# Operators

## 1. Understanding Java Operators

Un operador de Java es un símbolo especial que se puede aplicar a un conjunto de variables, valores o literales, conocidos como operandos, y que devuelve un resultado.

### 1.1. TYPES OF OPERATORS

Tres tipos de operadores están disponibles en Java: unario, binario y ternario. Los operadores de Java no se evalúan necesariamente de izquierda a derecha.

### 1.2. OPERATOR PRECEDENCE

A menos que se utilicen paréntesis, los operadores Java siguen el orden de operación, enumerados en la Tabla siguiente, al disminuir el orden de precedencia del operador. Si dos operadores tienen el mismo nivel de precedencia, entonces Java garantiza la evaluación de izquierda a derecha.

| Operador | Símbolos y ejemplos |
|---|---|
| Operadores Post-unario | `expression++, expression--` |
| Operadores Pre-unario | `++expression, --expression` |
| Otros operadores unarios | `+, -, !, (type)` |
| Multiplicación/División/Módulo | `*, /, %` |
| Adición/Sustracción | `+, -` |
| Operadores de desplazamiento | `<<, >>, >>>` |
| Operadores relacionales | `<, >, <=, >=, instanceof` |
| Igual a/ No igual a | `==, !=` |
| Operadores lógicos | `&, ^, \|` |
| Operadores lógicos de corto circuito | `&&, \|\|` |
| Operadores ternarios de expresión booleana | `? expression1 : expression2` |
| Operadores de asignación | `=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=` |

## 2. Applying Unary Operators

Por definición, un operador unario es aquel que requiere exactamente un operando o variable para funcionar.

| Operador unario | Descripción |
|---|---|
| + | Indica que un número es positivo, aunque se supone que los números son positivos en Java a menos que estén acompañados por un operador unario negativo. |
| - | Indica que un número literal es negativo o niega una expresión |
| ++ | Incrementa un valor en 1 |
| -- | Disminuye un valor en 1 |

| | |
|---|---|
| ! | Invierte un valor lógico booleano |

## 2.1. LOGICAL COMPLEMENT AND NEGATION OPERATORS

El operador de complemento lógico: `!`, invierte el valor de una expresión booleana. Por ejemplo, si el valor es verdadero, se convertirá en falso y viceversa. Por ejemplo:

```
boolean isAnimalAsleep = false;
System.out.println(isAnimalAsleep);  // false
isAnimalAsleep = !isAnimalAsleep;
System.out.println(isAnimalAsleep);  // true
```

Del mismo modo, el operador de negación, `-`, invierte el signo de una expresión numérica. Por ejemplo:

```
double zooTemperature = 1.21;
System.out.println(zooTemperature);  // 1.21
zooTemperature = -zooTemperature;
System.out.println(zooTemperature);  // -1.21
zooTemperature = -(-zooTemperature);
System.out.println(zooTemperature);  // -1.21
```

Según la descripción, puede ser obvio que algunos operadores requieren que la variable o expresión sobre la que actúan sea de un tipo específico. Por ejemplo, ninguna de las siguientes líneas de código compilará:

```
int pelican = !5;          // DOES NOT COMPILE
boolean penguin = -true;   // DOES NOT COMPILE
boolean peacock = !0;      // DOES NOT COMPILE
```

**Note**
A diferencia de otros lenguajes de programación, en Java, 1 y true no están relacionados de ninguna manera, al igual que 0 y false no están relacionados.

## 2.2. INCREMENT AND DECREMENT OPERATORS

Los operadores de incremento y decremento, `++` y `--`, respectivamente, se pueden aplicar a los operandos numéricos y tienen un orden o precedencia mayor, en comparación con los operadores binarios. En otras palabras, a menudo se aplican primero a una expresión.

Si el operador se coloca antes del operando, denominados operador de pre incremento y operador de decremento previo, entonces el operador se aplica primero y el retorno de valor es el nuevo valor de la expresión. Alternativamente, si el operador se coloca después del operando, denominados operador de incremento posterior y operador de decremento posterior, se devuelve el valor original de la expresión, con el operador aplicado después de que se devuelve el valor. Por ejemplo:

```
int parkAttendance = 0;
System.out.println(parkAttendance);    // 0
System.out.println(++parkAttendance);  // 1
System.out.println(parkAttendance);    // 1
System.out.println(parkAttendance--);  // 1
System.out.println(parkAttendance);    // 0
```

Ejemplos con operadores en una misma línea:

```
int lion = 3;
int tiger = ++lion * 5 / lion--;
System.out.println("lion is " + lion);
System.out.println("tiger is " + tiger);
```

El resultado sería:

```
lion is 3
tiger is 5
```

## 3. Working with Binary Arithmetic Operators

Comenzaremos nuestra discusión con los operadores binarios, de lejos los operadores más comunes en el lenguaje Java:

| Operador | Descripción |
|---|---|
| + | Suma dos valores numéricos |
| - | Resta dos valores numéricos |
| * | Multiplica dos valores numéricos |
| / | Divide un valor numérico por otro |
| % | El operador de módulo devuelve el resto después de la división de un valor numérico por otro |

### 3.1. ARITHMETIC OPERATORS

Los operadores aritméticos a menudo se encuentran en las operaciones matemáticas básicas e incluyen la suma (`+`), la resta (`-`), la multiplicación (`*`), la división (`/`) y el módulo (`%`). También incluyen operadores unarios, `++` y `--`, aunque los cubriremos más adelante. Como habrá notado en la Tabla anterior, los operadores multiplicativos (`*`, `/`, `%`) tienen un orden de precedencia mayor que los operadores aditivos (`+`, `-`). Por ejemplo:

```
int price = 2 * 5 + 3 * 4 - 8;
```

Se reduce a:

```
int price = 10 + 12 - 8;
```

Si agregamos paréntesis:

```
int price = 2 * ((5 + 3) * 4 - 8);
```

…luego:

```
int price = 2 * (8 * 4 - 8);
```

…luego:

```
int price = 2 * (32 - 8);
```

…y finalmente:

```
int x = 2 * 24;
```

Al trabajar con paréntesis, debe asegurarse de que siempre son válidos y balanceados. Por ejemplo:

```
long pigeon = 1 + ((3 * 5) / 3;        // DOES NOT COMPILE
int blueJay = (9 + 2) + 3) / (2 * 4;   // DOES NOT COMPILE
```

```
short robin = 3 + [(4 * 2) + 4];      // DOES NOT COMPILE, solo paréntesis
```

Asegúrese de entender la diferencia entre la división aritmética y el módulo. Para los valores enteros, la división da como resultado el valor del máximo entero más cercano que cumple la operación, mientras que el módulo es el valor restante. Por ejemplo:

```
System.out.print(9 / 3);  // Outputs 3
System.out.print(9 % 3);  // Outputs 0
System.out.print(10 / 3);  // Outputs 3
System.out.print(10 % 3);  // Outputs 1
```

Para un divisor `y` dado, que es 3 en estos ejemplos, la operación del módulo da como resultado un valor entre `0` y (`y - 1`) para dividendos positivos. Esto significa que el resultado de una operación de módulo es siempre 0, 1 ó 2 para los ejemplos.

**Nota**
La operación de módulo no está limitada a valores enteros positivos en Java, también se puede aplicar a enteros negativos y enteros de coma flotante. Para un divisor `y`, y un dividendo negativo dado, el valor del módulo resultante es entre `y (-y + 1)` y `0`. Sin embargo, para el examen, no es necesario conocer que se puede tomar el módulo de un número entero negativo o de punto flotante.

## 3.2. NUMERIC PROMOTION

Cada primitivo tiene una longitud de bit. No necesita saber el tamaño exacto de estos tipos para el examen, pero debe saber cuáles son más grandes que otros. Por ejemplo, debe saber que un `long` ocupa más espacio que un `int`, y que a su vez ocupa más espacio que un `short`, y así sucesivamente.

### 3.2.1. Numeric Promotion Rules

Reglas:

1. Si dos valores tienen tipos de datos diferentes, Java promocionará automáticamente uno de los valores al mayor de los dos tipos de datos.
2. Si uno de los valores es entero y el otro es de coma flotante, Java promocionará automáticamente el valor integral al tipo de datos del valor de coma flotante.
3. Los tipos de datos más pequeños, a saber, `byte`, `short` y `char`, se promueven primero a `int` en cualquier momento en que se usan con un operador aritmético binario Java, incluso si ninguno de los operandos es `int`.
4. Después de que se haya producido la promoción y los operandos tengan el mismo tipo de datos, el valor resultante tendrá el mismo tipo de datos que sus operandos promocionados.

Ejemplos:
¿Cuál es el tipo de datos de x * y?

```
int x = 1;
long y = 33;
```

Respuesta: `long`

¿Cuál es el tipo de datos de x + y?

```
double x = 39.21;
float y = 2.1;
```

Respuesta: `double`

¿Cuál es el tipo de datos de x * y / z?
```
short x = 14;
float y = 13;
double z = 30;
```
Respuesta: `double`

## 4. Assigning Values

Los errores de compilación de los operadores de asignación a menudo se pasan por alto en el examen. Para dominar los operadores de asignación, debe comprender con fluidez cómo el compilador maneja la promoción numérica y cuándo se requiere la conversión.

### 4.1. ASSIGNMENT OPERATOR

Un operador de asignación es un operador binario que modifica, o asigna, la variable en el lado izquierdo del operador, con el resultado del valor en el lado derecho de la ecuación. Por ejemplo:

```
int heard = 1;   // Asigna a x el valor de 1
```

Java promocionará automáticamente tipos de datos más pequeños a más grandes, pero lanzará una excepción de compilación si detecta que está tratando de convertir tipos de datos de mayor a menor.

## 5. CASTING VALUES

El *Casting* es una operación unaria en la que un tipo de datos se interpreta explícitamente como otro tipo de datos. La conversión es opcional e innecesaria cuando se convierte a un tipo de datos más grande o ampliado.

El *Casting* conversión se realiza colocando el tipo de datos, entre paréntesis, a la izquierda del valor que desea transmitir. Aquí hay algunos ejemplos de casting:

```
int fur = (int)5;
int hair = (short) 2;
String type = (String)  "Bird";
short tail = (short)(4 + 10);
long feathers = 10(long);  // DOES NOT COMPILE
```

Vea si puede averiguar por qué no se compila ninguna de las siguientes líneas de código:

```
float egg = 2.0 / 9;         // DOES NOT COMPILE
int tadpole = (int)5 * 2L;   // DOES NOT COMPILE
short frog = 3 - 2.0;        // DOES NOT COMPILE
```

Todos estos ejemplos implican poner un valor mayor en un tipo de datos más pequeño. Finalmente, lo siguientes ejemplos no compilan debido al tipo de dato:

```
int fish = 1.0;         // DOES NOT COMPILE
short bird = 1921222;   // DOES NOT COMPILE No alcande en short, muy grande
int mammal = 9f;        // DOES NOT COMPILE
long reptile = 1923013981938103 23;  // DOES NOT COMPILE, Java interpreta como int
```

Es posible corregir el ejemplo anterior al convertir los resultados a un tipo de datos más pequeño:

```
int trainer = (int)1.0;
short ticketTaker = (short)1921222;  // Stored as 20678
int usher = (int)9f;
long manager = 192301398193810323L;
```

**Overflow and Underflow**
1,921,222, es demasiado grande para almacenarse como un `short`, por lo que se produce un `overflow` numérico y se convierte en 20,678. El exceso es cuando un número es tan grande que ya no se ajusta al tipo de datos, por lo que el sistema "se adapta" al siguiente valor más bajo y cuenta desde allí. También hay un `underflow` análogo, cuando el número es demasiado bajo para ajustarse al tipo de datos.

Otro ejemplo:

```
short mouse = 10;
short hamster = 3;
short capybara = mouse * hamster;  // DOES NOT COMPILE
```

No compila debido a que `short` se promueven automáticamente a `int` al **aplicar cualquier operador aritmético**. Sin embargo, si necesita que el resultado sea `short`, se puede anular este comportamiento realizando un casting al resultado de la multiplicación:

```
short mouse = 10;
short hamster = 3;
short capybara = (short)(hamster * hamster);
```

Sin embargo:

```
hort mouse = 10;
short hamster = 3;
short capybara = (short)mouse * hamster;      // DOES NOT COMPILE
short gerbil = 1 + (short)(mouse * hamster);  // DOES NOT COMPILE
```

El *Casting* es un operador unario y se aplica al operando inmediato derecho, luego ambos son promovidos a enteros.

## 5.1. COMPOUND ASSIGNMENT OPERATORS

Además del operador de asignación simple, `=`, también hay numerosos operadores de asignación compuesta. Sólo dos de los operadores compuestos enumerados en la tabla inicial son necesarios para el examen, `+=` y `-=`.

| Operador | Descripción |
|---|---|
| += | Agrega el valor de la derecha a la variable de la izquierda y asigna la suma a la variable |
| -= | Resta el valor de la derecha de la variable de la izquierda y asigna la diferencia a la variable |
| *= | Multiplica el valor de la derecha con la variable de la izquierda y asigna el producto a la variable |
| /= | Divide la variable de la izquierda por el valor de la derecha y asigna el cociente a la variable |

Por ejemplo:

```
int camel = 2, giraffe = 3;
camel = camel * giraffe;   // Simple operador de asignación
camel *= giraffe;          // Operador de asignación compuesta
```

Analicemos lo siguiente:

```
long goat = 10;
int sheep = 5;
sheep = sheep * goat;   // DOES NOT COMPILE
```

Hay una mejor manera de usar el operador de asignación compuesta aplicado al ejemplo anterior:

```
long goat = 10;
int sheep = 5;
sheep *= goat;   // Equivalente a: sheep = (int) (sheep * goat);
```

El operador compuesto primero realizará un *cast* a `y` a un `long`, aplicará la multiplicación de dos valores `long`, y luego el resultado le hará un *cast* a `int`. Es decir, realiza adicionalmente una operación de *cast*.

### 5.2. ASSIGNMENT OPERATOR RETURN VALUE

Una última cosa para saber sobre el operador de asignación es que el resultado de la asignación es una expresión en sí misma, igual al valor de la asignación. Por ejemplo:

```
long wolf = 5;
long coyote = (wolf=3);
System.out.println(wolf);   // muestra 3
System.out.println(coyote); // También, muestra 3
```

La clave aquí es que (`wolf=3`) hace dos cosas. Primero, establece el valor de la variable `wolf` en `3`. En segundo lugar, devuelve un valor de la asignación, que también es `3`.

## 6. Comparing Values

El último conjunto de operadores binarios gira en torno a la comparación de valores. Se pueden usar para verificar si dos valores son iguales, verificar si un valor numérico es menor o mayor que otro y realizar aritmética booleana.

### 6.1. EQUALITY OPERATORS

El operador *igual* (==) y *no es igual* (`!=`) como operadores relacionales, comparan dos operandos y retornan un valor booleano:

| Operador | Aplicar a primitivos | Aplicar a objetos |
|---|---|---|
| == | Devuelve verdadero si los dos valores representan el mismo valor | Devuelve verdadero si los dos valores hacen referencia al mismo objeto |
| != | Devuelve verdadero si los dos valores representan valores diferentes | Devuelve verdadero si los dos valores no hacen referencia al mismo objeto |

Los operadores de igualdad se utilizan en uno de tres escenarios:

1. Comparando dos tipos primitivos numéricos. Si los valores numéricos son de tipos de datos diferentes, los valores se promueven automáticamente como se describió anteriormente. Por ejemplo, `5 == 5.00` devuelve verdadero ya que el lado izquierdo se promueve a un `double`.
2. Comparando dos valores booleanos.
3. Comparando dos objetos, incluidos los valores `null` y `String`.

Por ejemplo:

```
boolean monkey = true == 3;        // DOES NOT COMPILE
boolean ape = false != "Grape";    // DOES NOT COMPILE
boolean gorilla = 10.2 == "Koko"; // DOES NOT COMPILE

boolean bear = false;
boolean polar = (bear = true); // Cuidado!, esto es una asignación: bear = true,
                               //  y dicho valor se asigna a polar
System.out.println(polar);     // Outputs true
```

Para la comparación de objetos, dos referencias son iguales si y solo si apuntan al mismo objeto, o ambos apuntan a nulo. Por ejemplo:

```
File monday = new File("schedule.txt");
File tuesday = new File("schedule.txt");
File wednesday = tuesday;
System.out.println(monday == tuesday);    // false
System.out.println(tuesday == wednesday); // true
```

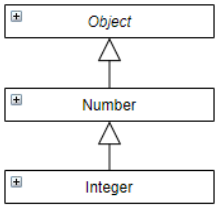
En algunos lenguajes, comparar nulo con cualquier otro valor siempre es falso, aunque este no es el caso en Java.

```
System.out.print(null == null);  // true
```

### 6.1.1. instanceof Operator

Es útil para determinar si un objeto arbitrario es miembro de una clase o interfaz en particular en tiempo de ejecución. Java admite polimorfismo. Por ahora, eso solo significa que algunos objetos se pueden pasar usando una variedad de referencias. Por ejemplo, ¿cuántos objetos se crean y usan en el siguiente fragmento de código?

```
Integer zooTime = Integer.valueOf(9);
Number num = zooTime;
Object obj = zooTime;
```




En este ejemplo, solo hay un objeto creado en la memoria, pero tres referencias diferentes a él porque `Integer` hereda tanto `Number` como `Object`. Esto significa que puede llamar a `instanceof` en cualquiera de estas referencias con tres tipos de datos diferentes y devolvería verdadero para cada uno de ellos:

```
System.out.println(zooTime instanceof Integer);  // true
```

```
System.out.println(zooTime instanceof Number);  // true
System.out.println(zooTime instanceof Object);  // true
System.out.println(num instanceof Integer);  // true
System.out.println(num instanceof Number);  // true
System.out.println(num instanceof Object);  // true
System.out.println(obj instanceof Integer);  // true
System.out.println(obj instanceof Number);  // true
System.out.println(obj instanceof Object);  // true
```

### 6.1.2. Invalid instanceof

¿Qué sucede si llama a `instanceof` en una variable nula? Para el examen, debe saber que llamar a `instanceof` en el literal nulo o una referencia nula siempre devuelve falso.

```
System.out.print(null instanceof Object);

Object noObjectHere = null;
System.out.print(noObjectHere instanceof String);
```

Los ejemplos anteriores se imprimen falso. Casi no importa cuál sea el lado derecho de la expresión. Decimos "casi" porque hay excepciones. El último ejemplo no se compila, ya que `null` se usa en el lado derecho del operador `instanceof`:

```
System.out.print(null instanceof null);  // DOES NOT COMPILE
```

## 6.2. LOGICAL OPERATORS

Los operadores lógicos, (`&`), (`|`) y (`^`), se pueden aplicar a los tipos de datos numéricos y booleanos. Cuando se aplican a tipos de datos booleanos, se los denomina operadores lógicos. Alternativamente, cuando se aplican a tipos de datos numéricos, se los denomina operadores bit a bit, ya que realizan comparaciones bit a bit de los bits que componen el número.

| Operador | Descripción |
|---|---|
| & | El AND lógico es verdadero solo si ambos valores son verdaderos. |
| \| | El OR inclusivo es verdadero si al menos uno de los valores es verdadero. |
| ^ | XOR exclusivo es verdadero solo si un valor es verdadero y el otro es falso. |

Sus reglas serían:

| | | operador | | |
|---|---|---|---|---|
| **x** | **y** | **&** | **\|** | **^** |
| true | true | true | true | false |
| true | false | false | true | true |
| false | true | false | true | true |
| false | false | false | false | false |

Algunos consejos para ayudar a recordar esta tabla:

1. Y solo es verdadero si ambos operandos son verdaderos.
2. El OR inclusivo solo es falso si ambos operandos son falsos.

3. La O exclusiva solo es verdadera si los operandos son diferentes.

Por ejemplo:

```
boolean eyesClosed = true;
boolean breathingSlowly = true;

boolean resting = eyesClosed | breathingSlowly;
boolean asleep = eyesClosed & breathingSlowly;
boolean awake = eyesClosed ^ breathingSlowly;
System.out.println(resting);  // true
System.out.println(asleep);   // true
System.out.println(awake);    // false
```

## 6.3. SHORT-CIRCUIT OPERATORS

Finalmente, presentamos los operadores condicionales, `&&` y `||`, que a menudo se conocen como operadores de cortocircuito.

| Operador | Descripción |
|---|---|
| && | El AND de cortocircuito es verdadero solo si ambos valores son verdaderos. Si el lado izquierdo es falso, no se evaluará el lado derecho. |
| \|\| | El OR de cortocircuito es verdadero si al menos uno de los valores es verdadero. Si el lado izquierdo es verdadero, entonces no se evaluará el lado derecho. |

Los operadores de cortocircuito son casi idénticos a los operadores lógicos, `&` y `|`, respectivamente, excepto que el lado derecho de la expresión nunca se evaluará si el resultado final puede determinarse por el lado izquierdo de la expresión. Por ejemplo:

```
int hour = 10;
boolean zooOpen = true || (hour < 4);
System.out.println(zooOpen);  // true
```

### 6.3.1. Avoiding a NullPointerException

Un ejemplo más común de dónde se utilizan operadores de cortocircuito es la comprobación de objetos nulos antes de realizar una operación. En el siguiente ejemplo, el cortocircuito previene el `NullPointerException`:

```
if(duck!=null & duck.getAge()<5) { // Podría lanzar una NullPointerException
   // Do something
}
```

Sin embargo, lo siguiente lanza una excepción:

```
if(duck!=null && duck.getAge()<5) { // Lanza una excepción si x es null
   // Do something
}
```

### 6.3.2. Checking for Unperformed Side Effects

Esto se conoce como un efecto secundario no realizado. Por ejemplo, ¿cuál es el resultado del siguiente código?

```
int rabbit = 6;
boolean bunny = (rabbit >= 6) || (++rabbit <= 7);
System.out.println(rabbit);
```

Debido a `rabbit >= 6` es `true`,

## 7. Making Decisions with the Ternary Operator

El operador condicional, `? :`, también conocido como el operador ternario, es el único operador que toma tres operandos y tiene la forma:

```
booleanExpression ? expression1 : expression2
```

El primer operando debe ser una expresión booleana, y el segundo y el tercero pueden ser cualquier expresión que devuelva un valor. Por ejemplo:

```
int owl = 5;
int food;
if(owl < 2) {
   food = 3;
} else {
   food = 4;
}
System.out.println(food);  // 4

//Se puede reecribir:
int owl = 5;
int food = owl < 2 ? 3 : 4;
System.out.println(food); // 4
```

No es necesario que las expresiones segunda y tercera en operadores ternarios tengan los mismos tipos de datos, aunque pueden entrar en juego cuando se combinan con el operador de asignación. Por ejemplo:

```
int stripes = 7;
System.out.print((stripes > 5) ? 21 : "Zebra");
int animal = (stripes < 9) ? 3 : "Horse";  // DOES NOT COMPILE
```

**Evaluación de expresiones ternarias**
A partir de Java 7, solo una de las expresiones del lado derecha del operador ternario se evaluará en tiempo de ejecución. De manera similar a los operadores de cortocircuito, si una de las dos expresiones de la derecha en un operador ternario tiene un efecto secundario, entonces no puede ser aplicado en tiempo de ejecución. por ejemplo:

```
int sheep = 1;
int zzz = 1;
int sleep = zzz<10 ? sheep++ : zzz++;
System.out.print(sheep+","+zzz);  // 2,1
```

Compare el ejemplo anterior con la siguiente modificación:

```
int sheep = 1;
int zzz = 1;
int sleep = sheep>=10 ? sheep++ : zzz++;
System.out.print(sheep+","+zzz);  // 1,2
```

## 8. Review Questions

### 8.1.

What data type (or types) will allow the following code snippet to compile? (Choose all that apply.)

```
byte apples = 5;
short oranges = 10;
_______ bananas = apples + oranges;
```

A. int
B. long
C. boolean
D. double
E. short
F. byte

### 8.2.

What is the output of the following code snippet?

```
3: boolean canine = true, wolf = true;
4: int teeth = 20;
5: canine = (teeth != 10) ^ (wolf=false);
6: System.out.println(canine+", "+teeth+", "+wolf);
```

A. true, 20, true
B. true, 20, false
C. false, 10, true
D. false, 20, false
E. The code will not compile because of line 5.
F. None of the above

### 8.3.

Which of the following operators are ranked in increasing or the same order of precedence? Assume the + operator is binary addition, not the unary form. (Choose all that apply.)

```
A. +, *, %, --
B. ++, (int), *
C. =, ==, !
D. (short), =, !, *
E. *, /, %, +, ==
F. !, ||, &
G. ^, +, =, +=
```

## 8.4.

What are the unique outputs of the following code snippet? (Choose all that apply.)

```
int a = 2, b = 4, c = 2;
System.out.println(a > 2 ? --c : b++);
System.out.println(b = (a!=c ? a : b++));
System.out.println(a > b ? b < c ? b : 2 : 1);
```

```
A. 1
B. 2
C. 3
D. 4
E. 5
F. 6
```

G. The code does not compile.

## 8.5.

How many lines of the following code contain compiler errors?

```
int note = 1 * 2 + (long)3;
short melody = (byte)(double)(note *= 2);
double song = melody;
float symphony = (float)((song == 1_000f) ? song * 2L : song);
```

```
A. 0
B. 1
C. 2
D. 3
E. 4
```