Operators And Statements

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. Tres tipos de operadores están disponibles en Java: unario, binario y ternario.

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	`+, -, !`
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. Working with Binary Arithmetic Operators

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

2.1. 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 x = 2 * 5 + 3 * 4 - 8;
```

Se reduce a:

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

Si agregamos paréntesis:

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

…luego:

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

…luego:

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

…y finalmente:
```
int x = 2 * 24;
```

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 OCA, no es necesario conocer que se puede tomar el módulo de un número entero negativo o de punto flotante.

2.2. 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.

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`

3. Working with 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

3.1. 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 x = false;
System.out.println(x);  // false
x = !x;
System.out.println(x);  // true
```

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

```
double x = 1.21;58
System.out.println(x);  // 1.21
x = -x;
System.out.println(x);  // -1.21
x = -x;
System.out.println(x);  // 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 x = !5;  // DOES NOT COMPILE
boolean y = -true;  // DOES NOT COMPILE
boolean z = !0;  // DOES NOT COMPILE
```

3.2. 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 counter = 0;
System.out.println(counter);  // Outputs 0
System.out.println(++counter);  // Outputs 1
System.out.println(counter); // Outputs 1
```

```
System.out.println(counter--);  // Outputs 1
System.out.println(counter);  // Outputs 0
```

Ejemplos con operadores en una misma línea:

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

El resultado sería:
```
x is 2
y is 7
```

4. Using Additional Binary Operators

Esto incluye operadores que realizan asignaciones, aquellos que comparan valores aritméticos y devuelven resultados booleanos, y aquellos que comparan valores booleanos y de objetos y arrojan resultados booleanos.

4.1. Assignment Operators

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 x = 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. Por ejemplo:

```
int x = 1.0;         // NO COMPILA
short y = 1921222;   // NO COMPILA, Fuera del rango de short
int z = 9f;          // NO COMPILA, se indica que lo trate como punto flotante
long t = 192301398193810323;  // NO COMPILA, intepreta literal como int y
                              // estaría fuera de rango de los int.
```

4.1.1. Casting Primitive Values

Podemos arreglar los ejemplos en la sección anterior al convertir los resultados a un tipo de datos más pequeño:

```
int x = (int)1.0;
short y = (short)1921222;  // Guardado como 20678 int z = (int)9l;
long t = 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 x = 10;
short y = 3;
short z = x * y;  // 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 x = 10;
short y = 3;
short z = (short)(x * y);
```

4.2. 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 `-=`. Por ejemplo:

```
int x = 2, z = 3;
x = x * z;  // Simple operador de asignación
x *= z;     // Operador de asignación compuesta
```

Analicemos lo siguiente:

```
long x = 10;
int y = 5;
y = y * x;  // DOES NOT COMPILE
```

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

```
long x = 10;
int y = 5;
y *= x;
```

El operador compuesto primero realizará un *cast* a `x` 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*.

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

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

4.3. Relational Operators

Ahora pasamos a los operadores relacionales, que comparan dos expresiones y devuelven un valor booleano:

Operador	Descripción
`<`	Estrictamente menor que
`<=`	Menor o igual que
`>`	Estrictamente mayor que
`>=`	Mayor o igual que
`a instanceof b`	true si la referencia a la que apunta es una instancia de una clase, subclase o clase que implementa una interfaz particular, como se nombra en b. No es parte del examen OCA.

Por ejemplo:
```
int x = 10, y = 20, z = 10;
System.out.println(x < y);  // Outputs true
System.out.println(x <= y);  // Outputs true
System.out.println(x >= z);  // Outputs true
System.out.println(x > z);  // Outputs false
```

El quinto operador relacional de la tabla anterior se aplica a referencias de objeto y clases o interfaces.

4.4. 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		
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.

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

```
boolean x = true || (y < 4);
```

En el siguiente ejemplo, el cortocircuito previene el `NullPointerException`:
```
if(x != null && x.getValue() < 5) {
  // Do something
}
```

Sin embargo, lo siguiente lanza una excepción:
```
if(x != null & x.getValue() < 5) { // Lanza una excepción si x es null
  // Do something
}
```

Finalmente:
```
int x = 6;
boolean y = (x >= 6) || (++x <= 7);
System.out.println(x);
```

Nos resulta 6.

4.5. Equality Operators

Comencemos con lo básico, el operador igual (==) y no es igual (`!=`). 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 nulos y `String`.

Por ejemplo:

```
boolean x = true == 3;  // DOES NOT COMPILE
boolean y = false != "Giraffe";  // DOES NOT COMPILE
boolean z = 3 == "Kangaroo";  // DOES NOT COMPILE

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

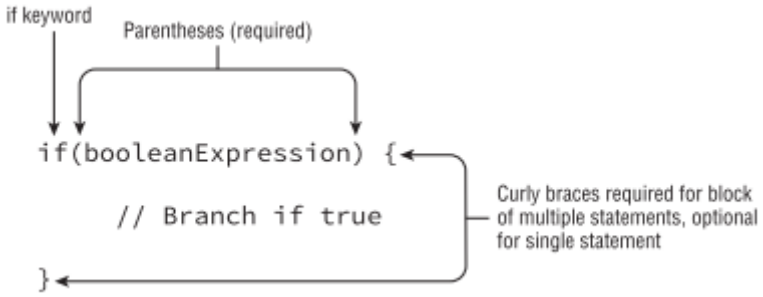
Para la comparación de objetos, dos referencias son iguales si y solo si apuntan al mismo objeto, o ambos apuntan a nulo. Por ejemplo:

```
File x = new File("myFile.txt");
File y = new File("myFile.txt");
File z = x;
System.out.println(x == y);  // Outputs false
System.out.println(x == z);  // Outputs true
```

5. Understanding Java Statements

Las sentencias de control de flujo dividen el flujo de ejecución mediante la toma de decisiones, bucles y la bifurcación, lo que permite a la aplicación ejecutar determinados segmentos de código de forma selectiva.

5.1. The if-then Statement



Por ejemplo:

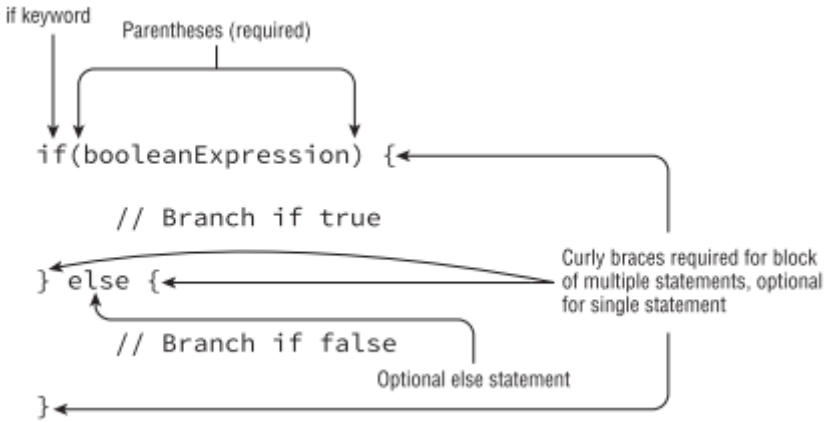
```
if(hourOfDay < 11)
   System.out.println("Good Morning");

if(hourOfDay < 11) {
   System.out.println("Good Morning");
   morningGreetingCount++;
}
```

Sangría y llaves
Por ejemplo:
```
if(hourOfDay < 11)
  System.out.println("Good Morning");
  morningGreetingCount++;
```

Según la sangría, es posible que piense que la variable `morningGreetingCount` solo se incrementará si `hourOfDay` es menor que 11, pero eso no es lo que hace este código. Ejecutará la instrucción de impresión solo si se cumple la condición, pero siempre ejecutará la operación de incremento.

5.2. The if-then-else Statement



Por ejemplo, si partimos de lo siguiente:

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else {
  System.out.println("Good Afternoon");
}
```

El operador `else` toma una instrucción o bloque de instrucción, de la misma manera que la instrucción `if`. De esta manera, podemos agregar declaraciones `if-then` adicionales a un bloque `else` para llegar a un ejemplo más refinado:

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else {
  System.out.println("Good Evening");
}
```

Si ahora reordenamos el código anterior del siguiente modo:

```
if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else if(hourOfDay < 11) {
  System.out.println("Good Morning");  // UNREACHABLE CODE
} else {
  System.out.println("Good Evening");
}
```

Verificando si la sentencia `if` evalúa una expresión booleana
Otro lugar común en el que el examen puede intentar desviarlo es proporcionar un código donde la expresión booleana dentro de la declaración `if-then` no es en realidad una expresión booleana. Por ejemplo:

```
int x = 1;
if(x) {  // DOES NOT COMPILE
  ...
}

int x = 1;
if(x = 5) {  // DOES NOT COMPILE
  ...
}
```

5.2.1. 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 y = 10;
final int x;
if(y > 5) {
  x = 2 * y;
} else {
  x = 3 * y;
}

//Se puede reecribir:
int y = 10;
x = (y > 5) ? 2 * y : 3 * y;
```

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:

```
System.out.println((y > 5) ? 21 : "Zebra");
int animal = (y < 91) ? 9 : "Horse";  // DOES NOT COMPILE
```

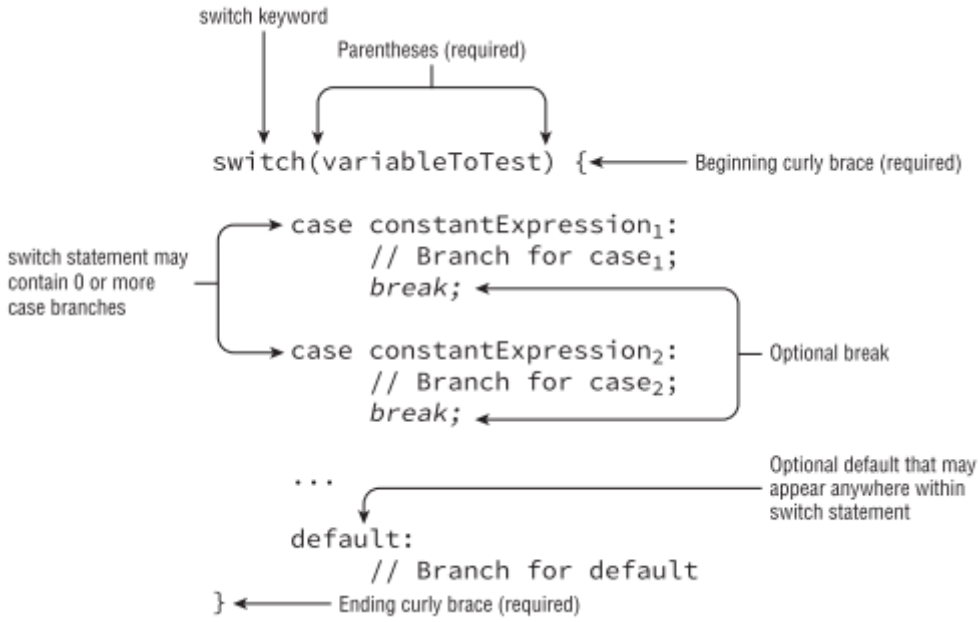
Evaluación de expresiones ternarias
A partir de Java 7, solo una de las expresiones del lado derecha del operador ternario se evaluará en tiempo de ejecución. De manera similar a los operadores de cortocircuito, si una de las dos expresiones de la derecha en un operador ternario tiene un efecto secundario,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

```
int y = 1;
int z = 1;
final int x = y<10 ? y++ : z++;
System.out.println(y+","+z); // Outputs 2,1

int y = 1;
int z = 1;
final int x = y>=10 ? y++ : z++;
System.out.println(y+","+z); // Outputs 1,2
```

5.3. The switch Statement

5.3.1. Supported Data Types



Los tipos de datos admitidos por las sentencias `switch` incluyen lo siguiente:

1. `int and Integer`
2. `byte and Byte`
3. `short and Short`
4. `char and Character`
5. `int and Integer`
6. `String`
7. Valores enumerados.

5.3.2. Compile-time Constant Values

Los valores en cada declaración de `case` deben ser valores constantes de tiempo de compilación del mismo tipo de datos que el valor de conmutación. Esto significa que puede usar solo literales, constantes de enum o variables constantes finales del mismo tipo de datos. Por constante final, queremos decir que la variable debe

estar marcada con el modificador final e inicializada con un valor literal en la misma expresión en la que se declara.

En el siguiente ejemplo, preste atención a la sentencia `break`:

```
int dayOfWeek = 5;
switch(dayOfWeek) {
  default:
    System.out.println("Weekday");
    break;
  case 0:
    System.out.println("Sunday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
}
```

El resultado es:
```
Weekday
```

Considere la siguiente variación:

```
int dayOfWeek = 5;
switch(dayOfWeek) {
  case 0:
    System.out.println("Sunday");
  default:
    System.out.println("Weekday");
  case 6:
    System.out.println("Saturday");
    break;
}
```

Si `dayOfWeek` es 5, la salida es:
```
Weekday
Saturday
```

Si `dayOfWeek` es 6, la salida es:
```
Saturday
```

Finalmente, si `dayOfWeek` es 0, la salida es:
```
Sunday
Weekday
Saturday
```

Por ejemplo, dada la siguiente sentencia `switch`, observe cuáles declaraciones `case` compilarán y cuáles no:

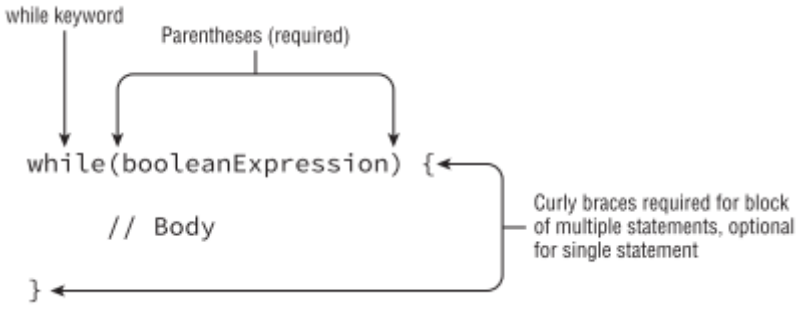
```
private int getSortOrder(String firstName, final String lastName) {
  String middleName = "Patricia";
  final String suffix = "JR";
  int id = 0;
  switch(firstName) {
    case "Test":
      return 52;
    case middleName:  // DOES NOT COMPILE
      id = 5;
      break;
    case suffix:
```

```
      id = 0;
      break;
    case lastName:  // DOES NOT COMPILE
      id = 8;
      break;
    case 5:  // DOES NOT COMPILE
      id = 7;
      break;
    case 'J':  // DOES NOT COMPILE
      id = 10;
      break;
    case java.time.DayOfWeek.SUNDAY:  // DOES NOT COMPILE
      id=15;
      break;
  }
  return id;
}
```

Del código anterior, tenemos:

1. Para el `case` (2): `middleName` no es una variable `final`.
2. Para el `case` (3): `suffix` es una variable `final` constante.
3. Para el `case` (4): A pesar de que `lastName` es `final`, no es una constante al ser pasada al método.
4. El resto de `case`: No compilan porque no corresponden a un tipo `String`.

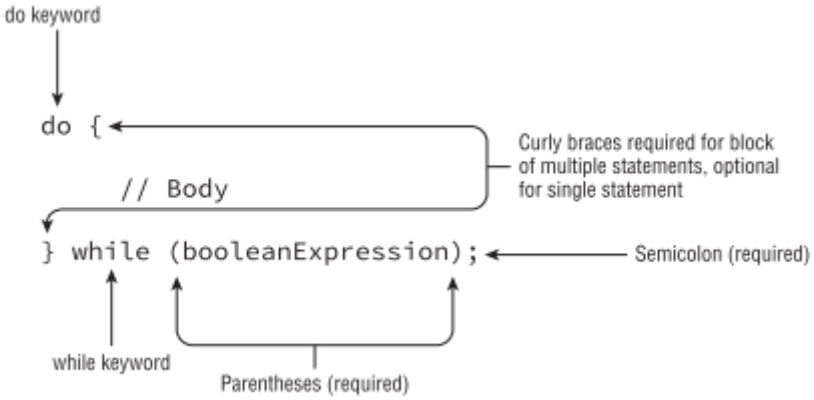
5.4. The while Statement



Por ejemplo:

```
int roomInBelly = 5;
public void eatCheese(int bitesOfCheese) {
  while (bitesOfCheese > 0 && roomInBelly > 0) {
    bitesOfCheese--;
    roomInBelly--;
  }
  System.out.println(bitesOfCheese+" pieces of cheese left");
}
```

5.5. The do-while Statement



La principal diferencia entre la estructura sintáctica de un bucle `do-while` y un bucle `while` es que un bucle `do-while` ejecuta a propósito el enunciado o bloque de enunciados antes de la expresión condicional, para reforzar que el enunciado se ejecutará antes de que la expresión booleana sea evaluada. Por ejemplo:

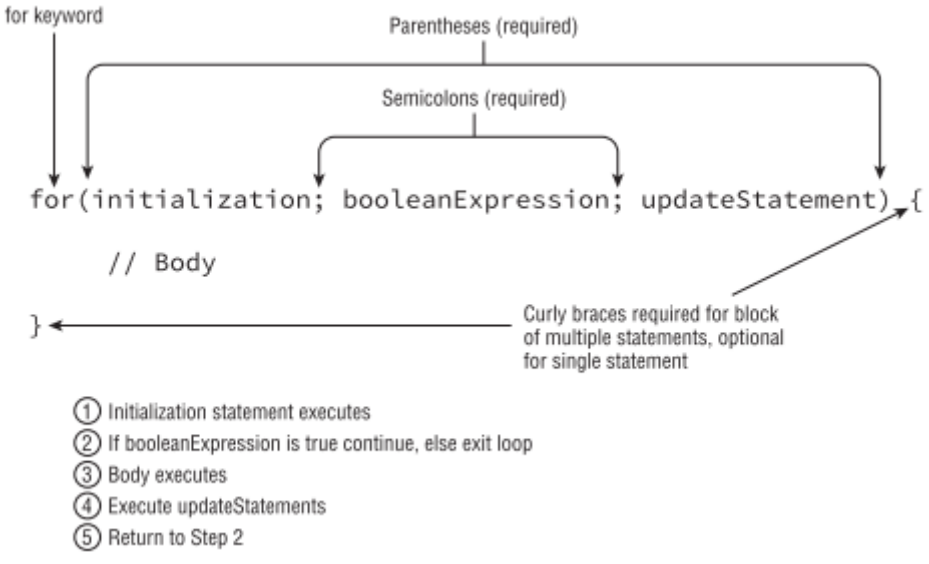
```
int x = 0;
do {
  x++;
} while(false);
System.out.println(x);  // Outputs 1
```

Cuándo utilizar `while` vs. `do-while` Loops
Java recomienda utilizar un ciclo `while` cuando un ciclo podría no ejecutarse y un ciclo `do-while` cuando el ciclo se ejecutará al menos una vez. Pero la determinación de si se debe usar un bucle `while` o un bucle `do-while` en la práctica a veces es más por una preferencia personal y legibilidad del código.

5.6. The for Statement

Desde Java 5.0, hay dos tipos de declaraciones `for`. El primero se conoce como el bucle básico `for`, y el segundo se llama a menudo `for` mejorado (`enhanced for`). Para mayor claridad, nos referiremos al bucle `for` mejorado como el enunciado `for-each`.

5.6.1. The Basic for Statement



Por ejemplo:

```
for(int i = 0; i < 10; i++) {
  System.out.print(i + " ");
}
```

Revisemos algunos típicos usos de `for`.

1. Creando un bucle infinito:
   ```
   for( ; ; ) {
     System.out.println("Hello World");
   }
   ```

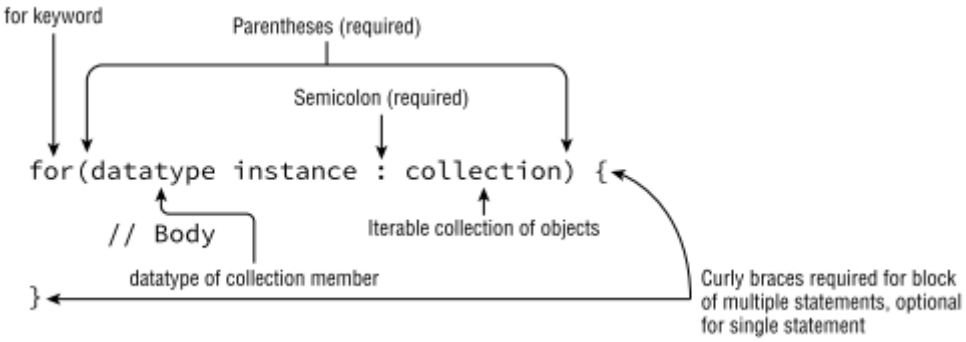
2. Agregar varios términos a la declaración `for`:
   ```
   int x = 0;
   for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
     System.out.print(y + " ");
   }
   System.out.print(x);
   ```

3. Volver a declarar una variable en el bloque de inicialización:
   ```
   int x = 0;
   for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {   // DOES NOT COMPILE
     System.out.print(x + " ");
   }
   ```

4. Usar tipos de datos incompatibles en el bloque de inicialización:
   ```
   for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) {   // DOES NOT COMPILE
     System.out.print(x + " ");
   }
   ```

5. Uso de variables de bucle fuera del bucle:
   ```
   for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
     System.out.print(y + " ");
   }
   System.out.print(x);  // DOES NOT COMPILE
   ```

5.6.2. The for-each Statement



Nota: Para el examen OCA, los únicos miembros del *framework* de Colecciones que debe conocer son `List` y `ArrayList`.

Cuando vea un bucle `for-each` en el examen, asegúrese de que el lado derecho sea una matriz o un objeto Iterable y que el lado izquierdo tenga un tipo que coincida. Por ejemplo:

```
// (1)
```

```
final String[] names = new String[3];
names[0] = "Lisa";
names[1] = "Kevin";
names[2] = "Roger";
for(String name : names) {
  System.out.print(name + ", ");
}
//compila e imprime:
Lisa, Kevin, Roger,

// (2)
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
for(String value : values) {
  System.out.print(value + ", ");
}
//compila e imprime:
Lisa, Kevin, Roger,

// (3)
String names = "Lisa";
for(String name : names) {   // DOES NOT COMPILE
  System.out.print(name + " ");
}

// (4)
String[] names = new String[3];
for(int name : names) {  // DOES NOT COMPILE
  System.out.print(name + " ");
}
```

Comparando bucles `for` y `for-each`

Cuando `for-each` se introdujo en Java 5, se agregó como una mejora en tiempo de compilación. Esto significa que Java realmente convierte el bucle `for-each` en un bucle `for` durante la compilación. Por ejemplo, a continuación, se muestra un bucle `for-each` y su equivalente `for`:

```
// (1)
for(String name : names) {
  System.out.print(name + ", ");
}
for(int i=0; i < names.length; i++) {
  String name = names[i];
  System.out.print(name + ", ");
}

// (2)
for(int value : values) {
  System.out.print(value + ", ");
}
for(java.util.Iterator<Integer> i = values.iterator(); i.hasNext(); ) {
  int value = i.next();
  System.out.print(value + ", ");
}
```

6. Understanding Advanced Flow Control

6.1. Nested Loops

En primer lugar, los bucles pueden contener otros bucles. Por ejemplo:

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
for(int[] mySimpleArray : myComplexArray) {
  for(int i=0; i<mySimpleArray.length; i++) {
    System.out.print(mySimpleArray[i]+"\t");
  }
  System.out.println();
}
```

Lo cual imprime:

```
5       2       1       3
3       9       8       9
5       7       12       7
```

Y también:

```
int x = 20;
while(x>0) {
  do {
    x -= 2
  } while (x>5);
  x--;
  System.out.print(x+"\t");
}
```

Imprime:

```
3       0
```

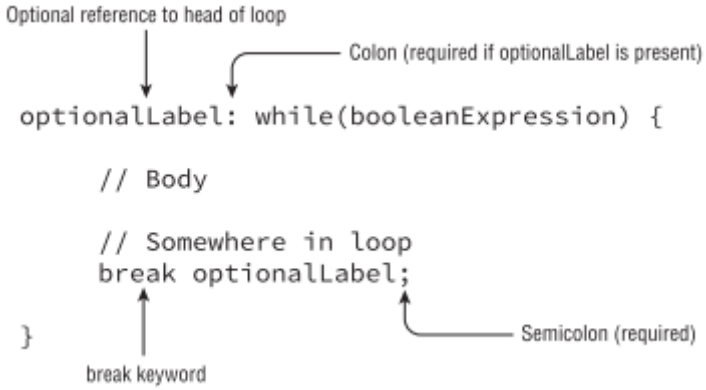
6.2. Adding Optional Labels

Una etiqueta es un puntero opcional al encabezado de una declaración que permite que el flujo de la aplicación salte o se salga de ella. Es una sola palabra que está precedida por dos puntos (:). Por ejemplo:

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
OUTER_LOOP:  for(int[] mySimpleArray : myComplexArray) {
  INNER_LOOP:  for(int i=0; i<mySimpleArray.length; i++) {
    System.out.print(mySimpleArray[i]+"\t");
  }
  System.out.println();
}
```

Para formatear, las etiquetas siguen las mismas reglas de los identificadores. Por legibilidad, se expresan comúnmente en mayúsculas, con guiones bajos entre las palabras, para distinguirlas de las variables regulares.

6.3. The break Statement



Como vio al trabajar con declaraciones de `switch`, una instrucción `break` transfiere el control del flujo de a la instrucción que lo rodea. Lo mismo ocurre con las sentencias `break` que aparecen dentro de `while`, `do-while` y `for`, que terminarán el bucle anticipadamente.

En el siguiente ejemplo, buscamos la primera posición de índice de matriz (x, y) de un número dentro de una matriz bidimensional no ordenada:

```
public class SearchSample {
  public static void main(String[] args) {
    int[][] list = {{1,13,5},{1,2,5},{2,7,2}};
    int searchValue = 2;
    int positionX = -1;
    int positionY = -1;
    PARENT_LOOP: for(int i=0; i<list.length; i++) {
      for(int j=0; j<list[i].length; j++) {
        if(list[i][j]==searchValue) {
          positionX = i;
          positionY = j;
          break PARENT_LOOP;
        }
      }
    }
    if(positionX==-1 || positionY==-1) {
      System.out.println("Value "+searchValue+" not found");
    } else {
      System.out.println("Value "+searchValue+" found at: " +
        "("+positionX+","+positionY+")");
    }
  }
}
```

Cuando se ejecuta, este código dará como resultado:

```
Value 2 found at: (1,1)
```

Ahora, imagine lo que sucedería si reemplazamos el cuerpo del bucle interno con lo siguiente:

```
if(list[i][j]==searchValue) {
  positionX = i;
  positionY = j;
  break;
}
```

En lugar de salir cuando se encuentra el primer valor coincidente, el programa ahora solo saldrá del bucle interno cuando se cumpla la condición. En otras palabras, la estructura ahora encontrará el primer valor coincidente del último bucle interno para contener el valor, lo que da como resultado el siguiente resultado:

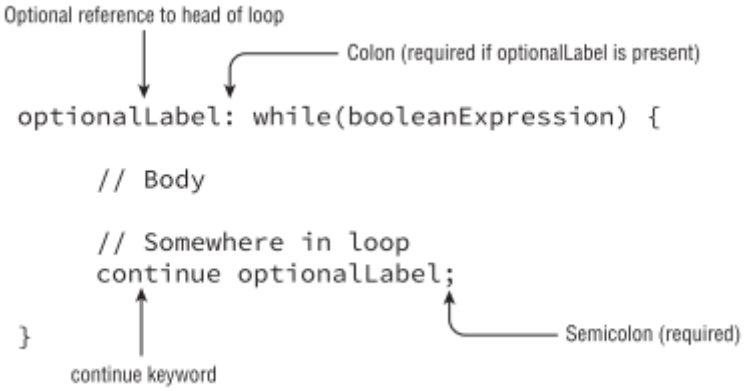
```
Value 2 found at: (2,0)
```

Finalmente, ¿qué pasa si eliminamos el break por completo?

```
if(list[i][j]==searchValue) {
  positionX = i;
  positionY = j;
}
```

La salida se verá así:
```
Value 2 found at: (2,2)
```

6.4. The continue Statement



La sintaxis de la instrucción `continue` se asemeja a la declaración `break`. Mientras que la sentencia `break` transfiere el control a la instrucción que lo rodea, la instrucción `continue` transfiere el control a la expresión booleana que determina si el ciclo debe continuar. En otras palabras, finaliza la iteración actual del bucle. De forma similar que la instrucción `break`, la instrucción `continue` se aplica al bucle interno más cercano en ejecución y, utilizando declaraciones de etiqueta opcionales, se puede anular este comportamiento. Por ejemplo:

```
public class SwitchSample {
  public static void main(String[] args) {
    FIRST_CHAR_LOOP: for (int a = 1; a <= 4; a++) {
      for (char x = 'a'; x <= 'c'; x++) {
        if (a == 2 || x == 'b')
          continue FIRST_CHAR_LOOP;
        System.out.print(" " + a + x);
      }
    }
  }
}
```

Imprime:
```
1a 3a 4a
```

Ahora imagine que eliminamos la etiqueta `FIRST_CHAR_LOOP` en la instrucción `continue` para que el control se devuelva al bucle interno en lugar de al externo. El resultado será:

```
1a 1c 3a 3c 4a 4c
```

Finalmente, si eliminamos por completo la instrucción `continue` y la declaración `if-then` asociada, llegamos a una estructura que genera todos los valores:

```
1a 1b 1c 2a 2b 2c 3a 3b 3c 4a 4b 4c
```

La siguiente tabla lo ayudará a recordar cuándo se permiten declaraciones de etiquetas, `break` y `continue` en Java:

Sentencia	Permite etiquetas	Permite break	Permite continue
`if`	Si *	No	No
`while`	Si	Si	Si
`do-while`	Si	Si	Si
`for`	Si	Si	Si
`switch`	Si	Si	No

* Las etiquetas están permitidas para cualquier instrucción de bloque, incluidas aquellas que están precedidas por una instrucción `if-then`.

7. Review Questions

7.1.

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

```
byte x = 5;
byte y = 10;
_____ z = x + y;
```

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

7.2.

What is the output of the following application?

```
1: public class CompareValues {
2:   public static void main(String[] args) {
3:     int x = 0;
4:     while(x++ < 10) {}
5:     String message = x > 10 ? "Greater than" : false;
6:     System.out.println(message+","+x);
7:   }
8: }
```

A. `Greater than,10`
B. `false,10`
C. `Greater than,11`
D. `false,11`
E. The code will not compile because of line 4.
F. The code will not compile because of line 5.

7.3.

What is the output of the following code snippet?

```
3: int x = 4;
```

```
4: long y = x * 4 - x++;
5: if(y<10) System.out.println("Too Low");
6: else System.out.println("Just right");
7: else System.out.println("Too High");
```

A. `Too Low`
B. `Just Right`
C. `Too High`
D. Compiles but throws a `NullPointerException`.
E. The code will not compile because of line 6.
F. The code will not compile because of line 7.

7.4.

What is the output of the following code snippet?

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

A. `true, 10, true`
B. `true, 20, false`
C. `false, 20, true`
D. `false, 20, false`
E. `false, 20, true`
F. The code will not compile because of line 5.

7.5.

What is the output of the following code snippet?

```
3: boolean keepGoing = true;
4: int result = 15, i = 10;
5: do {
6:   i--;
7:   if(i==8) keepGoing = false;
8:   result -= 2;
9: } while(keepGoing);
10: System.out.println(result);
```

A. `7`
B. `9`
C. `10`
D. `11`
E. `15`
F. The code will not compile because of line 8.

7.6.

What is the output of the following code snippet?

```
3: int count = 0;
4: ROW_LOOP: for(int row = 1; row <=3; row++)
5:   for(int col = 1; col <=2 ; col++) {
6:     if(row * col % 2 == 0) continue ROW_LOOP;
7:     count++;
8:   }
9: System.out.println(count);
```

A. `1`
B. `2`
C. `3`
D. `4`
E. `6`
F. The code will not compile because of line 6.

7.7.

What is the result of the following code snippet?

```
3: final char a = 'A', d = 'D';
4: char grade = 'B';
5: switch(grade) {
6:   case a:
7:   case 'B': System.out.print("great");
8:   case 'C': System.out.print("good"); break;
9:   case d:
10:  case 'F': System.out.print("not good");
11: }
```

A. `great`
B. `greatgood`
C. The code will not compile because of line 3.
D. The code will not compile because of line 6.
E. The code will not compile because of lines 6 and 9.