# PARADIGMA DE ORIENTACIÓN A OBJETOS

Guillermo Vega Gorgojo – guiveg@tel.uva.es
María Ángeles Pérez Juárez – mperez@tel.uva.es

# INDICE

# 1 INTRODUCCIÓN

En este documento se abordan los conceptos más importantes del paradigma de Orientación a Objetos, cuya comprensión es necesaria para la adecuada comprensión de diversos artefactos del Lenguaje de Modelado Unificado.

# 2 ¿QUÉ ES UN LENGUAJE DE PROGRAMACIÓN?

Un ordenador es una máquina que solo comprende las instrucciones que se le proporcionen en determinados formatos. Habitualmente, se le indica al ordenador lo que debe hacer a través de una colección de instrucciones, también denominadas sentencias o proposiciones, inteligibles para él que se denominan **programa**.

La **programación** serían el conjunto de operaciones que conducen a la construcción de programas[1] a partir de algoritmos, diagramas u otras especificaciones. Un **lenguaje de programación** sería el idioma utilizado para escribir programas o secuencias de instrucciones a partir de un determinado algoritmo, diagrama u otra especificación. Un lenguaje de programación consiste en un conjunto de reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos, respectivamente.

Hay quienes diferencian entre lenguaje de programación y lenguaje informático, puntualizando que los lenguajes informáticos engloban a los lenguajes de programación y a otros más, como, por ejemplo, los Lenguajes de Marcado (ML, Markup Language)[2] como es el Lenguaje de Marcas de Hipertexto (HTML, Hypertext Markup Language)[3].

# 3 CLASIFICACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

En esta sección se presenta una posible clasificación de los lenguajes de programación. No obstante, es importante tener en cuenta que son muchas y diferentes las clasificaciones de los lenguajes de programación que pueden encontrarse en los diferentes textos y manuales.

[1] En el contexto de la Ingeniería de Software, la tarea de programar sería solo una parte de las tareas necesarias para construir un sistema o producto software, objetivo que, como ya es sabido, involucra muchas más tareas además de la codificación propiamente dicha.

[2] Los Lenguajes de Marcado (ML, Markup Languages) se utilizan para especificar la estructura de documentos. Cada Lenguaje de Marcado se vincula a un determinado tipo de documento. Los Lenguajes de Marcado se definen a partir de los metalenguajes. El vínculo entre un Lenguaje de Marcado y el metalenguaje a partir del cual está definido es un DTD (Document Type Definition, Definición de Tipo de Documento). El Lenguaje de Marcado Estándar Generalizado (SGML, Standard Generalized Markup Language) es el metalenguaje a partir del cual está definido el Lenguaje de Marcas de Hipertexto (HTML, Hypertext Markup Language) que es un ejemplo típico de Lenguaje de Marcado que se utiliza para la creación de lo que se conoce como páginas web, aunque existe una reformulación de HTML, denominada XHTML (extensible HTML), como una *aplicación* del metalenguaje XML (Extensible Markup Language). El término aplicación se refiere a los Lenguajes de Marcado, en ocasiones se les denomina *aplicaciones*, puesto que se crean *aplicando* las normas de un determinado metalenguaje.

[3] El Lenguaje de Marcas de Hipertexto (HTML, Hypertext Markup Language) es el primer lenguaje que surge para la creación de lo que conocemos como páginas web.

## 3.1 SEGÚN EL NIVEL DE ABSTRACCIÓN

La clasificación de los lenguajes de programación según su nivel de abstracción los divide en lenguajes de bajo y de alto nivel. Esta clasificación está íntima e inevitablemente ligada a la evolución de los lenguajes de programación puesto que los de alto nivel son cronológicamente posteriores a los de bajo nivel.

Los **lenguajes de programación de bajo nivel** fueron los primeros que surgieron y se denominaron así porque estaban directamente relacionados con el hardware del ordenador ya que el programador introducía una serie de códigos numéricos (binarios[4]) que la máquina interpretaba como instrucciones. Era por tanto imprescindible que el programador conociera el funcionamiento de la máquina al más bajo nivel y los errores de programación eran muy frecuentes.

El lenguaje de más bajo nivel es, por excelencia, el **código máquina**. A éste le sigue el **lenguaje ensamblador**, ya que al programar en ensamblador se trabaja con los registros de memoria del ordenador de forma directa. Se cuenta con un conjunto muy reducido y sencillo de instrucciones, las cuales se traducen fácilmente al lenguaje de la máquina ya que cada instrucción en lenguaje ensamblador equivale a una instrucción en lenguaje máquina, en realidad se utilizan mnemotécnicos[5] en lugar de cadenas o secuencias de bits. Para programar en lenguaje ensamblador se requiere aún un conocimiento bastante profundo por parte del programador de cómo funciona el ordenador, el uso de la memoria, etc.

Los **lenguajes de programación de alto nivel** surgieron con posterioridad con el primer compilador de FORTRAN (`FOR`mula `TRAN`slation), que, como su nombre indica, comenzó como un simple esfuerzo de traducir un lenguaje de fórmulas al lenguaje máquina, facilitando así la labor a los programadores.

A partir de este punto, se han desarrollado innumerables lenguajes, que siguen la misma filosofía que intenta facilitar la labor del programador, aumentando su productividad. Los lenguajes de alto nivel utilizan instrucciones, que se aproximan al lenguaje natural, normalmente en inglés, pero siguiendo estrictas reglas gramaticales. Los lenguajes de alto nivel son normalmente fáciles de aprender. Por ejemplo, `IF CONTADOR = 10 THEN STOP` puede utilizarse para pedir al ordenador que pare si el valor de la variable `CONTADOR` es igual a `10`.

Finalmente, hay que mencionar también que hay lenguajes de programación que son considerados por algunos expertos como **lenguajes de programación de nivel medio** (como podría ser el caso del lenguaje C que otros expertos consideran como un lenguaje de alto nivel[6]) ya que poseen ciertas características que los acercan a los lenguajes de bajo nivel pero teniendo, al mismo tiempo, ciertas cualidades que lo hacen un lenguaje más cercano al humano y, por tanto, de alto nivel.

## 3.2 SEGÚN LA FORMA DE EJECUCIÓN

Es necesario percatarse de que, aunque al utilizar un lenguaje de alto nivel el programador se distancie de esta forma del hardware del ordenador, éste sigue trabajando en lenguaje máquina ya que el lenguaje máquina es el único que entiende un ordenador. Por ello, se hace necesaria una traducción a

[4] Es decir, series de `1's` y `0's` como por ejemplo `01011110`.

[5] Por ejemplo, `ADD B,1 (instrucción sumar)` o `MOV A,B (instrucción asignar (mover))`.

[6] El lenguaje C dispone de las estructuras típicas de los lenguajes de alto nivel, pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel.

una secuencia de instrucciones inteligibles por el ordenador. Esta labor es la que llevan a cabo los compiladores e intérpretes que se pueden denominar genéricamente traductores:

Un **traductor** sería un programa que traduce el código fuente de un programa a lenguaje máquina.

Un **intérprete** sería un traductor que toma un programa fuente escrito en un lenguaje de alto nivel y lo traduce a lenguaje máquina y ejecuta línea a línea. Ejemplos típicos de lenguajes interpretados son HTML, XHTML (Extensible HTML), JavaScript o PHP (PHP Hypertext Preprocesor).

Un **compilador** es un traductor que toma un programa fuente escrito en un lenguaje de alto nivel y lo traduce a lenguaje de máquina, produciendo un **programa objeto** que se puede utilizar posteriormente. Un ejemplo típico de un lenguaje compilado es C.

Es decir, el compilador es un programa que se encarga de la traducción global del programa realizado por el programador, operación que recibe el nombre de **compilación**. El programa es traducido completamente antes de que se ejecute, por lo que la ejecución se realiza en un periodo breve de tiempo. El intérprete por el contrario lleva a cabo una traducción inmediata en el momento de la ejecución, es decir, va traduciendo y ejecutando las instrucciones una a una haciendo que el proceso de ejecución[7] requiera un periodo de tiempo mayor del que necesitaría un programa compilado. En este caso no se produce ningún programa objeto por lo que la próxima vez que se utilice una instrucción, se debe interpretar y traducir a lenguaje máquina nuevamente. Por ejemplo, durante el procesamiento repetitivo de las sucesivas iteraciones de un bucle, cada instrucción tendrá que volver a ser interpretada cada vez que se ejecute una iteración del bucle.

En resumen, un programa interpretado es más lento en tiempo de ejecución que uno compilado, puesto que el código se va traduciendo en tiempo de ejecución, pero más rápido en tiempo de desarrollo, porque no es necesario compilarlo después de cada modificación realizada en el programa.

## 3.3 SEGÚN EL PARADIGMA DE PROGRAMACIÓN

Un **paradigma de programación** proporciona y determina la visión y métodos de un programador en la construcción de un programa. Diferentes paradigmas resultan en diferentes formas de pensar la solución de problemas y, con ello, en diferentes estilos de programación.

Un lenguaje de programación puede estar basado en uno o más paradigmas. Por ejemplo, **Java** es un lenguaje de programación basado en el paradigma de la Programación Orientada a Objetos (POO, Object Oriented Programming (OOP)) mientras que Python soporta múltiples paradigmas.

La siguiente clasificación de los lenguajes de programación se refiere al paradigma de programación en el que se basan. Es importante aclarar que existen otros paradigmas y subparadigmas de programación no incluidos dentro de la clasificación que a continuación se presenta. Además, existen conflictos en las definiciones y alcances de ciertos paradigmas.

**Paradigma Imperativo**

Concibe la programación como una secuencia de instrucciones que cambian el estado de un programa.[8]

**Paradigma Declarativo[9]**

[7] En realidad, de traducción y ejecución.

[8] El código máquina en general está basado en el paradigma imperativo.

No se basa en indicar cómo se hace algo, es decir, en especificar cómo se logra un objetivo paso a paso, sino en describir (declarar) cómo es ese algo que se ha planteado como objetivo a lograr. Este paradigma se orienta por tanto a describir las propiedades de la solución buscada, dejando indeterminada la forma de llegar a esa solución.

Hay textos en los que se distingue únicamente entre estos dos paradigmas de programación contrapuestos, puesto que un lenguaje, o se orienta a especificar el objetivo buscado, o se orienta a especificar cómo se logra dicho objetivo paso a paso.

Otros paradigmas de programación serían los siguientes:

**Paradigma Estructurado (procedural o procedimental)**: La programación se divide en bloques (procedimientos, operaciones o funciones) que pueden o no comunicarse entre sí. Se utilizan tres estructuras: secuencial[10], selectiva[11] e iterativa[12]. El tener un programa organizado en procedimientos, permite reutilizar código programado y facilita la compresión de la programación. Sin embargo, si se trata de un gran proyecto que maneje datos complejos, no resulta el enfoque de trabajo más adecuado y deriva en un mantenimiento costoso del software.

**Paradigma Orientado a Objetos**

La Programación Orientada a Objetos es un paradigma de programación que define los programas en términos de clases de objetos. La Programación Orientada a Objetos expresa un programa como un conjunto de objetos que colaboran entre ellos para realizar tareas.

Un objeto contiene toda la información, los denominados atributos, que permiten definirlo e identificarlo frente a otros objetos pertenecientes a otras clases, e incluso frente a objetos de una misma clase, al poder cada objeto tener diferentes valores para sus atributos. A su vez, dispone de mecanismos de interacción, denominados métodos, que favorecen la comunicación entre objetos de una misma clase o de clases distintas y, en consecuencia, el cambio de estado en los propios objetos. De esta forma, mediante este paradigma se permite una mayor proximidad de los conceptos de modelado a las entidades del mundo real, así como un modelado conjunto de aspectos estáticos y dinámicos del problema.

Los objetos deben tratarse como unidades indivisibles, en las que no se separan, ni deben separarse, información o datos (atributos) y procesamiento (métodos). Dada la existencia de una clase de objetos, que al contar con una serie de atributos definitorios, requiere de unos métodos para poder tratarlos, lo que hace que ambos conceptos estén íntimamente entrelazados, el programador debe pensar globalmente en ambos, y no debe nunca separar o dar mayor importancia a los atributos en detrimento de los métodos, ni viceversa. Hacerlo puede llevar al programador a adquirir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen o procesen esa información por otro, llegando a una programación estructurada encubierta en un lenguaje de Programación Orientado a Objetos. De hecho, uno de los principales problemas de la Programación Orientada a Objetos es que las ideas se conocen muy bien, pero, en ocasiones, se hace una utilización muy poco purista de las mismas. [ver sección 4.8 Mal uso de la Orientación a Objetos].

---

[9] Un ejemplo de lenguaje declarativo es el Lenguaje de Consultas Estructurado (SQL, Structured Query Language) con el que se puede especificar que se desea `obtener los nombres de todos los empleados que tengan más de 32 años`.

[10] Implica una serie ordenada de acciones a ejecutar.

[11] Implica la elección entre dos o más acciones a ejecutar en base a una o varias condiciones evaluadas utilizando para ello lo que típicamente se denomina una sentencia para el control del flujo del programa.

[12] Implica la ejecución repetida de una o varias acciones utilizando para ello lo que típicamente se denomina un bucle.

La Programación Orientada a Objetos facilita la construcción y mantenimiento de programas y módulos que permiten resolver programas complejos.

Finalmente, hay que señalar que en algunos textos y manuales se distingue la Programación Orientada a Objetos sin clases, es decir, con objetos, pero sin clases de objetos, la cual se denomina **Programación Basada en Objetos**, no Orientada a Objetos.

**Paradigma Orientado a Eventos**

La estructura y ejecución de un programa se ve determinada por los sucesos que ocurran en el sistema o que se provoquen.

Para entender este paradigma de programación se puede oponer a lo que no es. Así, mientras en la programación secuencial (estructurada) es el programador el que define el flujo del programa, en la programación orientada a eventos será el propio usuario del programa el que determine su flujo. Aunque en la programación secuencial puede haber intervención de un agente externo al programa, estas intervenciones ocurrirán cuando el programador lo haya determinado, y no en cualquier momento como puede ocurrir en el caso de la programación orientada a eventos.

El programador debe definir los eventos que manejará su programa y las acciones que se realizarán al producirse cada uno de ellos a través de lo que se conoce como manejadores de eventos[13]. Los eventos soportados estarán determinados por el lenguaje de programación utilizado.

Al comenzar la ejecución del programa se ejecutará el código inicial y a continuación el programa quedará bloqueado hasta que se produzca algún evento. Cuando alguno de los eventos esperados por el programa tenga lugar, el programa pasará a ejecutar el código del correspondiente manejador de evento. Por ejemplo, si el evento consiste en que el usuario ha pulsado un botón, se ejecutará el código vinculado al correspondiente manejador de evento, que puede que lo que haga sea mostrar una ventana de alerta con un aviso al usuario.

La programación orientada a eventos se utiliza de forma habitual para la construcción de interfaces que interactúan con el usuario. Un ejemplo sería el caso del lenguaje de script del lado del cliente[14] JavaScript que es un lenguaje interpretado[15] que se utiliza en la construcción de interfaces web.

# 4 LA ORIENTACIÓN A OBJETOS

## 4.1 INTRODUCCIÓN A LA ORIENTACIÓN A OBJETOS

En primer lugar, cabe destacar la diferencia que existe entre la Programación Orientada a Objetos y un lenguaje de programación orientado a objetos. La Programación Orientada a Objetos es una filosofía, un paradigma de programación, con su teoría y su metodología, mientras que un lenguaje de programación orientado a objetos es un lenguaje de programación que permite trabajar con dicha filosofía. Por tanto, el procedimiento a seguir para desarrollar aplicaciones orientadas a objetos será

[13] En los textos ingleses, los manejadores de eventos se denominan `event handlers`.

[14] En los textos ingleses, se habla de JavaScript como un `client-side scripting language`. JavaScript es un lenguaje de script porque la forma de utilizarlo es insertando pequeños scripts o porciones de código típicamente en documentos con extensión `.html` o páginas web.

[15] El código JavaScript es interpretado por el cliente, no por el servidor, como ocurre, por ejemplo, con PHP.

aprender primero la filosofía o forma de pensar y después el lenguaje de programación, puesto que la filosofía es única mientras que lenguajes de programación que aplican dicho paradigma existen varios.

Por tanto, cuando se hace referencia a la Programación Orientada a Objetos no se está hablando de una serie de características nuevas añadidas a un lenguaje de programación, sino de una nueva forma de pensar acerca del proceso de descomposición de problemas y de desarrollo de soluciones de programación. La Programación Orientada a Objetos surge como un intento para dominar la complejidad que, de forma innata, posee el software. Tradicionalmente, la forma de enfrentarse a esta complejidad ha sido emplear el paradigma de programación estructurada, que consiste en descomponer el problema objeto de resolución en subproblemas y más subproblemas hasta llegar a acciones muy simples y fáciles de codificar, es decir, se trataba de descomponer el problema en acciones, en verbos. En el ejemplo de un programa que resuelva ecuaciones de segundo grado, el problema podría descomponerse en las siguientes acciones: en primer lugar, `solicitar` el valor de los coeficientes `a`, `b` y `c`, a continuación, `calcular` el valor del discriminante y, por último, en función del signo del discriminante[16], `calcular` ninguna, una o dos raíces. Tal y como puede observarse, el problema se ha descompuesto en acciones, en verbos, por ejemplo, el verbo `solicitar`, el verbo `calcular`, etc.

El paradigma de Programación Orientada a Objetos propone otra forma de descomponer problemas, en concreto, propone la descomposición en objetos. La programación tradicional no orientada a objetos basa su funcionamiento en el concepto de procedimiento o función entendida como un conjunto de instrucciones que operan sobre unos argumentos y producen un resultado. De esta forma, un programa se articula como una sucesión de llamadas a funciones, ya sean proporcionadas por el propio lenguaje, desarrolladas por el ingeniero software, etc. En la Programación Orientada a Objetos, el elemento básico no es la función, sino un ente denominado precisamente **objeto**.

En la observación de cómo se plantea un problema cualquiera en la realidad se descubre que lo que hay son entidades, que también podrían denominarse agentes u objetos. Estas entidades poseen un conjunto de propiedades o atributos, así como un conjunto de métodos mediante los cuales muestran su comportamiento. En la observación de la realidad, también es posible descubrir todo un conjunto de interrelaciones entre las entidades, guiadas por el intercambio de mensajes, las entidades involucradas responden a estos mensajes mediante la ejecución de ciertas acciones. Así, en la Programación Orientada a Objetos una aplicación se define como un conjunto de objetos con sus características y las operaciones que se pueden realizar sobre los mismos. Además, los objetos de una aplicación no están aislados, sino que se comunican entre ellos mediante el envío de mensajes.

A continuación, se plantea un ejemplo que se espera ayude al lector a aclarar los conceptos expuestos.

Imagínese la siguiente situación: Un domingo por la tarde alguien está en casa viendo la televisión, y de repente su madre siente un fuerte dolor de cabeza, por lo que el sujeto en cuestión busca una caja de aspirinas.

A continuación se describe dicha situación en clave de objetos: el objeto `hijo` ha recibido un mensaje procedente del objeto `madre`. El objeto `hijo` responde al mensaje o evento ocurrido

[16] En álgebra, el discriminante de un polinomio es una cierta expresión que permite determinar las características de la solución de dicho polinomio. Así, por ejemplo, una ecuación de segundo grado `ax2 + bx + c = 0`, puede tener una, dos o ninguna solución dependiendo del valor de su discriminante `D = b2 - 4ac`. `Si D > 0` existen dos soluciones reales diferentes, si `D = 0` las soluciones son repetidas, es decir, que ambas soluciones son la misma y si `D < 0` existen dos soluciones complejas o imaginarias diferentes – un número imaginario es un número cuyo cuadrado es negativo, un número complejo describe la suma de un número real y de un número imaginario –. Por lo tanto, calcular el valor del discriminante permite anticipar los tipos de soluciones que se llegarán a encontrar.

mediante la acción `buscar aspirina`. La `madre` no tiene que indicar al `hijo` dónde debe buscar, puesto que es responsabilidad del `hijo` el resolver el problema como considere más oportuno. Al objeto `madre` le basta con haber emitido un mensaje.

El hijo no encuentra aspirinas en el botiquín y decide acudir a la farmacia de guardia más cercana para comprar aspirinas. En la farmacia es atendido por una señorita que le pregunta qué desea, a lo que el hijo responde "una caja de aspirinas, por favor". La farmacéutica desaparece para regresar al poco tiempo con una caja de aspirinas en la mano. El hijo paga el importe, se despide y vuelve a su casa. Allí le da un comprimido a su madre, la cual al cabo de un rato comienza a experimentar una notable mejoría hasta la completa desaparición del dolor de cabeza.

El objeto `hijo`, como objeto responsable de un cometido, en este caso concreto el conseguir una aspirina, entra en relación con un nuevo objeto, la `farmacéutica`, quien responde al mensaje o evento de petición del objeto `hijo` con la acción `buscar aspirina`. El objeto `farmacéutica` es ahora el responsable de la búsqueda de la aspirina. El objeto `farmacéutica` lanza un mensaje al objeto `hijo` cuyo objetivo es `solicitar el pago` del importe correspondiente, y el objeto `hijo` responde a tal evento con la acción `pagar`.

Tal y como puede observarse en esta situación existen objetos que se diferencian cada uno del resto por un conjunto de **características** o **propiedades**, y por un conjunto de **acciones** que realizan en respuesta a unos eventos que se originaban en otros objetos, o en el entorno, de los cuales tienen noticia porque reciben los oportunos mensajes. También es posible observar que aunque todos los objetos involucrados `hijo`, `madre` y `farmacéutica` tienen características distintas, como el color del cabello, el grado de simpatía o el peso, todos tienen un conjunto de atributos en común por ser ejemplos de una entidad superior denominada `ser humano`. Este patrón de objetos, en nuestro ejemplo el `ser humano`, se denomina **clase**. Es decir, los objetos son **instancias** o casos concretos de las clases o tipos de objetos, que pueden verse como plantillas que definen los **atributos** y los **métodos** comunes a todos los objetos de un cierto tipo. La clase `SerHumano` tendrá, entre sus muchos atributos, por ejemplo, el `color del cabello`, el `color de los ojos`, la `estatura`, el `peso`, la `fecha de nacimiento`, etc. A partir de una clase se podrán generar todos los objetos que se desee especificando valores particulares para cada uno de los atributos definidos por la clase. Así, por ejemplo, se tendrá el objeto `farmacéutica`, cuyo `color de cabello` es `rubio`, su `color de ojos` es `azul`, su `estatura` es `175 cm`, su `peso` es `50 Kg`, y así sucesivamente.[17]

El resto de esta sección 4, dedicada a La Orientación a Objetos, del presente documento aborda los fundamentos de dicho paradigma de programación, analizando conceptos como objetos, clases, mensajes, etc., y, en aquellos casos en los que se presentan códigos de ejemplo, estos se han realizado usando el lenguaje de programación Java. Como ya se ha comentado, el lenguaje de programación Java incorpora el uso de la orientación a objetos como uno de los pilares básicos y fundamentales del lenguaje. Esto constituye una importante diferencia con respecto a otros lenguajes de programación como C++ que, aunque es un lenguaje de programación pensado para su utilización como lenguaje orientado a objetos, también es cierto que con el lenguaje de programación C++ se puede escribir código sin conocer dicho paradigma de programación, lo cual no es posible con el lenguaje de programación Java, donde no cabe obviar la orientación a objetos para el desarrollo de programas. Al contrario que en el lenguaje de programación C++, en el lenguaje de programación Java nada se puede hacer nada sin usar al menos un objeto.

[17] Otra visión del ejemplo presentado podría incluir la utilización del concepto de herencia siendo `ser humano` la superclase o clase base y `farmacéutica`, `hijo` y `madre` las clases derivadas. Los conceptos de herencia, superclase y clase derivada se analizan más adelante en la presente documentación. Véase la sección 4.6.1 Herencia.

## 4.2 ¿ QUÉ ES UN OBJETO ?

Los **objetos software** se utilizan para modelar objetos o entidades del mundo real, por tanto, un objeto software, al igual que un objeto o entidad del mundo real, tendrá estado y comportamiento. El estado se almacena en variables que se denominan atributos. Y el comportamiento se representa mediante funciones que se denominan métodos, es decir, que un método es un procedimiento o una función perteneciente a un objeto. [ver Figura 1]

Por tanto, un objeto software es un conjunto de **atributos** (o **variables**) y **métodos** (o **funciones**) relacionados entre sí, o dicho de otra forma, es la representación en un programa de un concepto, y contiene toda la información necesaria para abstraerlo: datos que describen sus características y operaciones que pueden realizarse sobre los mismos. Como un objeto está formado por una serie de datos (atributos) y una serie de operaciones (métodos), no puede concebirse únicamente en función de los datos o de las operaciones sino en su conjunto.

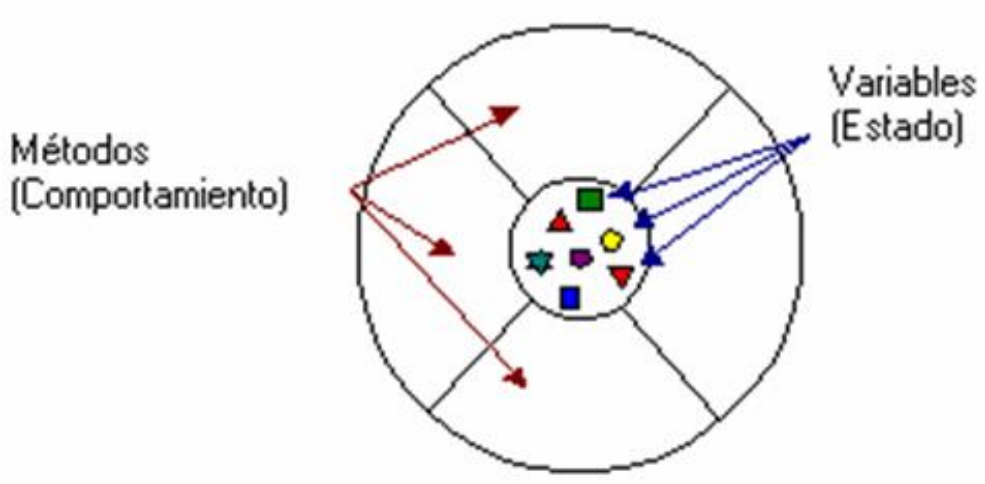



Figura 1: Representación visual del concepto de objeto.

Los características del objeto (**estado**) y lo que el objeto puede hacer (**comportamiento**) están expresados por los atributos, almacenados mediante variables, y los métodos que componen el objeto, respectivamente. Por ejemplo, un objeto que modelase una bicicleta en el mundo real utilizaría variables para indicar el estado actual de la bicicleta, por ejemplo, que su `velocidad` es de `20 km/h`, que su `cadencia de pedaleo` es de `90 r.p.m.`[18] y que su `marcha` actual es la `5ª`. Estos atributos, o variables, se conocen formalmente como **atributos instancia** o **atributos miembro** porque contienen el estado de un objeto bicicleta particular y, en Programación Orientada a Objetos, un objeto particular se denomina una instancia. Además de estas variables, el objeto bicicleta podría tener métodos para `cambiar la cadencia de pedaleo`, `cambiar la marcha`, `cambia el piñón` o `cambiar el plato`. La bicicleta no tendría por qué tener un método para cambiar su velocidad pues ésta es función de la cadencia de pedaleo, la marcha en la que está y de si los frenos están siendo utilizados o no, entre otros muchos factores. Estos métodos se denominan formalmente **métodos instancia** o **métodos miembro**, ya que cambian el estado de una instancia u objeto bicicleta particular. Una bicicleta podría estar caracterizada también por el `piñón` y el `plato` que estuviera seleccionado, su `color`, su `peso` etc., además una bicicleta `frena` y `acelera`, etc. Otros ejemplos de objetos del mundo real, además de la bicicleta, serían, por ejemplo, el perro o la televisión. Cada uno de ellos tiene estado y comportamiento. Y, así, por ejemplo, en el caso del perro, de

[18] `r.p.m.` hace referencia a revoluciones por minuto.

forma análoga al caso de la bicicleta, podría decirse que entre sus características se cuentan el `color`, la `edad`, la `raza`, el `nombre`, etc., y que el perro `ladra`, `corre`, `mueve la cola`, etc.

Por tanto, en términos ajenos a la programación, un objeto es una persona, animal o cosa que se distingue de otros objetos tanto por sus características como por su utilidad. Y en términos de programación, será misión del programador determinar qué características y qué operaciones interesa mantener sobre cada objeto. Así, por ejemplo, para el objeto casa, sus características podrían ser `número de pisos`, `altura total en metros`, `color de la fachada`, `número de ventanas`, `ubicación`, etc., y las operaciones que se pueden realizar sobre dicho objeto `construir`, `destruir`, `pintar fachada`, modificar alguna de las características, como, por ejemplo, `abrir una nueva ventana`, etc. Sin embargo, sobre el objeto casa podría no ser necesario conocer su ubicación y, por lo tanto, dicha característica no sería necesario que formara parte del objeto software definido por el programador. Un razonamiento análogo podría realizarse en lo relativo a las operaciones.

La Figura 2 muestra cómo podría modelarse una bicicleta como un objeto. El diagrama del objeto bicicleta muestra sus atributos, que conforman su estado, en el núcleo o centro del objeto y los métodos rodeando al núcleo y protegiéndolo de otros objetos del programa.

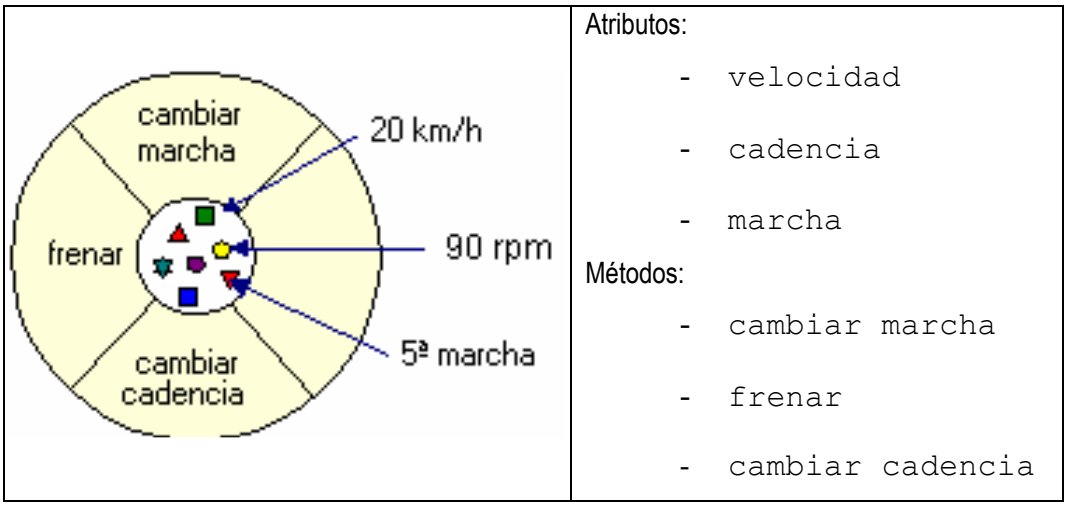



Figura 2: Bicicleta modelada como un objeto.

## 4.3 ¿ QUÉ ES UN MENSAJE ?

Normalmente un único objeto por sí solo no es muy útil, sino que, en general, un objeto aparece como un componente más de un programa que contiene otros muchos objetos. Y es precisamente haciendo uso de la interacción entre objetos como se consigue una funcionalidad de mayor orden, así como modelar comportamientos más complejos. Así, por ejemplo, una bicicleta no es más que una estructura de aleación de titanio y un poco de goma que, por sí sola, es incapaz de desarrollar ninguna actividad. Sin embargo, una bicicleta es realmente útil en tanto que otro objeto, como por ejemplo un ciclista, interactúa con ella pedaleando.

Los objetos de un programa interactúan y se comunican entre ellos por medio de mensajes. Cuando un objeto `A` quiere que otro objeto `B` ejecute una de sus funciones o métodos miembro, el objeto

`A` manda un mensaje al objeto `B` tal y como puede observarse en la Figura 3. Al enviar un mensaje es posible esperar a que el objeto invocado responda o bien continuar sin esperar.

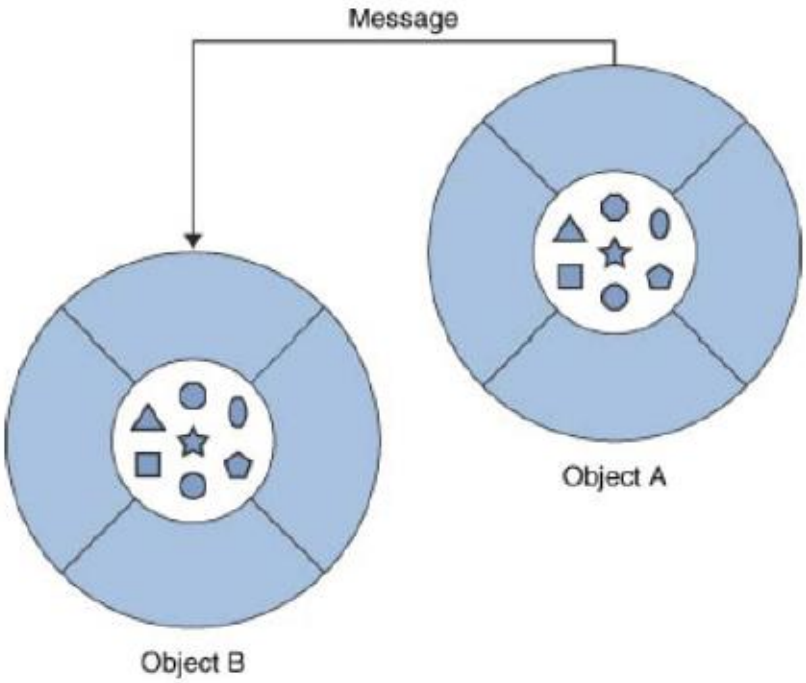



Figura 3: Comunicación entre objetos por medio de mensajes (I).

En ocasiones, el objeto que recibe el mensaje necesita más información para saber exactamente lo que debe hacer; por ejemplo, cuando se desea cambiar la marcha de una bicicleta, se debe indicar la marcha a la que se quiere cambiar. Esta información se pasa junto con el mensaje en forma de parámetro. En la Figura 4 se muestra un ejemplo en el que se envía un mensaje al objeto `TuBicicleta` indicándole que debe ejecutar el método o función `CambiarDeMarcha()` que hace uso del parámetro `Marcha`. El método y sus parámetros es la información que necesita el objeto destinatario que recibe el mensaje para ejecutar adecuadamente la función miembro solicitada.

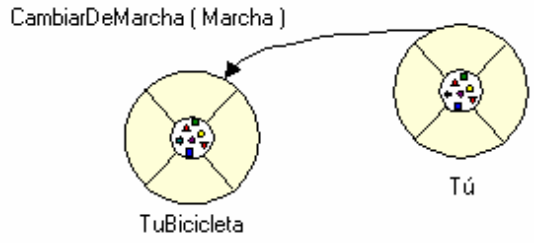
| | |
|---|---|
|   | El Objeto `Tú` manda el mensaje `CambiarDeMarcha` que hace uso del parámetro `Marcha` al Objeto `TuBicicleta`. |

Figura 4: Comunicación entre objetos por medio de mensajes (II).

## 4.4 ¿ QUÉ ES UNA CLASE ?

Normalmente en el mundo real existen varios objetos de un mismo tipo, o en la terminología empleada en el paradigma de orientación a objetos, de una misma clase. Por ejemplo, `miBicicleta` es una de las muchas bicicletas que existen. Usando la terminología del paradigma de orientación a objetos, se dirá que `miBicicleta` es una instancia de la clase de objetos `Bicicletas`, es decir, que `miBicicleta` (objeto) es una instancia de `Bicicletas` (clase). Todas las bicicletas tienen una serie de atributos: `color`, `marcha actual`, `cadencia actual`, dos `ruedas`, etc., y de métodos: `cambiar de marcha`, `frenar`, etc. Sin embargo, el estado particular de cada bicicleta es independiente del estado de las demás bicicletas ya que el valor de esos atributos puede variar de unos objetos a otros. Es decir, una bicicleta podrá ser azul, y otra roja, pero ambas tienen en común el hecho de tener el atributo `color`. De este modo es posible definir una plantilla de atributos y métodos para todas las bicicletas.

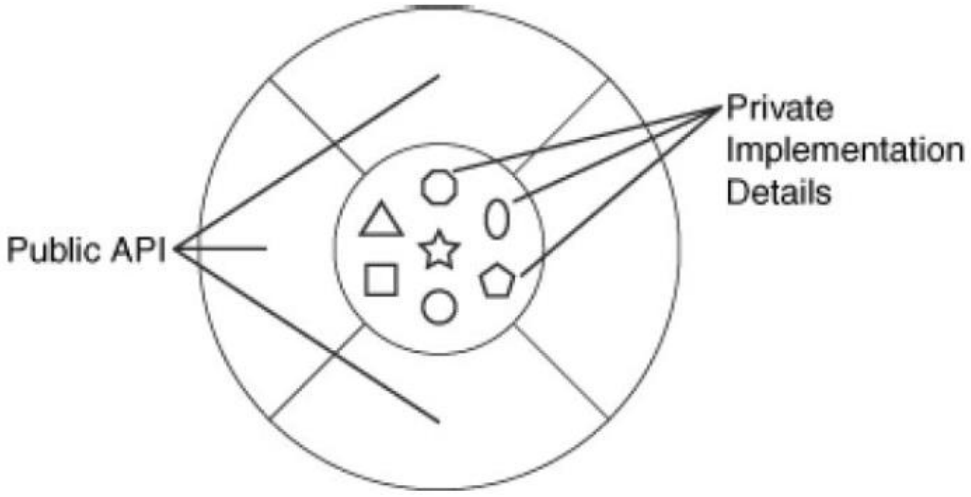



Figura 5: Representación visual del concepto de clase.

Estas plantillas o moldes para la creación de objetos se denominan clases, por tanto, una clase podría definirse como una plantilla o molde que define los atributos (datos) y los métodos (funciones) que son comunes para todos los objetos de un cierto tipo y que, por tanto, permitiría crear objetos de ese cierto tipo. Una clase también puede verse como una abstracción de aquello que tienen en común todos los objetos de un determinado tipo. [ver Figura 5]

En el ejemplo con el que se está trabajando, la clase `Bicicletas` definiría atributos miembro comunes a todas las bicicletas, como la `marcha` actual, la `cadencia` actual, etc. Esta clase también debe declarar e implementar los métodos miembro que permitan al ciclista `cambiar de marcha`, `frenar` o `cambiar la cadencia de pedaleo`, tal y como se muestra en la Figura 6.

Después de haber creado la clase `Bicicletas`, es posible crear cualquier número de objetos o instancias a partir de la clase. [ver Figura 7] Cuando se declara un objeto, por tanto, hay que definir el tipo de objeto o clase al que pertenece. Un ejemplo de la declaración de objetos en el lenguaje

de programación Java sería la siguiente sentencia en la que se definen dos objetos `bicicleta1` y `bicicleta2` de la clase `Bicicletas`:

```
Bicicletas bicicleta1, bicicleta2;
```

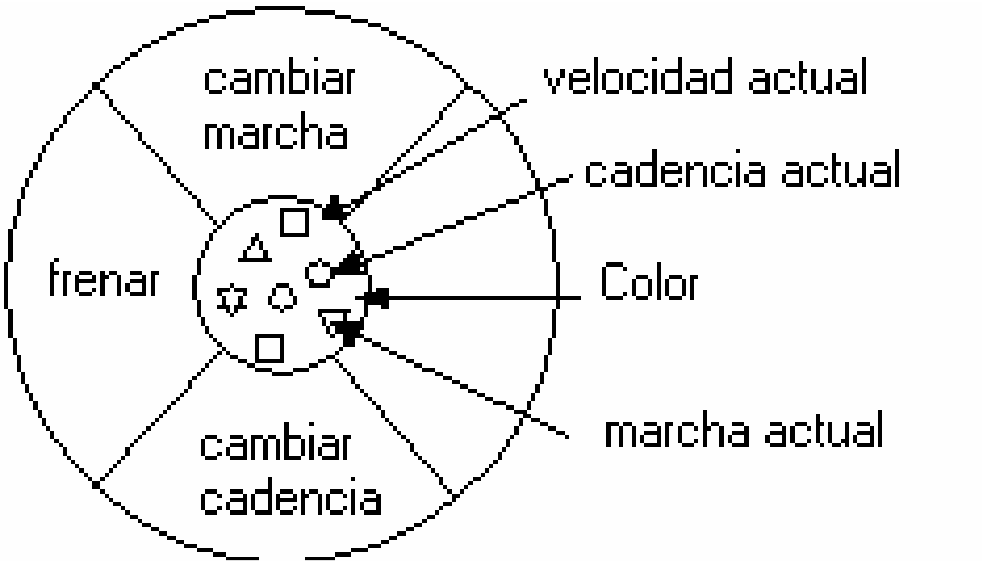



Figura 6: Representación visual de la clase `Bicicletas`.

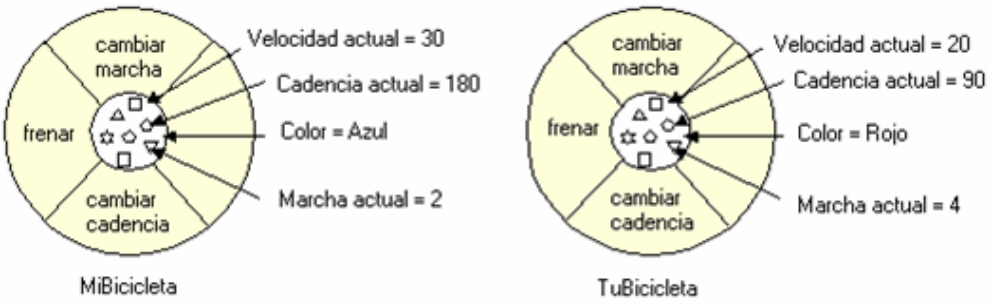



Figura 7: Objetos o instancias de la clase `Bicicletas`.

Cuando se crea una instancia de una clase, el sistema reserva suficiente memoria[19] para el objeto con todos sus atributos miembro, ya que cada instancia tiene su propia copia de las variables miembro definidas en la clase.

[19] Para instanciar un objeto de una determinada clase es también necesario llamar a su constructor después de la palabra reservada `new` y es en este instante en el que se reserva espacio en memoria para el nuevo objeto. El código necesario en el lenguaje de programación Java sería, por tanto, el siguiente:

```
Biclicletas bicicleta1;
biclibleta1 = new Bicicletas();
```

o

```
Biclicletas bicicleta1 = new Bicicletas();
```

Es importante señalar que existen ciertos atributos denominados **atributos de clase** de los que solo existirá una copia compartida por todas las instancias que se creen de la clase. Para que esto tenga sentido, debe verificarse que todos los objetos de esa clase compartan siempre el mismo valor para dicho un atributo, ya que, con un atributo de este tipo, si se modificara su valor, todas las instancias verían el nuevo valor puesto que, tal y como se ha comentado, solo existe una copia de dicho atributo que es compartida por todas las instancias de la clase. Así, un ejemplo de un atributo de clase podría ser `númeroRuedas` cuyo valor podría ser `2` para todos los objetos de la clase `Bicliceletas`. De forma análoga, se puede hablar de **métodos de clase**, como métodos vinculados a una clase, y no a una instancia particular de dicha clase. Dado que los métodos de clase tienen sentido a nivel de clase y no a nivel de objeto o instancia, dichos métodos deberán manipular atributos de clase, pero no atributos miembro vinculados a las instancias u objetos particulares.

## 4.5 ¿ QUÉ ES UNA INTERFAZ ?

El concepto de interfaz en el contexto de la Programación Orientada a Objetos no tiene nada que ver con el concepto de Interfaz de Usuario (Gráfica) ((G)UI, (Graphical) User Interface) que es como se denomina la parte que se muestra al usuario de una determinada aplicación. En el contexto de la Programación Orientada a Objetos, una interfaz puede definirse, en términos generales, como la forma en la que se ve a una clase desde el exterior. No se puede ver el estado directamente, sí se ve la declaración de una serie de métodos, aunque no la implementación de los mismos. Es decir, que una interfaz sería una clase especial en la que solamente se declaran los métodos que una clase que la implemente debe codificar. En este sentido se puede decir que las interfaces representan un **contrato**, de forma que cualquier clase que implemente una determinada interfaz debe codificar los métodos de dicha interfaz, respetando la forma en que estos se han declarado, lo cual implica, un mismo número, orden y tipo de argumentos y/o mismo tipo de dato devuelto.

Gracias a la implementación de interfaces es posible crear relaciones entre clases que implementen la misma interfaz o interfaces, ya que dichas clases podrán tener métodos comunes, al menos en la forma, aunque no necesariamente en el fondo. Así, por ejemplo, el método `guardar` se podría definir en una interfaz, y las clases que quisieran implementar un método `guardar` *estandarizado* firmarían un contrato con la interfaz que lo especifica. La forma interna de funcionamiento sería responsabilidad del programador de cada clase. No obstante, sí estaría garantizado el hecho de que cualquier clase que haya firmado un contrato con esa interfaz tendría que seguir las condiciones impuestas por dicha interfaz, lo cual asegura que todas las clases que implementen dicha interfaz tendrían un método `guardar` compatible, por tanto, el cómo se realice esa acción de guardar no debería ser motivo de preocupación, ya que simplemente sería necesario saber que dicha acción se implementaría adecuadamente para almacenar los datos que cada clase manipule. Puede decirse, que las interfaces permiten al programador, definir un conjunto de funcionalidades sin saber cómo se implementarán las mismas.

En el caso concreto del lenguaje de programación Java, una interfaz es una clase completamente abstracta[20] en la que solo se declaran métodos, pero sin implementarlos, y tampoco se

---

Un constructor es un método especial de las clases que permite instanciar, e inicializar los objetos que se instancian, como miembros de la clase en cuestión. Para declarar un constructor basta con declarar un método con el mismo nombre que la clase. No obstante, todas las clases tienen un constructor por defecto que no es necesario declarar, aunque es posible redefinirlo e incluso declarar diferentes constructores, de hecho, esta es una de las aplicaciones más habituales de la sobrecarga de métodos. Véase el concepto de sobrecarga de métodos en sección 4.6.3 Polimorfismo.

[20] Véase más información sobre el concepto de Clases abstractas en la sección 4.6.2 Abstracción.

definen atributos, aunque sí es posible definir alguna constante[21]. Es decir, que una interfaz ofrece una declaración de una serie de métodos, y se dice que una clase implementa una determinada interfaz si ofrece implementaciones para *todos* los métodos de dicha interfaz, ya que la interfaz es un contrato entre la clase y el mundo exterior que debe cumplirse en el momento de la compilación, de forma que en dicho instante, si la clase implementa una interfaz, todos los métodos definidos por la interfaz deben aparecer en el código fuente de la clase para que la compilación de la misma se lleve a cabo con éxito.

Como en el lenguaje de programación Java no está permitida la herencia múltiple, es decir, de más de una superclase[22], una aproximación a este concepto es la utilización de una sola superclase, pero de una o varias interfaces, por ejemplo:

```
class MiClase extends SuPadre[23] implements Interfaz1,Interfaz2{…}
```

Como una interfaz en el lenguaje de programación Java es un conjunto de métodos abstractos[24] y, quizás, de constantes, cuando una clase declara una lista de interfaces mediante la cláusula `implements`, hereda todas las constantes que pudieran haber sido definidas en la interfaz y se compromete a definir todos los métodos de la interfaz ya que estos métodos son abstractos por lo que no tienen implementación o cuerpo. Mediante el uso de interfaces en el lenguaje de programación Java se puede lograr, como ya es sabido, que distintas clases implementen métodos con el mismo nombre[25], todas aquellas que así lo manifiesten mediante la cláusula `implements` en su declaración, ya que, de esta forma, se comprometen a codificar los diferentes métodos de dicha interfaz.

## 4.6 PROPIEDADES DE LA ORIENTACIÓN A OBJETOS

Para que un lenguaje de programación sea considerado Orientado a Objetos debe tener diversas propiedades entre las que destacan las que a continuación se describen.

### 4.6.1 Herencia

Una de las características básicas de la Programación Orientada a Objetos es el uso de la herencia que es un mecanismo que permite definir nuevas clases partiendo de otras ya existentes. Las clases que derivan de otras heredan automáticamente todos sus elementos, pero además pueden introducir elementos particulares propios que las diferencien.

Tal y como se ha explicado, los objetos se definen a partir de clases. Por el simple hecho de conocer a qué clase pertenece un objeto, ya se conoce bastante información sobre él. Puede que no se sepa lo que es la `Espada`, pero si nos informan de que es una bicicleta, ya se conocerá que tiene ruedas, manillar, pedales, etc. La Programación Orientada a Objetos va más allá, permitiendo definir

[21] Las constantes en el lenguaje de programación Java se declaran como atributos de tipo `final` y `static`. La palabra reservada `final` calificando a un atributo o variable sirve para declarar constantes, no permitiéndose la modificación de su valor. Si además se califica como `static`, se puede acceder a dicha constante simplemente anteponiendo el nombre de la clase, sin necesidad de instanciarla creando un objeto de la misma, ya que dicha palabra reservada indica que se trata de un atributo de clase y no de un atributo instancia o miembro. El valor de un atributo `final` deberá ser asignado en la declaración del mismo y cualquier intento de modificar su valor generará el consiguiente error por parte del compilador.

[22] Véase más información sobre el concepto de Superclase en la sección 4.6.1 Herencia.

[23] La clase derivada `MiClase` solo hereda de la clase base `SuPadre`, y no podría heredar de varias clases simultáneamente.

[24] Véase más información sobre el concepto de Métodos abstractos en la sección 4.6.2 Abstracción.

[25] En realidad, con la misma declaración.

clases a partir de otras clases ya construidas. Por ejemplo, las bicicletas de montaña, las de carretera y los tándems son todos diferentes tipos de bicicletas. En términos de Programación Orientada a Objetos, son **subclases** o **clases derivadas** de la clase `Bicicletas`. Análogamente, la clase `Bicicletas` es la **superclase** o **clase base** de las bicicletas de montaña, las de carretera y los tándems. Esta relación de herencia entre las diferentes clases (superclase y subclases) se muestra en la Figura 8.

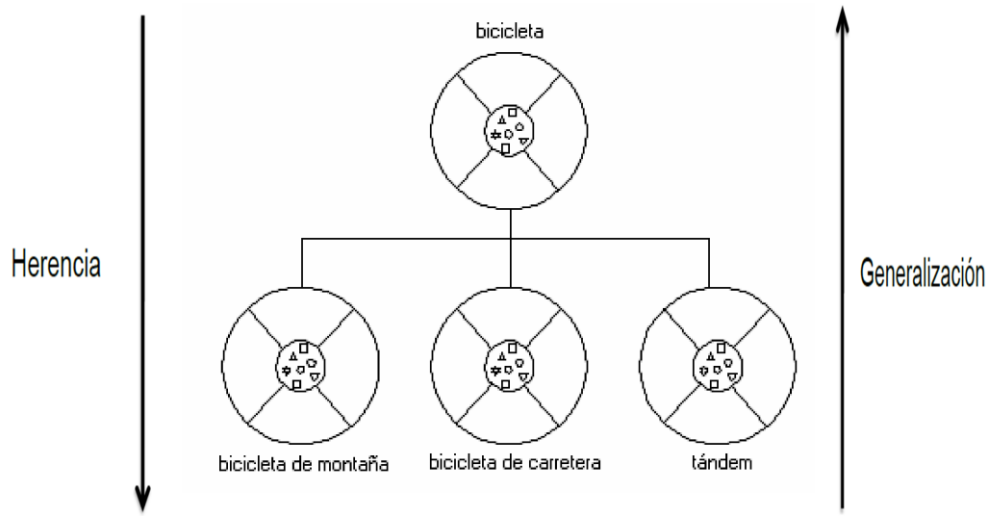



Figura 8: Representación visual del concepto de herencia.

Cada subclase hereda el estado, en forma de declaración de atributos, de la superclase de la cual deriva. Las bicicletas de montaña, las de carretera y los tándems comparten diversas características que conforman su estado: `cadencia`, `velocidad`, etc. Además, cada subclase hereda el comportamiento, en forma de declaración de métodos, de su superclase, así las bicicletas de montaña, las de carretera y los tándems comparten algunos procedimientos como, por ejemplo, `frenar` o `cambiar la cadencia de pedaleo`. Sin embargo, las clases derivadas no se encuentran limitadas por los estados y comportamientos que heredan de su superclase, al contrario, estas subclases pueden añadir atributos o variables y métodos o funciones a aquellos que han heredado. Los tándems tienen dos asientos y dos manillares, algunas bicicletas de montaña tienen una catalina adicional con un conjunto de marchas con relaciones de transmisión mucho más cortas.

Por tanto, como toda subclase hereda la definición de estado y comportamiento de la superclase, si la clase `Bicicletas` tiene los siguientes atributos y métodos: `piñónSeleccionado`, `platoSeleccionado`, `color`, `frenar()`, `acelerar()`, `cambiarPiñón()` y `cambiarPlato()`, las bicicletas de montaña, de carreras y tándem, subclases de la clase `Biblicletas`, tendrán todos esos atributos y métodos, pero también pueden tener más atributos o métodos adicionales.

La herencia es una forma natural de definir objetos en la vida real, así, por ejemplo, la mayoría de personas dirían que un chalet es una *casa con jardín*. Por tanto, un chalet tiene las mismas características que una casa y sobre un chalet se pueden realizar las mismas operaciones que se pueden realizar sobre una casa, solo que además el chalet incorpora una nueva característica que es el jardín. En otras ocasiones, se añadirá funcionalidad y no características, así, por ejemplo, un pato es un *ave que*

*nada*, así el pato mantiene las mismas características y el mismo comportamiento que las aves, pero además puede realizar una nueva función que es la de nadar.

Tal y como se ha comentado, la herencia permite definir clases derivadas de otras, de forma que la nueva clase derivada hereda de su clase base todos sus atributos y métodos. Y, además, la nueva clase puede definir nuevos atributos y métodos o, incluso, puede redefinir atributos y métodos ya existentes, por ejemplo, cambiar el tipo de dato de un atributo o las operaciones que realiza un determinado método. Es decir que, cuando existe herencia puede haber **sobreimplementación de métodos**[26], esto significa que una subclase puede implementar de nuevo un método especificado en la superclase, definiendo de forma distinta su cuerpo, es decir, las acciones a llevar a cabo. Así, por ejemplo, la superclase `Bicicletas` puede tener definido el método `frenar()`.

```
class Bicicletas {
        double velocidad;
        [...]
        void[27] frenar() {
                velocidad = velocidad*0.9;
        }
}
```

Y la subclase `BicicletaCarreras` puede volver a implementarlo si resultara necesario que este tipo de bicicletas frene de una manera particular:

```
class BicicletaCarreras extends Bicicletas {
        [...]
        void frenar() {
                velocidad = velocidad*0.7;
        }
}
```

Es decir que, como se ha dicho, las clases derivadas pueden sobrescribir los métodos heredados y proporcionar implementaciones más especializadas para esos métodos.

Además, la herencia no se limita a un único nivel. El árbol de herencias o jerarquía de clases puede ser tan extenso como se necesite. Los métodos o funciones y los atributos o variables miembro se heredarán hacia abajo a través de todos los niveles de la jerarquía. Normalmente, cuanto más abajo está una clase en la jerarquía de clases, más especializado será su comportamiento. En el ejemplo propuesto, se podría hacer que la clase `Bicicletas` derivase de una superclase `Vehículos`.

Cuando se trabaja con objetos se dice que existe herencia si se puede afirmar que los objetos de un tipo son siempre objetos de un tipo más general. Es decir, existe una relación de herencia-generalización si toda instancia de `B` es instancia de `A`, o dicho de otro modo, la clase `B` deriva o hereda

[26] En los textos ingleses, para hablar de la sobreimplementación de métodos se utiliza el término `override`.

[27] El modificador `void` indica que el método `frenar()` no devuelve ningún valor.

de la clase base `A` o bien la clase `B` es subclase de la clase base `A` y la clase `A` es superclase de la clase derivada `B`.

La herencia es una herramienta clave para abordar la resolución de un problema de forma organizada, pues permite definir una relación jerárquica entre todos los conceptos que se están manejando[28]. Es posible emplear esta técnica para descomponer un problema de cierta magnitud en un conjunto de problemas subordinados a él. La resolución del problema original se consigue cuando se han resuelto cada uno de los problemas subordinados, que a su vez pueden contener otros. Esto es así porque el descomponer un problema o concepto en un conjunto de objetos relacionados entre sí cuyo comportamiento es fácilmente identificable puede ser extraordinariamente útil para dar solución a dicho problema.

En definitiva, puede decirse que la herencia permite que las clases derivadas o subclases proporcionen comportamientos y estados especializados a partir de los elementos comunes que heredan de la clase base o superclase, lo cual, además, facilitará la reutilización del código de clases ya existentes, modificándolas en tanto en cuanto sea necesario para adaptarlas a las nuevas especificaciones. Así, por ejemplo, si se tiene construida la clase `Casas` y se desea definir la nueva clase `Chalets` puede ser conveniente definir un nuevo atributo que represente los `metros cuadrados de jardín`. En esta situación, en lugar de volver a definir la nueva clase `Chalets` desde cero, puede partirse de la clase `Casas`, realizando la codificación de la siguiente forma:

```
class Chalets extends Casas {
        int mJardin;
        public[29] Chalets(int np[30], int nv[31], String co[32], int mj[33]) {
                super(np,nv,co);[34]
                mJardin=mj;
        }
}
[…]
Chalets chalet1;
chalet1 = new Chalets();
[…]
```

Tal y como puede observarse, basta con declarar que la nueva clase `Chalets` deriva de la clase base `Casas`, lo cual se indica mediante la cláusula `extends Casas`, y declarar el nuevo atributo `mJardin`. También ha sido posible definir el método constructor para poder inicializar el nuevo

[28] Nótese no obstante que la herencia supone un cierto Acoplamiento entre las clases involucradas en dicha relación, concepto que se abordará con mayor detalle al presentarse la teoría de Patrones de Diseño para la definición de objetos software.

[29] El modificador `public` indica que el método constructor `Chalets()` es público y que, por tanto, es accesible o visible desde cualquier clase.

[30] `np` se refiere al número de puertas.

[31] `nv` se refiere al número de ventanas.

[32] `co` se refiere al color.

[33] `mj` se refiere a los metros de jardín.

[34] Cuando existen herencia, especificada mediante la cláusula `extends`, `super()` realiza una llamada al constructor de la clase base o superclase de la que se hereda.

atributo `mJardin`. Pero el resto de los métodos, por ejemplo para `abrir o cerrar puertas o ventanas`, no sería necesario redefinirlos, puesto que se heredan de la clase `Casas` y podrían utilizarse, por ejemplo como en:

```
chalet1.pintar("blanco");
```

donde `chalet1` es un objeto o instancia de la clase `Chalets` creado previamente y `pintar("color")` se supone que es uno de los métodos definidos en la superclase `Casas`.

En definitiva, las clases no están aisladas, sino que se relacionan entre sí formando una jerarquía. Los objetos toman el estado y el comportamiento de la clase a la que pertenecen y además heredan el estado y comportamiento de las clases de las que deriven. La herencia permite que los objetos sean definidos y creados como tipos especializados de objetos preexistentes los cuales pueden compartir y extender su estado y comportamiento sin tener que volver a implementarlo.

### 4.6.2 Abstracción

También es posible implementar clases abstractas que definan comportamientos genéricos. Las **clases abstractas** definen e implementan comportamientos, pero no de una forma completa, de manera que otros programadores pueden hacer uso de estas clases detallando esos comportamientos con subclases especializadas. El propósito de una clase abstracta es servir de modelo base para la creación de otras clases derivadas, pero cuya implementación depende de las características particulares de cada una de ellas.

Un ejemplo de clase abstracta podría ser la clase `Vehículos` que sería una clase base genérica a partir de la cual podrían crearse diferentes clases derivadas como `Bicicletas`, `Coches`, `Barcos`, etc. Otro ejemplo podría ser la clase `Animales` que sería una clase base genérica a partir de la cual podrían crearse diferentes clases derivadas como `Mamíferos`, `Aves`, `Peces`, etc., así como las clases `Perros`, `Patos`, etc., que serían a su vez clases derivadas de las anteriores.

Las clases abstractas son clases que no pueden tener instancias que no lo sean también de una de sus clases derivadas o subclases, o dicho de otra forma, aquellas clases que ofrecen estado y comportamiento únicamente para ser heredado. Por tanto, una clase abstracta no se puede instanciar ya que solo se usa para definir subclases. Además, toda clase abstracta tendrá algún **método abstracto**, lo que significa que no estaría implementado, obligando esto a las subclases a ofrecer una implementación para el mismo, así, por ejemplo, todos los animales comen, pero cada uno come de manera distinta.

En la Figura 9 se presenta un ejemplo según el cual toda instancia de la clase `Figura` tiene como característica su posición, especificada mediante las coordenadas `X` e `Y`, a partir de la cual se puede calcular su área. No obstante, no se calcula el área de las instancias de la clase `Figura`, sino únicamente el área de las instancias de las clases derivadas `Círculo` o `Cuadrado`, ya que el método `área()` es un método abstracto que únicamente implementan las subclases `Círculo` y `Cuadrado`, pero no la superclase `Figura`.

La abstracción es una de las principales características de la orientación a objetos. Esta característica permite que un objeto en el sistema sirva como modelo de un agente abstracto que puede realizar acciones, informar sobre y cambiar su estado, y comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características. Además, los métodos, funciones o procesos, tal y como se acaba de explicar, también pueden ser abstraídos.

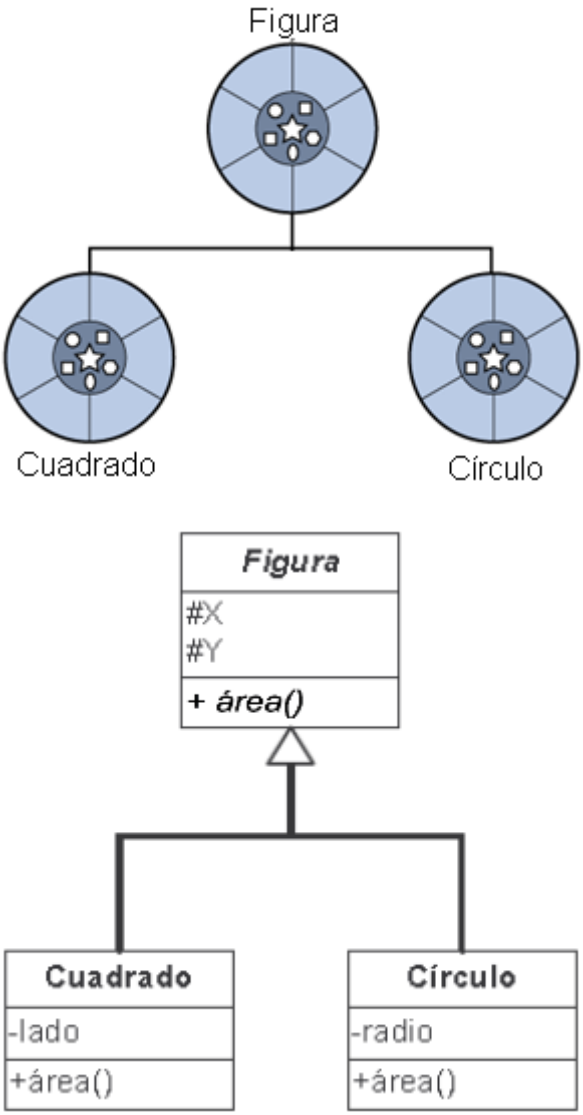



Figura 9: Representación visual de un ejemplo del concepto de abstracción.[35]

[35] Nótese que el nombre de la clase `Figura` se muestra en cursiva en el Diagrama de Clases por tratarse de una clase abstracta e, igualmente, el nombre de su método `área()` también aparece en cursiva por tratarse de un método abstracto.

### 4.6.3 Polimorfismo[36]

El Polimorfismo se refiere al hecho de que comportamientos diferentes, asociados a objetos distintos, compartan el mismo nombre, la misma declaración en realidad. Al hacer uso de dicho nombre se invocará el comportamiento correspondiente al objeto que se esté usando.

Esta propiedad permite que un mismo mensaje enviado a objetos de clases distintas haga que estos se comporten también de forma distinta[37]. En el Polimorfismo, las declaraciones de las funciones polimórficas en las diferentes clases deben coincidir en cuanto a número y tipo de los parámetros y

[36] También es posible que una clase tenga más de un método con el mismo nombre, pero diferencias en algún detalle de su declaración como por ejemplo, en su número y/o tipo de parámetros pasados. Esto se denomina **Sobrecarga** y no debe confundirse con el Polimorfismo. Así sería posible tener varios métodos `multiplicar()` pero el compilador sabe cuál utilizar porque cada uno podría tener definidos parámetros de distinto tipo o un número diferente de parámetros, o ambos. A continuación, se presentan dos ejemplos:

```
class Articulo {
        private[36] float precio;
        public void setPrecio()[36] { precio = 3.50; }
        public void setPrecio(float nuevoPrecio) { precio = nuevoPrecio; }
        public void setPrecio(float coste, int porcentajeGanancia) { precio = coste + (coste * porcentajeGanancia); }
}
```

En el ejemplo anterior, el método sobrecargado se denomina `setPrecio()` y se observa como en sus diferentes implementaciones tiene diferente número y/o tipo de parámetros.

```
class Casas {
        […]
        public Casas(int np, int nv, String co) {
                nPuertas=np;
                nVentanas=nv;
                color=co;
        }
        public Casas() {
                nPuertas=2;
                nVentanas=5;
                color="blanco";
        }
```

En el ejemplo anterior, el método sobrecargado se denomina `Casas()` y es el constructor de la clase de igual nombre. En el primer caso los atributos del objeto o instancia creado se inicializarán con los parámetros pasados al método, mientras que en el segundo caso se inicializarán utilizando unos valores fijos.

[37] Esto se observará de forma clara cuando se estudien los Diagramas de Interacción propuestos por el Lenguaje de Modelado Unificado.

también, salvo en casos excepcionales, en el valor devuelto y su tipo. Cuando existen versiones diferentes (polimórficas) de un método, es la naturaleza del objeto que realiza la invocación lo que permite al compilador conocer la versión del método que debe utilizar.

El Polimorfismo permite tomar como algo igual a un nivel superior dos cosas que son diferentes a un nivel más bajo. Por ejemplo, gato y perro son ambos animales a un nivel alto de abstracción, pero diferentes a un nivel inferior. El Polimorfismo permite olvidarse de ciertas características específicas de varios objetos para centrarse en lo que tengan en común, es decir, en lo que los haga iguales. El lenguaje de programación Java ofrece esta posibilidad gracias a las clases y métodos abstractos[38] y a las Interfaces[39]. Las clases abstractas tienen cierta similitud con las Interfaces, de hecho, las Interfaces son clases abstractas, en realidad clases *completamente* abstractas. Las clases abstractas pueden ofrecer una implementación parcial (atributos y métodos) que luego heredarán sus subclases, dejando a las subclases la posibilidad de completar la implementación. No obstante, si una clase abstracta solo contiene declaraciones de métodos abstractos, y, opcionalmente, constantes, entonces debería declararse como una Interfaz y no como una clase abstracta. Es decir, una clase abstracta puede definir más concretamente la forma de sus futuras subclases, mientras que una Interfaz nunca puede definir nada acerca de la forma de las clases que la implementen, solo los servicios que éstas deberán implementar, y no puede definir atributos, solo constantes[40]. En definitiva, una Interfaz es un conjunto de constantes y métodos abstractos, mientras que una clase abstracta puede ofrecer una implementación parcial tanto en cuanto a atributos como a métodos.

Como ejemplo de Polimorfismo se puede suponer que se tienen dos objetos `piso1` y `chalet1` de las clases `Pisos` y `Chalets` respectivamente, ambas clases derivadas de la superclase abstracta `Casas` que tiene definido un método abstracto `pintar()`. Si se llama al método `chalet1.pintar()`, en este caso se ejecutará el código del procedimiento `pintar()` de la clase `Chalets` y no el de la clase `Pisos`. Igualmente, el método `área()` que tendría una implementación diferente para la clase `Cuadrado` y para la clase `Círculo` respectivamente [ver Figura 9] sería otro ejemplo de operación polimórfica.

### 4.6.4 Encapsulamiento

El Encapsulamiento o Encapsulado consiste en la propiedad que tienen los objetos de ocultar sus atributos, e incluso sus métodos, a otros objetos del programa. La forma natural de construir una clase es la de definir una serie de atributos que, en general, no serán accesibles fuera del propio objeto, sino que únicamente podrán manipularse a través de los métodos que sean definidos como accesibles desde el exterior de esa clase.

```
class Casas {
        private int nPuertas, nVentanas;
        private String color;
        public Casas(int np, int nv, String co) {
```

[38] Véase más información sobre el concepto de clases y métodos abstractos en la sección 4.6.2 Abstracción.

[39] Véase más información en la sección 4.5 ¿ Qué es una Interfaz ?

[40] Las constantes en el lenguaje de programación Java se declaran como atributos de tipo `final` y `static`. La palabra reservada `final` calificando a un atributo o variable sirve para declarar constantes, no permitiéndose la modificación de su valor. Si además se califica como `static`, se puede acceder a dicha constante simplemente anteponiendo el nombre de la clase, sin necesidad de instanciarla creando un objeto de la misma, ya que dicha palabra reservada indica que se trata de un atributo de clase y no de un atributo instancia o miembro. El valor de un atributo `final` deberá ser asignado en la declaración del mismo y cualquier intento de modificar su valor generará el consiguiente error por parte del compilador.

```
                nPuertas=np;
                nVentanas=nv;
                color=co;
        }
        public void pintar(String co) {
                color=co;
        }
        public void abrirVentanas(int n) {
                nVentanas=nVentanas+n;
        }
        public void cerrarVentanas(int n) {
                nVentanas=nVentanas-n;
                if (nVentanas<0)
                        nVentanas=0;
        }
        public void abrirPuertas(int n) {
                nPuertas=nPuertas+n;
        }
        public void cerrarPuertas(int n) {
                nPuertas=nPuertas-n;
                if (nPuertas<0)
                        nPuertas=0;
        }
}
[…]
Casas casa1,casa2;
casa1 = new Casas();
casa2 = new Casas();
[…]
```

Así, en el código anterior, al declarar la clase `Casas` se definen una serie de atributos que únicamente son accesibles a través de determinados métodos. Así, si se quiere abrir una nueva ventana en la casa `casa1`, la opción tradicional consistiría en utilizar `casa1.nVentanas = casa1.nVentanas + 1`, sin embargo, la forma natural de hacerlo en la Programación Orientada a Objetos sería llamando al método `abrirVentanas()`, utilizando para ello `casa1.abrirVentanas(1)`, ya que será ese método, pasándole como parámetro el valor `1`, el que se encargue de incrementar en una unidad el valor del atributo `nVentanas`. Esto no quiere decir que el atributo `nVentanas` no pueda ser accedido de la forma tradicional si se hubiera definido como `public`, pero, para que el lenguaje pueda ser considerado como orientado a objetos, debe

permitir la posibilidad de prohibir el acceso directo a los atributos, lo cual, por otra parte, será lo habitual para evitar manipulaciones indebidas, intencionadas o no, del estado de los objetos.

El Encapsulamiento permite por tanto reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Este principio se utiliza típicamente de forma conjunta con el **Principio de Ocultación** en virtud del cual cada objeto está aislado del exterior y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones no deseadas.

En la Figura 2 se mostraba cómo podía modelarse una bicicleta como un objeto. El diagrama del objeto bicicleta mostraba sus atributos en el núcleo o centro del objeto y los métodos rodeando al núcleo y protegiéndolo de otros objetos del programa. Este hecho de empaquetar o proteger los atributos miembro mediante los métodos miembro es lo que se denomina Encapsulamiento. Es decir, en base al Encapsulamiento los atributos de un objeto solo pueden ser leídos o modificados a través de sus métodos. Esta aproximación conceptual que muestra el núcleo de atributos miembro del objeto protegido por una membrana protectora de métodos miembro es la representación ideal de un objeto y es el ideal que los programadores que trabajan con el paradigma de orientación a objetos deberían intentar perseguir. Sin embargo, esto debe matizarse, ya que, también es posible establecer grados de visibilidad puesto que, por razones prácticas, puede que un objeto desee, por ejemplo, exponer alguno de sus atributos miembro, u ocultar alguno de sus métodos miembro. Es decir, que algunos lenguajes de programación que utilizan el paradigma de orientación a objetos relajan las restricciones impuestas por el Encapsulamiento permitiendo un acceso directo a los datos internos del objeto de una manera controlada. En este sentido, hay que señalar que el lenguaje de programación Java permite establecer diversos niveles, en concreto cuatro niveles, de protección de los atributos y de los métodos miembro que permiten determinar qué objetos de qué clases pueden acceder a qué atributos y a qué métodos.

Esta propiedad permite proteger la información o estado de un objeto, proporcionando a la par una interfaz pública perfectamente definida que otros objetos podrán usar para comunicarse con él. De esta forma, los objetos pueden mantener información privada y cambiar el modo de operar de sus métodos miembro sin que esto afecte a otros objetos que usen los mismos. Es decir, y en relación al ejemplo de la clase `Bicicletas`, podría decirse que no es necesario entender cómo funciona el mecanismo de cambio de marcha para hacer uso de él.

## 4.7 VENTAJAS DE LA ORIENTACIÓN A OBJETOS

En contra de lo que se piensa, la reutilización de código no es la principal ventaja de la orientación a objetos, sino que las principales ventajas de la orientación a objetos serían la facilidad de mantenimiento del código, así como la utilidad de esta filosofía para tratar de solucionar problemas complejos. En relación con esto, Craig Larman señala que “El valor de la orientación a objetos es, principalmente, la capacidad de manejar problemas complejos y de crear sistemas comprensibles que pueden escalar a una complejidad mayor y que son fácilmente adaptables siempre que se hayan diseñado bien”. (Larman)

## 4.8 MAL USO DE LA ORIENTACIÓN A OBJETOS

Las ideas de la filosofía de la orientación a objetos son bien conocidas pero su uso es, en ocasiones, poco purista, en parte por la existencia de numerosos lenguajes de programación híbridos, es decir, en los que el paradigma de programación soportado no es exclusivamente la orientación a objetos.

Una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos: atributos y métodos, sin separar ni dar mayor importancia a uno de ellos, puesto que hacerlo podría producir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen a las primeras por el otro, por lo tanto, se estarían utilizando **objetos degenerados** que son aquellos que solo tiene datos, es decir, que son estructuras de datos, o bien aquellos que solo tienen métodos, es decir, que son bibliotecas de funciones. De hecho, buena parte del software que supuestamente hace uso del paradigma de orientación a objetos está construido haciendo uso de objetos degenerados. De esta manera se estaría realizando una programación estructurada camuflada en un lenguaje de programación orientado a objetos.

La Programación Orientada a Objetos difiere de la programación estructurada tradicional en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. Es decir, en la programación estructurada se escriben funciones que procesan datos. Los programadores que aplican el paradigma de Programación Orientada a Objetos, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que ejecuten sus métodos por sí mismos. [ver Figura 10 y Figura 11]

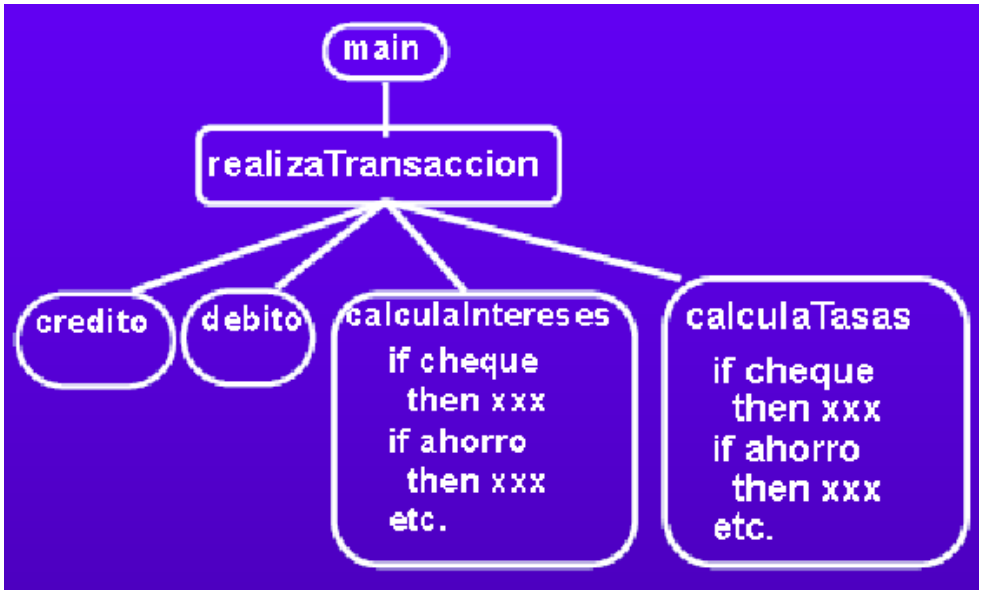



Figura 10: Aproximación procedimental.

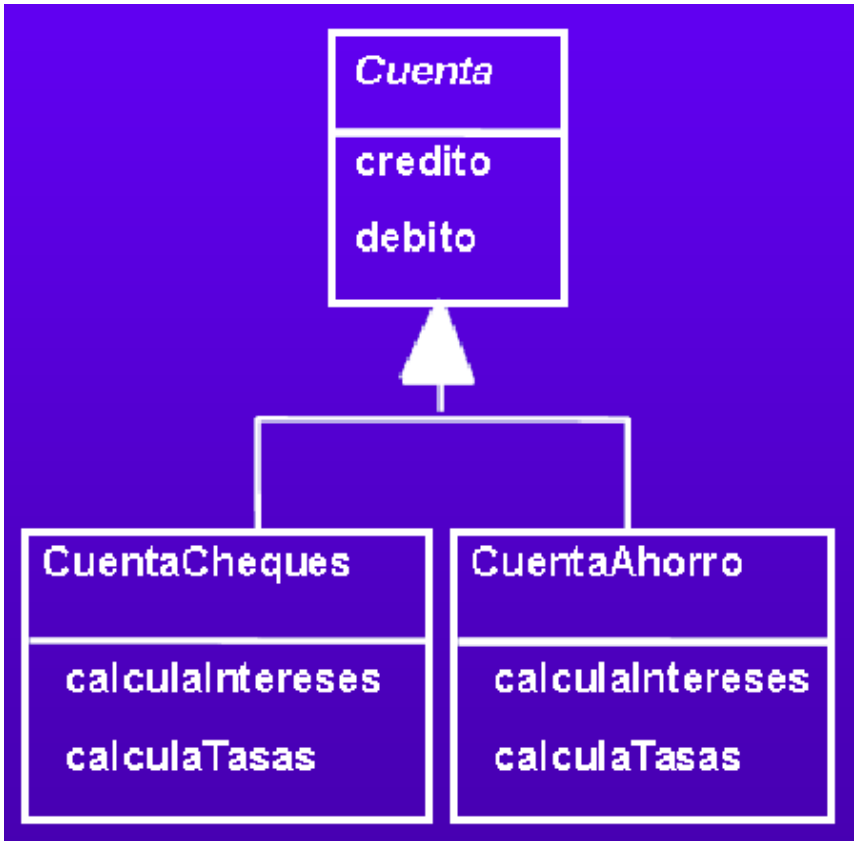



Figura 11: Aproximación mediante orientación a objetos.

Finalmente hay que señalar que en el Anexo I: Object-Oriented Programming Concepts se proporciona una lección sobre la Orientación a Objetos extraída del tutorial Learning the Java Language disponible en `https://docs.oracle.com/javase/tutorial/java/` en lengua inglesa, con el único objetivo de que el alumno se familiarice con la jerga utilizada en el paradigma de programación orientado a objetos en dicho idioma por si en algún momento le resultara de utilidad. La lección se centra en la orientación a objetos en el lenguaje de programación Java.

Nota de los profesores: Si detectas algún error, piensas que alguna información está presentada de manera confusa, o que sobra o falta algún contenido, por favor, envía un mensaje a mperez@tel.uva.es. Gracias por anticipado. Vuestras sugerencias se utilizarán para mejorar esta documentación.

# 5 ANEXO I: OBJECT-ORIENTED PROGRAMMING CONCEPTS

Esta lección está extraída del tutorial Learning the Java Language disponible en `https://docs.oracle.com/javase/tutorial/java/`.

## 5.1 INTRODUCTION

This lesson teaches you the core concepts behind object-oriented programming: objects[41], messages[42], classes[43], and inheritance[44]. This lesson ends by showing you how these concepts translate into code.

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces[45], and packages[46]. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

### 5.1.1 What Is an Object?

An object is a software bundle[47] of related state[48] and behavior[49]. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation[50], and explains the benefits of designing your software in this manner.

### 5.1.2 What Is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

### 5.1.3 What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses[51], and explains how to derive one class from another using the simple syntax provided by the Java programming language.

---

41 Objetos.
42 Mensajes.
43 Clases.
44 Herencia.
45 Interfaces.
46 Paquetes.
47 Paquete software.
48 Estado.
49 Comportamiento.
50 Encapsulado.
51 Superclases.

### 5.1.4 What Is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

### 5.1.5 What Is a Package?

A package is a namespace[52] for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

### 5.1.6 Questions: Object-Oriented Programming Concepts

Use the questions presented in this section to test your understanding of objects, classes, inheritance, interfaces, and packages.

## 5.2 WHAT IS AN OBJECT?

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

[52] Espacio de nombres.

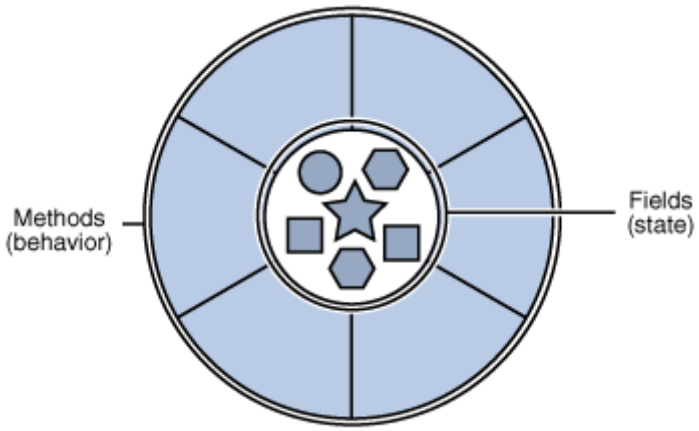



Figure 1: A software object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields[53] (variables[54] in some programming languages) and exposes its behavior through methods[55] (functions[56] in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.

Consider a bicycle, for example:

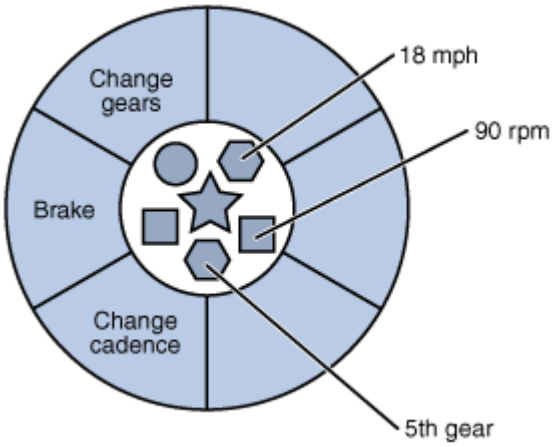



Figure 2: A bicycle modeled as a software object.

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

[53] Atributos.
[54] Variables.
[55] Métodos.
[56] Funciones.

Bundling code[57] into individual software objects provides a number of benefits, including:

- Modularity[58]: The source code[59] for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

- Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

- Code re-use[60]: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

- Pluggability[61] and debugging[62] ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

## 5.3 WHAT IS A CLASS?

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance[63] of the class of objects[64] known as bicycles. A class is the blueprint from which individual objects are created.

The following `Bicycle` class[65] is one possible implementation of a bicycle:

```
class Bicycle {
        int cadence = 0;
        int speed = 0;
        int gear = 1;

        void changeCadence(int newValue){
                cadence = newValue;
        }
        void changeGear(int newValue){
```

[57] Empaquetar código.
[58] Modularidad.
[59] Código fuente.
[60] Reutilización de código.
[61] Facilidad de extender, o añadir funcionalidades a los programas utilizando plugins (conectores o componentes).
[62] Depuración de errores.
[63] Instancia.
[64] Clase de objetos.
[65] Copyright (c) 1995, 2008, Oracle and/or its affiliates. All rights reserved.

```
                gear = newValue;
        }
        void speedUp(int increment){
                speed = speed + increment;
        }
        void applyBrakes(int decrement) {
                speed = speed - decrement;
        }
        void printStates() {
                System.out.println("cadence:"+cadence+"  speed:"+speed+"  gear:"+gear);
        }
}
```

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects. The fields `cadence`, `speed`, and `gear` represent the object's state, and the methods `changeCadence`, `changeGear`, `speedUp` etc., define its interaction with the outside world.

You may have noticed that the `Bicycle` class does not contain a main method[66]. That's because it's not a complete application; it's just the blueprint for bicycles that might be used in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a `BicycleDemo` class[67] that creates two separate Bicycle objects and invokes their methods:

```
class BicycleDemo {
        public static void main(String[] args) {

        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
```

[66] La clase `BicycleDemo` sería la clase cliente de la clase `Bycicle`, puesto que es la que contiene el método `main` con la declaración `public static void main(String arguments[])` a partir del cual comenzará la ejecución del código. Tal y como puede observarse, en la clase `BicycleDemo` se instancia la clase `Bicycle` y se invocan métodos de dicha clase sobre las instancias creadas de la misma.

```
        bike1.printStates();


        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
        }
}
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

## 5.4 WHAT IS INHERITANCE?

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In this example, `Bicycle` now becomes the superclass of `MountainBike`, `RoadBike`, and `TandemBike`. In the Java programming language, each class is allowed to have one direct superclass[68], and each superclass has the potential for an unlimited number of subclasses[69]:

[68] En el lenguaje de programación Java no está permitida la herencia múltiple.
[69] Subclases.

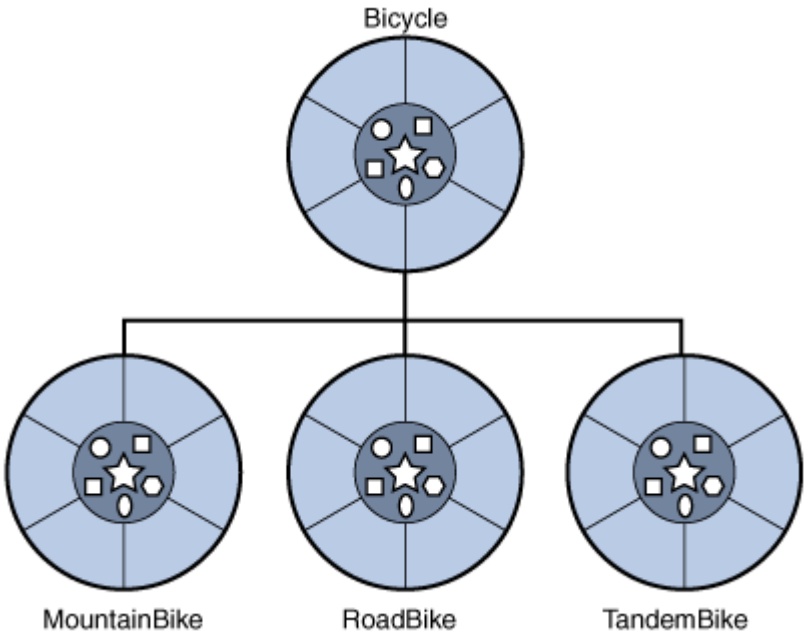



Figure 3: A hierarchy of bicycle classes.

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the `extends` keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {
        // new fields and methods defining a mountain bike would go here
}
```

This gives `MountainBike` all the same fields and methods as `Bicycle`, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

## 5.5 WHAT IS AN INTERFACE?

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's interface with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface Bicycle {
        void changeCadence(int newValue);
        // wheel revolutions per minute
```

```
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as `ACMEBicycle`), and you'd use the `implements` keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {
    // remainder of this class implemented as before
}
```

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

## 5.6 WHAT IS A PACKAGE?

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML[70] pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library[71] (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a String object contains state and behavior for character strings; a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a Socket object allows for the creation and use of network sockets[72]; various GUI[73] objects control buttons and checkboxes[74] and anything else related to Graphical User Interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

---

[70] Hypertext Markup Language, Lenguaje de Marcas de Hipertexto.

[71] Biblioteca de clases.

[72] El término socket designa un concepto abstracto por el cual dos programas, posiblemente ejecutándose en máquinas diferentes, pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada. Un socket queda definido por, un protocolo de comunicaciones del nivel de Transporte del Modelo OSI (Open Systems Interconnection, Interconexión de Sistemas Abiertos) que permite realizar una comunicación para el intercambio de un flujo de datos, una dirección de protocolo de nivel de Red del Modelo OSI, típicamente una dirección IP (Internet Protocol, Protocolo de Internet) si se utiliza la pila de protocolos TCP (Transport Control Protocol, Protocolo de Control de la Transmisión)/IP en los niveles de Red y de Transporte del Modelo OSI, que identifica una máquina, y un número de puerto de protocolo que es un punto abstracto de destino en una máquina que identifica a un programa dentro de dicha máquina.

[73] Graphical User Interface, Interfaz Gráfica de Usuario.

[74] Casillas de verificación.

The Java Platform API Specification[75] contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java Platform 6, Standard Edition. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

[75] Disponible en `http://docs.oracle.com/javase/8/docs/api/index.html`.