logical Operators(E.g)

- E.g

if (**denom != 0** && num / denom > 10)

- Here if denom is 0 the second expression is not validated
  - So there is no risk of causing a run-time exception when denom is zero.
- If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when denom is zero.

Assignment Operator

- *var = expression;*
- Here, the type of *var must be compatible with the type of expression.*
- It allows you to create a chain of assignments

int x, y, z;

x = y = z = 100; // set x, y, and z to 100

Ternary (conditional or three-way) operator

- The ? Operator has this general form:

expression1 ? expression2 : expression3

- Here, expression1 can be any expression that evaluates to a boolean value.
  - If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.
  - The result of the ? operation is that of the expression evaluated.
  - Both expression2 and expression3 are required to return the same type, which can't be void

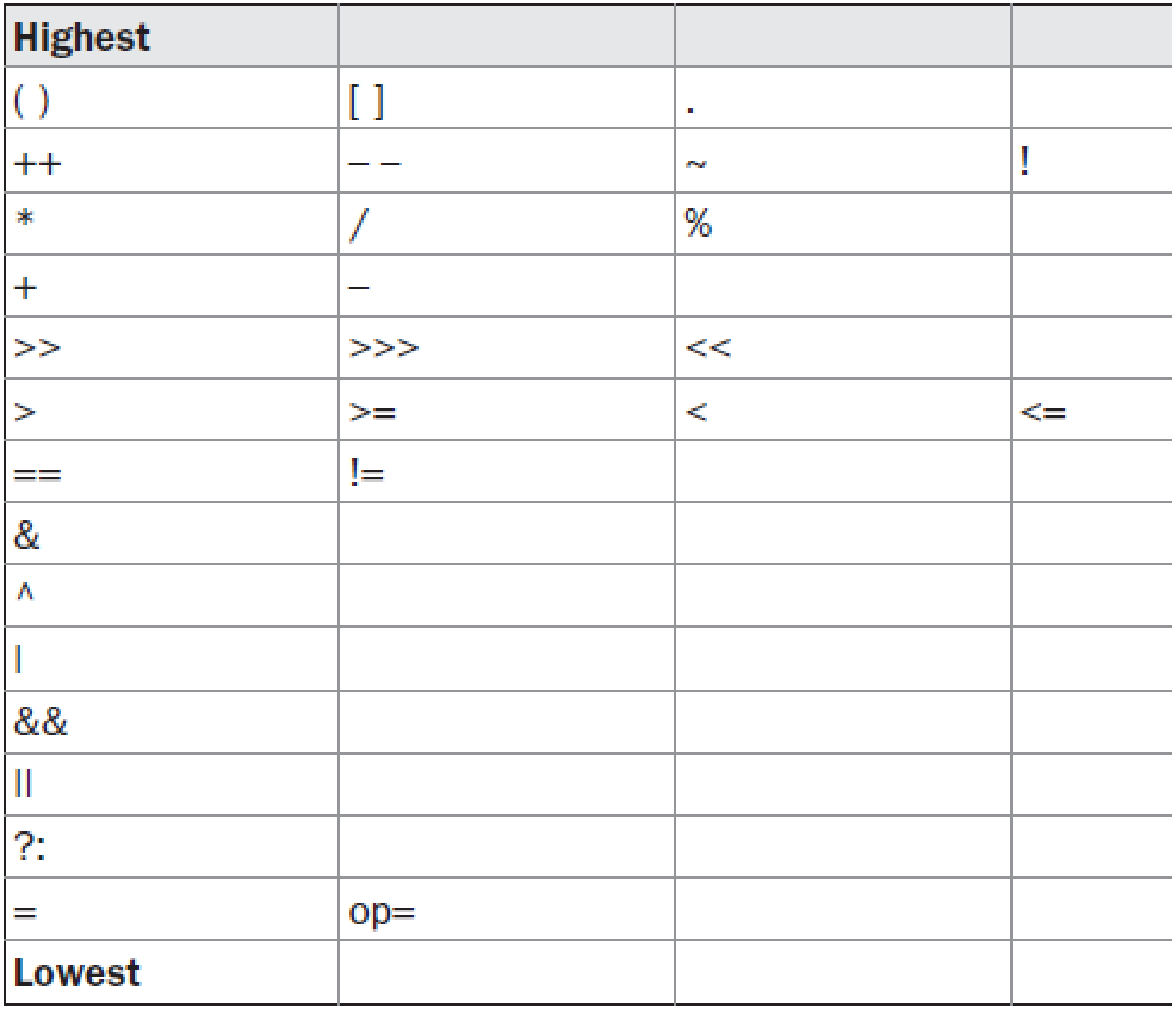
E.g.

- int ratio = denom == 0 **?** 0 : **num / denom**;
  - If denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? expression.
    - Here 0 is stored in ratio
  - If denom does **not equal zero**, then the expression after the colon is evaluated and used for the value of the entire ? expression.
  - i.e num/denom is stored in ratio
- The result produced by the ? operator is then assigned to ratio.

```
int a=3,b=5;
int c=(a>b?a:b);
```

- Here a>b is **false** so the value of b is stored in c.

Operator Precedence

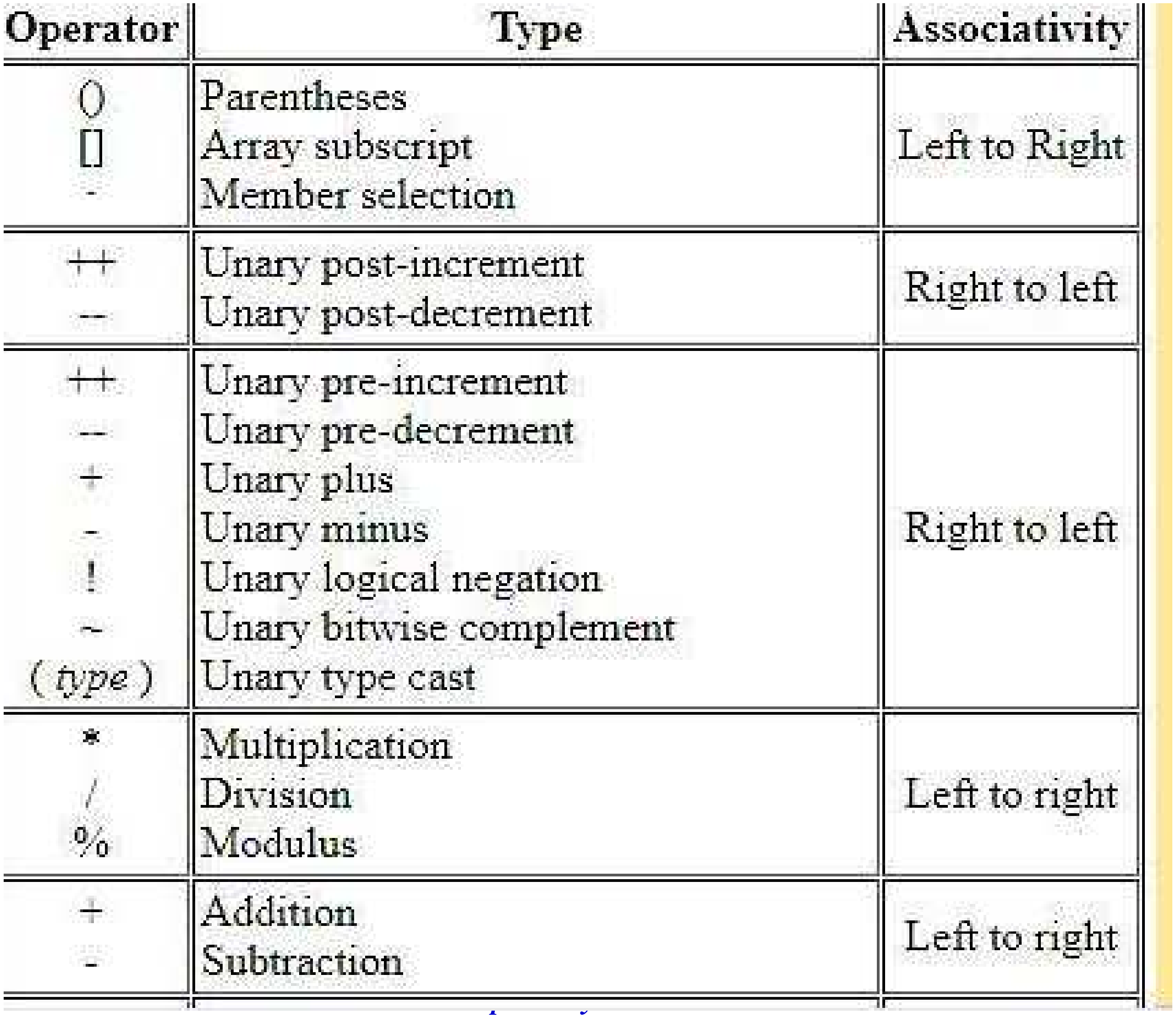


Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

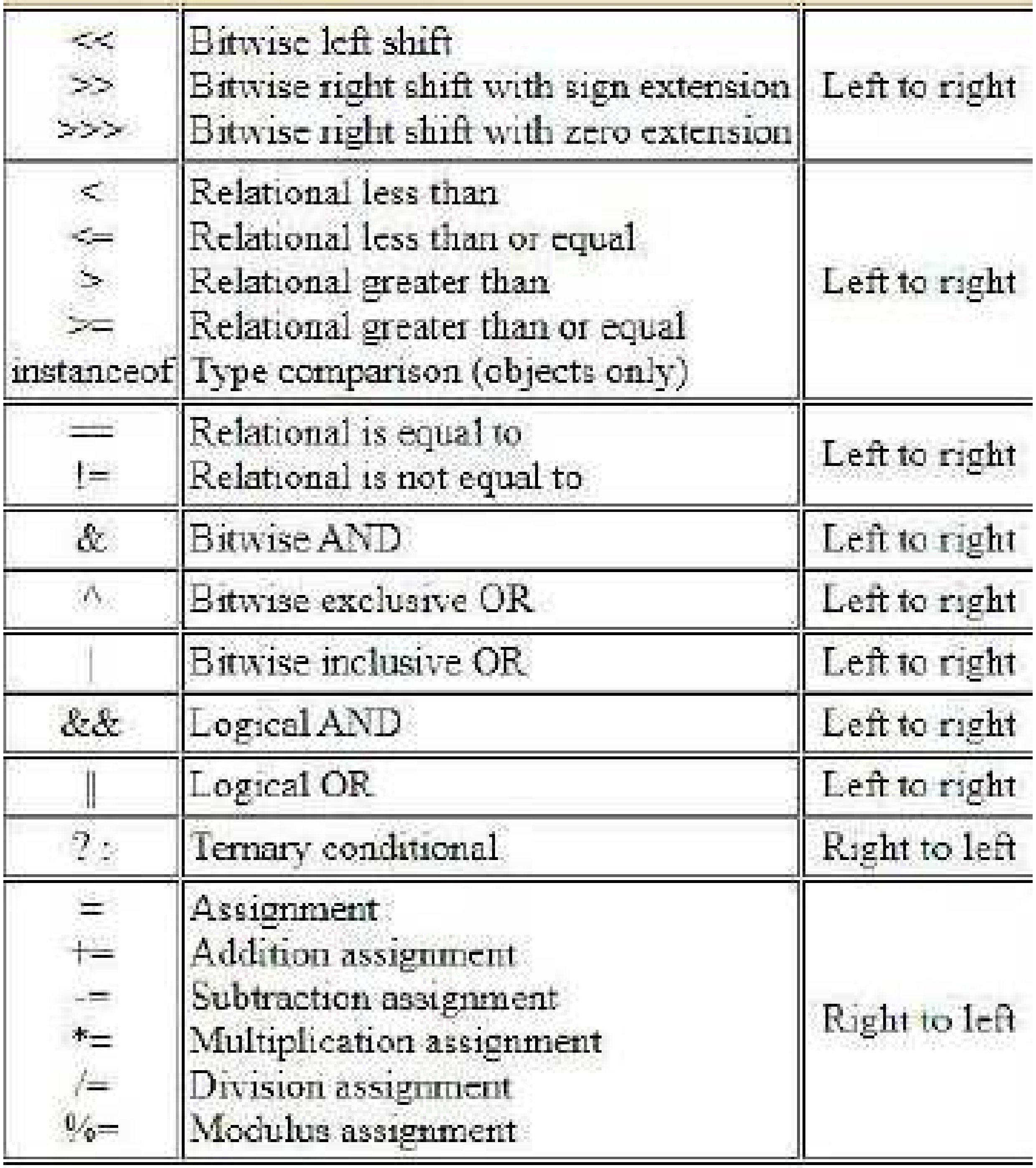
Associativity of operators

- When an expression has two or more operators with the same precedence, the expression is evaluated according to its **associativity**.
    - It is the order of applying operators

Operator Associativity



Operator	Type	Associativity
() [] .	Parentheses Array subscript Member selection	Left to Right
++ --	Unary post-increment Unary post-decrement	Right to left
++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
* / %	Multiplication Division Modulus	Left to right
+ -	Addition Subtraction	Left to right



<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
== !=	Relational is equal to Relational is not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Ternary conditional	Right to left
= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Associativity

- Right to Left associative
  - Unary operators
  - Assignment operators
  - Conditional(ternary) operators)
- All other operators are Left to Right associative

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.