PRO

UT5: Introducción al lenguaje Java

José Antonio
Benítez Chacón

Índice

1 Introducción

2 Asignaciones

3 Condicionales

4 Bucles

5 Tipos de datos

6 Visibilidad de las variables

7 Estructuras de datos

8 Iteradores

1. Introducción

Java

Características:

- Es un lenguaje de alto nivel.
- Es un lenguaje compilado (a bytecode) e interpretado (en JVM).
- Es un lenguaje con tipado estático y fuerte.
- Sintaxis similar a C++ pero más simple y segura.
- Es un lenguaje de programación multiparadigma (orientación a objetos, programación estructurada, programación funcional, etc.).

La **programación estructurada** es un paradigma que se basa en utilizar solo funciones (o subrutinas) y tres estructuras de control:

- secuencias de instrucciones (una tras otra)
- selecciones, elecciones o condicionales
- iteraciones, ciclos o bucles

2. Asignaciones

En Java, una asignación permite almacenar un valor de una variable utilizando el operador “=”. **Las variables tienen que ser declaradas con un tipo de datos antes de poder ser usadas**.

El nombre de esta variable o constantes (y también el de otras cosas como funciones) se llama **identificador**.

```
String saludo = "Hola, mundo.";
```

```
Scanner scanner = new Scanner(System.in);
System.out.println("Buenas, introduce tu edad: " );
int edad = scanner.nextInt();

System.out.println("Buenas, introduce tu nombre: " );
String nombre = scanner.next();

String mensaje = String.format("Bienvenido, %s, de %d años." , nombre, edad);
System.out.println(mensaje);
```

2. Asignaciones

Los identificadores de variables han de comenzar por una letra (minúscula por convención). El resto del nombre puede estar formado por caracteres alfanuméricos siguiendo el estilo “camelCase”. Aunque el guion bajo (“_”) y el símbolo del dólar (“$”) son válidos, no suelen utilizarse para este tipo de identificadores (el “_” solo suele usarse en constantes, que siempre irían en mayúsculas).

Hay distinción de mayúsculas y minúsculas. Es decir, las variables:
`varAux = 12;` y `varAUX = 13;`
son variables diferentes.

Además, un identificador no podrá ser nunca una de las palabras reservadas por el lenguaje (*while, for, if, class, break, public*, etc.)

NOTA: En **Java**, las **constantes** se declaran como **final** (no pueden cambiar su valor una vez establecido). Por convención, una constante tendrá su nombre entero en mayúsculas (pudiendo unirse con guiones bajos).

```
static final int MAYORIA_EDAD =18;
static final private String ABREV_DOCTOR = "Dr.";
```

3. Condicionales

Sirven para tomar una decisión en base a una condición. Si la condición se cumple, se ejecuta el bloque asociado. Este bloque estará indentado a la derecha, y cuando termine, se volverá a la indentación que tenía.

if: evalúa la condición y si se cumple ejecuta el bloque asociado
else: (opcional)sirve para asociar un bloque de código cuando no se han cumplido las evaluaciones anteriores del bloque condicional. Un bloque if/else puede encadenarse al anterior (pareciendo una especie de rama “else if”)

```
System.out.println("Introduce una letra: ");
Scanner scanner = new Scanner(System.in);
String letra = (scanner.next()).toLowerCase();

List<String> vocales = Arrays.asList("a", "e", "i", "o", "u");

if(vocales.contains(letra)){
 System.out.println("Has introducido una vocal no acentuada.");
}else if(Character.isLetter(letra.toCharArray()[0]) &&
letra.length()==1){
 System.out.println("Has introducido una letra");
}else{
 System.out.println("No has introducido una letra");
}
```

3. Condicionales

También puede utilizarse **switch** (parecido al **match** de python).

Con switch se evalúa una expresión y se comprueba sus resultados con case. Si un patrón coinciden se ejecutan todas los casos que vengan a posteriori (si no se corta con un **break**):

```
System.out.println("Introduce un animal: " );
Scanner scanner = new Scanner(System.in);
String animal = scanner.next();
switch(animal){
 case "perro":
   System.out.println("El animal elegido es un perro." );
   break;
 case "gato":
   System.out.println("El animal elegido es un gato." );
   break;
 default:
   System.out.println("El animal elegido no es un perro ni un gato." );
}
```

3. Condicionales

A partir de Java 14, se pueden usar las “switch expressions”, que eliminan la necesidad de terminar con un break cada bloque:

```
System.out.println("Introduce un animal: " );
Scanner scanner =  new Scanner(System.in);
String animal = scanner.next();
String mensaje =  switch(animal){
 case "perro" -> "El animal elegido es un perro." ;
 case "gato" -> "El animal elegido es un gato." ;
 default -> "El animal elegido no es un perro ni un gato." ;
};
System.out.println(mensaje);
```

4. Bucles

Un **bucle** es una estructura que repite el bloque de instrucciones que contenga mientras se cumpla una condición.

while

Evalúa una condición y, mientras se cumpla, ejecuta el bloque de instrucciones asociado

```
int i = 0;
while (i < 10) {
 System.out.println("Hola: " + i);
 i++;
}
```

NOTA: En Java **NO** puede tener un **else** al final, como sí ocurría en Python

4. Bucles

for (clásico)

Con un bucle for sirve para realizar una serie de operaciones mientras se cuempla una condición.

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

Su declaración tiene 3 partes, separadas por punto y coma (;):

for (inicialización de variables ; comprobación de condición ; actualización de variables){

    //Bloque de código que se ejecuta

}

- **inicialización de variables**: bloque de inicialización (y definición) de variables para realizar las iteraciones. Aunque suele aparecer es un bloque **opcional**.
- **comprobación de condición**: lógica que ha de cumplirse (devolver true) para que se ejecute una nueva iteración del bucle. Este bloque es **obligatorio**, ya que es el que marca que el bucle pare o no.
- **actualización de variables**: bloque para actualizar el valor de las variables al terminar cada iteración (incremento en 1, decremento, etc.). Aunque suele aparecer, es un bloque **opcional.**

4. Bucles

for (clásico)

Más ejemplos:

```
int i=0;
for (int j=1; i<10 ; i=i+2, j = j*i*i){
 System.out.println("i vale: "+i+" y j vale: "+j);
}
```

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

4. Bucles

for (colección de elementos)

Otra forma de utilizar el for es para recorrer elementos de una colección (List, Set, etc.). Se conoce como **for-each**:

```
int[] array1 = {1,2,3,4,5,6,7,8,9,10};

for(int elem : array1){
 System.out.println(elem);
}
```

```
Persona p1 = new Persona("Javi","López",23);
Persona p2 = new Persona("Sara","Pinto",21);
Persona p3 = new Persona("Andrea","Espino",19);
List<Persona> listaPersonas =  new ArrayList<>();
listaPersonas.add(p1);
listaPersonas.add(p2);
listaPersonas.add(p3);
for (Persona persona : listaPersonas){
 System.out.println(persona);
}
```

4. Bucles

break

Instrucción que sirve para romper el bucle en el que nos encontramos (el programa continuaría por la línea siguiente al bucle).

continue

Instrucción que sirve para terminar la iteración en curso del bucle. Se pasa a la siguiente.

```
for(int i = 10;i >= 0;i--){
 if(i%2==0){
   System.out.println("El valor de i es par");
 }else{
   continue;
 }

 if(i<=2){
   break;
 }else{
   System.out.println("El valor de i es "+i);
 }
}
```

5. Tipos de datos

Java es un lenguaje fuertemente tipado. Veamos primero algunos de los tipos básicos o primitivos:

Carácter (char)
Representa un solo carácter (letra, número, símbolo), ocupa 2 bytes y utiliza la codificación Unicode.
Se representa por un carácter entre comillas simples:

```
char car1 = 'A';
char car2 = '^';
char car3 = "Saludo".toCharArray()[2];
```

5. Tipos de datos

números

enteros:

- byte (1 byte)
- short (2 bytes)
- int (4 bytes)
- long 8 bytes

decimales o coma flotante:

- float (4 bytes)
- double (8 bytes)

booleanos (boolean)

```
short n1 = 231;
int n2 = 34123112;
long n3 = 10000000000000000000L;
double n4 = 241.23E20;
```

```
boolean a = true;
boolean b = 10 < 9;
boolean c = a || b;
boolean d = !c;
System.out.println(d);
```

5. Tipos de datos

Es necesario hacer conversión de tipos (o casting) si necesitamos convertir datos a otros tipos (si es posible)

```
double a = 23.0;
System.out.println("El doble de a es "+(a*2));
System.out.println("El doble de a es "+(int)(a*2));
```

Cada tipo de datos, además, admite unos operadores. Hay ciertos operadores que pueden adoptar una forma abreviada.

```
int a = 23;
a += 2; //Equivale a: a = a +2
a++; //Equivale a: a = a + 1 (postincremento)
++a; //Equivale a: a = a + 1 (preincremento)
a--; //Equivale a: a = a - 1 (postdecremento)
--a; //Equivale a: a = a - 1 (predecremento)
int a1 = 1,a2 = 1;
int b = ++a1;
int c = a2++;
System.out.println("a1: "+a1);
System.out.println("a2: "+a2);
System.out.println("b: "+b);
System.out.println("c: "+c);
```

6. Visibilidad de las variables

La visibilidad (o ámbito) de una variable determina la región de un programa donde se puede referenciar a esa variable por su nombre y acceder a su valor.

Variables locales
Variables que solo son accesibles (y existen) en el bloque de código en el que se declaran.

```
public static int calculaFibonacci(int a){
 int resultado = 0;
 if(a == 0 || a == 1){
   resultado = a;
 }else{
   resultado = calculaFibonacci(a-1)+calculaFibonacci(a-2);
 }
 return resultado;
}
```

En este caso, tenemos una variable (resultado) que solo es visible dentro del método “calculaFibonacci” en el que ha sido declarada.

6. Visibilidad de las variables

Variables globales

Java no admite variables globales tal cual (ya que fomenta la encapsulación y Orientación a Objetos).

Aún así, este comportamiento puede conseguirse utilizando atributos/variables con acceso público y de tipo static. Esto no es recomendable y debe evitarse en la mayoría de los casos.

```
public class PruebaGlobal {
 public static String variableGlobal = "Valor";
}
```

```
public class EjemploNuevo {
 public static void main(String[] args) {
   System.out.println("\"Variable Global\" vale: " + PruebaGlobal.variableGlobal);
   PruebaGlobal.variableGlobal = "cambio de valor";
   System.out.println("\"Variable Global\" vale: " + PruebaGlobal.variableGlobal);
 }
}
```

```
"Variable Global" vale: Valor
"Variable Global" vale: cambio de valor
```

6. Visibilidad de las variables

Variables de instancia (atributos)

Pueden utilizarse en toda la clase salvo en métodos estáticos (ya que cada objeto tiene su propia copia de valores).

```
public class Persona {
 private String nombre;
 private String apellido;

 public Persona(String nombre, String apellido) {
   this.nombre = nombre;
   this.apellido = apellido;
 }

 …
 public static String infoPersona(){
  return "Las personas son seres humanos" ;
 }
}
```

```
public static String infoPersona(){ no usages
  return "Las personas como "+nombre+" "+apellido+"son seres humanos";
}
```

6. Visibilidad de las variables

Variables estáticas

Se declaran con la palabra static y son compartidas por todos los objetos de la clase. Pueden usarse en métodos estáticos y no estáticos.

```
public class UsuarioRegistrado {
 private String nombre;
 private int id;

 private static int contadorUsuarios = 0;

 public UsuarioRegistrado(String nombre) {
   this.nombre = nombre;
   contadorUsuarios++;
   this.id = contadorUsuarios;
 }

 public static String muestraTotal(){
   return "Hasta la fecha, se han registrado " +contadorUsuarios+" usuarios.";
 }
}
```

6. Visibilidad de las variables

Ciclo de Vida de las Variables

- Variables locales: Existen mientras se ejecuta el bloque donde fueron declaradas.
- Variables de instancia: Existen mientras el objeto esté en memoria.
- Variables estáticas: Existen mientras la clase esté cargada en la JVM.

Buenas Prácticas

- Usa nombres descriptivos para las variables.
- Inicializa las variables locales antes de usarlas.
- Limita el uso de variables estáticas para evitar problemas con datos compartidos.
- Sigue las reglas de encapsulación: utiliza private para las variables y métodos de acceso (getters y setters) cuando sea necesario.

7. Estructuras de datos

Java proporciona una amplia gama de estructuras de datos, tanto en su API estándar como en librerías externas.

Ejemplos de estas colecciones son:

- Arrays
- Listas
- Mapas (“Diccionarios”)
- Conjuntos (Sets)

7. Estructuras de datos

Arrays

Los arrays son estructuras de tamaño fijo que permiten almacenar elementos del mismo tipo.
Existen dos formas principales de usarlos:

Mediante corchetes ([ ])

```java
public class EjemploArray {
 public static void main(String[] args) {
   // Declaración y creación de un array
   int[] numeros = new int[5];
   numeros[0] = 10;
   numeros[1] = 20;

   // Declaración e inicialización directa
   String[] nombres = {"Alicia", "Robe", "Ricky"};

   for (String nombre : nombres) {
     System.out.println("Nombre: " + nombre);
   }
 }
}
```

7. Estructuras de datos

Arrays

Usando la clase Arrays

```
import java.util.Arrays;

public class EjemploArray2 {
 public static void main(String[] args) {
   int[] numeros = {5, 2, 9, 1};
   int[] numeros2 = Arrays.copyOf(numeros, 7);
   numeros2[4]=3;
   numeros2[5]=10;
   numeros2[6]=-4;

   Arrays.sort(numeros2);
   System.out.println("Array ordenado: " + Arrays.toString(numeros2));

   int indice = Arrays.binarySearch(numeros2, 9);
   System.out.println("Índice del 9: " + indice);
 }
}
```

```
Array ordenado: [-4, 1, 2, 3, 5, 9, 10]
Índice del 9: 5
```

7. Estructuras de datos

Arrays

Cuando los arrays tienen una única dimensión se les llama, en ocasiones, vectores. Cuando tienen más de una dimensión (especialmente cuando tienen 2), se les llama matrices.

```
import java.util.Arrays;

public class EjemploArray2 {
 public static void main(String[] args) {
   int[][] numeros = {{1,2,5,0},{5, 2, 9, 1}};
   int[][][] numeros2 = new int[2][4][3];
   System.out.println(numeros2.length);
   System.out.println(numeros2[0].length);
   System.out.println(numeros2[0][0].length);
   for (int i=0;i<numeros2.length;i++){
     for (int j=0;j<numeros2[i].length;j++){
       for (int k=0;k<numeros2[i][j].length;k++){
         numeros2[i][j][k] = (i+1)*(j+1)*(k+1);
       }
     }
   }
   System.out.println(Arrays.deepToString(numeros));
   System.out.println(Arrays.deepToString(numeros2));
 }
}
```

7. Estructuras de datos

Listas

En Java, las listas se definen mediante la interfaz **List** (ya veremos qué es una interfaz), que permite trabajar con colecciones dinámicas. Las implementaciones más comunes son con las clases **ArrayList** y **LinkedList**.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class EjemploListas {
 public static void main(String[] args) {
   List<String> lista = new ArrayList<>();

   lista.add("A");
   lista.add("B");
   lista.add("C");
   System.out.println("Lista: " + lista);
   List<String> lista2 = new LinkedList<>(lista);

   lista.remove("B");
   System.out.println("Lista después de eliminar: " + lista);
   System.out.println("Lista2: " + lista2);
 }
}
```

7. Estructuras de datos

Conjuntos

En Java, los conjuntos se definen mediante la interfaz **Set**. Las implementaciones más comunes son con las clases **HashSet** (sin orden), **LinkedHashSet** (mantiene orden de inserción), **TreeSet** (ordena los elementos).

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class EjemploConjuntos {
 public static void main(String[] args) {
   Set<Integer> numeros = new HashSet<Integer>();
   numeros.add(10);
   numeros.add(20);
   numeros.add(10);
   numeros.add(5);
   Set<Integer> numeros2 = new TreeSet<Integer>(numeros);
   System.out.println("Conjunto: " + numeros);
   numeros2.add(21);
   numeros2.add(-3);
   System.out.println("Conjunto: " + numeros2);
 }
}
```

7. Estructuras de datos

Mapas

En Java, los mapas relacionan claves únicas con valores (como los diccionarios de Python). Se definen mediante la interfaz **Map** y las implementaciones más comunes son con las clases **HashMap** (sin orden garantizado), **LinkedHashMap** (mantiene orden de inserción), **TreeMap** (ordena los elementos en base a las claves).

```
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class EjemploMapas {
 public static void main(String[] args) {
   Map<String, Integer> edades =new HashMap<>();
   edades.put("Silvia", 30);
   edades.put("Antonio", 25);
   edades.put("Tina", 19);
   edades.put("Lope", 21);
   Map<String, Integer> edades2 =new TreeMap<>(edades);

   System.out.println("Edad de Silvia: " + edades.get("Silvia"));
   System.out.println("Edad de María: " + edades.get("María"));
   System.out.println("Edades: "+edades);
   System.out.println("Edades2: "+edades2);
 }
}
```

7. Estructuras de datos

Además de los ya vistas hay más tipos de estructuras de datos:

Pilas: Conjuntos de datos en los que el último que se introduce es el primero que se saca (LIFO). Tradicionalmente se utilizaba la interfaz **Stack**, pero a día de hoy suele utilizarse **Deque** (que soporta operaciones eficientes para pilas y colas). Deque suele implementarse con **ArrayDeque** o **LinkedList**.

Colas: Conjuntos de datos en los que el primero que se introduce es el primero que se saca (FIFO). Se suele usar **Queue** como interfaz y puede ser implementada por **LinkedList**, **ArrayDeque** o **PriorityQueue** (colas con prioridad).

7. Estructuras de datos

La clase Collections

Esta clase proporciona métodos estáticos para operar sobre datos de conjuntos, listas o mapas.

Operaciones más comunes:

- **Ordenación**: Collections.sort(List<T>)
- **Búsqueda**: Collections.binarySearch(List<T>, elemento)
- **Invertir**: Collections.reverse(List<T>)
- **Máximo y mínimo**: Collections.max(Collection<T>), Collections.min(Collection<T>)

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class EjemploCollections {
 public static void main(String[] args) {
   List<String> nombres = new ArrayList<>();
   nombres.add("Carlos");
   nombres.add("Aarón");
   nombres.add("Berta");

   Collections.sort(nombres);
   System.out.println("Nombres ordenados: " + nombres);

   Collections.reverse(nombres);
   System.out.println("Nombres invertidos: " + nombres);

   System.out.println("Primer nombre (alfabéticamente): " + Collections.min(nombres));
 }
}
```

8. Iteradores

Un **iterador** es un objeto que permite recorrer una colección en Java de manera secuencial, sin necesidad de conocer la estructura interna de la colección.

Mediante su uso se simplifica el recorrido de colecciones, asegurando que no se salten elementos y que se pueda manipular su contenido durante el recorrido.

Iterable

Es una interfaz que define el método **iterator()**.
Todas las colecciones de Java ArrayList, HashSet, LinkedList, etc.) implementan esta interfaz.

Iterator

Es una interfaz que permite recorrer una colección.
Proporciona los métodos clave:

- boolean hasNext(): indica si quedan elementos por recorrer.
- E next(): devuelve el siguiente elemento de la colección.
- void remove(): Elimina el último Elemento retornado.

8. Iteradores

Ejemplo de uso de Iterator:

```
public class EjemploIterador {
 public static void main(String[] args) {
   // Crear una colección (ArrayList)
   ArrayList<String> lista = new ArrayList<>();
   lista.add("Cerveza");
   lista.add("Ron-cola");
   lista.add("Gin-tonic");

   // Obtener el iterador de la colección
   Iterator<String> iterador = lista.iterator();

   // Recorrer la colección con el iterador
   while (iterador.hasNext()) {
     String elemento = iterador.next();
     System.out.println("Elemento: " + elemento);

     // Ejemplo de eliminar un elemento
     if (elemento.equals("Ron-cola")) {
       iterador.remove();
     }
   }
   System.out.println("Lista después de usar el iterador: " + lista);
 }
}
```

8. Iteradores

Iteradores avanzados: ListIterator

Si se trabaja con listas, se puede utilizar la clase **ListIterator**.

Además de los métodos que proporciona Iterator, ListIterator (que hereda de Iterator) proporciona:

- boolean hasPrevious(): comprueba si hay un elemento anterior
- E previous(): devuelve el elemento anterior.
- void add(E): añade un elemento a la lista en la posición actual.