Capitulo 2
Certificación de Java 17

Operadores

1.	A,D,G. Con valores booleans podemos usar los siguientes operadores: • == • ! • Cast with (boolean) : Sí es posible y no marca error pero no tendría sentido porque el boolean es el único tipo que se pudiera castear a boolean, por lo que sería repetitivo.
2.	A,B,D. A. las operaciones de byte, short y char, deben tener como resultado un int (incluso en un long o un double porque el tamaño es mayor) pero no podría ser un short ni byte. If two values have different data types, Java will automatically promote one of the values to the larger of the two data type.
3.	B,C,D,F. Un long no cabe en int que es hearing, por lo que no compilaría como está, podríamos castear ear a int o algún tipo de dato más pequeño para que quepa.
4.	B boolean canine = true, wolf = true; int teeth = 20; // **true** se reasigna valor de wolf a **false** canine = (teeth != 10) ^ (wolf=false); // true 20 false System.out.println(canine+", "+teeth+", "+wolf); ^ Sólo cuando los valores son diferentes, arroja true. La respuesta sería: true 20 false

5.

A, C.

A. es correcta porque va de menor a mayor importancia. En un ejercicio real, primero se resolvería el --, después %, luego * y finalmente, +

B. ++ o – siempre es el más importante, por lo que el orden en que están es de mayor a menor, va decrementando.

C. sí va del menor al mayor.

D, E, F, G. no siguen el orden de menor a mayor.

6.

F.

static long addCandy dice qe regresará un long. El return si fuera int podría caber en long automáticamente pero el cast de int solo aplica a fruit, por lo que estamos sumando un int con un float que es vegetables, lo que daría como resultado un float, éste necesitaría un casteo para regresar como long, que es lo que solicita el método addCandy. Otra manera sería convertir todo a int para que de manera implícita hiciera el casteo a long.

7.

D.

```
int ph = 7, vis = 2;

//                  true    &   (true) = true

boolean clear = vis > 1 & (vis < 9 || ph < 2);

//                    (false) &&  ya no se evalúa la segunda parte

boolean safe = (vis > 2) && (ph++ > 1);

// ph=7 se queda con su valor porque no s ejecutó la segunda parte por el &&

// 7 es menor o igual a 6? = false

boolean tasty = 7 <= --ph;

System.out.println(clear + "-" + safe + "-" + tasty); // true false false
```

8.

A.

```
int pig = (short)4;  // un short cabe en un int por lo que no hay problema

pig = pig++;  //agarra el valor de pig y antes de incrementarlo, lo asigna a pig, quedando en 4.
```

	long goat = (int)2; // un int cabe en un long, por lo que no hay problema. goat -= 1.0; // goat = goat-1.0. Cuando uso las operaciones compuestas, automáticamente se hace el cast, en este caso a long, por lo que sería válido. System.out.print(pig + " -" + goat); // como hizo el cast a long, ya no tiene punto decimal y el resultado de goat es 1. // 4-1
9.	A,D,E. El incremento __++ no se aplica en la misma línea, sino hasta que sale de la línea. ++__ O --__, sí se aplican inmediatamente. El incremento __++ se pierde cuando se vuelve a asignar a la misma variable. porque gobierna la asignación. Ejemplo: int x = 10; x = x++; System.out.print(x); / /10 En el caso de b, se anula el incremento y queda en 5.
10.	G. El resultado de una operación entre bytes, short o chars, el resultado siempre debe debe guardarse en un int, por lo que **no compilaría** en esta línea: **short zebra = (byte) weight * (byte) height;**
11.	D. int sample1 = (2 * 4) % 3; **// primero la multiplicación y luego el % resultado: 2** int sample2 = 3 2 % 3; **// tienen el mismo orden de precedencia, se resuelve de izauierda a derecha resultado: 0** int sample3 = 5 * (1 % 2); **// resultado: 5** System.out.println(sample1 + ", " + sample2 + ", " + sample3); **//2, 0, 5**
	D.

12.	The **post-increment** operator increases a value and returns the original value, while the **pre-decrement** operator decreases a value and returns the new value.
13.	F. boolean sunny = true, raining = false, sunday = true; boolean goingToTheStore = sunny & raining ^ sunday; **// primero se evalua el & y después el ^resultado:** true boolean goingToTheZoo = sunday && !raining; **// true && true =** true **// !(true && true) =** false boolean stayingHome = !(goingToTheStore && goingToTheZoo); System.out.println(goingToTheStore + "-" + goingToTheZoo + "-" +stayingHome); // **true - true - false**
14.	B,E,G. A. The return value of an assignment operation expression can be void. Falso. void sólo lo usamos en métodos para decir que no existe un valor de retorno. B. The inequality operator (!=) can be used to compare objects. Verdadero, la usaríamos para preguntar si las variables de referencia apuntan a diferentes objetos. C. The equality operator (==) can be used to compare a boolean value with a numeric value. Falso, son de diferente tipo. D. During runtime, the & and | operators may cause only the left side of the expression to be evaluated. Falso, los operadores que podrían hacer eso serían && o ||. En el caso de & y | , están obligados a checar ambos lados. E. The return value of an assignment operation expression is the value of the newly assigned variable. Verdadero, en el mundo primitivo. F. In Java, 0 and false may be used interchangeably. Falso, depende del tipo de dato. G. The logical complement operator (!) cannot be used to flip numeric values. Verdadero. Únicamente se puede usar en booleans.

15.	D. el operador ternario ? : es el único que maneja 3 valores.
16.	B. int note = 1 * 2 + (long)3; //error de compilación, pues no podríamos meter un long en un int. short melody = (byte)(double)(note *= 2); // un byte sí cabe en un short, no hay error. double song = melody; // un short sí cabe en un double, no hay error. // false, por tanto regresa song, y lo castea a float, no hay error float symphony = (float)((song == 1_000f) ? song * 2L : song);
17.	C,F. int ticketsTaken = 1; int ticketsSold = 3; / / ticketsSold = 3 + (1 + 1) = 5 ticketsSold += 1 + ticketsTaken++; / / ticketsTaken = 2 / / ticketsTaken = 2 * 2 = 4 ticketsTaken *= 2; / / ticketsSold = 5 + (long)1 / / el resultado está en long pero como es operación compuesta, hay un cast implícito y se pasa a int. ticketsSold += (long)1; / / 6
18.	C. Lo único que podemos usar para modificar el orden de los operadores, son los paréntesis ( ).
19.	B,F.

start = (byte)(Byte.MAX_VALUE + 1);

indica que debe buscar el máximo valor de byte, es decir, 127, pero si se le suma 1 más, el resultado se saldrá del rango permitido de byte, por lo que ya no podría meterlo, esto provoca que vuelva a iniciar desde los números negativos, su valor = -128

20.

A, D, E.

A. Unary operators are always executed before any surrounding numeric binary or ternary operators. Verdadero. Los operadores unarios son aquellos que actúan sobre un solo operando (un solo valor), por ejemplo, ++, --, +(positivo), -(negativo), !

La afirmación es VERDADERA porque los operadores unarios siempre tienen prioridad de ejecución sobre los operadores binarios (que usan dos operandos) y ternarios (que usan tres operandos).

```
int a = 5;
int b = 10;
int resultado = -a + b;  // El operador unario - se ejecuta primero
```

1. Primero se ejecuta el -a (operador unario negativo)
2. Luego se ejecuta la suma (operador binario)

B. The - operator can be used to flip a boolean value. FALSO

C. The pre-increment operator (++) returns the value of the variable before the increment is applied. Falso, el pre-increment primero lo incremant y después lo regresa.

D. The post-decrement operator (--) returns the value of the variable before the decrement is applied. Verdadero.

E. The ! operator cannot be used on numeric values. Verdadero, sólo se usa con los booleans.

F. None of the above.

21.

E.

En la línea **int bird = ~myFavoriteNumber;**

El operador de complemento ~ trabaja a nivel de bits, invierte los 0 a 1 y viceversa.

La fórmula general para el operador de complemento a uno es: ~n= -(n+1)

En el ejemplo, el resultado sería -9


10:17