Java Building Blocks

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
+	Agrega 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
```

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
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)(x * 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*.

6. Review Questions

6.1.

6.2.

6.3.

6.4.

6.5.