Introducción a la POO

- Introducción a la POO
- Objetos
- Clases
- Atributos
- Constructores
- Métodos
- Static
- Creando objetos
- Herencia
- Polimorfismo
- Sobreescritura de métodos
- Control de acceso
- Encapsulamiento
- Interfaces y clases abstractas
- Organización del código: Paquetes
- Código utilizado en los ejemplos
- Tarea

Copyright 2026

Introducción a la POO

Por fin tenemos la suficiente base como para comenzar a trabajar con Programación Orientada a Objetos (POO) mas allá de las pequeñas pinceladas que hemos introducido.

Como ya indiqué en el primer módulo la POO es un paradigma de programación. Este paradigma se supone que es una evolución de la programación estructurada con la que quizás estés acostumbrado a trabajar en otros lenguajes de programación. En este paradigma vamos a necesitar un cambio de chip ya que será el objeto quien realice las acciones.

En este tercer módulo del curso vamos familiarizarnos con los principales conceptos de la POO como son la **herencia**, el **polimorfismo** y el **encapsulamiento**.

Copyright 2026

Objetos

Una vez que hemos hablado de las clases el siguiente paso es hablar de los **objetos**. Los objetos son la clave para entender la programación orientada a objetos. Todo a nuestro alrededor puede ser considerado un objeto.

Consideramos que los objetos de nuestro alrededor tienen 2 características comunes: el **estado** y el **comportamiento**. El estado hace referencia al estado actual de una característica del objeto, pensando en un coche su color, velocidad, marcha,... Mientras que el comportamiento hace referencia a las acciones que el objeto puede llevar a cabo, continuando con el coche: acelerar, frenar, cambiar de marcha,... En ocasiones un objeto puede estar compuesto por otros objetos. Los objetos guardan su estado en campos y exponen su comportamiento a través de métodos. Los métodos son quienes operan con el estado de los objetos con lo que son los encargados de proporcionar la comunicación con el objeto. Al hacer que se acceda a los campos/atributos a través métodos garantizados que en los campos únicamente haya valores que nos interese (lo trataremos en el apartado encapsulación).

Al trabajar con objetos obtenemos una serie de beneficios como:

- Modularidad: Al escribir y mantener el código fuente de cada objeto por separado.
- Ocultación de información: Al hacer únicamente se interacciones con los métodos de un objeto conseguimos ocultar la implementación interna
- Reusabilidad: Cuando trabajemos con la herencia veremos que si tenemos objetos funcionando podemos beneficiarnos de su implementación y evitarnos codificar lo ya hecho.
- Facilidad de uso: Si un determinado objeto nos ocasiona problemas será suficiente con eliminarlo de la aplicación y programar uno que lo reemplace.

Desde un punto de vista técnico, **un objeto es una instancia de una clase**, pero ¿qué significa esto?

Cuando creamos una instancia estamos reservando una zona de memoria dedicada para el objeto en cuestión y, en Java, esa zona va a permanecer ahí mientras la variable en cuestión (u otra) la referencie. Tras leer lo anterior podemos pensar en un objeto como que un puntero a la zona de memoria que ocupa pero en POO no suele hablarse siguiendo esa terminología.

A diferencia de otros lenguajes de programación, como C, **en Java no es necesario liberar las zonas de memoria cuando dejan de utilizarse**. En Java existe un mecanismo llamado

Copyright 2026

recolector de basura que se encarga de ir liberando las zonas de memoria que no están siendo referenciadas por ningún objeto.

Con ánimo de aportar otro punto de vista y definiciones vuelvo a recurrir a la wikipedia:

> En el paradigma de programación orientada a objetos (POO, o bien OOP en inglés), un objeto es una unidad dentro de un programa de computadora que **consta de un estado y de un comportamiento**, que a su vez constan respectivamente de datos almacenados y de tareas realizables durante el tiempo de ejecución. Un objeto puede ser creado instanciando una clase, como ocurre en la programación orientada a objetos(...)
>
> **Estos objetos interactúan unos con otros**, en contraposición a la visión tradicional en la cual un programa es una colección de subrutinas (funciones o procedimientos), o simplemente una lista de instrucciones para el computador. Cada objeto es capaz de recibir mensajes, procesar datos y enviar mensajes a otros objetos de manera similar a un servicio.
>
> En el mundo de la programación orientada a objetos (POO), **un objeto es el resultado de la instanciación de una clase**. Una clase es el anteproyecto que ofrece la funcionalidad en ella definida, pero ésta queda implementada sólo al crear una instancia de la clase, en la forma de un objeto. Por ejemplo: dado un plano para construir sillas (una clase de nombre clase_silla), entonces una silla concreta, en la que podemos sentarnos, construida a partir de este plano, sería un objeto de clase_silla. Es posible crear (construir) múltiples objetos (sillas) utilizando la definición de la clase (plano) anterior. Los conceptos de clase y objetos son análogos a los de tipo de datos y variable(...)
>
> https://es.wikipedia.org/wiki/Objeto_(programaci%C3%B3n)

Clases

Lo primero que vamos a hacer va a ser ver la **definición** que nos ofrece la wikipedia sobre lo que es una clase:

> En informática, una clase es una **plantilla para la creación de objetos** de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. **Cada clase** es un modelo que **define un conjunto de variables** -el estado, **y métodos** apropiados para operar con dichos datos -el comportamiento. Cada objeto creado a partir de la clase se denomina **instancia** de la clase.
>
> Las clases son un pilar fundamental de la programación orientada a objetos. Permiten abstraer los datos y sus operaciones asociadas al modo de una caja negra. Los lenguajes de programación que soportan clases difieren sutilmente en su soporte para diversas características relacionadas con clases. La mayoría soportan diversas formas de **herencia**. Muchos lenguajes también soportan características para proporcionar **encapsulación**, como especificadores de acceso.
>
> https://es.wikipedia.org/wiki/Clase_(inform%C3%A1tica)

En la definición he querido reseñar una serie de ideas clave:

- Plantilla para la creación de objetos
- Define variables y métodos.
- A partir de la clase se crean los objetos. La clase se instancia para crear un objeto.
- Java soporta herencia y encapsulación

Ahora que tenemos una definición de lo que es una clase vamos a ver cual es la **sintaxis** de la misma.

```
NombreClase{
    sentencias del cuerpo de la clase
}
```

Copyright 2026

Cuando he definido la sintaxis de una clase he indicado que en su interior existen las *sentencias del cuerpo de la clase*, ahora voy a afinar mas. **En Java las clases**, generalmente, **están formadas por una serie de atributos, constructores y métodos**:

- **Atributos**: En caso de existir atributos (variables o constantes) estos pueden ser de tipo primitivo (int, float,...) o puede ser otro objeto (String, Coche, Integer,...).
- **Constructores**: Son métodos especiales que nos van a permitir crear la instancia de nuestra clase. Todas las clases tienen sin excepción al menos 1 constructor, aunque no esté escrito en el programa.
- **Métodos**: Es lo mismo que una función (las veíamos en el módulo 2) pero en este caso reciben este nombre cuando hablamos de objetos.

Como ya indicamos en el módulo anterior, el convenio establece que **la primera letra de una clase debe ser una letra mayúscula**. Personalmente, cuando creo una clase lo hago en el orden que he indicado en la lista anterior. Primero escribo los atributos, a continuación los constructores y después los métodos.

Hasta aquí fácil, ¿verdad? Pues bien, a esta sintaxis básica (que ya habíamos utilizado) le iremos añadiendo una serie de modificadores como public, final, extend y/o implements para ir poco a poco avanzando en nuestros programas.

En Java existe una **jerarquía de clases** en la que **todas las clases derivan de una clase llamada Object**. Si miramos la documentación de cualquier clase veremos que todas, sin excepción, derivan de ella. **Cuando una clase extiende a otra hereda todas las variables y métodos de la superclase (clase de la que hereda)**.

A continuación voy a poner una imagen de la jerarquía de clases de la clase ArrayList según obtenemos de la propia documentación:

Explicación de la documentación

Lo anterior es una captura de pantalla de la documentación de la clase ArrayList.

- En rojo aparece el paquete al que pertenece la clase
- En amarillo el nombre de la clase
- En verde la jerarquía de clases. ArrayList hereda de AbstractList que a su vez hereda de AbstractCollecion que a su vez hereda de Object. Hablaremos de la herencia mas adelante.
- En azul vemos las interfaces que implementa la clase. Hablaremos de las interfaces mas adelante.
- En morado vemos las clases que son descendiente de la clase ArrayList.

Copyright 2026

En caso de no poder distinguir los colores, la explicación sigue el orden de los recuadros. Es decir, el primer recuadro es el rojo, el segundo el amarillo,...

Para terminar con este apartado voy a enumerar los distintos **tipos de clases** existentes.

- **Públicas.**
- **Abstractas.**
- **Finales.**

Existe una cuarta opción que es la de no poner modificar a la clase.

Profundizaremos en los tipos de clases en el apartado control de acceso.

Copyright 2026

Atributos

Anteriormente hemos indicado que una clase puede contener o no atributos (también llamados campos o variables miembro) y que estos atributos podían ser de tipo primitivo o bien otras clases.

Vamos a ver un ejemplo:

Ejemplo de código con atributos

En la clase Clase existe var1 (línea 7) que es un atributo variable de tipo int (tipo primitivo). Existe VAR_2 (línea 8) que es un atributo constante de tipo int (tipo primitivo). Existe var3 (línea 10) que es un atributo variable de la clase Integer y, por último, existe VAR_4 (línea 11) que es un atributo constante de la clase Integer. Aquí he utilizado la clase Integer perteneciente al API de Java, pero si tuviese creadas varias clases podría crear variables de la clase que me interesase. Vamos a ver otro ejemplo:

Ejemplo de código con atributos y composición

En la imagen anterior vemos 2 clases. La clase Motor, que tiene 3 atributos, y la clase Coche, con otros 3 atributos. Si nos fijamos en la línea 7 de la clase Coche veremos que uno de sus atributos es de tipo Motor. En este caso se dice que la clase Coche está compuesta por la clase Motor, no confundir con la herencia (la tratamos mas adelante)

Las clases pueden contener atributos estáticos, para ello antes del tipo utilizaremos la palabra reservada static, pero de esto hablaremos mas adelante.

Copyright 2026

Constructores

Vamos a ver que **definición** de constructor encontramos en la wikipedia:

> En programación orientada a objetos (POO), un constructor es una subrutina cuya misión es inicializar un objeto de una clase. En el constructor se asignan los valores iniciales del nuevo objeto.
>
> https://es.wikipedia.org/wiki/Constructor_(inform%C3%A1tica)

Es decir, **el constructor nos va a permitir crear una instancia de una clase (un objeto)**. Un constructor es un método especial. Vamos a ver su sintaxis:

```
modificadorDeAcceso NombreDeLaClase(OpcionalmenteParámetros){
    sentencias;
}
```

Voy a ilustrarlo con un ejemplo:

Ejemplo de código con constructores

En la imagen anterior la clase Coche tiene 2 constructores. Un constructor vacío entre las líneas 9 y 11 y un constructor con todos sus parámetros entre las líneas 13 y 17. Por lo que podremos crear objetos de cualquiera de los 2 modos que vemos a continuación:

```
Coche coche = new Coche();
Coche coche2 = new Coche( "Ford", "Grand CMAX", new Motor() );
```

Cuando una clase no tiene escrito ningún constructor el compilador asume que la clase en cuestión tiene el constructor vacío. Es decir, si en el ejemplo anterior suprimimos las líneas 9 a 17 sería equivalente a dejar las líneas 9 a 11 y podríamos crear objetos únicamente así:

```
Coche coche3 = new Coche();
```

Copyright 2026

Sin embargo, si en el ejemplo anterior suprimimos las líneas 9 a 11 el único constructor que tendrá la clase será el que obliga a dar valor a todos sus atributos, es decir, sólo podríamos utilizar el siguiente modo para crear objetos:

```
Coche coche4 = new Coche( "Ford", "Fiesta", new Motor() );
```

Antes comenté que eran métodos especiales porque si te fijas la sintaxis es igual a la de los métodos con la salvedad de que no se indica el tipo de dato que devuelve y esto es así porque un constructor siempre devuelve como tipo de dato una instancia de la propia clase.

En las líneas 14 a 16 hago uso de la palabra reservada **this**. Cuando en la línea 14 escribo this.marca me estoy refiriendo a la variable marca de la clase y no al parámetro marca. Es decir, cuando escribo this.marca = marca; estoy diciendo que la variable marca de la clase pase a valer lo que valga el parámetro marca. Además de this, en un constructor podemos encontrarnos la palabra reservada **super**, pero el uso de esta lo veremos cuando hablemos de herencia.

En este curso siempre vamos a utilizar el modificador de acceso public para los constructores. Si quieres ver cuando podría tener sentido tener un constructor de acceso privado (private) puedes leer acerca del patrón de diseño Singleton.

Copyright 2026

Métodos

En el módulo 2 de este curso estuvimos hablando de las funciones. **Un método es** básicamente **lo mismo que una función pero asociado a un objeto**. Cuando en el siguiente apartado de este módulo hablemos de la palabra reservada static espero aclarar mejor esta diferenciación.

La sintaxis es básicamente la misma que en las funciones.

Vamos a ver un ejemplo:

```
1  /**
2   * @author Pablo Ruiz Soria
3   */
4  public class Coche {
5      String marca;
6      String modelo;
7      Motor motor;
8
9      public Coche(){
10
11     }
12
13     public Coche(String marca, String modelo, Motor motor){
14         this.marca = marca;
15         this.modelo = modelo;
16         this.motor = motor;
17     }
18
19     public void escribirInformacion(){
20         System.out.println("Marca: " + marca + ", modelo: " + this.modelo +
21                 ", motor: " + motor);
22     }
23 }
```

En el ejemplo anterior aparece un método entre las líneas 19 y 22. El método es de acceso público (public), no devuelve nada (void), se llama escribirInformacion y no tiene parámetros. El método tiene una única sentencia que ocupa las líneas 20 y 21 y que muestra por pantalla un texto. Fíjate que puedes acceder a los atributos de la clase con o sin this.

En Java, **dentro de una clase puedes tener 2 métodos que se llamen igual siempre y cuando el número de parámetros que utilicen o el tipo varíe**. A continuación tenemos un ejemplo:

Copyright 2026

```java
/**
 * @author Pablo Ruiz Soria
 */
public class Coche {
    String marca;
    String modelo;
    Motor motor;

    public void escribirInformacion(){
        System.out.println("Marca: " + marca + ", modelo: " + this.modelo +
                ", motor: " + motor);
    }

    public void escribirInformacion(boolean muestraMotor){
        if(muestraMotor){
            escribirInformacion();//puedes anteponer .this
        }else{
            System.out.println("Marca: " + marca + ", "
                    + "modelo: " + this.modelo);
        }
    }
}
```

En el ejemplo anterior vemos que dentro de la misma clase tenemos 2 métodos que se llaman igual (escribirInformacion. Podemos tenerlos porque en la línea 9 definimos uno de ellos sin parámetros y en la línea 14 definimos otro con un solo parámetro de tipo boolean. Podríamos tener otro mas de un solo parámetro siempre y cuando este no fuera de tipo boolean.

En Java existe la [**sobreescritura de métodos**](https://es.wikipedia.org/wiki/Herencia_(inform%C3%A1tica). Esto significa que cuando una clase hereda de otra se puede sobreescribir los métodos de esta. Lo veremos con mas detalle en el apartado de herencia.

Si eres observador/a te habrás dado cuenta de que cuando creo un método no uso la palabra reservada static como hacía con las funciones, en el siguiente capítulo voy a tratar de explicarte el porqué.

Copyright 2026

Static

Tenemos la posibilidad de crear **variables de clase** y **métodos de clase** haciendo uso de la palabra reservada *static*. Para ello haremos uso de la siguiente sintaxis para las variables de clase:

```
static tipoVariable nombreVariable;
```

Y la siguiente sintaxis para los métodos de clase:

```
static tipoDevuelvo nombreMetodo(parámetros) {
    sentencias;
}
```

En ocasiones nos puede interesar que una variable no esté asociada a un objeto sino que esté asociada a la clase, en ese momento es cuando nos interesa crear una variable de clase. Análogamente puede ocurrir con un método. Si en un momento dado nos interesa que el método esté asociado a la clase y no al objeto entonces debemos hacerlo estático, debemos hacer un método de clase.

Para acceder a una variable de clase debemos hacerlo con la sintaxis

```
NombreDeLaClase.nombreVariable;
```

Y para acceder a un método de clase debemos hacerlo con la sintaxis

```
NombreDeLaClase.nombreMetodo(parámetros);
```

Presta atención a que escribimos el nombre de la clase y no el nombre de ningún objeto.

Copyright 2026

Creando objetos

Cuando queremos **acceder a un atributo** de una determinada clase debemos hacerlo utilizando la siguiente sintáxis:

```
nombreObjeto.nombreAtributo;
```

Del mismo modo procederemos cuando a lo que queramos **acceder** sea al **método** de un objeto:

```
nombreObjeto.nombreMetodo(parámetros);
```

Lo anterior estará permitido siempre y cuando tengamos el control de acceso adecuado para el atributo o método o cuestión. Sobre el control de acceso os hablaré un poco mas adelante en este módulo, de momento vamos a considerar que no existen estas limitaciones.

Vamos a ver un ejemplo:

Copyright 2026

Perro.java

```java

/**
 * @author Pablo Ruiz Soria
 */
public class Perro {
    public String nombre;
    public int anioNacimiento;

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

    public void establecerAnioNacimiento(int anioNacimiento){
        this.anioNacimiento = anioNacimiento;
    }

}
```

Pato.java

```java

/**
 * @author Pablo Ruiz Soria
 */
public class Pato {
    public String nombre;
    public int anioNacimiento;

    public Pato(String nombre, int anioNacimiento){
        this.nombre = nombre;
        this.anioNacimiento = anioNacimiento;
    }
}
```

Cliente.java

```java


import java.util.ArrayList;

/**
 * @author Pablo Ruiz Soria
 */
public class Cliente {
    String nombre;
    String pape;//Primer apellido
    String tfnoContacto;
    ArrayList<Perro> perros; //perros que tiene el cliente
    ArrayList<Pato> patos; // patos que tiene el cliente

}
```

Lanzador.java

```java
import java.util.ArrayList;
/**
 * @author Pablo Ruiz Soria
 */
public class Lanzador {
    public static void main(String[] args) {
        Cliente vanesa = new Cliente();

        Perro lassie = new Perro();
        lassie.nombre = "Lassie";
        lassie.establecerAnioNacimiento(1954);//El atributo es privado, no es accesible

        Perro laika = new Perro();
        laika.estableceNombre("Laika");
```

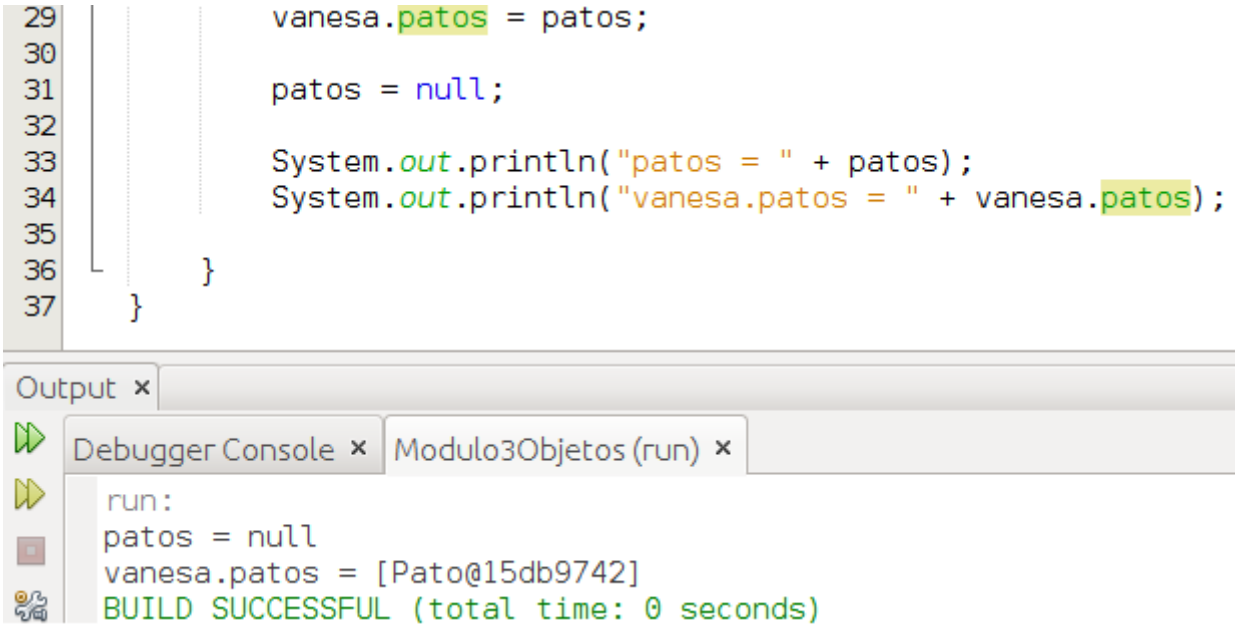
En el ejemplo anterior tenemos 4 clases Perro, Pato, Cliente y Lanzador. Vamos a centrarnos en la clase Lanzador que es la que contiene la función main y es en ella donde ocurre la acción. En las líneas 7, 9, 13, 17, 21 y 23 creamos objetos, fíjate que para ello siempre hacemos uso de la palabra reservada new. En las líneas 10, 15 y 26 a 29 accedemos a los atributos de los objetos en cuestión. En las líneas 11, 14, 18, 19 y 24 accedemos a los métodos de los objetos. Aunque en este ejemplo hayamos accedido a los atributos de este modo no es lo adecuado, generalmente se hará uso del **encapsulamiento**, pero de esto hablaremos mas adelante.

¿Y si sobre el ejemplo anterior quisiésemos acceder al nombre del primer perro de la cliente Vanesa? Pues haríamos lo siguiente:

```
String nombreDelPrimerPerroDeVanesa = vanesa.perros.get(0).nombre;
```

La sentencia anterior es un poco peligrosa y evitaremos utilizarla si antes no hacemos una serie de comprobaciones o incluimos la misma en un bloque try-catch para controlar las excepciones.

En Lanzador.java vamos a decir que ahora la variable patos valga nulo. Con ella está claro que pasará a valer nulo, pero ¿qué pasará con vanesa.patos que hemos dicho que valga lo que vale esa variable?



Lo que ocurre en la línea 29 es que al atributo patos del objeto vanesa le decimos que pase a apuntar a la misma zona de memoria que apunta la variable patos. Posteriormente decimos en la línea 31 que la variable patos que apuntaba a una determinada zona de memoria pase a no apuntar a ningún sitio (null) pero eso no afecta en que el atributo patos del objeto vanesa cambie su valor. Lo podemos ver en el resultado que aparece en consola ([Pato@15db9742] es la dirección de memoria que la JRE le asigna al objeto)

Otra duda, si creo 2 objetos cuyos atributos tengan exactamente los mismos valores, ¿qué pasará?

```
36              Perro gemelo1 = new Perro();
37              Perro gemelo2 = new Perro();
38              gemelo1.nombre = "Javier";
39              gemelo2.nombre = "Javier";
40
41              if(gemelo1 == gemelo2){
42                  System.out.println("gemelo1 y gemelo2 son iguales");
43              }else{
44                  System.out.println("gemelo1 y gemelo2 NO son iguales");
45              }
46          }
47      }
```

```
Output ×
Debugger Console × | Modulo3Objetos (run) ×
run:
gemelo1 y gemelo2 NO son iguales
BUILD SUCCESSFUL (total time: 0 seconds)
```

Lo que ocurre es que gemelo1 y gemelo2 apuntan a zonas de memoria distintas por eso el compilador nos dice que los objetos son distintos.

En Java para comparar objetos utilizaremos el método equals. Todos los objetos tienen dicho método puesto que en Java todas las clases derivan de la clase Object y la clase Object tiene este método. Ahora bien, para que equals funcione del modo esperado deberemos sobreescribir el método en nuestra Clase, lo veremos en el apartado de herencia.

Cuando tratamos de acceder a un atributo o método de un objeto y el objeto no apunta a ninguna zona (es nulo) obtendremos una excepción de tipo NullPointerException por lo que antes de acceder a un objeto deberemos realizar las comprobaciones necesarias para evitar este error.

Copyright 2026

Herencia

Una de las características de un lenguaje de programación orientado a objetos es la **herencia**. En algún capíutlo anterior a este ya hemos visto alguna funcionalidad de la herencia, pero de aquí en adelante vamos a profundizar en ella.

En POO, la herencia es un mecanismo que nos permite extender las funcionalidades de una clase ya existente. De este modo vamos a favorecer la **reutilización** de nuestro código.

La sintaxis cuando queremos heredar de una clase es la siguiente:

```
class nombreClase extends nombreClaseQueQueremosExtender {
```

Hay que indica que **solo podemos heredar de una clase a la vez**. Llamaremos **superclase** a la clase padre y **subclase** a la clase que hereda.

La herencia nos va a acerca a otros conceptos como el **polimorfismo** y la **sobrescritura de métodos** que trataremos a lo largo de este módulo. También va a traer consigo la utilización de nuevas palabras reservadas como **extends** o **super**.

¿Qué ocurre cuando una clase hereda de otra? Pues que la subclase tiene acceso (en función del control de acceso que veremos mas adelante) a los métodos y atributos de la superclase. Vamos a ver un ejemplo a continuación:

Código con ejemplo de herencia

Lo que vemos en el código anterior son 4 clases públicas (Abuelo, Padre, Hijo y Modulo3Clases. La clase Abuelo la definimos en la línea 4 de su código y **al no usar la palabra reservada extends no hereda nada mas que de la clase Object**. En la clase Abuelo definimos una variable y un método. En la línea 5 de la clase Padre indicamos que esta clase extiende a Abuelo con lo que **hereda de dicha clase y puede hacer uso de las variables y médotos** de Abuelo además de los suyos propios. La clase Hijo hereda de Padre (que a su vez heredaba de Abuelo) por lo que en la clase Hijo tendremos acceso tanto a las variables y métodos de Padre como los de Abuelo. En la clase Padre ya hacemos uso de una variable de Abuelo en su línea 8. En la clase Hijo hacemos uso de una varible de Abuelo en la línea 8 y un método de la clase Padre en la línea 9. Por último, tenemos la clase Modulo3Clases que contiene el método main. En las líneas 8 a 10 creamos los objetos de tipo Abuelo, Padre e Hijo. Estos objetos son los que nos dan acceso al código asociado a la clase. En la línea 12 y 13 vemos como desde el objeto padre tenemos acceso a métodos de la

Copyright 2026

clase Abuelo y de la propia clase Padre. Lo mismo ocurre con el objeto hijo en las líneas 18 a 20.

El ejemplo anterior símplemente pretende ilustrar la herencia de atributos y métodos pero ahora vamos a ir un poco mas allá. Vamos a pensar en perros, patos, gatos y vamos a tratar de abstraernos sobre lo que son. Todos ellos son mamíferos y tienen una determinada cantidad de patas. Probablemente todos ellos realicen las mismas acciones. Por ello voy a crear una clase llamada Mamifero con los atributos y métodos comunes y luego crearé una clase para cada animal que contenga sus peculiaridades. Vamos allá:

Código con ejemplos de herencia

En la clase Mamifero no hay nada reseñable, contiene 1 atributo, 2 constructores y 1 método. La clase Perro en la línea 4 indica que hereda de la clase Mamifero (extends Mamifero), tiene 1 atributo propio y 1 atributo que hereda, tiene 2 constructores y tiene 1 método propio. Dentro de los 2 constructores veo escrito super(4), esto lo que hace es llamar al constructor de su superclase que tiene 1 parámetro de tipo entero, es decir, hace uso del constructor de Mamifero. Dentro del método saludar vemos que existe 1 llamada a un método heredado (getPatas()). La clase Pato tiene 1 atributo propio y 1 atributo que hereda y tiene 1 contrctor y 1 método propio. En el constructor también referencia al constructor de la superclase (con super(2)) y en este caso no hace uso del método que hereda. La clase Gato es prácticamente equivalente a la clase Pato. Si nos fijamos en Lanzador vemos que tenemos la función main y en ella creamos 4 objetos.

En el ejemplo anterior hay que recordar que el hecho de que una clase no indique explícitamente que hereda de ninguna otra implica que dicha clase hereda de Object. Es decir, en el ejemplo anterior Mamifero hereda de Object. Como Perro hereda de Mamifero y Mamifero hereda de Object la clase Perro tendrá acceso a sus propios atributos y métodos y a los de Mamifero y Object.

Vamos a llevar nuestro ejemplo un poco mas allá. Vamos a pensar que quisiésemos organizar un concurso de animales en el cual pudiesen apuntarse un máximo de 100 animales siendo que solo pueden participar perros, patos y gatos ¿haríamos una lista para cada animal? Vamos a ver en el siguiente capítulo como el **polimorfismo** puede ayudarnos.

Copyright 2026

Polimorfismo

El polimorfismo es la capacidad que nos proporciona un lenguaje de programación orientado a objetos para tratar un objeto como si fuera un objeto de otra clase.

Existen lenguajes de programación donde una variable puede contener prácticamente cualquier tipo de dato, es el caso de los lenguajes PHP, Python y Javascript (recuerda, Javascript no es Java, son lenguajes distintos), mientras que existen otros lenguajes de programación en los que una variable definida de un modo solo pude contener variables de dicho tipo, es el caso de Java. Java es un lenguaje fuertemente tipado.

Debido a este fuerte tipado en ocasiones nos encontramos con que tenemos que moldear un determinado objeto de modo que quepa en el molde de otro, para ello utilizaremos el llamado **casting** o **typecasting**.

Vamos a ver un ejemplo

Ejemplo de código con polimorfismo

El ejemplo anterior no cambia en nada las clases Perro, Pato y Gato utilizadas en el ejemplo visto en el apartado Herencia. Todos los cambios ocurren en la clase Lanzador. Si nos fijamos en ella vemos que en la línea 12 creamos un ArrayList en la cual indicamos que vamos a añadir objetos de la clase Mamifero sin embargo en las líneas 13 y 14 vemos que añadimos 2 objetos de tipo Perro, en la línea 15 un objeto de tipo Pato y en la línea 16 añadimos un objeto de tipo Gato. En este caso el compilador realiza un **casting implícito** ya que todos los tipos anteriores son descendientes de Mamifero. Si en lugar de crear un ArrayList huviesemos creado un ArrayList entonces no habríamos podido insertar al pato ni al gato. Prosigamos, en la línea 18 nos encontramos con un bucle for mejorado que recorreaá todos los elementos del array y los guardará en una variable llamada animal de tipo Mamifero. Vemos que en las líneas 19, 21 y 23 usamos una palabra reservada nueva (instanceof), esta palabra lo que hace es devolver verdadero en caso de que la variable animal sea una instancia de Perro, Pato o Gato respectivamente. Cuando la condición es cierta entramos dentro del if y se realiza un **casting explícito**, es decir, le decimos que comportamiento queremos que tenga ese objeto para que así podamos acceder a sus métodos y atributos. En la línea 20 nos encontramos ((Perro) animal).saluda(); aquí lo que ocurre es que se realiza el cast sobre animal para que se comporte como si fuera de tipo Perro y una vez que se ha realizado el cast y ya no es de tipo animal sino que es de tipo Perro, entonces es cuando tenemos acceso al método saludar y no antes. Ocurre lo mismo en las líneas 22 y 24.

Copyright 2026

En nuestro ejemplo de herencia y polimorfismo hemos creado una clase (Mamifero) que queríamos sirviese de base para otras (Perro, Pato y Gato) pero que no hemos llegado a utilizar. En el capítulo Clases abstractas e interfaces veremos cual habría sido la forma mas adecuada de diseñar esta jerarquía de clases.

Pregunta Verdadero-Falso

Java y Javascript son el mismo lenguaje de programación. Nos referimos a Javascript como Java para tener un lenguaje mas fluido.

VerdaderoFalsoJava y Javascript son lenguajes de programación distintos. Su sintaxis es parecida al igual que ocurre entre C# y Java o entre C++ y Java.

En el ejemplo anterior, sería correcto escribir Gato gato = (Gato) nala;

VerdaderoFalsoNo podría hacerse lo que se indica en la pregunta ya que nala es un objeto de tipo Perro que hereda de Mamifero y Gato hereda de Mamifero. En la jerarquía de clases derivan de la clase Mamifero pero a partir de ese momento son ramas distintas de la jerarquía. Si podría hacer Mamifero mamifero = (Mamifero) nala;

Copyright 2026

Sobreescritura de métodos

Otra característica asociada a la herencia es la **sobreescritura de métodos**. La wikipedia se refiere a ella como [redefinición de métodos](https://es.wikipedia.org/wiki/Herencia_(inform%C3%A1tica) pero yo nunca la he visto denominada de ese modo con anterioridad.

Cuando dentro del apartado Clases estuvimos hablando de los métodos nombramos por primera vez la sobreescritura de métodos. No hay que confundir la sobreescritura de métodos con que un mismo método pueda ser definido de modos distintos.

La sobreescritura de métodos nos permite redifinir un método que heredamos para que este funcione de acuerdo a nuestras necesidades y no a lo definido en la superclase. Cuando en un objeto llamamos a un método el compilador comprueba si el método existe en nuestro objeto, si existe lo usa y si no existe en nuestro objeto entonces lo busca en la superclase. Esto ocurre así hasta que el compilador encuentra el método definido. El compilador busca el método de abajo a arriba.

Vamos a ver un ejemplo

Copyright 2026

```java
/**
 * @author Pablo Ruiz Soria
 */
public class Persona {
    int anioNac;
    String nombre;
    String pape; //1er apellido

    public Persona(int anioNac, String nombre, String pape){
        this.anioNac = anioNac;
        this.nombre = nombre;
        this.pape = pape;
    }

    @Override
    public String toString(){
        return "Me llamo " + nombre + " " + pape
                + " y nací en " + anioNac + ".";
    }
}
```

```java
/**
 * @author Pablo Ruiz Soria
 */
public class Animal {
    String nombre;
    int patas;

    public Animal(String nombre, int patas){
        this.nombre = nombre;
        this.patas = patas;
    }
}
```

```java
/**
 * @author Pablo Ruiz Soria
 */
public class NewMain {
    public static void main(String[] args) {
        Persona aldara = new Persona(2016, "Aldara", "Ruiz");
        Persona pablo = new Persona(1983, "Pablo", "Ruiz");
        Animal nala = new Animal("Nala", 4);
        Animal donald = new Animal("Donald", 2);

        System.out.println(aldara.toString());
        System.out.println(donald.toString());
        System.out.println(nala.toString());
        System.out.println(pablo.toString());
    }
}
```

Output ×

Debugger Console × | Modulo3SobreescrituraDeMetodos (run) ×

```
run:
Me llamo Aldara Ruiz y nací en 2016.
Animal@15db9742
Animal@6d06d69c
Me llamo Pablo Ruiz y nací en 1983.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Copyright 2026

Por simplicidad he decidido escribir 2 clases (Persona y Animal) que al no usar la palabra reservada extended derivan implicitamente de la clase Object. Si miramos la documentación de Object veremos que tiene un método llamado toString. Este método lo que hace es indicar la dirección de memoria en que el compilador ha guardado el objeto. En la clase Persona en las líneas 15 a 19 he redefinido el método toString dándole un comportamiento distinto mientras que en la clase Animal no lo he redefinido. Si volvemos a la clase Persona vemos que en la línea 15 aparece una anotación que le indica que estamos sobreescribiendo el método, esta anotación no es necesaria pero en teoría agiliza la compilación y ejecución de nuestros programas. Si vamos a la función main vemos que en las líneas 6 a 9 creamos los objetos y en las líneas 11 a 14 llamamos al método toString de dichos objetos. En el caso de los objetos de tipo persona se ejecuta el método que hemos redefinido mientras que en los objetos de tipo animal se utiliza el método de la superclase.

Cuando redefinimos un métodos podemos hacer uso del propio método que estamos redefiniendo, para ello haremos uso de la palabra reservada super. Voy a modificar el código del ejemplo anterior para que podamos verlo:

```
16      public String toString(){
17          return "Me llamo " + nombre + " " + pape
                    + " y nací en " + anioNac +
                    ". El compilador me ubica en " + super.toString();
19      }
```

```
Debugger Console × | Modulo3SobreescrituraDeMetodos (run) ×
run:
Me llamo Aldara Ruiz y nací en 2016. El compilador me ubica en Persona@15db9742
Animal@6d06d69c
Animal@7852e922
Me llamo Pablo Ruiz y nací en 1983. El compilador me ubica en Persona@4e25154f
BUILD SUCCESSFUL (total time: 0 seconds)
```

En el extracto anterior vemos que he añadido super.toString() para decirle a mi método toString que llame al método toString de la superclase. Debajo del código aparece el resultado que se produce al ejecutar el programa tras el cambio.

Control de acceso

Hasta el momento hemos visto que desde cualquier clase se puede acceder a cualquier otra clase y a cualquier atributo o método sin ningún control. Esto lo hemos hecho de este modo para facilitar la comprensión de los distintos conceptos tratados a lo largo del curso y de este modo no distraernos con cuestiones de control de acceso pero a partir de este momento vamos a hablar del **control de acceso** en clases, atributos, constructores y métodos para así empezar a desarrollar nuestros programas de un modo mas adecuado.

a clases

Cuando definimos una clase podemos hacerlo con la palabra reservada public, abstract, final o ninguna de los anteriores. Existe alguna otra posibilidad pero no entraremos en ella.

|Clase|Descripción| |:--|:--| |public|Una clase pública es accesible desde cualquier clase. Para ser utilizada desde otro paquete debe ser importada.| |abstract|Una clase abstracta es una clase destinada a que se herede de ella. No puede ser instanciada. Debe tener al menos un método abstracto.| |final|Una clase final es aquella que no permite que se herede de ella.|

Así podemos combinar las distintas palabras clave:

```
public class Ejemplo1{

class Ejemplo2{

public abstract class Ejemplo3{

abstract class Ejemplo4{

public final class Ejemplo5{

final class Ejemplo6{
```

Copyright 2026

En las 6 líneas anteriores tenemos todas las combinaciones con las que vamos a trabajar en el curso.

La clase Ejemplo1 es una clase pública por lo que será accesible desde cualquier paquete y se podrán crear instancias de ella y usarla como superclase.

La clase Ejemplo2 será accesible únicamente desde el paquete en que sea definida y se podrán crear instancias de ella y usarla como superclase.

La clase Ejemplo3 es una clase pública por lo que será accesible desde cualquier paquete y al ser abstracta no se podrán crear instancias de ella, deberá contener al menos un método abstracto y se podrá usar como superclase.

La clase Ejemplo4 será accesible únicamente desde el paquete en que sea definida y al ser abstracta no se podrán crear instancias de ella, deberá contener al menos un método abstracto y se podrá usar como superclase.

La clase Ejemplo5 es una clase pública por lo que será accesible desde cualquier paquete pudiendo ser instanciada pero no usada como superclase.

La clase ejemplo6 será accesible únicamente desde el paquete en que sea definida pudiendo ser instanciada pero no usada como superclase.

a atributos

Existen las siguientes palabras clave cuando queremos controla el acceso a las variables miembro de nuestras clases. Vamos a verlas:

|Atributo|Descripción| |:--|:--| |public|El campo es accesible desde todas las clases| |private|El campo es accesible únicamente desde la propia clase| |final|El campo no puede ser modificado y al definirse debe establecerse su valor.|

Vamos a ver algún ejemplo:

public String ejemplo1;

String ejemplo2;

private String ejemplo3;

public final String ejemplo4 = "trivinet.com";

final String ejemplo5 = "recurso didáctico gratuito";

private final String ejemplo6 = "de alta calidad";

Las 6 combinaciones anteriores serían todas las posibles.

ejemplo1 sería accesible y modificable desde cualquier clase.

ejemplo2 sería accesible únicamente desde las clases que pertenezcan al mismo paquete y cualquiera de ellas podría modificarla.

ejemplo3 sería accesible únicamente desde la clase en que se ha definido y podría ser modificada.

ejemplo4 sería accesible desde cualquier clase. No se podría modificar su valor.

ejemplo5 sería accesible únicamente desde las clases que pertenezcan al mismo paquete. No se podría modificar su valor.

ejemplo6 sería accesible únicamente desde la clase en que se ha definido. No se podría modificar su valor.

Es común que los campos se definan como privados para así sacar todo el potencial de la **encapsulación** (la veremos en el siguiente capítulo).

a constructores

|Constructor|Descripción| |:--|:--| |public|Cualquier clase puede crear instancias de la clase que contenga un constructor público.| |protected|Solamente las subclases de la clase que contiene un constructor protegido pueden crear instancias de la clase.| |private|Ninguna clase puede crear instancias de una clase que tiene un constructor privado excepto la propia clase a través de alguno de sus métodos públicos.|

El patrón de diseño singleton es un ejemplo a través del cual ver la utilidad de definir un constructor privado.

a métodos

A la hora de establecer el control de acceso a un método en el curso vamos a trabajar con public, private, protected, abstract y final.

|Método|Descripción| |:--|:--| |public|El método es accesible desde la propia clase, el paquete al que pertenezca la clase, las subclases y el resto del mundo.| |protected|El método es accesible desde la propia clase, el paquete al que pertenezca la clase y las subclases.| |private|El método es accesible desde la propia clase.| |abstract|El método no está definido en esta clase sino que será definido en la clase que herede de la clase que contenga este método.| |final|El método no puede ser sobreescrito.|

Cuando un método es abstracto no se pone el bloque de código que va entre las llaves ({}) ni las propias llaves, esto tiene sentido porque el método en cuestión debe definirse en la clase que herede la clase que contenga el método abstracto.

Vamos a ver algunas combinaciones:

public void ejemplo1(){

protected void ejemplo2(){

void ejemplo3(){

private void ejemplo4(){

public abstract void ejemplo5();

protected abstract void ejemplo6();

abstract void ejemplo7();

public final void ejemplo8(){

protected final void ejemplo9(){

final void ejemplo10(){

private final void ejemplo11(){

En los 11 ejemplos anteriores no hay valor de retorno (void) y como puede apreciarse no existe la combinación private abstract ya que no tendría sentido. Existe alguna otra palabra reservada que da lugar a mas combinaciones pero estas son suficientes para el objetivo del curso.

Copyright 2026

En la documentación oficial de Java sobre el control de acceso podemos encontrar mas información (en inglés).

Encapsulamiento

La encapsulación es un principio fundamental de la programación orientada a objetos y consiste en ocultar el estado interno del objeto y obligar a que toda interacción se realice a través de los métodos del objeto.

En el código que hemos desarrollado hasta el momento hemos dado libre acceso a los atributos de nuestros objetos lo cual presenta riesgos ya que no tenemos ningún tipo de control sobre lo que puede o no hacerse con ellos. Nos va a interesar que a nuestros atributos solo se pueda acceder a través de los métodos, de modo que obtengamos control sobre lo que puede hacerse con los atributos.

Para ello, el acceso a los atributos se establece como privado y se crean 2 métodos por cada atributo, un **getter** y un **setter**. El getter de un atributo se llamará getNombreAtributo mientras que el setter de un atributo se llamará setNombreAtributo. Vamos a ver un ejemplo para entender esto mejor:

Código con ejemplo de encapsulamiento

Si nos fijamos en Mesa.java veremos que es una clase con 1 atributo privado llamado color y 2 métodos que se llaman getColor y setColor. Los métodos anteriores serían, respectivamente, el getter y setter de del atributo color. Ahora que ya conocemos el control de acceso que sobre clases, atributos, constructores y métodos puede realizarse sabemos que no puede accederse directamente al atributo ni para obtener su valor ni para establecer su valor, siempre que quieran realizarse esas acciones habrá que pasar respectivamente por el getter y el setter y en ellos podremos programar aquello que nos interese. Si nos fijamos en Principal.java veremos que si ahora queremos acceder directamente al atributo para modificar u obtener su valor no podemos (he dejado comentado el código que falla) y que para hacerlo estamos obligados a utilizar los getter y setters que hemos creado antes.

Si bien es cierto que podríamos llamar a los getters y setters de cualquier otro modo **por convenio se utiliza la sintaxis que hemos visto con anterioridad**. Hacer uso de este convenio nos facilitará trabajar con el resto del mundo y nos permitirá ampliar las capacidades de nuestro código utilizando frameworks existentes que hacen uso del convenio y que si no seguimos no podremos utilizar.

¿Aún no les ves sentido?

Voy a modificar ligeramente el código de Mesa.java para tratar de crear un ejemplo mejor:

Ejemplo de código con encapsulamiento desarrollado

A la clase Mesa le he añadido un nuevo atributo llamado numeroDeVecesPintada en el cual voy a almacenar el número de veces que he cambiado el color de la mesa. Además, ahora la mesa quiero que solo pueda pintarse de 3 colores distintos. El número de veces que se pintado la mesa no quiero que pueda ser cambiado en cualquier momento, solo quiero que se autoincremente en 1 cuando realmente se pinte la mesa. Por ello en el setter de color hago que solo cambie el valor del atributo cuando recibo un color válido y si eso ocurre además aumento en 1 las veces que he pintado la mesa. Si en el setter de color (setColor) no recibo un valor válido entonces no cambio el atributo color de la mesa ni incremento en 1 las veces que se ha pintado la mesa. Como además no quiero que nadie pueda modificar el valor de las veces que se pinta la mesa no hago un setter para este atributo.

Copyright 2026

Interfaces y clases abstractas

Ya hemos hablado con anterioridad de las **clases abstractas** (capítulo polimorfismo), en este capítulo profundizaremos un poco mas sobre ellas y además hablaremos por primera vez de las **interfaces**.

Interfaces

Cuando alguien te pregunta que es una interface suele decirse que es un contrato, una forma de asegurarse que todo el mundo que implemente una interface va a hacer algo. No nos importa como vayan a hacerlo pero estamos seguros que si una clase implemente una interface entonces va a tener que cumplir el contrato e implementar lo acordado.

Vamos a ver la sintaxis

```
public interface NombreInterface{
    sentencias;
}
```

Y cuando una clase implementa una interface la sintaxis es

```
class NombreClase implements NombreInterface{
    sentencias;
}
```

Una clase puede implementar 0 o varias interfaces al mismo tiempo. En caso de implementar varias interfaces separaremos sus nombres por comas. El hecho de poder implementar varias interfaces al mismo tiempo nos sirve para esquivar la limitación de que una clase solo pueda heredar de una única clase. En caso de que una clase extendise a otra clase y a la vez implementase 1 o varias interfaces la sintaxis sería la siguiente:

```
class NombreClase extends SuperClase implements NombreInterface1, nombreInterface2,...{
    sentencias;
}
```

Las interfaces pueden ser públicas (accesibles desde cualquier lugar) o sin modificador de acceso (accesibles desde el paquete de la interface).

Una interface no puede ser instanciada (si se podría pero es demasiado avanzado para el curso).

Una interface puede contener métodos abstractos y métodos estáticos, ambos se considerarán públicos (aunque no se indique). También pueden añadir métodos por defecto (con el modificador default) pero no los vamos a ver en este curso.

Una interface puede contener atributos que a todos los efectos será una constante, estaremos obligados a darle valor. Los atributos que creemos se considerarán públicos, estáticos y finales por lo que podemos omitir los modificadores.

Una interface puede extender otras inferfaces, lo hará con la siguiente sintaxis

```
public interface NombreInterface extends NombreOtraInterface1, NombreOtraInterface2,...{
    sentencias;
}
```

Si, es correcto extends, no me he equivocado.

Clases abstractas

La sintaxis necesaria para crear una clase abstracta es la siguiente

```
abstract class NombreClase{
    sentencias;
}
```

Las clases abstractas pueden incluir o no **métodos abstractos**. Las clases abstractas no pueden ser instanciadas pero si pueden ser usadas como subclases.

Un método abstracto es un método que está declarado pero no está implementado. En su sintaxis se suprimen las llaves ({}) y tras cerrar el paántesis que contiene los parámentros se pone un

punto y coma.

```
public abstract void montar();
```

Si una clase contiene un método abstracto tiene que ser obligatoriamente una clase abstracta.

Cuando una clase hereda de una clase abstracta y la superclase contiene un método abstracto estamos obligados a implementarlo.

Diferencias entre clases abstractas e interfaces

Las clases abstractas y las interfaces son similares. No puedes instanciarlas y pueden contener métodos implementados o no.

En las clases abstractas puedes disponer de atributos sin que estos sean constantes (como ocurre en las interfaces) y además puedes crear métodos públicos, protegidos o privados (en las interfaces todos eran públicos).

Solo puedes extender una clase (abstracta o no) mientas que puedes implementar cualquier número de interfaces.

Entonces, ¿cuándo elijo una u otra?

- Clases abstractas cuando se cunpla alguna de las condiciones siguientes:
  - Quieres compartir código entre muchas clases relacionadas
  - Esperas que las clases que extiendan tu clase abstracta tengan en común métodos o campos
  - Quieres disponer de atributos no estáticos o no finales. Esto te habilita para definir métodos que puedan acceder y modificar el estado del objeto al que pertenecen
- Interfaces cuando se cumpla alguna de las condiciones posteriores:
  - Esperas que clases sin relación entre si implementen tu interface.
  - Quieres un comportamiento específico sin importante la implementación
  - Quieres disponer de herencia múltiple.

Si deseas ampliar la información, en la documentación oficial de Java existe un tutorial específico sobre interfaces y herencia (en inglés).

En el apartado "código utilizado en los ejemplos" dejo un proyecto en el cual hago uso de interfaces y clases abstractas.

Copyright 2026

Organización del código: Paquetes

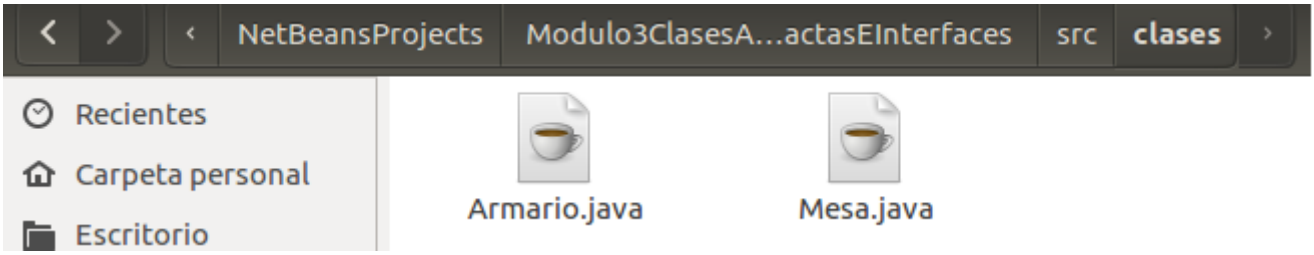
Un paquete es una agrupación lógica de clases o interfaces relacionadas. La creación de paquetes nos permite dotar a nuestro código de mayor protección a través del control de acceso y además nos permite facilitar la organización de nuestros programas. Si pensamos en los ficheros con los que trabajamos a diario no los dejamos todos sueltos en el escritorio sino que creamos una serie de carpetas/directorios donde los agrupamos de modo lógico con el fin de facilitar nuestra organización.

El convenio dice que los paquetes deben nombrarse en minúsculas y cuando se necesitan usar varias palabras separarlas por un guión bajo. Además, suele utilizarse el dominio de nuestra empresa invertido, de este modo, podemos diferenciar clases que compartan nombre.

Para indicar que una clase pertenece a un determinado paquete debemos indicarlo usando package nombreDelPaquete y ubicando la clase dentro de dicha ruta de carpetas. Vamos a ver un ejemplo:

```
1  package clases;
2
3  import clases_abstractas.Mueble;
4  import interfaces.Transporte;
5
6  /**
7   * @author Pablo Ruiz Soria
8   */
9  public class Armario extends Mueble implements Transporte{
```

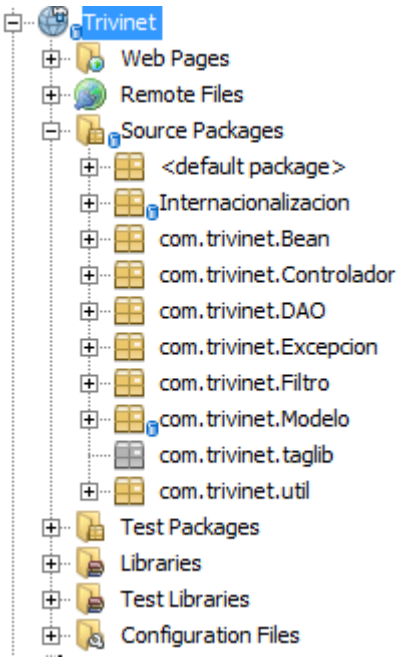
Por lo que la clase Armario deberá estar ubicada en la carpeta clases



Cuando en nuestros programas queremos usar una clase que no pertenece a nuestro paquete (y tenemos permiso para ello) existen 2 posibilidades:

- Importar la clase, tal como vemos en las líneas 3 y 4 de la imagen anterior
- Referenciarla completamente cada vez que vayamos a usarla. Eliminaríamos las líneas 3 y 4 y cada vez que quisiesemos hacer referencia a Mueble escribiríamos clases_abstractas.Mueble y cada vez que quisiésemos hacer referencia a Transporte escribiríamos interfaces.Transporte

A continuación vamos a ver la organización en paquetes de un proyecto real, el proyecto a partir del cual se desarrolla el recurso didáctico trivinet.com



El proyecto de la imagen está formado por 80 clases agrupadas en 9 paquetes distintos en función de las funcionalidades que cada clase tiene.

Java nos proporciona una cantidad de paquetes enormes lista para utilizar en nuestros paquetes, esta librería recibe el nombre de API (Application Programming Interface) La especificación completa del API de Java 8 para su edición estándar está accesible aquí.

Código utilizado en los ejemplos

Módulo 3 Clases (zip - 0.01 MB).

Módulo 3 Atributos (zip - 0.01 MB).

Módulo 3 Objetos (zip - 0.01 MB).

Módulo 3 Herencia (zip - 0.01 MB).

Módulo 3 Polimorfismo (zip - 0.02 MB).

Módulo 3 Sobreescritura de métodos (zip - 0.01 MB).

Módulo 3 Control de acceso (zip - 0.01 MB).

Módulo 3 Encapsulamiento (zip - 0.01 MB).

Módulo 3 Interfaces y clases abstractas (zip - 0.01 MB).

Copyright 2026

Tarea

Tu tarea una vez acabado el tercer módulo consiste en:

- Crear un proyecto llamado Modulo3NombreApellido donde Nombre sea tu nombre y Apellido tu primer apellido. Ejemplo: Modulo3PabloRuiz
- En el proyecto deberá estar organizado por paquetes de modo que tengas, al menos, los paquetes clase, clase_abstracta, interf y lanzador.
- Dentro del paquete lanzador deberás crear una clase llamada Principal. En esta clase estará el método main.
- Vamos a modelizar un tienda de bicicletas.
  - Tendrás una clase abstracta llamada Bicicleta
  - Tendrás unas subclases de Bicicleta llamadas BiciMontania, BiciPaseo, Tandem. Estas subclases deberán sobreescribir al método toString de Object haciendo que digan de que tipo son, su color y precio.
  - Todas las bicicletas tendrán dos campos comunes color (String) y precio (double).
  - Quiero que todas las bicicletas implementen el método pintar, que lo que hará será cambiar el color pero me interesa que este método pintar también se pueda aplicar a otras posibles modelizaciones futuras que pudiera hacer, no debe ser exclusivo de las bicicletas. Además quiero que el coste de pintar (lo que sea) sea fijo y sea 90.
  - Las bicicletas de montaña deberán almacenar la marcha que tiene la bici (pudiendo tomar valores únicamente entre 1 y 6).
  - Las bicicletas de paseo deberán almacenar la velocidad a la que se circula.
  - Los tandem deberán almacenar el número de asientos (pudiendo tomar únicamente los valores 2 y 3).
  - Todos los campos de todas las clases deberán seguir los principios de encapsulamiento.
  - La clase BiciMontania deberá tener un único constructor a través del cual se establezca el valor del color, precio y marcha.
  - La clase BiciPaseo deberá tener 2 constructores, uno a través del cual se establezca el valor del color, precio y velocidad y otro a través del cual se establezca el color y precio.
  - La clase Tandem deberá tener un único constructor a través del cual se establezca el valor del color, precio y número de asientos.
  - Los constructores deberán controlar las condiciones existentes.
  - La BiciMontania tendrá un método llamado aumentarMarcha que no tendrá parámetros y aumentará en 1 la marcha actual (siempre que cumpla las condiciones anteriores). También tendrá un método llamado disminuirMarcha que no tendrá parámetros y reducirá en 1 la marcha actual (siempre que cumpla las condiciones

anteriores). No se podrá modificar la marcha desde ningún otro lugar.

- En la clase Principal deberás crear una lista con todas las bicicletas que tengas en la tienda. En este momento tienes 2 bicicletas de montaña, 1 bici de paseo y 2 tandems. Crea los objetos y añádelos a la lista. Posteriormente interacciona con ellos y al final recorre la lista para llamar al método toString de cada objeto.

Copyright 2026