Flow Control

If | if-else | operador ternario

Evalúa una condición y únicamente espera un `boolean` o `Boolean` .

Es aceptable omitir la parte `else` , pero no omitir el `if` ni su bloque o instrucción.

```
boolean testValue = false;
if (testValue == true)
else
    System.out.println("value is false");
```

Won't compile

No hay límites de if anidados teóricamente

Operador ternario

`?:` → Es un `if-else` compacto.

Se pueden utilizar métodos con retorno

En el lado de la evaluación se espera un `boolean` o no compila

Las 3 partes del operador son obligatorias o no compila.

El operador ternario debe tener una asignación de variable al final, de lo contrario no compila.

```
(5000 > 2000)? 15 : 10;
```

Won't compile; not a statement

No puede incluir bloques de código

La asignación final debe ser congruente.

Switch

Sólo se puede definir un caso `default` .

El caso `default` se ejecuta cuando no se encuentran coincidencias con otros casos.

`Break;` es utilizado para salir del switch.

Argumentos válidos

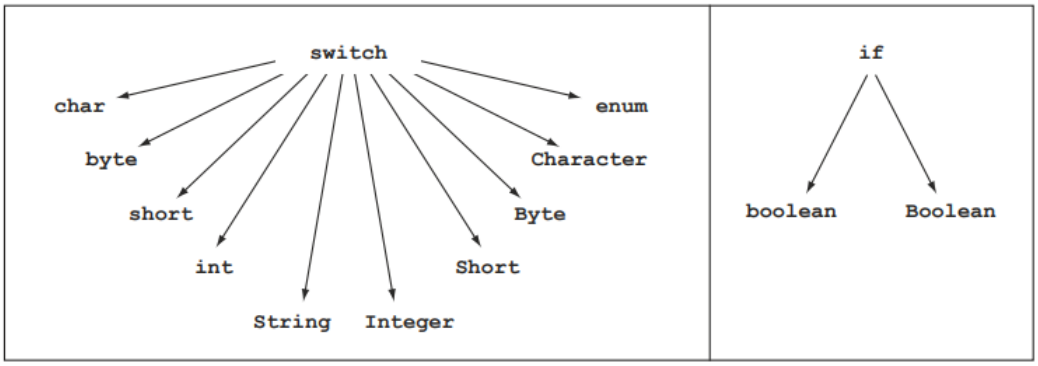

Figure 5.9 Types of arguments that can be passed to a `switch` statement and an `if` construct

No se aceptan `long` `float` `double` `boolean` → No compila

Es posible pasar expresiones cuyo retorno coincida con los tipos aceptados.

En el caso de objetos, no puede pasarse `null` → Causa `NullPointerException`

Etiquetas

No se permiten etiquetas duplicadas

Las etiquetas en los `case` deben ser una constante de tiempo de compilación, quiere decir que su valor debe conocerse cuando el código compila.

```
int a=10, b=20, c=30;
switch (a) {
    case b+c: System.out.println(b+c); break;
    case 10*7: System.out.println(10*7512+10); break;
}
```

❶ Not allowed

❷ Allowed

La única manera de hacerlo es marcar las variables como finales y definidas, así se garantiza que no hay cambio en su valor.

```
final int a = 10;
final int b = 20;
final int c = 30;
switch (a) {
    case b+c: System.out.println(b+c); break;
}
```

❶ **Expression b+c is compile-time constant**

Una constante en tiempo de compilación es una variable definida como final e inicializada. → Esas son las que puede utilizar switch.

Null no permitido

No se permite que una etiqueta en switch sea `null` → No compila

```
String name = "Paul";
switch (name) {
    case "Paul": System.out.println(1);
            break;
    case null: System.out.println("null");
}
```

null isn't allowed as a case label.

For

```
for (initialization; condition; update) {
     statements;
}
```

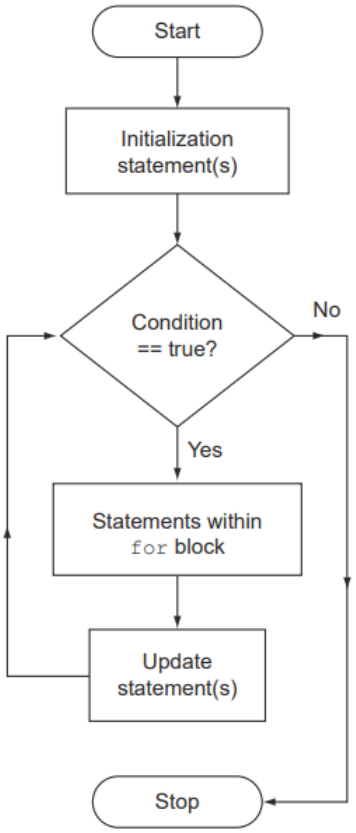

Figure 5.11 The flow of control in a `for` loop

Bloque de inicialización

Se ejecuta una sola vez

Se pueden definir múltiples variables pero de un sólo tipo

```
int tableOf = 25;
for (int ctr = 1, num = 100000; ctr <= 5; ++ctr) {
    System.out.println(tableOf * ctr);
    System.out.println(num * ctr);
}
```

Define and assign multiple variables

```
for (int j=10, long longVar = 10; j <= 1; ++j) { }
```

Can't define variables of different types in an initialization block

El scope de las variables definidas en este bloque están limitadas al for.

Condición

Se evalúa por cada iteración antes de ejecutar el cuerpo.

El loop termina una vez que la condición es `false`

Un for puede definir una condición, no más, no menos.

Actualización

El código se ejecuta después del cuerpo

Puede definir múltiples sentencias separadas por `,` , incluyendo llamadas a métodos.

```
public class ForIncrementStatements {
    public static void main(String args[]) {
        String line = "ab";
        for (int i=0; i < line.length(); ++i, printMethod())
            System.out.println(line.charAt(i));
    }
```

The increment block can also call methods.

Partes opcionales

Las tres partes del loop son opcionales.

Hay que seguir indicando las tres partes utilizando `;`

```
int a = 10;
for(; a < 5; ++a) {
    System.out.println(a);
}
```

Valid for loop without any code in the initialization block

In the following example, the termination condition is missing, resulting in a *potentially* infinite loop (*potentially* because it can be stopped with a `break` statement or an exception):

```
for(int a = 10; ; ++a) {
    System.out.println(a);
}
```

Missing termination condition implies infinite loop

The following code doesn't include code in its update clause but compiles successfully:

```
for(int a = 10; a > 5; ) {
    System.out.println(a);
}
```

Missing update clause

It's interesting to note that the following code is valid:

```
for(;;)
    System.out.println(1);
```

Loop mejorado

Se leé: por cada elemento de tipo `int var` en el arreglo `myArray` , haz.....

Limitaciones

- No se puede utilizar para inicializar un arreglo y modificar su elementos.
- No puede borrar o modificar elementos de una colección.
- No se puede iterar múltiples colecciones.

Es recomendable utilizar el ciclo mejorado para iterar, no para inicializar, modificar o filtrar

while y do-while

while

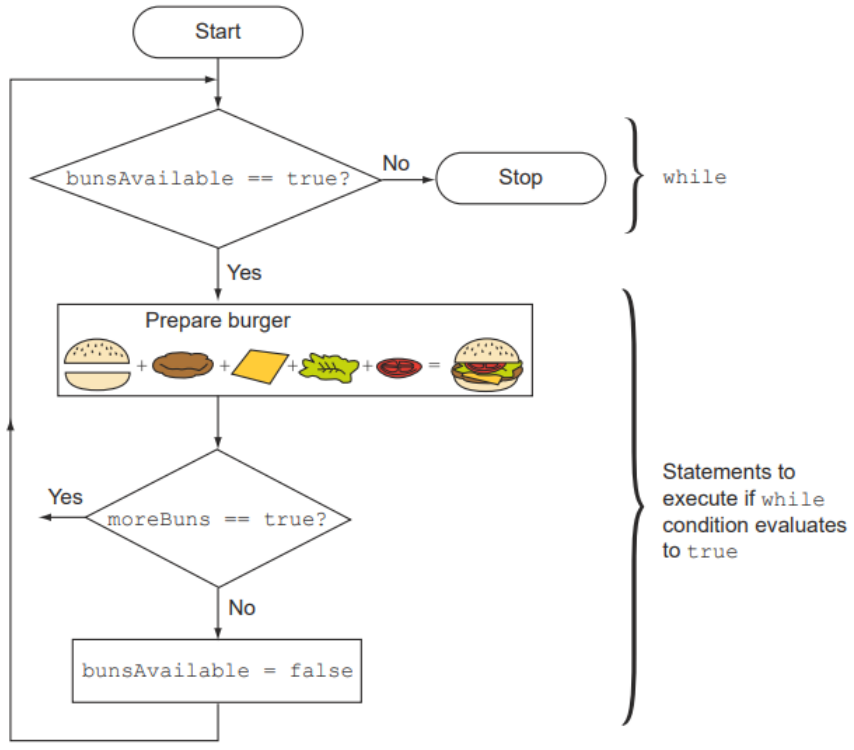

Figure 5.18 A flowchart depicting the flow of code in a `while` loop

do-while

Se debe colocar `;` al final de la sentencia `while`

Sentencias `break` y `continue`

`break`

La sentencia `break` se utiliza para romper o salir de un `for` `for-each` `do` `do-while` `switch`

Cuando se utiliza `break` en ciclos anidados se afecta al ciclo inner.

`continue`

La sentencia `continue` interrumpe la iteración actual y continua con la siguiente.

Al utilizar `continue` en ciclos anidados se afecta al ciclo inner.

Sentencias etiquetadas

Las etiquetas pueden ser utilizadas en:

- Códigos de bloques
- Loops
- Condicionales (switch, if)
- Expresiones
- Asiganciones
- Sentencias `return`
- bloques try
- sentencias `throws`

No se pueden añadir etiquetas a declaraciones de variables → no compila

```
outer :
    int[] myArray = {1,2,3};
```

Variable declaration that fails compilation

Sentencias `break` con etiquetas

```
String[] programmers = {"Outer", "Inner"};
outer:
for (String outer : programmers) {
    for (String inner : programmers) {
        if (inner.equals("Inner"))
            break outer;
        System.out.print(inner + ":");
    }
}
```

Exits the outer loop, marked with label outer

The output of the preceding code is

```
Outer:
```

Sentencias `continue` con etiquetas

```
String[] programmers = {"Paul", "Shreya", "Selvan", "Harry"};
outer:
for (String name1 : programmers) {
    for (String name : programmers) {
        if (name.equals("Shreya"))
            continue outer;
        System.out.println(name);
    }
}
```

Skips remaining code for current iteration of outer loop and starts with its next iteration

The output of the preceding code is

```
Paul
Paul
Paul
Paul
```