Capítulo 7

METODOLOGÍA OO Y JAVA

1. INTRODUCCIÓN

En los primeros capítulos hemos aprendido las estructuras de control que se usan para la construcción de programas y hemos estado usando estructuras de datos simples tales como variables y constantes en la realización de ejercicios sencillos y de mediana complejidad y en la construcción de módulos, subprogramas, específicos.

Diseñar y construir aplicaciones relativamente complejas es un reto importante en nuestros días, ya que una aplicación puede llegar a tener cientos de miles de líneas de código. Considerando que cada problema en particular se adapta a un modelo de programación que determina su patrón de desarrollo, la técnica de análisis, diseño y programación orientada a objetos ha demostrado suficientemente su eficiencia a la hora de enfocar y enfrentarse a la complejidad de las aplicaciones informáticas, permitiendo además, la construcción de un software de más calidad y más barato, cuando se trata de desarrollar aplicaciones de propósito general.

En este capítulo, se presenta la evolución histórica de las técnicas de programación y los conceptos básicos que soporta el paradigma orientado a objetos.

2. EVOLUCIÓN HISTÓRICA

2.1. Orígenes del paradigma OO

El modelo orientado a objetos es relativamente antiguo. El germen nació con el lenguaje Simula, presentado en Oslo en 1967, cuyo objetivo era realizar un lenguaje de programación que permitiera la simulación de procesos paralelos. Con él aparece la idea de objeto como eje central del paradigma, asociando datos y procesos en una misma entidad.

Las ideas de Simula fueron reutilizadas y desarrolladas ampliamente por SMALLTALK, convirtiéndose en el lenguaje paradigmático por excelencia de la programación orientada a objetos. Fue diseñado en el Centro de Investigación de

Xerox (Palo Alto, USA) a mediados de la década de los 70, con el fin de facilitar al usuario un sistema potente con una interfaz ágil, para lo que generó un entorno gráfico de sistema multiventana con pantalla "bitmap". Fue utilizado con gran éxito en la primera estación gráfica de Xerox Parc (antecesor del Macintosh).

Actualmente, SMALLTALK se usa con fines pedagógicos como lenguaje puramente académico, siendo al paradigma orientado a objetos lo que en otro tiempo fue PASCAL al paradigma imperativo estructurado.

Con la aparición de máquinas más potentes, fueron surgiendo nuevos lenguajes que iban incorporando el nuevo paradigma. Unos son extensiones de lenguajes diseñados como imperativos, como C++, Objet LISP, Objet PASCAL, Oberon (Modula orientado a objetos), no considerados puramente orientados a objetos pero que incorporan sus características más importantes. Otros nacen con el nuevo modelo incorporado, ADA, EIFELL, JAVA, ACTOR, etc.

2.2. Evolución histórica del diseño de software

Antes de los años 60, el diseño y la programación de aplicaciones era considerado como un arte, no había método para el diseño ni la implementación de software, no había medios para el mantenimiento de programas, de manera que si algo iba mal, era mejor sustituirlo que revisarlo. Por esta razón, la mayoría del software escrito en esta época tuvo que ser posteriormente reescrito. Esto desembocó en una conocida crisis del software.

En los años 60 surge la programación procedural, basada en procedimientos, que consiste en describir etapa a etapa el modo de construir la solución de un problema y cuyas características más importantes consisten en el aumento de la legibilidad de programas, bajar los costes de mantenimiento y acelerar el proceso de desarrollo. Aparece también la técnica de programación estructurada que pone límites al programador con el uso de estructuras de control de flujo y estructuras de datos englobadas dentro de los propios lenguajes de programación, como una primera aproximación de la abstracción de datos. Esta idea es incorporada y desarrollada por el paradigma orientado a objetos, de manera que actualmente la metodología de programación reconoce la necesidad de la abstracción de datos para diferenciar las especificaciones de su implementación y su representación interna.

La aparición en esta época de la programación modular ayuda a la producción de software, dividiendo la construcción de programas en módulos que pueden ser reutilizados en otras aplicaciones, idea igualmente incorporada a la OO, e introduciendo por primera vez el concepto de Tipo Abstracto de datos. Los datos empiezan a tener un papel importante.

A medida que el tiempo transcurría, los proyectos iban creciendo de tamaño y abarcaban nuevos y distintos campos, a la vez que eran realizados por un equipo más grande de personas. Curiosamente, se observó que cuanto mayor era el equipo de desarrollo menor era el rendimiento individual por la cantidad de esfuerzo que había que invertir en coordinación.

En este contexto, nace la POO con la idea de que el trabajo pueda ser dividido de forma coherente en pequeñas unidades independientes y de forma que la necesidad de coordinación se redujera al mínimo. Brand J. Cox (Objetive-C) acuñó el término *software-IC* (*componentes integrados software*), con la misma idea que se hablaba de componentes integrados hardware. Por ejemplo, si un diseñador de hardware necesita construir una placa con cierta funcionalidad, estudia los componentes que necesita de forma que, si existe un determinado componente en el mercado, la compra sin más, pero si, por el contrario no existe, tiene dos alternativas: proponer a un equipo de diseño que la desarrolle o producir el efecto deseado por otros medios. En cualquier caso, lo que se destaca es que el trabajo del diseñador no es construir todos los componentes.

La POO divide a la comunidad de programadores en dos grupos:

- **Productores de componentes**. Son equipos que crean componentes, los someten a pruebas de depuración y proporcionan una interfaz de comunicación de manera que el producto llega al usuario en perfecto estado para su uso. La interfaz suministrada impide conocer el interior del componente, lo cual permite que los cambios internos que sufran no afecten a los consumidores de dichas componentes.

- **Consumidores de componentes**. La misión del consumidor de componentes (programador) es conocer los distintos componentes que existen en el mercado y usarlos convenientemente para conseguir el objetivo deseado.

Con la misma filosofía de la POO que puede llamarse clásica, implementada por ejemplo en C++ y Java, aparecieron a partir de los años 90 nuevos paradigmas de programación tales como: la programación orientada a eventos (Visual Basic), orientada a componentes (.Net), programación por contrato, que es una idea básica de la programación moderna, popularizada gracias al lenguaje Eiffel y la más actual, la **Programación Orientada a Aspectos** (Aspectj).

Programar por contrato es una forma de declarar las condiciones de corrección como parte del propio programa. Esta idea la hemos estado aplicando en la descripción de interfaces de los subprogramas y será usada también en la descripción de interfaces de objetos.

La **POA** es una de las últimas ideas en Informática y puede ayudar a traspasar la barrera de complejidad de las aplicaciones cuando se tratan ciertos "aspectos" del sistema como la seguridad, el control de errores y las transacciones.

2.3. ¿Qué significa orientado a objetos?

La aproximación orientada a objetos se caracteriza por la idea central, el objeto. Un objeto es un concepto que asocia datos y procesos a una misma entidad, manteniendo una fuerte relación entre las estructuras de datos y los comportamientos en respuesta a mensajes, dejando visible sólo la interfaz de comunicación con el

exterior. De esta forma, el software se organiza como una colección de objetos que representan abstracciones del mundo real.

La esencia del desarrollo orientado a objetos reside en:

- La identificación y la organización de entidades en el dominio de la aplicación y no de la representación final en un lenguaje de programación.
- Es un proceso intelectual independiente de cualquier lenguaje.
- Se atienden temas conceptuales de primer nivel, no implementaciones que corresponden a un nivel inferior.
- El paradigma es aplicable a todas las fases y elementos que forman parte de la ingeniería del software: análisis, diseño, programación y bases de datos.

Como consecuencia, con la puesta en práctica de esta forma de pensar:

- Existe la posibilidad de representar directamente las entidades del mundo real en los escenarios informáticos, sin necesidad de deformarlas.
- Se facilita enormemente la reutilización y modificabilidad del software a partir, por ejemplo, de la construcción de librerías especializadas.
- Es posible trabajar con entornos de desarrollo más potentes. Por ejemplo, para depuración y traza de aplicaciones.
- Es posible disponer de entornos gráficos de comunicación interactiva usuario-máquina de gran calidad.
- Se integran utilidades para la construcción de prototipos de aplicaciones, sin necesidad de escribir todo el código para observar su funcionamiento.
- Se puede trabajar en equipo, desarrollando en paralelo módulos para una aplicación con un esfuerzo mínimo en coordinación.

Por tanto, el modelo orientado a objetos contribuye al diseño de algoritmos con una aportación importante, en cuanto a la abstracción, generalización e interacción, y en cuanto a la sencillez de los procedimientos de adaptación y de extensión de software que permite.

3. LA FAMILIA ORIENTADA A OBJETOS

Los conceptos de la filosofía orientada a objetos no sólo alcanzan al contexto de la programación, dando lugar a los lenguajes mencionados anteriormente, Smalltalk, Eifell, Java, C++, Object Pascal y C# entre otros, sino que se han extendido a todos los ámbitos que integran el desarrollo de software.

En el terreno del análisis y diseño de aplicaciones, en la década de los años 80, surgieron distintas metodologías (las consideradas clásicas o de "primera generación"), por ejemplo: *OOSD* (*Object Oriented Structured Design*) [WASSERMAN et al.,1990], *OMT* (*Object Modeling Technique*) [RUMBAUGH et al., 1991] y *JSD* de Jackson, que han influido en metodologías más recientes como *MOON* y *OO-JSD*.

Posteriormente, ya en la década de los años 90, comenzaron a aflorar nuevas metodologías de desarrollo orientado a objetos, catalogadas como de "segunda generación", de las que forman parte *FUSION* propuesta por [COLEMAN et al., 1994], *OOA/D* de [BOOCH, 1994] o *MEDEA* [PIATTINI, 1994a]. En esta segunda generación se aprovechan las técnicas y procedimientos de las primeras: fichas CRC de Wirfs-Brock [WIRFS-BROCK et al. 1990], modelo de clases de OMT, para generar metodologías más completas donde se incorporan además métricas y lenguajes formales de especificación.

Podría hablarse de una tercera generación, aparecida en la segunda mitad de los años 90, que se caracteriza por la integración de varios modelos existentes, Booch, Rumbaugh y Jacobson, derivando en *UML* (*Lenguaje Unificado de Modelado*), metodología establecida como estándar en 1997 y adoptada en *MÉTRICA 3*.

Actualmente, se realiza un esfuerzo de unificación que actúa principalmente sobre las notaciones de las técnicas usadas para el modelado orientado a objetos, ganando espacio lo que se denomina "técnicas de desarrollo ágil", en un intento de simplificar la gran complejidad y diversidad de metodologías existentes. En esta línea está el *método XP de Programación extrema* desarrollado por Kent Beck en 1996 métodos como *DSDM* (*Dynamic System Development Method*) de Stapleton en 1997. Y en estos últimos años comienzan a emerger otros métodos como *Cristal* o *RUP* (*Proceso Unificado de Rational*) [KRUCHTEN, 2000] que se adaptan perfectamente a las reglas del desarrollo ágil.

En el entorno de la arquitectura de sistemas de información y con el objetivo de facilitar el desarrollo de sistemas basados en arquitectura multinivel, aparecen estándares como *CORBA* para el desarrollo de sistemas de objetos distribuidos, *J2EE* propuesta de SUN para facilitar el diseño, desarrollo, ensamblaje, distribución y despliegue de sistemas distribuidos multinivel basados en componentes o, por último, *DCOM* de Microsoft que soporta comunicaciones entre objetos en diferentes ordenadores a través de redes de área local, banda ancha y a través de Internet.

Las **BDOO** representan el siguiente paso en la evolución de los sistemas de gestión de bases de datos para soportar el análisis, el diseño y la programación orientada a objetos. Las BDOO almacenan y manejan información que puede ser representada mediante objetos que proporcionan una estructura flexible con un modo de acceso rápido, con gran capacidad de modificación, permitiendo el desarrollo y mantenimiento de aplicaciones complejas ya que se puede utilizar un mismo modelo conceptual y así aplicarlo al análisis, diseño y programación, lo cual reduce el problema entre los diferentes modelos utilizados a través del ciclo de vida y disminuye considerablemente el costo de la aplicación.

En este contexto, existen extensiones de las relacionales como Ingres u Oracle y nuevos sistemas que nacen con los conceptos de la orientación a objetos incorporados tales como: *Object Store* (Object Design, Inc., Burlington, MA), *Versant* (Versant Object Technology, Menlo Park, CA), *Objectivity* (Objectivity, Menlo Park, Ca), *Ontos* (Ontos Inc., Bellerica, MA) o *Genstone* (Servio Corporation, Alameda, CA).

El modelo orientado a objetos también se extiende en el campo de los sistemas operativos dando lugar a interfaces como *X/Windows*, *Motif* o *AWT* y a sistemas completos, por ejemplo, *Next* (escrito en Objetive C), *Oberon*, *SCOUT* o *Nemesis*.

4. POO. CONCEPTOS BÁSICOS

La programación orientada a objetos se fundamenta en estructuras modulares basadas en datos, objetos que incluyen los procedimientos de acceso y comportamiento de esos datos, servicios o métodos del objeto. Estos objetos deben modelar el mundo real. El diseño orientado a objetos ofrece un modelo de construcción de software basado en los objetos que se deben manipular, no en la función que dicho software debe realizar, en un intento de acercar el mundo real a los entornos informáticos.

Dicho de una manera más intuitiva la Programación Orientada a Objetos es una forma de construir programas donde las entidades principales son los objetos. Está basada en la forma que tenemos los humanos de concebir objetos del mundo real, distinguir unos de otros mediante sus **propiedades** y asignarles **funciones** o capacidades. Éstas dependerán de las propiedades que sean **relevantes** para el problema que se quiere resolver.

La idea del diseño y la programación OO es estrechar la franja que separa el mundo real del entorno informático (ver figura 7.1), de manera que el modelo que se construya se parezca lo máximo posible al problema planteado originalmente, el resultado computacional al resultado real y el algoritmo que se establece al proceso que ocurre realmente.

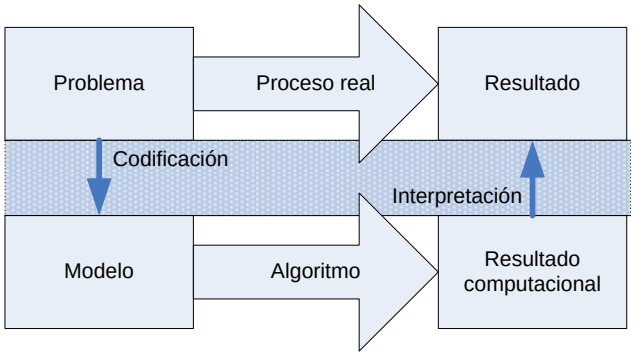

Figura 7.1.

En un primer acercamiento puede decirse que un programa diseñado mediante el modelo orientado a objetos *es un conjunto de objetos que intercambian mensajes que activan procesos los cuales pueden cambiar sus estados internos y pueden devolver parámetros*.

El usuario también es visto como un objeto que interactúa con el resto de objetos del sistema, generalmente por medio del ratón, iconos, menús desplegables y cuadros de diálogos.

4.1. Tipos abstractos de datos

En la actualidad las metodologías de programación parten de la necesidad de definir abstracciones de datos para el desarrollo de programas. El objetivo es separar las especificaciones de una estructura de su implementación y de la representación interna que tengan finalmente, de manera que la principal característica de los lenguajes orientados a objetos es la de incorporar los **tipos abstractos de datos** (**TAD**). Puede definirse un tipo abstracto como la descripción de una estructura de datos junto a un conjunto de operaciones que se pueden realizar sobre dicha estructura (son los métodos o servicios), integrando en la implementación las propiedades formales de esas operaciones (axiomas). Además, se deben incorporar las precondiciones para ejecutar los servicios y postcondiciones resultantes una vez ejecutados esos servicios.

Los axiomas pueden expresarse de distintas maneras, en lenguaje natural o usando expresiones lógicas. Aquí se usará el lenguaje natural.

Un TAD especifica claramente cuáles son los servicios que responden los objetos pertenecientes a él. A este conjunto de servicios, como se ha escrito anteriormente, se le conoce como *interfaz* o *protocolo*. La **interfaz** es la visión que se ofrece a los usuarios respecto a cómo usar las operaciones. Es evidente que los servicios deben implementarse y el TAD debe incorporar los mecanismos para separar dicha implementación de la interfaz.

Un TAD permite:

- La encapsulación: facilidad para guardar conjuntamente datos y operaciones que actúan sobre ellos.

- Ocultación: capacidad de ocultar cara al exterior la representación interna de los objetos y el código de los algoritmos utilizados para implementar su comportamiento.

El TAD tiene las siguientes ventajas:

- Facilidad para modelar el mundo real.

- Al separar implementación de interfaz, se facilita la modificación de la primera sin que afecte a la segunda ni a los programas de aplicación que los usen.

- Los tipos se generan y pueden depurarse independientemente.

- Asegura la consistencia de los datos.

- Aumenta la reutilización de código, facilitando la creación de librerías de componentes, fáciles de usar, mantener y ampliar.

Ejemplo:

Implementación del TAD *Pila compuesta de elementos enteros*.

Función de acceso: los elementos se almacenan en orden de llegada, de manera que el último que llega es el primero que sale (LIFO, Last Input First Ouput).

Tipo Tpila

Elementos de datos: la pila está compuesta por elementos enteros.

Interfaz o protocolo, funcionalidades: (parte visible)

CrearPila: dar sitio en memoria para albergar la pila.

PilaVacia: devuelve un valor lógico Verdadero si la pila no contiene ningún elemento y Falso en otro caso.

MeterElemento: inserta un nuevo elemento en la pila.

SacarElemento: saca elemento de la pila y lo devuelve.

CimaPila: devuelve el elemento cima de la pila.

Aunque la siguiente no es una funcionalidad inherente al tipo, podemos añadirla para ofrecer una interfaz más amplia.

VerElementosPila: Visualiza en pantalla los elementos de la pila.

Axiomas o restricciones:

- Si la pila está vacía, no se puede eliminar ningún elemento.
- Si la pila está llena, no se puede insertar ningún elemento. En este caso deberemos añadir como restricción el número máximo de elementos que admite nuestra pila.
- Si se inserta un nuevo elemento, la cabeza de la pila pasa a ser el elemento insertado.
- Si se elimina un elemento, la cabeza de la pila pasa a ser el siguiente al eliminado.

Implementación: (parte no visible para el cliente)

1.- Representación interna: una estructura para albergar los elementos enteros de datos.

2.- Operaciones que implementan los servicios: en este apartado habría que codificar los servicios especificados en la interfaz, teniendo en cuenta los axiomas establecidos para una pila.

4.2. Objetos

Aunque la idea de objeto está profundamente relacionada con la vida cotidiana, se tratará de definir sus características en el entorno de la programación. El concepto de objeto depende del nivel de abstracción elegido para dicha definición. Por ejemplo, en un nivel poco formal puede decirse que un objeto representa a los datos de un problema real.

Según Booch, un objeto es una "entidad tangible que tiene un comportamiento bien definido". Desde la perspectiva del conocimiento humano, un objeto puede ser algo tangible, algún ente que puede ser comprendido intelectualmente o algo hacia lo que se dirige el pensamiento o la acción [ALONSO y SEGOVIA 1995].

Smith y Tockey [SMITH, 1988] proponen como definición que "un objeto representa un elemento individual e identificable, una unidad o entidad, real o abstracta, con un comportamiento bien definido en el dominio del problema".

Devis afirma que [DEVIS, 1993] "un objeto es una encapsulación abstracta de información, junto con los métodos o procedimientos para manipularla" y Wegner dice que "un objeto contiene operaciones que definen su comportamiento y variables que definen su estado entre llamadas a las operaciones".

Por tanto, un objeto es una mesa, una pelota, un automóvil, como entes tangibles, la televisión, una sala de cine, como elementos comprensibles, o incluso el proceso de fabricación de un televisor, con sus propios elementos y límites conceptuales bien determinados.

En definitiva, un objeto es una abstracción de datos que tiene unas propiedades que determinan su estado, un comportamiento o funcionalidad asociada y una identidad que lo identifican unívocamente y por tanto, lo diferencian de los demás objetos. Esto puede esquematizarse con la siguiente ecuación [BOUZEGHOUB et al.1995]:

Objeto = identidad + estado + comportamiento

Dicho de otro modo, los objetos tendrán la **información** y las **capacidades o funcionalidades** necesarias para resolver los problemas en los que participen. El conjunto de objetos con estructura y comportamiento similares se representa mediante una abstracción que en los lenguajes orientados a objetos se describe con un **tipo** abstracto que se conoce como **clase**.

OBJETO. PROPIEDADES

Las **propiedades** también llamadas **atributos**, son las características "observables" de un objeto desde el exterior del mismo. Pueden ser una o varias y además ser de diversos tipos, enteros, reales, cadenas, booleanos o cualquier otro tipo no básico definido con anterioridad.

Las propiedades pueden ser independientes entre sí y se llaman **básicas** o pueden ser **derivadas** si dependen de alguna otra propiedad, es decir si puede ser calculada a partir de otras propiedades del mismo objeto.

Por otro lado, las propiedades pueden ser o no **modificables** y **consultables** desde el exterior.

OBJETO. IDENTIDAD

Se entiende como identidad de un objeto la propiedad característica que posee dicho objeto que lo distingue absolutamente de todos los demás objetos. La identidad de un objeto es independiente de su estado. En POO cada objeto tiene su propia identidad, que está asociada a él mientras exista, del mismo modo que toda persona tiene una identidad única durante toda su vida, de manera que puede cambiar el color de su pelo, su estado civil, su edad o incluso su nombre, sin que por ello cambie su identidad.

Hay que tener en cuenta que dos objetos con las mismas propiedades y los mismos valores en sus propiedades pueden ser iguales pero no ser idénticos. Por ejemplo, dos coches recién salidos de fábrica, con todos sus atributos iguales no son el mismo coche.

Hablaremos de igualdad e identidad de objetos más adelante.

OBJETO. ESTADO

El **estado** indica ¿cómo se encuentra el objeto? Es decir cuál es el valor de sus propiedades en un momento dado y que naturalmente puede o no variar a lo largo de la vida del objeto. El valor de las propiedades pertenecerá al dominio permitido para una propiedad en concreto. El estado representan la **estructura estática** del objeto.

Por ejemplo, un objeto de tipo coche dispone de propiedades tales como el color, que toma valores dentro de un dominio determinado. Si "mi coche" es rojo, tiene 4 puertas y su marca es Seat, esta información forma parte de la estructura estática del objeto "mi coche".

Los atributos de un objeto dependerán del entorno en el que deba desarrollarse un problema en particular. Por ejemplo, si alguien quiere comprar un coche, los atributos que interesan son el precio, color, potencia del motor, accesorios, número de puertas... Pero si lo que se desea es participar en un rally, interesan otros aspectos como, velocidad, aceleración, anchura de las ruedas o capacidad de frenado.

El que un objeto disponga de un estado supone que ocupa un espacio, ya sea en el mundo real o en la memoria del ordenador, por lo que existe durante un tiempo y puede cambiar el valor de sus atributos, ser referido, creado y destruido.

Los atributos de un objeto deben ser privados a él mismo, es decir, nadie puede acceder a ellos directamente, por tanto, habrá que implementar la forma de accederlos.

OBJETO. COMPORTAMIENTO O FUNCIONALIDAD

El comportamiento de un objeto viene determinado por la manera en que éste interactúa con el exterior y se ofrece a través de un conjunto de **métodos**. y son el mecanismo de comunicación del objeto con el exterior.

La forma de comunicarse con un objeto es enviándole un *mensaje*. El comportamiento, por tanto, es la forma de reaccionar del objeto cuando recibe un mensaje y puede llevar consigo, una consulta o un cambio de estado en dicho objeto.

El **mensaje** representa una orden que se manda a un objeto para que realice una acción con un propósito específico. El mensaje activará un *comportamiento, servicio* o *método*, que será el que determine la forma de actuar.

En el ejemplo del coche, además de las propiedades señaladas podrían definirse algunos servicios, por ejemplo *Parada*, de manera que si *micoche* es una instancia de la abstracción coche, *micoche.Parada* sería el mensaje que habría que mandar para que se activara la acción de parar, lo que provocaría el cambio de estado en el vehículo de en marcha a parado.

Existen en la práctica algunas operaciones que suelen realizar los objetos, pudiendo generar métodos para:

- Permitir consultas sobre aspectos internos del objeto, operación ésta que accede a los atributos del objeto pero no cambia su estado.
- Modificar atributos del objeto, alterando con ello su estado.
- Otros.

Como se ha dicho anteriormente la **encapsulación** es un concepto clave en POO. Consiste en **ocultar** la forma en que se almacena la información que determina el estado del objeto. Esto supone la obligación de que toda la **comunicación con** el objeto se haga a través de los **métodos** implementados con ese propósito. Se trata de **ocultar** información **irrelevante** para quien utiliza el objeto. Las propiedades de un objeto sólo serán accesibles para consulta o modificación a través de sus **métodos.**

ENVÍO DE MENSAJES

El conjunto de mensajes que un objeto es capaz de responder se conoce como su **protocolo** o **interfaz** y define su comportamiento.

En un mensaje intervienen:

- El **receptor,** que es el objeto que recibe el mensaje.
- El **selecto**r, que es el comportamiento o servicio implicado en el mensaje.

El envío de mensajes es la forma de comunicación entre objetos por llamada a un método público del objeto que recibe el mensaje. El mensaje está representado por el identificador del objeto receptor, el nombre del método que se desea activar y los argumentos si los hubiera, según el siguiente formato:

nombreObjeto.nombreMétodo (Argumentos)

Donde **nombreObjeto** es el nombre del objeto al que se manda el mensaje, **nombreMétodo** el método que se desea activar y **Argumentos** son cada uno de los parámetros que se envían separados por comas.

Por ejemplo, siendo *oj1* el objeto receptor del mensaje y *met1* el selector o comportamiento implicado en dicho mensaje y no se pasan parámetros, el envío se escribirá:

```
obj1.met1 ( )
```

El resultado esperado tras el envío de un mensaje dependerá:

- Del estado en que se halle el objeto en el momento de recibir el mensaje.
- Del método invocado en el mensaje.
- De la información que el mensaje transporte.

Cualquier método de un objeto tiene **total visibilidad** sobre sus propiedades. Por tanto, cuando a un objeto se le envía un mensaje, el método activado puede utilizar las propiedades del objeto receptor del mensaje, pudiendo incluso modificarlas.

La única forma de acceder al estado de un objeto debe ser por activación de algún método que lo permita y siempre a través del envío de un mensaje, para garantizar la encapsulación de datos.

Por ejemplo, suponga un objeto lámpara, una lupa y un papel colocados convenientemente. La lámpara es encendida y envía un mensaje lumínico a la lupa, que concentra el haz de luz, aumentando con ello la fuente de calor. Ésta manda un mensaje calórico al papel poniendo en marcha un comportamiento que incide en sus atributos haciéndolo cambiar de estado, por ejemplo, quemándolo si no lo estaba. Podría escribirse el proceso del siguiente modo:

```
lupa.concentraHaz(luz)
papel.aumentaCalor(calor)
```

Donde `lupa` y `papel` son los objetos receptores, `concentraHaz` y `aumentaCalor` los métodos que se desea activar, y `luz` y `calor` los parámetros.

Referencias al propio objeto

Suponga que al recibir el papel el mensaje `aumentaCalor`, comprueba su estado y pone en marcha un método interno `quémate` que lo hace cambiar de estado si el papel no estuviera quemado ya. Este envío de mensaje se referenciará en pseudocódigo con la palabra **mimismo**. El envío al propio papel se escribe `mimismo.quemate()`.

La referencia al propio objeto depende del lenguaje, en Java y C++ se conoce como *this* y en SMALLTALK como *self*.

4.3. Interfaz

Es un elemento de la POO que permite, a priori, establecer **cuáles** son las propiedades de un objeto, ***qué*** *se puede hacer con un objeto de un determinado tipo*, pero no se preocupa de saber **cómo** se hace. Suele llamarse también **protocolo**.

Por ejemplo de un objeto alumno podríamos observar las propiedades nombre, apellidos, DNI, dirección, notas, media de notas, etc y un objeto de este tipo puede tener las siguientes operaciones: conocer el valor de sus propiedades, modificar alguna de ellas o calcular la media.

Cuando empezamos a pensar en un tipo nuevo de objeto debemos modelarlo, debemos indicar qué cosas puede hacer el objeto. Esto significa que deberemos establecer sus propiedades, señalando además cuáles se pueden modificar y cuáles no. Para ello definiremos un **contrato** que indique cómo es posible interaccionar con el objeto. Ese **contrato** establece como se puede relacionar un objeto con los demás que le rodean. Una parte del contrato se recoge en una **interfaz**. El concepto es el mismo que el aplicado hasta ahora en el diseño de subprogramas, dónde la signatura no nos indica todos los detalles de la interfaz. Esta se implementa completamente con el resto de información sobre el uso de la función, es decir, precondiciones y postcondiciones de las funcionalidades y las posibles excepciones que se deban considerar.

Puede haber restricciones al tipo, es decir limitaciones que abarcan a todos los objetos de ese tipo, restricciones de las propiedades o de las funcionalidades compartidas y también puede haber restricciones particulares al resto de funcionalidades.

Formalmente, una interfaz (**interface** en inglés**)** contiene las **signaturas (cabecera, prototipo o firma)** de los métodos que responderá el objeto y que permiten consultar y/o cambiar las propiedades.

Cada vez que se declara un interfaz se está declarando un tipo de dato nuevo que a todos los efectos, puede ser usado como cualquier otro tipo proporcionado por el

lenguaje. Con el nuevo tipo se pueden declarar variables y con ellas construir expresiones o bien puede ser usado para declarar estructuras de datos complejas.

Para hacer el modelo de un objeto es conveniente seguir una metodología para obtener una **interfaz**, en principio conceptual, adecuada para el tipo de objeto.

UNA METODOLOGÍA DE DISEÑO DE INTERFAZ DE OBJETOS

Una metodología apropiada puede ser la siguiente:

1. Conocer a nivel conceptual el objeto, es decir encontrar una definición si la hay y a partir de aquí establecer sus propiedades relevantes. Las propiedades pueden depender de parámetros.
2. Para cada propiedad indicar su tipo, si es consultable o no, posibles parámetros de los que pueda depender.
3. Indicar las relaciones o ligaduras entre propiedades si las hubiera e indicar las propiedades derivadas.
4. Indicar, en su caso, si una propiedad es compartida por todos los objetos del tipo. De aquí saldrán las futuras propiedades de clase (*static*)
5. Por cada propiedad:
   - Si es **consultable**: escribir un método que comience por **get** y continúe con el nombre de la propiedad y devolverá valores del tipo de la propiedad. Estos métodos no modificarán el estado del objeto.
   - Si es **modificable**: escribir un método que comience por **set** y continúe con el nombre de la propiedad. Este método tomará al menos un parámetro con valores del tipo de la propiedad y no devolverá nada (void).
   - Si es **derivada** no puede ser en ningún caso modificable, así que se escribirá un método que empiece por **get**. Recuerda que ser derivada significa que debe ser calculada.
6. Junto a las propiedades se pueden definir operaciones o funcionalidades sobre el objeto. Una operación puede tener parámetros o no y puede suponer una forma de cambiar desde el exterior las propiedades de un objeto. Por cada funcionalidad escribiremos un método con el mismo nombre que dependerá de los mismos parámetros que la operación.
7. Para construir un contrato hay que añadir a la interfaz el resto de información necesaria para su correcto funcionamiento. Cosas tales como precondiciones, postcondiciones o casos de error de cada funcionalidad si los hubiera.

A partir de las propiedades y del resto de información añadida se diseñará la interfaz completa que será implementada posteriormente por una clase.

En capítulo posterior veremos cómo se trata en Java una **interface** pero de momento la usaremos sólo a nivel de diseño, es decir a nivel conceptual.

Ejemplo: Queremos modelar un objeto punto del plano que llamaremos **p**. Para ello definiremos una interfaz que representa un **contrato** con los posibles usuarios del objeto. Aunque para especificar completamente un contrato necesitaremos más elementos, además de una interfaz, por ahora hablaremos indistintamente de contrato o interface.

A nivel conceptual un punto en el plano está definido por dos coordenadas.

Tipo **Punto**:

Propiedades:

X: Double, consultable y modificable
Y: Double, consultable y modificable

Operaciones o Funcionalidades:

De momento no destacamos ninguna.

Interfaz:

```
Double getX()//consulta
Double getY()
void setX(Double x1) //modificación
void setY(Double y1)
```

Para establecer un contrato para Punto podemos añadir algunas restricciones. Por ejemplo, como precondición para `setX` y `setY` debe considerarse que `x1` e `y1` no pueden ser menor que 0 ni mayor que el máximo permitido según la configuración de la pantalla. Esto puede escribirse también mediante una expresión lógica. Por ejemplo `x1>=0` y `x1< 80` y para `y1` lo mismo con los límites adecuados. Además si las operaciones `setX` y `setY` tienen esta responsabilidad se lanzará la excepción correspondiente si no se cumple la precondición.

Para indicar que `p` es una variable de tipo punto se seguirá el mismo formato de declaración de variables de la Unidad 4.

```
Punto p;
```

Si queremos enviar mensajes a la variable `p` deberemos usar el formato visto para el envío de mensajes. Estas expresiones, cuando están bien formadas, tienen un tipo y un valor, como se ha visto en el estudio de subprogramas, es decir la expresión debe ser congruente con la cabecera del método.

```
p.setX(-5);
p.setX(8);
```

Con ello conseguimos que las propiedades X e Y del objeto pasen a ser (-5,8). Ambas son expresiones con efectos laterales porque al ser evaluadas cambia el estado del objeto.

Si queremos consultar las propiedades de `p`, el envío de mensaje será:

```
p.getX();
p.getX();
```

Ejemplo: Diseñemos ahora el tipo `Punto` de una forma más general. Se desea considerar un origen de coordenadas, saber la distancia a dicho origen y la distancia entre dos puntos en el plano.

Tipo **Punto**:

Propiedades:

X: Double, consultable y modificable
Y: Double, consultable y modificable
Origen: Punto, Consultable, compartida //propiedad de la clase
DistanciaAlOrigen: Double, consultable, derivada //será calculada
DistanciaAOtroPunto(Punto p):Double, consultable, derivada //calculada

La propiedad Origen será una propiedad de la clase, por lo tanto, no aparecerá en la interfaz sino que se verá en la implementación de la clase

Interfaz:

```
Double getX(); //consulta
Double getY();
void setX(Double x1); //modificación
void setY(Double y1);
Punto getOrigen();//consulta de la propiedad compartida

//cálculo de las propiedades derivadas
Double  getDistanciaAlOrigen();
Double  getDistanciaAOtroPunto(Punto  p);
```

Ejemplo: Diseñemos ahora un objeto de tipo `Círculo`. El círculo se caracteriza por su centro, su radio y su área. Estamos también interesados en conocer la longitud de la circunferencia que describe.

Como Punto ya es un tipo definido con anterioridad puede ser usado para el diseño del nuevo tipo Circulo

Tipo **Circulo**:

Propiedades:

Centro: Punto, consultable y modificable
Radio: Double, consultable y modificable
Area: Double, consultable, derivada de Radio
Longitud: Double, consultable, derivada de Radio

Operaciones:

MoverElCentroAlOrigen: mueve el centro al origen de coordenadas

La interfaz asociada en este caso es:

```
Punto  getCentro();
Double  getRadio();
Double  getArea(); //derivada
Double  getLongitud(); //derivada
void  setCentro(Punto  p);
void  setRadio(Double  r);
void  moverElCentroAlOrigen();
```

Ejercicio: Establecer un contrato para las dos últimas interfaces añadiendo las restricciones que se consideren convenientes.

TIPO DEFINIDO POR LA INTERFAZ

Como se ha comentado anteriormente al declarar una interfaz estamos definiendo (a nivel de diseño de momento) un tipo nuevo que puede ser usado como cualquier otro tipo del lenguaje. Este tipo tiene el mismo identificador que la interfaz y tiene como características todos los métodos señalados en dicha interfaz. Con ese nuevo tipo se pueden declarar objetos, formar aglomerados de objetos, construir nuevas expresiones usarlos en asignaciones, etc.

Por ejemplo si `c` es de tipo `Circulo` y `x` de tipo `Double`. La expresión siguiente está bien formada y el valor de la variable `x` es actualizado al valor de la propiedad `X` contenido en el estado del centro del círculo.

```
x=c.getCentro().getX();
```

En general al definir un tipo nuevo usamos otros tipos ya definidos. En este caso el tipo `Punto` usa el tipo `Double` ya definido en Java. El tipo `Circulo` usa el tipo `Double` y el tipo `Punto`. Como puede observarse el envío de mensajes va añadiendo "." y nombre de métodos para activar adecuadamente el que se desea.

EJERCICIO PRÁCTICO DE APLICACIÓN

De una urna que contiene inicialmente un determinado número de bolas blancas y bolas negras se pretende diseñar un programa que realice el siguiente proceso: deben extraerse dos bolas de la urna mientras quede más de una bola en ella. Si las dos bolas sacadas son del mismo color, se introducirá una bola negra en la urna, pero si ambas bolas son de distinto color, deberá meterse en la urna una bola blanca. Cuando sólo quede una bola en la urna, se extraerá y determinará su color.

Vamos a estudiar a nivel conceptual el objeto urna, su estado y su conducta, aunque de momento no se implementarán sus métodos, ya que se necesitan otros conocimientos.

Tipo Urna

Propiedades:

NumBolasBlancas: Entero, consultable y modificable

NumBolasNegras: Entero, consultable y modificable

TotBolas: Entero, consultable, derivada de NumBolasBlancas y NumBolasNegras

BomboBlancas: Blancas, Consultable, compartida //propiedad de la clase

BomboNegras: Negras, Consultable, compartida //propiedad de la clase

Operaciones

SacaBola: saca una bola. Devuelve el color (*Color*) de la bola sacada y disminuye en uno el número de bolas de ese color. Solo podrá sacarse bola si existe al menos una bola en la urna.

MeteBola: incrementa en uno el número de bolas del color pasado como parámetro.

Diseña la interfaz, añade las restricciones adecuadas para convertirla en contrato y haz un Pseudocódigo que resuelva el problema.

Pseudocódigo

```
/* Nombre del programa: Urna2 */
ENTORNO:
   DEFINICIÓN DE TIPOS
      Enumerado Color {BLANCA, NEGRA, SINCOLOR}
      Clase Urna /* Más adelante se estudiará su definición */
   OBJETOS
      Urna u
   VARIABLES
      Color a,b
      entero tot= 0,
/*Como almacén de las bolas que se sacan de la urna o de
dónde deben cogerse para meter en ella se usan los bombos de
bolas blancas y negras*/
Vamos a realizar el pseudocódigo detallado e insertamos los
módulos del generalizado como comentarios*/
```

Programa principal

```
Inicio
   Urna u(40,60) /* se inicializa la urna a las bolas indicadas,
                    se verá más adelante*/
   //calcular el número de bolas
   tot= u.totBolas()

   Mientras(tot >1)
       //sacar dos bolas de la urna
       a = u.SacaBola()
       b = u.SacaBola()
```

```
      /*este módulo se expandirá cuando se tengan más
      conocimientos*/

      <MeterLasbolasEnCajonCorrespondientes>

      //Meter bolas en la urna según restricciones

      Si(a==b)
          <DisminuirCajonBolasNegras>
          u.MeteBola(Color.NEGRA)
      en otro caso
          <DisminuirCajonBolasBlancas>
          u.MeteBola(Color.BLANCA)
      Finsi
      tot= u.totBolas()

  Finmientras

  //Sacar la última bola
  c = u.SacaBola()
  Escribir("La última bola es de color ", c)

FinPP
```

4.4. Clases

En el ejemplo anterior, se ha descrito el comportamiento no sólo de la urna **u**, sino de todas las urnas con bolas de dos colores que respondan los mismos servicios, de las que **u** es sólo un ejemplo o una **instancia**. Se ha realizado, por tanto, una abstracción para generar un tipo y se dice que **u** pertenece a la clase *Urna*.

Las **clases** son elementos de la POO que permiten definir los detalles del **estado interno** de un objeto mediante los **atributos**, a partir de las **propiedades** de la interfaz, e implementar las **funcionalidades** a través de los **métodos**.

Normalmente para implementar una clase partimos de la interfaz que un objeto debe ofrecer y que ha sido definida previamente. En la clase se dan los detalles del estado y de los métodos. Decimos que la clase implementa la correspondiente interfaz.

Se puede definir formalmente una clase como una abstracción de un tipo de dato (TAD), caracterizada por atributos y métodos comunes a todas sus instancias (sus objetos). Esta definición puede esquematizarse del siguiente modo:

Clase = atributos + métodos + instanciación

ATRIBUTOS

Los atributos, también llamados **datos miembros** o **variables de instancia**, sirven para establecer los detalles del **estado interno** de los objetos. Son un conjunto de variables, cada una de un tipo y con determinadas restricciones sobre su visibilidad exterior.

Aunque más adelante aprenderemos otras posibilidades restringiremos el acceso a los atributos para que sólo sean visibles desde dentro de la clase, **private**.

Los atributos tienen un nombre y un tipo que puede ser básico o definido previamente por el programador, incluido un tipo de clase. En este último caso, el atributo refiere un objeto de otra clase.

Como regla general, a los atributos les damos el mismo nombre de la propiedad que almacenan, pero comenzando en minúscula.

Qué propiedades se convierten atributos y cuáles no:

- Como una primera aproximación la clase tendrá un atributo por cada propiedad no derivada de la interfaz que implementa y será del mismo tipo que esta propiedad. Es decir, solo definimos atributos para guardar los valores de las propiedades que sean básicas, compartidas o no.
- Las propiedades derivadas, es decir aquellas que se pueden calcular a partir de otras, no tienen un atributo asociado. Aunque aprenderemos otras posibilidades más adelante.
- Las **propiedades compartidas** son aquellas, que como indica su nombre, se comparten por todos los objetos de un mismo tipo. Aunque más adelante lo veremos con más detalle por ahora sólo indicaremos que los atributos que guardan valores de propiedades compartidas deben llevar el modificador **static** además del tipo y la visibilidad.

La declaración de un atributo tiene un ámbito que va desde su declaración dentro de la clase hasta el fin de la clase incluido el cuerpo de los métodos.

La sintaxis para declarar los atributos responde al siguiente patrón:

```
[Modificadores] tipo Identificador [= valor inicial ];
```

`Modificadores` representa los modificadores de visibilidad.. Como puede observarse son opcionales, pero serán **private** de momento y en la mayoría de los casos. Más adelante se estudiarán con detalle.

`tipo` puede ser cualquier tipo de dato definido previamente.

Un atributo puede tener, de forma opcional, un valor inicial.

Ejemplo para definir los atributos de una clase que implemente la interfaz *Punto:*

```
private Double x = 0;
private Double y = 0;
private static Punto origen;
```

Y para una que implemente el interface *Circulo*:

```
private Double radio;
private Punto centro;
```

En estos ejemplos `private` es un modificador que restringe la visibilidad de un atributo sólo al interior de la clase y `static` otro modificador que indica que el atributo y su correspondiente valor serán compartidos por todos los objetos del tipo. Más adelante aprenderemos otros modificadores posibles.

MÉTODOS

Los métodos indican la forma concreta de **consultar o modificar** las propiedades de un objeto determinado y puede consultar o modificar la totalidad o parte del estado de un objeto. Tienen la misma sintaxis que lo estudiado para subprogramas, vistos en la Unidad 6:

```
[Modificadores] TipoRetorno nombreMétodo ([param. formales])
{
    //Cuerpo;
}
```

Para los métodos en la mayoría de los casos el modificador será **public**.

En una interfaz sólo aparecen las cabeceras, sin embargo en la clase deben aparecer la cabecera y el cuerpo de cada uno de los métodos, pues se trata de su definición.

En la comunidad de programadores Java es habitual distinguir dos tipos de métodos, **observadores** y **modificadores**:

- Los **métodos observadores o consultores** son los que implementan las propiedades consultables y por tanto, no modifican el estado del objeto, es decir no modifican los atributos. O lo que es lo mismo los atributos no pueden aparecer, dentro de su cuerpo, en la parte izquierda de una asignación. Normalmente serán métodos observadores aquellos cuyo nombre empiezan por *get*.
- Los métodos **modificadores** o **mutadores** implementan las propiedades modificables, modifican el estado del objeto. Los atributos aparecen, dentro de su cuerpo, en la parte izquierda de una asignación. Normalmente todos los métodos que empiecen por *set* son modificadores.

Ejemplo de un método modificador (setX).

```
public void setX(Double  x1)
{
 x  =  x1;
}
```

Ejemplo de un método consultor (getX).

```
public Double getX(  )
{
```

```
 return x  ;
}
```

MÉTODOS CONSTRUCTORES

Crear un objeto es un proceso similar a crear una variable de tipo primitivo, requiere la asignación de una zona de memoria, declaración y la inicialización de su estado. La creación de objetos se realiza a través de la activación de un método constructor.

El constructor de una clase es un método estándar para inicializar los objetos de esa clase. Es una función que se ejecuta siempre al crear un objeto. Los constructores pues sirven para crear objetos nuevos y establecer su estado inicial. Habitualmente existe más de un método constructor en una clase y todos tienen el mismo nombre que la dicha clase. Lo específico del constructor es que no tiene tipo de retorno asociado.

Para inicializar un objeto basta con llamar a su constructor después de la palabra reservada **new** y asignarlo a una referencia declarada previamente. El propósito de este procedimiento es el de inicializar los datos del objeto.

Los constructores deben inicializar de manera general cada uno de los atributos salvo los que tienen el modificador **static**. Estos últimos se inicializan cuando se declaran. Puede haber otras posibilidades que aprenderemos más adelante.

Ejemplos de métodos constructores de la clase Punto son:

```
public Punto()
{
   x=0.;
   y=0.;
}
public Punto(Double x1,Double y1)
{
   x=x1;
   y=y1;
}
```

Java, cuando no se implementa ningún constructor en la clase, proporciona por defecto un constructor sin parámetros, que deja de estar disponible cuando se añade algún constructor a la clase.

La forma de implementar un constructor es muy dependiente del lenguaje de programación que se utilice, de manera que deberá estudiarse en cada uno de ellos en particular la forma de realizarlo.

TIPOS DE CONSTRUCTORES

Los constructores los define el programador según conviene a las aplicaciones que desarrolla, pero no obstante existen algunos constructores llamados de una manera particular que es conveniente conocer.

Constructor de servicio

Si el programador no define un constructor para una clase, el sistema generará uno automáticamente, sin argumento ni cuerpo, para crear y copiar, pero no para inicializar. Este constructor se conoce con el nombre de **constructor de servicio**. Los atributos de tipo primitivo se inicializan a `0` o `false`, según su tipo, mientras que los atributos de tipo *objeto* (referencia) se inicializan a `null`.

Constructor por defecto

Un constructor puede no tener argumentos y se llama, en este caso, **constructor por defecto**. Se usa generalmente para "limpiar" con ceros, blancos o null las variables de instancia, según estas sean numéricas o cadenas.

Por ejemplo,

```
public Punto()
{
   x=0.;
   y=0.;
}
```

Llamada,

```
Punto pt1 = new Punto();
```

Constructores sobrecargados ordinarios

Aunque el concepto de sobrecarga es más amplio, como primera aproximación podemos decir que se entiende por *sobrecarga* la posibilidad de definir métodos con el mismo nombre dentro de una clase o bien dentro de clases diferentes. Generalmente estos métodos realizan tareas similares pero tienen un comportamiento diferente.

En java los métodos sobrecargados no pueden tener signatura idéntica, debiendo diferenciarse al menos en el tipo de dato devuelto.

Se hablará más delante de la sobrecarga de nuevo por ser esta propiedad fundamental cuando se implementa la herencia.

Es bastante habitual la sobrecarga de constructores en una clase. En estos casos, el compilador resuelve a quién llamar cuando se crea un objeto en razón de los parámetros utilizados en su inicialización. Estos constructores, que pueden aceptar argumentos, 1, 2, ... n siendo n el número de variables de instancia, se conocen como **constructores sobrecargados ordinarios**.

Por ejemplo,

```
public Punto(Double x1,Double y1)
{
   x=x1;
   y=y1;
}
```

El constructor por defecto también puede definirse así,

```
public Punto( )
{
   this(0.0,0.0);//Llamada al constructor con dos parámetros
}
```

Llamada

```
Punto pt2 = new Punto(3,6);
```

Constructor de copia

Cuando se define un constructor de una clase teniendo como parámetro un objeto de dicha clase se le llama **constructor de copia**. Este constructor es llamado por el compilador cuando al declarar un objeto se inicializa con otro definido anteriormente.

Puede ser útil para hacer una copia local de un objeto, pero la forma preferida de obtener la copia de un objeto es utilizar el método **clone**.

Por ejemplo,

```
public Punto(Punto pto)
{
   x = pto.x;
   y = pto.y;
}
```

Llamada:

```
Punto pt3 = new Punto(pt2);
```

Devuelve un nuevo objeto `pt3` cuyo estado inicial es una copia del estado actual del objeto `pt2` declarado y definido anteriormente.

No se usa mucho dentro de las bibliotecas de clases de Java, existe en la clase ***String*** y en las colecciones.

MÉTODOS DESTRUCTORES

La destrucción de un objeto consiste en liberar la memoria que ocupa. Este proceso puede ser automático o manual. Para el proceso automático se necesita la incorporación en el lenguaje de un *recolector de basura*. La recolección de basura es una tarea que se realiza cuando un objeto deja de ser referenciado en el programa que lo ha creado. Algunos lenguajes de POO tienen implementada esta funcionalidad, por ejemplo Java y Smalltalk.

La destrucción manual se realiza mediante la llamada a un método destructor. Ésta es la forma que implementa C++.

DEFINICIÓN DE LA CLASE

Una clase queda descrita por la definición del tipo abstracto de dato. Tal como se ha detallado en la clase se especifican sus variables de instancia y los métodos o servicios que pueden responder los objetos pertenecientes a dicha clase.

El formato de definición de una clase Java sigue por lo tanto el siguiente patrón:

```
[Modificadores] class NombreClase{
  // definición de los atributos de la clase
  tipo1 identificador1;
  tipo2 identificador2;
  .........
  // definición de los métodos de la clase
  // Constructores
  public NombreClase ()
  {
       //instrucciones del constructor1
  }
  //Métodos públicos
  public tipoDevuelto nombreMetodo1 (listaParametros)
  {
       //instrucciones del método1
  }
  .........
  //Métodos privados
  private tipoDevuelto nombreMetodo2 (listaParametros)
  {
       //instrucciones del método2
  }
  .........
} //fin de definición de la clase
```

Donde **class** será una palabra reservada y `NombreClass` el nuevo tipo de dato abstracto (TAD).

Esta definición es solo la declaración de la plantilla, por tanto, no ocupa memoria, solo sirve como tipo de dato, serán los objetos declarados de este tipo los que ocupen sitio en memoria, tanta como la suma de los bytes que ocupen los atributos.

Ejemplo de clase que implementa la segunda versión de Punto.

```
public class Punto {
  private Double x;
  private  Double  y;
  private static Punto origen = new Punto();
  public Punto(){
   this.x=0.; //puedo hacer referencia a this o no
   y=0.;
  }
  public Punto(Double x1, Double y1){
```

```
  this.x=x1;
  y=y1;
 }
 public Double getX() {
  return x;
 }
 public Double getY() {
  return y;
 }
 public void setX(Double x1) {
    x=x1;
 }
 public void setY(Double y1) {
  y=y1;
 }
 public Punto getOrigen(){
  return origen;
 }
 public Double getDistanciaAOtroPunto(Punto p){
  Double dx = this.getX()-p.getX();
  Double dy = getY()-p.getY(); //puedo referir o no this
  return Math.sqrt(dx*dx+dy*dy);
 }
 public Double getDistanciaAlOrigen(){
  return this.getDistanciaAOtroPunto(origen)
 }
 public static void mostrar(){
  System.out.println("("+this.getX()+","+
  this.getY()+")");
 }
 public String toString()  {
  String s="("+this.getX()+","+ this.getY()+")";
  return s;
 }
}//fin de definición de la clase
```

Si una clase implementa una interfaz debe implementar cada uno de los métodos de esa interfaz, pero algunas veces estamos interesados en añadir alguna más que hemos considerado interesante. En este caso la clase implementa también los métodos *mostrar* que muestra un *string* y a continuación el objeto y *toString* que convierte el objeto en una cadena de caracteres.

Una clase en Java se define en un fichero independiente con extensión `.java` que puede ser compilado independientemente generando el archivo de bytecodes `.class`, que será cargado por la máquina virtual cuando se necesite. Este archivo no es directamente ejecutable por el intérprete, ya que el código fuente no incluye ningún método principal (*main*).

Ejercicio: Implementar todas las clases correspondientes a las interfaces estudiadas anteriormente. Incluye en ella todos los términos del contrato que consideres necesarios.

INSTANCIACIÓN O DECLARACIÓN DE OBJETOS

La clase es una generadora de instancias, de manera que, una vez definido el tipo de dato, podrán crearse, declararse, objetos simples o estructuras complejas de ese tipo de datos, es decir, pueden declararse ***instancias*** o agrupaciones de instancias. Una instancia es *referenciada* por una variable que almacena su dirección de memoria. Por ejemplo, una instancia de la clase "Coche" sería el objeto "micoche".

El concepto de instancia relaciona íntimamente el concepto de objeto con el de clase. Son las instancias las que reciben los mensajes definidos en la clase a la que pertenecen.

El hecho de declarar un objeto se conoce como instanciación y su formato es:

```
Tipo nombreObjeto;
```

Cuando se declara un objeto de una clase:

- Se crea una referencia a una instancia de dicha clase
- Si la clase no proporciona constructor inicialmente toma el valor *null* porque no existe objeto al que referirse.

```
Coche micoche; //la referencia coche toma valor null
```

- Para poder usar el objeto hay que crearlo además de declararlo, esto se hace mediante el operador *new*.

```
Coche micoche = new Coche();; //la referencia coche toma
                 // el valor previsto en el constructor.
```

Para poder probar el código de la clase deberá construirse un conductor, llamado habitualmente en estos entornos *TestNombreClase*.

Los programas en Java constarán por lo general de varias clases de Java en distintos archivos fuente y una más que llevará el método *main*.

Vemos a continuación un conductor para probar la clase *Punto*.

```
public  class  TestPunto{

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

```
        Punto p= new Punto(2.0,3.0);
        mostrar("Punto:", p);
        p.setX(3.0);
        mostrar("Punto:", p);
        p.setY(2.0);
        mostrar("Punto:", p);
    }
}
```

Los resultados esperados en la consola son:

```
Punto:  (2.0,3.0)
Punto:  (3.0,3.0)
Punto:  (3.0,2.0)
```

Nota: Cuando se dice que *Java no tiene punteros* simplemente se indica que Java no tiene punteros que el programador pueda *ver*, ya que todas las referencias a objeto son de hecho punteros en la representación interna estos.

Ejercicio: Construye conductores para probar exhaustivamente las clases implementadas anteriormente.

5. REFERENCIAS EN JAVA

Hay una distinción crítica entre la forma de manipular los tipos simples y las clases en Java. Las referencias a objetos realmente no contienen a los objetos a los que referencian. De esta forma se pueden crear múltiples referencias al mismo objeto, como por ejemplo:

```
Punto p = new Punto (2,3),p1 =p;
```

Aunque tan sólo se creó un objeto *Punto*, hay dos variables (*p* y *p1*) que lo referencian. Cualquier cambio realizado en el objeto referenciado por *p* afectará al objeto referenciado por *p1*. La asignación de *p* a *p1* no reserva memoria ni modifica el objeto.

De hecho, las asignaciones posteriores de *p* simplemente desengancharán *p* del objeto, sin afectarlo:

```
p = null; // p1 todavía apunta al objeto creado con new
```

Aunque se haya asignado *null* a *p*, *p1* todavía apunta al objeto creado por el operador *new*.

Cuando ya no haya ninguna variable que haga referencia a un objeto, Java reclama automáticamente la memoria utilizada por ese objeto, a lo que se denomina *recogida de basura*.

5.1. La referencia *this*

Cada objeto de una clase mantiene su propia copia de sus datos miembros, su memoria privada, pero sólo existe una copia del código de cada función miembro que es compartida por todos los objetos de una clase, por eso es necesario particularizar la función para que conozca la identidad del objeto al que se envía el mensaje, a fin de acceder a los datos de ese objeto.

this es una referencia que se pasa como parámetro implícito a cada función miembro en la llamada y contiene la dirección de inicio del objeto al que se envía el mensaje.

Uno de los objetivos al usar *this* es resolver las ambigüedades cuando los supuestos por omisión no son suficientes. Por ejemplo:

```
public Fecha(int dd, int mm, int aa)
  {
    this.dd = dd;
    this.mm = mm;
    this.aa = aa;
  }
```

Es necesario usar *this*:

- Al devolver en una función una referencia al objeto asociado. Por ejemplo, suponiendo que la clase fecha tuviera otra función miembro tal que sumara la fecha enviada a la del objeto al que va dirigido el mensaje,

```
Fecha SumaFecha(Fecha f)
{
 this.dd = dd+f.dd;
 mm = mm+F.mm;
 aa = aa+f.aa;
 // devuelve una referencia a la memoria privada del
 //objeto al que se envía el mensaje
 return (this);
}
```

- Al asignar un nuevo valor al objeto asociado o hacer una copia del mismo. Por ejemplo,

```
void Func (Miclase x, Miclase y)
{
 this = x; // Asignar un nuevo valor al objeto
 y = this; // Hacer una copia del objeto
}
```

- Otro uso de *this* es en la sobrecarga de un constructor, es decir, puede llamar al constructor en un constructor sobrecargado.

```
public Fecha(int aa)
{
 this(); //llama al constructor por defecto.
```

```
        this.aa = aa; //inicializa año
      }
```

Como hemos dicho con la palabra *this* y el operador punto podemos formar expresiones correctas dentro de una clase. Tras el *this* y el punto (.) puede ir cualquier atributo o método de la clase que no sea *static*. También hemos visto que si bien en muchos casos se puede omitir su uso, otras veces nos ayuda a resolver ambigüedades o para devolver referencias del objeto que ejecutó el método. Veamos un ejemplo completo:

```
// Referencia this.

    class Ventana {
     //atriabutos
        private int largo;
        private int ancho;

    //constructores

     Ventana(){
      // No es necesario usar this.
        largo = 0;
        ancho = 0;
      }

     Ventana(int largo, int ancho) {
        //this.DATOMIEMBRO elimina la ambigüedad.
        this.largo = largo;
        this.ancho = ancho;
      }

    //Métodos

    // Devuelve una referencia al mismo objeto
     public Ventana incrementarLargo() {
        ++largo;
        return this;
     }

     public Ventana incrementarAncho() {
        ++ancho;
        return this;
     }

     public int getAncho() {
        return ancho;
     }

     public int getLargo() {
        //utilizamos la referencia aunque no es necesario.
         return this.largo;
     }

    public void mostrar() {
        // Aquí no hay ambiguedad, podemos utilizar
```

```
        // explicitamente this, o bien
        // no especificarlo.
         System.out.println("Ancho: "+getAncho()+" y Largo: "+
         this.getLargo());
    }

}//fin de la clase Ventana

//Test de la clase Ventana

public class TestVentana {

   public static void main (String [] args) {
   //El constructor con dos parámetros usa this para
   //resolver la ambigüedad.
     System.out.println("Referencia A");
     Ventana A = new Ventana(0,0);
     A.mostrar();

   //incrementarAncho() devuelve una referencia al
   //objeto.
     System.out.println("Se incrementa dos veces el ancho
    del objeto al que apunta A");
     A.incrementarAncho().incrementarAncho();
    //¡OJO!, no usar expresiones que compliquen la lectura
     A.mostrar();

     // El siguiente modo de asignar resulta curioso
     // Creamos una referencia, pero no un nuevo objeto.
     Ventana B;
    // Asignamos la referencia que devuelve
    //incrementarLargo();
     B = A.incrementarLargo();

     //Ahora las dos referencias apuntan al mismo objeto.
     System.out.println("Referencia A y referencia B
    apuntan al mismo objeto.");
     A.mostrar();
     B.mostrar();
   }
}
```

La salida del programa es:

```
Referencia A
Ancho: 0 y Largo: 0
Se incrementa el ancho del objeto al que apunta A
Ancho: 2 y Largo: 0
Referencia A y referencia B apuntan al mismo objeto.
Ancho: 2 y Largo: 1
Ancho: 2 y Largo: 1
```

6. LA CLASE OBJECT

Independientemente de utilizar la palabra reservada `extends` en su declaración, todas las clases de Java derivan de la superclase `Object`. Ésta es la clase raíz de toda la jerarquía de clases de Java (Figura 7.2). Sin embargo, el hecho de que todas las clases deriven implícitamente de la clase `Object` no se considera herencia múltiple.

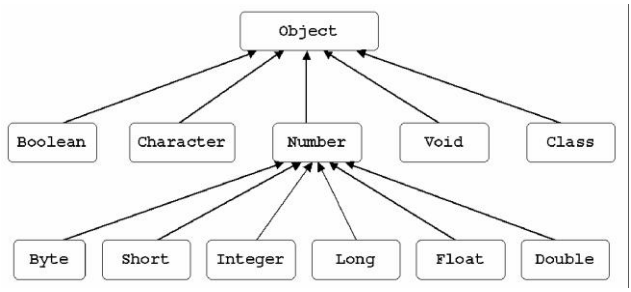

Figura 7.2. Jerarquía de clases predefinidas en Java

Como consecuencia de ello, todas las clases tienen algunos métodos heredados de la clase `Object`.

Tabla 7.3. Algunos de los métodos de la clase predefinida Object

Método	Función
`clone()`	Genera una instancia a partir de otra de la misma clase.
`equals()`	Devuelve un valor lógico que indica si dos instancias de la misma clase son iguales.
`toString()`	Devuelve un `string` que contiene una representación como cadena de caracteres de una estancia
`finalize()`	Finaliza una instancia durante el proceso de recogida de basura.
`hashCode()`	Devuelve una clave hash (su dirección de memoria) para la instancia
`getClass()`	Devuelve la clase a la que pertenece una instancia.

EJERCICIO PRÁCTICO DE APLICACIÓN

Siguiendo con el ejercicio anterior, definir la clase *`URNA`* e implementar los métodos que se establecieron en la interfaz.

Definición del tipo abstracto

Implementación de los métodos