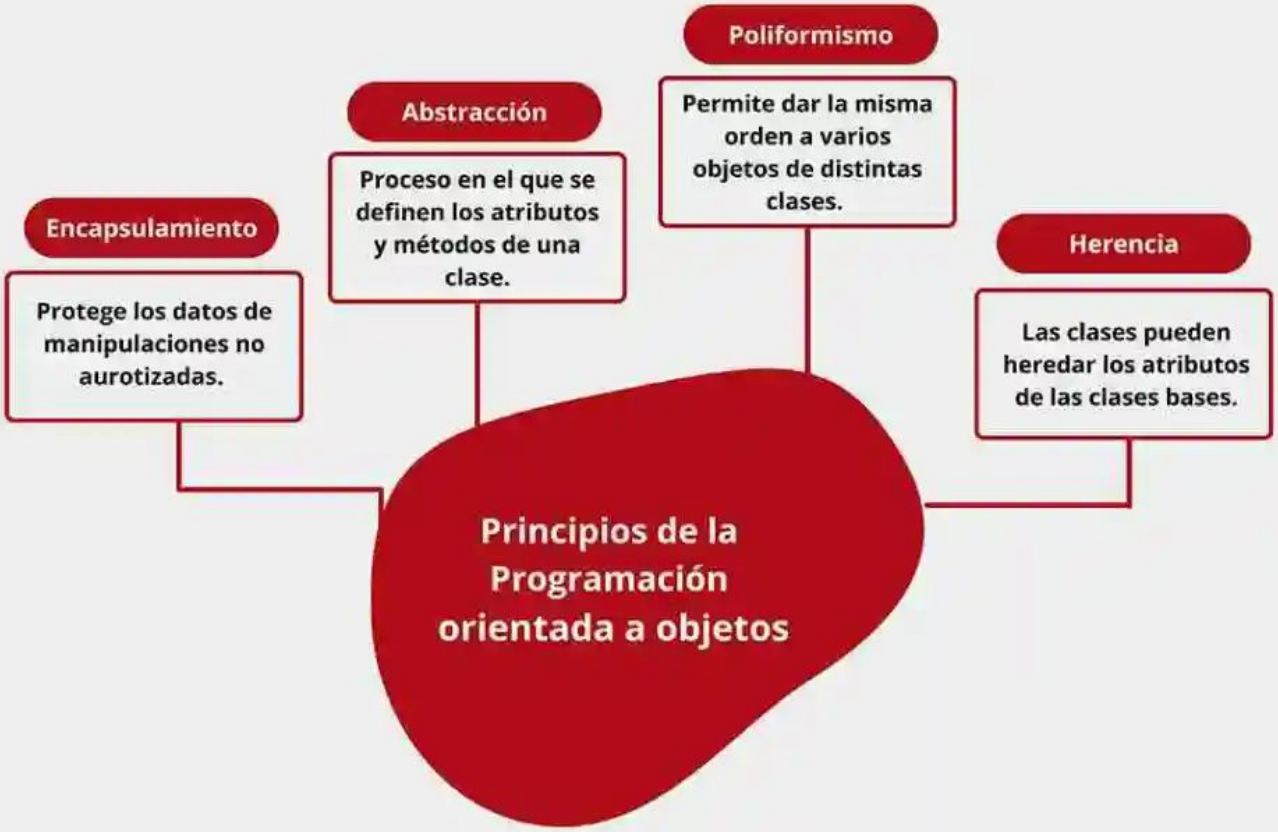
Orientación a Objetos

1º DAM

Características Orientación a Objetos



Características Orientación a Objetos

1. Abstracción

La abstracción es uno de los pilares fundamentales de la programación orientada a objetos: nos permite modelar el mundo real en términos de código, eligiendo **qué propiedades y comportamientos son esenciales** y cuáles pueden ser ignorados.

En la programación, la abstracción se logra mediante la creación de clases (plantillas) y objetos (instancias con valores concretos).

Estas clases sirven como plantillas o modelos tanto la información (atributos) como el comportamiento (métodos).

Características Orientación a Objetos

2. Encapsulación

La encapsulación permite **ocultar la complejidad interna** de una clase, proporcionando una interfaz controlada para interactuar con los objetos de esa clase. Esto mejora la seguridad y la integridad de los datos, ya que solo se pueden modificar a través de métodos específicos de la clase, permitiendo así una validación adecuada antes de realizar cualquier cambio.

Encapsulación

La encapsulación permite **ocultar la complejidad interna** de una clase, proporcionando una interfaz controlada para interactuar con los objetos de esa clase.

Para ocultar propiedades o métodos en Java, se disponen niveles de visibilidad:

- **public**: nos indica que es accesible desde cualquier clase o interfaz
- **private**: sólo es accesible desde la clase actual.
- **protected**: accesible desde la clase actual, sus descendientes o el paquete del que forma parte.
- **sin ninguna palabra**: accesible desde cualquier clase del paquete.

Encapsulación

Un ejemplo común de encapsulación en Java es:

- tener **atributos privados**
- proporcionar **métodos públicos** de acceso, conocidos como **“getters” y “setters”,** para leer y modificar esos atributos de forma controlada.

Este enfoque garantiza que el acceso a los datos sea a través de métodos que puedan aplicar validaciones o lógica adicional antes de realizar cualquier modificación.

Encapsulación

```
public class Coordenada {
    private double coordenadaX, coordenaY;

    public Coordenada() { //Constructor por defecto
        super();
        this.coordenadaX = 0;
        this.coordenaY = 0;     }

    public Coordenada (double coordenadaX, double coordenaY)
{//Constructor por parámetros
        super();
        this.coordenadaX = coordenadaX;
        this.coordenaY = coordenaY;    }

//Métodos get → obtener atributo
    public double getCoordenadaX() {
        return coordenadaX; }
//Método set → modificar valor atributo
    public void setCoordenadaX(double coordenadaX) {
        this.coordenadaX = coordenadaX;
    } ...}
```

Características de la Orientación a Objetos

3. Herencia

Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase.

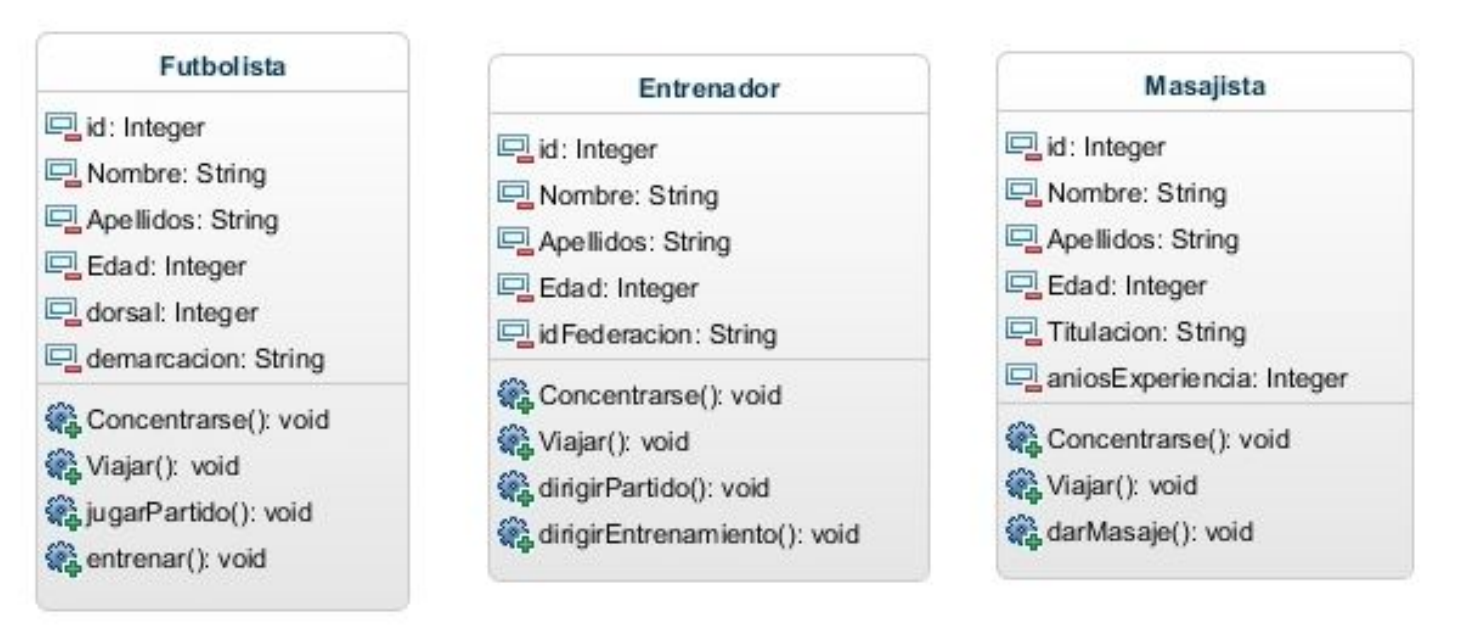
La herencia permite que se puedan definir **nuevas clases (subclases) basadas de unas ya existentes (superclases)** a fin de reutilizar el código, generando así una jerarquía de clases dentro de una aplicación.

Si una clase deriva (**extends**) de otra, esta hereda sus atributos y métodos y puede añadir nuevos atributos, métodos o redefinir los heredados.

Herencia

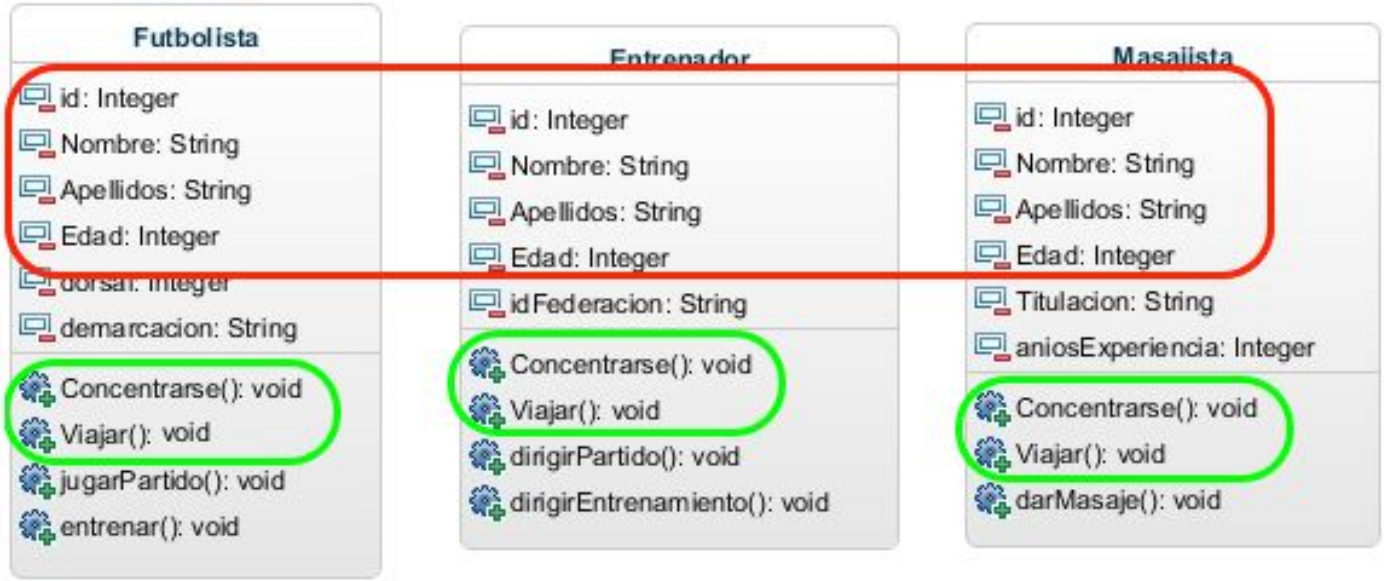
Si una clase, ClaseHija, deriva o **extiende** a otra, ClasePadre, quiere decir que esta clase recibe todos los atributos y métodos heredables de la clase padre.
Luego veremos qué significa **heredables**.

Es tu turno: Codifica estas tres clases.



Herencia

En estas clases hay muchas partes comunes. Por lo que habría mucho código triplicado.



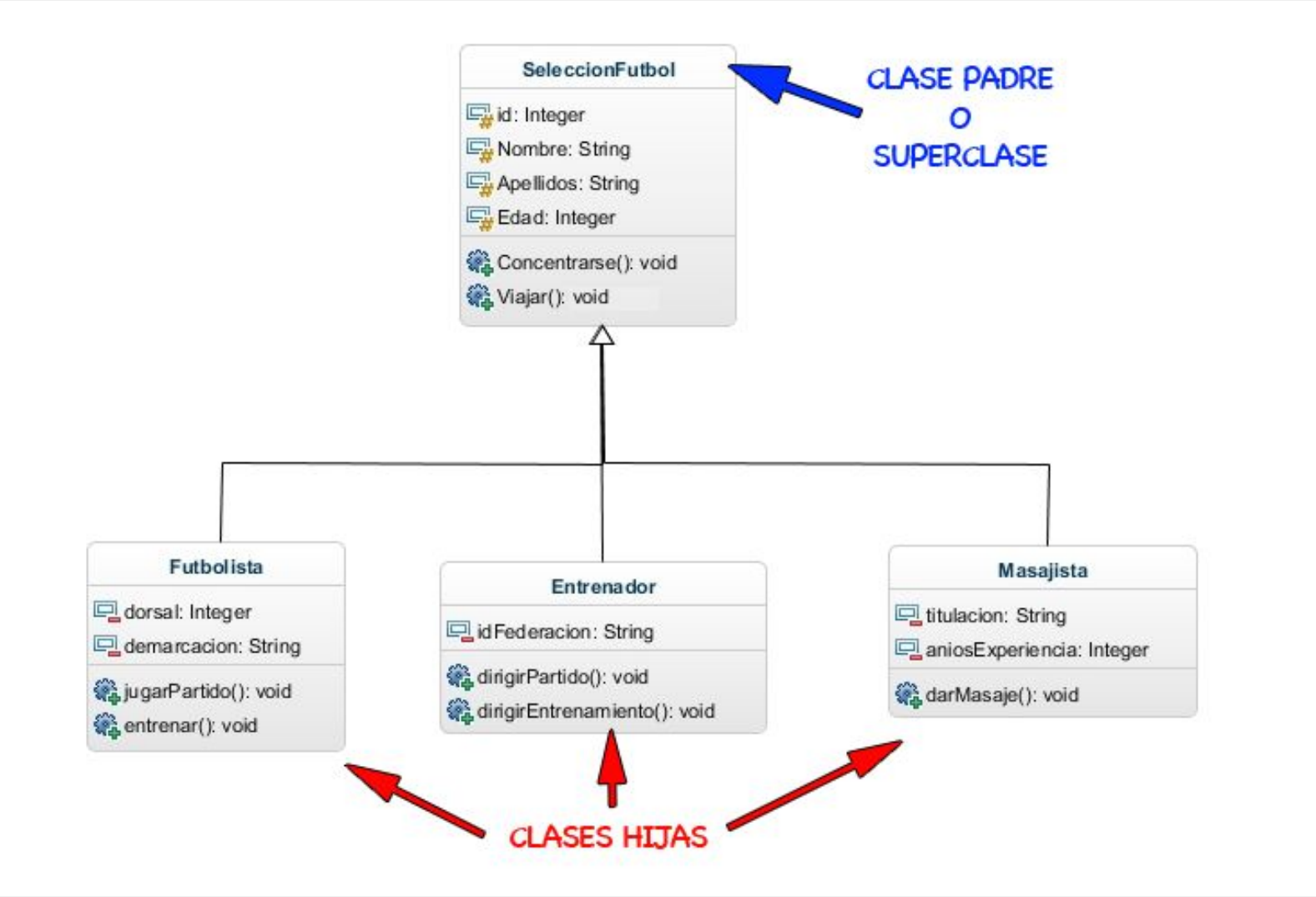
Herencia

La herencia nos permitirá:

- Construir una clase padre en la que pondremos todo lo común o genérico
- Construir tres clases hijas que extenderán a la clase padre y tendrán lo específico, lo que la diferencia de su padre y del resto de clases hijas.

```
public class ClaseHija extends ClasePadre
{
...
}
```

Herencia


SeleccionFutbol
id: Integer
Nombre: String
Apellidos: String
Edad: Integer
Concentrarse(): void
Viajar(): void
CLASE PADRE
O
SUPERCLASE
Futbolista
dorsal: Integer
demarcacion: String
jugarPartido(): void
entrenar(): void
Entrenador
idFederacion: String
dirigirPartido(): void
dirigirEntrenamiento(): void
Masajista
titulacion: String
aniosExperiencia: Integer
darMasaje(): void
CLASES HIJAS

Herencia

Para el ejemplo anterior:

```
public class Persona
{
 String nombre;
 String apellidos;
 int edad;
 float salario;
 public void concentrarse() { … }
 public void viajar() { … }
}
```

En la clase Padre → Lo General

Entrenador hereda los atributos y los métodos del padre y, además, define los suyos propios

```
public class Entrenador extends Persona {
 String idFederacion;
 public void jugarPartido() { … }
 public void entrenar() { … }
}
```

En la clase Hija → Lo Específico

Herencia

Herencia Simple y Jerarquía de Clases

En Java la herencia es simple, lo que quiere decir que una clase sólo puede extender a una única clase.

NO siendo posible definir una clase con dos padres.

```
public class Entrenador
extends Persona, DeportistaProfesional {


}
```

Esto sí es posible en otros lenguajes de programación.

Sin embargo, sí que puede establecerse una jerarquía de clases.

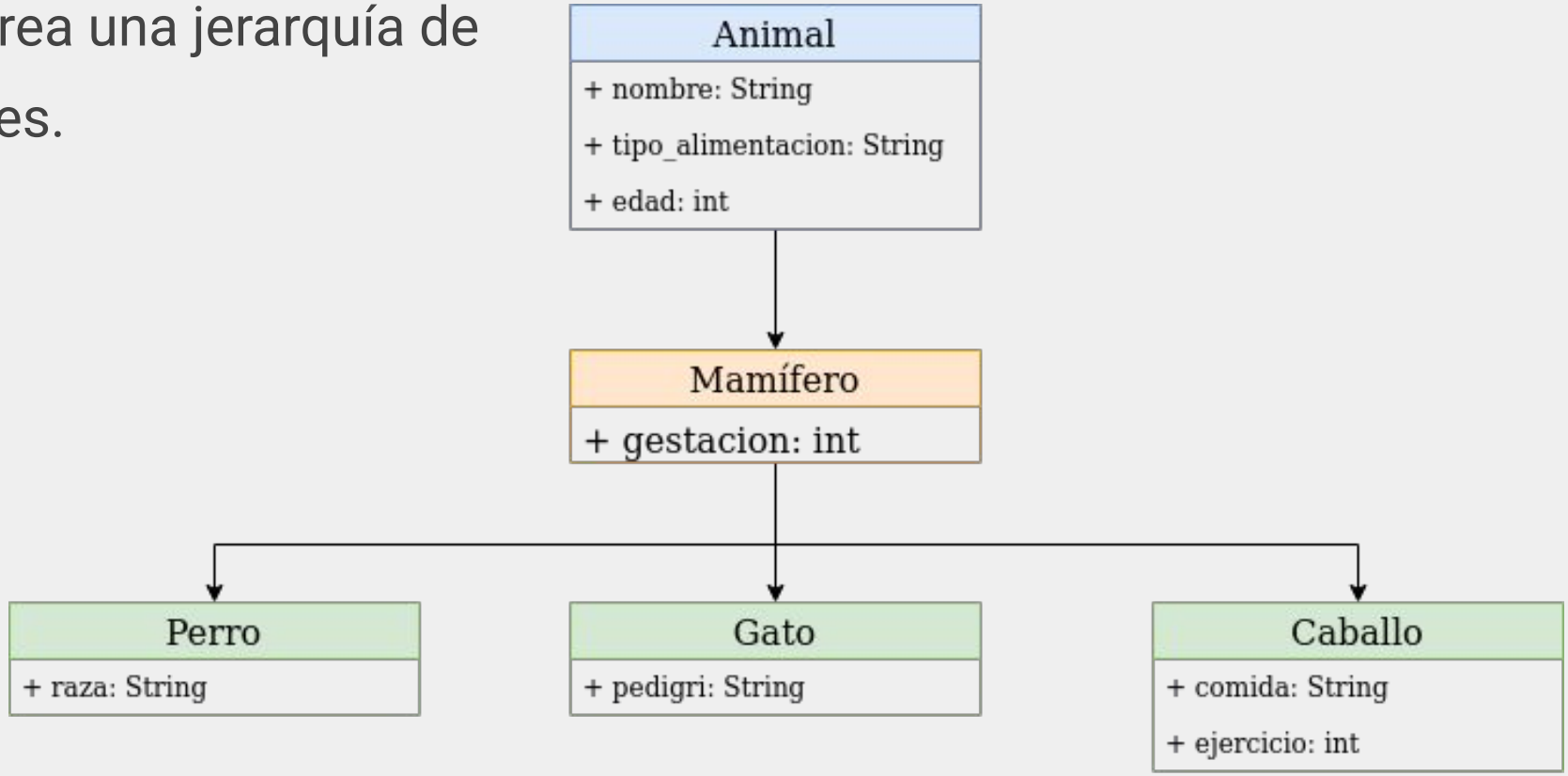
Herencia

Herencia Simple y Jerarquía de Clases

Una clase Hija (Mamífero), que hereda de un padre (Animal) puede ser a su vez clase padre para otra tercera clase (Perro).

La clase nieta heredará de su padre y de su abuelo.

Se crea una jerarquía de clases.



Herencia

Herencia Simple y Jerarquía de Clases

Es tu turno.

1. Codifica el diagrama de clases anterior.
2. Diseña y construye una clase Vertebrado y prueba a definir la clase Perro como hija de Mamífero y de Vertebrado

Herencia

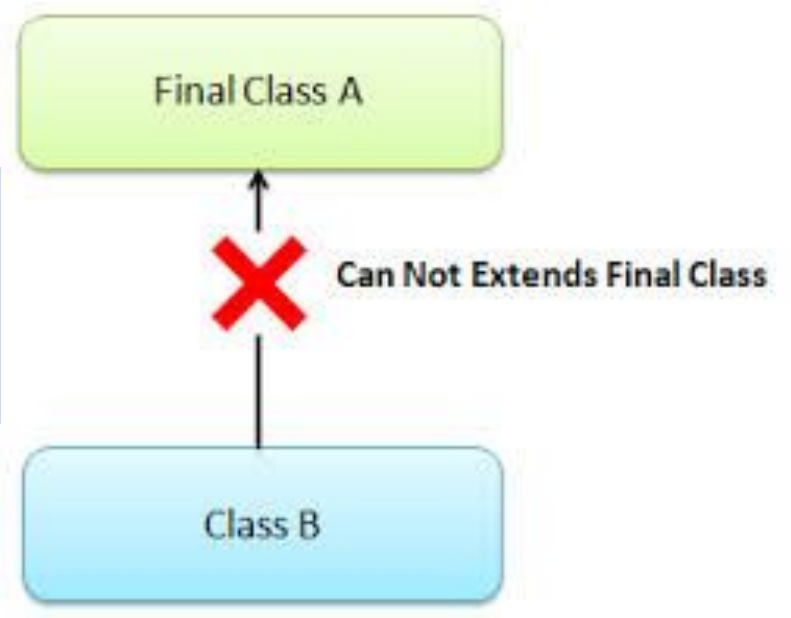
Clases finales

Como hemos visto antes, cualquier clase puede ser extensible por otra/s subclase/s hija/s.

Para evitar que una clase tenga hijos, definiremos esa clase NO padre como final

```
public final class Entrenador extends Persona {
...
}
```

La clase Entrenador no puede ser extendida por ninguna otra clase



Herencia

¿Qué se considera heredable?

Serán heredables según su visibilidad:

Modificador	Visibilidad
`public`	Pública (+)
`protectec`	Protegida / en la herencia(#)
`private`	Privada(-)
`package`	De paquete (~)

Herencia

¿Qué se considera heredable?

Visibilidad	Public	Protected	Default	Private
Desde la misma Clase	SI	SI	SI	SI
Desde cualquier Clase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase fuera del mismo Paquete	SI	SI, a través de la herencia	NO	NO
Desde cualquier Clase fuera del Paquete	SI	NO	NO	NO

Características Orientación a Objetos

4. Polimorfismo

Es la capacidad que tienen los objetos de la clase de responder a un mismo mensaje o evento de manera diferente según los parámetros utilizados en la llamada.

Esto se consigue de varias formas:

- **Paramétrico**: Existen funciones con el mismo nombre pero se usan diferentes parámetros (nombre o tipo).
- **Sobrecarga**: una subclase define un método que existe en una superclase con una lista de argumentos

Herencia

Sobrecarga u overloading

En este caso vamos a tener métodos dentro de una misma clase que se llaman igual pero que reciben distintos parámetros. En función de lo que reciba, se llamará a un método y a otro. A esto se le llama sobrecarga.

Por ejemplo:

```
public class Persona {
    String nombre;
    String apellidos;
    int edad;
    float salario;

    public void concentrarse() {
        System.out.println("método concentrarse del padre");
    }

    //Sobrecarga
    public void concentrarse(String lugar) {
        System.out.println("método concentrarse del padre con String "+ lugar);
    }
```

Herencia

Sobreescritura u override

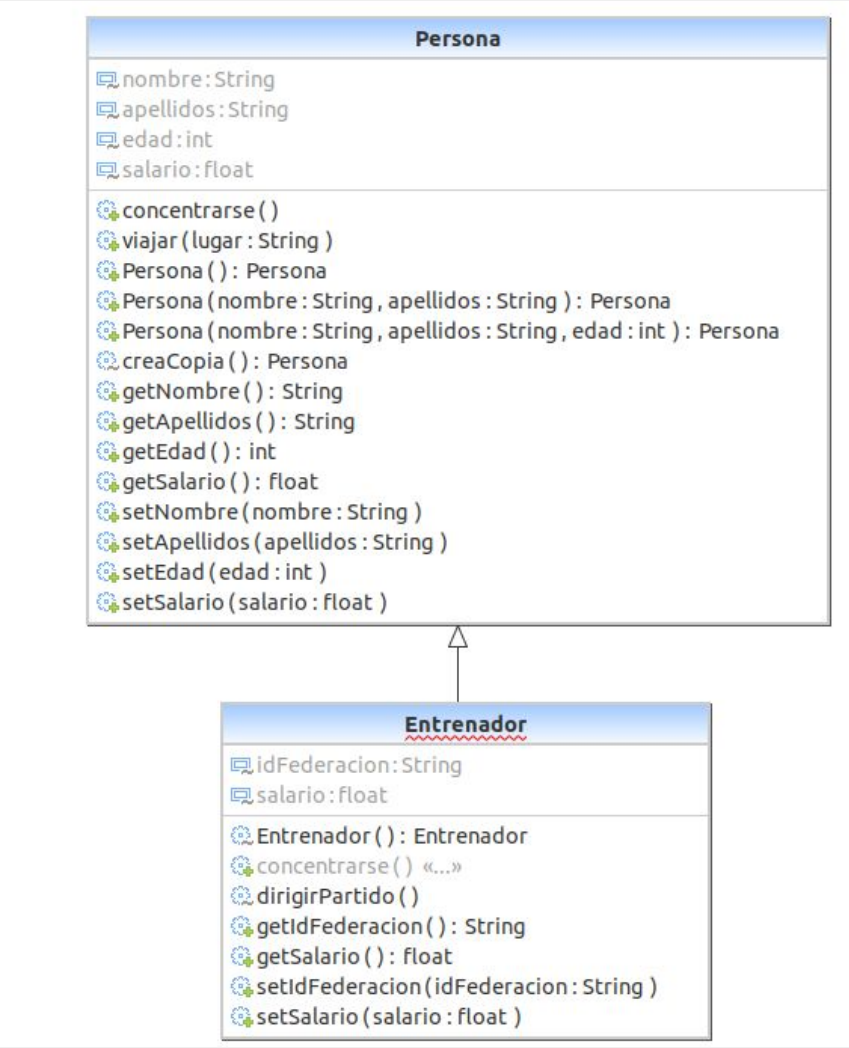
Una subclase puede querer modificar el comportamiento de uno o varios de los métodos que recibe de su padre. Puede ocurrir a dos niveles:

- A nivel de atributo —> Ocultación. Se declara un atributo que se llama igual que el que recibe heredado del padre. El hijo lo sustituye por el hijo.
- A nivel de método → Sustitución u overriding. Un método que está definido en su padre, se implementa de diferente manera.

Veamos estos dos conceptos con un ejemplo.

Herencia

Sobreescritura u override



Herencia

Sobreescritura u override

```
public class Persona {
    String nombre;
    String apellidos;
    int edad;
    float salario;

    public void concentrarse() {
        System.out.println("método concentrarse del padre");
    }

    public float getSalario() {
        System.out.println("getSalario padre");
        return salario;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
```

Herencia

Sobreescritura u override

```
public class Entrenador  extends Persona {
    String idFederacion;
    float salario; // ocultación del atributo salario del padre

    //Sobreescribimos el método concentrarse que hereda de su padre
    @Override
    public void concentrarse() {
        System.out.println("método concentrarse del hijo");
    }

    public float getSalario() {
        System.out.println("getSalario hijo");
        return salario;
    }
    Entrenador()
    {
        super();
        System.out.println("Llamo a constructor Entrenador");
        this.idFederacion ="";
        this.nombre = "entrenador sin nombre";

    }
    void dirigirPartido()
```

Herencia

Sobreescritura u override

```
public class Entrenador  extends Persona {
    String idFederacion;
    float salario; // ocultación del atributo salario del padre

    //Sobreescribimos el método concentrarse que hereda de su padre
    @Override
    public void concentrarse() {
        System.out.println("método concentrarse del hijo");
    }

    public float getSalario() {
        System.out.println("getSalario hijo");
        return salario;
    }
    Entrenador()
    {
        super();
        System.out.println("Llamo a constructor Entrenador");
        this.idFederacion ="";
        this.nombre = "entrenador sin nombre";

    }
    void dirigirPartido()
```

Herencia

Sobreescritura u override

```
public class Entrenador  extends Persona {
    String idFederacion;
    float salario; // ocultación del atributo salario del padre

    //Sobreescribimos el método concentrarse que hereda de su padre
    @Override
    public void concentrarse() {
        System.out.println("método concentrarse del hijo");
    }

    public float getSalario() {
        System.out.println("getSalario hijo");
        return salario;
    }
    Entrenador()
    {
        super();
        System.out.println("Llamo a constructor Entrenador");
        this.idFederacion ="";
        this.nombre = "entrenador sin nombre";

    }
    void dirigirPartido()
```

Herencia

Sobreescritura u override

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

        Persona p2 = new Persona();
        Persona p1 = new Entrenador();
        Entrenador e = new Entrenador();
        Persona[] personas = new Persona[3];
        personas[0]= p2;
        personas[1]= p1;
        personas[2]= e;

        p2.concentrarse();
        p2.getSalario();

        p1.concentrarse();
        p1.getSalario();

        e.concentrarse();
        e.getSalario();
```

Herencia

Clase Object

La clase Object del paquete java.lang es la superclase de la que derivan en última instancia todas las Clases de Java. Esta herencia se hace por defecto, sin que tenga que aparecer la palabra extends.

De la la clase Object las clases heredan una serie de métodos:

- public String **toString**()
- public boolean **equals**(Object obj)
- public int **hashCode**()
- public **final** Class<?> **getClass()** (this.getClass().getSimpleName();)

Herencia

Clases abstractas

Una clase abstracta es una clase con atributos y métodos en los que puede haber métodos abstractos, es decir métodos que están declarados pero no están definidos. De manera, que sus subclases tendrán que definir la implementación de esos métodos.

Por ejemplo:4

```
public abstract class Figura
{
    private String color;
    public Figura(String color)
    {
        this.color = color;
    }
```

Herencia

Clases abstractas

Una **clase abstracta** es una clase con atributos y métodos en los que puede haber **métodos abstractos**, es decir métodos que están declarados pero no están definidos.

De manera, que sus **subclases,** no finales, tendrán que **definir** la implementación de esos **métodos abstractos obligatoriamente**.

Una clase abstracta puede ser extendida por otra subclase también abstracta. En ese caso, la subclase abstracta no tendrá que implementar los métodos abstractos.

Las clases abstractas **no son instanciables** aunque sí que pueden tener constructor para inicializar los atributos que contienen.

Herencia

Clases abstractas

```
public abstract class Figura
{
    private String color;
    public Figura(String color)
    {
        this.color = color;
    }
    public abstract double calcularArea();
    public String getColor()
    {
        return color;
    }
}
```

Herencia

Clases abstractas

```
public class Cuadradro extends Figura
{
    private double lado;

    public double calcularArea()
    {
        return lado*lado;
    }
}
```

Herencia

Clases abstractas

Es tu turno:

1. ¿Qué ocurre cuando hago esto: Figura f = new Figura();?
2. ¿Puede ser una clase abstract y final?

Herencia

Operador instanceof Y Conversión Casting de tipos

Java proporciona el operador **instanceof** que permite conocer un objeto es de una la clase o no (devuelve booleano)

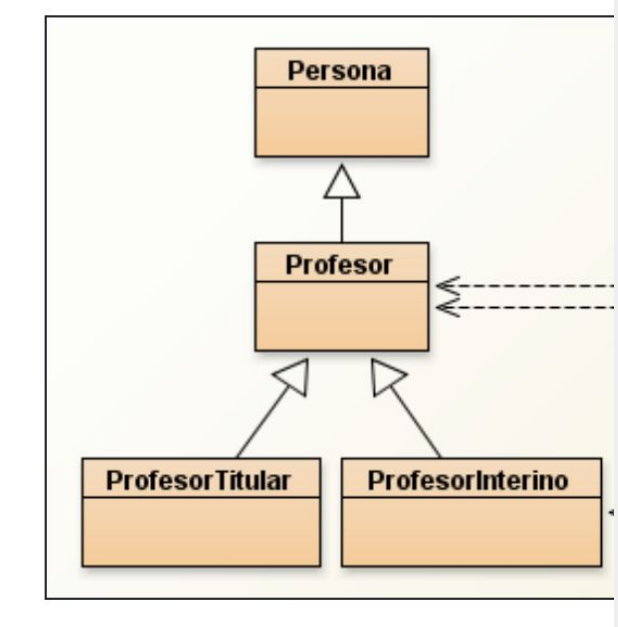
```
if (v instanceof Coche) {
```

Se pueden hacer conversiones o casting a otros tipos.

1. Casting para convertir en una superclase
2. Casting para convertir de una superclase a una subclase. No siempre es posible.

Herencia

Operador instanceof Y Conversión Casting de tipos



1. ProfesorTitular pepe = new ProfesorTitular();

   Persona persona = (Persona) pepe; // Se pierde lo específico

1. Persona persona = new Persona();

   ProfesorTitular pepe = (ProfesorTitular) persona;

Herencia

Operador instanceof Y Conversión Casting de tipos

Es tu turno:

1. Prueba cuándo es posible hacer el casting de un supertipo a un subtipo.

   ¿Qué ocurre cuando no se puede hacer la conversión?

Interfaces

Una interfaz es una plantilla o contrato que todos las clases que la **implementan** debe seguirla. La interfaz puede estar compuesta de:

- Métodos **públicos** sin implementación (todos se declaran implícitamente abstractos)
- **Atributos constantes y públicos**

No está compuesta de:

- Métodos con implementación
- Atributos o variables de instancia

Interfaces

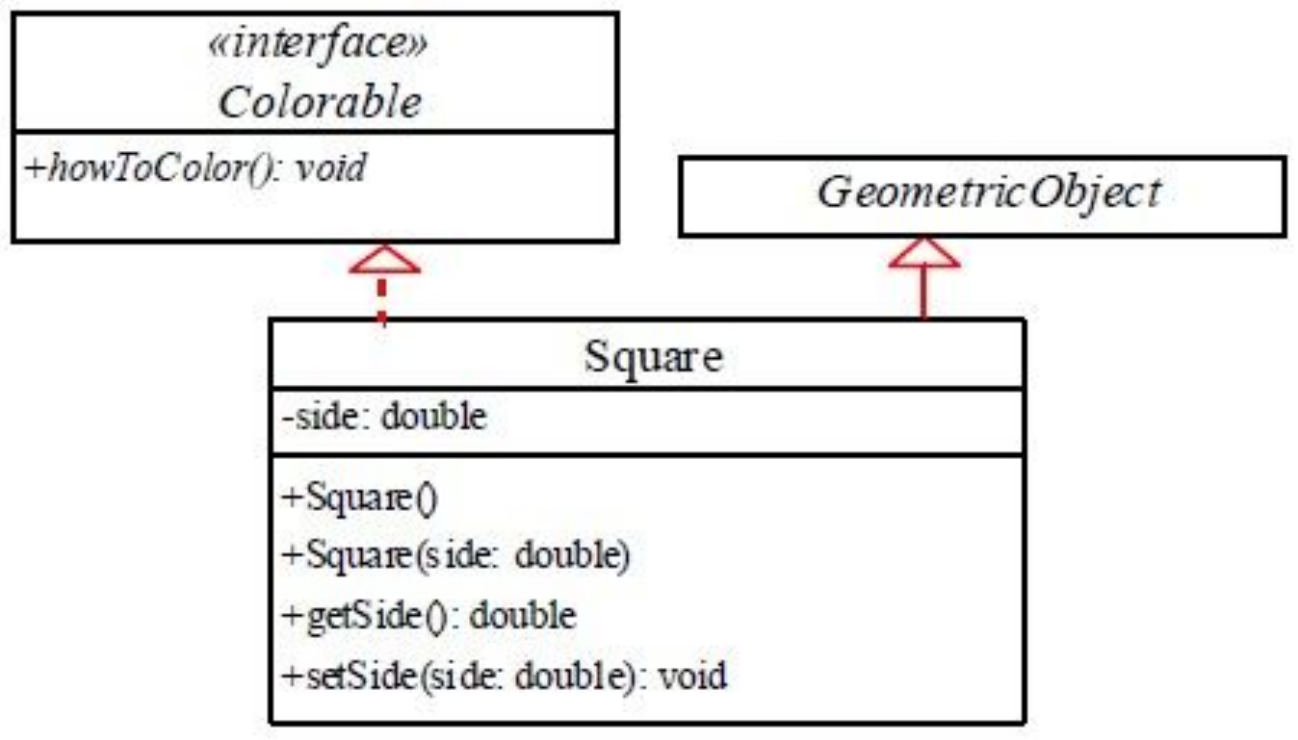
Una interfaz puede implementar a otra interfaz, no teniendo que implementar sus métodos. Las clases que implementen la interfaz tendrá que hacerlo.

A diferencia de lo que pasa con la herencia, una clase SÍ que puede implementar varias interfaces.

De este modo, se consigue en Java la herencia múltiple.

La interfaz es similar a la clase abstracta siendo la interfaz totalmente declarativa sin implementar nada. Como se ha comentado, una interfaz es un simple contrato.

Interfaces


«interface»
Colorable
+howToColor(): void
GeometricObject
Square
-side: double
+Square()
+Square(side: double)
+getSide(): double
+setSide(side: double): void

Interfaces

Declaración de interfaz:

```
public interface IColorable {
 public static String ROJO="rojo";
 void comoColorear();
}
```

Implementación de una interfaz:

```
public class Triangulo implements IColorable{
    @Override
    public void comoColorear() {
        System.out.println("Imprimo Rojo:"+this.ROJO);

    }
}
```

Interfaces

Jerarquía de interfaces:

Una interfaz puede implementar otras interfaces. Por ejemplo, la interfaz IColorable podría implmentar la interfaz IGestionaArea y la interfaz IGestionaPosicion.

```
public interface IColorable
    implements IGestionaArea, IGestionaPosicion {
 public static String ROJO="rojo";
 void comoColorear();
}
```

La clase Triangulo tendrá que implementar todos los métodos que le vienen dados de los contratos de las tres interfaces

Interfaces

Es tu turno:

Codifica el siguiente diagrama de clases considerando que:

- SerVivo es una clase abstacta que tiene un método abstracto llamado estaVivo(): boolean y otro que sea seDesplaza(): boolean y un atributo edad.
- Una clase Animal que extiende a SerVivo que añada un método come() y respira()
- Una interfaz denominada ICanino que tiene un método denominado getRaza():String

Static

Atributos y métodos static

Los elementos estáticos (atributos o métodos) son aquellos que **pertenecen a la clase**, en lugar de pertenecer a un objeto en particular.

Son elementos que **existen dentro de la propia clase** y para acceder los cuales no necesitamos haber creado ningún objeto de esa clase. Osea, en vez de acceder a través de un objeto (no se accede desde el this), accedemos a través del **nombre de la clase.**

Los métodos estáticos NO podrán hacer uso de atributos o métodos de la clase si NO son estáticos.

Static

Atributos y métodos static

```
//Definición
public static precioDefecto = 1.20;

public static int suma (int x, int y){ return x+y;}

//Uso variable
if(precio >= Producto.precioDefecto){ //lo que sea}

//Uso método
public static void main(String[] args)
{
    suma(2,3)
}
```

Static

Constantes de Clase

Cuando queremos crear una constante usamos la palabra final para decir que valor será usado por todos los objetos de la clase y será inmutable.

Si sólo usamos final, estaremos creando una constante para cada instancia de la clase (para cada objeto).

Esto puede no ser muy óptimo en cuanto a memoria, es por ello que convendría que las constantes se crearán asociadas a las clases y no asociadas a cada instancia.

```
//Declaración

public static final double PI=3.14;

//Uso desde fuera de la propia clase
System.out.println(NombreClaseEnLaQueSeDefine.PI);
```

Static

Bloque static

El bloque static es un área de inicialización global de clase.

Al declararlo, se **ejecuta** cuando se **accede por primera vez a una clase**, para instanciarla o para usar un método o propiedad estático de la misma.

Declaración:

```
static{
    //código a ejecutar de tipo estático
}
```


¿Dudas?