

## 3.1 Programación orientada a objetos en Java

La programación orientada a objetos (POO) representa un importante paradigma para mejorar la construcción, el mantenimiento y la utilización de software. Esta metodología de programación está compuesta de objetos con estado propio dotados de funcionalidad. Los objetos se comunican entre sí y tienen cada uno una forma propia de respuesta, que viene determinada por una serie de procedimientos que son asociados a cada objeto.

La **clase** es la unidad fundamental de programación en Java, que define la interfaz y la implementación de los objetos de esa clase. Cada **objeto** es un ejemplar de una clase o *instancia* creada en tiempo de ejecución.

La declaración de una clase tiene la forma básica:  
[public] [final] [abstract] class *NombreClase* { cuerpo de la clase }

Los **modificadores de clase** son palabras reservadas que se anteponen al nombre de las clases con el fin de definirles un comportamiento concreto. Los tipos de modificadores de clase que podemos definir son: public, final y abstract.

**public** Son accesibles desde otras clases, directamente o por herencia. Accesibles dentro del mismo paquete en el que se han declarado. Para acceder desde otro paquete es necesario importarlas (**import**). Si no se especifica ningún modificador de acceso se entiende que cualquier objeto que se encuentre en el mismo paquete podrá hacer uso de ella (**friendly**).

**final** Si se aplica esta palabra a una clase significa que no puede tener subclases. Termina la cadena de herencia.

**abstract** clase abstracta. Una clase abstracta no se instancia sino que se utiliza como clase base para la herencia. Tiene al menos un método abstracto.

**Atributos:** Determinan una estructura de almacenamiento para cada objeto de la clase.

**Métodos:** Son operaciones aplicables a los objetos y son el único modo de acceder a los atributos.

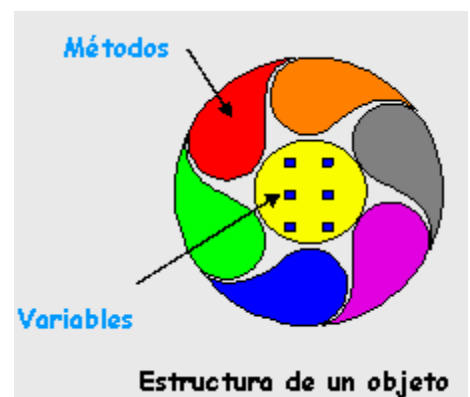
En el cuerpo de la clase se declaran los miembros de la clase:

1. **Atributos:** Determinan una estructura de almacenamiento para cada objeto de la clase.
2. **Métodos:** Son operaciones aplicables a los objetos y son el único modo de acceder a los atributos.

### 3.1.1 Clases y objetos en Java

Una **clase** es un tipo definido que determina la estructura de datos y las operaciones asociadas a ese tipo. Una clase también se puede ver como una plantilla que describe objetos que tendrán la misma estructura y el mismo comportamiento. Los servicios proporcionados por una clase, vista como un módulo, son las operaciones disponibles sobre las instancias de la clase, vista como un tipo. Las Clases tienen doble naturaleza:

1. Es un **módulo** (concepto sintáctico [reglas de combinación de las palabras] de clase) es un mecanismo para organizar el software encapsulando los componentes que lo forman.
2. Es un **tipo de datos** (concepto semántico [significado] de clase) es un mecanismo de definición de nuevos tipos de datos: describe una estructura de datos (objetos) para representar valores de un dominio y las operaciones aplicables.



Un **objeto** es:

1. Un paquete de software independiente formado por un conjunto de datos junto con los procedimientos que operan sobre estos datos.
2. Es una estructura de datos formada por tantos campos y atributos como tiene la clase.
3. El estado de un objeto viene dado por el valor de los campos.
4. Los métodos permiten consultar y modificar el estado del objeto.
5. Durante la ejecución de un programa orientado a objetos se crearán un conjunto de objetos, lo que se denomina **instanciación**.

Un objeto puede ser:

1. **Real**. En el mundo real un objeto se caracteriza porque tiene:
  - a. Una identidad.
  - b. Unas características
  - c. Un estado
  - d. Unos comportamientos
2. **Abstracto**. Un objeto es una simulación informática y se caracteriza por:
  - a. Una identidad
  - b. Unos datos
  - c. Un estado
  - d. Unos procedimientos

### 3.1.2 Componentes de una clase

Una clase en Java está compuesta de:

1. **Atributos** (variables). Determinan una estructura de almacenamiento para cada objeto de la clase. La sintaxis básica para declarar atributos es:

[final] [static] *acceso tipo lista-de-identificadores*

**final**: significa que el valor de ese campo no puede ser modificado (constante)

**static**: significa que todas las instancias comparten la misma variable (atributo de clase). Desde fuera de la clase se puede acceder a una variable static si no se marca como private. Las instancias de la clase comparten la variable static.

acceso: public | private | protected | -

tipo: es cualquier tipo primitivo o clase o array de ellos

2. **Métodos** (operaciones). Son operaciones aplicables a los objetos y representan el único modo de poder acceder a sus atributos. La sintaxis básica para declarar **métodos** es:

[final] [static] [abstract] *acceso tipo identificador(args) {...}*

**final**: significa que el método no puede ser sobrecargado por las clases que hereden de ésta

**static**: significa que es un método de clase

**abstract**: Sólo es posible en clases abstractas y no puede coincidir con los modificadores final, static, private

acceso: es igual que para campos: public | private | protected | -

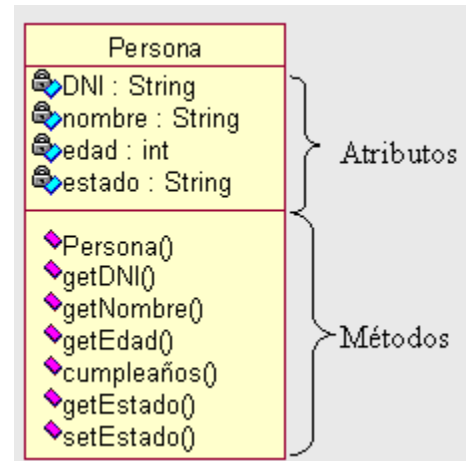
tipo: el tipo de retorno del método. Puede ser cualquier tipo primitivo, clase, array o void

args: lista de 0 o más argumentos separados por comas, donde cada argumento tiene la forma: tipo identificador

- Los métodos y los atributos se definen dentro del cuerpo de la clase.
- Todos los atributos y métodos de una clase son accesibles desde el código de la propia clase.
- Java no soporta métodos o variables globales.
- Para controlar el acceso desde otras clases, se utilizan los modificadores de acceso:
  1. **public**. Accesible desde cualquier lugar en que sea accesible la clase.
  2. **protected**. Accesible por las subclases y las clases del mismo paquete.
  3. **private**. Sólo accesible por la propia clase.
  4. **friendly** (por defecto). Accesible por las clases del mismo paquete.

Mientras que los elementos de un sistema físico se pueden agrupar en categorías, en el lenguaje Java los objetos que muestran la misma estructura y comportamiento se pueden agrupar en clases.

En esta lista de atributos (propiedades) y de métodos (operaciones) se asegura el hecho de que el conjunto de instancias de la clase **Persona** dispondrá de estas propiedades y de los valores asociados, así como de la facultad de ejecutar las operaciones citadas.



## 3.2 Clases e instancias

Para poder utilizar objetos de una clase, debemos crear una **instancia** del objeto, es decir, declarar objetos de dicha clase. Definiciones:

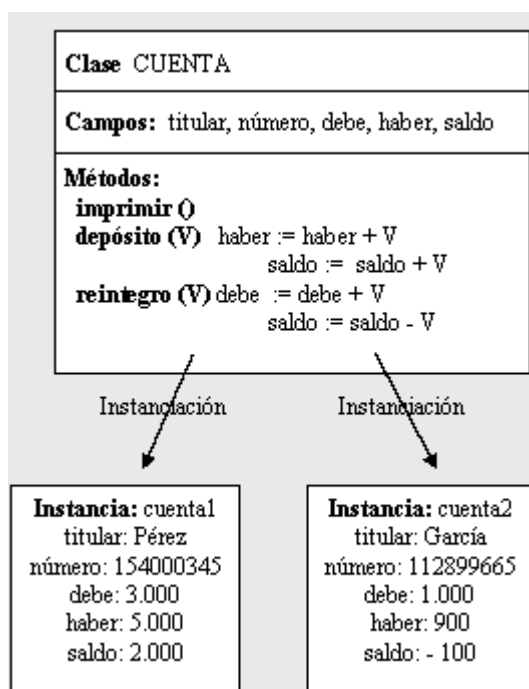
- Una clase es un generador de objetos o instancias de una clase.
- Una instancia es una estructura constituida por los atributos descritos para la clase.
- La creación de un objeto a partir de una clase se llama **instanciación**.
- Muchas instancias de una misma clase pueden existir simultáneamente en memoria.

La nomenclatura general que se adopta para la **Instanciación de objetos de una clase** es:

```
//DECLARACIÓN DE UNA CLASE
NombreClase
//Atributos y métodos de la clase
Fin NombreClase
```

```
//CREACIÓN DE INSTANCIAS DE UNA CLASE
NombreClase Objeto1
NombreClase Objeto2(Parametro1, Parametro2, ...)
```

Ejemplo:



En esta figura se define la clase Cuenta y se instancian dos objetos cuenta1 y cuenta2.

La **clase Cuenta** describe a todas las cuentas.

A partir de ella podemos dirigirnos a una cuenta en concreto dando valores a sus atributos.

- Ahora tenemos dos instancias de Cuenta (NombreClase) llamadas cuenta1 y cuenta2.
- Cada una de estas instancias es ahora operativa y sus métodos y atributos son accesibles.
- Para acceder a ellos utilizamos el nombre del objeto y el nombre del método o atributo concatenados con el operador punto ".".

```
//CREACIÓN DE INSTANCIAS DE LA CLASE
```

```
NombreClase Objeto1
```

```
//ACCESO A ATRIBUTOS DEL OBJETO
```

```
Resultado = Objeto1.NombreAtributo
```

```
//ACCESO A MÉTODOS DEL OBJETO
```

```
Objeto1.NombreMétodo()
```

### 3.2.1 ¿Qué es una instancia?

Una **instancia** de una clase en JAVA es una variable de tipo class. Para crear una instancia debemos crear un puntero a dicha instancia. En Java esta operación se denomina **referencia**.

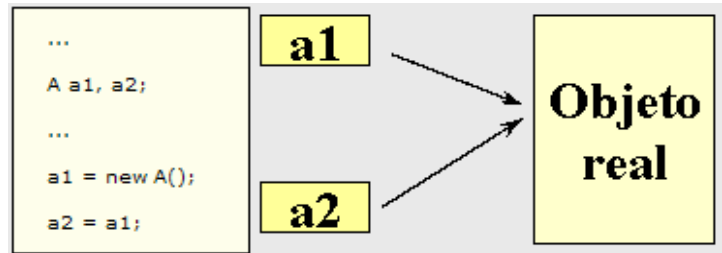
Es importante distinguir entre la **referencia** a un objeto y el objeto en sí mismo. Una referencia es una variable que indica dónde está guardado un objeto en la memoria del ordenador. Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor *null*.

Si se desea que esta referencia apunte a un nuevo objeto es necesario crear el objeto utilizando el operador **new**. Este operador reserva en la memoria del ordenador espacio para ese objeto. También es posible igualar la referencia declarada a otra referencia a un objeto ya existente.

La declaración de una variable (referencia) de una cierta clase, **no implica la creación de un objeto asociado**.

**Cualquier objeto ha de ser creado explícitamente mediante el operador new.**

Java no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los punteros.



### 3.2.2 Nomenclatura general de la definición

```
NombreInstancia = new NombreClase(parámetros)
```

```
NombreClase NombreInstancia = new NombreClase(parámetros)
```

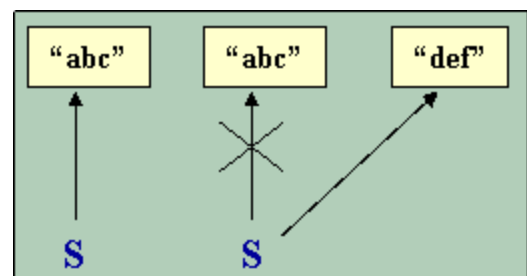
- **NombreClase:** Indica el nombre de la clase, la cual identifica el tipo del objeto.
- **NombreInstancia:** Indica el nombre de la instancia, o referencia y también se utiliza como el nombre del objeto.
- **Parámetros:** Aquí se deben escribir los parámetros que exige el *constructor* de la clase.

### 3.2.3 Liberación de espacio en memoria

Los objetos consumen recursos durante su tiempo de vida. Cuando un objeto no se va a utilizar más, debería liberar el espacio que ocupaba en la memoria de forma que las aplicaciones no la agoten.

En Java, la recolección y liberación de memoria es responsabilidad de un hilo (*thread*) llamado **automatic garbage collector** (recolector automático de basura) (gc). Este thread monitoriza el alcance de los objetos y marca los objetos que se han salido de alcance. Ejemplo:

```
String s;
/*no se ha asignado memoria todavía*/
s = new String( "abc" );
/*memoria asignada*/
s = "def";
/*se ha asignado nueva memoria (nuevo objeto)*/
```



### 3.3 Variables, objetos y referencias

Una **variable** es un nombre que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

- **Variables de tipos primitivos:** variables sencillas que contienen los tipos de información más habituales: valores boolean, caracteres y valores numéricos enteros o de punto flotante.
- **Variables referencia:** referencias o nombres de una información más compleja como por ejemplo, los objetos de una determinada clase.

Desde el punto de vista de la misión que tienen en el programa, las variables pueden ser:

- **Variables miembro de una clase:** Se definen en una clase, fuera de cualquier método. Pueden ser tipos primitivos o referencias.
- **Variables locales:** Se definen dentro de un método o más en general dentro de cualquier bloque entre llaves {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también tipos primitivos o referencias.

Java dispone de ocho tipos primitivos de variables:

- El tipo **boolean** no es un valor numérico: sólo admite los valores true o false.
- El tipo **char** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.
- Los tipos **byte**, **short**, **int** y **long** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos.
- Los tipos **float** y **double** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.
- Se utiliza la palabra **void** para indicar la ausencia de un tipo de variable determinado.

Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la siguiente tabla:

Tipo de variable	Descripción
boolean	1 byte. Valores true y false
char	2 bytes. Unicode. Comprende el código ASCII
byte	1 byte. Valor entero entre -128 y 127
short	2 bytes. Valor entero entre -32768 y 32767
int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes. Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Existen dos referencias de objetos especiales que son **this** y **super**.

### 3.4 Atributos y su inicialización

Cuando se crea un objeto, los atributos pueden:

- **No inicializarse**, tendrán asignados valores por defecto según el tipo del campo.
- **Inicializarse en la declaración**, asignándoles una expresión.

Java asigna un valor por defecto a los atributos que no se inicializan que depende del tipo de éstos. Hemos de asignar algún valor a una variable local antes de usarla. Sin embargo, no asigna ningún valor por defecto inicial a las variables locales de los métodos o constructores.

Un **constructor** es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del constructor es reservar memoria e inicializar las variables miembro de la clase. Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase. Su argumento implícito es el objeto que se está creando. Por lo general, una clase tiene varios constructores, que se diferencian por el tipo y número de argumentos (son un ejemplo típico de métodos sobrecargados).

Se llama constructor por defecto al constructor que no tiene argumentos. Existe un constructor por defecto para cualquier clase que permite crear una instancia de un objeto con el método **new X()**.

La definición de un constructor invalida al constructor por defecto. El constructor es tan importante que, si el programador no prepara ningún constructor para una clase, el compilador crea un constructor por defecto, inicializando las variables de los tipos primitivos a su valor por defecto, y los Strings y las demás referencias a objetos a null.

Ejemplos:

Tipo del campo	Valor Inicial
<i>boolean</i>	0
<i>char</i>	'\u0000'
<i>byte, short, int, long</i>	0
<i>float, double</i>	+0.0f ó +0.0d
<i>referencia a objeto</i>	null

```
private String miNombre = "Juan " + "Carlos";
private int edad = obtenerMiEdad();
```

## 3.5 Métodos

Los **métodos** son funciones definidas dentro de una clase que determinan su comportamiento.

```
tipoDevuelto nombreMetodo (listaParametros)
{
    //instrucciones del método
}
```

La **invocación** a los métodos desde una instancia se hace mediante el **operador de acceso ( . )**.

```
referenciaObjeto.nombreMetodo(listaArgumentos);
referenciaClase.nombreMetodoEstatico(listaArgumentos);
```

- Todo método debe finalizar con una sentencia **return** (excepto los métodos de tipo void) seguida de una expresión del mismo tipo que el valor de retorno.
- Los métodos estáticos sólo pueden acceder a miembros estáticos.
- **Sobrecarga de métodos.** Podemos tener métodos con el mismo nombre pero diferente número y/o tipo de los argumentos. En ese caso el compilador escoge el método a invocar en función del número y tipo de los argumentos suministrados

Lista de **argumentos**:

- El lenguaje Java sólo pasa la lista de argumentos de tipos básicos por valor.
- Cuando se pasa un objeto instanciado como argumento a un método, el valor del argumento es el puntero al objeto.
- Los contenidos del objeto se pueden cambiar dentro del método al que se ha llamado (los objetos se pasan por referencia), pero el puntero no se puede cambiar.

Ejemplo:

```
class X
{
    public int producto (int a, int b)
    { return a * b; }
    public double producto ( double a, double b)
    { return a * b; }
}
```

```

X x = new X();
/*Instanciamos un objeto x de la clase X*/
x.producto(2, 3);
/*Se llama al método producto de enteros*/
x.producto(2.0, 3.5);
/*Se llama al método producto de doubles*/

```

## 3.6 Atributos y métodos estáticos

### Variables static

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama **variables de clase** o variables **static**. Estas se suelen utilizar para definir variables comunes para todos los objetos de la clase o que sólo tienen sentido para toda la clase. Las variables de clase son lo más parecido que Java tiene a las variables globales de C/C++.

Las variables de clase se crean anteponiendo la palabra **static** a su declaración, y en el momento en el que pueden ser necesarias, como por ejemplo, cuando se va a crear el primer objeto de la clase, cuando se llama a un método static o cuando se utiliza una variable static de dicha clase. Para llamar a las variables static se suele utilizar el nombre de la clase, en lugar del nombre de un objeto de la clase.

Si no se les da valor en la declaración, las variables miembro static se inicializan con los valores por defecto para los tipos primitivos, y con *null* si es una referencia.

Lo **importante** es que las variables miembro static se inicialicen siempre antes que cualquier objeto de la clase.

### Métodos static

Análogamente, también puede haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama **métodos de clase** o **métodos static**.

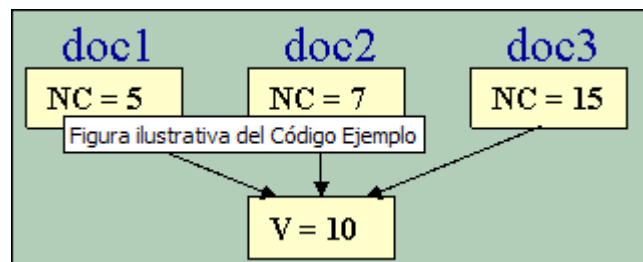
- Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumentos implícitos, ni pueden utilizar la referencia **this**.
- Un ejemplo típico de métodos static son los métodos matemáticos de la clase `java.lang.Math` (`sin()`, `cos()`, `exp()`, `pow()`, etc.).
- Por lo general, el argumento de estos métodos será de un tipo primitivo y se le pasará como argumento explícito. Estos métodos no tienen sentido como métodos de objeto.

Ejemplo:

```

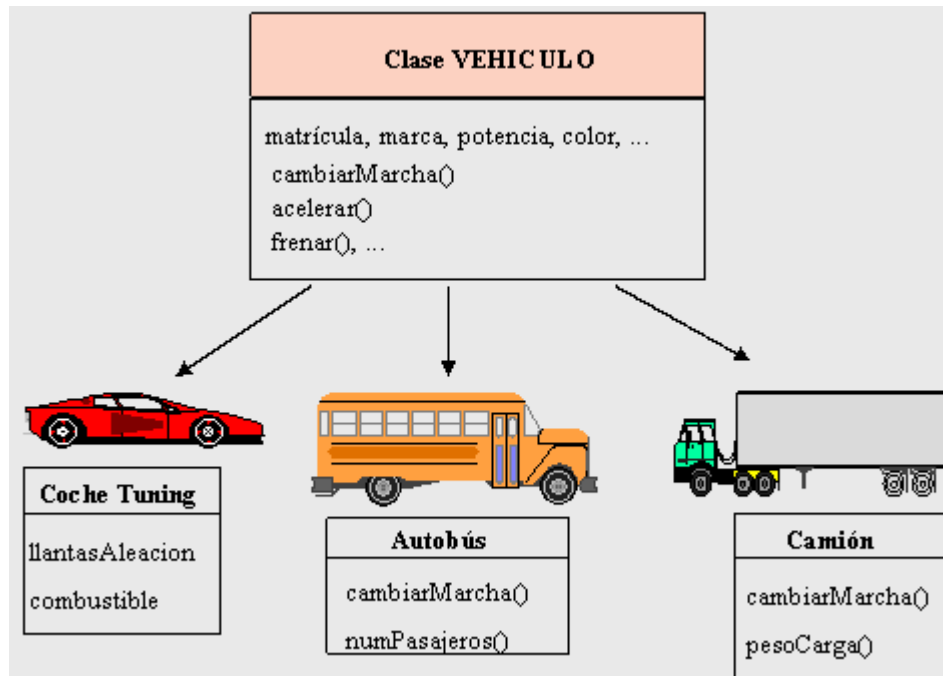
class Documento {
    static int version = 10;
    int numero_de_capitulos;
    void         añade_un_capitulo()
    { numero_de_capitulos++;}
    static void  modifica_version  ( int )
    { version = i; }
}

```



## 3.7 Herencia de clase

La **herencia** es el mecanismo mediante el cual se crean nuevos objetos definidos en términos de objetos ya existentes. Se establece una estructura jerárquica en la que cada clase hereda los atributos y los métodos de las clases que están por encima de ella. Un objeto busca los métodos y los atributos en su clase y después en sus Superclases.



Cada objeto tiene acceso a una referencia de sí mismo, llamada referencia **this**. Esta referencia se usa para referirse al propio objeto y se usa explícitamente para referirse tanto a las variables de instancia como a los métodos de un objeto. Al acceder a las variables de instancia de una clase, la palabra clave **this** hace referencia a los miembros de la propia clase.

Ejemplo:

```

public class MiClase {
    int i;

    public MiClase () {
        i=10; // Este constructor establece el valor de i
    }
    public MiClase (int valor) {
        this.i=valor; // asignamos valor a la variable i
    }
    public void suma_a_i (int j) {
        i=i+j;
    }
}
  
```

Aquí **this.i** se refiere al entero *i* en la clase *MiClase*.

Una **superclase** de una clase significa el ancestro más directo de la clase así como a todas sus clases ascendentes. Los coches, autobuses y los camiones se pueden agrupar dentro de la superclase *vehículo*. **Subclase** es la clase derivada, es decir, la clase que desciende de otra clase. Una subclase hereda el estado y el comportamiento de su superclase. Cada subclase puede tener nuevos atributos y métodos y/o redefinir los heredados.

Para heredar de una clase utilizamos la palabra clave **extends**. En Java sólo es posible extender de una clase, que es lo que se denomina **Herencia Simple**.

```

class Camion extends Vehiculo
{
    ...
}
  
```



La clase Camion hereda todos los atributos y métodos de Vehiculo. A partir de aquí, podemos:

- **Extender** la clase Vehiculo añadiendo nuevos métodos a la clase Camion aumentando su interfaz.
- **Anular** métodos de Vehiculo con la propia implementación de Camion.

La referencia **super** se usa para referirse a métodos de la clase padre. Si se necesita llamar al método padre dentro de una clase que ha reemplazado ese método, se puede hacer referencia al método padre con la palabra **super**. Un ejemplo de su uso es el siguiente:

```
import MiClase;
public class MiNuevaClase extends MiClase {
    public void suma_a_i (int j) {
        i=i+(j/2);
        super.suma_a_i(j);
    }
}
```

### 3.8 Clases abstractas

Una clase abstracta es una clase de la que no se pueden crear objetos, es decir, no se puede instanciar. Tienen la función de encapsular un concepto abstracto que compartirán sus subclasses. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles una interfaz común y algunos métodos de utilidad general. Es decir, no proporcionan la implementación de todos sus métodos, y éstos se declaran como **abstract**. Una clase abstracta se declara anteponiendo el modificador **abstracta** la palabra clave **class**.

```
abstract class NombreClaseAbstracta {
    ...
}
```

Una clase abstracta puede tener:

- **Métodos abstractos.** En este caso no se da la definición del método.
  - Si una clase tiene algún método *abstract* es obligatorio que la clase sea *abstract*, pero una clase puede declararse como *abstract* aunque no tenga ningún método abstracto.
  - En cualquier subclase este método deberá ser redefinido o declarar como *abstract* el método y la subclase.
  - Un método *abstract* no puede ser *static* puesto que estos últimos no pueden ser redefinidos.
- **Métodos que no son abstractos.** Aunque se puedan crear objetos de esta clase, sus subclasses heredarán el método completamente a punto para ser utilizado.

Ejemplos:

```
abstract class Figura {
    Punto posicion;
    public void mover(Punto nuevaPos) {
        posicion = nuevaPos;
        dibujar(); }
    abstract public void dibujar();
}
```

El siguiente ejemplo no se puede extender porque es **final**. Suele utilizarse cuando resultaría *peligroso* permitir que las subclasses dieran otra implementación de ciertos métodos.

```
final class Circulo extends Figura {
    public void dibujar() {
        ...
    }
}
```

### 3.9 Características de la encapsulación

Técnica que permite localizar y ocultar los detalles de un objeto respecto al usuario, aislándolo del aspecto interno. Previene que un objeto sea manipulado por operaciones distintas a las que le son propias, así como que

sus operaciones manipulen datos ajenos a ellas. El empaquetado de atributos y métodos dentro de un objeto mediante una **Interfaz de Mensajes** (conjunto de operaciones externamente visibles que definen el comportamiento de un objeto) es lo que denominamos **encapsulación**.

Su objetivo es garantizar que todas las interacciones del objeto tengan lugar a través de un sistema de mensajería predefinido con el que sea capaz de entenderse. No existe ninguna otra manera con la que un objeto externo pueda acceder a los atributos y métodos escondidos dentro de la interface de mensajes de otro objeto.

- Los objetos agrupan su estado (atributos) y su comportamiento (métodos).
- Hace que el código sea más fácil de mantener.
- Fuerza al usuario a utilizar una interfaz para acceder a los datos. Es lo que se denomina **Ocultación de Datos**( La palabra reservada `private` permite una accesibilidad total desde cualquier método de la clase, pero no desde fuera de esta. Como los datos son inaccesibles, la única manera de leer o escribirlos es a través de los métodos de la clase. Esto proporciona consistencia y calidad) .

Un objeto tiene una interfaz pública que otros objetos pueden usar para comunicarse con él. El objeto puede mantener información privada y métodos que pueden cambiar sin que esto afecte a otros objetos que dependen de él.

Las **ventajas** que ofrece la **encapsulación** son las siguientes:

- Supresión de los riesgos de corrupción de los datos por eventuales conflictos entre funciones del programa.
- Reducción del tamaño del código fuente.
- Mejor legibilidad de los programas (las funciones se reagrupan en las entidades que las manejan).
- Constitución de bibliotecas de objetos genéricos reutilizables (ladrillos elementales).
- Facilita el mantenimiento y extensión.

## 3.10 Polimorfismo

Significa múltiples formas y es la propiedad que tiene un lenguaje de programación de que una clase pueda tener diferentes comportamientos. El polimorfismo es una aplicación particular de la herencia de comportamiento. Una misma operación (un mismo elemento de comportamiento) podrá interpretarse de diversas maneras según la posición en la que se encuentre dentro de la jerarquía. Normalmente se necesita el empleo de clases abstractas.

El **polimorfismo** está asociado a una **ligadura dinámica** (*dynamic binding*), es decir, la asociación de un método con su nombre no se determina hasta el momento de su ejecución. Un objeto tiene sólo una forma, pero una variable tiene muchas y puede apuntar a un objeto de diferente manera. En Java hay una clase que es la clase padre de todas las demás: `java.lang.Object`. Un método de esta clase (por ejemplo: `toString()` que convierte cualquier elemento de Java a cadena de caracteres), puede ser utilizada por todos.

Las **ventajas** que nos aporta el **polimorfismo** son las siguientes:

- Da uniformidad a la sintaxis.
- Disminuye la cantidad del código a escribir (gracias a la eliminación de las estructuras alternativas con opciones múltiples).
- Facilita el tratamiento de colecciones heterogéneas.

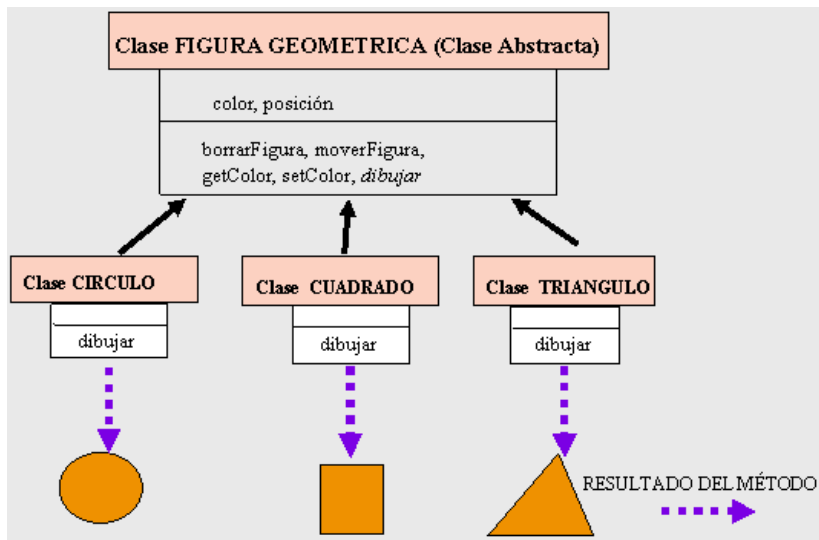
Ejemplos:

Java permite apuntar a un objeto con una variable definida como tipo de clase padre. La siguiente sentencia la clase **Jefe** tiene acceso a los métodos de la clase **Empleado**.

```
Empleado e = new Jefe ();
```

Pero sólo se puede acceder a las partes del objeto que pertenecen a la clase **Empleado**. Ya que las partes específicas de la clase **Jefe** no se ven. Este efecto se consigue porque, para el compilador, `e` es sólo una variable de tipo `Empleado`, no de `Jefe`.

```
e.departamento = "Finanzas"; //Incorrecto
```



```
abstract class FigGeo
    ... dibujar()
```

```
class CUADRADO extends FigGeo
```

```
    ... dibujar(){...}
```

```
class CIRCULO extends FigGeo
```

```
    ... dibujar() {...}
```

```
class TRIANGULO extends FigGeo
```

```
    ... dibujar() {...}
```

```
Class Prueba
```

```
    FigGeo fg[] new FigGeo[8]
```

```
    ...
```

```
    for (i=0;i<=7;i++) fg[i].dibujar();
```