Operators

1. Creating Decision-Making Statements

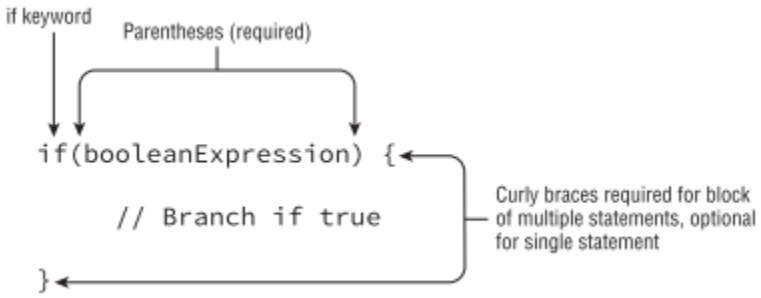
1.1. STATEMENTS AND BLOCKS

Una declaración de Java (*Java Statement*) es una unidad completa de ejecución en Java, terminada con un punto y coma (;).

Estas declaraciones se pueden aplicar a expresiones individuales, así como a un bloque de código Java. Un bloque de código en Java es un grupo de cero o más declaraciones entre llaves equilibradas ({}) y se puede usar en cualquier lugar donde se permita una sola declaración. Por ejemplo, los siguientes dos fragmentos son equivalentes:

```
// Single statement
patrons++;

// Statement inside a block
{
   patrons++;
}
```

1.2. THE IF STATEMENT



Por ejemplo:

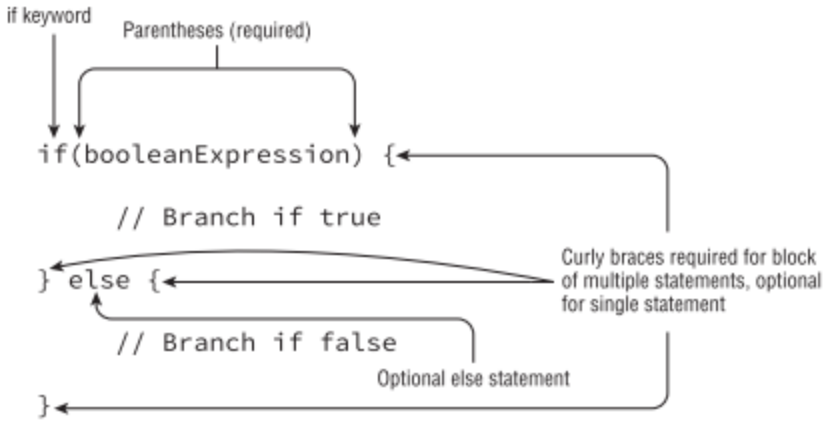
```
if(hourOfDay < 11)
   System.out.println("Good Morning");

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

WATCH INDENTATION AND BRACES
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.

1.3. THE 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
```

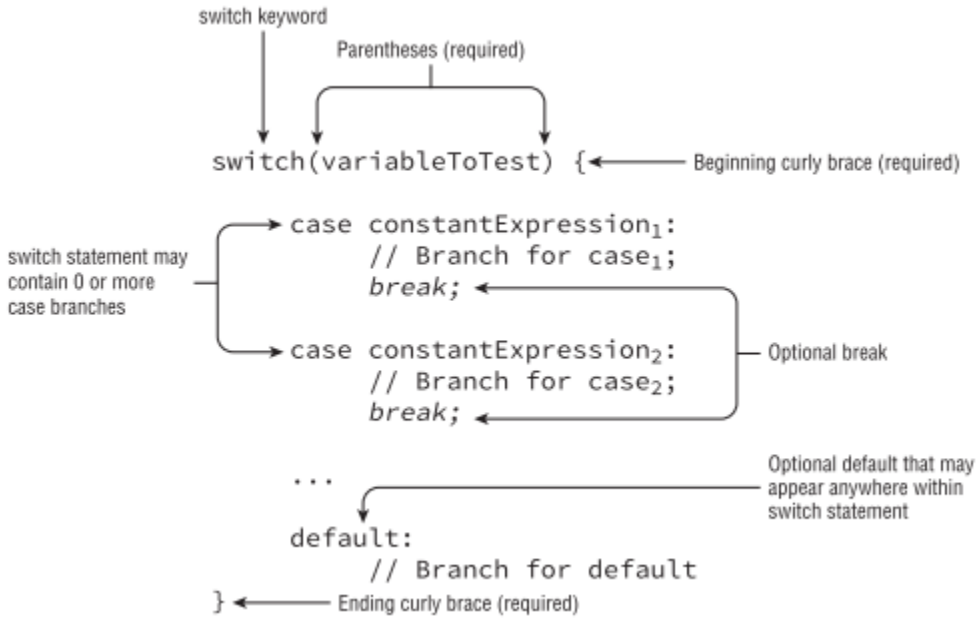
```
} else {
  System.out.println("Good Evening");
}
```

VERIFYING THAT THE IF STATEMENT EVALUATES TO A BOOLEAN EXPRESSION

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 hourOfDay  = 1;
if(hourOfDay ) {  // DOES NOT COMPILE
  ...
}

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

2. THE SWITCH STATEMENT



2.1.1. Proper Switch Syntax

Vea si puede averiguar por qué cada una de las siguientes declaraciones de cambio no se compila:

```
int month = 5;

switch month {  // NO COMPILA:
   case 1: System.out.print("January");
}

switch (month)  // NO COMPILA:
   case 1: System.out.print("January");
```

```
switch (month) {
   case 1: 2: System.out.print("January"); // NO COMPILA:
}

switch (month) {
   case 1 || 2: System.out.print("January"); // NO COMPILA:
}
```

2.1.2. Switch 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. `String`
6. Valores enumerados.
7. `Var` (Si el tipo se resuelve a uno de los tipos anteriores)

2.1.3. Switch Control Flow

Puede notar que el bloque predeterminado no está al final de la declaración de cambio. No es necesario que el caso o la declaración predeterminada estén en un orden particular, a menos que tenga rutas que lleguen a varias secciones del bloque de interruptores en una sola ejecución.

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
```

2.1.4. Acceptable Case Values

Los valores en cada declaración de `case` **deben ser valores constantes de tiempo de compilación** del mismo tipo de datos que el valor del `switch`. 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.

Por ejemplo:

```
final int getCookies() { return 4; }
void feedAnimals() {
   final int bananas = 1;
   int apples = 2;
   int numberOfAnimals = 3;
   final int cookies = getCookies();
   switch (numberOfAnimals) {
      case bananas:
      case apples:        // DOES NOT COMPILES
      case getCookies():  // DOES NOT COMPILE
      case cookies :      // DOES NOT COMPILE
      case 3 * 5 :
   }
}
```

La última declaración de caso, con valor 3 * 5, se compila, ya que se permiten expresiones como valores de `case`, siempre que el valor se pueda resolver en tiempo de compilación. También deben poder caber en el tipo de datos del `switch` sin una conversión explícita.

Un ejemplo más complejo, 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 (2)
```

```
      id = 5;
      break;
    case suffix:      // (3)
      id = 0;
      break;
    case lastName:  // DOES NOT COMPILE (4)
      id = 8;
      break;
    case 5:  // DOES NOT COMPILE7
      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`.

2.1.5. Numeric Promotion and Casting

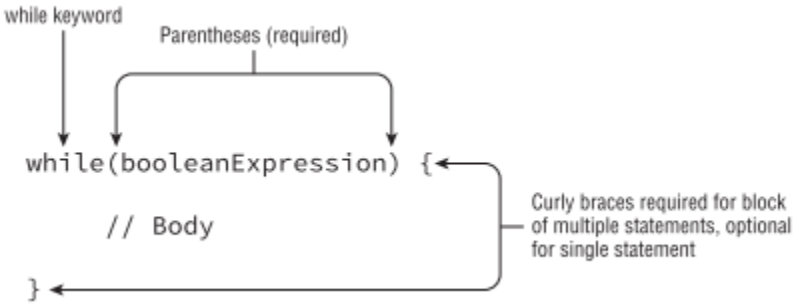
Las declaraciones `switch` admiten la promoción numérica que no requiere una conversión explícita. Por ejemplo:

```
short size = 4;
final int small = 15;
final int big = 1_000_000;
switch(size) {
   case small:
   case 1+2 :
   case big:  // DOES NOT COMPILE
}
```

Del ejemplo, `small`, `1+2` pueden ser convertidos a `short` mediante `cast` sin problema. Sin embargo, `big` es muy grande para caber en short, y por ello no compila.

3. Writing while Loops

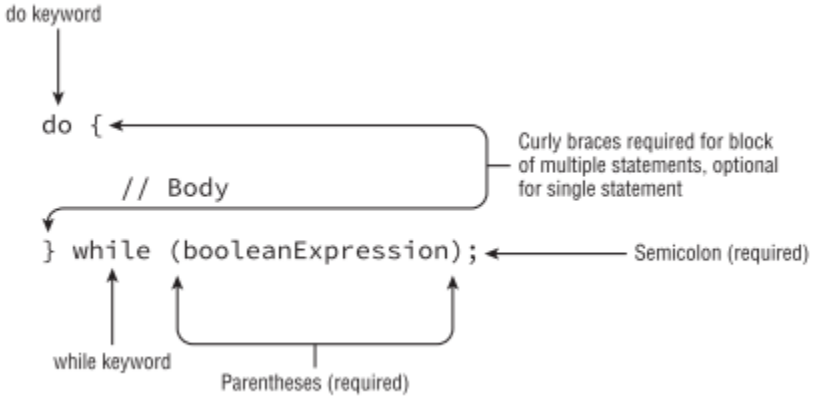
3.1. THE WHILE STATEMENT



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");
}
```

3.2. 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 lizard = 0;
do {
   lizard++;
} while(false);
System.out.println(lizard);  // 1
```

3.3. COMPARING WHILE AND DO/WHILE LOOPS

En la práctica, puede ser difícil determinar cuándo debe usar un bucle `while` y cuándo debe usar un bucle `do/while`. La respuesta corta es que en realidad no importa. Cualquier bucle `while` se puede convertir en un bucle `do/while`, y viceversa. Por ejemplo, compare este ciclo `while`:

```
while(llama > 10) {
   System.out.println("Llama!");
   llama--;
}
```

y este bucle `do/while`:

```
if(llama > 10) {
   do {
      System.out.println("Llama!");
      llama--;
   } while(llama > 10);
}
```

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.

3.4. INFINITE LOOPS

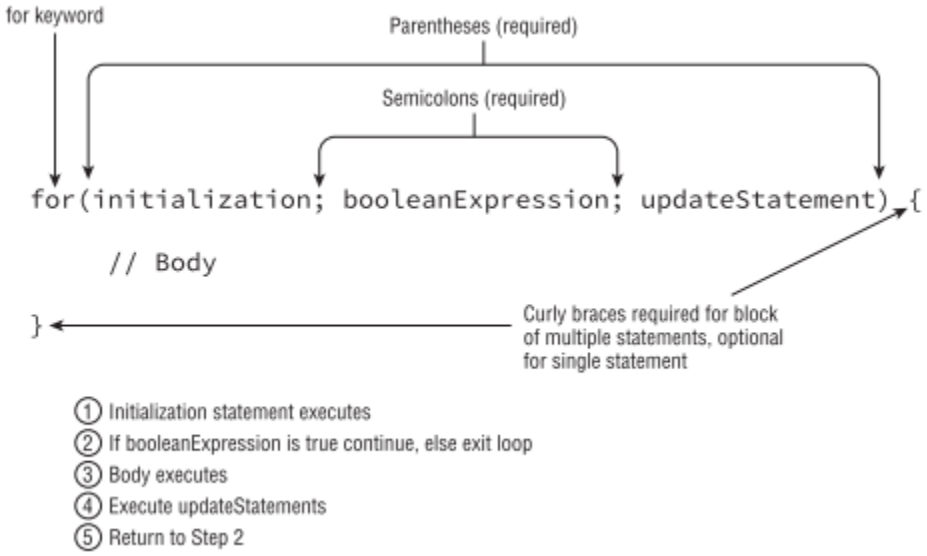
Lo más importante que debes tener en cuenta cuando utilices cualquier estructura de control de repetición es asegurarte de que siempre terminen. Veamos un ejemplo:

```
int pen = 2;
int pigs = 5;
while(pen < 10)
   pigs++;
```

La variable `pen` nunca se modifica, por lo que la expresión (`pen <10`) siempre se evaluará como verdadera. El resultado es que el ciclo nunca terminará, creando lo que comúnmente se conoce como un ciclo infinito. Un bucle infinito es un bucle cuya condición de terminación nunca se alcanza durante el tiempo de ejecución.

4. Constructing for Loops

4.1. THE FOR LOOP



Por ejemplo:

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

4.1.1. Printing Elements in Reverse

El objetivo entonces es imprimir `4 3 2 1 0`.

¿Cómo lo harías tú? Comenzando con Java 10, ahora puede ver `var` usado en un bucle `for`, así que usémoslo para este ejemplo.

```
for (var counter = 5; counter > 0; counter--) {
   System.out.print(counter + " ");
}
//La salida sería:
5 4 3 2 1
```

Si queremos imprimir el mismo `0` a `4` en nuestro primer ejemplo, necesitamos actualizar la condición de terminación, así:

```
for (var counter = 4; counter >= 0; counter--) {
   System.out.print(counter + " ");
}
```

4.1.2. Working with for Loops

Debe familiarizarse con los siguientes cinco ejemplos; Es probable que se vean variaciones de estos en el examen.

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
```

4.1.3. Modifying Loop Variables

Eche un vistazo a los siguientes tres ejemplos:

```
for(int i=0; i<10; i++) // For infinito
   i = 0;  //La variable se inicializa siempre

for(int j=1; j<10; j++)  // For infinito
   j--;    // La variable se decrementa siempre

for(int k=0; k<10; )
   k++;    // En lugar de incrementar en el for se
           // se realizar en el cuerpo
```

Java permite la modificación de variables de bucle, pero debe tener cuidado si ve preguntas en el examen que hacen esto.

4.2. THE FOR-EACH LOOP

La declaración de bucle for-each se compone de una sección de inicialización y un objeto sobre el que se iterará. El lado derecho del bucle for-each debe ser uno de los siguientes:

1. Una matriz de Java
2. Un objeto cuyo tipo implementa `java.lang.Iterable`.

Cubriremos lo que significa implementos en "Diseño de clase avanzado", pero por ahora solo necesita saber que el lado derecho debe ser una matriz o colección de elementos. Para el examen, debe saber que esto no incluye

todas las clases o interfaces del *framework* de colecciones, sino solo aquellas que implementan o amplían esa interfaz de colección. En el ejemplo anterior, tendríamos:

```
public void printNames(String[] names) {
   for(String name : names)
      System.out.println(name);
}
```

4.2.1. Tackling the for-each Statement

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 – Se debe usar String y no int
  System.out.print(name + " ");
}
```

4.2.2. Switching Between for and for-each Loops

Es posible que haya notado que en los ejemplos anteriores de `for-each`, había una coma adicional impresa al final de la lista:

```
Lisa, Kevin, Roger,
```

Si quisiéramos imprimir solo la coma entre los nombres, podríamos convertir el ejemplo en un bucle for estándar, como en el siguiente ejemplo:

```
List<String> names = new ArrayList<String>();
names.add("Lisa");
names.add("Kevin");
names.add("Roger");

for(int i=0; i<names.size(); i++) {
   String name = names.get(i);
   if(i > 0) {
      System.out.print(", ");
   }
   System.out.print(name);
}
```

Este código de muestra generaría lo siguiente:

```
Lisa, Kevin, Roger
```

Este no es tan corto como nuestro ejemplo para cada uno, pero crea la salida que queríamos, sin la coma adicional.

COMPARING FOR AND FOR-EACH LOOPS

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 + ", ");
}
```

5. Controlling Flow with Branching

5.1. 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
```

Los bucles anidados pueden incluir `while` y `do/while`, por ejemplo:

```
int hungryHippopotamus = 8;
while(hungryHippopotamus>0) {
   do {
      hungryHippopotamus -= 2;
   } while (hungryHippopotamus>5);
   hungryHippopotamus--;
   System.out.print(hungryHippopotamus+", ");
}
```

Imprime:

```
3, 0,
```

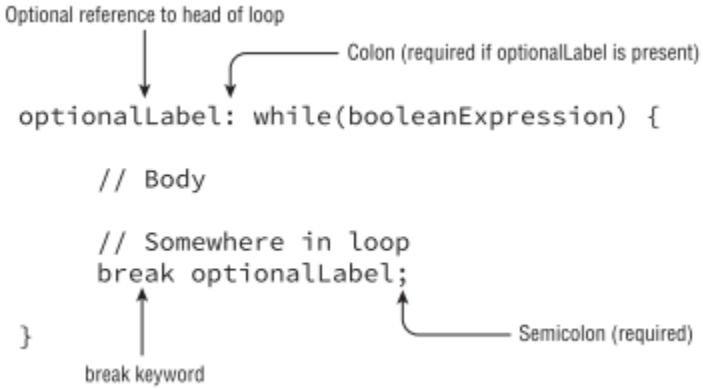
5.2. ADDING OPTIONAL LABELS

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.

5.3. 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 FindInMatrix {
  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:

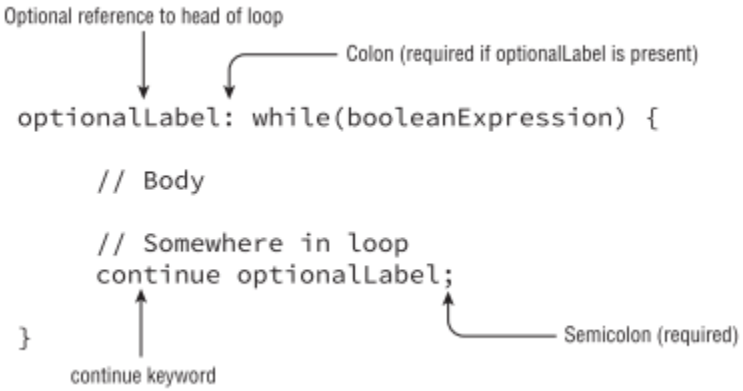
```
Value 2 found at: (2,0)
```

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)
```

5.4. 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:

```
1: public class CleaningSchedule {
2:    public static void main(String[] args) {
3:       CLEANING: for(char stables = 'a'; stables<='d'; stables++) {
4:          for(int leopard = 1; leopard<4; leopard++) {
5:             if(stables=='b' || leopard==2) {
6:                continue CLEANING;
7:             }
8:             System.out.println("Cleaning: "+stables+","+leopard);
9: } } } }
```

Imprime:
```
Cleaning: a,1
Cleaning: c,1
Cleaning: d,1
```

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

```
Cleaning: a,1
Cleaning: a,3
Cleaning: c,1
Cleaning: c,3
Cleaning: d,1
Cleaning: d,3
```

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

```
Cleaning: a,1
Cleaning: a,2
Cleaning: a,3
Cleaning: b,1
Cleaning: b,2
Cleaning: b,3
Cleaning: c,1
Cleaning: c,2
Cleaning: c,3
Cleaning: d,1
Cleaning: d,2
Cleaning: d,3
```

5.5. THE RETURN STATEMENT

Por ahora, sin embargo, debe estar familiarizado con la idea de que la creación de métodos y el uso de declaraciones de retorno se pueden usar como una alternativa al uso de etiquetas y declaraciones de interrupción. Por ejemplo, eche un vistazo a esta reescritura de nuestra clase anterior `FindInMatrix`:

```
public class FindInMatrixUsingReturn {
   private static int[] searchForValue(int[][] list, int v) {
      for (int i = 0; i < list.length; i++) {
         for (int j = 0; j < list[i].length; j++) {
            if (list[i][j] == v) {
               return new int[] {i,j};
            }
         }
      }
      return null;
   }

   public static void main(String[] args) {
      int[][] list = { { 1, 13 }, { 5, 2 }, { 2, 2 } };
      int searchValue = 2;
      int[] results = searchForValue(list,searchValue);

      if (results == null) {
         System.out.println("Value " + searchValue + " not found");
      } else {
         System.out.println("Value " + searchValue + " found at: " +
            "(" + results[0] + "," + results[1] + ")");
      }
```

```
   }
}
```

Esta clase es funcionalmente igual que la primera clase `FindInMatrix` que vimos anteriormente usando break. Dicho esto, encontramos código sin etiquetas y declaraciones de ruptura mucho más fáciles de leer y depurar.

Para el examen, necesitará conocer ambas formas. Solo recuerde que las declaraciones de retorno se pueden usar para salir de los bucles rápidamente.

5.6. UNREACHABLE CODE

Una faceta de `break`, `continue` y `return` que debe tener en cuenta es que cualquier código colocado inmediatamente después de ellos en el mismo bloque se considera inalcanzable y no se compilará. Por ejemplo:

```
int checkDate = 0;
while(checkDate<10) {
   checkDate++;
   if(checkDate>100) {
      break;
      checkDate++;  // DOES NOT COMPILE
   }
}
```

Lo mismo ocurre con las declaraciones `continue` y `return`, como se muestra en los dos ejemplos siguientes:

```
int minute = 1;
WATCH: while(minute>2) {
   if(minute++>2) {
      continue WATCH;
      System.out.print(minute);  // DOES NOT COMPILE
   }
}

int hour = 2;
switch(hour) {
   case 1: return; hour++;  // DOES NOT COMPILE
   case 2:
}
```

Independientemente de la ejecución, el compilador informará un error si encuentra algún código que considere inalcanzable, en este caso cualquier declaración inmediatamente después de una declaración de break, `continue` o `return`.

5.7. REVIEWING BRANCHING

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`
`while`	Si	Si	Si
`do-while`	Si	Si	Si
`for`	Si	Si	Si

`switch`	Si	Si	No

6. Review Questions

6.1.

Which of the following data types can be used in a switch statement? (Choose all that apply.)

```
A. enum
B. int
C. Byte
D. long
E. String
F. char
G. var
H. double
```

6.2.

Which statements, when inserted independently into the following blank, will cause the code to print 2 at runtime? (Choose all that apply.)

```
int count = 0;
BUNNY: for(int row = 1; row <=3; row++)
 RABBIT: for(int col = 0; col <3 ; col++) {
      if((col + row) % 2 == 0)
        _____________________;
      count++;
 }
System.out.println(count);
```

```
A. break BUNNY
B. break RABBIT
C. continue BUNNY
D. continue RABBIT
E. break
F. continue
```

G. None of the above, as the code contains a compiler error.

6.3.

Given the following method, how many lines contain compilation errors? (Choose all that apply.)

```
private DayOfWeek getWeekDay(int day, final int thursday) {
 int otherDay = day;
 int Sunday = 0;
 switch(otherDay) {
      default:
      case 1: continue;
      case thursday: return DayOfWeek.THURSDAY;
      case 2: break;
      case Sunday: return DayOfWeek.SUNDAY;
      case DayOfWeek.MONDAY: return DayOfWeek.MONDAY;
 }
```

```
 return DayOfWeek.FRIDAY;
}
```

A. None, the code compiles without issue.
B. 1
C. 2
D. 3
E. 4
F. 5
G. 6
H. The code compiles but may produce an error at runtime.

6.4.

What is the output of the following code snippet?

```
2: double iguana = 0;
3: do {
4:    int snake = 1;
5:    System.out.print(snake++ + " ");
6:    iguana--;
7: } while (snake <= 5);
8: System.out.println(iguana);
```

A. 1 2 3 4 -4.0
B. 1 2 3 4 -5.0
C. 1 2 3 4 5 -4.0
D. 0 1 2 3 4 5 -5.0
E. The code does not compile.
F. The code compiles but produces an infinite loop at runtime.
G. None of the above

6.5.

What is the output of the following code snippet? (Choose all that apply.)

```
2: var tailFeathers = 3;
3: final var one = 1;
4: switch (tailFeathers) {
5:    case one: System.out.print(3 + " ");
6:    default: case 3: System.out.print(5 + " ");
7: }
8: while (tailFeathers > 1) {
9:    System.out.print(--tailFeathers + " "); }
```

A. 3
B. 5 1
C. 5 2
D. 3 5 1
E. 5 2 1
F. The code will not compile because of lines 3–5.
G. The code will not compile because of line 6.