Estructuras repetitivas en JAVA

Cómo y porqué se usan los bucles o ciclos en Java.

Los ciclos o también conocidos como bucles, son una estructura de control de total importancia para el proceso de creación de un programa.

Java utiliza los ciclos que ya se han visto en lenguaje C. Recordando que un ciclo en Java permite repetir una o varias instrucciones cuantas veces sea necesario.

Diferentes tipos de ciclos en Java:

Ciclo for en Java

Ciclo while en Java

Ciclo do-while en Java

Sintaxis del Ciclo For en Java:

La sintaxis de un ciclo for es simple en Java, en realidad en la mayoría de los lenguajes de alto nivel es incluso muy similar, de hecho, con tan solo tener bien claros los 3 componentes del ciclo for (inicio, final y tamaño de paso) tenemos prácticamente todo hecho

```
for (inicialización; condición; expresión de incremento) {
   // Código a ejecutar en cada iteración
}
```

Ejemplo

Mostrar por pantalla los números pares pertenecientes al intervalo [500,1000]

```
2  public class For
3  {
4      public static void main(String args[])
5      {
6          for(int i=500;i<=1000;i+=2)
7          {
8              System.out.println(i);
9          }
10     }
11 }
12
```

Sintaxis del Ciclo while en Java:

while (condición) {

// Código a ejecutar en cada iteración

// Actualización de la condición

}

Ejemplo

En este ejemplo, se utiliza un ciclo while para repetir una serie de instrucciones mientras se cumpla una condición.
En cada iteración, se imprime en la consola el valor del contador y luego se incrementa en 1. El ciclo se repetirá mientras la condición contador <= 5 sea verdadera.

```java
public class CicloWhileEjemplo {
    public static void main(String[] args) {
        int contador = 1;

        while (contador <= 5) {
            System.out.println("El valor del contador es: " + contador);
            contador++;
        }
    }
}
```

En el ejemplo anterior, la condición contador <= 5 indica que el ciclo se repetirá mientras el contador sea menor o igual a 5. Dentro del ciclo, se imprime el valor del contador y se incrementa en 1 con contador++.

Al ejecutar el programa, verás que se imprimirán los valores del 1 al 5 en la consola, ya que el ciclo while se repetirá cinco veces.

Recuerda que es importante asegurarte de que la condición se actualice en algún momento dentro del ciclo para evitar un ciclo infinito. En este ejemplo, el contador se incrementa en cada iteración, lo que finalmente hará que la condición sea falsa y el ciclo se detenga.

Sintaxis del Ciclo Do-while en Java:

```
do {
   // Código a ejecutar en cada iteración
   // Actualización de la condición
} while (condición);
```

Ejemplo

```java

import java.util.Scanner;

public class CicloDoWhileEjemplo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int numero;

        do {
            System.out.print("Ingrese un número mayor que cero: ");
            numero = scanner.nextInt();
        } while (numero <= 0);

        System.out.println("Número ingresado válido: " + numero);

        scanner.close();
    }
}

```

En el ejemplo anterior, el bloque de código dentro del do se ejecuta primero sin verificar la condición. Luego, se verifica la condición numero <= 0 en el while. Si la condición es verdadera, el ciclo se repetirá y se solicitará al usuario que ingrese un número nuevamente. Si la condición es falsa, es decir, el número ingresado es válido, el ciclo se detendrá y se imprimirá el mensaje con el número válido.

Al ejecutar el programa, el ciclo do-while solicitará al usuario que ingrese un número mayor que cero. Si el usuario ingresa un número negativo o cero, se le pedirá que ingrese nuevamente un número válido hasta que se cumpla la condición.

Recuerda que el ciclo do-while es útil cuando se necesita ejecutar un bloque de código al menos una vez, independientemente de la condición. Luego, la condición se verifica al final de cada iteración.

Aparte de los ciclos for, while y do-while, Java también ofrece el ciclo **for-each**, que se utiliza específicamente para recorrer elementos de una colección o arreglo.

Sintaxis del Ciclo for-each en Java:

```
for (tipo elemento : colección) {
   // Código a ejecutar en cada iteración
}
```

Ejemplo

```
1
2  public class CicloForEachEjemplo {
3      public static void main(String[] args) {
4          int[] numeros = {1, 2, 3, 4, 5};
5
6          for (int numero : numeros) {
7              System.out.println(numero);
8          }
9      }
10 }
11
```

En este ejemplo, se utiliza el ciclo for-each para recorrer los elementos del arreglo números. En cada iteración, el valor del elemento se asigna a la variable número y se puede utilizar dentro del bloque del ciclo.

En el ejemplo anterior, el tipo del elemento es int y la colección es el arreglo numeros. En cada iteración, se imprime el valor del elemento en la consola.

Es importante tener en cuenta que el ciclo for-each solo se utiliza para recorrer elementos en una dirección hacia adelante y no permite modificar los elementos de la colección o el arreglo durante la iteración. Además, no se tiene acceso al índice del elemento en el ciclo for-each.

El ciclo for-each es particularmente útil cuando se necesita recorrer una colección o arreglo sin procuparse por los detalles de la indexación y se desea una sintaxis más concisa y legible.

Recuerda que puedes utilizar el tipo adecuado para el elemento en el ciclo for-each, según el tipo de datos de la colección o arreglo que estés recorriendo.

Cuando conviene usar for each?

El ciclo for-each es conveniente cuando necesitas recorrer todos los elementos de una colección o arreglo sin necesidad de conocer o manipular los índices. Aquí hay algunas situaciones en las que el ciclo for-each puede ser especialmente útil:

1. Recorrido de arreglos: Si tienes un arreglo y solo necesitas acceder a los elementos uno por uno, sin necesidad de realizar operaciones en los índices, el ciclo for-each es una buena opción. Proporciona una sintaxis más concisa y fácil de leer que el ciclo for tradicional.

2. Recorrido de colecciones: El ciclo for-each es particularmente útil para recorrer colecciones de objetos, como listas, conjuntos o mapas. Te permite iterar sobre los elementos de la colección de manera eficiente y sencilla, sin preocuparte por los detalles de la implementación subyacente.

3. Evitar errores de índice: Al utilizar el ciclo for-each, no necesitas preocuparte por los índices de los elementos, lo que reduce la posibilidad de errores relacionados con el manejo incorrecto de los índices. Esto es especialmente útil cuando no necesitas utilizar los índices y solo te interesa trabajar con los elementos en sí.

4. Mejor legibilidad: El ciclo for-each tiene una sintaxis más clara y legible en comparación con el ciclo for tradicional, ya que te enfocas en los elementos en lugar de los índices. Esto facilita la comprensión del código por parte de otros programadores y mejora la mantenibilidad del mismo.

En resumen, el ciclo for-each es útil cuando solo necesitas recorrer los elementos de una colección o arreglo y no necesitas acceder a los índices o modificar los elementos durante la iteración. Proporciona una sintaxis más sencilla y legible, lo que facilita el desarrollo y mantenimiento del código.

Estructuras de control en JAVA

Uso del If en Java:

La estructura if en Java se utiliza para ejecutar un bloque de código si se cumple una condición determinada.

Ejemplo

```
int edad = 25;


if (edad >= 18) {
    System.out.println("Eres mayor de edad");
} else {
    System.out.println("Eres menor de edad");
}
```

Este programa verifica una edad y devuelve en el caso que se verifique la condición `edad >= 18`, el mensaje: `”Eres mayor de edad”` y caso contrario: `”el mensaje Eres menor de edad”`

Ejemplo

Programa que verifica la positividad de un número

```
int numero = 7;


if (numero > 0) {
    System.out.println("El número es positivo");
} else if (numero < 0) {
    System.out.println("El número es negativo");
} else {
    System.out.println("El número es cero");
}
```

La estructura **IF** en Java también se puede anidar, lo que significa que puedes tener un **IF** dentro de otro **IF**. Esto se utiliza cuando necesitas realizar comprobaciones más complejas.

Por ejemplo con 2 o más condiciones se utiliza un if anidado

En este caso para comprobar si la persona es mayor de edad y si tiene una licencia de conducir. Dependiendo de las condiciones, se muestra el mensaje correspondiente.

Si es mayor de edad y tiene licencia de conducir “Puedes conducir un coche"

Si es menor de edad "Eres menor de edad, no puedes conducir"

Si no tiene licencia "No puedes conducir sin licencia"

Ejemplo

```
int edad = 25;
boolean tieneLicencia = true;
if (edad >= 18) {
    if (tieneLicencia) {
        System.out.println("Puedes conducir un coche");
    } else {
        System.out.println("No puedes conducir sin licencia");
    }
} else {
    System.out.println("Eres menor de edad, no puedes conducir");
}
```

También podría hacerse con un solo IF y operadores lógicos, pero en ese caso nos daría la información de manera ambigua

```
int edad = 25;
boolean tieneLicencia = true;
if (edad >= 18 && tieneLicencia) {
    System.out.println("Puedes conducir un coche");
} else {
    System.out.println("No puedes conducir sin licencia o eres menor de
edad");
}
```

Uso del switch en JAVA

Sintaxis básica de la declaración switch

```
switch (expresion) {
    case valor1:
        // Bloque de código a ejecutar si la expresión es igual a valor1
        break;
    case valor2:
        // Bloque de código a ejecutar si la expresión es igual a valor2
        break;
    // ... más casos si es necesario
    default:
        // Bloque de código a ejecutar si la expresión no coincide con
ninguno de los valores anteriores
}
```

El uso es similar al del lenguaje C

En JAVA solo aceptaba variables int, byte, short, y char

Aunque en versiones más actuales se admiten expresiones de tipo String.

Ejemplo

```
String calificacion = "B";
switch (calificacion) {
    case "A":
        System.out.println("Excelente");
        break;
    case "B":
        System.out.println("Bueno");
        break;
    case "C":
        System.out.println("Regular");
        break;
    case "D":
        System.out.println("Insuficiente");
        break;
    case "F":
        System.out.println("Reprobado");
        break;
    default:
        System.out.println("Calificación no válida");
}
```

Estructuras de salto en JAVA

En Java, tanto **default** como **break** son palabras clave que se utilizan en el contexto de las estructuras de control **switch**. Aunque pueden parecer similares, tienen diferentes funciones y comportamientos:

- **default:**

En una estructura de control switch, default se utiliza como una opción predeterminada cuando ninguna de las expresiones case coincide con el valor evaluado.

Si ninguna de las expresiones case coincide con el valor, el bloque de código asociado a default se ejecutará. **default** se coloca al final del bloque switch y es opcional. Si no se proporciona un **default**, no ocurrirá ninguna acción cuando ninguna de las expresiones case coincida.

- **break**:

En una estructura de control switch, **break** se utiliza para salir del bloque switch una vez que se ejecuta un caso correspondiente al valor evaluado.

Después de ejecutar el bloque de código asociado a un case, se sale automáticamente del switch si no se incluye un **break**. Si se omite el **break**, el programa continuará ejecutando el código en el siguiente case sin importar si la expresión coincide o no. Esto se conoce como "fall-through" (caída a través).

El **break** se utiliza para evitar la ejecución de los case siguientes y salir del switch una vez que se ha encontrado una coincidencia.

En resumen, **default** se utiliza como una opción predeterminada cuando ninguna de las expresiones case coincide, mientras que **break** se utiliza para salir del switch después de ejecutar un case. **default** se ejecuta si no se encuentra ninguna coincidencia, mientras que **break** se utiliza para controlar el flujo de ejecución dentro del switch.

El siguiente ejemplo es para ilustrar el uso de **default** y **break** en una estructura de control **switch:**

```java
int numero = 2;

switch (numero) {
    case 1:
        System.out.println("El número es 1");
        break;
    case 2:
        System.out.println("El número es 2");
        break;
    case 3:
        System.out.println("El número es 3");
        break;
    default:
        System.out.println("El número no coincide con ningún caso");
        break;
}
```

En este ejemplo, tenemos una variable numero que tiene un valor de 2.
En la estructura de control switch, comparamos el valor de numero con diferentes casos utilizando case.

En el primer case, comparamos numero con 1. Si coincide, se imprime "El número es 1" y se utiliza **break** para salir del switch.

En el segundo case, comparamos número con 2. Como coincide con el valor de número, se imprime "El número es 2" y nuevamente se utiliza **break** para salir del switch.

No hay un case para el valor **4**, por lo que el programa pasa al **default**.
Aquí se imprime "El número no coincide con ningún caso" y se utiliza **break** para salir del switch.

Si **break** no se utilizara después de cada case, el programa continuaría ejecutando el código en los case siguientes sin importar si hay coincidencia o no. Sin embargo, en este ejemplo, cada case tiene un break para controlar el flujo y salir del switch después de ejecutar el código correspondiente.

Es importante incluir el bloque **default** para manejar casos en los que no haya ninguna coincidencia en los case. Esto garantiza que haya una acción predeterminada definida en caso de que ninguna de las expresiones case coincida con el valor evaluado.