

Operators

1. Creating Decision-Making Statements

1.1. STATEMENTS AND BLOCKS

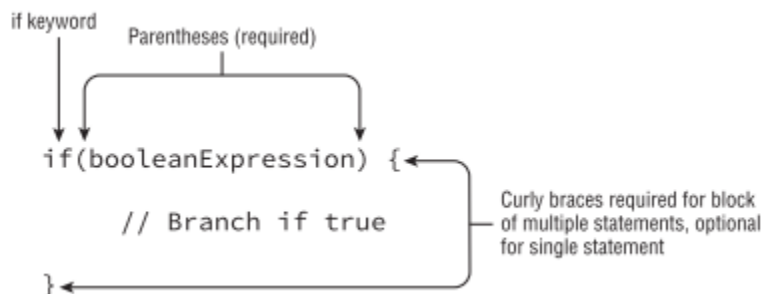
Una declaración de Java (*Java Statement*) es una unidad completa de ejecución en Java, terminada con un punto y coma (;).

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");

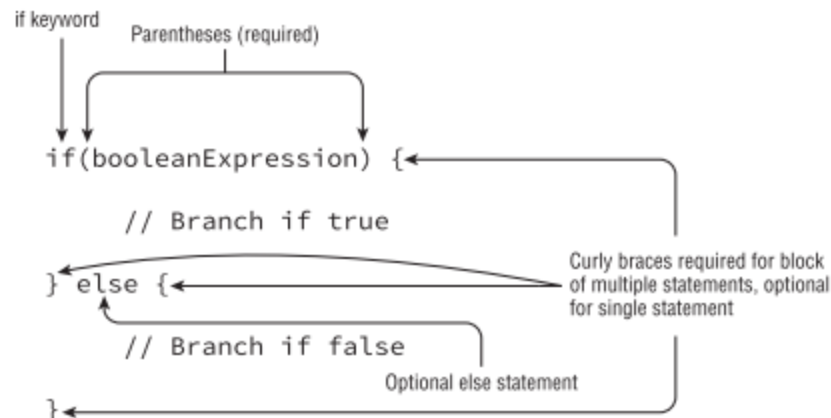
if(hourOfDay < 11) {
    System.out.println("Good Morning");
    morningGreetingCount++;
}
```

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:

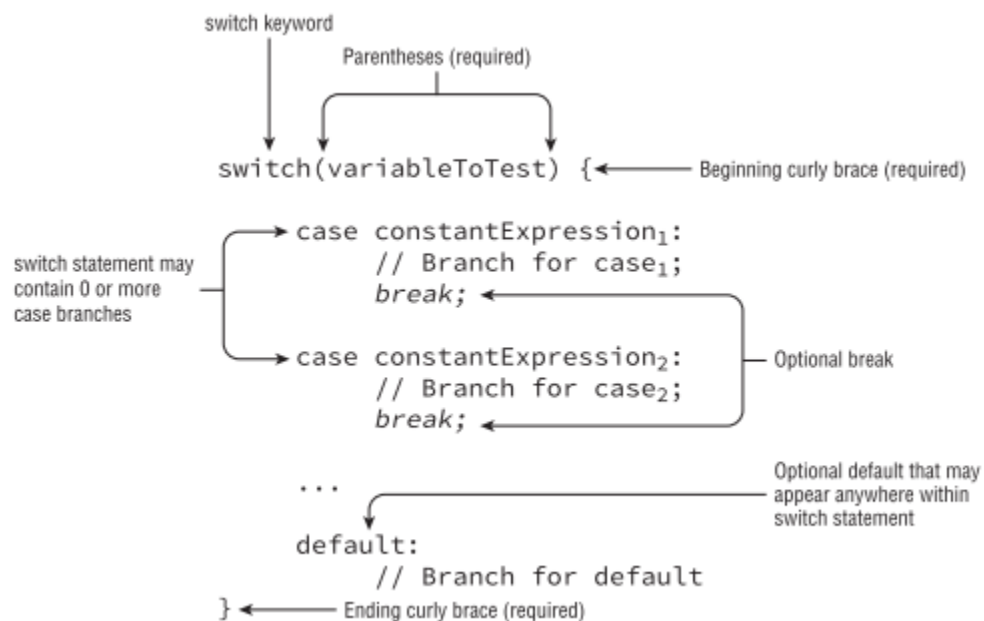
```

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 COMPILE
        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:

```

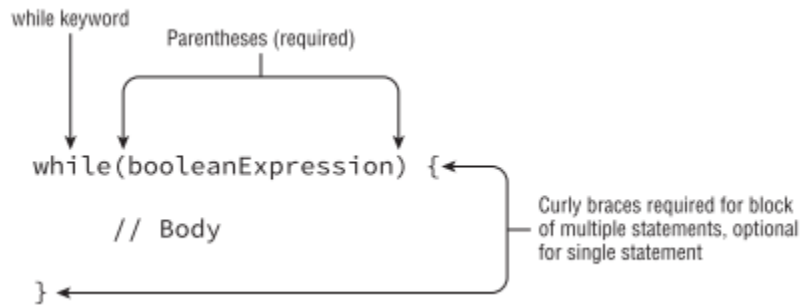
short size = 4;
final int small = 15;
final int big = 1_000_000;
switch(size) {
    case small:
    case 1+2 :
    case big: // DOES NOT COMPILE
}

```

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

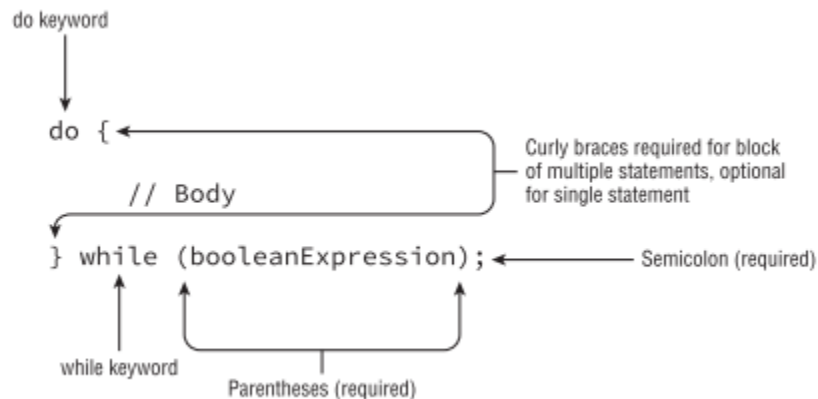


Por ejemplo:

```

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

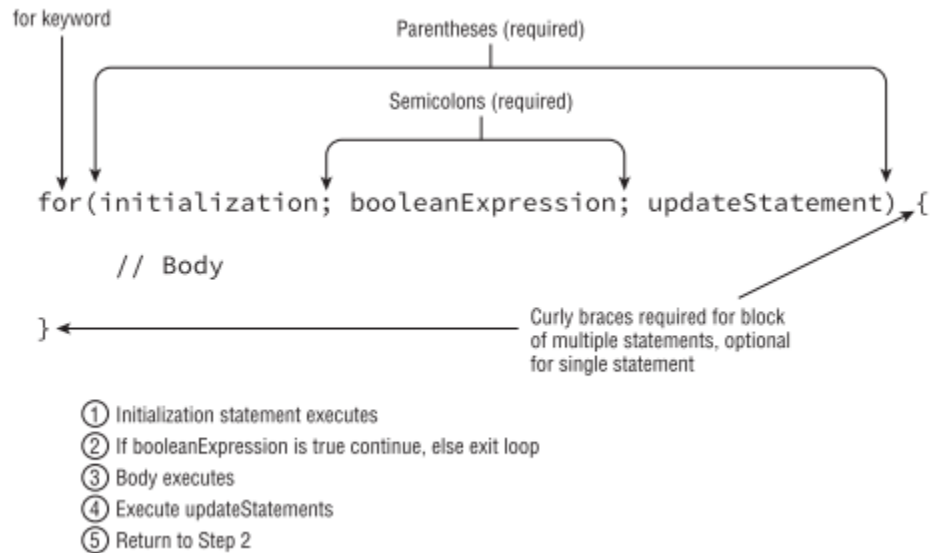
Lo más importante que debes tener en cuenta cuando utilices cualquier estructura de control de repetición es asegurarte de que siempre terminen. Veamos un ejemplo:

```
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
    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

Y también:

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

Imprime:

3	0
---	---

6. Review Questions

6.1.

6.2.

6.3.

6.4.

6.5.