PRO

UT6: Programación Orientada a Objetos

José Antonio
Benítez Chacón

Índice

1. Introducción

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que permite modelar problemas y soluciones basándose en entidades del mundo real, denominadas objetos. Estos objetos tienen características (**atributos**) y comportamientos (**métodos**) que reflejan propiedades y acciones de las entidades que representan.

La POO se fundamenta en cuatro pilares principales:

Abstracción: Simplificar la realidad para centrarse en los aspectos más relevantes del problema.
Encapsulamiento: Proteger los datos de un objeto para evitar accesos no controlados, exponiendo solo aquello que sea necesario.
Herencia: Permitir la reutilización de características de una clase base en nuevas clases derivadas.
Polimorfismo: Habilidad de los objetos de una clase derivada para comportarse como objetos de la clase base, pero con comportamientos específicos.

1. Introducción

Ventajas de la Programación Orientada a Objetos:

1) Mejora la legibilidad y el mantenimiento
2) Facilita el trabajo en equipo
3) Reutilización y escalabilidad

2. Clases y Objetos

Una **clase** es un modelo o plantilla que define las características (atributos) y comportamientos (métodos) de un tipo de entidad del mundo real. Representa un concepto general, pero no tienen una existencia concreta hasta que no se crea una instancia de ella.
Por ejemplo, una clase llamada **Coche** puede definir atributos color, marca y modelo, y métodos como acelerar() o frenar().

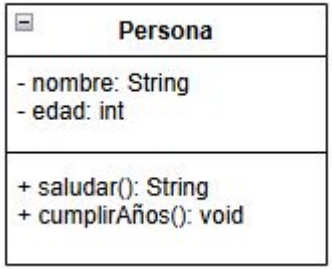
Por otro lado, un **objeto** es una instancia específica de una clase. Mientras que la clase es el concepto abstracto, el objeto es algo tangible que se puede usar en un programa.
Por ejemplo, a partir de la clase **Coche**, se puede crear un objeto llamado **miCoche** con valores como color = “rojo”, marca = “Toyota” y modelo=”Corolla”.

2. Clases y Objetos

En los diagramas UML (Unified Modeling Language), una clase se representa cmo un rectángulo dividido en tres partes:

1. El nombre de la clase
2. Los atributos de la clase
3. Los métodos de la clase

```
--------------------------------
|          Persona             |
--------------------------------
| - nombre: String             |
| - edad: int                  |
--------------------------------
| + saludar(): String          |
| + cumplirAños(): void        |
--------------------------------
```



2. Clases y Objetos

Estructura de una clase

Una clase consta de dos partes principales:

Atributos

Son las variables que definen el estado o las propiedades de un objeto.
Ejemplo: Para una clase **CuentaBancaria**, los atributos pueden ser:

saldo: Representa el dinero disponible en la cuenta.
titular: Nombre del propietario de la cuenta.

Métodos

Son las funciones que definen el comportamiento de un objeto. Los métodos manipulan los atributos de la clase y permiten interactuar con el objeto.
Ejemplo: Para la clase **CuentaBancaria**, los métodos pueden incluir:

depositar(cantidad): Añade una cantidad al saldo.
retirar(cantidad): Resta una cantidad del saldo, si es posible.

3. Creación y uso de clases en POO

Creación de Clases

En Java, una clase se define con la palabra reservada **class**, y sus atributos y métodos se especifican dentro del bloque delimitado por sus llaves:

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

 void saludar() {
   System.out.println("Hola, soy " + nombre + " y tengo " + edad + " años.");
 }
}
```

3. Creación y uso de clases en POO

Uso de constructores para inicializar objetos

Los **constructores** son métodos especiales que tienen el mismo nombre de la clase y se utilizan para inicializar los atributos de los objetos.

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

 // Constructor
 public Persona(String nombre, int edad) {
   this.nombre = nombre;
   this.edad = edad;
 }

 void saludar() {
   System.out.println("Hola, soy " + nombre + " y tengo " + edad + " años.");
 }
}
```

```
public class Main {
 public static void main(String[] args) {
   Persona persona1 = new Persona("Ana", 30);
   persona1.saludar();
 }
}
```

3. Creación y uso de clases en POO

Métodos estáticos y su uso

Los **métodos estáticos** pertenecen a la clase y no a las instancias. Se declaran con el modificador static y se utilizan par utilidades o funciones generales.

```
public class Calculadora {
 public static int sumar(int a, int b) {
   return a + b;
 }
}
```

```
public class Main {
 public static void main(String[] args) {
   System.out.println(Calculadora.sumar(5, 3));
 }
}
```

3. Creación y uso de clases en POO

Visibilidad y modificadores de acceso

Los modificadores de acceso controlan la visibilidad de los atributos y métodos:

- **public**: Accesible desde cualquier parte del programa.
- **private**: Accesible solo dentro de la clase donde se declara.
- **protected**: Accesible dentro de la clase, sus subclases y clases del mismo paquete.

```
public class Cuenta {
 private String titular;
 private double saldo;

 public Cuenta(String titular, double saldoInicial) {
   this.titular = titular;
   this.saldo = saldoInicial;
 }

 public void depositar(double cantidad) {
   if (cantidad > 0) {
     saldo += cantidad;
   }
 }

 public double obtenerSaldo() {
   return saldo;
 }
}
```

3. Creación y uso de clases en POO

Visibilidad y modificadores de acceso

Como buena práctica, los atributos suelen declararse como privados (private), por lo que para acceder a ellos para recuperar o modificar su valor se hace a través de métodos públicos que recuperar el valor (**get**) o lo fijan a otro valor (**set**). De manera general se llaman **getters** y **setters**.

De manera general, si los atributos se han nombrado de manera convencional, estos métodos comenzarán por get y set y vendrán seguidos del nombre del atributo empezando en mayúsculas:

```
public class Cuenta {
 private String titular;
 private double saldo;

 public double getSaldo() {
   return saldo;
 }
 public String getTitular() {
   return titular;
 }
 public void setTitular(String titular) {
   this.titular = titular;
 }
 public void setSaldo(double saldoInicial) {
   this.saldo = saldoInicial;
 }
}
```

4. Herencia

La **herencia** en Java permite que una clase (subclase) herede atributos y métodos de otra clase (superclase).
Desde otro punto de vista, puede verse que la subclase extiende la funcionalidad de la superclase.
Esto se consigue mediante la palabra reservada **extends**.

Ejemplo de **Herencia simple**:

```
public class Animal {
 String nombre;

 public Animal(String nombre) {
   this.nombre = nombre;
 }

 public void hacerSonido() {
   System.out.println("El animal hace un
sonido.");
 }
}
```

```
public class Perro extends Animal {
 public Perro(String nombre) {
   super(nombre);  // Llamada al constructor de la superclase
 }

 @Override
 public void hacerSonido() {
   System.out.println(nombre + " dice: Guau");
 }
}
```

4. Herencia

Ejemplo de **Herencia jerárquica** (varias clases heredan de otra):

```java
public class Animal {
 String nombre;

 public Animal(String nombre) {
   this.nombre = nombre;
 }

 public void hacerSonido() {
   System.out.println("El animal hace un
sonido.");
 }
}
```

```java
public class Perro extends Animal {
 public Perro(String nombre) {
   super(nombre);
 }

 @Override
 public void hacerSonido() {
   System.out.println(nombre + " dice: Guau");
 }
}
```

```java
public class Gato extends Animal {
 public Gato(String nombre) {
   super(nombre);
 }

 @Override
 public void hacerSonido() {
   System.out.println(nombre + " dice: Miau");
 }
}
```

4. Herencia

Polimorfismo

El **polimorfismo** en POO permite que una operación, método o función pueda comportarse de diferentes maneras según el contexto en que se utilice. En Java se logra principalmente de dos formas: sobrecarga de métodos (overloading) y sobrescritura de métodos (overwriting).

Overloading

La sobrecarga de métodos ocurre cuando una clase define varios métodos con el mismo nombre, pero diferentes tipos, disposición o número de parámetros.
Es una forma de polimorfismo en tiempo de compilación.

```java
public class Calculadora {
 // Método para sumar enteros
 public int sumar(int a, int b) {
   return a + b;
 }

 // Método para sumar números de tipo double
 public double sumar(double a, double b) {
   return a + b;
 }

 // Método para sumar tres números enteros
 public int sumar(int a, int b, int c) {
   return a + b + c;
 }
}
```

4. Herencia

Overwriting

La sobrescritura de métodos ocurre cuando una clase hija redefine un método que ya existe en su clase padre.
Es una forma de polimorfismo en tiempo de ejecución.

Reglas para sobrescritura:

1. El método debe tener el mismo nombre, parámetros y tipo de retorno que en la clase padre.
2. La visibilidad del método en la subclase debe ser igual o más permisiva que en la superclase.
3. El método de la clase padre debe ser heredable (**no** puede ser **private** ni **final**).
4. Se puede utilizar la anotación **@Override** para asegurarse de que el método está sobrescribiendo correctamente.

```java
public class Animal {
 public void hacerSonido() {
   System.out.println("El animal hace un sonido.");
 }
}

public class Perro extends Animal {
 @Override
 public void hacerSonido() {
   System.out.println("El perro dice: ¡Guau!");
 }
}

public class Gato extends Animal {
 @Override
 public void hacerSonido() {
   System.out.println("El gato dice: ¡Miau!");
 }
}
```

4. Herencia

Ventajas del polimorfismo:

1. **Flexibilidad**: Permite escribir código más genérico y reutilizable.
2. **Extensibilidad**: Facilita la ampliación del programa sin alterar el código existente.
3. **Legibilidad y organización**: Ayuda a organizar el comportamiento relacionado en una jerarquía lógica.

5. Librerías de clases

Las librerías de clases en Java son conjuntos de clases que contienen métodos reutilizables. Pueden ser tan simples como un archivo **.java** o tan complejas como un archivo **.jar** que contenga múltiples clases empaquetadas.

Creación de una librería básica

1) Estructura de archivos

Crea una estructura de paquetes para organizar las clases de tu librería (por ejemplo: paquete.matematicas).

```
src/
└── paquete/
    └── matematicas/
        └── Matematicas.java
```

2) Clases que se quieren incluir en la librería

```java
package paquete.matematicas;

public class Matematicas {

 // Método para sumar dos enteros
 public static int suma(int a, int b) {
   return a + b;
 }
…
}
```

5. Librerías de clases

3) Compilación de la librería

1. Navega al directorio raíz del proyecto donde se encuentra el paquete.
2. Compila la clase (Matematicas.java en este caso):

```
javac -d . src/paquete/matematicas/Matematicas.java
```

Esto generará un archivo .class dentro de una estructura de carpetas que refleja el paquete: paquete/matematicas/Matematicas.class y que puede utilizarse en otra clase si se importa:

```java
import paquete.matematicas.Matematicas;

public class Main {
 public static void main(String[] args) {
   System.out.println("Suma: " + Matematicas.suma(5, 3));
   System.out.println("Resta: " + Matematicas.resta(10, 4));
   System.out.println("Multiplicación: " + Matematicas.multiplicar(6, 7));
   System.out.println("División: " + Matematicas.dividir(12, 4));
 }
}
```

5. Librerías de clases

3) Compilación de la librería

Lo anterior es desde consola, pero también se puede hacer desde el IDE. Desde INtelliJ:

Método 1: Generar un .jar manualmente

1. Abrir Project Structure
   Ve a File → Project Structure (Ctrl + Alt + Shift + S en Windows/Linux, Cmd + ; en Mac).
2. Ir a "Artifacts"
   En el panel izquierdo, selecciona Artifacts.
   Clic en + → JAR → From modules with dependencies.
3. Seleccionar la clase principal (si aplica)
   Si el .jar es ejecutable, selecciona la clase con el método main().
4. Establecer la ruta de salida
   Define dónde guardar el .jar en Output Directory.
5. Compilar y generar el .jar
   Clic en Apply y luego OK.
   Ve a Build → Build Artifacts... → Build para generar el .jar.

Método 2: Usando Gradle o Maven (opcional)

Si usas Gradle o Maven, puedes generar un .jar automáticamente con sus herramientas de compilación.

5. Librerías de clases

4) Empaquetar como .jar

Para facilitar su reutilización, se puede empaquetar la librería en un archivo **.jar**.

1. Crear el archivo .jar:

```
jar cf matematicas.jar paquete/matematicas/Matematicas.class
```

2. Incluye el archivo .jar en el proyecto que lo necesita, añadiéndolo al classpath:

```
javac -cp matematicas.jar Main.java
java -cp .:matematicas.jar Main
```

Esto era desde consola. Desde el IDE (IntelliJ)podría hacerse:

Clic derecho en el proyecto → Open Module Settings.
Ir a Libraries → + → Java.
Selecciona matematicas.jar y agrégalo.
Aplica los cambios y ejecuta el programa.

6. Clases de uso común

En Java, hay una serie de clases de uso común (además de las ya vistas para estructuras en la UT5), de las que veremos algunos detalles.

String

La clase se utiliza para representar cadenas de texto. String es inmutable, lo que significa que cualquier operación que modifique una cadena realmente devuelve una nueva instancia de String.

Hay muchos método útiles (muchos los podéis ver en Moodle), como (dada una cadena **a**):

- a.substring(int beginIndex, int endIndex): Extrae una subcadena.
- a.charAt(int index): Retorna el carácter en la posición dada.
- a.indexOf(String str): Devuelve la posición de la primera aparición de una subcadena.
- a.lastIndexOf(String str): Devuelve la posición de la primera aparición de una subcadena.
- a.toLowerCase(): Convierte el string a minúsculas.
- a.toUpperCase(): Convierte el string a mayúsculas.
- a.equals(String otro): Compara el contenido de dos cadenas.
- a.equalsIgnoreCase(String otro): Compara dos cadenas ignorando mayúsculas y minúsculas.
- a.compareTo(String otro): Compara dos cadenas lexicográficamente (si la cadena pasada como parámetro es mayor, igual o menor que la que llama al método).
- a.replace(String vieja, String nueva): Reemplaza en la cadena llamante la cadena vieja por la nueva.
- a.trim(): Elimina espacios en blanco al inicio y al final.
- a.split(String regex): Divide una cadena en un array según un delimitador.
- a.startsWith(String prefijo): Verifica si comienza con un prefijo.
- a.endsWith(String sufijo): Verifica si termina con un sufijo.
- a.contains(CharSequence s): Verifica si contiene una secuencia de caracteres.

6. Clases de uso común

String

Ejemplo:

```
public class EjemploString {
 public static void main(String[] args) {
   String texto = "La lluvia en Sevilla es una maravilla." ;

   System.out.println("Subcadena (5-7): " + texto.substring(5, 7));
   System.out.println("Carácter en índice 3: " + texto.charAt(3));
   System.out.println("Posición de 'Sevilla': " + texto.indexOf("Sevilla"));
   System.out.println("Minúsculas: " + texto.toLowerCase());
   System.out.println("Mayúsculas: " + texto.toUpperCase());
 }
}
```

6. Clases de uso común

Integer

La clase Integer envuelve un valor primitivo int en un objeto. Es parte de las clases envolventes (como Long, Float, Double, Boolean, Character, Short o Byte).

Entre los métodos comunes están (dado un Integer **a**):

- Integer.parseInt(String s): Convierte un String en un int.
- Integer.valueOf(String s): Convierte un String en un objeto Integer.
- a.toString(int i): Convierte un int a String.
- Integer.max(int a, int b) → Devuelve el máximo entre dos enteros.
- Integer.min(int a, int b) → Devuelve el mínimo entre dos enteros.
- Integer.compare(int x, int y): Compara dos valores int.
- a.compareTo(int x): Compara el valor del Integer llamante con el int que se pasa por parámetro.
- a.equals(Object obj): Compara dos objetos Integer.

Similar a Integer, podemos considerar otras clases numéricas, como Long, Decimal o Float, que tendrán sus propios métodos, como:

- Long.parseLong(String s)
- Float.parseFloat(String s)
- Double.parseDouble(String s)

NOTA: Las cadenas para pasar a Double o Float deben tener el punto (.) como separador decimal.

6. Clases de uso común

Math

La clase Math contiene métodos estáticos para cálculos matemáticos:

- Math.abs(double a): Valor absoluto.
- Math.pow(double a, double b): Potencia (a^b).
- Math.sqrt(double a): Raíz cuadrada.
- Math.round(double a): Redondeo estándar (a long).
- Math.ceil(double a): Redondeo hacia arriba.
- Math.floor(double a): Redondeo hacia abajo.
- Math.random(): Devuelve un número aleatorio entre 0.0 y 1.0.

```
public class EjemploMath {
 public static void main(String[] args) {
   System.out.println("Valor absoluto de -5: " + Math.abs(-5));
   System.out.println("2^3: " + Math.pow(2, 3));
   System.out.println("Raíz cuadrada de 16: " + Math.sqrt(16));
   System.out.println("Redondeo de 4.7: " + Math.round(4.7));
   System.out.println("Redondeo hacia arriba de 4.1: "+ Math.ceil(4.1));
   System.out.println("Redondeo hacia abajo de 4.9: "+ Math.floor(4.9));
   System.out.println("Número aleatorio entre 1 y 10: "+ (int)(Math.random() * 10 + 1));
 }
}
```

6. Clases de uso común

LocalDate

Sirve para manejar fechas sin tiempo.

Tenemos, por ejemplo, los siguientes métodos (teniendo una fecha de tipo LocalDate f):

- LocalDate.now(): Obtiene la fecha actual.
- LocalDate.of(year, month, day): Crea una fecha específica.
- f.plusDays(n): Suma n días (puede ser una cantidad negativa para restar).
- f.plusWeeks(n): Suma n semanas (puede ser una cantidad negativa para restar).
- f.plusMonths(n): Suma n meses (puede ser una cantidad negativa para restar).
- f.plusYears(n): Suma n años (puede ser una cantidad negativa para restar).
- f.getDayOfMonth(): Devuelve el día del mes.
- f.getDayOfWeek(): Devuelve el día de la semana (por defecto muestra el número).
- f.getMonth(): Devuelve el mes (por defecto muestra el número).
- f.getYear(): Devuelve el año.
- f.format(DateTimeFormatter): Formatea una fecha en base a un DateTimeFormatter dado (patrón de fecha).

NOTA: Además de los métodos plusXXX para añadir tiempo, tenemos los métodos minusXXX para restar tiempo.

Para mostrar el nombre del mes o del día de la semana podemos llamar a getDisplayName(TextStyle.FULL, Locale.getDefault()) sobre el campo devuelto (probad diferentes opciones de configuración).

```
System.out.println(f.getMonth().getDisplayName(TextStyle.SHORT, Locale.getDefault()));
System.out.println(f.getDayOfWeek().getDisplayName(TextStyle.FULL, Locale.JAPANESE));
System.out.println(f.getDayOfWeek().getDisplayName(TextStyle.NARROW, Locale.forLanguageTag("FR")));
```

6. Clases de uso común

LocalDate

Para ver la diferencia en una unidad de tiempo entre dos fechas (LocalDate) dadas, se pueden utilizar las siguientes opciones:

1) **ChronoUtils**

Se puede ver la diferencia en una unidad de tiempo dada entre dos LocalDate usando el método between de la siguiente manera:

```
LocalDate fechaInicio = LocalDate.of(2000, 2, 15);
LocalDate fechaFin = LocalDate.of(2024, 2, 5);

long meses = ChronoUnit.MONTHS.between(fechaInicio, fechaFin);
System.out.println("Diferencia en meses: " + meses);
```

2) **Period**

Se puede ver la diferencia en una unidad de tiempo dada entre dos LocalDate usando el método between y luego recuperando las unidades deseadas:

```
LocalDate fechaInicio = LocalDate.of(2000, 2, 15);
LocalDate fechaFin = LocalDate.of(2024, 2, 5);

Period periodo = Period.between(fechaInicio, fechaFin);
System.out.println("Años: " + periodo.getYears());
```

6. Clases de uso común

LocalTime

Si lo que queremos es manejar horas sin fecha, tenemos la clase LocalTime.

Tenemos, por ejemplo, los siguientes métodos (teniendo una hora de tipo LocalTime h):

- LocalTime.now(): Obtiene la hora actual.
- LocalTime.of(hour, minute): Crea una hora específica (con hora y minutos).
- LocalTime.of(hour, minute, second): Crea una hora específica (con hora. minutos y segundos).
- LocalTime.of(hour, minute, second, nanosecond): Crea una hora específica (con hora, minutos, segundos y nanosegundos).
- h.plusHours(n): Suma n horas (puede ser una cantidad negativa para restar).
- h.plusMinutes(n): Suma n minutos (puede ser una cantidad negativa para restar).
- h.plusSeconds(n): Suma n segundos (puede ser una cantidad negativa para restar).
- h.plusNanos(n): Suma n nanosegundos (puede ser una cantidad negativa para restar).
- h.getHour(): Devuelve la hora.
- h.getMinute(): Devuelve los minutos.
- h.getSecond(): Devuelve los segundos.
- h.getNano(): Devuelve los nanosegundos.
- h.format(DateTimeFormatter): Formatea una hora en base a un DateTimeFormatter dado (patrón de fecha).

NOTA: Además de los métodos plusXXX para añadir tiempo, tenemos los métodos minusXXX para restar tiempo.

6. Clases de uso común

DateTimeFormatter

Esta clase sirve para dar formato a las fechas según queramos mostrarlas. No solo sirve con LocalDate y LocalTime, también funciona con otras clases del paquete java.time, como LocalDateTime, ZonedDateTime, OffsetDateTime y OffsetTime.

Veremos algunos de los patrones más comunes usados con DateTimeFormatter.ofPattern() y una salida de ejemplo:

LocalDate (solo fecha)

Patrón	Ejemplo
"dd/MM/yyyy"	03/04/2000
"yyyy-MM-dd"	2000-04-03
"dd MMM yyyy"	03 abr 2000
"EEEE, d 'de' MMMM yyyy"	Lunes, 3 de abril de 2000

LocalTime (solo hora)

Patrón	Ejemplo
"HH:mm:ss"	14:30:15
"hh:mm a"	02:30 PM
"HH:mm"	14:30

6. Clases de uso común

DateTimeFormatter

LocalDateTime

Patrón	Ejemplo
"dd/MM/yyyy HH:mm:ss"	03/04/2000 14:30:15
"yyyy-MM-dd HH:mm"	2000-04-03 14:30
"EEEE, d MMMM yyyy HH:mm"	Lunes, 3 abril 2000 14:30
"EEEE, d 'de' MMMM yyyy"	Lunes, 3 de abril de 2000

ZonedDateTime y **OffsetDateTime** (Fecha + hora + zona horaria)

Patrón	Ejemplo
"yyyy-MM-dd HH:mm:ss z"	2000-04-03 14:30:15 UTC
"yyyy-MM-dd HH:mm:ss XXX"	2000-04-03 14:30:15 +02:00
"EEE, d MMM yyyy HH:mm:ss z"	lun, 3 abr 2000 14:30:15 UTC

```
LocalDate ld = LocalDate.of(2000, 2, 15);
System.out.println(ld.format(DateTimeFormatter.ofPattern("dd/MM/yyyy")));
LocalTime lt2 = LocalTime.of(16,2,23,23);
System.out.println(lt2.plusHours(20).format(DateTimeFormatter.ofPattern("HH:mm:ss")));
ZonedDateTime zf = ZonedDateTime.now();
System.out.println(zf.format(DateTimeFormatter.ofPattern("EEE, d MMM yyyy HH:mm:ss z")));
System.out.println(zf.format(DateTimeFormatter.ofPattern("EEE, d MMM yyyy HH:mm:ss z",Locale.CHINA)));
```

6. Clases de uso común

DateTimeFormatter

En el caso de que estemos trabajando con la antigua clase **Date**, no podrá utilizarse DateTimeFormatter. Para estos casos, se utilizará **SimpleDateFormat**:

```
Date date = new Date();
SimpleDateFormat sdf =  new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
String fechaFormateada = sdf.format(date);

System.out.println("Fecha formateada: "  + fechaFormateada);
```

6. Clases de uso común

System

La clase System pertenece al paquete java.lang y proporciona métodos y variables útiles para interactuar con el sistema en tiempo de ejecución. Entre sus usos más comunes están entrada/salida, control de tiempo, manejo de propiedades del sistema y finalización de la aplicación.

System.out: salida estándar. El uso más común es System.out.println, que imprime por consola.

- System.out.println("Hola"): Imprime por consola con salto de línea.
- System.out.print("Sin salto"): Imprime por consola sin salto de línea.
- System.out.printf("El número es: %d%n", 42): Imprime con formato

System.err: salida de error. Como System.out, pero para mensajes de error.

- System.err.println("¡Error crítico! Algo salió mal."): Imprime error con salto de línea.

System.in: entrada estándar. Se usa para leer entrada de usuario a través de la consola (normalmente se combina con Scanner)

- Scanner scanner = new Scanner(System.in);

6. Clases de uso común

System

System.currentTimeMillis: Devuelve el tiempo actual en milisegundos (desde el 1 de enero de 1970). Suele utilizarse para ver tiempos de ejecución de un programa, bucle, función, etc, haciendo una llamada al comienzo, otra al finalizar, y restando ambos valores.

```
long inicio = System.currentTimeMillis();
for(int var=0;var< 100000; var++){
 if(var%400==0){
   System.out.println(var+" es divisible por 400");
 }
}
```

System.nanoTime: similar a System.currentTimeMillis(), pero con mayor precisión (nanosegundos)

System.exit: Finaliza el programa inmediatamente (no se ejecutan más instrucciones). Si el número que se pasa es 0 se entiende que la salida es exitosa. Si no, indica error o terminación inesperada.

System.getProperty: Permite obtener información del sistema operativo, versión de Java, etc.

```
System.out.println("Sistema Operativo: " + System.getProperty("os.name"));
System.out.println("Versión de Java: " + System.getProperty("java.version"));
System.out.println("Carpeta de usuario: " + System.getProperty("user.home"));
System.exit(0);
System.out.println("Esto no va a mostrarse");
```

6. Clases de uso común

System

System.getenv: Permite acceder a las variables de entorno del sistema operativo

```
System.out.println("PATH: " + System.getenv("PATH"));
System.out.println("Usuario: " + System.getenv("USERNAME")); // En Windows
System.out.println("Home: " + System.getenv("HOME")); // En Linux/Mac
```

System.gc: pide al recolector de basura (**G**arbage **C**ollector) que intente liberar memoria (NOTA: no garantiza que ocurra de inmediato, solo indica un buen momento para ejecutarla).

```
System.gc();
```

7. Comparación de objetos

equals

En Java se utilizan dos formas para la comparación de objetos:

1) **==** Compara referencias de objetos en memoria (en tipos primitivos, no objetos, sí sirve para comparar su valor)
2) **equals**(Object obj): Compara el contenido de los objetos

Todas las clases, como heredan de **Object** (aunque no esté explicitado), si no redefine el método equals (como sí ocurre, por ejemplo, con las clases String, Integer, Double, etc.), utilizan el método equals de **Object**, que comparan las referencias en memoria (lo mismo que ==).

Si deseamos que en una clase personalizada se compare en base a valores de atributos, hay que sobreescribir el método equals, para que solo devuelve cierto en el caso que nosotros deseemos (dependerá de la clase, de la lógica de negocio, etc.).

7. Comparación de objetos

hashCode

hashCode es un método que devuelve un hash basado, por defecto, en la referencia en memoria al objeto.
hashCode se utiliza en colecciones basadas en hash, como HashMap, HashSet o HashTable.

Si dos objetos son iguales según **equals**(), deben tener el mismo **hashCode**(). Sin embargo, objetos con el mismo **hashCode()** no necesariamente son iguales (puede haber colisiones).

Cuando agregamos objetos a un HashSet o usamos un HashMap, estos funcionan en dos pasos:

- Primero, verifican el hashCode() para ubicar rápidamente el objeto en la estructura.
- Luego, si hay colisión (dos objetos con el mismo hash), usan equals() para comprobar si realmente son iguales.

Por este motivo, ambos métodos deben definirse en las clases que creemos si queremos que estos se utilicen bien en colecciones basadas en hash.

7. Comparación de objetos

hashCode

```
public class  Persona {
   String  nombre;

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

public class  HashCodeEjemplo {
   public static void  main (String[] args) {
       Persona p1 =  new  Persona ("Juan");
       Persona p2 =  new  Persona ("Juan");

       System. out .println(p1.hashCode());  // Valor
diferente
       System. out .println(p2.hashCode());  // Valor
diferente
   }
}
```

```
public class Persona {
   String nombre;

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

   @Override
   public boolean equals(Object obj) {
       if (this == obj) return true; // Mismo objeto en memoria
       if (obj == null || getClass() != obj.getClass())return false; // Diferente clase
       Persona persona = (Persona) obj;
       return nombre.equals(persona.nombre); // Comparación de contenido
   }

   @Override
   public int hashCode() {
       return Objects.hash(nombre); // Genera un hash basado en el atributo "nombre"
   }
}

public class HashCodeEqualsEjemplo {
   public static voidmain(String[] args) {
       Persona p1 =new Persona("Juan");
       Persona p2 =new Persona("Juan");

       System.out.println(p1.equals(p2));// true (porque tienen el mismo nombre)
       System.out.println(p1.hashCode() == p2.hashCode());// true correctamente)
   }
}
```

En el primer caso, se considerarían objetos diferentes, por lo que podría haber varios del mismo en un conjunto.
En el segundo caso, se consideran iguales, por lo que no vuelven a insertarse si ya están.

7. Comparación de objetos

Interfaz Comparable

La interfaz Comparable permite que los objetos se comparen entre sí de manera natural (es decir, con un orden predefinido). Es necesario que las clases implementen esta interfaz si se quiere tener una ordenación por defecto en caso de que se creen colecciones de este tipo de objetos y quieran ordenarse.

Esta Interfaz indica que ha de implementarse el método **compareTo**.

```
int compareTo(T o);
```

Devuelve:

- Valor negativo si el objeto actual (que llama al método) es "menor" que el objeto comparado (pasado por parámetro).
- Valor cero si son iguales.
- Valor positivo si el objeto actual es "mayor".

Hay clases, como String, que ya tienen predefinido este método. En clases que creemos nosotros tendremos que implementar la lógica en base a qué consideremos “ser mayor” o “ser menor” para una ordenación.

7. Comparación de objetos

Interfaz Comparator

La interfaz **Comparator** permite definir un orden de comparación externo a la clase de los objetos a comparar, lo que es útil cuando no podemos o no queremos modificar la clase original.

Esta Interfaz indica que ha de implementarse el método **compare**, que compara dos objetos de cualquier tipo (no tiene que ser del mismo tipo que el de la clase).

```
int compare(T o1, T o2);
```

Devuelve:

- Valor negativo si o1 es "menor" que o2.
- Valor cero si son iguales.
- Valor positivo si o1 es "mayor" que o2.

7. Comparación de objetos

Interfaz Comparator

```
class CompararPorEdad implements Comparator<Persona> {
 @Override
 public int compare(Persona p1, Persona p2) {
   return Integer.compare(p1.getEdad(), p2.getEdad());
 }
}

class CompararPorNombre implements Comparator<Persona> {
 @Override
 public int compare(Persona p1, Persona p2) {
   return p1.getNombre().compareTo(p2.getNombre());
 }
}

public class CompararConComparator {
 public static void main(String[] args) {
   List<Persona> personas =  new ArrayList<>();
   personas.add(new Persona("Juan","Pérez",25));
   personas.add(new Persona("Ana","López" ,30));
   personas.add(new Persona("Carlos", "Wenger",22));

   // Ordenar por edad
   Collections.sort(personas,  new CompararPorEdad());
   System.out.println(personas);  // [Carlos (22), Juan (25), Ana (30)]

   // Ordenar por nombre
   Collections.sort(personas,  new CompararPorNombre());
   System.out.println(personas);  // [Ana (30), Carlos (22), Juan (25)]
 }
}
```

7. Comparación de objetos

Cuando se usa el Collections.sort() o el método sort() de las listas en Java, podemos proporcionar un Comparator para personalizar la ordenación concreta que queremos utilizar.

Hay diferentes opciones dependendiendo del uso que vaya a hacerse y las veces que vaya a reutilizarse:

1) **Usar un Comparator como clase separada**
   Se puede crear una clase que implemente Comparator y que defina internamente el método compare (como hemos visto en la diapositiva anterior)

2) **Usar un Comparator anónimo**
   En este caso, definimos el método compare en el momento de la instanciación de Comparator en la ordenación:

```
Collections.sort(personas, new Comparator<Persona>() {
 @Override
 public int compare(Persona p1, Persona p2) {
   return Integer.compare(p1.getEdad(), p2.getEdad());
 }
});
```

7. Comparación de objetos

3) Usar una expresión lambda como Comparator

Desde Java 8 se pueden usar expresiones lambdas para crear comparaciones de manera concisa y la más recomendada si solo se necesita un Comparator en un sort concreto.

```
Collections.sort(personas, (p1, p2) -> Integer.compare(p1.getEdad(),
p2.getEdad()));
```

4) Usar un Comparator.comparing()

A partir de Java8, Comparator ofrece varios métodos estáticos como comparing(), que facilitan la creación de comparadores. Este método permite indicar una clave (atributo) por el que queremos ordenar.

```
personas.sort(Comparator.comparing(p -> p.getEdad()));
```

7. Comparación de objetos

5) Usar Comparator.reversed()

Si ya tenemos un Comparator y queremos invertir el orden, se puede utilizar el método estático reversed() de la clase Comparator.

```
personas.sort(Comparator. comparing((Persona p) -> p.edad).reversed());
```

6) Usar un Comparator.thenComparing()

Si se necesita ordenar por múltiples criterios, puede utilizarse thenComparing() útil cuando hay criterios de comparación que son iguales y se requiere de otro para desempatar).

```
personas.sort(Comparator.comparing((Persona p) -> p.getApellido()).reversed().thenComparing(p -> p.getNombre()));

personas.sort(Comparator.comparing((Persona p) -> p.getApellido()).reversed().thenComparing(p -> p.getNombre()).reversed());

personas.sort(Comparator.comparing((Persona p) -> p.getApellido()).reversed().thenComparing(
 Comparator.comparing((Persona p) -> p.getNombre()).reversed()));
```

7. Comparación de objetos

7) Usar referencias a métodos

En vez de expresiones lambda, se pueden utilizar también referencias a métodos para hacer el código más limpio (y pueden combinarse con thenComparing y reversed)

```
personas.sort(Comparator.comparingInt(Persona::getEdad));
System.out.println("Ordenado por edad ascendente: "+ personas);

personas.sort(Comparator.comparing(Persona::getApellido)
 .thenComparing(Persona::getNombre));
System.out.println("Ordenado por apellido y nombre ascendente: "+ personas);


personas.sort(Comparator.comparing(Persona::getApellido)
 .reversed()
 .thenComparing(Persona::getNombre));
System.out.println("Ordenado por apellido descendente y nombre ascendente: "+
personas);

personas.sort(Comparator.comparing(Persona::getNombre)
 .thenComparing(Comparator.comparingInt(Persona::getEdad).reversed()));
System.out.println("Ordenado por nombre ascendente y edad descendente: "+ personas);
```

7. Comparación de objetos

7) Control de nulos

Si se quiere controlar qué hacer con los objetos nulos se puede utilizar los métodos nullsFirst() y nullsLast() de Comparator.

```
List<Persona> personas =  new ArrayList<Persona>();
Persona per =  null;
personas.add(new Persona("Juan", "Pérez", 25));
personas.add(new Persona("Ana", "Pérez", 20));
personas.add(new Persona("Carlos", "Wenger", 22));
personas.add(per);

personas.sort(Comparator.nullsFirst(new CompararPorEdad()));
System.out.println(personas);
personas.sort(Comparator.nullsLast(new CompararPorNombre()));
System.out.println(personas);
```

NOTA: El incluir nulos en una lista puede hacer que muchos métodos fallen al no poder ser llamados sobre un nulo. En este caso los Comparator personalizados tienen que tener en cuenta esta lógica para devolver un entero en caso de que alguno de los dos objetos comparados sea nulo.

7. Comparación de objetos

Interfaz Collator

La interfaz java.text.**Collator** sirve para tener en cuenta las reglas de un idioma a la hora de ordenar texto (por ejemplo si hay acentos o caracteres especiales). Necesitamos indicar el Locale (la localización, idioma, etc.) que vamos a utilizar.

```
List<String> palabras = Arrays. asList("árbol", "casa", "cuándo", "cuesco");
List<String> katakana = Arrays. asList("ア","エ","イ","オ","ウ");
palabras.sort(Comparator. naturalOrder());
System.out.println(palabras);
Collator collator = Collator. getInstance(Locale.of("es","ES"));
palabras.sort(collator);
System.out.println(palabras);
katakana.sort(Collator. getInstance(Locale.JAPANESE));
System.out.println(katakana);
```

```
[casa, cuesco, cuándo, árbol]
[árbol, casa, cuándo, cuesco]
[ア, イ, ウ, エ, オ]
```

8. Expresiones regulares

Las expresiones regulares (RegEx) son patrones que permiten buscar, validar, dividir o reemplazar texto de manera eficiente.

En Java se utilizan las clases del paquete java.util.regex:

- **Pattern**: para una expresión regular compilada (patrón)
- **Matcher**: para realizar coincidencias sobre una cadena dada.

```
String texto = "El coche de Juan es rojo y el de Ana es azul.";

Pattern pattern = Pattern.compile("\\b[A-Z][a-z]+\\b");
Matcher matcher = pattern.matcher(texto);

while (matcher.find()) {
 System.out.println("Encontrado: " + matcher.group());
}
```

El método Pattern.compile convierte una expresión regular en un objeto reutilizable que mejora el rendimiento en iteraciones o múltiples búsquedas. Además, admite opciones avanzadas, como Pattern.CASE_INSENSITIVE o Pattern.MULTILINE.

8. Expresiones regulares

Métodos clave de Pattern y Matcher para trabajar con expresiones regulares:

Método	Descripción
Pattern.compile(String regex)	Verifica si toda la cadena coincide con el patrón.
Matcher matcher(String input)	Crea un Matcher para analizar la cadena dada.
find()	Busca la siguiente coincidencia en la cadena.
matches()	Verifica si la cadena completa coincide con la regex.
group()	Devuelve la última coincidencia encontrada.
replaceAll(String replacement)	Reemplaza todas las coincidencias con la cadena dada.
split(String input)	Divide una cadena según el patrón.

```
String regex = "\\d{3}-\\d{3}-\\d{4}";

Pattern pat = Pattern.compile(regex);
Matcher mat = pattern.matcher(texto);

if (mat.find()) {
 System.out.println("Número encontrado: " + mat.group());
} else {
 System.out.println("No se encontró ningún número." );
}
```

8. Expresiones regulares

Elemento	Descripción	Función	Descripción
.	Coincide con cualquier carácter excepto una nueva línea. Ej: a.b coincide con "a*b" y "a1b".	\\w	Coincide con cualquier carácter alfanumérico o guion bajo. Ej: \\w+ coincide con "abc123".
^	Indica el inicio de una cadena. Ej: ^Hola coincide con "Hola mundo", pero no con "Mundo Hola"	\\W	Coincide con cualquier carácter no alfanumérico. Ej: \\W+ coincide con "@#!".
$	Indica el final de una cadena. Ej: mundo$ coincide con "Hola mundo", pero no con "mundo Hola".	\\s	Coincide con cualquier espacio en blanco (espacio, tabulación, salto de línea). Ej: \\s+ coincide con " " o "\t".
*	Coincide con 0 o más repeticiones del carácter o grupo anterior. Ej: a* coincide con "", "a", "aa", "aaa".	\\S	Coincide con cualquier carácter que no sea un espacio en blanco. Ej: \\S+ coincide con "abc".
+	Coincide con 1 o más repeticiones del carácter o grupo anterior. Ej: a+ coincide con "a", "aa", pero no con "".	()	Define un grupo o captura subexpresión. Ej: (abc)+ coincide con "abcabc".
?	Coincide con 0 o 1 aparición del carácter o grupo anterior. Ej: colou?r coincide con "color" y "colour".	(?<name>...)	Define un grupo con nombre. Permite referenciar el grupo por su nombre. Ej: (?<letra>\w) captura un carácter alfanumérico.
{n}	Coincide exactamente con n repeticiones del carácter o grupo anterior. Ej: a{3} coincide con "aaa".	(?:...)	(?:abc)+ no almacena "abc".
{n,}	Coincide con al menos n repeticiones. Ej: a{2,} coincide con "aa", "aaa", "aaaa", etc.	(?=...)	Lookahead positivo: coincide si lo que sigue satisface el patrón. Ej: a(?=b) coincide con "a" en "ab".
{n,m}	Coincide con entre n y m repeticiones (ambas incluidas). Ej: a{2,4} coincide con "aa", "aaa", y "aaaa".	(?!...)	Lookahead negativo: coincide si lo que sigue no satisface el patrón. Ej: a(?!b) coincide con "a" en "ac".
[]	Define un conjunto de caracteres permitidos. Ej: [aeiou] coincide con cualquier vocal minúscula no acentuada.	(?<=...)	Lookbehind positivo: coincide si lo que precede satisface el patrón. Ej: (?<=b)a coincide con "a" en "ba".
[^]	Define un conjunto de caracteres no permitidos. Ej: [^aeiou] coincide con cualquier carácter no vocal.	(?<!...)	Lookbehind negativo: coincide si lo que precede no satisface el patrón. Ej: (?<!b)a coincide con "a" en "ca".
\\	Escapa un carácter especial o introduce una secuencia especial. Ej: \\. coincide con "." literal.		
\\d	Coincide con cualquier dígito (0-9). Ej: \\d+ coincide con "123".	\\b	Coincide con un límite de palabra (inicio o final). Ej: \\bcat\\b coincide con "cat" pero no con "catalog".
\\D	Coincide con cualquier carácter que no sea un dígito. Ej: \\D+ coincide con "abc".	\\B	Coincide con una posición que no es un límite de palabra. Ej: \\Bcat\\B coincide dentro de palabras como "Hécate", pero no "cat".

8. Expresiones regulares

Hay métodos de **String** que trabajar directamente con expresiones regulares (también funcionan con cadenas literales, pero entonces es como si la expresión regular fuera fija).

Método	Descripción
boolean matches(String regex)	Verifica si toda la cadena coincide con el patrón.
String replaceAll(String regex, String nuevo)	Reemplaza todas las coincidencias del patrón con la cadena dada.
String replaceFirst(String regex, String nuevo)	Reemplaza solo la primera coincidencia.
String[] split(String regex)	Divide la cadena según el patrón.
String[] split(String regex, int limit)	Divide la cadena en un número máximo de partes.

```
String textoRE = "Hola123, esto es un 456 texto con números 789." ;
String resultado = textoRE.replaceAll( "\\d+", "X");
System.out.println(resultado);
```

NOTA: el método **matches** de String solo acepta coincidencias totales. Para parciales habría que utilizar Matcher.