

**IES**

**Francisco de Goya**

1º DAW. Programación



***Unidad de Trabajo 6: Uso de clases y objetos.***

## ***Introducción: objetos, atributos y métodos.***

- ✗ La Programación Orientada a Objetos intenta desarrollar programas que representen **objetos en el “mundo real”**.
- ✗ En el mundo real podemos encontrar **objetos** como **coches, videoconsolas, libros, humanos, animales...**
- ✗ Cada objeto tiene características (campos o atributos), ej:
  - **Vehículo** tiene: marca, modelo, potencia, color...
  - **Videoconsola** tiene: marca, modelo, disco duro, unidad de disco...
  - **Libro** tiene: título, autor, páginas...
- ✗ El objeto tiene métodos (funciones) para realizar acciones:
  - **Vehículo** puede: arrancar el motor, frenar, acelerar, encender luces...
  - **Videoconsola** puede: encender, jugar, apagar...
- ✗ Tenemos clases: definen los atributos y métodos de los coches, de las videoconsolas, de los libros, de los humanos...

## *Introducción: clases y objetos.*

### ✗ Tenemos clases:

- Son como las especies o los tipos de los objetos.
- Definen los atributos y métodos de todos los coches, videoconsolas, libros, humanos, animales...

### ✗ Tenemos objetos:

- Tienen los valores específicos para los atributos de este objeto.
- Son un individuo específico de una clase, por ejemplo:
- Para los coches de la clase un objeto podría ser VW Golf, otro Peugeot 206, otro Seat 600...
- Para las videoconsolas de clase un objeto puede ser Play Station 5, otra Xbox Series S, Nintendo switch...
- Para los libros de clase un objeto puede ser “Los pilares de la Tierra”, “El código da Vinci” o “La búsqueda del sentido por parte del hombre”.

✗ Un objeto se llama "instancia de una clase".

## ***Mi primera clase***

✗ Imagina que quieres crear un programa que funcione con personas. Lo primero que tengo que hacer es crear una Clase para definir a las personas:

```
public class Person {  
    String name;  
    int age;  
}
```

✗ En este ejemplo, nos interesa almacenar el “nombre” y la “edad” de la persona.

✗ La clase es como la especie, como la especie “humana”.

✗ Ahora bien, si tenemos 3 personas diferentes, estos son 3 objetos diferentes que pertenecen a la clase "Persona".

✗ Vamos a ver...

## Constructor

- ✗ Para crear un objeto, lo primero que tenemos que hacer es agregar una función especial a nuestra clase llamada “constructor”:

```
public class Person {  
    String name;  
    int age;  
  
    public Person(String myName, int myAge) {  
        name = myName;  
        age = myAge;  
    }  
}
```

- ✗ Tenga en cuenta que el constructor tiene el mismo nombre que la clase.
- ✗ Ahora vamos a utilizar este constructor para crear 3 objetos diferentes de la clase “Persona”:

## *Crear objetos*

```
public class Person {  
    ... // Put the rest of the code here  
    public static void main() {  
        Person person1 = new Person("Juan", 25);  
        Person person2 = new Person("Luisa", 29);  
        Person person3 = new Person("Ana", 33);  
    }  
}
```

✗ Ahora tenemos 3 objetos que pertenecen a la clase Persona.

✗ Cada objeto tiene su propio nombre y edad. Imprimámoslo.

```
System.out.println(person1.name+"-"+person1.age);  
System.out.println(person2.name+"-"+person2.age);  
System.out.println(person3.name+"-"+person3.age);
```

## *toString() – para imprimir el objeto correctamente*

✗ Si imprimimos por ejemplo “persona1”:

```
System.out.println(persona1);
```

✗ Aparece algo extraño (la identificación del objeto):

```
com.daw.programming.Person@5a07e868
```

✗ Si queremos mejorar la impresión, tenemos que definir el método “toString()”:

```
public String toString() {  
    return name + " tiene " + age + " años.";  
}
```

✗ Ahora si imprimimos el objeto nuevamente:

```
System.out.println(persona1);
```

✗ Tendremos:

```
Juan tiene 25 años.
```

**Desafío6\_1,6\_2**

## ***Crear métodos que accedan a los campos.***

✗ Se puede acceder a los campos desde cualquier método del objeto. Por ejemplo, imagina que quieres crear un método para ver si la persona es menor de edad o mayor de edad:

```
public void isUnderAge() {  
    int legalAge = 18;  
    if (age < legalAge) {  
        System.out.println(name + " is underage");  
    }  
    else {  
        System.out.println(name + " is of legal age");  
    }  
}
```

✗ Podemos llamar al método como:

```
personal.isUnderAge();
```

✗ ***Ver notas de diapositivas***

**Desafío6\_3,6\_4**



## *Estático: compartir campo con todos los objetos de la clase 1/2*

✗ Algunas veces queremos compartir un campo con **todos los objetos de una clase**. En este caso, declaramos estático el campo. Cuando una clase tiene un campo estático, todos los objetos de la clase comparten el mismo campo.

```
public class Person{  
    String nombre;  
    int age;  
    static int numberOfPersons = 0;  
    ...  
}
```

✗ Ahora, cada vez que creamos una Persona incrementamos el campo:

```
Public Person(String myName, int miAge) {  
    name = myAge;  
    age = myName;  
    numberOfPersons++;  
}
```

## ***Estático: compartir campo con todos los objetos de la clase 2/2***

✗ Ahora, si creamos un método para imprimir el número de personas, puede ser estático:

```
public static void printNumberOfPersons() {  
    System.out.println("The total number of persons is:");  
    System.out.println(numberOfPersons);  
}
```

✗ Cuando llamas a un método estático no necesitas hacerlo con un objeto concreto (incluso puedes llamarlo antes de crear el primer objeto:

```
public static void main(String[] args) {  
    Person.printNumberOfPersons();  
    Person person1 = new Person("Juan", 25);  
    Person person2 = new Person("Luisa", 29);  
    Person.printNumberOfPersons();  
}
```

**Desafío6\_5-6\_6**

## ***Métodos estáticos***

- ✗ Con los métodos estáticos no necesitamos crear un objeto para usar el método. Por ejemplo, la clase Math en Java tiene métodos para calcular operaciones matemáticas:

```
int integerPart = Math.floor(12.8765);  
System.out.println(integerPart);
```

- ✗ Tendremos:

```
12
```

- ✗ El método floor obtiene la parte entera de un número decimal.
- ✗ Note que es un método “**static**”, entonces, para llamarlo, usamos el nombre de la clase sin tener que crear un objeto.

**Desafío6\_7**

## *Visibilidad privada*

✗ Nuestros campos (atributos) deben ser privados. Por ejemplo,

```
public class Person {  
    private int age;  
    ...  
}
```

✗ Esto significa que sólo dentro de la clase puedes acceder a este campo. No puedes acceder desde otra clase:

```
public class ChallengeX {  
    public static void main(String args[]) {  
        Person person1 = new Person("Juan", 34);  
        System.out.println(person1.age); => !!!ERROR!!!  
    }  
}
```

✗ Sólo puede acceder al campo dentro de la Clase en la que está definido el campo.

<https://www.arquitecturajava.com/java-encapsulamiento-y-reutilizacion/>

## *Obtener y establecer métodos*

✗ Normalmente definimos métodos "get" y "set" para acceder a los campos:

```
public class Person {  
    public getAge() {  
        return age;  
    }  
    public setAge(int myAge) {  
        age = myAge;  
    }  
}
```

✗ ¿Cuál es la ventaja de utilizar métodos? Puedes controlar la forma en que se modifican o acceden los campos. Por ejemplo, sin el método "set", puedes hacer:

```
Person person1 = new Person("Juan", 20);  
person1.age = -56;
```

✗ Pero con el método que podrías hacer: (ver la siguiente diapositiva)

## *Obtener y configurar: controlar el acceso*

```
public setAge(int myAge) {  
    if (myAge<0) {  
        System.out.println("Error, age can't be negative");  
    }  
    else {  
        age = myAge;  
    }  
}
```

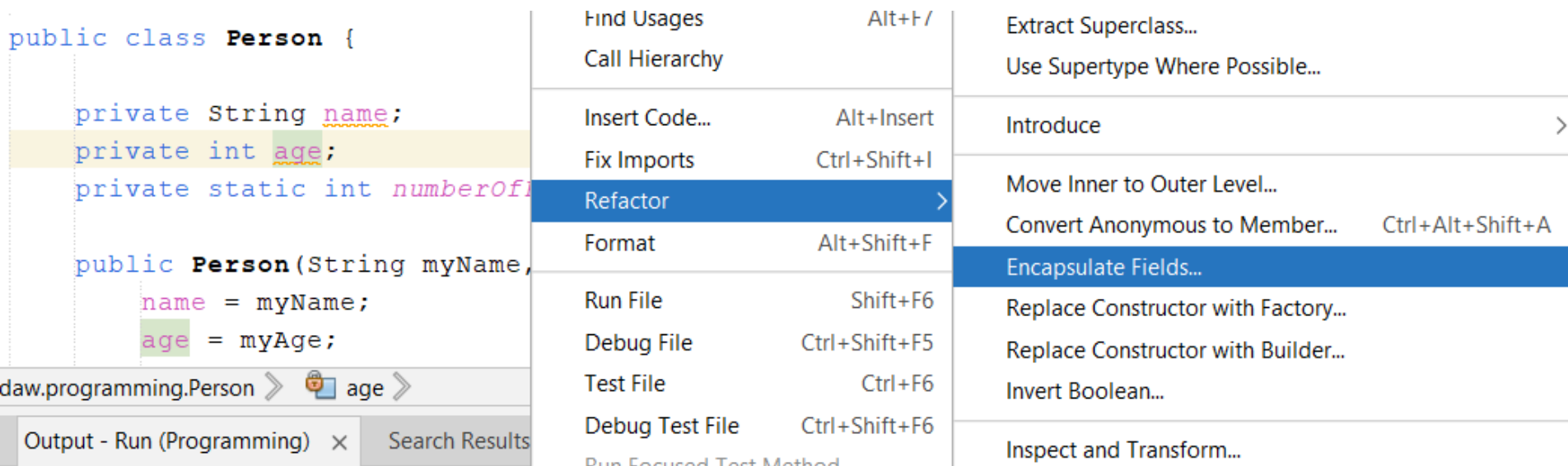
- ✗ Así podrías controlar que una edad negativa no fuera posible.
- ✗ Incluso, para un campo podrías no generar el método “set”. Por ejemplo, para “**número de personas**”, podrías generar el método **get** pero no el metodo **set**, porque no querrás que nadie cambie el número total de personas (el número total de personas solo debe cambiarse cuando se crea una nueva persona y solo incrementarse en 1).

## *Get and set: automatic generatin (1/2)*

✗ Con NetBeans puedes generar automáticamente métodos get y set. Sólo tienes que escribir los campos:

```
private String name;  
private int age;
```

✗ Haga clic derecho > Refactorizar > Encapsular campos...



The screenshot shows the NetBeans IDE interface. On the left, a code editor displays the following Java code:

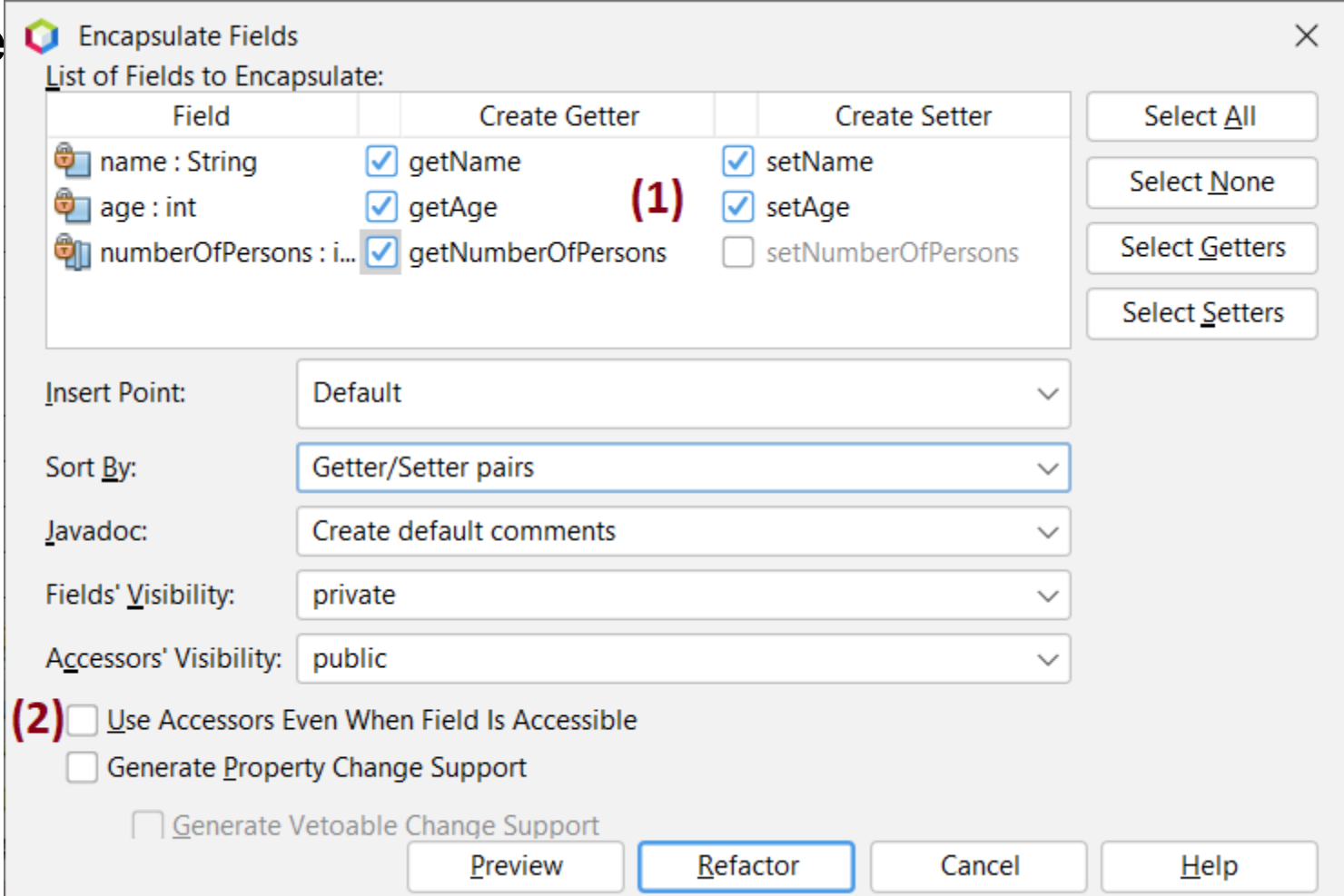
```
public class Person {  
  
    private String name;  
    private int age;  
    private static int numberOfInstances;  
  
    public Person(String myName,  
                   int myAge) {  
        name = myName;  
        age = myAge;  
        numberOfInstances++;  
    }  
}
```

The fields `name` and `age` are highlighted in yellow. A right-click context menu is open over the `age` field. The menu is divided into two panes. The left pane contains options like 'Find Usages', 'Call Hierarchy', 'Insert Code...', 'Fix Imports', 'Refactor', 'Format', 'Run File', 'Debug File', 'Test File', 'Debug Test File', and 'Run Enclosed Test Method'. The 'Refactor' option is selected and expanded, showing a submenu on the right. The submenu contains options like 'Extract Superclass...', 'Use Supertype Where Possible...', 'Introduce', 'Move Inner to Outer Level...', 'Convert Anonymous to Member...', 'Encapsulate Fields...', 'Replace Constructor with Factory...', 'Replace Constructor with Builder...', 'Invert Boolean...', and 'Inspect and Transform...'. The 'Encapsulate Fields...' option is highlighted in blue. The bottom status bar shows 'Output - Run (Programming)' and 'Search Results'.



## *Get and set: automatic generation (2/2)*

✗ Marque los getters y setters deseados en la parte superior **(1)** y desmarque "Usar accesores incluso cuando el campo sea accesible" **(2)**



The dialog box titled "Encapsulate Fields" is used for generating getters and setters for class fields. It features a table for selecting which fields to encapsulate and checkboxes for additional options.

Field	Create Getter	Create Setter
name : String	<input checked="" type="checkbox"/> getName	<input checked="" type="checkbox"/> setName
age : int	<input checked="" type="checkbox"/> getAge	<input checked="" type="checkbox"/> setAge
numberOfPersons : i...	<input checked="" type="checkbox"/> getNumberOfPersons	<input type="checkbox"/> setNumberOfPersons

Buttons on the right: Select All, Select None, Select Getters, Select Setters.

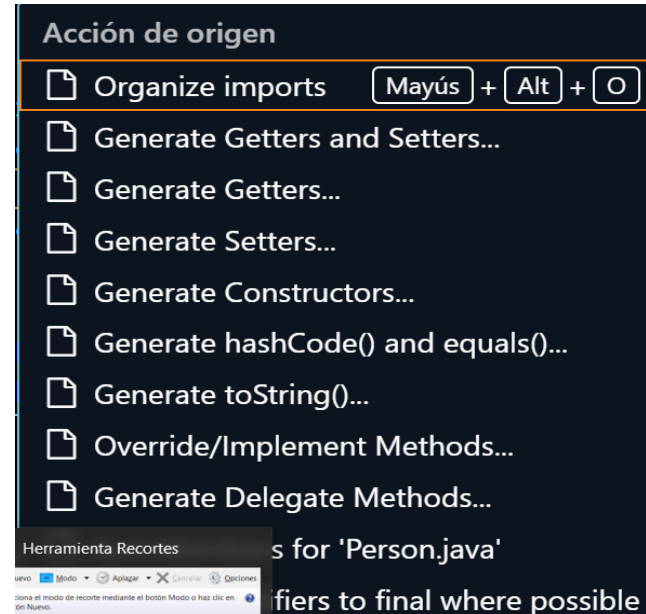
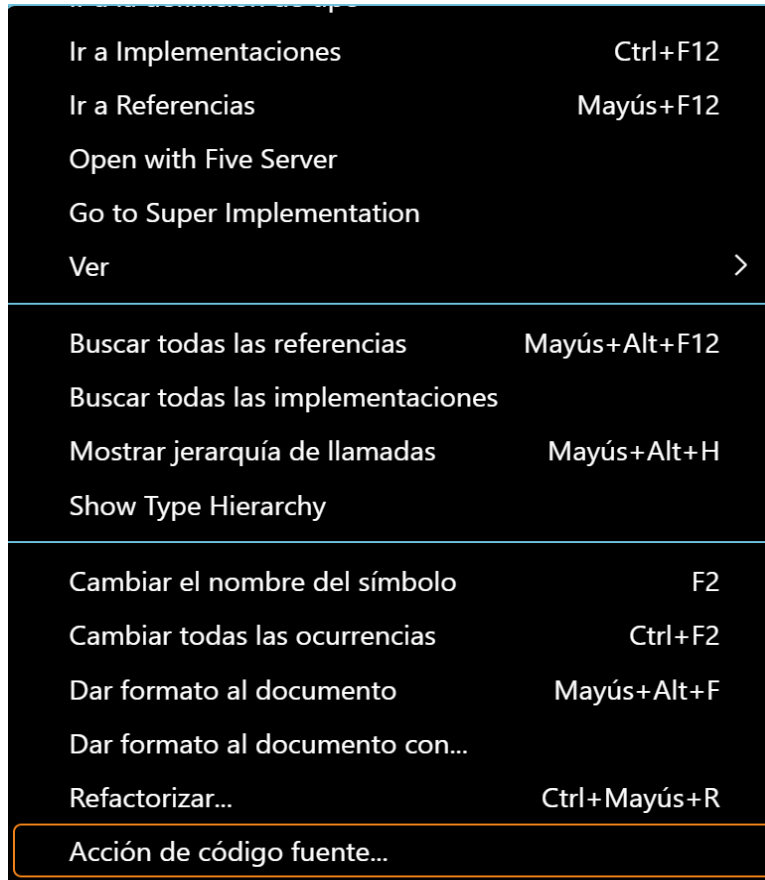
Options at the bottom:

- Insert Point: Default
- Sort By: Getter/Setter pairs
- Javadoc: Create default comments
- Fields' Visibility: private
- Accessors' Visibility: public
- ☐ Use Accessors Even When Field Is Accessible
- ☐ Generate Property Change Support
- ☐ Generate Vetoable Change Support

Buttons at the bottom: Preview, Refactor, Cancel, Help.

## *Get and set: automatic generation (VSCode)*

En **VSCode**, click derecho y en menú emergente seleccionar:



## *Excepción a la encapsulación: constantes*

✗ Una **constante** es un campo que no cambia. Por ejemplo:

```
public class Circle {  
    public static final double PI = 3.141592654;  
}
```

✗ Es un error hacer:

```
Círculo.PI = 4; => ¡¡¡ERROR!!!
```

✗ Declaramos las constantes como públicas, y sin métodos get y set, porque son definitivas y no se pueden cambiar, por lo que ya están protegidas.

✗ *Observe que las constantes están escritas con todas las letras en mayúsculas.*

**Desafío6\_8-6\_9**

## *this*

✗ **this** se refiere al “objeto actual”. Por ejemplo:

```
public class Person {  
    String name;  
    int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

✗ Tenga en cuenta que los campos y los parámetros del constructor tienen los mismos nombres. si ponemos **this.age** nos referimos al campo, y si ponemos solo **age** nos referimos al parámetro.

✗ Podemos usar **this** para referirnos al objeto actual que estamos usando.

**Desafío6\_9**

## *Herencia*

✗ Una clase puede "**heredar**" de otra clase. Por ejemplo:

```
public class Student extends Person {  
    private String course;  
}
```

✗ Esto quiere decir que Student tiene el mismo contenido (métodos y atributos) que Person, y además tiene un atributo llamado curso (para guardar en qué curso estás estudiando: 1ºdaw, 2ºdaw, 1ºasir...)

## *Herencia: acceso a los campos*

✗ No se puede acceder a los campos de la clase “madre” (Persona) en la clase heredada (Estudiante) porque son privados:

```
public class Student extends Person {  
    ...  
    public test() {  
        System.out.println(name);  
    }  
}
```

✗ En su lugar tienes que usar “get” o “set”:

```
...  
public test() {  
    System.out.println(getName());  
}
```

## *Herencia: constructor*

- ✗ “super(name, age, height)” significa que ejecutas el constructor de la clase “madre” (Person).
- ✗ El constructor en person almacenará nombre, edad y altura, y nosotros almacenaremos el curso.
- ✗ super se refiere a la clase madre (en este caso Persona).

```
public class Student extends Person {  
    ...  
    public Student(String name, int age, int height, String course) {  
        super(name, age, height);  
        this.course = course;  
    }  
}
```

**Desafío6\_10, 6\_11**

## ***Visibilidad de los modificadores: private, package, protected, public***

✗ Aparte de private y public tenemos modificadores de visibilidad de paquetes y protegido en Java:

Modificador (visibilidad)	Dentro de la propia clase	Clase en el mismo paquete	Subclase (hereda), fuera del paquete	Clase fuera del paquete
private	✓	✗	✗	✗
package	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

✗ Package son las clases que están en el mismo paquete (carpeta). Es el valor "predeterminado".



## Herencia: súper 1/2

✖ Tenemos una clase *Cuenta bancaria*:

```
public class CuentaBancaria {
    private int saldo=0;
    private ArrayList<Integer> movimientos = new ArrayList();

    public int getSaldo() {
        return this.saldo;
    }
    public void ingresar(int importe) {
        saldo = saldo + importe;
        movimientos.add(importe);
    }
    public void retirar(int importe) {
        saldo = saldo - importe;
        movimientos.add(-importe);
    }
    public void print() {
        System.out.println("Saldo actual: "+saldo);
        System.out.println("Movimientos de la cuenta: ");
        for (int transaccion : movimientos) {
            System.out.println(transaccion);
        }
    }
    public static void main(String[] args) {
        CuentaBancaria vipAccount = new CuentaBancaria();
        vipAccount.ingresar(100);
        vipAccount.retirar(80);
        vipAccount.retirar(50);
        vipAccount.print();
    }
}
```

## *Herencia: super 2/2*

✗ Creamos una *CuentaBancariaLimitada*, que es igual pero no puedes tener saldo negativo:

```
public class CuentaBancariaLimitada extends CuentaBancaria {  
    public void retirar(int importe) {  
        if (importe > getSaldo()) {  
            System.out.println("FORBIDDEN: There is not enough balance");  
        }  
        else {  
            super.retirar(importe);  
        }  
    }  
    public static void main(String[] args) {  
        CuentaBancariaLimitada limitedAccount = new CuentaBancariaLimitada();  
        limitedAccount.ingresar (100);  
        limitedAccount.retirar(80);  
        limitedAccount.retirar(50);  
        limitedAccount.print();  
    }  
}
```

**Desafío6\_12-6\_20**

✗ Redefines el método **retirar** y cuando quieres hacer lo mismo que este método en la clase principal, llamas a `super.retirar(importe)`.

## *Clases Abstractas*

Una **clase abstracta** es prácticamente idéntica a una clase convencional; las clases abstractas pueden poseer atributos, métodos, constructores, etc ...

La principal diferencia entre una clase convencional y una clase abstracta es que la clase abstracta debe poseer por lo menos **un** método abstracto. Ok, pero ahora, ¿Qué es un método abstracto? Verás, un método abstracto no es más que un método vacío, un método el cual no posee cuerpo, por ende no puede realizar ninguna acción. La utilidad de un método abstracto es definir qué se debe hacer pero no el cómo se debe hacer.

Veamos un ejemplo para que nos quede más en claro

## *Clases Abstractas*

```
public class Figura {  
    private int numeroLados;  
    public Figura() { this.numeroLados = 0; }  
    public float area() { return 0f; }  
}
```

En este caso la clase posee una atributo, un constructor y un método, a partir de esta clase podré generar la n cantidad de figuras que necesite, ya sean cuadrados, rectangulos, triangulos, circulos etc...

Dentro de la clase encontramos el método área, método que se encuentra pensado para obtener el área de cualquier figura, sin embargo cómo sabemos todas las figuras poseen su propia fórmula matemática para calcular su área. Si yo comienzo a heredar de la clase Figura todas las clases hijas tendrían que sobre escribir el método área e implementar su propia formula para así poder calcular su área. En estos casos, en los casos la clase hija siempre deba que sobreescribir el método lo que podemos hacer es convertir al método convencional en un método abstracto, un método que defina qué hacer, pero no cómo se deba hacer.

## *Clases Abstractas*

```
public abstract float area();
```

Ahora que el método área es un método abstracto la clase se convierte en una clase abstracta.

```
public abstract class Figura {
```

Las clases abstractas pueden ser heredadas por la n cantidad de clases que necesitemos, pero **no** pueden ser instanciadas. Para heredar de una clase abstracta basta con utilizar la palabra reservada *extends*.

## *Clases Abstractas*

```
public class Triangulo extends Figura {
```

✘ Al nosotros heredar de una clase abstracta es obligatorio implementar todos sus métodos abstractos, es decir debemos definir comportamiento, definir cómo se va a realizar la tarea

## *Interfaces*

Ahora hablaremos de interfaces. A pesar que es un tema un poco complejo si nosotros hemos comprendido el tema de clases abstractas y el por que de ellas, el tema de interfaces será un tema muy sencillo.

Veamos. A diferencia de otros lenguajes de programación, en **Java no es posible la herencia múltiple**, nuestras clases únicamente podrán heredar de una y solo una clase.

Si conceptualizamos esto una representación pudiese ser la siguiente.

## *Clases Abstractas*



El nivel de jerarquía es descendente. Esto sin duda funciona, sin embargo, si queremos representar conceptos de la vida real necesitaremos una jerarquía mucho más compleja, algo como esto.





## *Clases Abstractas*

✗ Para que podamos implementar nuestro proyecto de esta forma, teniendo en cuenta que únicamente es posible heredar de una clase, entonces haremos uso de interfaces.

```
public interface Canino {  
    public abstract void aullar();  
    public abstract void ladrar();  
}
```

```
public class Perro extends Mascota implements Canino
```

Una clase hija solo podrá heredar de una clase abstracta, por otro lado podrá hacer uso de la n cantidad de interfaces que necesite.

## *Interfaces: definición 1/6*

✗ En esencia una Interfaz es como una clase totalmente abstracta, solo define los métodos que debe tener una clase, pero no define el contenido de los métodos. Por ejemplo:

```
public interface Drawable {  
    public void draw(Graphics g);  
}
```

✗ Al igual que una clase abstracta, no puedes crear objetos que pertenezcan a una interfaz:

```
Drawable d = new Drawable();
```

## *Interfaces: implementación 2/6*

✗ Puedes crear una clase que "implemente" esta interfaz:

```
public class Line implements Drawable {  
    ...  
    public void draw(Graphics g) {  
        g.drawLine(x1, y1, x2, y2);  
    }  
}
```

✗ Esto te obliga a crear un método "draw" y puedes tener un "ArrayList<Drawable>" e insertar en ellos líneas, rectángulos, triángulos que implementan Drawable. Luego puedes usar "dibujar" de forma polimórfica.

✗ De esta manera, tu clase es "libre" de heredar de otra clase, por ejemplo:

```
public class Line extends Figure implements Drawable {
```

## *Interfaces 3/6*

✗ Ahora, si tienes un elemento que queremos pintar en el editor pero no es una “Figura” (por ejemplo una imagen no tiene color), podemos implementar la interfaz “Drawable”:

```
public class Image implements Drawable {  
    public void draw(Graphics g) {  
        g.drawImage(...  
    }  
}
```

✗ Y puedes agregarlo a ArrayList y pintarlo.

✗ Esto te obliga a crear un método "draw" y puedes tener un "ArrayList<Drawable>" e insertar en ellos líneas, rectángulos, triángulos que implementan Drawable. Luego puedes usar "dibujar" de forma polimórfica.

✗ Una imagen es muy diferente a una Figura pero puedes integrarla si tiene el método “draw”.

## *Interfaces: implementación múltiple. y convenciones de nombres 4/6*

- ✗ No podemos heredar de múltiples clases pero podemos implementar múltiples clases:

```
public class Line implements Drawable, Comparable {
```

- ✗ Esto significa que podemos usar “Line” como Drawable y Comparable (porque tendrá los métodos de Drawable y Comparable).
- ✗ Al igual que las clases, los nombres de las interfaces deben comenzar con mayúscula seguido de minúsculas con la primera letra de cada palabra capitalizada.
- ✗ Los nombres de las interfaces, en general, deben ser **adjetivos**: Comparable, Serializable, Clonable.
- ✗ En algunos casos, las interfaces pueden ser **sustantivos** así como cuando presentan una familia de clases ej. Lista, Mapa o Figura.
- ✗ Pueden incluir campos y métodos estáticos e implementaciones “predeterminadas” (consulte las notas de la diapositiva).

## *Interfaces: comparables 5/6*

- ✗ Cuando tenemos un ArrayList de objetos, si podemos “ordenarlos” u “ordenarlos”, los objetos pueden ser muy diferentes y ordenarse de maneras realmente diferentes. Por ejemplo, es posible que deseemos ordenar una ArrayList de personas por edad y una ArrayList de rectángulos por su área.
- ✗ Debido a que estos objetos pueden ser tan diferentes usamos la interfaz “Comparable” (en general, usamos

```
public class Person implements Comparable<Person> {  
    public int compareTo(Person p) {  
        int ageDifference = this.age - p.getAge();  
        return ageDifference;  
    }  
}
```

## *Interfaces: comparables 6/6*

✗ Ahora si ordenamos un “ArrayList” con “Personas” las vamos a tener ordenadas por edad.

✗ Ahora si queremos ordenar Rectángulos:

```
public class Rectangle extends Figure implements
Comparable<Rectangle> {
    public int compareTo(Rectangle r) {
        int areaDifference = this.getArea() -
r.getArea();
        return areaDifference;
    }
}
```

**Reto6\_21-**