INSTITUTO UNIVERSITARIO DE TECNOLOGÍA
"ANTONIO JOSÉ DE SUCRE"
ANACO ESTADO ANZOÁTEGUI.

Programación III

Guía de Estudio

Versión: 2.6

Profesor: Ing. Jesús López

Índice general

1. Introducción a la programación orientada a objetos. 6
- 1.1. Conceptos Básicos . . . 6
- 1.2. ¿Qué es la Programación Orientada a Objetos (POO)?. . . . 6
  - 1.2.1. Ventajas de la POO con respecto a la Programación tradicional. . . . 6
- 1.3. Conceptos de la programación orientada a objetos . . . 7
- 1.4. Lenguaje unificado de modelado (UML) . . . 7
  - 1.4.1. Leyendo el diagrama UML de clases . . . 7

2. Introducción y programación en Java. 9
- 2.1. ¿Qué es Java? . . . 9
- 2.2. Principales características de Java . . . 9
- 2.3. ¿Qué es el kit de desarrollo de Java (JDK)? . . . 10
  - 2.3.1. Componentes del JDK . . . 10
  - 2.3.2. ¿Cómo funciona el JDK? . . . 10
- 2.4. Estructura básica de un programa en Java . . . 10
  - 2.4.1. El método *public static void main* . . . 11
- 2.5. Tipos de datos en Java . . . 11
  - 2.5.1. Datos primitivos . . . 11
  - 2.5.2. Datos de referencia . . . 12
- 2.6. ¿Qué son las variables en Java? . . . 12
  - 2.6.1. Tipos de variables en Java . . . 12
- 2.7. La API de Java . . . 13
  - 2.7.1. La clase BigInteger . . . 13
  - 2.7.2. La clase BigDecimal . . . 14
  - 2.7.3. La clase Scanner . . . 14
- 2.8. Operadores en java . . . 14
  - 2.8.1. Tipos Operadores en java . . . 14
  - 2.8.2. Precedencia de operadores . . . 17
- 2.9. Ejercicios . . . 17

3. Diagramas de flujo, estructuras de control y clases en Java. 20
- 3.1. Diagramas de flujo . . . 20
- 3.2. Tipos de estructuras de control . . . 20
  - 3.2.1. Estructuras secuenciales . . . 20
  - 3.2.2. Estructuras condicionales . . . 20
  - 3.2.3. Estructuras iterativas . . . 22
- 3.3. Constructores en java . . . 24
  - 3.3.1. Constructor implícito y Constructor explícito. . . . 25
  - 3.3.2. Constructor parametrizado. . . . 25
- 3.4. Modificadores de acceso dentro de una clase. . . . 26
  - 3.4.1. Setters y Getters. . . . 26
- 3.5. Métodos estáticos y variables de clase. . . . 27

3.6. La palabra reservada `super` . . . 27
3.7. Implementación de diagramas UML de clases usando constructores, constantes, variables de clase y métodos estáticos. . . . 28
3.8. Ejercicios . . . 29

4. Clases abstractas, manejo de errores y paquetes en Java. 30
4.1. Clases abstractas en Java . . . 30
4.2. Manejo de errores en java . . . 30
4.3. Paquetes (package) en Java . . . 32
4.4. Archivo JAR en java . . . 32

5. Estructuras de datos, operaciones con archivos y librerías en Java. 33
5.1. Estructuras de datos . . . 33
5.2. Operaciones con archivos . . . 33

Índice de tablas

2.1. Tabla de precedencia de operadores 19

3.1. Tabla de símbolos del diagrama de flujo 20
3.2. Tabla de modificadores de acceso 26

Lista de Códigos

2.1. Estructura básica de un programa en Java . . . 10
2.2. Variables en Java . . . 12
2.3. Tipos de variables en Java . . . 12
2.4. La clase BigInteger . . . 13
2.5. La clase BigDecimal . . . 14
2.6. La clase Scanner . . . 14
2.7. Operador de concatenación . . . 15
2.8. Operadores aritméticos . . . 15
2.9. Operador de conversión de tipo(casting) . . . 15
2.10. Operadores de comparación . . . 16
2.11. Operadores de igualdad . . . 16
2.12. Operadores lógicos . . . 16
2.13. Implementación del diagrama UML . . . 18
3.1. Estructura secuencial . . . 21
3.2. Estructura condicional if . . . 21
3.3. Estructura condicional if-else . . . 22
3.4. Estructura condicional if-elseif-else . . . 22
3.5. Estructura iterativa for . . . 23
3.6. Estructura iterativa for con break y continue . . . 23
3.7. Estructura iterativa while . . . 23
3.8. Estructura iterativa do-while . . . 24
3.9. Constructores en java . . . 25
3.10. Constructores parametrizados en java . . . 25
3.11. Métodos setters y getters en java . . . 26
3.12. Miembros estáticos en java . . . 27
3.13. Uso de la palabra reservada `super` en java . . . 27
3.14. Implementación del diagrama UML . . . 28
4.1. Clases abstractas en java . . . 30
4.2. Manejo de errores en Java . . . 31
4.3. Animal.java . . . 32
4.4. Gato.java . . . 32
5.1. Estructura secuencial . . . 33

Índice de figuras

1.1. Diagrama UML de clases. 8

2.1. Compilación del código fuente de Java a bytecode. 9
2.2. Diagrama UML de clases. 18

3.1. Diagrama UML de clases. 28
3.2. Diagrama UML de clases ejercicio 2. 29

Unidad 1

Introducción a la programación orientada a objetos.

1.1. Conceptos Básicos

- **¿Qué es programar?**

  Programar en un contexto informático es dar instrucciones a un computador para que las ejecute, con el propósito de resolver un problema o realizar alguna tarea.

- **¿Qué es un paradigma de programación?**

  Es un conjunto de principios y directrices que define un enfoque particular para diseñar, estructurar y escribir código. Cada paradigma impone una forma única de pensar sobre cómo debería desarrollarse el software y cómo interactúan sus componentes.

- **Tipos de paradigmas de programación**

  - **Programación imperativa:** Este es uno de los paradigmas de programación más antiguos y fundamentales. En la programación imperativa, se describen detalladamente los pasos que debe seguir el programa para alcanzar un estado deseado. Los lenguajes de programación como C y Java son ejemplos clásicos de este paradigma.
  - **Programación declarativa:** A diferencia de la programación imperativa, la programación declarativa se centra en describir el resultado deseado sin especificar los pasos detallados para llegar allí. SQL es un ejemplo de un lenguaje declarativo

1.2. ¿Qué es la Programación Orientada a Objetos (POO)?.

Es un paradigma de programación imperativa, en la que las instrucciones y los datos se encapsulan en entidades llamadas clases, de las que luego se crean los objetos.

1.2.1. Ventajas de la POO con respecto a la Programación tradicional.

- Modularidad para facilitar la resolución de problemas.
- Reutilización de código heredado.
- Flexibilidad a través del polimorfismo.
- Protección de la información a través de la encapsulación.

1.3. Conceptos de la programación orientada a objetos

- **Clase:** Una clase es un patrón para construir objetos. Por tanto, un objeto es una variable perteneciente a una clase determinada. Es importante distinguir entre objetos y clases: la clase es simplemente una declaración, no tiene asociado ningún objeto. Y todo objeto debe pertenecer a una clase.
- **Objeto:** Un objeto es una unidad que engloba dentro de sí un conjunto de datos y las funciones necesarias para el tratamiento de esos datos. Un objeto se caracteriza por:
  - **Su identidad:** cada objeto es único y diferente del resto. Internamente se le asigna un ID para diferenciarlo de otros objetos, aunque pertenezcan a la misma clase y tengan todos sus valores internos con el mismo valor.
  - **Su estado:** el estado de un objeto viene dado por el valor de sus atributos o variables internas.
  - **Su comportamiento:** el comportamiento de un objeto se define mediante los métodos o fragmentos de código que operan con los atributos internos del objeto e interactúan, si es necesario, con otros objetos.
- **Atributos:** Los atributos son los datos incluidos en un objeto. Son como las variables en los lenguajes de programación clásicos, pero están encapsuladas dentro de un objeto y, salvo que se indique lo contrario, son invisibles desde el exterior.
- **Métodos:** Se llaman métodos a las funciones que pertenecen a un objeto. Es decir: son fragmentos de código con un nombre que permite invocarlos y ejecutarlos, pero están encapsulados dentro del objeto. Tienen acceso a los atributos del mismo y son la forma de operar con los atributos desde el exterior del objeto. Los métodos pueden o no retornar valores.
- **Herencia:** Es posible diseñar nuevas clases basándose en clases ya existentes. Esto se llama herencia. Cuando una clase hereda de otra, toma todos los atributos y todos los métodos de su clase "madre", y puede añadir los suyos propios. A veces, algunos de los métodos o datos heredados no son útiles, por lo que pueden ser enmascarados, redefinidos o simplemente eliminados en la nueva clase.
- **Polimorfismo:** Este "palabro" se refiere a la posibilidad de crear varias versiones del mismo método, de forma que se comporte de maneras diferentes dependiendo del estado del objeto o de los parámetros de entrada.
- **Encapsulamiento:** Se denomina encapsulamiento al hecho de que cada objeto se comporte de modo autónomo, de manera que lo que pase en su interior sea invisible para el resto de objetos. Cada objeto sólo responde a ciertos mensajes (llamadas a sus métodos) y proporciona determinadas salidas. El encapsulamiento se logra a través de los distintos modificadores de acceso, siendo estos : *public*, *protected* y *private* y *default*(omitido), los cuales son aplicables principalmente a los métodos y atributos de una clase.

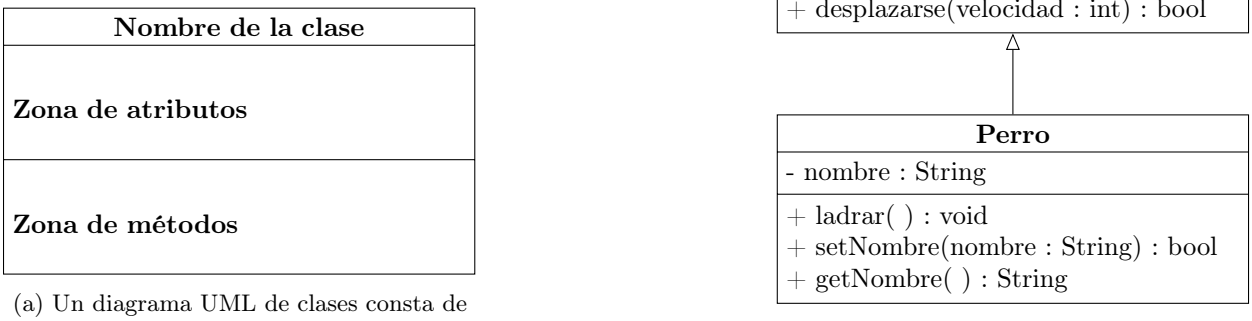
1.4. Lenguaje unificado de modelado (UML)

Es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad, es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

1.4.1. Leyendo el diagrama UML de clases

El digrama UML de clases de la figura 1.1 nos muestra 2 clases, en donde la clase *Perro* hereda los atributos y métodos de la clase *Animal*. La herencia implica una relación de **`‘‘Es un’’`** en este caso un Perro **es un** Animal.

La herencia se indica con una flecha que apunta desde la clase hija (en este caso la clase Perro) hacia la clase Madre (la clase Animal), es lo mismo que decir: la clase *Perro* es una **subclase** de la clase *Animal*. La clase *Animal* sería una clase **base** ya que no hereda de nadie y otras podrían heredar de ella.



(a) Un diagrama UML de clases consta de tres partes, el nombre de la clase,la zona de atributos y la zona de los métodos

(b) Diagrama UML de clases representando una herencia

Figura 1.1: Diagrama UML de clases.

Niveles de acceso:

- (+) **Acceso público:** Se pueden acceder desde cualquier parte del programa.
- (#) **Acceso protegido:** Solo son accesibles cuando se usa herencia o clases de un mismo paquete (java).
- (-) **Acceso privado:** Solo son accesibles dentro de la clase donde son declarados.

Unidad 2

Introducción y programación en Java.

2.1. ¿Qué es Java?

Java es un lenguaje de programación de propósito general, multiplataforma y orientado a objetos.

2.2. Principales características de Java

- **Orientado a objetos:**

  Java hace uso del paradigma de programación orientado a objetos.

- **Independencia de la plataforma:**

  Significa que programas escritos en el lenguaje Java pueden ejecutarse igualmente en cualquier tipo de hardware.

  Para ello, se compila el código fuente escrito en lenguaje Java, para generar un código conocido como "bytecode" (específicamente Java bytecode), instrucciones máquina simplificadas específicas de la plataforma Java. Esta pieza está "a medio camino" entre el código fuente y el código máquina que entiende el dispositivo destino. El bytecode es ejecutado entonces en la máquina virtual (JVM), un programa escrito en código nativo de la plataforma destino (que es el que entiende su hardware), que interpreta y ejecuta el código. Además, se suministran bibliotecas adicionales para acceder a las características de cada dispositivo.

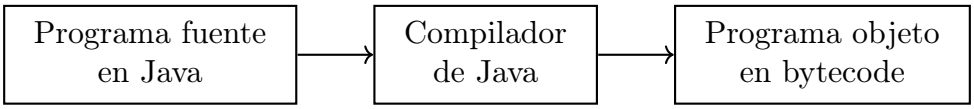


Figura 2.1: Compilación del código fuente de Java a bytecode.

- **El recolector de basura:**

  En Java el problema de fugas de memoria se evita en gran medida gracias a la recolección de basura (o automatic garbage collector). El programador determina cuándo se crean los objetos, y el entorno, en tiempo de ejecución de Java (Java runtime), es el responsable de gestionar el ciclo de vida de los objetos.

  Aun así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios. Pero en definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos y mayor seguridad.

2.3. ¿Qué es el kit de desarrollo de Java (JDK)?

El Java Development Kit (JDK) es un kit de desarrollo de software que incluye todas las herramientas necesarias para crear aplicaciones Java. El JDK incluye un compilador, un depurador y otras utilidades importantes que facilitan a los desarrolladores escribir, probar e implantar aplicaciones Java.

2.3.1. Componentes del JDK

El JDK está formado por varios componentes, cada uno de los cuales desempeña un papel fundamental en el desarrollo de aplicaciones Java. Estos son algunos de los componentes clave del JDK:

- **Compilador Java:**

  El compilador Java es una herramienta que convierte el código fuente Java en bytecode, que puede ser ejecutado por la JVM. El compilador comprueba la sintaxis del código y genera un mensaje de error si el código contiene algún error de sintaxis.

- **Depurador de Java:**

  El depurador de Java es una herramienta que permite a los desarrolladores depurar su código Java. Permite a los desarrolladores recorrer su código línea por línea, establecer puntos de interrupción e inspeccionar variables para encontrar errores en su código.

- **Máquina virtual Java (JVM):**

  La JVM es una máquina virtual que ejecuta el bytecode de Java. Es responsable de traducir el bytecode en código máquina que puede ser ejecutado por el hardware subyacente. La JVM está disponible para los principales sistemas operativos y es lo que hace que Java sea un lenguaje independiente de la plataforma.

- **Entorno de ejecución de Java (JRE):**

  El JRE es un subconjunto del JDK que incluye todo lo necesario para ejecutar aplicaciones Java. El JRE incluye la JVM y la biblioteca de clases Java, pero no incluye el compilador Java ni otras herramientas de desarrollo.

2.3.2. ¿Cómo funciona el JDK?

El JDK se utiliza para crear aplicaciones Java proporcionando a los desarrolladores todas las herramientas que necesitan para escribir, probar y desplegar su código. Los desarrolladores escriben su código utilizando un editor de texto o un entorno de desarrollo integrado (IDE) y, a continuación, utilizan el compilador de Java para compilar el código en bytecode.

Una vez compilado el código, puede ser ejecutado por la JVM, que traduce el bytecode a código máquina que puede ser ejecutado por el hardware subyacente. El depurador de Java puede utilizarse para encontrar y corregir errores en el código, y la biblioteca de clases Java proporciona a los desarrolladores una amplia gama de API que pueden utilizar para crear sus aplicaciones.

2.4. Estructura básica de un programa en Java

El programa más simple que se puede escribir en Java es aquel que solo tenga una clase y un solo método dentro de esa clase. En el código 2.1 podemos observar que hay una clase **pública** llamada `HolaMundo` que contiene un solo método **público** llamado `main`.

Código 2.1: Estructura básica de un programa en Java

```
1 public class HolaMundo {
2
3     public static void main(String[] args) {
4         // Este programa solo muestra un saludo por la pantalla
5         System.out.println("Hola, mundo!");
```

```
6       }
7   }
```

2.4.1. El método *public static void main*

Es el método que es llamado para iniciar el programa, debe encontrarse dentro de una clase *pública* (`public`), la cual debe estar contenida en un archivo que tenga el mismo nombre de la clase.

Compilador online para Java : https://www.codechef.com/java-online-compiler

2.5. Tipos de datos en Java

Los tipos de datos en Java se dividen en dos categorías principales: tipos de datos primitivos y tipos de datos de referencia.

2.5.1. Datos primitivos

Los datos primitivos son tipos de datos básicos que representan valores simples y se almacenan directamente en la memoria.

Algunos ejemplos de datos primitivos en Java incluyen **int** para números enteros, **double** para números decimales y **boolean** para valores lógicos verdaderos o falsos, entre otros.

La lista de datos primitivos pueden ser:

- **byte:**

  Representa un tipo de dato de 8 bits con signo. Puede almacenar valores numéricos en el rango de -128 a 127, incluyendo ambos extremos. Es útil para ahorrar memoria cuando se necesita almacenar valores pequeños.

- **short:**

  Este tipo de dato utiliza 16 bits con signo y puede almacenar valores numéricos en el rango de -32,768 a 32,767.

- **int:**

  Es un tipo de dato de 32 bits con signo utilizado para almacenar valores numéricos. Su rango va desde -2,147,483,648 hasta 2,147,483,647.

- **long:**

  Este tipo de dato utiliza 64 bits con signo y puede almacenar valores numéricos en el rango de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807. Se utiliza cuando se necesitan números enteros muy grandes.

- **float:**

  Es un tipo de dato diseñado para almacenar números en coma flotante con precisión simple de 32 bits. Se utiliza cuando se requieren números decimales con un grado de precisión adecuado para muchas aplicaciones.

- **double:**

  Este tipo de dato almacena números en coma flotante con doble precisión de 64 bits, lo que proporciona una mayor precisión que float. Se usa en aplicaciones que requieren una alta precisión en cálculos numéricos.

- **boolean:**

  Sirve para definir tipos de datos booleanos que pueden tener solo dos valores: true o false. Aunque ocupa solo 1 bit de información, generalmente se almacena en un byte completo por razones de eficiencia.

- **char:**

  Es un tipo de datos que representa un carácter Unicode sencillo de 16 bits. Se utiliza para almacenar caracteres individuales, como letras o símbolos en diferentes lenguajes y conjuntos de caracteres.

2.5.2. Datos de referencia

Los datos de referencia son tipos de datos más complejos que hacen referencia a objetos almacenados en memoria. Estos objetos pueden ser **instancias de clases personalizadas** o clases predefinidas en Java, como `String, BigInt, BigDecimal...`

Los datos de referencia no almacenan directamente el valor, sino una referencia a la ubicación en memoria donde se encuentra el objeto.

2.6. ¿Qué son las variables en Java?

Las variables en Java son contenedores que se utilizan para almacenar y manipular datos en un programa.

Una variable **tiene un tipo de dato** que define qué tipo de valor puede contener y un nombre que se utiliza para hacer referencia a ella en el código, los nombres de las variables no pueden contener espacios en blanco. Se llaman variables porque pueden cambiar su valor por otro que sea del mismo **tipo de dato**, ver código 2.2.

Código 2.2: Variables en Java

```
1  public class VariableYTiposDeDatos {
2      public static void main(String[] args) {
3          byte mi_variable_byte = 120;
4          mi_variable_byte = -45; //asignación de un nuevo valor
5
6          short mi_variable_short = 16000;
7          mi_variable_short = -25263; //asignación de un nuevo valor
8
9          int mi_variable_int = 2120365522;
10         long mi_variable_long = 92233720368547758O7L;
11         float mi_variable_float = 523.265f;
12         double mi_variable_double = 52132448525251412524741425254747658585255 4.252565;
13         boolean mi_variable_boolean = true;
14         char mi_variable_char = 'D';
15         String mi_varible_string = "hola a todos";
16     }
17 }
```

2.6.1. Tipos de variables en Java

En Java, las variables se dividen en tres tipos principales: variables locales, variables de instancia y variables de clase. Ver código 2.3.

- **Las variables locales** se declaran en un método y solo son accesibles dentro de ese método.
- **Las variables de instancia** pertenecen a una instancia específica de una clase y se declaran dentro de la clase pero fuera de cualquier método.
- **Las variables de clase** son compartidas por todas las instancias de una clase y se declaran utilizando la palabra clave static.

Código 2.3: Tipos de variables en Java

```
1  public class TiposDeVariables {
2
3      public int mi_variable; //variable de instancia
```

```
4       static public String texto_1 = "hola"; //variable de clase
5
6       public static void main(String[] args) {
7
8           //Declaración de una variable local(numero) de tipo entero
9           //a la cual se le asigna el valor 1.
10          int numero = 1;
11
12          //Declaracion de una variable local(texto) de tipo String
13          //a la cual se le asigna el valor "hola".
14          String texto_2 = "chao";
15
16          //Declaración de una variable local(condicion_1) de tipo boolean.
17          boolean condicion_1;
18      }
19  }
```

2.7. La API de Java

La API Java es una interfaz de programación de aplicaciones (API, por sus siglas del inglés: Application Programming Interface) provista por los creadores del lenguaje de programación Java, que da a los programadores los medios para desarrollar aplicaciones Java.

Como el lenguaje Java es un lenguaje orientado a objetos, la API de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. Por lo general estas clases se agrupan en colecciones llamadas paquetes.

Ejemplos de esto son las clases `BigDecimal` y `BigInteger` que se encuentran en el paquete `java.math`.

Documentación online de la API de Java (JavaOpenJDK 21) : https://devdocs.io/openjdk~21/

2.7.1. La clase BigInteger

La clase BigInteger en Java es una clase especializada para manejar números enteros de gran tamaño, que no se ajustan a los tipos de enteros primitivos de Java, como int, long, byte, short, etc. Esta clase se utiliza cuando se necesitan operaciones que involucren cálculos de números enteros muy grandes, cuyo resultado puede superar el rango de los tipos de enteros primitivos.

La clase BigInteger se encuentra en el paquete `java.math` y se puede importar en un proyecto Java utilizando `import java.math.BigInteger;`

Código 2.4: Ejemplo de uso de la clase BigInteger

```
1   import java.math.BigInteger;
2
3   public class Test_BI {
4       public static void main(String[] args) {
5           // Realizar operaciones con objetos BigInteger
6           BigInteger bi1 = new BigInteger("123456789");
7           BigInteger bi2 = new BigInteger("987654321");
8           System.out.println("Operación de adición:" + bi2.add(bi1));
9           System.out.println("Operación de resta:" + bi2.subtract(bi1));
10          System.out.println("Operación de multiplicación:" + bi2.multiply(bi1));
11          System.out.println("Operación de división:" + bi2.divide(bi1));
12
13          // Obtener el máximo y mínimo de dos objetos BigInteger
14          System.out.println("Número máximo:" + bi2.max(bi1));
15          System.out.println("Número mínimo:" + bi2.min(bi1));
16      }
17  }
```

2.7.2. La clase BigDecimal

La clase BigDecimal en Java se utiliza para realizar operaciones precisas con números decimales, especialmente con valores de dinero o cálculos que requieren exactitud.

Código 2.5: Ejemplo de uso de la clase BigDecimal

```java
import java.math.BigDecimal;

public class Test_BD {
    public static void main(String[] args) {
        //limitaciones de los tipos float y double
        double d_1 = 0.1;
        double d_2 = 0.2;
        double suma_1 = d_1 + d_2;
        System.out.println("El resultado usando float/double: "+suma_1);

        // Suma
        BigDecimal bd_1 = new BigDecimal("0.1");
        BigDecimal bd_2 = new BigDecimal("0.2");
        BigDecimal suma_2 = bd_1.add(bd_2);
        System.out.println("El resultado usando BigDecimal: "+suma_2); // Output: 0.3
    }
}
```

2.7.3. La clase Scanner

La clase Scanner en Java se utiliza para leer datos de entrada, como texto introducido por el usuario o desde un archivo de texto. Esta clase es parte de la librería **java.util** y permite capturar diferentes tipos de datos primitivos, como enteros, dobles y cadenas de texto, de manera sencilla y eficiente.

Para usar la clase Scanner, primero se debe importar el paquete **java.util.Scanner** al inicio del programa. Luego, se crea una instancia de la clase Scanner, generalmente para leer desde el teclado (System.in) o desde una cadena de texto.

Código 2.6: Ejemplo de uso de la clase Scanner

```java
import java.util.Scanner;

public class Test_Scanner {

    public static void main(String[] args) {
        // se crea un objeto Scanner
        Scanner input = new Scanner(System.in);

        // se esperan datos del usuario.
        String linea = input.nextLine();

        // se imprimen los datos que el usuario suministró.
        System.out.println(linea);

        // se cierra el objeto Scanner.
        input.close();
    }
}
```

2.8. Operadores en java

2.8.1. Tipos Operadores en java

- **operadores de asignación:**

Los operadores de asignación son fundamentales en Java ya que permiten asignar valores a variables. A través de estos operadores, se puede almacenar información en variables para su posterior uso en el programa.

El operador más comúnmente utilizado es el signo igual (=), que asigna el valor de la derecha a la variable de la izquierda. Por ejemplo, `int x = 10;` asigna el valor 10 a la variable `x`. Además del operador de asignación simple, Java proporciona operadores de asignación compuesta, como `+=, -=, *=, /=,` entre otros.

- **operadores de concatenación:**

  En el contexto de cadenas de texto **(string)**, Java proporciona el operador + para la concatenación de cadenas. Este operador se utiliza para unir dos o más cadenas de texto en una sola cadena.

Código 2.7: Operador de concatenación en Java

```
1 //Concatenación
2 String nombre = "pepe";
3 System.out.println("Hola mi nombre es " + nombre);
```

- **operadores aritméticos:**

  Los operadores aritméticos en Java son símbolos especiales que se utilizan para realizar operaciones matemáticas en variables numéricas. algunos de estos operadores son: + (suma), - (resta), * (multiplicación), / (división), % (módulo).

Código 2.8: Operadores aritméticos en Java

```
1  //una suma
2  int a = 2;
3  int b = 3;
4  int suma = a + b;
5  System.out.println("La suma de " + a + " y " + b + " es: " + suma);
6
7  //hallar el modulo
8  int r = 7;
9  int s = 2;
10 int modulo = r % s;
11 System.out.println("El módulo de " + r + " dividido por " + s + " es: " + modulo);
```

- **Operadores de conversión de tipo:** Cuando se trabaja con tipos de datos diferentes, es posible que necesites realizar conversiones de tipo.

  Java ofrece operadores de conversión explícita, como `(tipo)` y métodos de conversión de tipo, como `Integer.parseInt()` y `String.valueOf()`.

Código 2.9: Operador de conversión de tipo(casting) en Java

```
1 //dividir dos números
2 int m = 7;
3 int n = 2;
4 int div = m / n;
5 float div_f = m / n;
6 float div_f2 = (float)m / n; //aqui se hace la conversión de tipo
7 System.out.println("El cociente entre " + m + " y " + n + " es: " + div);
8 System.out.println("El cociente entre " + m + " y " + n + " es: " + div_f);
9 System.out.println("El cociente entre " + m + " y " + n + " es: " + div_f2);
```

- **operadores de comparación:**

  Los operadores de comparación son esenciales para evaluar condiciones en programas Java. Comparan dos valores y devuelven un resultado booleano, es decir, `true` si la comparación es verdadera y `false`

si es falsa. Algunos de los operadores de comparación más comunes son == (igual a), ! = (diferente de), < (menor que), > (mayor que), <= (menor o igual que) y >= (mayor o igual que).

Estos operadores son fundamentales para construir estructuras de control condicional, como las sentencias `if` y `switch`, que permiten que un programa tome decisiones basadas en condiciones.

Código 2.10: Operadores de comparación en Java

```java
//operadores de comparación
int n1 = 5;
int n2 = 7;
boolean igual = (n1 == n2); // Comprueba si n1 es igual a n2.
boolean noIgual = (n1 != n2); // Comprueba si n1 es diferente a n2.
boolean mayorQue = (n1 > n2); // Comprueba si n1 es mayor que n2.
boolean menorIgual = (n1 <= n2); // Comprueba si n1 es menor o igual que n2.
System.out.println(igual);
System.out.println(noIgual);
System.out.println(mayorQue);
System.out.println(menorIgual);
```

- **operadores igualdad:**

  Además de los operadores de comparación (== y !=), Java también ofrece operadores de igualdad `equals()` para comparar objetos por igualdad estructural en lugar de igualdad de referencia.

Código 2.11: Operadores de igualdad en Java

```java
//operadores de igualdad
String s1 = new String("hola!");
String s2 = new String("hola!");

System.out.println("(s1 == s2): " + (s1 == s2));
System.out.println("(s1.equals(s2)): " + (s1.equals(s2)));

String s3 = "saludos";
String s4 = "saludos";
System.out.println("s_3 == s_4: " + (s3 == s4));
```

- **operadores lógicos:**

  Los operadores lógicos son herramientas poderosas para combinar o invertir condiciones lógicas en un programa Java. Los operadores lógicos más comunes son && (AND lógico), || (OR lógico) y ! (NOT lógico).

Código 2.12: Operadores lógicos en Java

```java
// operadores lógicos
int a1 = 5;
int b1 = -1;
System.out.println(a1 > b1 && b1 < 0); //true
System.out.println(b1 > 0 || a1 > b1); //true
System.out.println(!(b1 > 0)); //true
```

- **operadores de incremento o decremento:**

  Los operadores de incremento (++) y decremento (−−) son útiles para modificar el valor de una variable en una unidad.

- **operadores ternarios:**

  Los operadores ternarios, también conocidos como operadores condicionales, son una característica concisa y poderosa de Java. Tienen la forma `condición ? valor_si_verdadero : valor_si_falso.`

- **operadores de bits:** Los operadores de bits se utilizan para realizar operaciones a nivel de bits en valores enteros. Estos operadores permiten manipular datos binarios y realizar operaciones de desplazamiento de bits.
- **operadores de instancia:**

  El operador `instanceof` se utiliza para verificar si un objeto es una instancia de una clase o interfaz específica.

  Esto es útil en programación orientada a objetos para verificar la relación de herencia.

2.8.2. Precedencia de operadores

La precedencia de operadores se refiere al orden en que se evalúan los operadores en una expresión matemática o lógica. En otras palabras, determina qué operador se evalúa primero en una expresión que contenga varios operadores de diferentes tipos.

En programación, la precedencia de operadores es importante porque puede afectar el resultado de una expresión. Por ejemplo, en la expresión $1 + 5 * 3$, la precedencia de los operadores determina que se realice la multiplicación antes de la adición, lo que da como resultado 16, en lugar de 18 si se realizara la adición antes de la multiplicación. ver la tabla 2.1.

2.9. Ejercicios

1. Elabore las siguientes actividades usando la estructura básica de un programa en Java:

   a) Calcular y mostrar en pantalla el área de un rectángulo que tiene como base 6 cm y de altura 5 cm, el resultado del área debe ser un número entero.

   Formula del área: Area = base x altura

   b) Mostrar la suma de los números 12.754 y 36.125, el resultado **no requiere** ser totalmente exacto.

   c) Mostrar la suma de los números 5.62 y 5.85, el resultado **requiere** ser totalmente exacto.

   d) Comprobar si la multiplicación de 2*6 es menor que 5*2 y mostrar el resultado.

   e) Preguntar a un usuario su nombre y luego imprimir el texto : `Hola {nombre_del_usuario}, saludos`.

   f) Preguntar al usuario por un número y luego otro número, realizar la suma de estos números y mostrar en pantalla lo siguiente: `La suma es : {suma_de_los_números}`

   g) Preguntar la altura y la base de un triángulo al usuario, calcular el área y mostrarla en pantalla (usar `float` como tipo de dato).

   Formula del área: Area = (base x altura)/2

   h) Preguntar a un usuario por un número entero y luego imprimir en pantalla si el número es par o impar.

   i) Preguntar a un usuario que edad tiene y cuanto dinero posee, la lógica del programa es que si el usuario es mayor de edad y si posee suficiente dinero para comprar la entrada, entonces podrá asistir al evento.

   Condiciones:

   - Un usuario es mayor de edad si tiene 18 años o más.
   - La entrada cuesta 50 dólares.
   - El programa finalmente deberá imprimir el siguiente mensaje: `Puedes entrar al evento?: {true|false}`

   j) Preguntar a un usuario por una palabra para luego transformarla y mostrarle la misma palabra pero en mayúsculas.

2. Implemente el siguiente diagrama UML de clases a código Java:

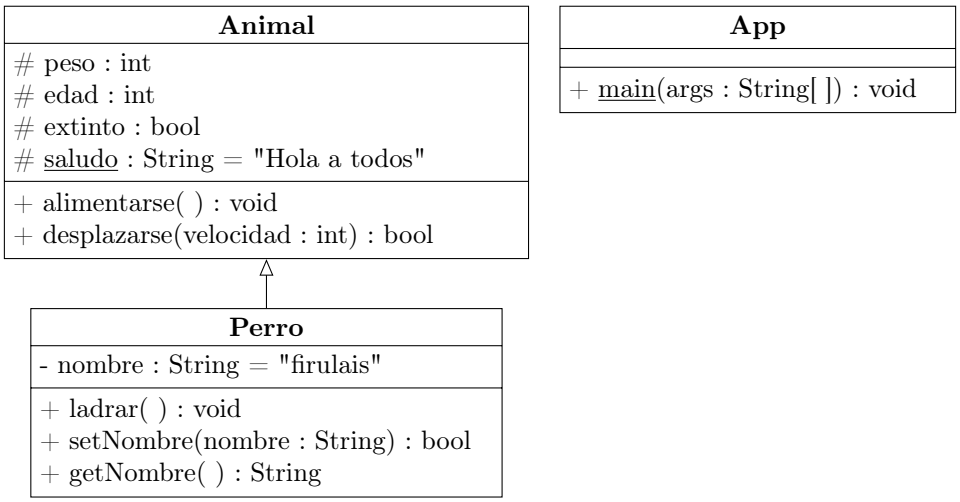


Figura 2.2: Diagrama UML de clases.

Respuesta:

Código 2.13: Implementación del diagrama UML en Java

```java
class Animal {
    protected int peso; //variable de instancia
    protected int edad;
    protected boolean extinto;
    protected static String saludo = "Hola a todos"; //Variable de clase

    public void alimentarse(){
        String mensaje = "me estoy alimentando"; //variable local
        System.out.println(mensaje);

    };
    public boolean desplazarse(int velocidad){
        return true;
    }
}

class Perro extends Animal {
    private String nombre = "firulais";

    public void ladrar(){};
    public boolean setNombre(String nombre){
        return true;
    }
    public String getNombre(){return nombre;}
}

public class App {
    public static void main(String[] args) {
        Perro firu = new Perro();
        System.out.println(firu.getNombre());
    }
}
```

Nivel	Nombre	Operador
16	Paréntesis	`()`
	Acceso a arreglo	`[]`
	Acceso a miembros	`.`
15	Post-incremento unario	`++`
	Post-decremento unario	`--`
14	Pre-incremento unario	`++`
	Pre-decremento unario	`--`
	Mas unario	`+`
	Menos unario	`-`
	Negación lógica unaria	`!`
	Negación/Complemento unario	$\sim$
13	Conversión de tipos (cast)	`()`
	Creación de objetos	new
12	Multiplicación	`*`
	División	`/`
	Módulo	`%`
11	Adición	`+`
	Sustracción	`-`
	Concatenación de strings	`+`
10	Despl. Bits Izq.	`<<`
	Despl. Bits Der. /Signo	`>>`
	Despl. Bits Der. /Cero	`>>>`
9	Menor que	`<`
	Menor o igual que	`<=`
	Mayor que	`>`
	Mayor o igual que	`>=`
	Comparación de tipos	`instanceof`
8	Igual a	`==`
	No igual a	`!=`
7	Si binario	`&`
6	O exclusivo binario	`^`
5	O inclusivo binario	`|`
4	Si lógico	`&&`
3	O lógico	`||`
2	Condicional ternario	`?:`
1	Asignación	`=`
	Suma y asignación	`+=`
	Sustracción y asignación	`-=`
	Multiplicación y asignación	`*=`
	División y asignación	`/=`
	Módulo y asignación	`%=`
	Si binario y asignación	`&=`
	O exclusivo binario y asignación	`^=`
	O inclusivo binario y asignación	`|=`
	Despl. Bits Izq. y asignación	`<<=`
	Despl. Bits Der. /Signo y asignación	`>>=`
	Despl. Bits Der. /Cero y asignación	`>>>=`
0	Flecha de expresión lambda	`->`

Tabla 2.1: Tabla de precedencia de operadores **a mayor nivel mayor precedencia**.
Todos los operadores binarios que tienen la misma precedencia (excepto los operadores de asignación) son evaluados de izquierda a derecha. Los operadores de asignación son evaluados de derecha a izquierda.

Unidad 3

Diagramas de flujo, estructuras de control y clases en Java.

3.1. Diagramas de flujo

Un diagrama de flujo, también conocido como flujograma o diagrama de actividades, es una representación gráfica y secuencial de un proceso, sistema o algoritmo que muestra los pasos, tareas o etapas de forma lógica y ordenada. Ver tabla 3.1 y figura 3.1.

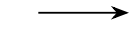
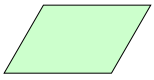
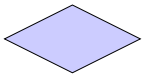
Símbolo	Nombre	Función
	Inicio/Final	Representa el inicio y el final de un proceso.
	Línea de flujo	Indica el orden de la ejecución de las operaciones. La flecha indica la siguiente instrucción.
	Entrada/Salida	Representa la lectura de datos en la entrada y la impresión de datos en la salida.
	Proceso	Representa cualquier tipo de operación.
	Decisión	Nos permite analizar una situación, con base en los valores verdadero y falso.

Tabla 3.1: Tabla de símbolos del diagrama de flujo.

3.2. Tipos de estructuras de control

3.2.1. Estructuras secuenciales

Esta es la estructura básica, ya que nos permite asegurar que una instrucción se ejecuta después de la otra siguiendo el orden en que fueron escritas, ver código 3.1.

3.2.2. Estructuras condicionales

Este tipo de estructuras de control nos sirven cuando necesitamos que se evalúe el valor de alguna variable o de alguna condición para decidir qué instrucciones ejecutar a continuación.

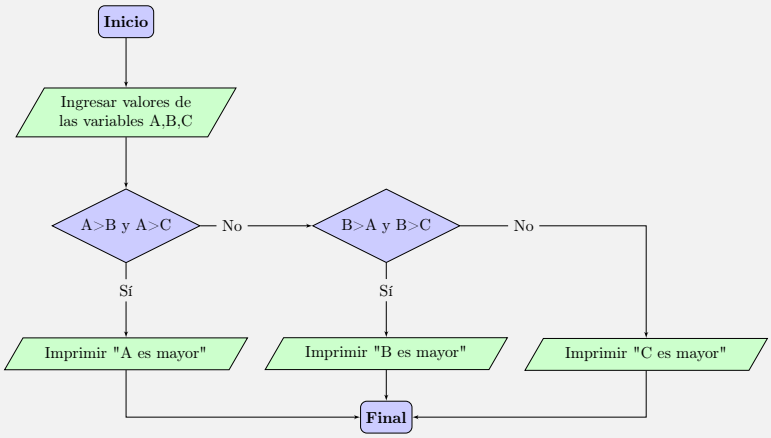


Figura 3.1: Diagrama de flujo para saber cual de las variables es mayor.

Código 3.1: Estructura de programación secuencial en Java

```
int a = 15;
int b = 70;
int suma = a + b;
System.out.println("la suma es : " + suma);
```

Estructura de control condicional `if`:

El código dentro del bloque `if` se ejecutará solo si la condición a evaluar es verdadera, ver código 3.2 y figura 3.2.

Código 3.2: Estructura de control condicional *if* en Java. **Diagrama de flujo:** 3.2

```
boolean valor_booleano = true;
if (valor_booleano) {
    System.out.println("if");
}
```

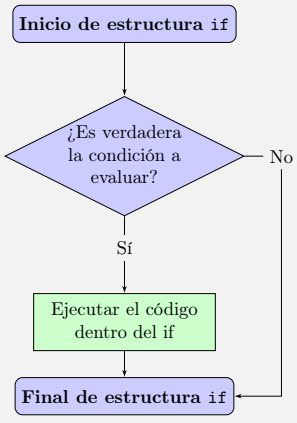


Figura 3.2: Diagrama de flujo para *if*.

Estructura de control condicional `if-else`:

El código dentro del bloque `if` se ejecuta si la condición a evaluar es verdadera, de lo contrario se ejecutará el código dentro del bloque `else`, ver código 3.3 y figura 3.3.

Código 3.3: Estructura de control condicional *if-else* en Java. **Diagrama de flujo:** 3.3

```
boolean valor_booleano2 = false;
if (valor_booleano2) {
    System.out.println("if");
} else {
    System.out.println("else");
}
```

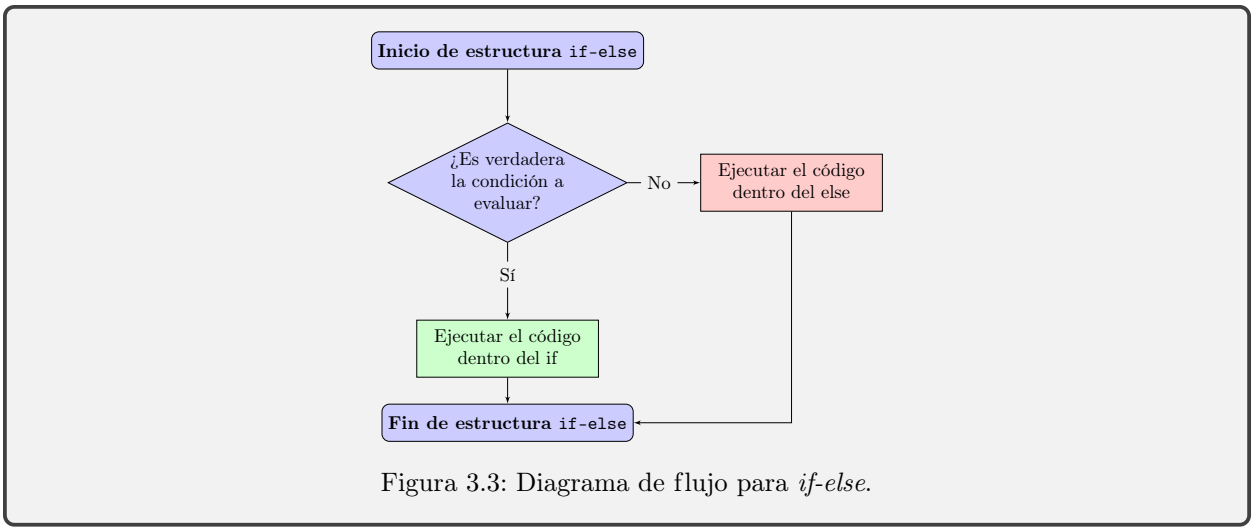


Figura 3.3: Diagrama de flujo para *if-else*.

Estructura de control condicional `if-elseif-else`:

El código dentro del bloque `if` se ejecuta si la condición a evaluar es verdadera, de lo contrario se evalúa si la condición para el bloque `else if` es verdadera, si esto es así se ejecuta el código en dicho bloque. El bloque `else` solo se ejecuta si todas las condiciones previas evaluadas son falsas, ver código 3.4 y figura 3.4.

Código 3.4: Estructura de control condicional *if-elseif-else* en Java. **Diagrama de flujo:** 3.4

```
int numero = 3;
if (numero > 5) {
    System.out.println("if (if-elseif-else)");
}else if (numero > 3) {
    System.out.println("else if");
}
 else {
    System.out.println("else (if-elseif-else)");
}
```

3.2.3. Estructuras iterativas

Este tipo de estructuras nos sirven cuando necesitamos que se ejecute un conjunto específico de instrucciones en diversas ocasiones. La cantidad de veces que se repite dicho bloque de acciones puede ser estático o puede depender del valor de alguna variable o de alguna condición.

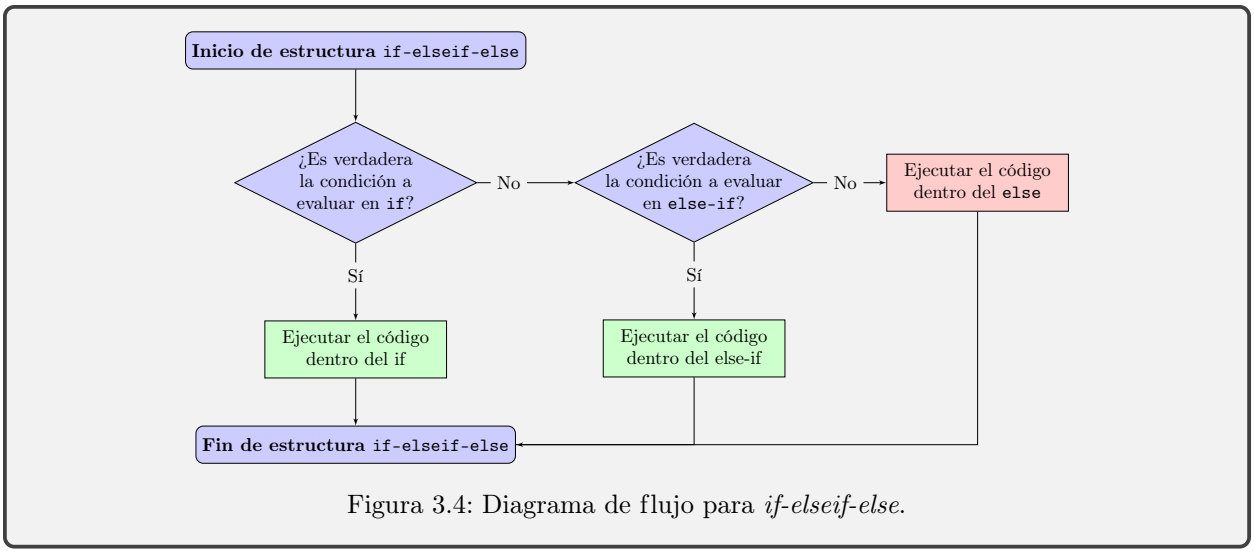


Figura 3.4: Diagrama de flujo para *if-elseif-else*.

Estructura de control iterativa `for`:

El código dentro del bucle `for` se ejecutará tantas veces como se indique en su sección de control que para el ejemplo dado es: `(i < veces)`, ver código 3.5.

Código 3.5: Estructura de control iterativa *for* en Java

```
int veces = 5;
for (int i = 0; i < veces; i++) {
    System.out.println("bucle for: " + i);
}
```

Estructura de control iterativa `for` usando `break` y `continue`:

La instrucción `break` se utiliza para salir del bucle y la instrucción `continue` se usa para saltar hacia la próxima iteración, ver código 3.6.

Código 3.6: Estructura de control iterativa *for* con break y continue en Java

```
int veces2 = 7;
for (int i = 0; i < veces2; i++) {
    if (i == 2) {
        continue;
    }
    System.out.println("usando break y continue: " + i);
    if (i == 5) {
        break;
    }
}
```

Estructura de control iterativa `while`:

El código dentro del bucle `while` se ejecutara tantas veces siempre y cuando la condición evaluada sea verdadera, ver código 3.7 y figura 3.5.

Código 3.7: Estructura de control iterativa *while* en Java. **Diagrama de flujo:** 3.5

```
int num = 0;
while (num < 3) {
    num++;
```

```
4       System.out.println("bucle while: " + num);
5   }
```

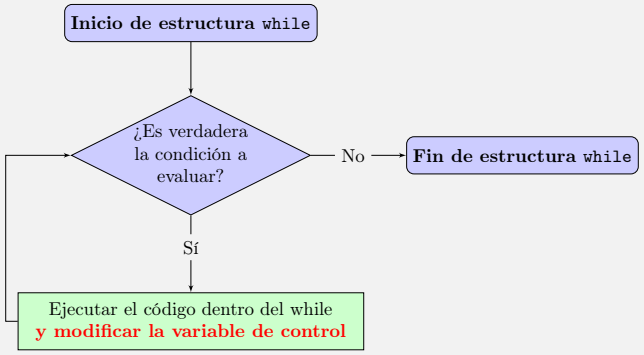


Figura 3.5: Diagrama de flujo para *while*.

Estructura de control iterativa `do-while`:

Siempre y cuando la condición en el `while` sea verdadera se ejecutará el bloque de código en el `do`, con excepción de la **primera iteración**, la cual siempre ejecutará el bloque de código en el `do`, ver código 3.8 y figura 3.6.

Código 3.8: Estructura de control iterativa *do-while* en Java. **Diagrama de flujo:** 3.6

```
1   int contador = 4;
2   do {
3       System.out.println("bucle do-while: "+ contador);
4       contador--;
5   } while (contador > 2);
```

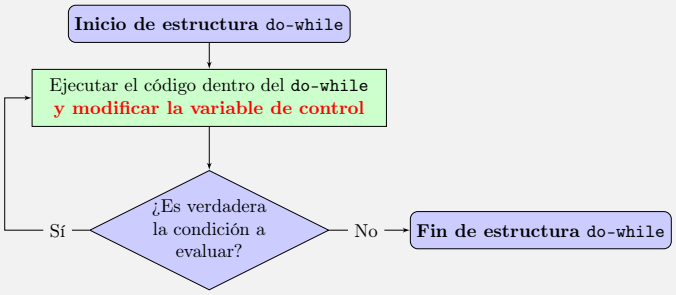


Figura 3.6: Diagrama de flujo para *do-while*.

3.3. Constructores en java

En Java, un constructor es un método especial que se llama automáticamente cuando se crea un objeto de una clase. Los constructores tienen como característica principal que se llaman igual que la propia clase, y su función es inicializar el objeto y asegurarse de que los objetos contengan valores válidos.

3.3.1. Constructor implícito y Constructor explícito.

El **constructor implícito** es aquel que Java proporciona por defecto, es decir aquel para el cual no escribimos nosotros ningún código, el **constructor explícito** es aquel que nosotros especificamos mediante un código, una vez creado cualquier constructor explícito el constructor implícito llamado también "por defecto", no estará disponible para ser usado.

Código 3.9: Constructores en java

```
class Animal {
    protected float peso;
}

class Persona {
    protected String nombre;

    // Constructor explicito:
    public Persona() {
        nombre = "Pablo";
    }
}

public class Constructores {
    public static void main(String[] args) {
        // Se crea un objeto Animal usando el constructor implícito/defecto:
        Animal animal_1 = new Animal();

        // Se crea un objeto Persona usando el constructor explicito:
        Persona persona_1 = new Persona();
    }
}
```

3.3.2. Constructor parametrizado.

Un constructor parametrizado es un tipo de constructor que acepta uno o más parámetros para inicializar los atributos de un objeto con valores específicos proporcionados durante la creación de la instancia.

Código 3.10: Constructores parametrizados en java

```
class Persona {
    private String nombre;
    private int edad;

    // Constructor explicito con parámetros:
    public Persona(String nombre, int edad) {
        this.nombre = nombre + " de Gómez";
        boolean edad_validad = setEdad(edad);
        if (!edad_validad) {
            throw new IllegalArgumentException("la edad debe estar entre 1 y 100");
        }
    }

    public boolean setEdad(int edad) {
        if (edad > 0 && edad <= 100) { this.edad = edad; return true;}
        else {return false;}
    }
}

public class ConstructorParametrizado {
    public static void main(String[] args) {
        Persona p1 = new Persona("Luisa", 6);
        Persona p2 = new Persona();//Error el constructor por defecto ya no está disponible.
    }
}
```

3.4. Modificadores de acceso dentro de una clase.

En Java, los modificadores de acceso controlan la visibilidad y el acceso a clases, métodos, atributos y constructores desde otras partes del código. Existen cuatro modificadores principales: **public**, **private**, **protected** y el acceso por defecto (también llamado **package-private**), que se aplica cuando no se especifica ningún modificador explícitamente, la tabla 3.2 muestra el nivel de encapsulamiento de los distintos modificadores de acceso.

Modificador	La propia clase	Mismo paquete	Clase hija mismo paquete	Clase hija diferente paquete	Desde cualquier clase
public	✓	✓	✓	✓	✓
protected	✓	✓	✓	✓	
sin modificador (package)	✓	✓	✓		
private	✓				

Tabla 3.2: Tabla de modificadores de acceso.
public : La menor encapsulación posible, **private** : La mayor encapsulación posible.

3.4.1. Setters y Getters.

Los métodos getter y setter son herramientas fundamentales en la programación orientada a objetos que permiten controlar el acceso y la modificación de los atributos de una clase. Un método setter se utiliza para asignar un valor a un atributo, generalmente recibiendo un parámetro y asignándolo al atributo correspondiente.

Por otro lado, un método getter sirve para obtener o retornar el valor de un atributo, devolviendo su contenido sin recibir parámetros. Estos métodos son esenciales para implementar el principio de encapsulación, que consiste en mantener los datos internos de una clase privados y permitir su acceso únicamente a través de métodos controlados, ver código 3.11

Código 3.11: Métodos setters y getters en java

```
class Persona{
    private String nombre;
    private int edad;

    public boolean setEdad(int edad) {
        if (edad >= 5 && edad <= 100) { this.edad = edad; return true;}
        else{return false;}
    }
    public int getEdad() {return edad;}

    public boolean setNombre(String nombre) {
        if (nombre.length() <= 10) {this.nombre = nombre; return true;}
        else{return false;}
    }
    public String getNombre(){return this.nombre;}
}

public class Modificadores {
    public static void main(String[] args) {
        Persona p1 = new Persona();
        p1.setEdad(15);
        System.out.println(p1.getEdad());
        p1.setNombre("Marcela");
        System.out.println(p1.getNombre());
    }
}
```

3.5. Métodos estáticos y variables de clase.

En Java, los métodos estáticos y las variables estáticas(variables de clase) están vinculados directamente a la clase y no a instancias específicas de la clase, lo que significa que existen independientemente de si se han creado objetos de la clase. Un método estático pertenece a la clase y no a un objeto, por lo que se puede invocar directamente usando el nombre de la clase sin necesidad de crear una instancia.

Un método estático solo puede acceder a otros miembros estáticos (variables y métodos) de la clase, ya que no tiene acceso a variables de instancia ni a la palabra clave `this` o `super`. Intentar acceder a un miembro no estático desde un contexto estático generará un error de compilación, ver código 3.12

Código 3.12: Miembros estáticos en java

```
class Animal {
    private static int cantidad_animales = 0;

    public Animal() {cantidad_animales++;}
    public static int getCantidad_animales() {return cantidad_animales;}
}

public class Estaticos {
    public static void main(String[] args) {
        Animal n1 = new Animal();
        Animal n2 = new Animal();
        Animal n3 = new Animal();
        //System.out.println(Animal.cantidad_animales); //Error: variable de clase no
            visible
        System.out.println(Animal.getCantidad_animales());
    }
}
```

3.6. La palabra reservada `super`

La palabra clave `super` en Java se utiliza para referirse al objeto de la clase padre inmediata, permitiendo a una subclase acceder a los constructores, métodos y campos de su superclase. Se emplea principalmente en tres contextos: para llamar al constructor de la superclase, para acceder a un método de la superclase que ha sido sobrescrito en la subclase, y para acceder a un campo de la superclase cuando está oculto por un campo del mismo nombre en la subclase, ver código 3.13.

Código 3.13: Uso de la palabra reservada `super` en java

```
class Animal {
    protected String actividades_basicas = "comer, reproducirse";
    protected float peso_kg;
    protected int edad;
    public Animal(float peso,int edad) {peso_kg = peso; this.edad = edad;}
    public String mostrar_datos() {return "Peso: "+peso_kg+"\n"+"Edad: "+edad;}
}

class Perro extends Animal{
    protected String actividades_basicas = "ladrar, jugar";
    protected String nombre;
    public Perro(float peso,int edad,String nombre){super(peso, edad);this.nombre = nombre;}
    @Override
    public String mostrar_datos(){
        String datos_animal = super.mostrar_datos();
        return datos_animal+"\nnombre: "+nombre+"\nactividades:
            "+super.actividades_basicas+", "+actividades_basicas;
    }
}

public class PalabraReservadaSuper {
    public static void main(String[] args) {
```

```
        Animal a1 = new Animal(24.3f, 25); System.out.println("DATOS ANIMAL:");
        System.out.println(a1.mostrar_datos()); Perro p1 = new Perro(15.3f, 10, "Rocky");
        System.out.println("\nDATOS PERRO:"); System.out.println(p1.mostrar_datos());
    }
}
```

3.7. Implementación de diagramas UML de clases usando constructores, constantes, variables de clase y métodos estáticos.

Se tiene en la figura 3.1 un diagrama UML genérico de una relación de herencia entre clases, se desea hallar su implementación en Java.

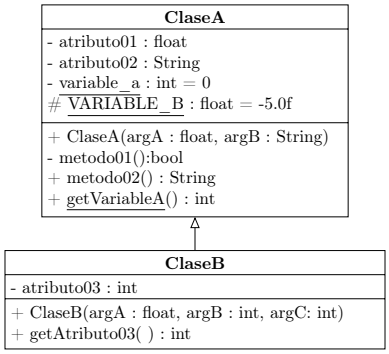


Figura 3.1: Diagrama UML de clases.

Código 3.14: Implementación del diagrama UML en Java de la figura 3.1

```
class ClaseA {
    private float atributo01;
    private String atributo02;
    private static int variable_a = 0;
    protected static final float VARIABLE_B = -5.f;
    public ClaseA(float argA, String argB){
        atributo01 = argA; atributo02 = argB; variable_a++;
    }
    private boolean metodo01(){return true;}
    public String metodo02(){if (metodo01()) {return "A";} else {return "B";}}
    public static int getVariableA(){return variable_a;}
}

class ClaseB extends ClaseA {
    private int atributo03;
    public ClaseB(float argA, String argB, int argC){super(argA, argB); atributo03 = argC;}
    public int getAtributo03(){ return atributo03;}
}

public class App {
    public static void main(String[] args) {
        ClaseB obj = new ClaseB(25, "hola", 20);
        System.out.println(ClaseB.VARIABLE_B);
        ClaseB.VARIABLE_B= 25.3f; //Error: variable de clase es constante.
        System.out.println(ClaseB.variable_a); //Error: variable de clase es privada
        System.out.println(ClaseA.getVariableA());
        System.out.println(ClaseB.getVariableA());
    }
}
```

3.8. Ejercicios

1. Implementar los siguientes diagramas de flujos en programas de Java. https://viewer.diagrams.net/?lightbox=1&highlight=0000ff&nav=1&title=guessNumber.drawio&page-id=AdtsBY_rc_XkscZplyJk#Uhttps%3A%2F%2Fdrive.google.com%2Fuc%3Fid%3D1JgZQM_DmjxdeNkUPIU6_JNwP-1DMTELd%26export%3Ddownload#%7B%22pageId%22%3A%22AEBO0bEPdG_oAkH96Bb4%22%7D

2. Implementar el diagramas UML de clases de la figura 3.2 en código Java. Detalles de la implementación:

   - El constructor de la clase `Vehiculo` debe inicializar los atributos `peso` y `potencia_motor`, con los argumentos recibidos, además cada vez que se llame al constructor de la clase `Vehículo` este deberá aumentar en uno el valor de la variable estática `cantidad_vehiculos`, por último este constructor deberá validar que el atributo `peso` se encuentre en el rango establecido por las constantes `PESO_MINIMO` y `PESO_MAXIMO` usando el método `validarPeso` y usando el valor devuelto por ese método para validar el valor del atributo `peso`, si este valor no es válido, el constructor deberá arrojar un error del tipo `IllegalArgumentException` de lo contrario se conserva el valor asignado al atributo `peso`.
   - El método `validarPeso` se encargará de retornar un valor **booleano** dependiendo si el atributo `peso` se encuentra o no en los límites válidos establecidos por las contantes `PESO_MINIMO` y `PESO_MAXIMO`.
   - El método `relacionPotenciaPeso` se encargará de hacer la división entre el atributo `peso` y el atributo `potencia_motor`: $(\frac{peso}{potencia})$ y de retornar el resultado de esa operación.
   - El método `getCantidadVehiculos` se encargará de retornar el valor de la variable `cantidad_vehiculos`.
   - El constructor de la clase `VehiculoDeportivo` deberá inicializar el atributo `capacidad_turbo` así como los otros atributos que han sido **heredados**, usando con los argumentos recibidos.
   - El método `relacionPotenciaPeso` de la clase `VehiculoDeportivo` se encargará de sumarle el valor del atributo `capacidad_turbo` al atributo `potencia_motor` para luego retornar la operación realizada por el método `relacionPotenciaPeso` de la clase `Vehiculo`.
   - En la función `main` de la clase pública deberán crearse 2 objetos de tipo `VehiculoDeportivo` y un objeto de tipo `Vehiculo` y luego responder o realizar lo siguiente:
     a) Escriba el código para imprimir la cantidad de objetos `Vehiculo` creados.
     b) Escriba el código para imprimir la relación Potencia-Peso de cualquiera de los vehículos deportivos creados.
     c) ¿Qué ocurre con el programa cuando se intenta crear un objeto `Vehiculo` o `VehiculoDeportivo` si el peso indicado no se encuentra entre los valores aceptados?
     d) ¿Tiene sentido declarar el método `validarPeso` como público?¿Por qué?

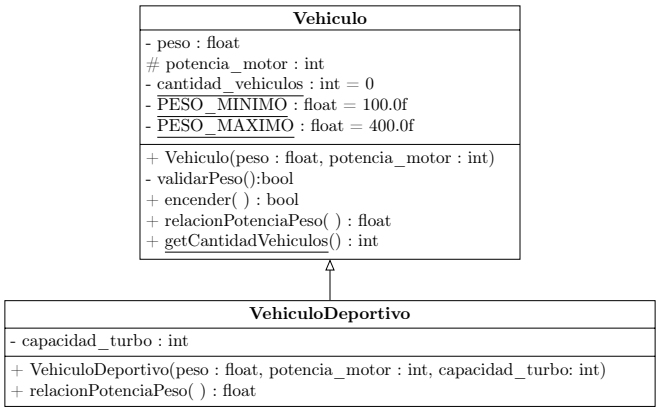


Figura 3.2: Diagrama UML de clases ejercicio 2.

Unidad 4

Clases abstractas, manejo de errores y paquetes en Java.

4.1. Clases abstractas en Java

Una clase abstracta es una clase que no se puede instanciar directamente y se utiliza como base para otras clases. Una clase abstracta puede tener métodos abstractos, que son métodos que no tienen una implementación y deben ser implementados por las clases que heredan de la clase abstracta.

Código 4.1: Clases abstractas en java

```java
abstract class FiguraGeometrica {
    protected String saludo_geometrico;
    public FiguraGeometrica(String texto) {saludo_geometrico = texto;}
    protected void saludo() {System.out.println(saludo_geometrico);}
    protected abstract double area();
}
class Triangulo extends FiguraGeometrica{
    private double base;
    private double altura;

    Triangulo(double base,double altura){
        super("hola");
        this.base = base;
        this.altura = altura;
    }

    public double area() {return (base*altura)/2;}
}
public class App {
    public static void main(String[] args) {
        //Error, no se puede instanciar una clase abstracta:
        //FiguraGeometrica f = new FiguraGeometrica("hola");
        Triangulo t1 = new Triangulo(3,5);
        System.out.println(t1.area());
        System.out.println(t1.saludo_geometrico);
    }
}
```

4.2. Manejo de errores en java

El manejo de errores en Java se basa en el uso de excepciones, que son eventos que ocurren durante la ejecución del programa y que interrumpen el flujo normal de las sentencias. El manejo de excepciones en Java también incluye el uso de bloques try-catch-finally, donde el bloque try contiene el código que puede

lanzar una excepción, el bloque catch captura y maneja la excepción, y el bloque finally se ejecuta siempre, independientemente de si se lanzó una excepción o no. Esto es especialmente útil para liberar recursos, como archivos o conexiones a bases de datos, asegurando que se cierren correctamente.

Código 4.2: Manejo de errores en Java

```java
import java.util.Scanner;

public class Errores {

    public static void main(String[] args) {
        int valor_max = Integer.MAX_VALUE;
        int valor_min = Integer.MIN_VALUE;
        System.out.println("valor Integer MAXIMO: " + valor_max);
        System.out.println("valor Integer MINIMO: " + valor_min);
        int valor_nuevo = valor_max + 1;
        System.out.println("valor_nuevo: " + valor_nuevo);

        // ERRORES DE OVERFLOW
        try {
            Math.addExact(valor_max, 1);
            Math.subtractExact(valor_min, 1);
        } catch (Exception e) {
            System.out.println(e.getMessage()); System.out.println(e.getClass());
        }

        // ERRORES DE DIVISIÓN POR CERO
        int numero = 100;
        try {
            int numero_2 = numero / 0;
        } catch (Exception e) {
            System.out.println(e.getMessage()); System.out.println(e.getClass());
        }

        // ERRORES CONVERSIÓN DE DATOS
        String texto = "jojo";
        try {
            int numero3 = Integer.parseInt(texto);
            System.out.println(numero3);
        } catch (Exception e) {
            System.out.println(e.getMessage()); System.out.println(e.getClass());
        }

        // CAPTURANDO UN ERROR ESPECÍFICO CON CATCH DE RESPALDO.
        // IDEPENDIENTEMENTE SI SE PRODUCE O NO EL ERROR, FINALLY SE EJECUTA.
        try {
            // int numero3 = Integer.parseInt("pepe");
            // int resultado = 25/0;
            // Scanner input = new Scanner(System.in);
            // int numero_ingresado = input.nextInt();
        } catch (NumberFormatException e) {
            System.out.println(e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            // input.close();
        }
    }
}
```

4.3. Paquetes (package) en Java

Un Paquete en Java es un contenedor de clases que permite agrupar las distintas partes de un programa y que por lo general tiene una funcionalidad y elementos comunes, definiendo la ubicación de dichas clases en un directorio de estructura jerárquica, tal como se muestra en la figura 4.1.

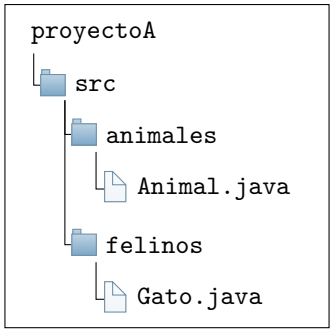


Figura 4.1: Estructura de directorios en los paquetes de Java

Tal como se muestra en la figura 4.1, los paquetes en Java consisten en una estructura de directorios, el ejemplo que se muestra en dicha figura es de un programa en Java que consta de dos paquetes, el paquete *animales* y el paquete *felinos*. Cada paquete puede contener una, ninguna o varias clases, en el caso del paquete *animales* se observa que contiene una sola clase (*Animales*), igualmente el paquete *felinos* solo contiene la clase *Gato*.

Código 4.3: Archivo Animal.java que continene a la clase *Animal* que pertenece al paquete *animales*

```
1 package animales;
2
3 public class Animal {
4     protected int edad = 10;
5
6     protected void hola() {
7         System.out.println("Hello, soy un animal");
8     }
9 }
```

Código 4.4: Archivo Gato.java que continene a la clase *Gato* que pertenece al paquete *felinos*

```
1 package felinos;
2 import animales.Animal;
3
4 public class Gato extends Animal{
5     public void gato_hola(){
6         System.out.println("soy un gato y mi edad es: " + edad );
7     }
8 }
```

4.4. Archivo JAR en java

Un archivo JAR (Java Archive) es un formato de archivo que combina varios archivos en uno solo. En Java, los archivos JAR son utilizados para almacenar y distribuir aplicaciones y bibliotecas de software. Estos archivos están comprimidos en formato ZIP y tienen la extensión *jar*.

Unidad 5

Estructuras de datos, operaciones con archivos y librerías en Java.

5.1. Estructuras de datos

5.2. Operaciones con archivos

Java proporciona una amplia gama de herramientas y clases para realizar operaciones con archivos, desde la creación y escritura hasta la lectura y eliminación de archivos.

Código 5.1: Estructura de programación secuencial en Java

```
1  import java.io.BufferedReader;
2  import java.io.BufferedWriter;
3  import java.io.IOException;
4  import java.nio.file.Files;
5  import java.nio.file.Path;
6  import java.nio.file.Paths;
7  import java.nio.file.StandardOpenOption;
8  import java.util.Scanner;
9
10 public class LeerEscribirArchivos {
11
12     public static void main(String[] args) {
13         Scanner input = new Scanner(System.in);
14         System.out.print("Escribe tu nombre : ");
15         String data = input.nextLine();
16
17         LeerEscribirArchivos archivos = new LeerEscribirArchivos();
18         String rutaArchivo = "archivo.txt";
19         archivos.guardarArchivo(rutaArchivo, data);
20         System.out.println("\nLeyendo archivo");
21         archivos.leerArchivo(rutaArchivo);
22         input.close();
23     }
24
25     private void guardarArchivo(String rutaArchivo, String data) {
26         Path myPath = Paths.get(rutaArchivo);
27         try (BufferedWriter writer = Files.newBufferedWriter(myPath,
               StandardOpenOption.APPEND,
28                 StandardOpenOption.CREATE)) {
29             writer.write(data + "\n");
30         } catch (IOException e) {
31             e.printStackTrace();
32         }
33     }
34
```

```
35      private void leerArchivo(String rutaArchivo) {
36          Path ruta = Paths.get(rutaArchivo);
37          try (BufferedReader reader = Files.newBufferedReader(ruta)) {
38              String line;
39              while ((line = reader.readLine()) != null) {
40                  System.out.println(line);
41              }
42              reader.close();
43          } catch (IOException e) {
44              e.printStackTrace();
45          }
46      }
47  }
```