

# CLASES Y OBJETOS

Representa una abstracción del mundo real

## **Clase:**

Una “plantilla” que define las propiedades y comportamientos (atributos y métodos) comunes a todos los objetos de ese tipo.

Representa una concreción de esa abstracción del mundo real

## **Objeto:**

Una instancia de una clase.

# EJEMPLO DE CLASES Y OBJETOS

**Clase:**  
*Coche*

Clase Coche

arrancar, ir, parar, girar

color, velocidad, carburante

← nombre de la clase

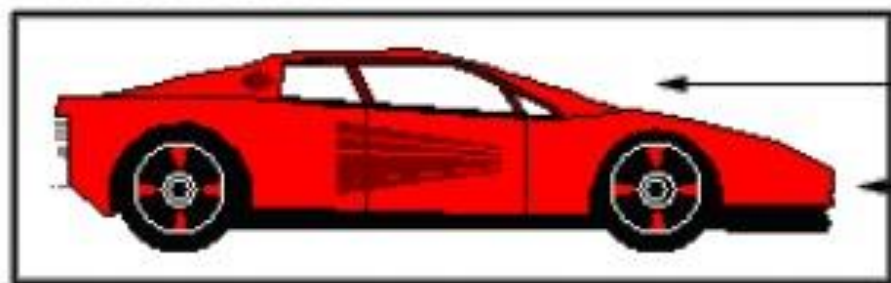
← métodos (funciones)

← atributos (datos)

◆ **Objeto:** *Ferrari*

coche.ferrari

← nombre del objeto



← métodos

arrancar, ir, parar, girar

← datos

rojo, 280 km/h, lleno

# HERENCIA (SUBCLASES Y SUPERCLASES)

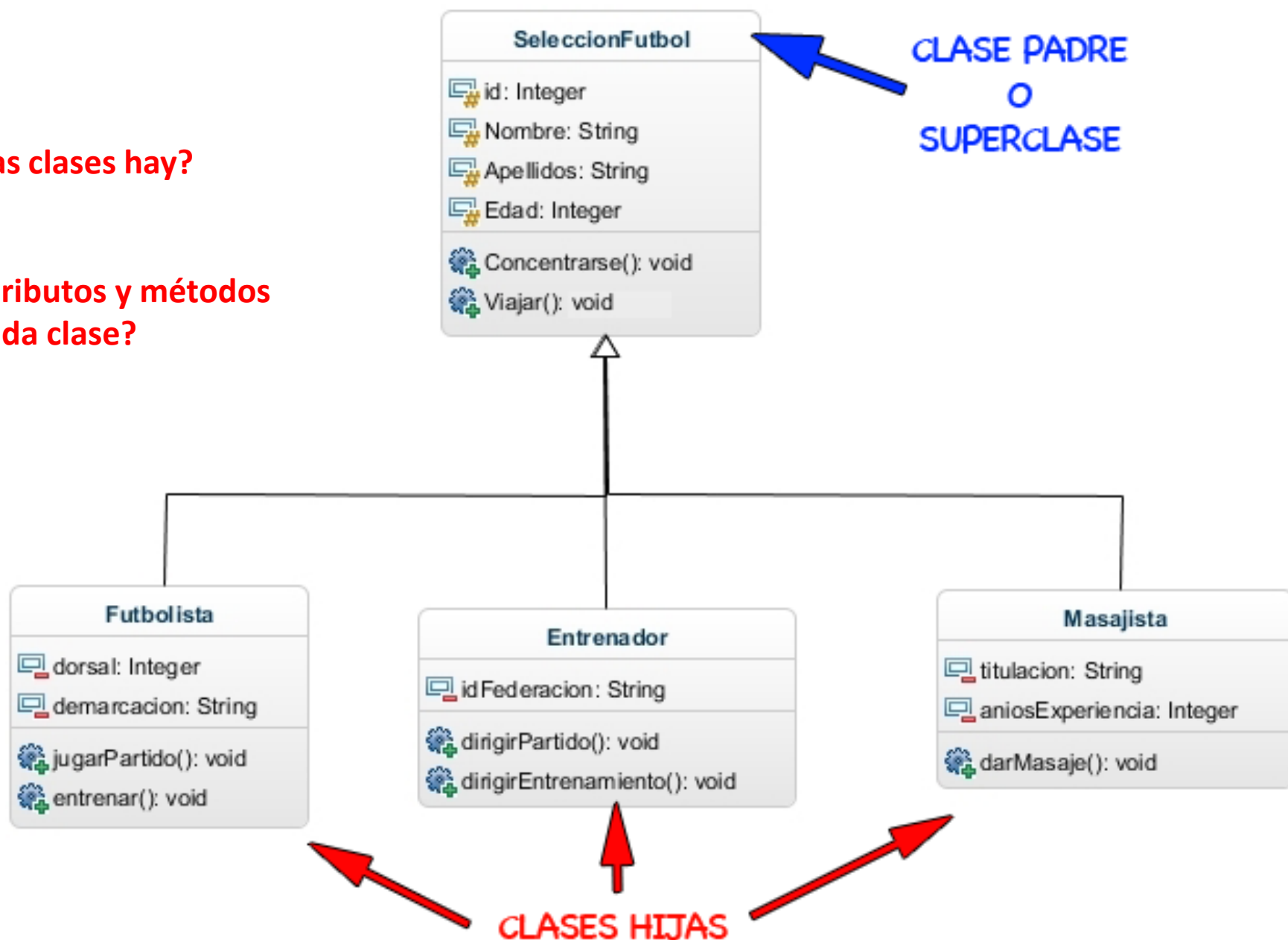
Permite crear nuevas clases basadas en clases existentes.

La clase que hereda se llama subclase. (HIJA)

La clase de la que hereda se llama superclase. (PADRE)

¿Cuántas clases hay?

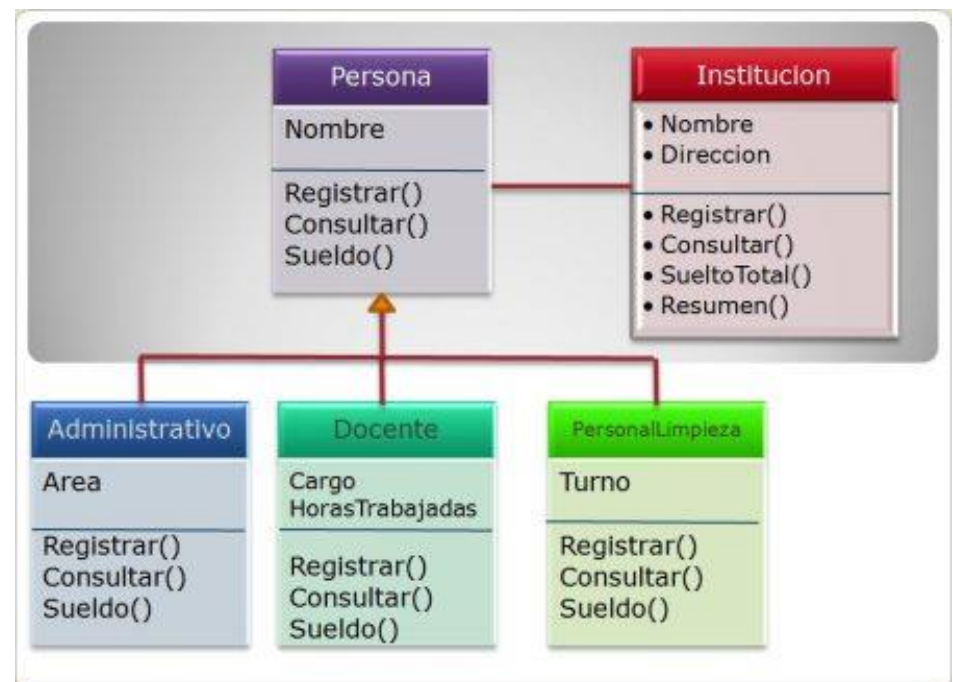
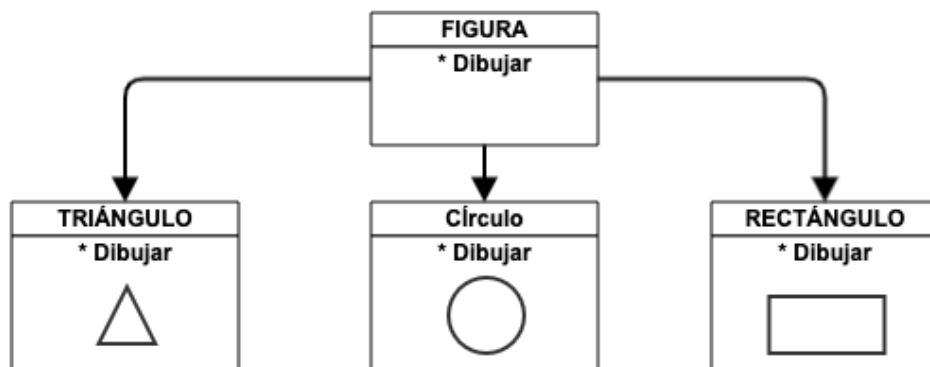
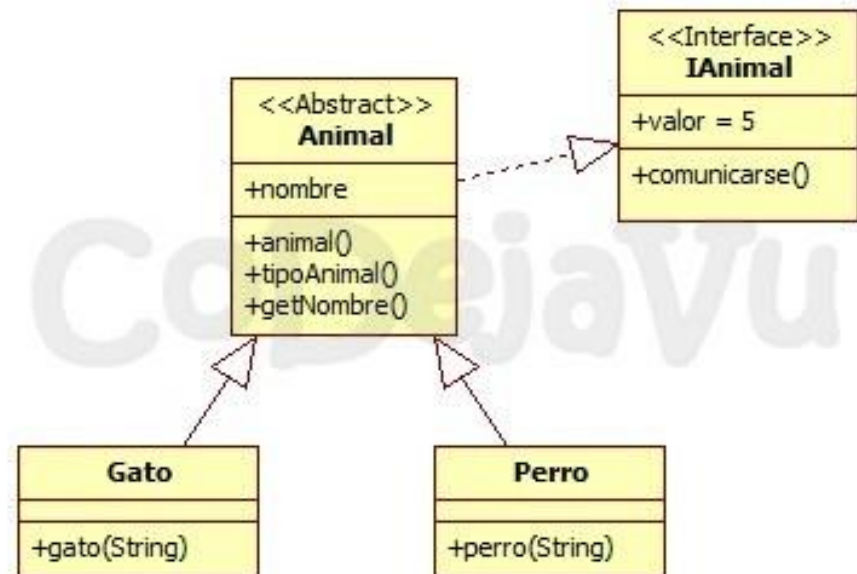
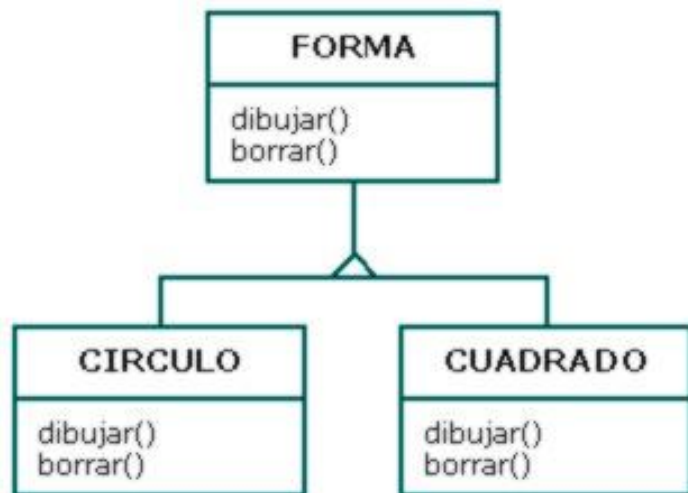
¿Qué atributos y métodos tiene cada clase?



# POLIMORFISMO

Permite que una misma operación se ejecute de diferentes formas en distintas clases.

Se manifiesta principalmente a través de la sobrecarga y la sobrescritura de métodos.



# ENCAPSULAMIENTO

Es la agrupación de datos (atributos) y métodos que operan sobre los datos en una sola unidad, la clase.

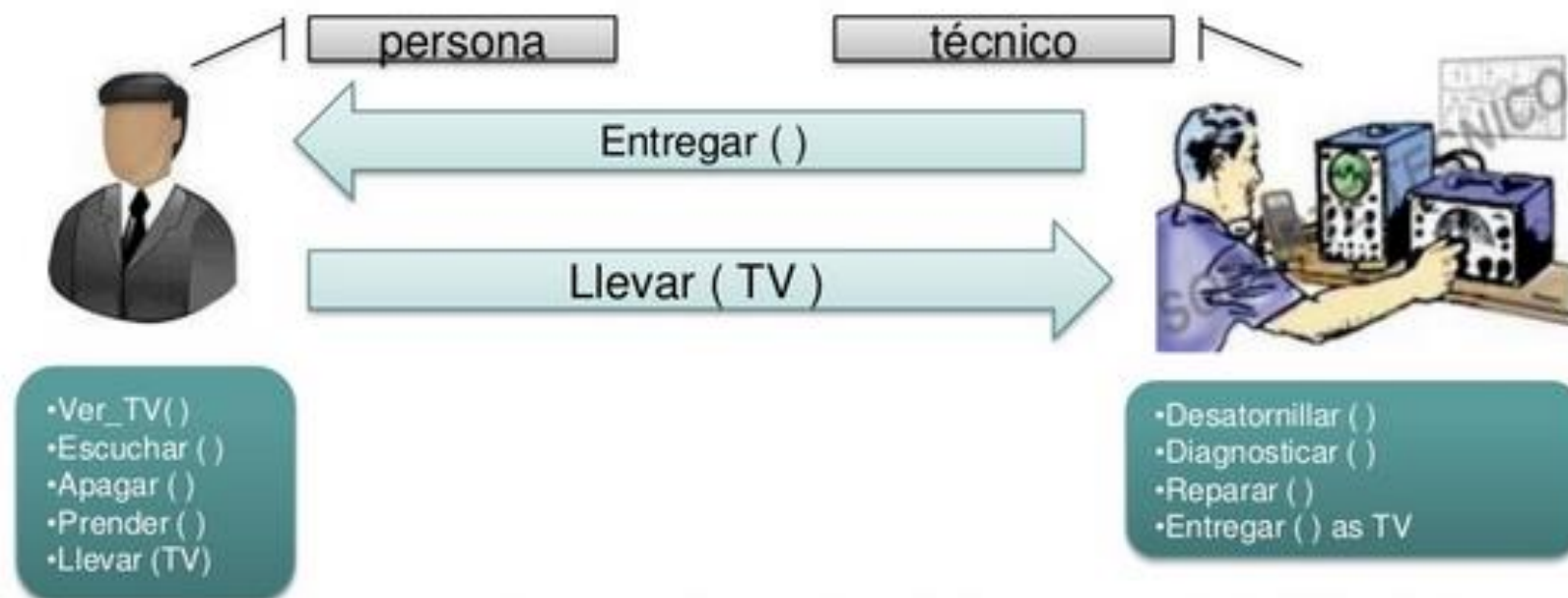
También controla el acceso a los datos mediante modificadores de acceso (privado, protegido, público).



# ENCAPSULAMIENTO



- Esta propiedad permite la ocultación de la información es decir permite asegurar que el contenido de un objeto se pueda ocultar del mundo exterior dejándose ver lo que cada objeto necesite hacer publico.
- **Ejemplo:** Una persona desea llevar su televisor descompuesto para que sea arreglado por un técnico.



¿Qué es público y privado en este ejemplo?



# ABSTRACCION

Es la capacidad de definir las características esenciales de un objeto sin incluir detalles innecesarios.

En Java, se implementa mediante clases abstractas e interfaces.



## Otros Conceptos Importantes

- Constructores: Métodos especiales que se utilizan para inicializar objetos.
- Sobrecarga de Métodos: Permite tener múltiples métodos con el mismo nombre pero diferentes parámetros.
- Interfaces: Similar a las clases abstractas, pero todos los métodos son abstractos por defecto. Las clases pueden implementar múltiples interfaces.
- Paquetes: Agrupaciones de clases relacionadas para organizar el código y controlar el acceso.
- Modificadores de Acceso: Controlan la visibilidad de clases, métodos y atributos (public, private, protected).

# MODIFICADORES DE ACCESO PARA CLASES

## Clases de Nivel Superior (Top-Level Classes)

**Public (public):** Accesible desde cualquier otra clase en cualquier paquete.

```
public class MiClasePublica { // Código de la clase }
```

**Default (Sin modificador):** Accesible solo dentro del mismo paquete.

```
class MiClasePorDefecto { // Código de la clase }
```

## Clases Anidadas (Nested Classes)

**Public (public):** Accesible desde cualquier otra clase.

```
public class ClaseExterna { public class ClaseAnidadaPublica { // Código de la clase anidada } }
```

**Protected (protected):** Accesible solo dentro del mismo paquete y por subclases.

```
public class ClaseExterna { protected class ClaseAnidadaProtegida { // Código de la clase anidada } }
```

**Default (Sin modificador):** Accesible solo dentro del mismo paquete.

```
public class ClaseExterna { class ClaseAnidadaPorDefecto { // Código de la clase anidada } }
```

**Private (private):** Accesible solo dentro de la clase contenedora.

```
public class ClaseExterna { private class ClaseAnidadaPrivada { // Código de la clase anidada } }
```

```
public class ClaseExterna {
```

**Public (public):** Accesible desde cualquier otra clase en cualquier paquete.

**Default (Sin modificador):** Accesible solo dentro del mismo paquete.

```
    public class ClaseAnidadaPublica {  
        // Código de la clase anidada pública  
    }
```

**Public (public):** Accesible desde cualquier otra clase.

```
    protected class ClaseAnidadaProtegida {  
        // Código de la clase anidada protegida  
    }
```

**Protected (protected):** Accesible solo dentro del mismo paquete y por subclases.

```
    class ClaseAnidadaPorDefecto {  
        // Código de la clase anidada por defecto  
    }
```

**Default (Sin modificador):** Accesible solo dentro del mismo paquete.

```
    private class ClaseAnidadaPrivada {  
        // Código de la clase anidada privada  
    }
```

**Private (private):** Accesible solo dentro de la clase contenedora.

```
}
```

# MODIFICADORES NO DE ACCESO PARA CLASES (COMPORTAMIENTO ADICIONAL)

**Abstract (abstract):** Una clase abstracta no puede ser instanciada y puede contener métodos abstractos que deben ser implementados por las subclases.

```
public abstract class MiClaseAbstracta { abstract void metodoAbstracto(); }
```

**Final (final):** Una clase final no puede ser subclaseada.

```
public final class MiClaseFinal { // Código de la clase }
```

**Static (static):** Una clase estática es una clase anidada que pertenece a la clase contenedora, no a una instancia particular de la clase.

```
public static class MiClaseContenedora { static class ClaseEstatica { // Código de la clase estática } }
```

Estos son los principales modificadores que se pueden aplicar a las clases en Java, proporcionando diferentes niveles de acceso y comportamiento.

# MODIFICADORES DE ACCESO PARA ATRIBUTOS Y METODOS

**Public (public):** El miembro es accesible desde cualquier otra clase.

```
public int miVariablePublica;  
public void miMetodoPublico() { // Código del método }
```

**Protected (protected):** El miembro es accesible dentro del mismo paquete y desde subclases (incluso si están en diferentes paquetes).

```
protected int miVariableProtegida;  
protected void miMetodoProtegido() { // Código del método }
```

**Default (Sin modificador):** El miembro es accesible solo dentro del mismo paquete.

```
int miVariablePorDefecto;  
void miMetodoPorDefecto() { // Código del método }
```

**Private (private):** El miembro es accesible solo dentro de la misma clase.

```
private int miVariablePrivada;  
private void miMetodoPrivado() { // Código del método }
```

```
public class MiClase {
```

```
    public int variablePublica;  
    protected int variableProtegida;  
    int variablePorDefecto;  
    private int variablePrivada;
```

```
    public void metodoPublico() { // Código del método }
```

```
    protected void metodoProtegido() { // Código del método }
```

```
    void metodoPorDefecto() { // Código del método }
```

```
    private void metodoPrivado() { // Código del método }
```

**Public (public):** El miembro es accesible desde cualquier otra clase.

**Protected (protected):** El miembro es accesible dentro del mismo paquete y desde subclases (incluso si están en diferentes paquetes).

**Default (Sin modificador):** El miembro es accesible solo dentro del mismo paquete.

**Private (private):** El miembro es accesible solo dentro de la misma clase.



# MODIFICADORES PARA COMPORTAMIENTO ADICIONAL DE ATRIBUTOS Y METODOS

**Static (static):** El miembro pertenece a la clase en lugar de a las instancias de la clase.

```
public static int miVariableEstatica;  
public static void miMetodoEstatico() { // Código del método }
```

**Final (final):** La variable no puede ser reasignada (constante) o el método no puede ser sobrescrito por subclases.

```
public final int miVariableConstante = 10;  
public final void miMetodoFinal() { // Código del método }
```

**Abstract (abstract):** El método no tiene implementación y debe ser implementado por las subclases. Las clases que contienen métodos abstractos deben ser declaradas como abstract.

```
public abstract class MiClaseAbstracta { public abstract void miMetodoAbstracto(); }
```

# MODIFICADORES PARA COMPORTAMIENTO ADICIONAL DE ATRIBUTOS Y METODOS

**Synchronized (synchronized):** El método puede ser accedido por un solo hilo a la vez.

```
public synchronized void miMetodoSincronizado() { // Código del método }
```

**Transient (transient):** La variable no será serializada.

```
private transient int miVariableTransitoria;
```

**Volatile (volatile):** La variable es modificada asincrónicamente por varios hilos.

```
private volatile boolean miVariableVolatil;
```

```
public class MiClase {
```

```
    public int variablePublica;  
    protected int variableProtegida;  
    int variablePorDefecto;  
    private int variablePrivada;  
    public static int variableEstatica;  
    public final int variableConstante = 10;  
    private transient int variableTransitoria;  
    private volatile boolean variableVolatil;
```

```
    public void metodoPublico() { // Código del método }
```

```
    protected void metodoProtegido() { // Código del método }
```

```
    void metodoPorDefecto() { // Código del método }
```

```
    private void metodoPrivado() { // Código del método }
```

```
    public static void metodoEstatico() { // Código del método }
```

```
    public final void metodoFinal() { // Código del método }
```

```
    public synchronized void metodoSincronizado() { // Código del método }
```

```
    private volatile boolean miVariableVolatil;
```

**Public (public):** El miembro es accesible desde cualquier otra clase.

**Protected (protected):** El miembro es accesible dentro del mismo paquete y desde subclases (incluso si están en diferentes paquetes).

**Default (Sin modificador):** El miembro es accesible solo dentro del mismo paquete.

**Private (private):** El miembro es accesible solo dentro de la misma clase.

**Static (static):** El miembro pertenece a la clase en lugar de a las instancias de la clase.

**Final (final):** La variable no puede ser reasignada (constante) o el método no puede ser sobrescrito por subclases.

**Abstract (abstract):** El método no tiene implementación y debe ser implementado por las subclases. Las clases que contienen métodos abstractos deben ser declaradas como abstract.

**Synchronized (synchronized):** El método puede ser accedido por un solo hilo a la vez.

**Transient (transient):** La variable no será serializada.

**Volatile (volatile):** La variable es modificada asincrónicamente por varios hilos.

# SOBRECARGA DE METODOS - OVERLOADING

```
public class Ejemplo {
```

```
    // Método sobrecargado
```

```
    public int suma(int a, int b) {  
        return a + b;  
    }
```

```
    // Método sobrecargado con diferente tipo de parámetros
```

```
    public double suma(double a, double b) {  
        return a + b;  
    }
```

```
    // Método sobrecargado con diferente número de  
    parámetros
```

```
    public int suma(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```

La sobrecarga de métodos es la capacidad de tener **múltiples métodos con el mismo nombre en una clase, pero con diferentes listas de parámetros.**

Los métodos sobrecargados tienen el mismo nombre pero diferentes firmas, lo que significa que pueden tener un número diferente de parámetros, tipos de parámetros diferentes o ambos.

# **SOBREESCRITURA DE METODOS - OVERRIDING**

```
public class Animal {  
    public void hacerSonido() {  
        System.out.println("Sonido de animal");  
    }  
}
```

```
public class Perro extends Animal {
```

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

La sobreescritura de métodos ocurre cuando una subclase proporciona una implementación específica de un método que ya está definido en su clase base (superclase).

La subclase puede cambiar la implementación del método de la superclase para adaptarse a sus necesidades específicas.

Para que la sobreescritura ocurra, el método en la subclase debe tener el mismo nombre, tipo de retorno y lista de parámetros que el método en la superclase.

**Sobrecarga de métodos →**

múltiples métodos con el mismo nombre pero diferentes parámetros en una clase

**Sobreescritura de métodos →**

implementación específica de un método en una subclase que anula la implementación en su clase base

# PAQUETES

Los paquetes en Java son una forma de organizar y estructurar el código fuente, proporcionando un espacio de nombres para evitar conflictos de nombres y facilitar la gestión del código en proyectos de cualquier tamaño.

- **Organización estructurada del código.**
- **Evitar conflictos de nombres.**
- **Facilitar la gestión y la navegación del código fuente.**
- **Mejorar la reutilización y la mantenibilidad del código.**

```
import java.util.Scanner;
```

PAQUETES JAVA



```
class MyClass {  
    public static void main(String[] args) {  
        Scanner myObj = new Scanner(System.in);  
        System.out.println("Enter username");  
  
        String userName = myObj.nextLine();  
        System.out.println("Username is: " + userName);  
    }  
}
```

PAQUETES DE  
USUARIO



```
package mypack;  
class MyPackageClass {  
    public static void main(String[] args) {  
        System.out.println("This is my package!");  
    }  
}
```



# LIBRERIAS

## DE JAVA

En Java, una biblioteca (o librería) es un conjunto de clases y recursos predefinidos que proporcionan funcionalidades adicionales que pueden ser utilizadas por otros programas.

Estas clases y recursos se empaquetan en archivos **JAR (Java ARchive)** para su distribución y uso.

## DE USUARIO

Además de las bibliotecas estándar y las bibliotecas de terceros mantenidas por organizaciones o comunidades, también puedes crear tus propias bibliotecas en Java para ser reutilizadas en tus proyectos o compartidas con otros usuarios.

Crear una biblioteca de usuario en Java implica organizar y empaquetar tus clases y recursos en un **archivo JAR** para que otros puedan importar y utilizar tus funcionalidades en sus propios proyectos.

Esto puede ser útil si tienes componentes de código que utilizas en múltiples proyectos o si deseas compartir funcionalidades específicas con otros desarrolladores.

# PAQUETES Y LIBRERIAS

## Librerías y Paquetes en Java


<https://www.youtube.com/watch?v=7rDODJ709Jk>

## Crea tus propias librerías con JAVA

<https://www.youtube.com/watch?v=Erj31Ed1F90>

# CONSTANTES

```
public class Constantes {  
    // Definición de constantes para los días de la semana  
    public static final String LUNES = "Lunes";  
    public static final String MARTES = "Martes";  
    public static final String MIERCOLES = "Miércoles";  
    public static final String JUEVES = "Jueves";  
    public static final String VIERNES = "Viernes";  
    public static final String SABADO = "Sábado";  
    public static final String DOMINGO = "Domingo";  
  
    public static void main(String[] args) {  
        // Utilización de las constantes  
        System.out.println("El primer día de la semana es: " + LUNES);  
        System.out.println("El tercer día de la semana es: " + MIERCOLES);  
        System.out.println("El último día de la semana es: " + DOMINGO);  
    }  
}
```



# CONSTRUCTORES

```
public class Persona {  
    // Atributos  
    private String nombre;  
    private int edad;  
    private String profesion;  
  
    // Constructor  
    public Persona(String nombre, int edad, String profesion) {  
        this.nombre = nombre;  
        this.edad = edad;  
        this.profesion = profesion;  
    }  
  
    // Método para imprimir información sobre la persona  
    public void imprimirInformacion() {  
        System.out.println("Nombre: " + nombre);  
    }  
}
```

# INTERFACES

Una interfaz en Java es un contrato que define un **conjunto de métodos abstractos**.

Las interfaces **no** pueden contener implementación de métodos (excepto métodos predeterminados desde Java 8).

Proveen una forma de lograr la abstracción y múltiples herencias.

Los métodos no los implementa el interfaz, **los implementan otras clases**.

Una clase que implementa una interfaz debe proporcionar implementaciones para **todos** los métodos de la interfaz.

## DEFINICION DE LA INTERFAZ

```
public interface Operaciones {  
    void sumar(int a, int b);  
    void restar(int a, int b);  
}
```

## IMPLEMENTACIÓN DE LA INTERFAZ

```
public class Calculadora implements Operaciones {  
    @Override  
    public void sumar(int a, int b) {  
        System.out.println("Suma: " + (a + b));  
    }  
  
    @Override  
    public void restar(int a, int b) {  
        System.out.println("Resta: " + (a - b));  
    }  
}
```

## USO DE LA INTERFAZ

```
public class Main {  
    public static void main(String[] args) {  
        Operaciones calculadora = new Calculadora();  
        calculadora.sumar(5, 3); calculadora.restar(10, 4); } }
```

# ANOTACIONES

Las anotaciones en Java son una forma de agregar **metadatos** al código.

Estos metadatos pueden ser utilizados por el compilador, herramientas de desarrollo y frameworks para diversas tareas, como generación de código, configuración y documentación.

Las anotaciones proporcionan una manera poderosa y flexible de influir en el comportamiento del programa sin introducir lógica de negocio en el propio código.

Existen anotaciones predefinidas en Java y se pueden definir también por el usuario.

# ARCHIVOS JAR

Los archivos JAR (Java ARchive) son archivos de archivo que permiten agrupar múltiples archivos de clase Java y otros recursos (como imágenes, archivos de texto, y archivos de propiedades) en un solo archivo comprimido.

Los archivos JAR facilitan la distribución y la implementación de aplicaciones y bibliotecas Java, ya que encapsulan todos los componentes necesarios en un solo archivo.

Características:

- **Compresión:** Los archivos JAR están comprimidos utilizando el formato ZIP, lo que reduce el tamaño del archivo y facilita su transferencia.
- **Portabilidad:** Los archivos JAR son independientes de la plataforma, lo que significa que se pueden ejecutar en cualquier entorno que tenga una Máquina Virtual de Java (JVM).
- **Agrupación de Recursos:** Un archivo JAR puede contener clases, interfaces, bibliotecas, imágenes, archivos de sonido y otros recursos necesarios para la aplicación.
- **Manifest:** Cada archivo JAR puede contener un archivo especial llamado META-INF/MANIFEST.MF que puede contener metadatos sobre el contenido del JAR, como la clase principal de una aplicación ejecutable.