Tipos primitivos, operadores, y sentencias de control de flujo

Java 21

JoeDayz 2025

Objectives

After completing this lesson, you should be able to:

- Describe primitive types
- Describe operators
- Explain primitives type casting
- Use the `Math` class
- Implement flow control with `if/else` and `switch` statements
- Describe JShell

Java Primitives

Java provides eight primitive types to represent simple numeric, character, and Boolean values.

Whole numbers			
byte	**short**	**int**	**long**
8 bits	16 bits	32 bits	64 bits
-128	-32,768	-2,147,483,648	-9,223,372,036,854,780,000
127	32,767	2,147,483,647	9,223,372,036,854,780,000
default value `0` or `0L`			
Binary `0b1001` Octal `072` Decimal `1234` Hex `0x4F` Upper or lowercase `L` at the end indicates `long` value			

Floating point numbers	
float	**double**
32 bits	64 bits
1.4E-45	4.9E-324
3.4028235E+38	1.79769313486231570E+308
default value `0.0F` or `0.0`	
Normal `123.4` or exponential notations `1.234E2`; Upper or lowercase `F` at the end indicates `float` value	

Boolean
boolean
default value `false`
`true` or `false`

Character (represents a single character value)
char
16 bits
0
65,535
default value `'\u0000'`
Character `'A'` ASCII code `'\101'` Unicode `'\u0041'` Escape Sequences: tab `'\t'` backspace `'\b'` new line `'\n'` carriage return `'\r'` form feed `'\f'` single quote `\'` double quote `\"` backslash `\\`

Declaring and Initializing Primitive Variables

- Variable declaration and initialization syntax:
  `<type> <variable name> = <value>;`
- A variable can be declared with no immediate initialization, as long as it is initialized before use.
- Numeric values can be expressed as binary, octal, decimal, and hex.
- Float and double values can be expressed in normal or exponential notations.
- Multiple variables of the same type can be declared and initialized simultaneously.
- Assignment of one variable to another creates a copy of a value.
- Smaller types are automatically promoted to bigger types.
- Character values must be enclosed in single quotation marks.

```
int a = 0b101010; // binary
short b = 052;    // octal
byte c = 42;      // decimal
long d = 0x2A;    // hex
float e = 1.99E2F;
double f = 1.99;
```

```
long a = 5, b = 3;
float c = a;
char d = 'A';
char e = '\u0041', f = '\101';
int g;
g = 77;
```

Primitive Declarations and Initializations: Restrictions

- Variables must be initialized before use.
- A bigger type value cannot be assigned to a smaller type variable.
- Character values must not be enclosed in double quotation marks.
- A character value cannot contain more than one character.
- Boolean values can be expressed only as `true` or `false`.

```
byte a;
byte b = a;
byte c = 128;
int d = 42L;
float e = 1.2;
char f = "a";
char g = 'AB';
boolean h = "true";
boolean i = 'false';
boolean j = 0;
boolean k = False;
```

- **Note:** Each incorrect example given here would cause Java code to **not** compile.

Java Operators

Java operators in the order of precedence:

Operators	Precedence
postfix increment and decrement	`++ --`
prefix increment and decrement, and unary	`++ -- + - ~ !`
multiplicative	`* / %`
additive	`+ -`
bit shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Assignment and Arithmetic Operators

Assignments and arithmetics

```
= + - * / %
```

```
int a = 1;   // assignment (a is 1)
int b = a+4; // addition (b is 5)
int c = b-2; // subtraction (c is 3)
int d = c*b; // multiplication (d is 15)
int e = d/c; // division (e is 5)
int f = d%6; // modulus (f is 3)
```

Compound assignments are combinations of an operation and an assignment that is acting on the same variable.

```
+= -= *= /= %=
```

```
int a = 1, b = 3;
a += b; // equivalent of a=a+b (a is 4)
a -= 2; // equivalent of a=a-2 (a is 2)
a *= b; // equivalent of a=a*b (a is 6)
a /= 2; // equivalent of a=a/2 (a is 3)
a %= a; // equivalent of a=a%a (a is 0)
```

Operator evaluation order can be changed using round brackets.

```
( )
```

```
int a = 2, b = 3;
int c = b-a*b;   // (c is -3)
int d = (b-a)*b; // (c is 3)
```

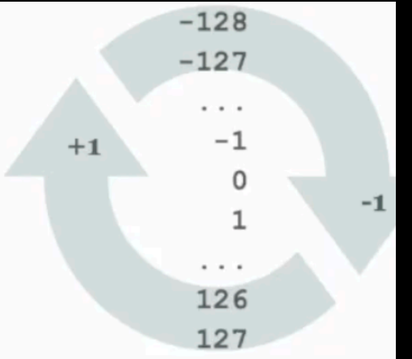
Increment and decrement operators have prefix and postfix positions:

`y=++x` x is incremented first and then the result is assigned to y.
`y=--x` x is decremented first and then the result is assigned to y.
`y=x++` y is assigned the value of x first and then x is incremented.
`y=x--` y is assigned the value of x first and then x is decremented.

```
++ --
```

```
int a = 1, b = 0;
a++;      // increment (a is 2)
++a;      // increment (a is 3)
a--;      // decrement (a is 2)
--a;      // decrement (a is 1)
b = a++; // increment postfix (b is 1, a is 2)
b = ++a; // increment prefix  (b is 3, a is 3)
b = a--; // decrement postfix (b is 3, a is 2)
b = --a; // decrement prefix  (b is 1, a is 1)
```

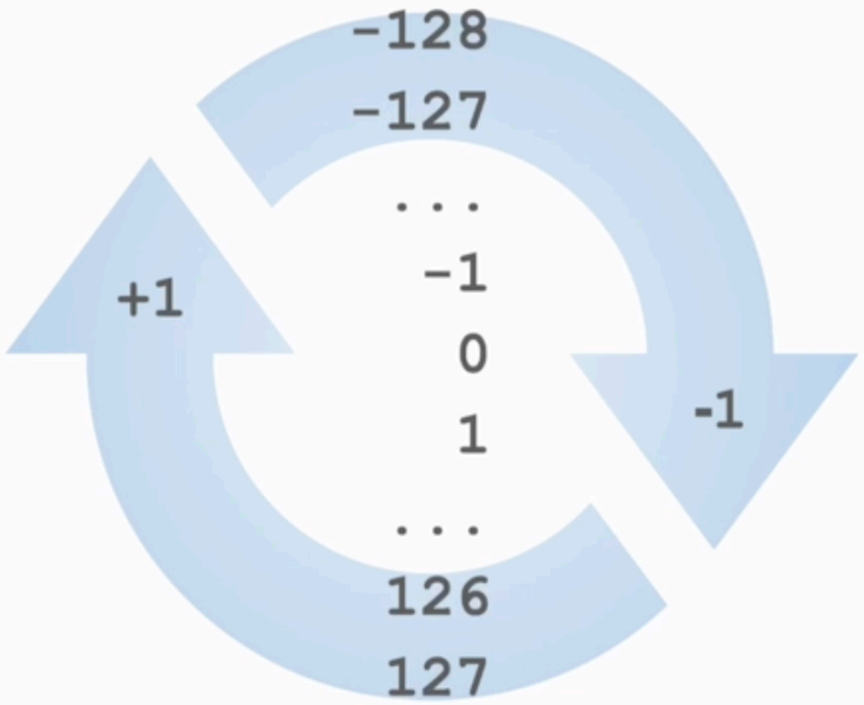
Arithmetic Operations and Type Casting



Rules of Java arithmetic operations and type casting:

- Smaller types are automatically casted (promoted) to bigger types.
  `byte->short->char->int->long->float->double`
- A bigger type value cannot be assigned to a smaller type variable without explicit type casting.
- Type can be explicitly casted using the following syntax: `(<new type>)<variable or expressi`
- When casting a bigger value to a smaller type, beware of a possible overflow.
- Resulting type of arithmetic operations on types smaller than int is an int; otherwise, the result is of a type of a largest participant.

```
  byte a = 127, b = 5;
⊗ byte c = a+b;             // compilation fails
             int
✓ int d = a + b;            // d is 132
△ byte e = (byte)(a+b);     // e is -124  (type overflow, because 127 is the max byte value)
△ int f = a/b;              // f is 25    (a/b is 25 because it is an int)
△ float g = a/b;            // g is 25.0F (result of the a/b can be implicitly or
△ float h = (float)(a/b);   // h is 25.0F  explicitly casted to float, but a/b is still 25)
✓ float i = (float)a/b;     // i is 25.4F (when either a or b
✓ float j = a/(float)b;     // j is 25.4F  is float the a/b becomes float)
△ b = (byte)(b+1);          // explicit casting is required, because b+1 is an int
✓ b++;                      // no casting is required for ++ and -- operators
  char x = 'x';
✓ char y = ++x;             // arithmetic operations work with character codes
```


-128
-127
...
-1
0
1
...
126
127
+1
-1

Arithmetic Operations and Type Casting

Rules of Java arithmetic operations and type casting:

- Smaller types are automatically casted (promoted) to bigger types.
  `byte->short->char->int->long->float->double`
- A bigger type value cannot be assigned to a smaller type variable without explicit type casting.
- Type can be explicitly casted using the following syntax: `(<new type>)<variable or expression>`
- When casting a bigger value to a smaller type, beware of a possible overflow.
- Resulting type of arithmetic operations on types smaller than int is an int; otherwise, the result is of a type of a largest participant.

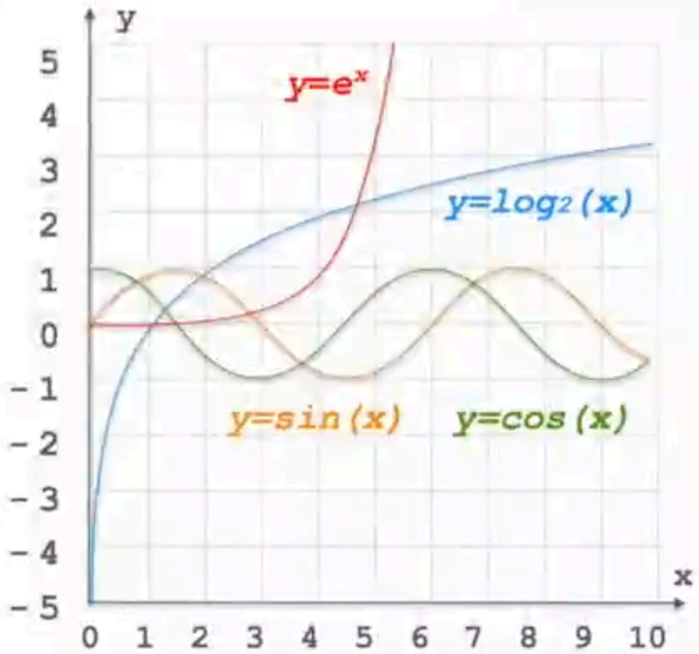
```
byte a = 127, b = 5;
byte c = a+b;             // compilation fails
         int
int d = a + b;            // d is 132
byte e = (byte)(a+b);     // e is -124  (type overflow, because 127 is the max byte value)
int f = a/b;              // f is 25    (a/b is 25 because it is an int)
float g = a/b;            // g is 25.0F (result of the a/b can be implicitly or
float h = (float)(a/b);   // h is 25.0F  explicitly casted to float, but a/b is still 25)
float i = (float)a/b;     // i is 25.4F (when either a or b
float j = a/(float)b;     // j is 25.4F  is float the a/b becomes float)
b = (byte)(b+1);          // explicit casting is required, because b+1 is an int
b++;                      // no casting is required for ++ and -- operators
char x = 'x';
char y = ++x;             // arithmetic operations work with character codes
```

Demo: operators

More Mathematical Operations

Class `java.lang.Math` provides various mathematical operations such as:

- Exponential such as e^x
- Logarithmic such as $\log_2(x)$
- Trigonometric such as `sin(x)` `cos(x)`



Examples of Math functions:

```
double a = 1.99, b = 2.99, c = 0;
c = Math.cos(a);    // cosine
c = Math.acos(a);   // arc cosine
c = Math.sin(a);    // sine
c = Math.asin(a);   // arc sine
c = Math.tan(a);    // tangent
c = Math.atan(a);   // arc tangent
c = Math.exp(a);    // e^a
c = Math.max(a,b);  // greater of two values
c = Math.min(a,b);  // smaller of two values
c = Math.pow(a,b);  // a^b
c = Math.sqrt(a);   // square root
c = Math.random();  // random number 0.0>=c<1.0
```

Numeric rounding example :

```
int a = 11, b = 3;
long c = Math.round(a/b);                          // c is 3
double d = Math.round(a/b);                        // d is 3.0
double e = Math.round((double)a/b*100)/100.0;      // e is 3.67
```

Binary Number Representation

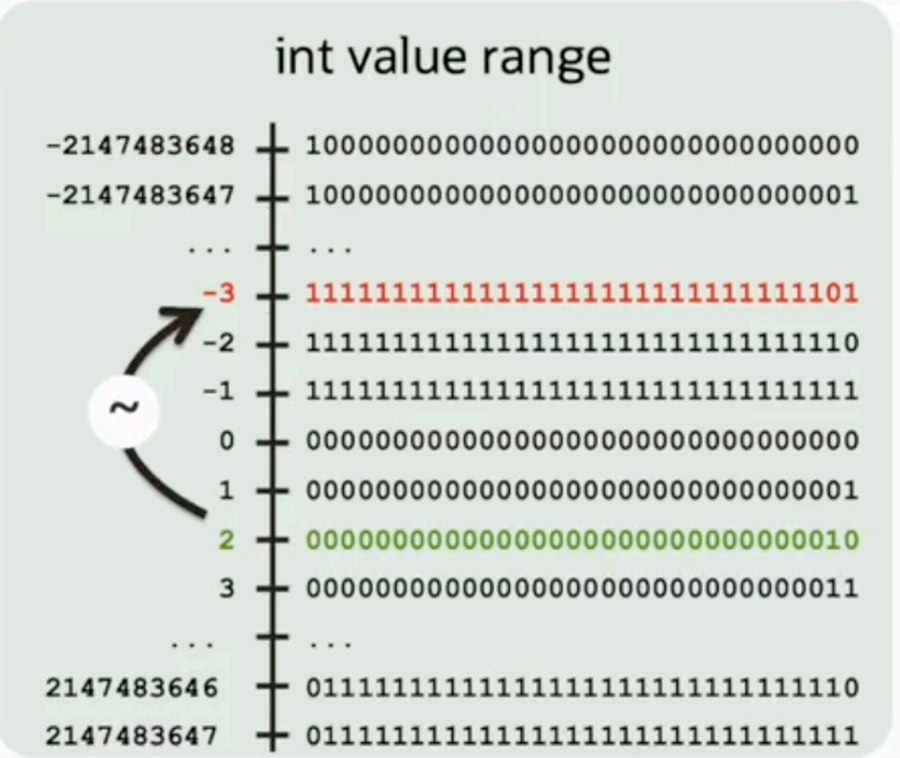
All Java numeric primitives are signed (*that is, could represent positive and negative values*).

- Java uses a two's complement implementation of a signed magnitude representation of an integer.
- For example, a byte zero value is represented as the `00000000` sequence of bits.
- Changing a sign (negative or positive) is done by inverting all the bits and then adding one to the result.
- For example, a byte value of one is represented as `00000001` and minus one is `11111111`.

The ***bitwise complement*** operator inverts all bits of a number: The result of the bitwise complement operator **`~a`** would be its "mirrored" binary value: `-(a+1)`.

```
int a = 2;  //
int b = ~a; // b is -3
```

✣ The next slide shows the rest of the bitwise operators.



Bitwise Logical Operators

Compare corresponding bits of two operands with bitwise operators:

- ***Bitwise AND*** `&` when both bits are `1`, the result is `1`, or either of the bits is not `1`, the result is `0`.
- ***Bitwise OR*** `|` when either of the bits is `1`, the result is `1`; otherwise, the result is `0`.
- ***Bitwise Exclusive OR*** `^` when corresponding bits are different, the result is `1`; otherwise, the result is `0`.

```
byte a = 5;               // 00000101
byte b = 3;               // 00000011
byte c = (byte)(a & b);   // 00000001 (c is 1)
byte d = (byte)(a | b);   // 00000111 (d is 7)
byte e = (byte)(a ^ b);   // 00000110 (e is 6)
```

Shift bits to the left or right with bitwise operators:

- ***Signed Left Shift*** `<<` shifts each bit to the left by specified number of positions, fills low-order positions with `0` bit values.
- ***Signed Right Shift*** `>>` shifts each bit to the right by specified number of positions.
- ***Unsigned Right Shift*** `>>>` is the same as above, but fills high-order positions with `0` bit values.

```
int a =  5;         // 00000000000000000000000000000101
int b = -5;         // 11111111111111111111111111111011
int c =  a << 2;    // 00000000000000000000000000010100 (c is 20)
int d =  b << 2;    // 11111111111111111111111111101100 (d is -20)
int e =  a >> 2;    // 00000000000000000000000000000001 (e is 1)
int f =  b >> 2;    // 11111111111111111111111111111110 (f is -2)
int g =  a >>> 2;   // 00000000000000000000000000000001 (e is 1)
int h =  b >>> 2;   // 00111111111111111111111111111110 (h is 1073741822)
```

Equality, Relational, and Conditional Operators

Compare values to determine the Boolean result:

- ***Equal to*** ==
- ***Not equal to*** !=
- ***Greater than*** >
- ***Greater than or equal to*** >=
- ***Less than*** <
- ***Less than or equal to*** <=
- ***NOT*** ! *(boolean inversion)*
- ***AND*** && &
- ***OR*** || |
- ***Exclusive OR*** ^

```
== != > >= < <= !
```

```
&& ||
```
(short-circuit evaluation)

```
& | ^
```
(full evaluation)

```
int a = 3, b = 2;
boolean c = false;
c = (a == b);           // c is false
c = !(a == b);          // c is true
c = (a != b);           // c is true
c = (a > b);            // c is true
c = (a >= b);           // c is true
c = (a < b);            // c is false
c = (a <= b);           // c is false
c = (a > b && b == 2); // c is true
c = (a < b && b == 2); // c is false
c = (a < b || b == 2); // c is true
c = (a < b || b == 3); // c is false
c = (a > b ^ b == 2);  // c is false
```

Notes

- Round brackets `()` are not required but could improve code readability.
- The difference between full and short-circuit evaluation is explained in the next slide.

Short-Circuit Evaluation

Short-circuit evaluation enables you to not evaluate the right side of the `AND` and `OR` expressions, when the overall result can be predicted from the left-side value.

`&& ||` *(short-circuit evaluation)*

`& | ^` *(full evaluation)*

```
 true && evaluated
false && not evaluated
false &  evaluated
false || evaluated
 true || not evaluated
 true |  evaluated
 true ^  evaluated
false ^  evaluated
```

```
int a = 3, b = 2;
boolean c = false;
c = (a > b && ++b == 3); // c is true,  b is 3
c = (a > b && ++b == 3); // c is false, b is 3
c = (a > b || ++b == 3); // c is false, b is 4
c = (a < b || ++b == 3); // c is true,  b is 4
c = (a < b |  ++b == 3); // c is true,  b is 5
```

Note: It is not advisable to mix Boolean logic and actions in the same expression.

Flow Control Using `if/else` Construct

Conditional execution of the algorithm using the `if/else` construct:

- The `if` code block is executed when the Boolean expression yields true; otherwise `else` block is executed.
- The else clause is optional.
- There is no `else/if` operator in Java, but you can embed an `if/else` inside another `if/else` construct.

```
if (<boolean expression>) {
  /* logic to be executed when
     if expression yields true */
} else {
  /* logic to be executed when
     if expression yields false */
}
```

- **Note:** Compilation fails because a block containing more than one statement must be enclosed with `{}`.

```
int a = 2, b = 3;
if (a > b) {          // is false
  a--;                // not executed
} else {              // algorithm enters else block
  if (a < b) {        // is true
    a++;              // a is 3
  } else {            // this else block is not executed
    b++;              // not executed
  }
}
```

```
int a = 2, b = 3;
if (a > b)
  a--;
  b++;
else
  a++;
```

- **Note:** It is optional to put braces `{}` around `if` or `else` blocks of code, when they contain only a single statement. This code fragment is identical to the example above, but `{}` omissions could make it harder to read.
- **Note:** Carriage returns and indentations in Java improve readability, but are irrelevant from the compiler perspective.

```
int a = 2, b = 3;
if (a > b)
  a--;
else if (a < b)
  a++;
else
  b++;
```

Ternary Operator

The ternary operator is used to perform conditional assignment.

- You can use the ternary operator `?:` instead of writing an `if/else` construct, if you need to only assign a value based on a condition.
- When the Boolean expression yields true, the value after the `?` is assigned. When the Boolean expression yields false, the value after the `:` is assigned.

```
<variable> = (<boolean expression>) ? <value one> : <value two>;
```

```
int a = 2, b = 3;
int c = (a >= b) ? a : b; // c is 3
```

These constructs produce identical results.

```
int a = 2, b = 3;
int c = 0;
if (a >= b) {
  c = a;
} else {
  c = b;
}
```

- **Note:** The ternary operator should be used to simplify conditional assignment logic. Do not use it instead of `if/else` statements to perform other actions, because it can make your code less readable.

```
int a = 2, b = 3;
int c = (a >= b) ? a : (--b == a) ? a : b; // c is 2
```

Flow Control Using `switch` Construct

Control program flow using the `switch` construct.

- The switch statement expression must be of one of the following types: **byte, short, int, char, String, enum, Object**
- Case labels must match the expression type.
- Execution flow proceeds to the case in which the label matches the switch statement expression value.
- Execution flow continues until it reaches the end of the switch or encounters an optional break statement.
- If the switch statement expression did not match any of the cases, then the default case is executed. It does not have to be the last case in the sequence and it is optional.

```
switch (<expression>) {
  case <label>:
    <case logic>
  case <label>:
    <case logic>
    break;
  default:
    <case logic>
}
```

Example cases:

- **S**pecial, increase price by 1
- **N**ew, increase price by 2
- **D**iscounted, decrease price by 4
- **E**xpired, set price to 0
- Any other, set price to 3

Execution path within the switch when status is '**N**' (New)

```
char status = 'N';
double price = 10;
switch (status) {
  case 'S':
    price += 1; // not executed
  case 'N':
    price += 2; // price is 12
  case 'D':
    price -= 4; // price is 8
    break;
  case 'E':
    price = 0;  // not executed
    break;
  default:
    price = 3;  // not executed
}
//the rest of the program logic
```

✣ **Note**: The use of switch with Strings, enums and Objects cases is covered later in the course.

`Switch` Expressions `yield` a Value

A `switch` expressions are used to evaluate a value.

- Contain exhaustive set of cases that yields a value.
- A single-line expression does not require explicit use of a `yield` statement.
- A block requires a `yield` statement to return a value.
- An overall statement must be terminated with a `;` symbol.

```
char status = 'N';
double cost = 10;
double price = switch (status) {
                 case 'S', 'N' -> cost += 1;
                 case 'D' -> {
                   double discount = cost*0.2;
                   cost -= discount;
                   yield cost;
                 }
                 case 'E' -> 0;
                 default -> 3;
               };
```

Notes:

- `yield` has special meaning only within `switch` expressions.
- If a variable called `yield` exists elsewhere, that code won't break.

Switch Statements and Expressions Summary

Switch statement using : syntax

```
switch(...) {
  case <label> : {
// May use break otherwise will
// fall-through to the next case.
  }
  ...
}
```

Switch expression using : syntax

```
x = switch(...) {
  case <label> : {
// Must produce the value using yield or
// fall-through to the next case, but eventually
// must yield a value, or throw an exception.
// Must not use break.
// }
  ...
};
```

Switch statement using -> syntax

```
switch(...) {
  case <label> -> {
// May use break.
// No fall-through to the next case.
  }
  ...
}
```

Switch expression using -> syntax

```
x = switch(...) {
  case <label> -> {
// Must either produce result from a simple expression,
// or return a value using yield, or throw an exception.
// Must not use break.
// No fall-through to the next case.
  }
  ...
};
```

Quiz

1. Which is NOT a Java primitive?

- ○ long
- ○ String
- ○ short
- ○ float
- ○ boolean

2. What is the correct evaluation of this expression: 8*8/2+2-3*2 ?

- ○ 26
- ○ 30
- ○ 10
- ○ 28

3. Which is a correct way to declare and initialize a variable?

- ○ char midInit = "D";
- ○ int age;
- ○ boolean x = 4>5;
- ○ age = 42;

4. Given:

```
int x = 1, y = 1, z = 0;
if (x == y | x < ++y) {
  z = x+y;
}else{
  z = 1;
}
System.out.println(z);
```

What would be printed?

- ○ 2
- ○ 3
- ○ 1

5. When would you use the ternary operator (?:) instead of writing an if/else construct?

- ○ When the Boolean expression evaluated should be deferred
- ○ When a shorter expression results in a faster code execution
- ○ If you only need to assign a value based on a condition
- ○ To defer calculation of the value