	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	83/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Guía práctica de estudio 06: Estructuras de repetición

---





---

***Elaborado por:***

M.C. M. Angélica Nakayama C.  
Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	84/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Guía práctica de estudio 06: Estructuras de repetición

### Objetivo:

Implementar programas utilizando estructuras de repetición en un lenguaje orientado a objetos.

### Actividades:


- Conocer la sintaxis para declarar diversas estructuras de repetición.
- Implementar el uso de estructuras de repetición en un programa.

### Introducción


Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para que esos elementos se combinen. Estas reglas se denominan **sintaxis del lenguaje**. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazados por la máquina.

La **sintaxis** de un lenguaje de programación se define como el conjunto de reglas que deben seguirse al escribir el código fuente de los programas para considerarse como correctos para ese lenguaje de programación.

Las **estructuras de control** permiten modificar el flujo de ejecución de las instrucciones de un programa. Todas las estructuras de control tienen un único punto de entrada. Las estructuras de control se pueden clasificar en: secuenciales, transferencia de control e iterativas. Básicamente lo que varía entre las estructuras de control de los diferentes lenguajes es su sintaxis, cada lenguaje tiene una sintaxis propia para expresar la estructura.

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	85/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

**NOTA:** En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	86/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Estructuras de control

Las estructuras de programación o estructuras de control permiten **tomar decisiones** o **realizar un proceso repetidas veces**. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a concepto, aunque su sintaxis varía de un lenguaje a otro. La sintaxis de Java coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no represente ninguna dificultad adicional.

### Estructuras de repetición

Las estructuras de repetición, lazos, ciclos o bucles se utilizan para realizar un proceso **repetidas veces**. El código incluido entre las llaves { } (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumplan determinadas condiciones.


Hay que prestar especial atención a los ciclos infinitos, hecho que ocurre cuando la condición de finalizar el ciclo no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

### WHILE

Las sentencias **statements** se ejecutan mientras **booleanExpression** sea **true**.

```
while (booleanExpression) {
    statements;
}
```

```
public class While {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int number = random.nextInt();
        System.out.println("number: " + number);
        while (number > 0) {
            number = random.nextInt();
            System.out.println("number: " + number);
        }
    }
}
```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	87/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## DO-WHILE

Es similar al ciclo **while** pero con la particularidad de que el control está **al final del ciclo** (lo que hace que el ciclo se ejecute **al menos una vez**, independientemente de que la condición se cumpla o no). Una vez ejecutados los **statements**, se evalúa la condición: si resulta **true** se vuelven a ejecutar las sentencias incluidas en el ciclo, mientras que si la condición se evalúa a **false** finaliza el ciclo.

```
do {
    statements
} while (booleanExpression);
```


```
public class DoWhile {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int number;
        do {
            number = random.nextInt();
            System.out.println("number: " + number);
        } while (number > 0);
    }
}
```

## FOR

La forma general del **for** es la siguiente:

```
for (initialization; booleanExpression; increment) {
    statements;
}
```

La sentencia o sentencias **initialization** se ejecutan al comienzo del **for**, e **increment** después de **statements**. La **booleanExpression** se evalúa al comienzo de cada iteración; el ciclo termina cuando la expresión de comparación toma el valor **false**. Cualquiera de las tres partes puede estar vacía. La **initialization** y el **increment** pueden tener varias expresiones separadas por comas.

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	88/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

public class Argumentos {
    public static void main(String args[]){
        if ( args.length == 0 ){
            System.out.println("ERROR!, al ejecutar el programa, se deben");
            System.out.println("pasar argumentos de la siguiente manera:.");
            System.out.println("java Argumentos ARG1 ARG2 ...");
        }

        for ( int i = 0 ; i < args.length ; i++ ){
            System.out.println("Parametro " + (i+1) + " : " + args[i]);
        }
    }
}

```

### BREAK y CONTINUE


La sentencia **break** es válida tanto para las bifurcaciones como para los ciclos. Hace que se **salga inmediatamente** del ciclo o bloque que se está ejecutando, sin realizar la ejecución del resto de las sentencias.

```

public class Break {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int number, limit = 5;
        while (limit > 0) {
            number = random.nextInt();
            System.out.println("number: " + number);
            if (number < 0){
                break;
            }
            limit--;
        }
        System.out.println("While done.");
    }
}

```


La sentencia **continue** se utiliza en los ciclos (no en bifurcaciones). Finaliza la iteración “i” que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la **siguiente iteración** (i+1).

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	89/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

public class Continue {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int number, limit = 5;
        do {
            number = random.nextInt();
            System.out.println("number: " + number);
            if (number < 0){
                continue;
            }
            limit--;
        } while (limit > 0);
        System.out.println("While done.");
    }
}

```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	90/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Colecciones

Una **colección** es un objeto que almacena un conjunto de **referencias a objetos**, es decir, es parecido a un arreglo de objetos. Sin embargo, a diferencia de los arreglos, las colecciones son **dinámicas**, en el sentido de que no tienen un tamaño fijo y permiten añadir y eliminar objetos en tiempo de ejecución.

Java incluye un amplio conjunto de clases para la creación y tratamiento de colecciones. Todas ellas proporcionan una serie de métodos para realizar las operaciones básicas sobre una colección, como son:

- Añadir objetos a la colección.
- Eliminar objetos de la colección.
- Obtener un objeto de la colección
- Localizar un objeto en la colección.
- Iterar a través de una colección.

Los principales tipos de colecciones que se encuentran por defecto en el API Java son:

### Conjuntos

Un conjunto (**Set**) es una colección desordenada (no mantiene un orden de inserción) y no permite elementos duplicados.


Clases de este tipo: HashSet, TreeSet, LinkedHashSet.

### Listas

Una lista (**List**) es una colección ordenada (debido a que mantiene el orden de inserción) pero permite elementos duplicados.

Clases de este tipo: ArrayList y LinkedList



	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	91/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Mapas

Un mapa (**Map** también llamado arreglo asociativo) es un conjunto de elementos agrupados con una llave y un valor: *<llave, valor>*

Donde las llaves no pueden ser repetidas y a cada valor le corresponde una llave. La columna de valores sí puede repetir elementos.

Clases de este tipo: HashMap, HashTable, TreeMap, LinkedHashMap.

Algunas de las clases más usadas para manejo de colecciones son las siguientes (todas ellas se encuentran en el paquete **java.util**):

## ArrayList

Se basa en un arreglo redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones. Para crear un objeto ArrayList se utiliza la siguiente sintaxis:

```
ArrayList<TipoDato> nombreVariable = new ArrayList<TipoDato>();
```


Ejemplo:

```
ArrayList<Integer> arreglo = new ArrayList<Integer>();
```

En este caso se creó un **ArrayList** llamado **arreglo**, el cual podrá contener elementos enteros (Integer).

Una vez creado, se pueden usar los métodos de la clase ArrayList para realizar las operaciones habituales con una colección, las más usuales son:

- **add(elemento)** – Añade un nuevo elemento al ArrayList y lo sitúa al final del mismo.
- **add(índice, elemento)** – Añade un nuevo elemento al ArrayList en la posición especificada por índice, desplazando hacia delante el resto de los elementos de la colección.
- **get(índice)** – Devuelve el elemento en la posición índice.

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	92/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

- **remove(índice)** – Elimina el elemento del ArrayList recorriendo los elementos de las posiciones siguientes. Devuelve el elemento eliminado.
- **clear()** – Elimina todos los elementos del ArrayList.
- **indexOf(elemento)** – Localiza en el ArrayList el elemento indicado devolviendo su posición o índice. En caso de que el elemento no se encuentre devuelve -1.
- **size()** – Devuelve el número de elementos almacenados en el ArrayList.

Ejemplo:

```
import java.util.ArrayList;

public class Colecciones {


    public static void main(String[] args) {
        ArrayList<Integer> arreglo = new ArrayList<Integer>();
        arreglo.add(1);
        arreglo.add(8);
        arreglo.add(5);
        arreglo.add(1, 9);
        System.out.println("Tamaño del array list " + arreglo.size());
        System.out.println("Elemento en la posición 3: " + arreglo.get(3));
        for (Integer elemento : arreglo) {
            System.out.println(elemento);
        }
    }
}
```


## Hashtable

La clase Hashtable representa un tipo de colección basada en claves, donde los elementos almacenados en la misma (valores) no tienen asociado un índice numérico basado en su posición, sino una clave que lo identifica de forma única dentro de la colección. Una clave puede ser cualquier tipo de objeto.

La utilización de colecciones basadas en claves resulta útil en aquellas aplicaciones en las que se requiera realizar búsquedas de objetos a partir de un dato que lo identifica. La creación de un objeto Hashtable se realiza de la siguiente manera:

```
Hashtable<TipoDatoClave, TipoDatoElemento> nombreVariable = new Hashtable<TipoDatoClave,
TipoDatoElemento>();
```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	93/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	94/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

```
Hashtable<String, Integer> miHashTable = new Hashtable<String, Integer>();
```

Los principales métodos de la clase Hashtable para manipular la colección son los siguientes:

- **put(clave, valor)** – Añade a la colección el elemento **valor**, asignándole la **clave** especificada. En caso de que exista esa **clave** en la colección el elemento se sustituye por el nuevo **valor**.
- **containsKey(clave)** – Indica si la **clave** especificada existe o no en la colección. Devuelve un **boolean**.
- **get(clave)** – Devuelve el **valor** que tiene asociado la **clave** que se indica. En caso de que no exista ningún elemento con esa **clave** asociada devuelve **null**.
- **remove(clave)** – Elimina de la colección el **valor** cuya **clave** se especifica. En caso de que no exista ningún elemento con esa **clave** no hará nada y devolverá **null**, si existe eliminará el elemento y devolverá una referencia al mismo.
- **size()** – Devuelve el número de elementos almacenados en el Hashtable.

Ejemplo:

```
import java.util.Hashtable;


public class Colecciones {

    public static void main(String[] args) {

        Hashtable<String, Integer> miTabla = new Hashtable<String, Integer>();
        miTabla.put("uno", 1);
        miTabla.put("dos", 2);
        miTabla.put("cinco", 5);

        System.out.println("Contiene a cuatro? " + miTabla.containsKey("cuatro"));

        for (String clave : miTabla.keySet()) {
            System.out.println(clave);
        }
        for (Integer valor : miTabla.values()) {
            System.out.println(valor);
        }
    }
}
```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	95/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Al no estar basado en índices, un Hashtable no se puede recorrer totalmente usando el for con una sola variable. Esto no significa que no se pueda iterar sobre un Hashtable, se puede hacer a través de una enumeración.

Los métodos proporcionados por la enumeración (**Enumeration**) permiten recorrer una colección de objetos asociada y acceder a cada uno de sus elementos.

Así, para recorrer completamente el Hashtable, se obtienen sus claves usando el método *keys()* y para cada una de las claves se obtiene su valor asociado usando *get(clave)*.

Ejemplo:

```
import java.util.Enumeration;
import java.util.Hashtable;

public class Colecciones {


    public static void main(String[] args) {

        Hashtable<String, Integer> miTabla = new Hashtable<String, Integer>();
        miTabla.put("uno", 1);
        miTabla.put("dos", 2);
        miTabla.put("cinco", 5);

        String clave;
        Integer valor;
        Enumeration<String> claves = miTabla.keys();
        while(claves.hasMoreElements()){
            clave = claves.nextElement();
            valor = miTabla.get(clave);
            System.out.println("Clave : " + clave + "\tValor : " + valor);
        }

    }

}
```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	96/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Bibliografía

*Martín, Antonio*

***Programador Certificado Java 2.***

*Segunda Edición.*

*México*

*Alfaomega Grupo Editor, 2008*

*Sierra Katy, Bates Bert*

***SCJP Sun Certified Programmer for Java 6 Study Guide***

*Mc Graw Hill*

*Dean John, Dean Raymond.*

***Introducción a la programación con Java***

*Primera Edición.*

*México*

*Mc Graw Hill, 2009*