3. Clases

Date	Empty
Status	Not started
Type	Empty
Units	5. Orientación a objetos

Clases

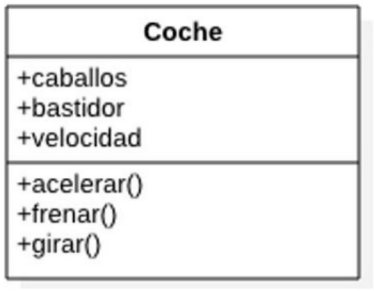
- Atributos - propiedades
- Métodos o funciones
- Constructores
- Objetos

Relación entre clases

Clases

Hasta ahora se han resuelto ejercicios en los que se han almacenado datos en variables de tipos básicos (o complejos si hablamos de String), y se han modificado dichos datos con métodos. Todas estas variables han sido llamadas desde una misma clase donde encontrábamos el método main. Por ejemplo, para leer un contacto se almacenaba en una variable de tipo cadena el nombre y en una de tipo entero, el teléfono. También necesitábamos métodos para leer

nombres y teléfonos válidos, modificarlos y mostrarlos por consola. Si se quiere trabajar de esta forma se podrían solucionar muchos problemas, pero en el 100% de los casos nos encontraríamos con muchísimo código para hacer tareas demasiado simples. Para ello se utiliza el concepto de clase, que lo definimos como el tipo que un programador crea para poder utilizarlo en diferentes partes del programa. Estas clases son creadas en archivos separados y utilizadas en cualquier parte (bien sea otras clases que representen otros tipos o directamente en la clase que tiene el método main). Por ejemplo, una clase sería un coche donde este tiene características (variables que cualifican) como color, marca, modelo ó cv y funcionalidades (métodos que dan funcionalidades) como acelerar, girar, frenar.



Para poder crear una clase se puede hacer directamente desde el la carpeta src, pero para mantener el código organizado (y así poder utilizar los modificadores de acceso que se explicarán) se utilizan los paquetes. Estos se crean organizando clases por funcionalidades por ejemplo. Una vez se ha creado un paquete, al inicio de todas las clases que pertenezcan al mismo aparecerá la palabra package

```
package introduccion; public class Coche { String bastidor; i
nt caballos, velocidad; public void acelerar(int v){ this.vel
ocidad = this.velocidad + v; } public void deceledad(int v){
this.velocidad = this.velocidad - v; } public void reprograma
r(int cv) { this.caballos = cv; } }
```

Una vez se crea un paquete, se puede crear una clase dentro. A la hora de declarar una clase se tienen los siguiente elementos:

```
Package nombre.paquete public class NombreClase{ // variables
de clase: existirán en todo el cuerpo de la clase public nomb
reVariable; private otraVariable; protected otraVariableMas
// métodos: funcionalidades que tendrá un objeto de la clase
public void miMetodoUno(){ // variable de método } private vo
```

```
id miMetodoDos(){ // variable de método } protected void miMe
todoTres(){ // variable de método } }
```

Los modificadores de acceso de los elementos de la clase son:

- private → Sólo se puede acceder a ese miembro en clase donde se define. Es decir, el atributo “otraVariable” sólo puede ser referenciado en la clase donde ha sido creado
- protected → Sólo se puede acceder a ese miembro en la clase donde se define y en las clases que deriven de dicha clase (es decir aquellas que hayan extendido de la misma) o estén en el mismo paquete.
- public → Se puede acceder a ese miembro desde cualquier clase. Por ejemplo, los métodos “metodoUno()” y la variable “nombreVariable” puede ser accedidos desde cualquier clase una vez que se haya creado un objeto de la clase.
- package → En caso de no poner nada sólo se puede acceder a ese miembro en la clase donde se define, y en las clases de su mismo paquete

Hay que tener en cuenta que las variables casi siempre debería ser *privadas* de forma que no puedan ser accedidas de forma directa, tan solo mediante métodos que sean públicos (son los llamados getters y setters). Las clases por defecto se ponen públicas para que puedan ser utilizadas desde cualquier parte creando objetos de su correspondiente tipo

Cuidado con poner un nombre de la clase diferente al nombre del archivo, ya que en ese caso estaremos haciendo dos clases dentro de un mismo fichero. Obligatoriamente una de las mismas tendría que tener el nombre del fichero

Consideraciones a la hora de crear las clases:

- Los nombres de las mismas siempre con la primera letra en mayúsculas
- Los nombres deben ser descriptivos
- Los nombres deben ir en minúsculas

Los elementos que tienen una clase son:

Atributos - propiedades

Aquellos elementos que cualifican un objeto de la clase y le dan

Aquellos elementos que califican un objeto de la clase y le dan propiedades o cualidades que lo hacen diferente. Como ya se ha visto a lo largo del curso las variables pueden ser

- Según su ámbito
  - atributo de clase
  - atributo de método
- Según su composición
  - atributo primitivo
  - atributo completo

```
public class Coche { // atributos de clase que son accesibles desde cualquier parte de la clase private String bastidor; private int caballos, velocidad; public void acelerar(int v){ // v representa un atributo de método ya que solo es accesible desde el bloque desde el cual ha sido definida this.velocidad = this.velocidad + v; }
```

Por regla general las variables se declaran como privadas para así respetar el principio de encapsulación, de forma que nadie pueda acceder a ellas de forma directa, solo a través de métodos públicos como se verá en el siguiente punto.

Métodos o funciones

Aquellos elementos que dan una funcionalidad a la clase. Se pueden crear tantos métodos como sean necesarios para cumplir con las funcionalidades necesarias que necesite el programa. La estructura de un método es:

```
[modificador_acceso]tipo_retorno nombre([argumentos]) [throws excepciones]{ cuerpo }
```

Aquellos campos que están englobados entre [] son optativos, por lo que puede no estar presentes. En cuanto a los modificadores, son los mismos que se han comentado anteriormente: *public*, *protected* (por defecto), *private*, y se añaden los siguientes:

- static, el cual indica que es un método accesible de forma directa
- abstract, el cual indica que el método no puede tener cuerpo o

definición, tan solo firma. Se obliga a que la subclase defina el cuerpo. Su uso se verá el el siguiente tema con la herencia

- final, el cual indica que el método no puede ser sobrescrito por una subclase. Su uso se verá el el siguiente tema con la herencia
- synchronized, el cual indica que el método solo puede ser utilizado por un hilo a la vez.

```
public static void generarCoche(){ System.out.println("el coc
he ha sido generado de forma automática"); } final void impri
mirDatos(){ System.out.printf("Los datos del coche son %n" +
"Marca: %s %n" + "Modelo: %s %n" ); } abstract void calcularA
celeracion(); synchronized void sacarCoche(Coche coche){ Syst
em.out.println("El coche cuyos datos son "); coche.imprimirDa
tos(); System.out.println("Ha sido sacado del garaje "); }
```

Además de todos los métodos que se necesiten crear, existen una serie de métodos que son muy recomendables utilizar:

Los métodos setter son aquellos que modifican el valor de la variable. Para la definición de estos métodos se pide como argumento en el método el valor que se quiere asignar a la variable en cuestión

Los métodos getter son aquellos que obtienen el valor de la variable. Para la definición de estos métodos no se piden argumentos en el método y se retorna el valor que tiene asignado la variable en cuestión.

El método toString devuelve el valor String del objeto que llame al método. Para la definición de este métodono se piden argumentos y se retorna el valor String que se quiera configurar.

El tipo de retorno puede se cualquier tipo, incluso uno creado por nosotros mismos (representado por el nombre de la clase). Si la firma de un método tiene un valor de retorno diferente a void (no retorna nada), es obligatorio que la última palabra del método sea la reservada **return**, acompañada del valor que devolverá la llamada al método

```
final String imprimirDatos(){ String datos = String.format("L
os datos del coche son %n" + "Marca: %s %n" + "Modelo: %s %n"
); return datos; }
```

Los argumentos son todos aquellos datos que un método necesita

Los argumentos son todos aquellos datos que un método necesita para su funcionamiento. No es obligatorio su uso, pero si muy recomendable ya que cuando el método sea llamado es posible que se le pasen argumentos diferentes.

```
public void acelerar (int v){ this.velocidad = this.velocidad
+v; } // cuando un objeto de coche sea creado podrá llamar a
este método, pasando argumentos diferentes.
```

```
public static void main(String[]args){ Coche coche = new
Coche(); coche.acelerar(50); coche.acelerar(100); }
```

Los argumentos representan datos de entrada a los métodos que deberán de ser introducidos cuando un objeto instanciado llame al método. Lo que ocurre en esa situación es que se sustituirá el valor pasado por parámetro por el valor por defecto del método. Por defecto los métodos se crean sin parámetros siempre que no necesiten ejecuciones dinámicas, es decir que su ejecución siempre sea la misma. Sin embargo si se pasan parámetros a un método, la ejecución de las instrucciones internas va a depender en gran medida de los parámetros

```
public void realizarSuma(){ int operadorUno=5, operadorDos=8,
suma; suma = operadorUno + operadorDos; } public void realiza
rSuma(int op1, int op2){ int suma; suma = op1+op2; } public s
tatic void main(String[]args){ realizarSuma(); realizarSuma
(8,9); }
```

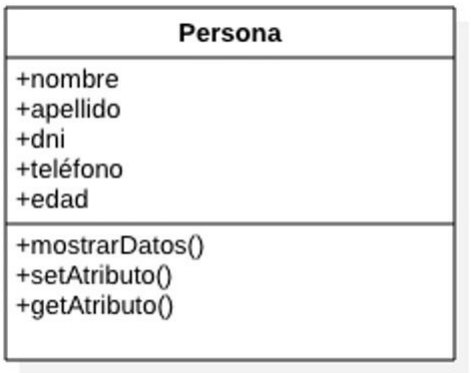
En los ejemplos indicados, se crea un método que tiene dos ejecuciones posibles. La primera es sin parámetros de entrada, donde se realizará una suma de dos números indicados en el propio método (5 y 8). La segunda realiza la suma de dos números indicados en el momento de llamar al método (8 y 9) que serán sustituidos por la variables op1 y op2 respectivamente. Como se puede ver los dos métodos se llaman igual pero tienen diferentes parámetros (bien en tipos o número) por lo que no daría error. Esta técnica recibe el nombre de sobrecarga de métodos.

Cosas a tener en cuenta a la hora de trabajar con métodos:

Cosas a tener en cuenta a la hora de trabajar con métodos:

- Los nombres deben ser descriptivos.
- El nombre debe empezar con minúsculas.
- En el caso de necesitar más de un argumento se separan por ,
- Los métodos se definen fuera de otros métodos, nunca dentro.
- Pueden existir dos métodos con el mismo nombre pero con diferentes argumentos (este concepto se verá mejor en el siguiente tema).
- Los métodos pueden estar acompañados del decorador @Override el cual indica que el método se ha sobrescrito (este concepto se verá mejor en el siguiente tema).

Ejercicio Crea una clase con la siguiente definición UML



Por defecto todos los métodos sin tratamiento tienen un retorno de void, es decir que no retornan nada, tan solo realizan las ejecuciones que tienen marcadas dentro del método.

```
public class Coche { private String marca, modelo; private
int cv, cc; private int velocidad; public void acelerar(int
cantidad){ this.velocidad += cantidad; } }
```

En el método acelerar, no se tienen ningún retorno, ya que se marca como void por lo que tan solo realizará las lineas de código que están escritas dentro del método. Sin embargo, en algunas ocasiones interesa que los métodos además de hacer lo que tienen codificado en su interior, también den un resultado concreto, lo que se conoce como retorno. Este retorno se indica poniendo el tipo de dato que va a retornar el método en vez de void, y como última linea dentro del método pondrá la palabra reservada return acompañado del dato que

retornará

```
public double calcularPar(){ double par = this.velocidad/2
*this.cv; return par; }
```

El método calcularPar realiza una operación que guarda en una variable, para como último paso retórnala. Ahora si se quiere utilizar este método, sería lo mismo que tener el dato resultante de la operación:

```
public class Main { public static void main(String[] args) {
Coche coche = new Coche(); System.out.println("El par del
coche es "+coche.calcularPar()); } }
```

Si nos damos cuenta este tipo de técnica ya las hemos utilizado en los getter, ya que la llamada al método retorna el valor de la variable.

Constructores

Todo esto está muy bien pero, ¿cómo se crea un objeto? La respuesta es muy simple: con los constructores. Hasta este momento hemos hablado que dentro de una clase tenemos métodos y atributos. Si bien es cierto, existe un tipo de método especial que permite crear un objeto de la clase en cualquier partes y estos son los constructores. Antes de continuar explicando hay que tener en cuenta una serie de cosas

- Los constructores son métodos especiales que tienen como retorno un objeto de la clase
- Su firma es especial, ya que no requiere indicar el tipo de retorno, va implicito
- Por defecto hay un constructor, el conocido como vacío
- Se pueden crear tantos constructores como se quieran, teniendo en cuenta que si se escribe uno, el vacío o por defecto desaparece

Para la clase coche estos serían algunos ejemplos de constructores

```
public class Coche { private String marca, modelo; private
int cv, cc; private int velocidad; // constructir por defecto
public Coche() { } public Coche(String marca, String modelo
```

```
public Coche() { } public Coche(String marca, String modelo,
int cv, int cc) { this.marca = marca; this.modelo = modelo;
this.cv = cv; this.cc = cc; } }
```

El primero de los constructores es el por defecto, el cual se utilizará para crear objetos que no inicialicen los atributos o lo hagan a sus datos más básicos: marca, modelo lo harán a null al ser tipo completos, mientras que cv, cc y velocidad lo harán a 0

Para poder utilizar los constructores se realizará la inicialización de una variable, igualando al método constructor creando así un objeto

```
public class Main { public static void main(String[] args) {
Coche coche1 = new Coche(); Coche coche2 = new Coche("Ford",
"Focus", 100, 2000); } }
```

La primera linea del método inicializa un objeto mediante el constructor por defecto, mientras que la segunda lo hace con el constructor de 4 parámetros, igualando cada uno de ellos a los valores pasados

Objetos

Un objeto es la representación “real” en un programa de lo escrito dentro de la clase. Cuando un objeto es creado se dice que se ha instanciado, asociando toda la funcionalidad de la clase que lo crea y dando acceso a todos sus métodos y atributos. Se puede realizar un simil donde la clase es el molde o prototipo de un coche que no puede ser utilizado salvo para la creación de todos los modelos que serán funcionales 100%. Para poder crear un objeto se utiliza la sintaxis

```
TipoClase nombre = new TipoClase(parámetros)
```

Una vez el objeto está creado y mediante el operador punto . se podrá acceder a todos los métodos y variables que la clase de instancia le ha otorgado, haciendo funcionalidad de ellos.

El método que crea el coche recibe un nombre especia que es constructor. De momento se utilizará el que todas las clases tienen por defecto sin necesidad de escribir nada.

Relación entre clases

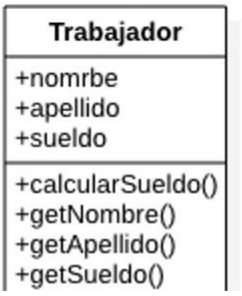
Por defecto, toda clase creada tienen sus elementos internos como son los métodos y atributos (tantos como se hayan creado), pero es posible que nos interese tener algún elemento de otra clase que sea útil para la misma. Suponer el siguiente ejemplo: se tiene una clase que representa un Trabajador de una empresa, el cual tiene nombre, apellido y sueldo

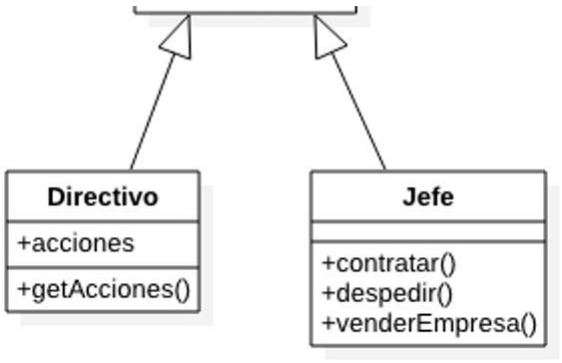
```
public class Trabajador { private String nombre, apellidos; i
nt sueldo; public String getNombre() { return nombre; } publi
c String getApellidos() { return apellidos; } public int getS
ueldo() { return sueldo; } }
```

Al mismo tiempo se necesita una clase que represente un Directivo, el cual tiene nombre, apellido, sueldo y acciones. La solución más sencilla es crear una clase muy parecida a la del ejemplo anterior pero con un atributo y un método getter más. Sin embargo esto produciría una generación de código enorme ya que habría que repetir muchas cosas. Para ello en Java existe lo que se conoce como herencia, donde una clase puede se creada a partir de otra que tiene la base que se necesita.

```
public class Directivo extends Trabajador { int acciones; pub
lic int getAcciones() { return acciones; } }
```

En el momento en el que la clase Directivo extiende de Trabajador, todos los métodos y atributos que este tiene (cuidado con los modificadores de acceso porque los private no se pasan) forman parte de la clase Directivo sin necesidad de que sean escritos. La clase que hace de base se la conoce como superclase y la clase que hereda se la conoce como subclase.





Hay que tener en cuenta que una clase puede ser superclase de varias subclases, es decir que n clases pueden heredar de una clase, pero una clase solo puede tener una superclase. En java por defecto todas las clases tienen una superclase de la cual heredan por arrastres. Esta super clases es Object