Entornos de desarrollo
Bloque 2
Tema 6: Clases y objetos en Java. Herencia. Generación automática de componentes

por Nacho Iborra

2.6.1. Clases y objetos en Java

En unidades anteriores aprendimos que es una clase, incluso como definirla en Java. Recuerda la clase `Vehicle` que implementamos, con sus atributos o variables instanciadas, su constructor (o constructores) y métodos.

```
class Vehicle {

    int passengers;
    int fuelcap;
    int kml;

    public Vehicle(int p, int f, int k) {
        passengers = p;
        fuelcap = f;
        kml = k;
    }

    private int autonomy() {
        return fuelcap * kml;
    }

    public void vehicleData() {
        System.out.println("This vehicle has an autonomy of "
            + autonomy() + " km.");
    }
}
```

En Java, cada fichero fuente debe tener una clase ***public*** con el mismo nombre que el fichero fuente. En otras palabras, si creamos un fichero llamado `MyClass.java`, éste deberá tener una clase pública llamada `MyClass`. También pueden haber otras clases (no públicas), aunque esto no es habitual, a menos que queramos todo nuestro proyecto en un único fichero:

```
// File MyClass.java

public class MyClass
{
    ... // Code of MyClass
```

```
}

class MyOtherClass
{
    ... // Code of MyOtherClass
}

...
```

2.6.1.1. Elementos principales de una clase Java

Una clase Java tiene los elementos comunes que tiene cualquier otra clase en otro lenguaje de programación orientado a objetos:

- **Variables de instancia o atributos** para almacenar la información de los diferentes objetos que creamos a partir de la clase.
- **Constructores** para inicializar los datos de los objetos cuando los creamos.
- **Métodos** para hacer algunas tareas relacionadas con los objetos o clase en sí. Un conjunto específico de métodos son los conocidos como *getters* y *setters*, los cuales se utilizan para recuperar o establecer valores en los diferentes atributos. Este tipo específico de métodos tiene el prefijo *get* o *set* respectivamente, seguido por el nombre del atributo al que hace referencia.

Este ejemplo mejora la versión anterior de la clase `Vehicle` añadiendo algunos *getters* y *setters* para recuperar/obtener algunos de sus atributos:

```
class Vehicle {

    int passengers;
    int fuelcap;
    int kml;

    public Vehicle(int p, int f, int k) {
        passengers = p;
        fuelcap = f;
        kml = k;
    }

    private int autonomy() {
        return fuelcap * kml;
    }

    public void vehicleData() {
        System.out.println("This vehicle has an autonomy of "
            + autonomy() + " km.");
    }

    public int getPassengers() {
```

```
        return passengers;
    }

    public void setPassengers(int p) {
        passengers = p;
    }
}
```

Podríamos utilizas esta clase en otra parte del código instanciando un objeto `Vehicle` con el constructor, y luego llamar a algunos de sus métodos.

```
Vehicle myCar = new Vehicle(5, 50, 30);
System.out.println("Passengers: " + myCar.getPassengers());
myCar.vehicleData();
```

2.6.1.2. Más acerca de los constructores

Si no definimos ningún constructor en nuestra clase, Java automáticamente añade un constructor por defecto (sin código ni parámetros), de forma que podamos instanciar objetos de nuestra clase de todos modos. Por ejemplo, si tenemos esta clase simple:

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

Podemos crear un objeto `Person` como este, incluso si no hemos especificado un constructor:

```
Person p = new Person();
```

Sin embargo, si establecemos un constructor a nuestra clase, el constructor por defecto añadido automáticamente dejará de estar disponible. En otras palabras, si añadimos este constructor a nuestra clase:

```
public class Person {

    String name;
    int age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }
}
```

Nos vemos forzados a instanciar los objetos de nuestra clase con nuestro constructor (o cualquier otro constructor que hayamos declarado explícitamente):

```
Person p = new Person("Nacho", 40);   // OK
Person p2 = new Person();             // ERROR!
```

2.6.1.3. Objectos y *arrays*

Podemos definir un *array* de objetos de una clase, como hicimos con datos primitivos en temas anteriores. Por ejemplo, así es como podríamos crear un *array* para almacenar hasta 10 vehículos:

```
Vehicle[] vehicles = new Vehicle[10];
```

Sin embargo, necesitamos instanciar (crear) un nuevo objeto para cada posición del *array* a la hora de añadirlo:

```
vehicles[0] = new Vehicle(5, 80, 20);
vehicles[1] = new Vehicle(4, 60, 15);
...

// Or with a loop:
for (int i = 0; i < vehicles.length; i++)
{
    vehicles[i] = new Vehicle(...);
}
```

> **Ejercicios propuestos**
>
> **2.6.1.1.** Crea un programa llamado *ListaVideoJuego* para almacenar objetos de una clase llamada *VideoJuego* que deberás definir. Para cada videojuego, almacenaremos su título, género y precio. Añade también los correspondientes *getters*, *setters* y constructores para establecer los valores. Define el *main*, la clase pública y la clase *VideoJuego* en el mismo fichero fuente. En el método *main* crea un *array* de 5 videojuegos, pide al usuario que rellene la información de cada videojuego y, muestra el título del videojuego más barato y más caro del *array*.

2.6.2. Asociaciones entre clase

La **asociación** es la relación entre dos clases, en la cual una de las dos utiliza la otra, es decir, un objeto de una clase es un atributo o variable de instancia de la otra clase. Esto se representa normalmente en el código con una referencia al objeto contenido o a una colección o un *array* de esos objetos. En el siguiente ejemplo podemos ver una clase llamada `Classroom` que contiene/utiliza un *array* de objetos de tipo `Student`:

```
public class Classroom {
    private Student[] students;
```

```
    ...
}
```

Podemos establecer una relación ***Has-A*** (Tiene un) entre estas dos clases. En ejemplos anteriores, por ejemplo, podemos decir que una `Classroom` *has a* (tiene una) lista de `Students`, por lo tanto definimos una asociación entre ambas clases.

Las asociaciones son (o pueden ser) bidireccionales, es decir, podemos tener variables instanciadas de ambas clases en la otra clase. En ejemplos anteriores, también podemos tener un elemento `Classroom` en la clase `Student`, de modo que podemos verificar fácilmente la clase a la cual pertenece el estudiante.

```
public class Student {

    Classroom studentClass;
    ...
}
```

El programador puede decidir si una asociación debe ser bidireccional o no, por tanto, sólo una de las clases (o ambas) estará relacionada con la otra.

2.6.2.1. Agregaciones y composiciones

Hay dos tipos especiales de asociaciones: composición y agregación. En ambos casos, una de las clases es considerada como el todo, y la otra clase como parte de ese todo. Pero, ¿cómo distinguir entre composición y agregación? Veamos unos sencillos ejemplos:

- **Composición**: la utilizamos cuando un objeto es una parte indivisible de otro objeto. Por ejemplo, una `Room` es parte de una `House` (y sólo de esa casa), un `Square` es parte de un `Chessboard`, y así sucesivamente. La principal característica de este tipo de relación es que cuando destruimos el objeto principal (el todo), todos los objetos que son parte él son también destruidos.
- **Agregación**: la utilizaremos cuando un objeto es parte de otro objeto (o tal vez parte de o más objetos) y puede existir sin el objeto que lo contiene. Un ejemplo de esto podría ser un `Player`, el cual es parte de un `Team` (o tal vez más), o un `Student`, el cual pertenece a una `Classroom` (o más). En estos casos, cuando el `Team` o la `Classroom` ya no existen, jugadores y estudiantes continúan existiendo y, pueden unirse a otro equipo o clase.

Composición y agregación en la práctica

En la práctica, la forma en la que definiremos la agregación o la composición dependerá del lenguaje de programación que vayamos a utilizar. Pero, en general, si los atributos internos, o variables de instancia, que forman la composición o agregación no pueden ser accedidos desde fuera de la clase que los contiene, tenemos una composición. De otra forma, tendremos una agregación. Veamos esto con el siguiente ejemplo: definimos la clase `Car` que tiene un objeto de tipo `Engine`. Si queremos definir una composición entre estas clases, lo haríamos de la siguiente manera:

```
class Car {

    private final Engine engine;

    Car(EngineParams params) {
        engine = new Engine(params);
    }
}
```

Tened en cuenta que creamos el objeto `Engine` en el interior de la clase `Car`, utilizando los parámetros especificados en el objeto `EngineParams`. En este caso, si el objeto `Car` es destruido, entonces el objeto `Engine` será también destruido. Por lo tanto, esto es una composición.

Sin embargo, si necesitamos definir una agregación entre la clase `Car` y la clase `Engine`, deberemos hacerlo así:

```
class Car {

    private Engine engine;

    void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

En este caso, estamos utilizando un objeto externo de tipo `Engine` para crear el objeto interno `Engine` del coche, por lo tanto el motor puede existir sin el coche: si destruimos el coche, el motor externo que utilizamos en el constructor seguirá existiendo. Esto puede ser útil si queremos utilizar el motor en otro coche, una vez destruido el anterior.

> **Ejercicios propuestos**
>
> **2.6.2.1** Mejora el ejercicio anterior *ListaVideoJuego* en otro fichero fuente llamado *ListaVideoJuego2*. Ahora, cada videojuego deberá tener una **Compañía** (clase) que lo ha creado. Para cada compañía, necesitamos almacenar su nombre y año de fundación. Asocia una compañía a cada videojuego, de forma que algunos videojuegos puedan compartir el mismo objeto compañía. Luego, después de pedir al usuario que introduzca 5 videojuegos y mostrar el más barato y el más caro, muestra la compañía que aparece en la mayor parte de los videojuegos de la lista.

2.6.3. Herencia de clase

Utilizamos **herencia** cuando queremos crear una nueva clase que recoge todas las características de otra, añadiendo sus particularidades. Por ejemplo, si tenemos una clase `Animal` con un conjunto de atributos (nombre, peso, ...) y métodos, podemos heredar para crear una nueva clase llamada `Dog` que tendrá también todas esas características, y podemos añadir algunas como el método `bark()`.

Hemos visto en secciones anteriores como identificar una relación de composición o agregación, encontrando una relación *Has-A* (tiene un) entre las clases involucradas. Cuando hablamos de herencia, la identificamos con la relación ***Is-A***, por lo tanto una clase es un subtipo de otra clase. En otras palabras, comparte las características de su "*padre*" e introduce algunas nuevas. Un ejemplo de esto es `Car`, el cual es un subtipo de `Vehicle`. Otro podría ser `ComputerClassroom`, que sería un subtipo de `Classroom` con ordenadores.

Cuando queremos que una clase herede las características de otra clase en Java utilizaremos la palabra reservada `extends` en la nueva clase (también llamada hijo o *subclase*), haciendo referencia a la clase de la cual queremos extender (también conocida como clase padre o *superclase*). En ejemplos anteriores, podríamos definir una nueva clase `Dog` como sigue:

```
public class Dog extends Animal {
    ...
}
```

Todos los atributos y métodos de la clase padre son heredados y pueden ser utilizados por la clase descendiente. Por ejemplo, si tenemos una clase `Store` con un método de bienvenida, podemos heredar dicho método y utilizarlo en la clase descendiente:

```
public class Store {
    public void welcome() {
        System.out.println("Welcome to our store!");
    }
}

public class LiquorStore extends Store {
    ...
}

public class MainClass {

    public static void main(String[] args) {
        LiquorStore lqStore = new LiquorStore();
        lqStore.welcome();
    }
}
```

2.6.3.1. Extendiendo funcionalidades del padre

Con respecto a la herencia, podemos anular (***override***) un método de la clase padre en la clase hijo, esto significa que podemos redefinir su funcionalidad en lugar de mantener la del padre.

```
public class LiquorStore extends Store {
    @Override
    public void welcome() {
        System.out.println("If you are younger than 18,
```

```
                                go back home!");
        }
    }
```

Remarcar que utilizamos la palabra `@Override`. Esto es una **anotación**, e indica que existe otro método llamado `welcome` en la clase padre, y que queremos redefinir su comportamiento. Aprenderás más sobre las anotaciones más adelante en este tema.

Extendiendo la clase *Object*

Debemos tener en cuenta que todas las clases en Java se heredan de una clase padre global llamada `Object`. Por tanto, si nuestra clase no hereda de cualquier otra clase, automáticamente será hijo de la clase `Object`, por lo que podrá utilizar o reemplazar métodos de dicha clase, como `equals` o `toString`.

Si reemplazamos el método `toString`, podemos convertir nuestros objectos a *String*, por ejemplo, para imprimirlos fácilmente. Supongamos que reemplazamos este método para nuestra clase `Person` vista en ejemplos anteriores, de tal forma que devolvamos una cadena con el nombre de la persona y su edad entre paréntesis:

```
public class Person {

    String name;
    int age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }

    @Override
    public String toString() {
        return name + " (" + age + " years)";
    }
}
```

Entonces, podemos imprimir fácilmente cualquier objeto `Person` con una simple llamada a la sentencia `System.out.println`:

```
Person p = new Person("Nacho", 40);
System.out.println(p); // Prints "Nacho (40 years)"
```

De la misma forma, también podemos reemplazar el método `equals` para determinar si dos objetos de tipo `Person` son iguales o no. En este ejemplo estamos diciendo que son iguales si tienen el mismo nombre y la misma edad:

```

```

```
public class Person {

    String name;
    int age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }

    @Override
    public String toString() {
        return name + " (" + age + " years)";
    }

    @Override
    public boolean equals(Object p) {
        Person p2 = (Person) p;
        return this.name.equals(p.name) && this.age == p.age;
    }
}
```

Entonces, podemos comparar dos objetos `Person` y determinar si son o no iguales:

```
Person p1 = new Person("Nacho", 40);
Person p2 = new Person("Nacho", 39);

if (p1.equals(p2)) {
    System.out.println("They are equal!");
} else {
    System.out.println("They are different");
}
```

2.6.3.2. Uso de *this* y *super*

En toda clase que implementemos, podemos utilizar la palabra reservada ***this*** para hacer referencia a cualquier elemento interno de la clase, ya sea un atributo, un método o un constructor. Su principal uso se basa en los constructors, para distinguir entre los atributos y los parámetros del constructor cuando tienen el mismo nombre:

```
class Vehicle {

    int passengers;
    int fuelcap;
    int kml;

    public Vehicle(int passengers, int fuelcap, int kml) {
```

```
        // this.passengers refers to passengers attribute
        this.passengers = passengers;
        // this.fuelcap refers to fuelcap attribute
        this.fuelcap = fuelcap;
        // this.kml refers to kml attribute
        this.kml = kml;
    }
    ...
}
```

Pero también podemos utilizar ***this*** en cualquier otra parte de nuestro código:

```
class Vehicle {

    ...

    public void vehicleData() {
        System.out.println("This vehicle has an autonomy of "
            + this.autonomy() + " km.");
    }
}
```

Además, podemos utilizar la palabra reservada ***super*** en una clase hijo para hacer referencia a cualquier elemento visible de la clase padre. Normalmente llamaremos a ***super*** para reutilizar código de la clase padre en punto concreto. Puede ser un método...

```
public class LiquorStore extends Store {
    @Override
    public void welcome() {
        super.welcome();    // Print parent's message as well
        System.out.println("If you are younger than 18, go back home!");
    }
}
```

... o incluso en el constructor:

```
public class Car extends Vehicle {
    int numberOfDoors;

    public Car(int passengers, int fuelcap,
                    int kml, int numberOfDoors) {
        super(passengers, fuelcap, kml);
        this.numberOfDoors = numberOfDoors;
    }

    public int getNumberOfDoors() {
```

```
        return numberOfDoors;
    }
}
```

En este último caso, estamos utilizando `super` para asignar atributos heredados desde la clase padre, en lugar de hacerlo nuevamente en la clase hijo.

2.6.3.3. Polimorfismo y herencia

Hay varios tipos de polimorfismo, pero en lo que respecta a la programación orientada a objetos, **polimorfismo** es la habilidad de un objeto para comportarse como otro objeto. Este término es comúnmente utilizado en herencia para mostrar que un objeto de cualquier clase puede comportarse como cualquiera de sus subclases. Por ejemplo, un objeto `Vehicle` de ejemplos anteriores podría comportarse como un objeto `Car`, por tanto podemos, por ejemplo:

- Instanciar un objeto `Car` en una variable `Vehicle`:

```
Vehicle myCar = new Car(...);
```

- Utilizar un objeto `Car` como parámetro en un método que recibe un objeto `Vehicle`:

```
public void aMethod(Vehicle v) {
    ...
}

...
Car anotherCar = new Car(...);
aMethoc(anotherCar);
```

- Rellenar un *array* de objetos `Vehicle` con cualquier subtipo de `Vehicle` en cada posición:

```
Vehicle[] vehicles = new Vehicle[10];

vehicles[0] = new Vehicle(...);
vehicles[1] = new Car(...);
vehicles[2] = new Van(...);
...
```

Sin embargo, debemos tener en cuenta que, cuando utilizamos polimorfismo, la variable polimórfica sólo puede acceder a los métodos del tipo al cual pertenece. En otras palabras, si creamos un objeto `Car` y lo almacenamos en una variable `Vehicle`, entonces sólo podremos llamar a métodos o elementos públicos de la clase `Vehicle` (no de la clase `Car`).

```

```

```
Vehicle myCar = new Car(...);
myCar.vehicleData();                              // OK
System.out.println(myCar.getNumberOfDoors()); // ERROR
```

Si queremos detectar el tipo concreto de un objeto para poder acceder a sus propios métodos (y no sólo a los que hereda del padre), entonces debemos utilizar el operador `instanceof`, y después hacer una conversión *typecast* al tipo concreto para usar sus métodos:

```
Vehicle[] vehicles = new Vehicle[10];
... // Fill the array with many vehicle types
for (int i = 0; i < vehicles.length; i++)
{
    if (vehicles[i] instanceOf Car)
    {
        System.out.println(((Car)vehicles[i]).getNumberOfDoors());
    } else if (vehicles[i] instanceOf Van) {
        ...
    } ...
}
```

2.6.3.4. Constructores y herencia

Cuando llamamos a un constructor desde una subclase, el constructor por defecto (p.e. uno sin parámetros) de la superclase es llamado automáticamente (a menos que usemos `super` para elegir otro constructor). Por lo que el siguiente código no compilará, ya que no hay un constructor por defecto en la clase `Animal`.

```
class Animal {
    public Animal(String name) {
        System.out.println("I am an animal called " + name);
    }
}

class Dog extends Animal {
    public Dog(String name) {
        System.out.println("I am a dog called " + name);
    }
}
```

Para solucionar el problema, debemos añadir un constructor por defecto a la clase `Animal`...

```
class Animal {
    public Animal() {
        ...
    }

    public Animal(String name) {
```

```
        System.out.println("I am an animal called " + name);
    }
}
```

... o elegir otro constructor desde la clase `Dog` mediante la clausula `super`:

```
class Dog extends Animal {
    public Dog(String name) {
        super(name);
        System.out.println("I am a dog called " + name);
    }
}
```

2.6.3.5. Modificadores de acceso y visibilidad

Hemos visto en el tema anterior que hay tres **niveles de visibilidad** para los elementos de una clase. Respecto a Java, podemos hablar de cuatro niveles de visibilidad:

- **public** son aquellos elementos que pueden ser accedidos desde cualquier otra parte del código (incluidas otras clases y paquetes).
- **protected** son los elementos que sólo pueden ser accedidos desde cualquier subclase de la clase actual, o cualquier clase desde el mismo paquete de la clase actual.
- **private** estos elementos sólo pueden ser accedidos desde la clase actual.
- Además de estos tres niveles de visibilidad típicos, Java añade un cuarto nivel, considerado nivel ***package***. Si no especificamos ninguno de los niveles anteriores, se le asigna este cuarto nivel al elemento. Esto significará que el elemento únicamente será accesible desde el mismo paquete.

Veamos estos modificadores con un ejemplo:

```
public class MyClass
{
    // Accessible everywhere
    public int number;
    // Accessible from subclasses or same package
    protected String name;
    // Only accessible from this class
    private float average;
    // Package level, accessible from same package
    char symbol;

    // Public method, accessible everywhere
    public float calculate() {
        ...
    }
```

```
    // Protected method, accessible from subclasses or same package
    protected int myMethod() {
        ...
    }

    // Package method, accessible from same package
    void myOtherMethod() {
        ...
    }

    // Private method, accessible only from within this class
    int myPrivateMethod(int number) {
        ...
    }

}
```

Visibilidad y herencia

Cuando heredamos de una clase, únicamente tenemos acceso a los elementos públicos y protegidos (no a los privados). Por tanto, si la clase tiene algún atributo privado, sólo tendremos acceso a éste si la clase proporciona algún *getter* y/o *setter* público/protegido que haga referencia a dicho atributo.

Encapsulación

Declarar cada atributo de la case como ***private*** y asignarle un *getter* y/o un *setter* nos permite ocultar parte del código que puede ser delicada y, mostrar otro que permita manipular la información de manera segura. Por ejemplo, vamos a suponer que tenemos el atributo `age` para representar la edad de una clase `Person`. Si declaramos este atributo como *public*, cualquier otra clase podría acceder a él y cambiar su valor (incluso con números negativos o ¡edades imposibles!). Pero, si lo declaramos como *private* y definimos unos *getter* y *setter* públicos para manipularlo...

```
public int getAge() {
    return age;
}

public void setAge(int age) {
    if (age >= 0 && age <= 120) {
        this.age = age;
    }
}
```

... entonces estaremos seguros que la edad siempre contendrá un valor correcto.

2.6.3.6. Elementos estáticos

El modificado **static** define elementos que pertenecen a la propia clase, y no a ningún objeto específico de la clase. Por ejemplo, si definimos un atributo *static*:

```
public class MyClass {
    static int count = 0;
}
```

Entonces cada objeto que creemos de dicha clase compartirá este atributo con el resto de objetos de la clase. Por lo que un atributo *static* es como una "*variable compartida*".

Con respecto a los métodos, un método *static* es un método que puede ser llamado sin crear ningún objeto de la clase:

```
public class MyClass {
    ...
    public static void myMethod() {
        ...
    }

    public void myOtherMethod() {
        ...
    }

}
...
MyClass.myMethod();          // OK
MyClass.myOtherMethod();     // ERROR

MyClass mc = new MyClass();
mc.myMethod();               // OK (but not necessary)
mc.myOtherMethod();          // OK
```

Teniendo en cuenta que en un método *static* sólo puedes utilizar otros elementos *static* (atributos o métodos), pero no cualquier método *non-static* existente fuera del método. Por ejemplo, esto no podría hacerse, ya `myOtherMethod` no es *static*:

```
    public static void myMethod() {
        myOtherMethod();
    }

    public void myOtherMethod() {
        ...
    }
```

Puedes utilizar muchos otros métodos *static* existentes en Java. Por ejemplo, la clase `Math` tiene muchos de ellos, como `Math.pow(...)`, o `Math.abs(...)`.

2.6.3.7. Elementos *final*

El modificador **final** nos permite definir elementos que no pueden ser modificados. Si aplicamos este modificador a un atributo o variable, estamos definiendo una constante: el elemento no puede ser re-asignado con otro valor:

```
final int number = 3;

// ERROR
number = 6;
```

También podemos aplicar este modificador a métodos, o incluso clases. Si aplicamos este modificador a un método, estamos indicando que este método no puede ser sobrescrito (*override*) por ninguna posible clase hijo. Con respecto a las clases, una clase *final* no puede ser heredada.

> **Ejercicios propuestos**
>
> **2.6.3.1.** Mejora el ejercicio anterior *ListaVideoJuego2* en otro fichero fuente llamado *ListaVideoJuego3*. Añade una nueva clase llamada *PCVideoJuego* la cual hereda de la clase *VideoJuego*. Ésta tendrá dos atributos nuevos llamados *RAMminima* y *HDminimo* para almacenar la cantidad mínima de memoria RAM y de espacio en disco duro requerido para jugar al videojuego (ambos enteros). Define el constructor correspondiente para asignar estos valores (y utiliza `super` para llamar al constructor del padre para establecer los valores heredados). Luego añade algunos videojuegos de PC un *array* y repite los mismos pasos que en el ejercicio anterior.
>
> Sobrescribe también el método `toString` en la clase *VideoJuego* de forma que podamos mostrar en pantalla la información de un videojuego con una simple llamada a `System.out.println`.
>
> **2.6.3.2.** Añade una variable *static* en la clase *VideoJuego* del ejercicio anterior para almacenar cuantos videojuegos (de cualquier tipo) han sido creados. Cada vez que se llame al constructor de *VideoJuego*, incrementa el valor de esta variable. Después de cargar todo el *array*, muestra el valor final del contador.

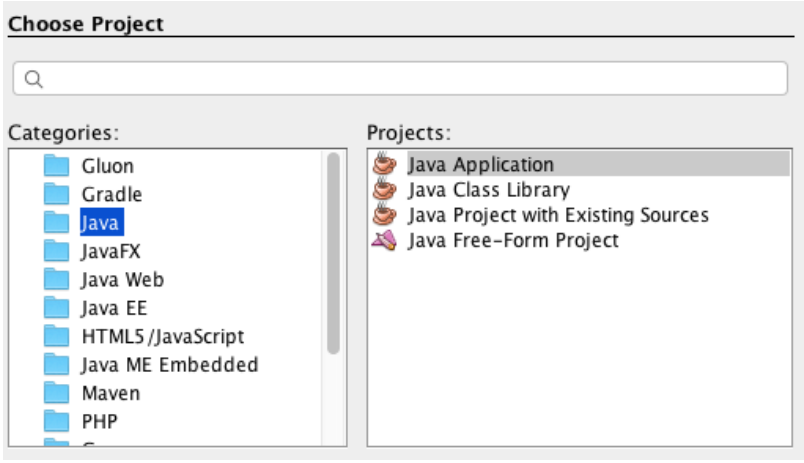
2.6.4. Proyectos Java en Eclipse o NetBeans

Ahora que hemos aprendido que es una clase y las principales relaciones que podemos crear entre ellas, debemos pensar que en un proyecto real normalmente consiste en utilizar varias clases, con sus correspondiente ficheros fuente.

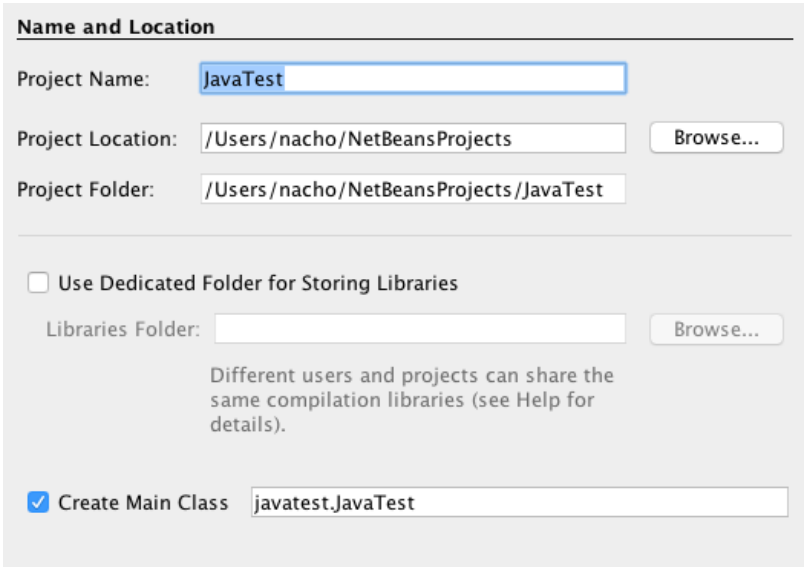
Tan pronto como nuestra aplicación Java tenga más de una clase, los IDEs simples como *Geany* pueden tener algunos problemas para tratar con todas las clases: cada vez que compilemos una clase, necesitamos tener en cuenta el resto de clases relacionadas con ella, y esto puede ser un trabajo duro.

Afortunadamente, IDEs avanzados com Eclipse o NetBeans (entre otros) nos permite manejar proyectos complejos fácilmente. Sólo necesitamos crear un proyecto nuevo desde estos IDEs (generalmente un nuevo proyecto Java), y elegir el nombre del proyecto.

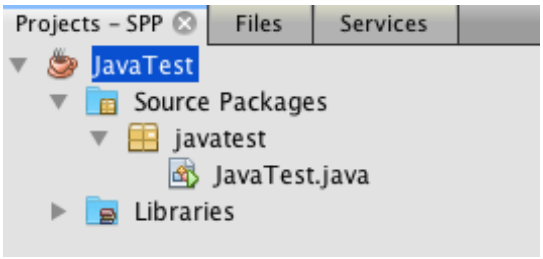
Con respecto a **NetBeans**, crearemos un nuevo proyecto Java desde el menú *File* > *New Project*, y elegiremos el proyecto tipo *Java Application*.



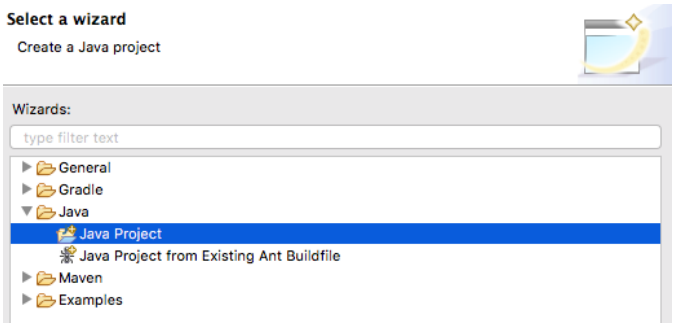
A continuación se nos preguntará por el nombre del proyecto. El resto de campos se pueden dejar con sus valores por defecto. La última opción (*Create Main class*), si está marcada, definirá la clase *main* por defecto con el mismo nombre que el proyecto, y con el método *main* en su interior. Podemos desmarcar esta opción si tenemos pensado crear nuestra propia clase principal más tarde, manualmente.



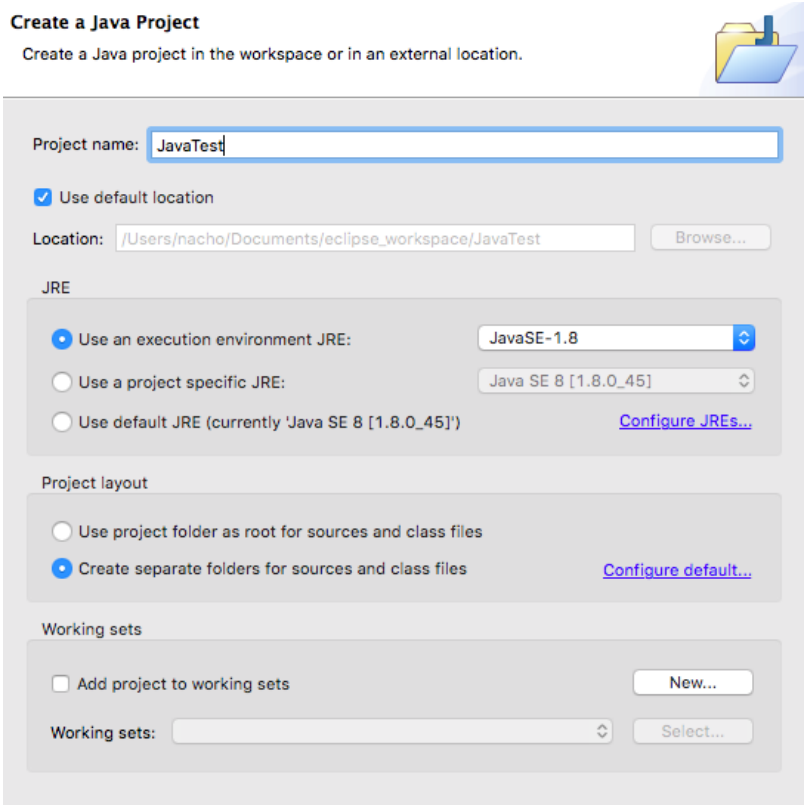
Tras estos pasos, tendremos un nuevo proyecto con el nombre deseado en nuestra ventana de proyecto.



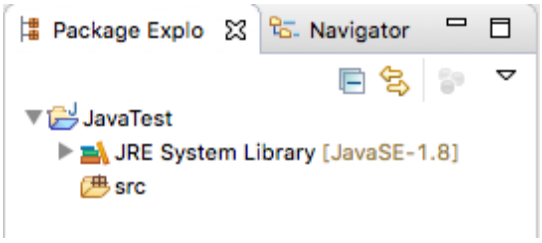
Si utilizamos **Eclipse**, necesitaremos ir al menú *File* > *New* > *Project*, y elegir el asistente *Java* > *Java Project*.



Necesitaremos especificar el nombre del proyecto, y entonces podremos finalizar el asistente.



Tendremos el nuevo proyecto en nuestra ventana de explorador de paquetes:



2.6.4.1. Organizando clases en *packages*

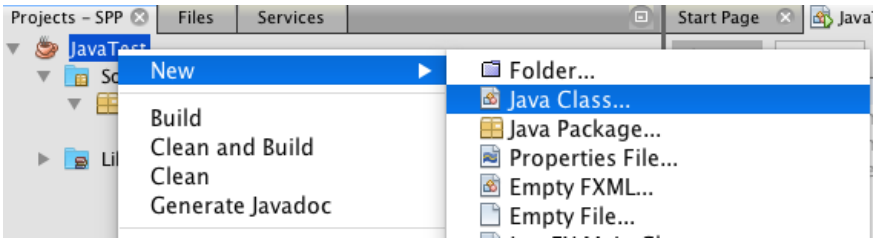
Los ***packages*** son la forma de organizar nuestras clases en un proyecto o librería, por tanto, cada clase (o grupo de clases) pertenece a un ***package*** determinado. En otros lenguajes de programación, como C#, los ***packages*** se conocen como ***namespaces***, pero el propósito es el mismo.

El nombre de un ***package*** consta de una o más palabras, separadas por puntos. Cada punto establece un nuevo subnivel del ***package***. Por tanto, si definimos el ***package*** `javatest`, éste será su primer nivel, pero si

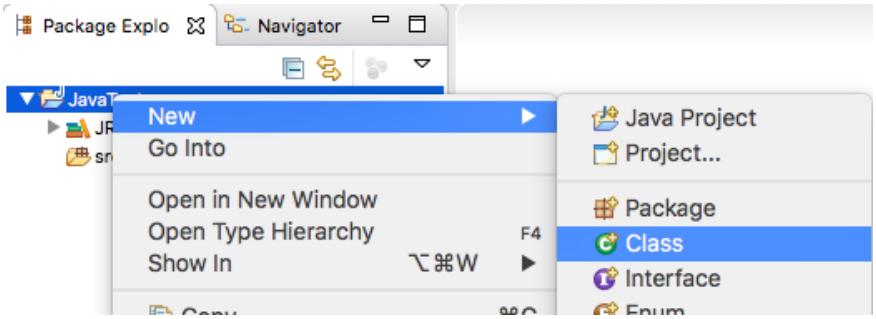
definimos un *package* llamado `mypackage.utils`, entonces el primer nivel será `mypackage`, y en el interior de éste habrá un segundo nivel llamado `utils`.

Cada vez que creamos un *package* en Java, físicamente se crea una nueva carpeta en el disco duro, con el mismo nombre que el *package*. Si el *package* tiene más de una palabra (más de un nivel), entonces, cada *subpackage* tendrá su propia subcarpeta. Según el ejemplo anterior, si definimos el *package* `mypackage.utils`, se creará una carpeta llamada `mypackage`, y dentro se creará una subcarpeta llamada `utils`.

Podemos **añadir clases** a nuestro proyecto *clicando* con el botón derecho sobre el nombre del proyecto (o cualquier *package* interno) y eligiendo *New* > *Java Class* desde el menú contextual (en NetBeans)...



...o *New* > *Class* (en Eclipse).



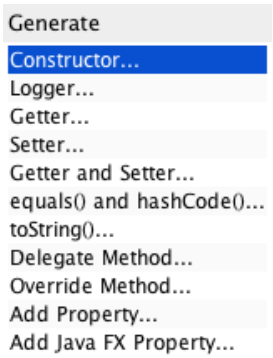
Luego podremos especificar el nombre de la clase y, también el nombre del *package* al cual pertenecerá. Si no especificamos ningún nombre de *package*, será asignado al *package* por defecto (*default*), aunque esta opción no es muy recomendable (siempre deberías asignar un *package* para tus clases).

> **NOTA**: si *clicamos* con el botón derecho en cualquier *package* interno y elegimos añadir una nueva clase, esta clase será asignada a ese *package* por defecto (aunque también podemos cambiar la nombre del *package* en este caso).

2.6.4.5. Generación de código fuente

Si utilizamos IDEs como Eclipse o NetBeans (entre otros), pueden ayudarnos a añadir código automáticamente a nuestras clases. Por ejemplo, si definimos el nombre de una clase y sus variables de instancia o atributos, podemos auto-generar el constructor(es), o los *getters* y *setters*, o sobrecargar (*override*) algunos métodos de la clase padre, si queremos hacerlo.

Desde **NetBeans**, deberemos elegir el menú *Source* > *Insert Code* y elegir el elemento que queremos insertar desde el menú emergente:



Si utilizamos **Eclipse**, únicamente tendremos que seleccionar el menú *Source* y seleccionar el elemento deseado:

- *Generate Getters and Setters*
- *Generate Constructor using Fields*
- ...

2.6.4.6. Uso de anotaciones

Ya conociste la anotación `@Override` cuando extendimos la funcionalidad de una clase padre. Las anotaciones son meta-instrucciones y no se compilarán como el resto de nuestro código. Proporcionan información útil al compilador y diferentes herramientas como: herramientas de generación automática de código, herramientas de documentación (JavaDoc), *Frameworks* para pruebas (*JUnit*, etc), ORM (*Hibernate*), etc.

Se colocan antes de una clase, un atributo o la definición de un método, afectando a lo que viene a continuación. Comienzan con el carácter **'@'**. Algunos ejemplos de anotaciones que el compilador puede interpretar son:

- `@Deprecated`: se aplica a un método y, el compilador muestra un *warning* cada vez que el programador utiliza el dicho método en su código, diciéndole que debería parar de utilizar este método y usar otro en su lugar.
- `@Override`: informa al compilador que el método está sobrescrito. No es necesario utilizar esta anotación, pero si se hace y el método no existe en la clase padre (no es un método sobrescrito), el compilador lanzará un error. Hemos visto un ejemplo de esta anotación cuando hablamos sobre herencia.
- `@SuppressWarnings("type")`: deshabilita el tipo de *warning* indicado dentro de un método, algunos ejemplos de *warnings* son "*unused*" (variables sin utilizar), "*deprecated*" (uso de métodos en desuso) y muchos otros.
- ...

> **Ejercicios propuestos**
>
> **2.6.4.1.** Crea un nuevo proyecto Java en **Eclipse** llamado *VideoJuegos* y añade el código de los ejercicios anteriores en este proyecto, separando cada clase en sus propios ficheros fuente, y siguiendo

estas reglas:

- Crea un *package* llamado `videojuegos.main` para la clase principal.
- Crea un *package* llamado `videojuegos.data` para las clases *VideoJuego*, *Compañía* y *PCVideoJuego*.

2.6.4.2. Crea un nuevo proyecto Java en **NetBeans** llamado *Barcos*, y escribe el código para representar el diagrama de clases definido en el tema anterior en el ejercicio 2.5.3.1. Intenta colocar las clases en *packages* de manera correcta.

2.6.4.3. Crea un nuevo proyecto Java en **Eclipse** llamado *VideoJuego* y escribe el código para representar el diagrama de clases definido en el tema anterior en el ejercicio 2.5.3.2. Intenta colocar las clases en *packages* de manera correcta.

2.6.4.4. Crea un nuevo proyecto Java en **NetBeans** llamado *Blog*, y escribe el código para representar el diagrama de clases definido en el tema anterior en el ejercicio 2.5.3.3. No necesitas escribir el código de los métodos (a parte de los *getters* y *setters*), únicamente un mensaje en pantalla cada vez que son llamados. Intenta colocar las clases en *packages* de manera correcta.

2.6.4.5. Crea un nuevo proyecto Java en **Eclipse** llamado *OrganizacionCultural* y escribe el código para representar el diagrama de clases definido en el tema anterior en el ejercicio 2.5.3.4. No necesitas escribir el código de los métodos (a parte de los *getters* y *setters*), únicamente un mensaje en pantalla cada vez que son llamados. Intenta colocar las clases en *packages* de manera correcta.