UD7. Programación Orientada a Objetos I

Módulo: Programación

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Germán Gascón Grau
g.gascongrau@edu.gva.es

Contenidos

- Orígenes de la POO
- Los límites de la programación clásica estructurada
- Definición de una clase
- Agregar atributos
- Los constructores
- La referencia this
- Los destructores. El recolector de basura
- Agregar métodos
- Principios de la POO
- Métodos Getters y Setters
- Modificadores de ámbito. Otros modificadores
- Recomendaciones para el diseño de clases

Origen

- Durante la década de los 70 el software empezó a crecer de tamaño y con él, la cantidad de errores que se cometían al desarrollarlo, por lo que muchos proyectos empezaron a fracasar debido a los sobrecostes. Estaba claro que ni las herramientas ni la metodología utilizada eran las adecuadas. La **crisis del software** había comenzado.
- Para paliar los efectos de esta crisis se hicieron muchas propuestas, de entre las que podemos destacar las siguientes:
  - Convertir el desarrollo del software en una **ingeniería**.
  - Crear un nuevo paradigma de programación llamado **P**rogramación **O**rientada a **O**bjetos (**POO**)

La POO al rescate

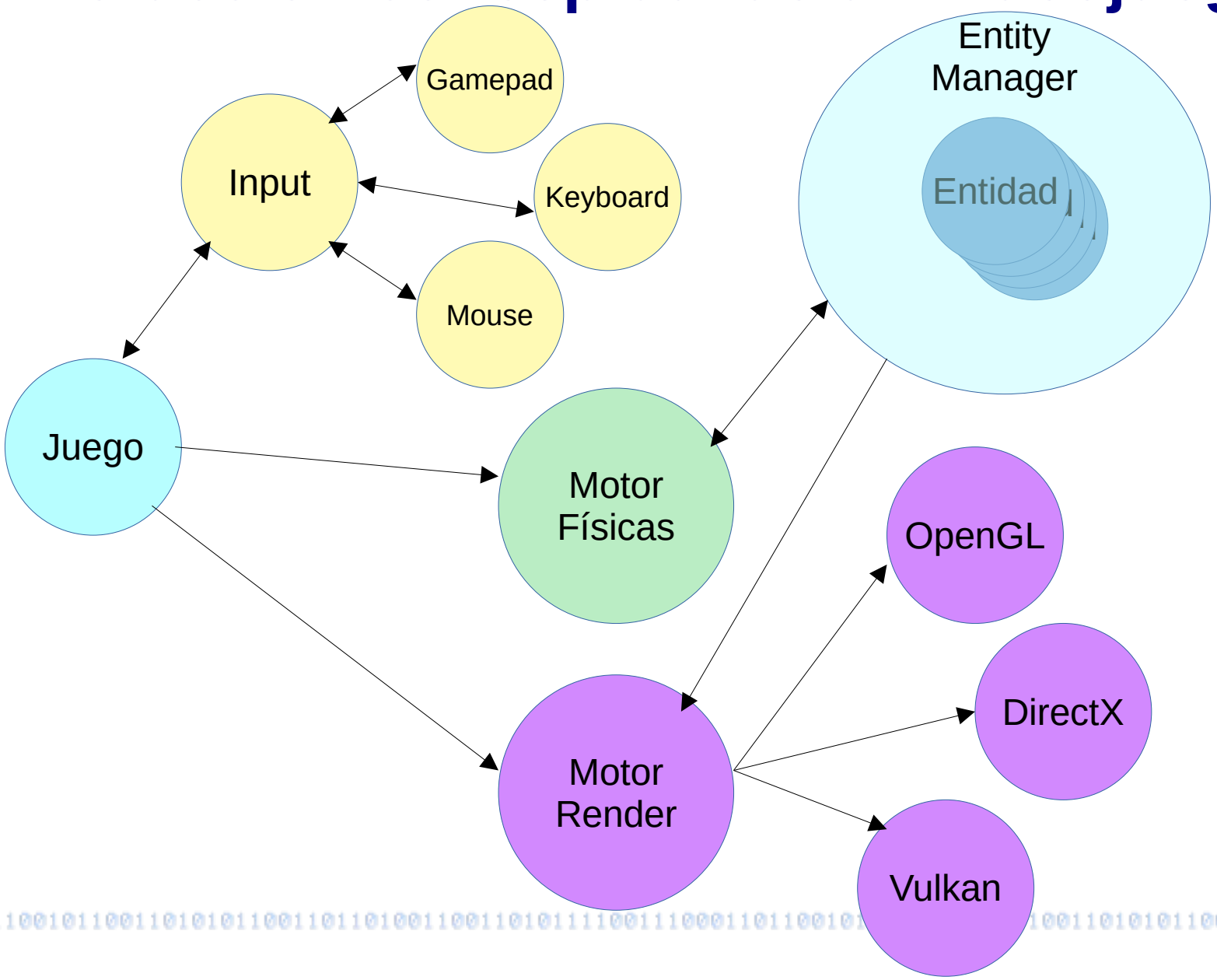
- Hasta la aparición de la POO el paradigma empleado en el desarrollo de software era la **programación estructura**, que es básicamente lo que hemos visto hasta ahora.
- La POO intenta ir un paso más allá, tomando como base la organización de los humanos como sociedad.
- Los humanos nos hemos organizado a través de los oficios creando **especializaciones** para dar solución a problemas de forma más eficiente.

- Si se te rompe el coche vas al mecánico porque él conoce las piezas y sabe como montar y desmontar un coche. Si estás enfermo vas al médico porque él conoce y sabe como tratar muchas de las enfermedades.

La especialización es la clave

- La POO pretende **simplificar** el desarrollo del software **dividiendo el programa en pequeñas partes**, donde cada una de esas partes (a las que llamaremos objetos) se encargará de una **tarea muy concreta** y en la que será **especialista**.
- Cuando un objeto necesite realizar una tarea en la que no es especialista, se la solicitará a un objeto que sí lo sea.
- Nuestro trabajo como programadores consistirá principalmente en determinar e implementar:
  - La cantidad y el tipo de especializaciones en las que debemos dividir el programa (las clases/objetos)
  - Los datos (propiedades/atributos) con los que trabajaran dichos objetos
  - Las acciones (métodos) que serán capaces de realizar dichos objetos.
  - Las interacciones que se producirán entre los objetos para dar la solución al problema.

Interacción conceptual de un videojuego



Métodos vs Objetos

- En programación estructura los métodos trabajan con datos recibidos como parámetros pero no son propietarios ni responsables de dichos datos.
- Además en un método las variables locales que utiliza se destruyen al finalizar el método por lo que no hay persistencia del estado.
- Los objetos tienen variables (propiedades/atributos) de las que son responsables y que reflejan en todo momento el estado del objeto.
- Los métodos del objeto realizan su función tomando como datos las variables (propiedades/atributos) del objeto. De esta forma el estado del objeto permanece y sobre todo, lo más importante el objeto es el responsable de dichas variables.

Métodos vs Objetos

- Supongamos que queremos realizar un juego de fútbol y para ello necesitamos representar las características de cada uno de l@s futbolistas. Por ejemplo:

Portero	32
Defensa	62
Pase	89
Regate	88
Tiro	81
EstadoForma	80

- ¿Cómo representarías esta información para l@s futbolistas de un equipo?
- ¿Cómo diseñarías el método entrenar que al aplicarlo a un@ de l@s futbolistas mejore alguna de sus características?

Métodos vs Objetos

- Una primera aproximación podría ser:

```
public class Juego {
    public static float porteroFutbolista1;
    public static float defensaFutbolista1;
    public static float paseFutbolista1;
    public static float regateFutbolista1;
    public static float tiroFutbolista1;
    public static float estadoFormaFutbolista1;
    public static float porteroFutbolista2;
    public static float defensaFutbolista2;
    public static float paseFutbolista2;
    public static float regateFutbolista2;
    public static float tiroFutbolista2;
    public static float estadoFormaFutbolista2;
    ...
    public void entrenar(????) {
       ??????
    }
}
```

¿A qué futbolista aplicamos el entrenamiento?

- Esta aproximación presenta varios inconvenientes:
  - Utiliza variables globales.
  - No queda claro quien es el responsable de los datos.
  - ¿Cómo indicamos a qué futbolista queremos aplicar el entrenamiento?
  - La cantidad de futbolistas es fija. Para ampliar el número tenemos que modificar el código fuente.

Métodos vs Objetos

- Una segunda aproximación podría ser utilizando una matriz donde cada fila de la matriz represente a un futbolista y cada columna represente una característica (portero, defensa, pase, regate, tiro) de dicho futbolista.

```
public class Juego {
    public static float[][] equipo = new float[20][6];
    // Cada fila de la matriz será un futbolista
    // y cada columna una característica (portero, defensa, pase, regate, tiro, estadoForma)
    public void entrenar(int futbolista, int caracteristica) {
        Random random = new Random();
        float mejora = random.nextFloat();
        equipo[futbolista][5] += mejora;
        // Probabilidad de mejorar algún atributo
        int num = random.nextInt(100);
        if (num >= 100 - 7) { // 7% de probabilidad de mejorar la característica
            mejora = random.nextFloat();
            equipo[futbolista][caracteristica] += mejora;
        }
    }
}
```

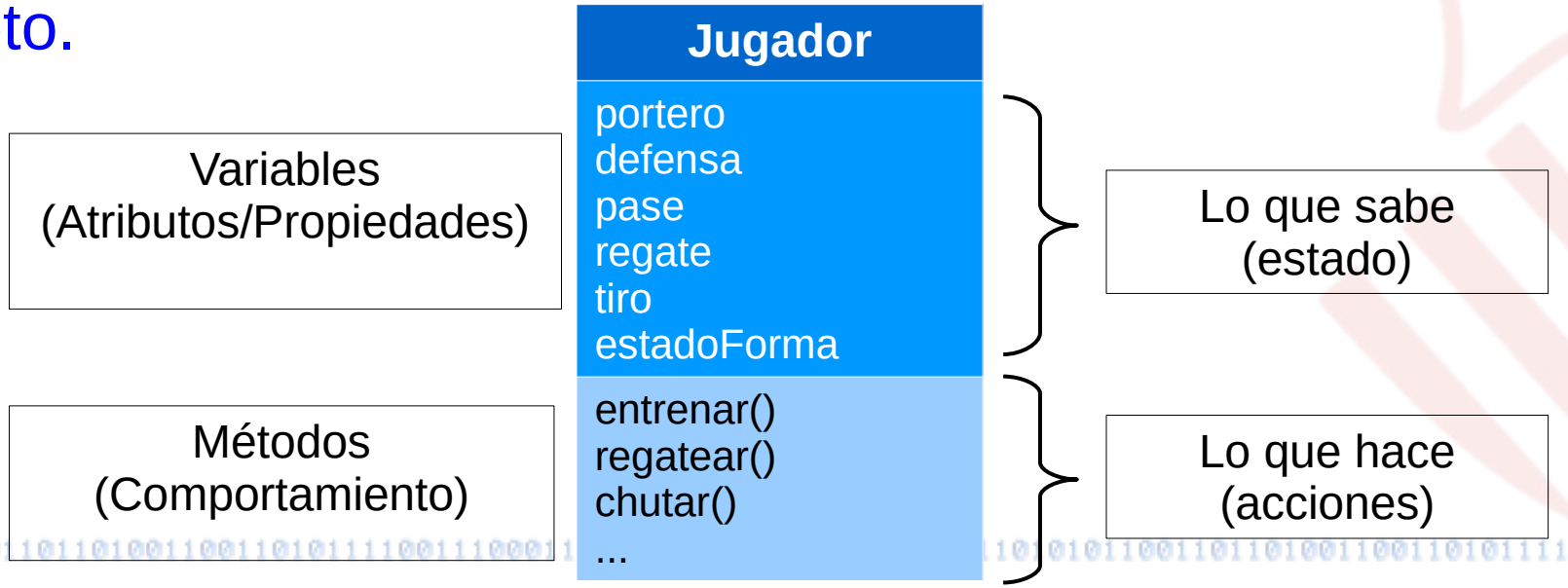
- Esta aproximación mejora sustancialmente la propuesta anterior pero sigue presentando algunos inconvenientes:
  - Utiliza variables globales.
  - Hay que “acordarse” donde está ubicado cada jugador y característica en la matriz.
  - Sigue sin haber un responsable claro de los datos.

Métodos vs Objetos

- Aplicando POO podemos crear **dividir en varias partes la funcionalidad**, haciendo que cada una de estas partes **sea responsable y especialista** de los datos que trata.
- Una de estas “partes” podría gestionar los datos de un/a futbolista. Otra de las partes podría gestionar los datos de un equipo.
- En POO llamamos **clase** a cada una de estas divisiones lógicas y **representa un nuevo tipo de datos** creado por nosotros.
- A cada variable que definamos sobre ese tipo de datos creado lo llamaremos **objeto**.

Definición de clase

- Podemos definir una **clase** como la "**plantilla**" o "molde" con el que daremos forma a los objetos.
- La clase determinará los **atributos** y los **métodos** que tendrán los objetos que se creen de dicha clase.
- Los **atributos** representarán el estado del objeto.
- Los **métodos** representarán las acciones que puede realizar dicho objeto.



Anatomía básica de una clase en Java

```
public class Jugador {
    private [final] tipo nombrePropiedad1;
    private [final] tipo nombrePropiedad2;
    private [final] tipo nombrePropiedad3;

    // Método constructor
    public NombreClase() {

    }
    // Métodos
    public void metodo1() {

    }
    public void metodo2() {

    }
}
```

Atributos/Propiedades
(representarán el **estado** del objeto)

Constructor
(método especial cuya función es **inicializar** el objeto. Es decir, darle valores iniciales a las propiedades)

Métodos
(los métodos determinarán el **comportamiento** del objeto)

Relación entre estado y comportamiento

- El **estado** de un objeto viene representado por los valores que tomen sus **atributos**.
- El **comportamiento** de un objeto viene representado por las acciones que puede realizar, es decir, por sus **métodos**.
- El estado de un objeto determina su **comportamiento**, y a su vez el comportamiento afecta al estado.

Atributos (variables de instancia)

- Las variables de instancia (atributos) son las que están declaradas dentro del cuerpo de la clase.

```
public class Jugador {
    private String nombre;
    private float portero;
    private float defensa;
    private float pase;
    private float regate;
    private float tiro;
    private float estadoForma;

    // Constructores y resto de métodos
}
```

nombre	Gayà
portero	32
defensa	85
pase	85
regate	81
tiro	79
estadoForma	90

- Para preservar el “estado correcto” del objeto, sus atributos sólo podrán ser modificados por los métodos del objeto.
- Por este motivo las haremos siempre privadas añadiendo la palabra reservada **private** en su declaración.

¿Qué pasaría si los atributos fuesen modificables directamente desde fuera del objeto?

Constructores

- Los constructores son métodos especiales que sirven para **darle valores a los atributos** (inicializar el estado) a un objeto al crearlo.
- Se caracterizan porque **su nombre es el mismo que el de la clase** y no tienen tipo de retorno.
- Pueden estar **sobrecargados**, es decir, pueden existir varios métodos constructores siempre que el número y/o tipo de parámetros sea distinto.

```
public Jugador(String nombre, float portero, float defensa, float pase,
float regate, float tiro, float estadoForma) {
    this.nombre = nombre;
    this.portero = portero;
    this.defensa = defensa;
    this.pase = pase;
    this.regate = regate;
    this.tiro = tiro;
    this.estadoForma = estadoForma;
}
```

- En este caso inicializaríamos todos los atributos.

¿Cómo añadirías otro constructor que sólo reciba como parámetro el nombre del Jugador e inicialice todas las características del jugador a 50?

Constructores

```
public Jugador(String nombre) {
    this(nombre, 50, 50, 50, 50, 50, 50);
}
```

- La palabra reservada **this** hace referencia al objeto actual. Para entender cuál es el objeto “actual”, hay que pensar que la clase es una plantilla/molde de datos (atributos) y código (métodos) que los objetos ejecutarán para llevar a cabo su tarea. Pues bien, el objeto que en ese momento está ejecutando el código es **this**.
- Si a **this** le pasamos valores entre paréntesis, estamos invocando al constructor que tenga ese número y tipo de parámetros.
- Debemos evitar código de inicialización repetido. Si tenemos varios constructores, intentaremos, en la medida de lo posible, poner todo el código en uno de ellos, habitualmente el que más parámetros recibe, y el resto de constructores harán su inicialización invocándolo mediante **this(lista de parámetros)**.

Constructores

- Para invocar a un constructor hay que utilizar la palabra reservada `new`.
- Con los 2 constructores anteriores, podríamos crear **objetos** de tipo **Jugador** de 2 formas:

```
Jugador futbolista1 = new Jugador("Aitana Bonmatí", 32, 82, 87, 80, 82, 95);
Jugador futbolista2 = new Jugador("Pablo Gavi", 32, 82, 87, 80, 82, 95);
Jugador futbolista3 = new Jugador("Ana García");
Jugador futbolista4 = new Jugador("Rubén Prieto");
```

- `futbolista1` y `futbolista2` utilizan el constructor que recibe como parámetros todas las características, por tanto se le asignarán los valores que se han pasado en el momento de la creación.
- `futbolista3` y `futbolista4` utilizan el constructor que sólo recibe el nombre, por tanto, el resto de características valdrán 50 ya que así lo hemos establecido en el constructor.

Constructores

- ¿Y qué pasa si no ponemos un constructor?
- Java crea uno llamado constructor por defecto que no recibe ningún parámetro e inicializa todos los atributos a sus valores por defecto:

enteros	0
decimales	0.0
booleanos	falso
referencias	null

- Java sólo añade un constructor por defecto si no hay ninguno definido. Eso quiere decir, que si añadimos un constructor, ya no existirá un constructor por defecto.
- Si hay algún constructor, Java se limita a hacer aquello que el constructor dice.

Constructores

```
public class Jugador {
    private final String nombre;
    private float portero;
    private float defensa;
    private float pase;
    private float regate;
    private float tiro;
    private float estadoForma;

    // Constructor
    public Jugador(String nombre, float portero, float defensa, float pase, float regate,
float tiro, float estadoForma) {
        this.nombre = nombre;
        this.portero = portero;
        this.defensa = defensa;
        this.pase = pase;
        this.regate = regate;
        this.tiro = tiro;
        this.estadoForma = estadoForma;
    }
    // Constructor
    public Jugador(String nombre) {
        this(nombre, 50, 50, 50, 50, 50, 50);
    }
    // Resto de métodos
}
```

¿Destructores?

- Al crear un objeto mediante la palabra reservada `new`, estamos **solicitando memoria** al sistema operativo para poder almacenar sus datos.
- Cuando ese objeto deja de ser útil, es necesario indicarle al sistema operativo que **libere** esa zona de memoria para que pueda ser utilizada por otros objetos o de lo contrario nos quedaremos sin memoria.
- En la mayoría de **lenguajes de sistemas** como C, C++ o Pascal, esto debe hacerlo el programador explícitamente llamando a una función, habitualmente llamada free().
- En Java existe un **recolector de basura** (Garbage Collector) que se encarga de realizar esta tarea automáticamente. Este proceso es llevado a cabo por la **JVM** de forma **automática** y **periódica**.

El recolector de basura

- Cada vez que se crea un objeto el recolector de basura crea una **lista** a la que se irán añadiendo las **referencias** que apuntan a dicho objeto, es decir variables que apuntan al objeto.
- Cuando un objeto se queda **sin referencias**, es decir no quedan variables que lo apunten, ese objeto **deja de ser accesible** para el programador y por tanto el recolector de basura **avisará automáticamente al sistema operativo para que libere la zona de memoria** ocupada por el objeto.
- Java ofrece el método estático `System.gc()` para "aconsejar" a la máquina virtual de Java que ejecute el recolector, pero en ningún caso está asegurada su ejecución.

El recolector de basura

- Veamos un ejemplo:

```
public void crearFutbolista() {
    Jugador futbolista1 = new Jugador("Alexia Putellas", 29, 81, 80, 83, 84, 90);
}
```

- Al crear el objeto se crea la lista y se añade a dicha lista la variable `futbolista1` como referencia que apunta al objeto creado.
- Pero como el ciclo de vida de la variable `futbolista1` acaba al finalizar el método `crearFutbolista()`, el objeto creado se quedará sin referencias al finalizar el método y el objeto creado será destruido.

¿Cómo podríamos hacer que la variable futbolista1 no sea destruida al finalizar el método?

El recolector de basura

- Una opción podría ser:

```
public Jugador crearFutbolista() {
    return new Jugador("Alexia Putellas", 29, 81, 80, 83, 84, 90);
}
```

- Al devolver la referencia al objeto creado, podrá recogida por alguna variable, que pasará a apuntar al objeto creado y por tanto se añadirá a la lista de referencias del objeto.
- Otra posible opción sería tener un array de objetos de tipo Jugador como atributo en la clase y añadir el objeto creado al array.

Métodos

- Los métodos definen como se van a comportar los objetos modificando su estado.

```
public void entrenarPase() {
    Random random = new Random();
    float mejora = random.nextFloat();
    estadoForma += mejora;
    // Probabilidad de mejorar el atributo pase
    int num = random.nextInt(100);
    if (num >= 100 - 7) { // 7% de probabilidad
        mejora = random.nextFloat();
        pase += mejora;
    }
}
```

- El método anterior se aplicará sobre una instancia (objeto) de la clase `Jugador`.
- Cómo podemos observar, puede modificar 2 atributos: `estadoForma` y `pase`.

Static

- Crearemos atributos y métodos `static` cuando el atributo o el método vale o da el mismo resultado en todos los objetos.
- Crearemos atributos y métodos dinámicos cuando el atributo o método vale o da diferentes resultados según el objeto.
- Por ejemplo, en una clase que represente aviones:
  - La `altura` sería un atributo dinámico, ya que cada avión tendrá su propia altura.
  - El `numeroTotal` de aviones sería un atributo `static` (el mismo para todos los aviones).

Comportamiento de los objetos

- En Java todos los objetos heredan un comportamiento base definido en la clase `Object` que es la clase padre de todos los objetos.
- El tema de la herencia lo trataremos en detalle en próximos temas, ahora simplemente necesitamos saber que 3 de los métodos que se heredan de la clase `Object` son el método `toString()`, que nos será útil para mostrar los datos de los objetos que creemos, el método `equals(Object o)`, que nos será útil para determinar cuando deben considerarse que son el mismo y el método `hashCode()`.

Método toString()

- Este método tiene la siguente firma:

```
public String toString()
```

- `toString()` es invocado automáticamente por varios métodos de la librería estándar de Java, como por ejemplo `System.out.println()`, cuando recibe un objeto como parámetro. De esta forma obtienen una representación del objeto en una cadena de texto.
- La implementación que tiene por defecto es mostrar la dirección de memoria del objeto. Por ello, si hacemos un `System.out.println(futbolista1)` de un `Jugador` veremos algo similar a: `JugadorClase@3fee733d`
- Para facilitar la depuración del código sobreescribiremos el método `toString()` en todas las clases.

Método toString()

- La mayoría de IDE's tienen la función de generar automáticamente este método y otros.
- En IntelliJ podemos ir al menú Code → Generate → toString()
- Nos preguntará qué atributos queremos incluir, habitualmente todas. Para la clase Jugador quedaría algo similar a:

```
@Override
public String toString() {
    return "Jugador{" +
            "portero=" + portero +
            ", defensa=" + defensa +
            ", pase=" + pase +
            ", regate=" + regate +
            ", tiro=" + tiro +
            ", estadoForma=" + estadoForma +
            '}';
}
```

¿Para qué sirve la anotación `@Override`?

Método equals(Object o)

- Este método tiene la siguente firma:

```
public boolean equals(Object o)
```

- `equals(Object o)` es invocado automáticamente por varios métodos de la librería estándar de Java cuando se necesita determinar si dos objetos son el mismo.
- En próximos temas veremos porqué está íntimamente relacionado con el método `hashCode()`.
- En la implementación que tiene por defecto devuelve true si el objeto actual y el recibido como parámetro son el mismo, es decir, tienen la misma dirección de memoria.
- Para que nuestras aplicaciones se comporten correctamente, hemos de sobreescribir el método `equals(Object o)` en todas las clases.

Método hashCode()

- Este método tiene la siguente firma:

```
public int hashCode()
```

- `hashCode()` es invocado automáticamente por las interfaces `Map` para calcular las tablas de indexación internas mediante funciones hash.
- En próximos temas trataremos a fondo la interfaz `Map`.

Métodos equals(Object o) y hashCode()

- La mayoría de IDE's tienen la función de generar automáticamente este método y otros.
- En IntelliJ podemos ir al menú Code → Generate → equals() and hashCode()
- Nos preguntará qué atributos queremos utilizar para determinar que dos objetos debe ser considerados como el mismo.

Método toString()

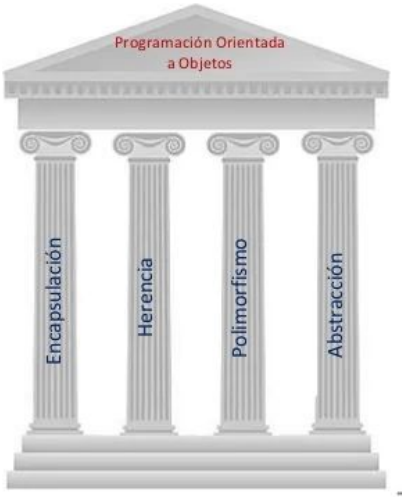
- La mayoría de IDE's tienen la función de generar automáticamente este método y otros.
- En IntelliJ podemos ir al menú Code → Generate → toString()
- Nos preguntará qué atributos queremos incluir, habitualmente todas. Para la clase Jugador quedaría algo similar a:

```
@Override
public String toString() {
    return "Jugador{" +
            "portero=" + portero +
            ", defensa=" + defensa +
            ", pase=" + pase +
            ", regate=" + regate +
            ", tiro=" + tiro +
            ", estadoForma=" + estadoForma +
            '}';
}
```

¿Para qué sirve la anotación `@Override`?

Principios de la POO

- La programación orientada a objetos es un paradigma de programación que se basa en la creación e interacción de objetos especializados en tareas concretas para resolver problemas de software.
- Para diseñar sistemas eficientes y escalables se fundamenta en 4 principios básicos:
  - Abstracción
  - Encapsulamiento
  - Herencia
  - Polimorfismo



Principio de abstracción

- La mayoría de los problemas de software que se quieren resolver suponen **simular** a través de un ordenador **un sistema** que existe en la realidad.
- Estos sistemas pueden llegar a ser muy complejos. Una de las tareas que debemos afrontar como programadores es **identificar** qué **partes** son la realmente **relevantes** para resolver el problema. A este proceso lo denominamos **abstracción**.
- En este proceso de abstracción deberemos **definir** las **características específicas** (atributos) y las **acciones** que deberá realizar (métodos), para llevar a cabo la simulación.
- El sistema real probablemente poseerá muchas más propiedades y será capaz de realizar más acciones, pero debemos centrarnos sólo en aquellas que son necesarias para resolver el problema y abstraernos del resto.

Principio de encapsulamiento

- Una vez identificadas cada una de las partes importantes del sistema y sus atributos, debemos diseñar las clases de forma que los objetos que creemos de dichas clases **protejan los datos** que manejan y **oculten la información interna**.
- Cada objeto debe ser responsable de su propia información y de su propio estado. La única forma en que la información **podrá ser modificada** será **mediante** los **propios métodos del objeto**.
- Por lo tanto, las **propiedades internas** de un objeto serán **inaccesibles directamente** desde fuera del objeto. La única forma de modificarlas será llamando a los métodos correspondientes.

Métodos Getters y Setters

- Con el objetivo de preservar el principio de **encapsulamiento** el acceso y modificación de los atributos sólo podrá realizarse a través de métodos.
- De esta forma **la clase propietaria** de los datos **puede establecer la reglas de acceso y modificación de los atributos**. Ningún otro objeto podrá cambiar sus atributos directamente, sinó que tendrá que utilizar los métodos getters y setters correspondientes.
- Los métodos que permiten **obtener los valores** de los atributos se llaman **getters** y los métodos que permiten **dar valores** a los atributos **setters**.
- Por defecto siempre implementaremos los getters, pero los setters sólo los implementaremos para aquellos atributos que necesiten cambiar su valor después de su inicialización en el constructor.
  - Getters → siempre
  - Setters → sólo si necesario

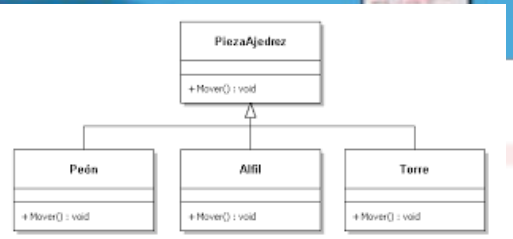
Métodos Getters y Setters

- Existe un convenio de nomenclatura para los getters y setters.
- El nombre de los métodos getters se construye con la palabra get seguida (sin espacios en blanco) del nombre del atributo, que al ser otra palabra empezará en mayúsculas. Por ejemplo para el atrituto `pase` el método getter podría ser:

```
public float getPase() {
    return pase;
}
```

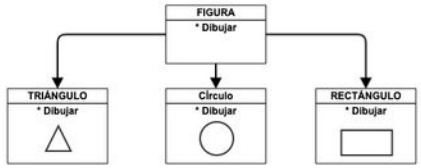
- Los métodos setters recibirán como parámetro el valor que se quiere asignar al atributo (que deberá del mismo tipo).
- El nombre de los métodos setters se construye con la palabra set seguida (sin espacios en blanco) del nombre del atributo con la inicial en mayúsculas. Para el atributo `pase` el método setter podría ser:

```
public void setPase(float pase) {
    this.pase = pase;
}
```



Principio de herencia

- La herencia es el principio que permite crear nuevas clases a partir de clases existentes, **reutilizando** el código y los comportamientos de sus ancestros.
- La nueva clase se conoce como **subclase** o derivada, mientras que la clase original se llama superclase o base.
- La herencia permite crear objetos especializados a partir de objetos más generales, y añadir o modificar sus atributos y comportamientos de manera independiente. Además, **reduce la duplicación de código** y **aumenta la eficiencia** y la **legibilidad** del programa.

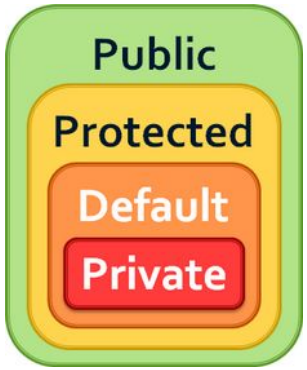


Principio de polimorfismo

- El polimorfismo es el principio que permite a los objetos comportarse de diferentes formas según su tipo o contexto.
- El polimorfismo aumenta la flexibilidad y la compatibilidad de los objetos, ya que permite interactuar con ellos de manera genérica y predecible, sin tener que conocer los detalles internos de su implementación. También permite extender y modificar las funcionalidades del programa de forma modular y escalable.
- Como definición el polimorfismo es la habilidad de un objeto de realizar una acción de diferentes maneras, utilizando métodos iguales que se implementen de forma diferente en varias clases.

Modificadores

- Con el objetivo de facilitar la implementación de los principios de POO, la mayoría de lenguajes definen unos modificares que permiten establecer los criterios de acceso.
- Podemos encontrar modificadores para:
  - Clases
  - Clases internas
  - Atributos
  - Métodos



Modificadores de ámbito para clases

Modificador	Definición
`public`	La clase es accesible desde otros paquetes
`(por defecto)`	La clase sólo es visible en el paquete (carpeta) donde ha sido declarada

Otros modificadores para clases

Modificador	Definición
`final`	Ninguna clase puede heredar de un clase final. Lo veremos en próximos temas.
`abstract`	La clase no puede ser instanciada. Muy útil para definir comportamientos comunes en jerarquías de clases. Lo veremos en próximos temas.

Clase interna

- Una clase interna es una **clase definida dentro de otra clase**.
- Esto significa que dentro de un archivo .java podemos tener varias clases. Una actuará como la clase “externa” y el resto serán clases internas.
- La existencia de un objeto de una clase interna está supeditada a la creación de un objeto de la clase externa, a no ser que la clase interna lleve el modificador de acceso `static`.

```
public class LibArrays {

    private class ArrayStats {

    }

}
```

- En este caso la creación de un objeto de la clase `ArrayStats` está supeditada a la creación de un objeto de la clase `LibArrays`.

Modificadores de ámbito para clases internas

Modificador	Definición
`public`	La clase es accesible desde otros paquetes
`protected`	La clase será visible en todas las clases declaradas en el mismo paquete (carpeta) y, además en las subclases.
`(por defecto)`	La clase sólo es visible en el paquete (carpeta) donde ha sido declarada
`private`	La clase solo es visible en el archivo donde está definida

Otros modificadores para clases internas

Modificador	Definición
`final`	Ninguna clase puede heredar de un clase final. Lo veremos en próximos temas.
`abstract`	La clase no puede ser instanciada. Muy útil para definir comportamientos comunes en jerarquías de clases. Lo veremos en próximos temas.
`static`	La clase puede ser instanciada sin depender de la existencia de un objeto de la clase externa.

Modificadores de ámbito para atributos

Modificador	Definición
`public`	El atributo es accesible desde cualquier lugar
`protected`	El atributo es accesible dentro del paquete (carpeta) donde está definido y, además en las subclases.
`(por defecto)`	El atributo sólo es visible en el paquete (carpeta) donde ha sido declarado.
`private`	El atributo sólo es accesible desde el archivo donde ha sido declarado.

Otros modificadores para atributos

Modificador	Definición
`final`	El valor del atributo es constante y por tanto no puede ser modificado una vez asignado.
`static`	El atributo pertenece a la clase (plantilla/molde) y por tanto su valor es compartido por todos los objetos que se creen de esa clase.
`transient`	El atributo no se serializará. Lo veremos en cuando en el tema de archivos.
`volatile`	Las modificaciones atributo del atributo deben reflejarse en memoria de forma inmediata. Esto evita que varios hilos lean valores distintos. Lo veremos en el tema de hilos de ejecución.

Modificadores de ámbito para métodos

Modificador	Definición
`public`	El método es accesible desde cualquier lugar
`protected`	El método es accesible dentro del paquete (carpeta) donde está definido y, además en las subclases.
`(por defecto)`	El método sólo es visible en el paquete (carpeta) donde ha sido declarado.
`private`	El método sólo es visible en el archivo donde ha sido declarado.

Otros modificadores para métodos

Modificador	Definición
`final`	El método no puede ser sobreescrito por las subclases. Lo veremos en el tema de herencia y polimorfismo.
`abstract`	El método no se va a implementar en la clase actual sino que será implementado por las subclases. Sólo se puede utilizar en métodos de clases abstractas.
`static`	El método pertenece a la clase (plantilla/molde) y por tanto sólo puede acceder a los atributos que sea también static.
`synchronized`	El método está sincronizado. Garantiza que no habrá dos hilos ejecutando este método al mismo tiempo. Lo veremos en el tema de hilos de ejecución.

Resumen niveles de visibilidad

Modificador	Archivo	Paquete (carpeta)	Subclase (mismo paquete)	Subclase (diferente paquete)	Todos
`public`	SI	SI	SI	SI	SI
`protected`	SI	SI	SI	SI	NO
`(por defecto)`	SI	SI	SI	NO	NO
`private`	SI	NO	NO	NO	NO

Resumen sintaxis clases

```
[modificador_acceso] class NombreClase {

    // Lista de atributos/propiedades
    [modificador_ámbito][static][final][transient][volatile] tipo atributo1;

    // Métodos constructores
    public NombreClase() {

    }

    // Métodos
    [modificador_ámbito][static][final][synchronized] tipoRetorno nombreMetodo() {

    }

    @Override
    public String toString() {

    }
}
```

Los veremos en próximos temas

Resumen diseño de clases

- El diseño de clases es un proceso iterativo que debemos realizar antes de ponernos a escribir código ya que, es poco probable que demos con un buen diseño en la primera iteración.
- Recuerda que el objetivo es simular un sistema real pero sólo al nivel de detalle necesario para resolver el problema. Para ello debemos realizar abstracciones siguiendo los siguientes pasos:
  - Identificar las partes (clases) relevantes en las que dividiremos el problema. Recuerda que deben ser partes especializadas en resolver un problema concreto.
  - Identificar los datos (atributos) relevantes de cada una de estas partes.
  - Identificar las acciones (métodos) relevantes de cada una de estas partes.
  - Identificar las interacciones entre las diferentes partes.

¿Cómo implementarías una clase llamada Equipo que represente los jugadores de un equipo de fútbol?

Equipo

```
public class Equipo {
    private static final int MAX_JUGADORES = 25;
    private final Jugador[] jugadores;
    private int numeroJugadores;

    public Equipo() {
        jugadores = new Jugador[MAX_JUGADORES];
        numeroJugadores = 0;
    }

    public boolean addJugador(String nombre, float portero, float defensa, float pase, float regate, float tiro, float
estadoForma) {
        if (numeroJugadores < MAX_JUGADORES) {
            jugadores[numeroJugadores] = new Jugador(nombre, portero, defensa, pase, regate, tiro, estadoForma);
            numeroJugadores++;
            return true;
        }
        return false;
    }

    public void entrenar() {
        for (Jugador j : jugadores) {
            j.entrenarPase();
        }
    }
    // Resto de métodos

    @Override
    public String toString() {
        return "Equipo{" +
                ", jugadores=" + Arrays.toString(jugadores) +
                ", numeroJugadores=" + numeroJugadores +
                '}';
    }
}
```

Agrupación de clases

- Como hemos visto en temas anteriores, los paquetes permiten agrupar las clases en carpetas.
- En POO los paquetes cobran mayor importancia ya que nos permite realizar y estructurar mejor el proceso de abstracción.
  - Estructurar las clases en grupos relacionados
  - Aprovechar la visibilidad a nivel de paquete
  - En proyectos de tamaño medio y grande, permite modularizar la aplicación de forma que podemos asignar cada módulo a un equipo de programación diferente.

Recapitulando

- Para repasar los conceptos de este tema, vamos a terminar con otro ejemplo implementando una clase que represente un semáforo de tráfico, teniendo en cuenta que sólo necesitamos saber el color y que las transiciones entre colores se hagan correctamente.

- ¿Cómo llevarías a cabo la implementación?

Abstracción del problema

- Según la descripción del problema no necesitamos simular todas las particularidad internas de un semáforo.
- Sólo necesitamos conocer su color y que al cambiar de color la transición sea válida.
- Los colores del semáforo sólo pueden ser rojo, amarillo o verde. Por tanto, una buena solución para representar el color sería utilizar un **enumerado**.
- Para cambiar el color tenemos dos opciones:
  - Establecer un **setter** para el atributo color.
  - Establecer un **método** que se encargue de realizar las **transiciones** entre colores.

  ¿Cuál de estas dos opciones piensas que es mejor?

Semaforo.java

```
public class Semaforo {
    public enum Color {
        ROJO, VERDE, AMARILLO
    }

    private Color color;

    public Semaforo(Color color) {
        this.color = color;
    }

    public void siguiente() {
        int indiceSiguiente = (color.ordinal() + 1) % Color.values().length;
        color = Color.values()[indiceSiguiente];
    }

    @Override
    public String toString() {
        return "Semaforo{" +
                "color=" + color +
                '}';
    }
}
```

Ampliando la funcionalidad

- Ahora nos piden modificar la clase semáforo para que tenga un identificador único y cambie de color automáticamente pasados un determinado número de ticks de reloj.
- Cada semáforo deberá poder ser configurado con un número de ticks distinto en el momento de la creación.

- ¿Qué cambios deberemos llevar en la clase `Semaforo`?

```java
public class Semaforo {
    public enum Color {
        ROJO, VERDE, AMARILLO
    }
    private static final int TICKS_TRANSICION_AMARILLO = 3;
    private static int contadorId = 0;
    private final int id;
    private Color color;
    private final int ticksTransicion;
    private int ticks;

    public Semaforo(Color color, int ticksTransicion) {
        this.id = ++contadorId;
        this.color = color;
        this.ticksTransicion = ticksTransicion;
        ticks = 0;
    }

    public boolean tick() {
        ticks++;
        boolean cambio = false;
        if (color == Color.AMARILLO) {
            if (ticks == TICKS_TRANSICION_AMARILLO) {
                siguiente();
                cambio = true;
            }
        } else {
            if (ticks == ticksTransicion) {
                siguiente();
                cambio = true;
            }
        }
        return cambio;
    }

    private void siguiente() {
        int indiceSiguiente = (color.ordinal() + 1) % Color.values().length;
        color = Color.values()[indiceSiguiente];
        ticks = 0;
    }

    @Override
    public String toString() {
        return "Semaforo{" +
                "id=" + id +
                "color=" + color +
                '}';
    }
}
```

Calle de semáforos

- Basándonos en el semáforo anterior, nos piden que simulemos los semáforos de una calle sabiendo que no importa la posición que ocupen. Sólo nos interesa su funcionalidad.

```java
public class Calle {
    private static final int MAX_SEMAFOROS = 100;
    private final Semaforo[] semaforos;
    private int contadorSemaforos;

    public Calle() {
        semaforos = new Semaforo[MAX_SEMAFOROS];
        contadorSemaforos = 0;
    }

    public boolean addSemaforo(Semaforo.Color color, int ticksTransicion) {
        if (contadorSemaforos < MAX_SEMAFOROS) {
            semaforos[contadorSemaforos] = new Semaforo(color, ticksTransicion);
            contadorSemaforos++;
            return true;
        }
        return false;
    }

    public void simular() {
        while (true) {
            for (int i = 0; i < contadorSemaforos; i++) {
                Semaforo semaforo = semaforos[i];
                if (semaforo.tick()) {
                    System.out.println(semaforo);
                }
            }
        }
    }

    @Override
    public String toString() {
        return "Calle{" +
                ", semaforos=" + Arrays.toString(semaforos) +
                ", contadorSemaforos=" + contadorSemaforos +
                '}';
    }
}
```