Tecnología de la programación

APUNTES DE CLASE

Marco Antonio Gómez Martín
Jorge Gómez Sanz

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Curso 2017-2018

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

Tecnología de la programación

Apuntes de clase

Grado en Ingeniería de Computadores
Grado en Ingeniería Informática
Grado en Ingeniería del Software
Doble grado de Matemáticas e Ingeniería Informática

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Curso 2017-2018

A todos los que la presente
vieren y entendieren

Resumen

Sé breve en tus razonamientos, que ninguno hay gustoso si es largo.

Miguel de Cervantes Saavedra (1547-1616)

Este documento contiene las notas de clase utilizadas para la impartición de la asignatura de Tecnología de la Programación en la Facultad de Informática de la Universidad Complutense de Madrid.

Su contenido viene a representar un guión de los conceptos que se explican en clase y que el profesor utiliza a modo de guión de la misma forma en la que se suelen utilizar las transparencias. Es por esto que *de ninguna manera* debe entenderse este documento como *completo* en el sentido de contener todo el temario que se imparte en la asignatura.

Capítulo 1

Introducción a la programación orientada a objetos

Estas máquinas no tienen sentido común; todavía no han aprendido a pensar, sólo hacen exactamente lo que se les ordena, ni más ni menos.

Donald Knuth

Resumen: Este primer capítulo presenta una breve introducción la programación orientada a objetos y muestra un primer ejemplo.

Paradigmas de programación

Un paradigma de programación es un estilo de programación que proporciona y determina la **visión** que el programador tiene de la ejecución del programa. De acuerdo al paradigma escogido para la resolución de un problema, variará el modelo que seguirá el programador para resolverlo. Durante la evolución de la tecnología de los sistemas informáticos y de los lenguajes de programación han surgido diversos paradigmas de programación, entre los cuales cabe destacar:

- Programación Imperativa: describe, paso a paso, qué hacer para obtener un resultado. Estos pasos manipulan directamente el estado del programa.
- Programación Funcional: describe la lógica evitando entrar en detalles de control de flujo. Para ello, manipula directamente funciones, usándolas como argumentos y resultados.
- Programación Orientada a Objetos (POO): usa objetos, que encapsulan tanto su propio estado como funciones para manipularlo. Emplea abstracción, encapsulación, herencia y polimorfismo.

Es importante saber que un paradigma de programación puede ser usado en diversos lenguajes de programación, y que los lenguajes de programación pueden permitir el uso de uno o más paradigmas simultáneamente (ver Figura 1).

Lenguaje	Paradigmas que soporta
Java:	Imperativo, orientado a objetos
Python, Ruby:	Imperativo, orientado a objetos y funcional
SmallTalk:	Orientado a objetos
C:	Imperativo
C++, C#:	Imperativo, orientado a objetos y funcional
Erlang:	Funcional
Perl, PHP:	Imperativo, orientado a objetos y funcional
JavaScript:	Imperativo, orientado a objetos[†] y funcional

[†]Con herencia basada en prototipos, en lugar de usar clases

Figura 1: Paradigmas soportados por algunos lenguajes de programación

Programación imperativa

El paradigma de programación imperativo es el que hemos utilizado hasta ahora y el que seguiremos utilizando, combinándolo con orientación a objetos.

En la familia de los lenguajes imperativos, los algoritmos se escriben indicando paso a paso las instrucciones que la máquina debe ejecutar para resolver un determinado problema. De esta forma, los programas imperativos son una secuencia finita de instrucciones, las cuales se ejecutan una tras otra. Los datos utilizados en estos programas se almacenan en memoria principal y se accede a ellos utilizando *variables*. El *estado* de un programa viene dado por el valor que en cada instante tiene el conjunto de variables del programa. Así la ejecución de las instrucciones van modificando el estado del programa de forma incremental hasta llegar a su estado final. Los lenguajes de programación imperativos permiten al programador crear condiciones (`if`), bucles (`for`, `while`), asignar valores a variables, etc. Este modelo de programación es una traducción casi directa de las capacidades *hardware* de las máquinas en el que el contenido de la memoria representa el estado del programa, y las instrucciones máquina son las operaciones.

Entre los lenguajes que utilizan este tipo de programación podemos destacar Pascal, Ada, Cobol, C y derivados (C++, C#, Objective-C), Modula-2 ó Fortran. Dentro de esta categoría se engloban la **programación estructurada**, la **programación modular** y la **programación orientada a objetos**. Cada una de estas extensiones o evoluciones han permitido mejorar el mantenimiento y la calidad de los programas imperativos.

Programación estructurada y programación modular

La programación estructurada es una forma de escribir programas de manera clara. Para ello utiliza únicamente tres estructuras de control que pueden ser combinadas para producir programas. Estas estructuras son:

- Secuencia: sucesión de dos o más operaciones.
- Selección: bifurcación condicional de una o más operaciones.
- Iteración: repetición de una operación mientras se cumple una condición.

El uso de estas estructuras garantizan que los programas tengan exactamente una entrada y una salida. Por esta razón, este modelo de programación evita el uso de la instrucción o instrucciones de transferencia incondicional como `goto` ó `exit`, y desaconseja el uso de

versiones más limitadas, como `break` y `continue` en bucles, o múltiples `return` dentro de una misma función.

La programación modular se presenta como una evolución de la programación estructurada para solucionar problemas de programación más grandes y complejos. Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente. A los subprogramas que resuelven estos subproblemas se les llama módulos - de ahí el nombre de *programación modular*. Adicionalmente, también pueden considerarse módulos las agrupaciones de funciones que resuelven tareas similares (por ejemplo, pintado en pantalla u operaciones matemáticas).

En programación no orientada a objetos, los datos y los subprogramas que los tratan se consideran siempre de forma separada. Los subprogramas no contienen sus propios datos, sino que deben recibirlos desde fuera; esto hace que sea posible reutilizar el mismo subprograma más adelante, con datos distintos (y seguramente desconocidos en el momento de codificarlo).

Cada subprograma debe tener una tarea bien definida; aunque puede necesitar colaborar con otros subprogramas para poder operar.

Entre las ventajas de la programación modular podemos destacar:

- Programas más legibles y manejables
- Simplificación del problema: divide y vencerás
- Favorecen la reutilización de código
- Permiten crear tipos abstractos de datos (TADs).

Los programas desarrollados en C++ en la asignatura de Fundamentos de la Programación siguen este modelo de programación. Eso *no* significa que no se pueda hacer programación orientada a objetos con C++, sino que tanto la filosofía utilizada en esta asignatura a la hora de escribir programas y las características del lenguaje utilizadas se han restringido a este paradigma de programación.

Programación orientada a objetos

Como hemos dicho en la Sección 1.1, la programación orientada a objetos es una forma de programación imperativa, puesto que al programar orientado a objetos se describe la secuencia que debe seguir el programa para resolver un problema dado. La diferencia con otros modelos de programación imperativa es que en la orientación a objetos se hace uso de dos nuevos conceptos: las *clases* y los *objetos*. En vez de tener por un lado los datos y por otro los subprogramas, ambos se juntan para dar lugar a clases y objetos.

Un objeto puede verse como una estructura abstracta que describe con la mayor precisión posible un objeto del mundo real y cómo se relaciona con el resto del mundo que lo rodea. Los objetos son entidades que tienen un determinado estado (sus datos o *atributos*) y un comportamiento (*métodos* o subprogramas que operan sobre ellos).

La clase es la entidad a través de la cual se definen las propiedades y el comportamiento de un conjunto de objetos.

Los programas orientados a objetos no se construyen como un conjunto de subprogramas, sino como un conjunto de objetos que interactúan entre ellos transmitiéndose órdenes e información, lo que se conoce como *envío de mensajes*.

La programación orientada a objetos se basa en el uso de conceptos como la abstracción, la encapsulación, la herencia y el polimorfismo.

- Abstracción: Es el proceso que permite seleccionar los aspectos relevantes (para resolver nuestro problema) dentro de un conjunto e identificar comportamientos comunes de los objetos. La abstracción es clave en el proceso de diseño orientado a objetos, ya que mediante ella podemos construir un conjunto de clases que permitan modelar el problema que se quiere resolver.

- Encapsulación y Ocultación: Aunque son conceptos diferentes suelen emplearse juntos. El término encapsular se refiere a la capacidad para mantener cohesionada toda la información del objeto (datos y comportamiento). La *ocultación* permite mantener los detalles de la implementación privados, de forma que, desde fuera, sólo se podrá acceder a un subconjunto de los datos y comportamiento (atributos y métodos) del objeto.

- Herencia: Es la propiedad a través de la cual es posible construir una clase a partir de otra ya existente.

- Polimorfismo: Es la habilidad que presentan distintas clases para responder al mismo mensaje. Se implementa por medio de la herencia. Hace que distintos objetos puedan responder al mismo mensaje pudiendo producir un comportamiento distinto.

Actualmente, la mayor parte de los lenguajes de programación permiten este modelo de programación. Algunos de ellos son: Java, C++ , C#, Objective-C, PHP, Python, Ruby ó Visual Basic .NET.

Evolución de los enfoques de programación imperativa

Podemos también analizar la evolución histórica adentrándonos en las características técnicas que tenían los lenguajes. Se puede ver que lenguajes más evolucionados disponen de un mayor nivel de abstracción:

- Primero aparece el *lenguaje máquina* y posteriormente el *lenguaje ensamblador*. No hay prácticamente nada de abstracción: las instrucciones utilizadas se corresponden directamente con instrucciones que la máquina es capaz de ejecutar.

- Aparecen los primeros *lenguajes de alto nivel*, que permiten abstraer instrucciones. Una sola instrucción puede producir varias líneas de código en lenguaje de máquina. Por ejemplo, permite utilizar expresiones aritméticas (`i = j + k`) y variables en vez de zonas de memoria. Aparece también la idea de tipo asociado a una variable, que marca qué instrucciones podrás utilizar con esa variable.

- Aparece la programación procedimental, que permite dividir el código en subprogramas (procedimientos y funciones).

- La programación modular permite agrupar varios procedimientos y funciones en módulos.

- Surge la idea de tipos abstractos de datos (TADs), dando lugar a la programación basada en tipos.

- Aparece el concepto de objeto y de clase y con ellos la programación orientada a objetos.

En realidad la lista anterior no es excluyente: en los lenguajes ensambladores se pueden utilizar funciones y procedimientos (programados en ensamblador), o incluso podríamos programar utilizando *las ideas* de la orientación a objetos (cualquier lenguaje "se puede domesticar"). Sin embargo, el lenguaje como tal *no* da soporte a esos conceptos, por lo que es el propio programador el que tiene también que lidiar con los detalles de bajo nivel (que un compilador de un lenguaje orientado a objetos haría por nosotros). Lo mismo ocurre con lenguajes imperativos no orientados a objetos como C (no C++).

Para evitar caer en la tentación de que, tras el párrafo anterior, todos los lenguajes quieran proclamarse ser orientados a objetos, diremos que un lenguaje es orientado a objetos si incorpora los mecanismos necesarios para implementar en los programas las técnicas y métodos establecidos por la programación orientada a objetos: encapsulación, ocultamiento de información, herencia y polimorfismo. Iremos viendo cada uno de estos conceptos en los capítulos siguientes.

Historia de la programación orientada a objetos

El concepto de objeto aparece a finales de los años 60 en Noruega con Simula-67. Simula-67 fue un lenguaje de programación, creado por Ole-Johan Dahl y Kristen Nygaard, diseñado para orientarlo especialmente al campo de las simulaciones. Este lenguaje ya incorpora los conceptos de clase y objeto, así como los de subclase, herencia y polimorfismo.

Tras Simula-67, a principio de los años setenta, aparece otro lenguaje que influye notablemente en este paradigma de la orientación a objetos. Este nuevo lenguaje de programación fue llamado SmallTalk, y sus orígenes se encuentran en las investigaciones realizadas por Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg y otros en el Centro de Investigación de Palo Alto de Xerox (conocido como Xerox PARC). Tenían en marcha el primer proyecto de computadora personal de uso general para conseguir una máquina pequeña, barata y con buenas prestaciones. La interacción con el usuario sería sencilla, utilizando un dispositivo novedoso por aquel entonces: el ratón. Para eso, habría una interfaz gráfica con ventanas, menús, botones, etc. La programación orientada a objetos facilita la programación de esa interfaz, por lo que desarrollan un nuevo lenguaje de programación: Smalltalk. Smalltalk era un lenguaje orientado a objetos *puro* (todo lo que se manejan son objetos). El éxito de la interfaz de usuario hace que otros comercialicen estas ideas. Es el caso de Apple en el Apple Macintosh.

En la década de los 80 renace el interés por la programación orientada a objetos. Aparecen conferencias internacionales dedicadas a la orientación a objetos y se extienden las técnicas de ingeniería del software para hacer uso del paradigma. Aparecen nuevos lenguajes (como Eiffel) y se incorporan características de orientación a objetos a lenguajes ya existentes. C++ fue diseñado por Bjarne Stroustrup y su intención era la de unir las características orientadas a objetos de Simula-67 con la eficiencia de C.

Durante la década de 1990 los objetos se hacen omnipresentes. Surgen sistemas operativos orientados a objetos y entornos de desarrollo orientados a objetos con muchas facilidades para la programación (incluyendo programación visual). En esta década hace su aparición otro lenguaje de programación orientado a objetos muy popular entre los desarrolladores: Java. En la década siguiente, en 2002, aparece C#.

Otros lenguajes que tradicionalmente han sido no orientados a objetos añaden los conceptos de clases y objetos con más o menos éxito: PHP, Perl, LUA, etc.

Un primer vistazo a la OO

Veamos una primera aproximación al concepto de objeto que une datos con funciones. Es una aproximación muy simple a la OO que deja de lado los conceptos de OO más ventajosos.

Como ejemplo, partimos del tipo de datos `Fecha` utilizando las ideas de la programación estructurada y modular utilizando C++. En ella aparece por un lado la definición del tipo de datos `Fecha` y por otro lado las funciones que manejan esas fechas. Es decir, no existe una relación explícita entre las funciones y los datos con los que se trabaja:

```
// Fichero Fecha.h

// Definición del tipo de datos
struct Fecha {
   int dia;
   int mes;
   int anyo;
};

// Declaración de funciones y métodos relacionados con
// el tipo Fecha.
// Las implementaciones aparecerían en Fecha.cpp

// Construye una fecha
Fecha construye(int dia, int mes, int anyo);

// Dada un fecha, le suma el número de días pasado como parámetro.
void suma(Fecha &fecha, int numDias);

// Dadas dos fechas, devuelve el número de días que hay entre ellas.
int resta(const Fecha &fecha1, const Fecha &fecha2);

// Escribe por pantalla la fecha
void escribe(const Fecha &fecha);
```

Para usar lo anterior, se declaran variables de tipo `Fecha` y se llama a las funciones pasando como parámetro dichas variables:

```
int main() {
   Fecha f1, f2;
   ...
   escribe(f1);
   escribe(f2);
   suma(f1, 3);
   std::cout << resta(f1, f2) << std::endl;
   return 0;
}
```

Cuando en vez de programación imperativa con registros (o `struct`), tenemos clases y objetos, todos los procedimientos y funciones relacionados con el tipo `Fecha` *se añaden a los datos de la fecha* y pasan a llamarse *métodos*. De esta forma, el tipo `Fecha` pasa a llamarse *clase* `Fecha`, que tendrá los tres atributos que antes tenía el `struct`. Además, la misma clase incorpora las tres funciones que trabajaban con fechas y que antes teníamos separadas. Estas funciones, como decimos, pasan a llamarse *métodos*.

En C++ la definición es:

```
// Fichero Fecha.h

class Fecha {

private:
   int dia;
   int mes;
   int anyo;

public:
  // Las implementaciones de los métodos aparecen en Fecha.cpp

   Fecha(int dia, int mes, int anyo);
   void suma(int numDias);
   void escribe() const;
   int resta(const Fecha &fecha);
};
```

Como se ve, el primer parámetro que todas las funciones tenían de tipo `Fecha` no aparece explícitamente cuando se hace la conversión a *clases*. En efecto, el parámetro queda oculto bajo la variable `this` de la que hablaremos más adelante.

El otro cambio (además de la desaparición del parámetro) es la forma de uso de `Fecha`:

```
int main() {

   Fecha f1(12, 10, 1492);
   Fecha f2(1, 1, 1970);
    ...
   f1.escribe();
   f2.escribe();
   f1.suma(3);
   std::cout << f1.resta(f2) << std::endl;

}
```

En primer lugar hay que crear objetos de la clase. La ejecución de los métodos de la clase se realiza mediante el envío de mensajes (utilizando el operador punto "."), siendo los receptores de estos mensajes cualquiera de los objetos previamente creados. Como se ve, en la llamada a los métodos *no* aparece como parámetro la variable (objeto) de la clase fecha, sino que ésta aparece inmediatamente antes del nombre del método.

La versión en Java de la misma clase aparece a continuación, y como puede observarse, es muy parecida a la de C++ ya que el estilo sintáctico de Java está tomado principalmente de C++. En Java la unión en las clases de datos y métodos se hace aún más evidente que en C++, pues en Java *no* se distingue entre los ficheros de cabecera (`.h`) y los de implementación (`.cpp`), sino que ambos se combinan en un único fichero `.java`. Aunque el código no lo muestra, las implementaciones de los métodos irían en ese mismo fichero.

```
// Fichero Fecha.java

public class Fecha {

   private int dia;
   private int mes;
   private int anyo;
```

```
    public Fecha(int dia, int mes, int anyo) {
        ...
    }

    public void suma(int numDias) {
        ...
    }

    public void escribe() {
        ...
    }

    public int resta(Fecha fecha) {
        ...
    }
};
```

Por último, un ejemplo de uso en Java sería:

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

        Fecha f1 = new Fecha(12, 10, 1492);
        Fecha f2 = new Fecha(1, 1, 1970);
        // ...

        f1.escribe();
        f2.escribe();
        f1.suma(3);
        System.out.println(f1.resta(f2));
    }
}
```

Capítulo 2

Introducción a Java

Hay sólo dos clases de lenguajes de programación: aquellos de los que la gente está siempre quejándose y aquellos que nadie usa

Bjarne Stroustrup (creador de C++)

Resumen: Este capítulo presenta una breve introducción a Java, asumiendo que el lector conoce el lenguaje C++.

Introducción a Java

Brewed coffee es la bebida preferida de los programadores (al menos en USA), y de ahí el icono que utiliza Java:

Un concepto diferente de lenguaje de programación:

- "*Rather than browsing pages as you surf, you'll be able to treat the Web as a giant hard disk loaded with a never-ending supply of software applications*" - Internet World
- "*The Web will be transformed from the information-delivery medium it is today into a completely interactive computing environment*" - Internet World
- "*MicroSoft's view of the world is Windows-centric and proprietary. Java enables multiplatform deployment and is a completely open system. This reduces the development and distribution costs for software*" - Sun Microsystems Computers

Historia de Java

A finales de los años ochenta Sun Microsystems decide introducirse en el mercado de la electrónica de consumo y más concretamente en los equipos domésticos desarrollando software para dispositivos electrónicos inteligentes y televisión interactiva. Java nace como un lenguaje ideado en sus comienzos para programar este tipo de dispositivos, y en sus primeras versiones se llamó OAK (James Gosling). El equipo (Green Team), compuesto por trece personas y dirigido por James Gosling, trabajó durante 18 meses en el desarrollo de este lenguaje. Los objetivos de Gosling eran implementar una máquina virtual y un lenguaje con una estructura y sintaxis similar a C++.

A mediados de 1994 el equipo de Gosling se reduce ante el escaso crecimiento del mercado PDA y STB y, tras perder acuerdos con Time-Warner y 3DO, se reorienta hacia la Web. Patrick Naughton creó un prototipo de navegador escrito en Java llamado WebRunner, que más tarde sería conocido como HotJava.

En 1994 se realiza una demostración de HotJava y la plataforma Java a los ejecutivos de Sun. Es en las conferencias de SunWorld'95 cuando Java y HotJava salen a la luz y se publica la primera versión de Java.

A finales de 1995 aparece la primera integración de Netscape con Java.

¿Qué es el lenguaje Java?

Java es un lenguaje de programación orientado a objetos y la primera plataforma informática creada por Sun Microsystems en 1995. Java toma mucha de su sintaxis y características de C++, pero su modelo de objetos es más simple y elimina herramientas de bajo nivel como la manipulación directa de punteros o memoria, que suelen inducir a muchos errores. Esto hace que Java se presente como un lenguaje simple que permite crear software altamente fiable.

Como lenguaje orientado a objetos soporta características propias de este paradigma, como la encapsulación, la herencia y el polimorfismo. En Java casi todo son objetos. En general los lenguajes orientados a objetos permiten que en cada momento una clase pueda heredar de dos o más clases, lo que se conoce como *herencia múltiple*. Este tipo de herencia no es permitido por Java, que sólo soporta **herencia simple**, evitando así confusiones y/o complicaciones innecesarias. La herencia simple significa que en cada momento cada clase sólo hereda de otra clase. Todas las clases Java derivan jerárquicamente de la clase "Object".

Se considera un lenguaje **fuertemente tipado**[1], como la mayoría de los lenguajes imperativos. Esta es una gran diferencia con C++ donde es posible la realización de conversión de tipos en tiempo de ejecución, momento en el cual no es posible saber si dicha conversión es posible. Esta operación puede ser muy peligrosa y es eliminada en Java. En Java cada objeto contiene información completa sobre la clase a la que pertenece, por lo que pueden realizarse comprobaciones en tiempo de ejecución sobre la compatibilidad de tipos y emitir la excepción correspondiente si no es posible aplicar la conversión.

En cuanto a la gestión de memoria, Java añade características muy útiles como el **recolector de basura**. No es necesario preocuparse de liberar memoria, el recolector se encarga de eliminar la memoria asignada que ha dejado de utilizarse. Gracias al recolector, el programador únicamente se tiene que preocupar de crear los objetos necesarios para el

[1] Un lenguaje fuertemente tipado no permite violaciones de los tipos de datos, es decir, dada una variable de un tipo concreto, no se puede usar como si fuera una variable de otro tipo distinto a menos que se haga una conversión explícita.

desarrollo de las aplicaciones ya que es el sistema el que se encarga de destruir los objetos en caso de no ser reutilizados.

En cuanto a la **conectividad**, Java proporciona una colección de clases para su uso en aplicaciones de red, que permiten abrir sockets y establecer y aceptar conexiones con servidores o clientes remotos, facilitando así la creación de aplicaciones distribuidas.

Java también da soporte a la **concurrencia** permitiendo desarrollar aplicaciones capaces de ejecutar varias acciones a la vez, lo que se conoce como hilos de ejecución (*multithreading*). Esta capacidad para sincronizar distintos hilos de ejecución es especialmente útil en la creación de aplicaciones de red distribuidas.

Filosofía de funcionamiento. Máquina Virtual de Java

En la Figura 1 se muestra la filosofía de funcionamiento de Java. Los ficheros fuente de Java se guardan en ficheros de extensión java (.java). Se trata de un lenguaje interpretado; el compilador Java traduce cada fichero fuente de clases a fichero objeto de código de bytes (*Bytecode*), produciendo ficheros de extensión .class: es un formato intermedio independiente de la arquitectura de la máquina en que se ejecutará. Estos ficheros .class pasan a través de un verificador de bytecodes que comprueba el formato de los fragmentos de código para detectar fragmentos de código ilegal[2]. Los ficheros .class están diseñados para ser interpretados y ejecutados por cualquier máquina que tenga el sistema de ejecución de Java (*runtime*), sin importar en modo alguno la máquina en que han sido generados. Esta máquina hipotética se conoce como Máquina Virtual de Java (JVM) y es dependiente tanto de la plataforma hardware como software.

Aunque en el párrafo anterior hemos dicho que Java es un lenguaje interpretado, en realidad es tanto interpretado como compilado. De hecho, sólo un 20 % del código Java es interpretado por la JVM ya que los pasos finales de la compilación se manejan localmente. De hecho, el bytecode no se completa hasta que se junta con un entorno de ejecución, que en este caso será la máquina virtual Java de la plataforma en la que estemos. Este tipo de compiladores se conocen como *"Just In Time"* o JIT.

Al ser código interpretado, la ejecución no es tan rápida como el código compilado para una plataforma particular. En los lenguajes de programación tradicionales como puede ser C++, el ordenador puede ejecutar directamente el código generado. Sin embargo en Java, debido a la interpretación que la máquina virtual tiene que hacer de los ficheros, los programas tienden a ejecutarse bastante más lentos que con otros lenguajes de programación. Sin embargo, el hecho de que la última fase de la compilación se lleve a cabo de forma dependiente de la plataforma se considera un rasgo positivo ya que libera al programador de la responsabilidad del mantenimiento de varios fuentes en distintas plataformas.

Los ficheros .jar (Java ARchives) son un tipo de archivos que permiten recopilar o empaquetar en un solo fichero comprimido todos aquellos ficheros (ficheros .class, ficheros de sonido, iconos, etc.) involucrados en una aplicación. Son similares a los ficheros .zip (de hecho están basados en ficheros .zip) y no es necesario descomprimirlos para ser ejecutados, ya que el intérprete de Java es capaz de ejecutar los archivos .jar directamente.

Entre los archivos contenidos en los ficheros .jar se encuentra un tipo de archivo de metadatos llamado MANIFEST.MF que se usa entre otras cosas para definir datos relativos al contenido del fichero .jar y su uso. Si se pretende usar el archivo .jar como ejecutable, el archivo de manifest debe especificar el punto de entrada a la aplicación. Sólo puede haber

[2]Se refiere a código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto.

un fichero de manifiesto en un fichero JAR. Estos ficheros .jar se pueden descomprimir y descompilar, obteniendo de nuevo los fuentes (ingeniería inversa).

El entorno de ejecución de Java es el *Java Runtime Environment* (JRE) que incluye la JVM, un conjunto de bibliotecas Java (APIs) y otros componentes necesarios para que una aplicación escrita en lenguaje Java pueda ser ejecutada. El software que provee herramientas de desarrollo para la creación de programas en Java es el Java Development Kit o JDK.

Entre las implementaciones de la JVM podemos nombrar Kaffe, SableVM y gcj, pero hay muchas más. OpenJDK es una implementación libre de J2SE y J2ME catalogada bajo la licencia GPL de GNU.

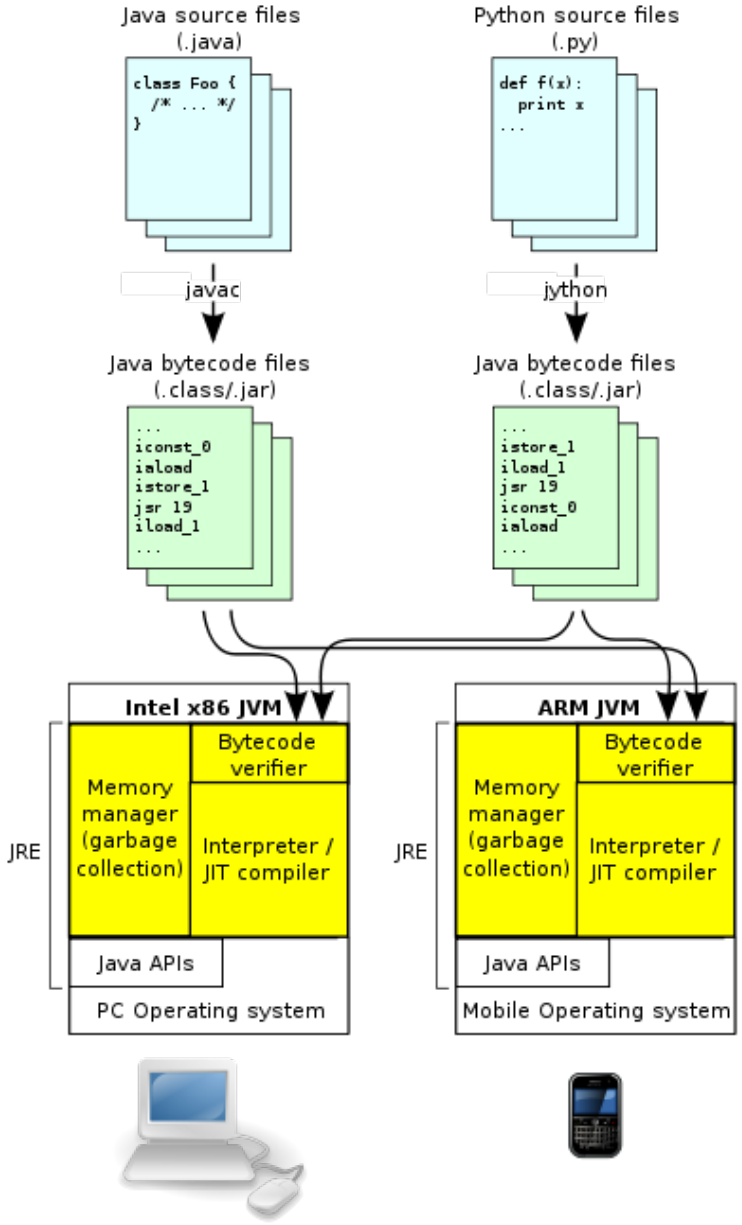


Figura 1: Arquitectura de la Máquina Virtual de Java. Imagen obtenida de `http://en.wikipedia.org/wiki/Java_virtual_machine`

Las especificaciones de la máquina virtual de Java se pueden encontrar en el enlace `http://docs.oracle.com/javase/specs/`. La plataforma Java crece y evoluciona bajo la tutela del JCP (*Java Community Process*). Este organismo es el encargado de dirigir, mediante la creación de nuevas especificaciones y el mantenimiento de las ya existentes, la evolución de la plataforma. Las incorporaciones de nuevas tecnologías a la plataforma Java se realizan a través de un JSR (*Java Specification Request*). Un JSR[3] es un documento que expone la necesidad de esa nueva tecnología, o de modificación de una tecnología ya existente, y cómo afectará este cambio al resto de la plataforma.

Tecnologías Java

Java puede ser usado para crear todo tipo de aplicaciones, desde aplicaciones independientes clásicas hasta aplicaciones web, para móviles[4], o para navegadores en forma de applets[5].

La Plataforma Java se compone de un amplio abanico de tecnologías, cada una de las cuales ofrece una parte del entorno de desarrollo o de ejecución:

- **J2SE** (Java Standard Edition). Es una colección de APIs de Java útiles para muchos programas. Permite programar y desarrollar aplicaciones Java en escritorio y applets, según las necesidades del entorno actual. Provee dos productos de software: Java SE Runtime Environment (JRE) y Java SE Development Kit (JDK). Entre las bibliotecas incluidas podemos destacar:
  - `java.lang`: Incluye clases como String.
  - `java.io`: Para la entrada y salida a través de flujos de datos y ficheros del sistema.
  - `java.util`: Contiene colecciones de datos y clases, el modelo de eventos, facilidades horarias, generación aleatoria de números y otras clases de utilidad.
  - `java.math`: Clases para realizar operaciones aritméticas con la precisión que se desee.
  - `java.text`: Clases e interfaces para manejo de texto, fechas, números y mensajes de una manera independiente de los lenguajes naturales.
  - `java.awt`: La Abstract Window Toolkit contiene rutinas para el desarrollo de interfaces gráficas de usuario. Este paquete también contiene la API de gráficos Java2D.
  - `javax.swing`: Un toolkit de widgets independiente de plataforma, construido sobre AWT.
  - `java.rmi`: Permite a una aplicación Java llamar a los métodos de otra aplicación Java que se está ejecutando en una JVM distinta (RMI: Invocación Remota de Métodos).
  - `java.security`: Incluye herramientas para soportar la seguridad en Java (encriptación, firma electrónica, ...).

[3] Se pueden descargar de `http://www.jcp.org/en/jsr/all`

[4] Android usa Java como lenguaje para código fuente, y soporta la mayoría de las librerías estándar; pero, internamente, usa una reimplementacion de estas librerias, compila el código a un "bytecode" distinto, y lo ejecuta en su propia máquina virtual, llamada ART (antes Dalvik).

[5] Los applets, al igual que Adobe Flash, están desapareciendo por problemas de seguridad y las constantes mejoras en HTML5 + JS

  - `java.sql`: Una implementación de la API JDBC (usada para acceder a bases de datos SQL).

- **J2ME** (Java Micro Edition). Plataforma para dispositivos móviles, PDAs, decodificadores y sistemas telemáticos para coches.
- **J2EE**(Java Enterprise Edition). Es una plataforma de programación para desarrollar y ejecutar software de aplicaciones en lenguaje de programación Java con arquitectura de N niveles distribuida, basándose ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones.
- **JavaFX**. Plataforma para crear contenido interactivo. Centrado en el desarrollo de interfaces gráficas multi-plataforma para Internet. Pensado para sustituir a Swing.
- **Java Card**. Es una tecnología que permite ejecutar applets de forma segura en tarjetas inteligentes.
- **Java TV**. Es un framework basado en Java orientado a su uso en decodificadores de TV, apoyándose en componentes llamados Xlets.
- **Java DB**. Una pequeña base de datos (2.5 MB) evolucionada de código abierto de Apache Derby que soporta transacciones. Soporta el estándar ANSI/ISO SQL a través de JDBC y las APIs de JavaEE. Java DB está incluida en el JDK.
- **Java Embedded**. Es una versión de Java para dispositivos con/sin pantalla de 32MB/8MB o menos sin gráficos. Para Java SE y Java ME. Destaca el bajo consumo de memoria.

Hay muchos más desarrollos en Java, incluyendo frameworks que facilitan el trabajo en dominios concretos. Algunos ejemplos:

- Hibernate. Conecta estructuras de objetos con diferentes bases de datos.
- Struts. Facilita el desarrollo de aplicaciones web usando el paradigma Modelo-Vista-Controlador.
- JUnit. Permite el desarrollo de pruebas automatizadas para cualquier programa Java.
- Marathon y Fest. Especializado en pruebas para interfaces gráficas hechas con Java.

Trabajando con Java

Compilando. El comando `javac`.

Javac es el compilador de java, un programa ejecutable que nos permitirá compilar nuestro código fuente: lee las clases y definición de interfaces escritas en el lenguaje Java y los compila generando los archivos .class. En Java no hay fichero .exe como ocurría al compilar (compilar + enlazar) un programa escrito en C++.

```
javac -d DIRECTORIO_DE_DESTINO_DE_BINARIOS \
      -cp PATH1;PATH2;...;PATH_N \
      NOMBRE_DEL_FICHERO_JAVA
```

Es necesario pasar el path (ruta) de nuestra clase o clases al comando javac.

Ejecutando (Interpretando)

En C++ los ficheros .exe se ejecutan directamente. Algunos ejecutables necesitan que ciertas librerías dinámicas (dll) estén presentes. Cada dll es dependiente del S.O. En Java los programas no se ejecutan sino que se interpretan. Para ello debemos tener instalado al menos el JRE de Java o el JDK completo, puesto que éste incluye el JRE.

Los programas pueden distribuirse con todo lo que necesitan en formato .jar. Se puede ejecutar cada clase desde la consola de comandos:

```
java -cp DIRECTORIO_DE_CLASES mipaquete.miclase
```

Puede haber tantos main como clases existan. Al lanzar Java, se indica cuál es la clase principal y por lo tanto el main a seguir. Esto es útil para probar por separado cada clase antes de ensamblarla con las otras.

Depurando

C++ tiene utilidades para depurar, como por ejemplo gdb o el depurador de Borland, pero no está preparado per se para depuración y no es todo lo versátil que debería.

Java, al ser intepretado, permite depurar mejor. Hay una definición estándar de plataforma de depuración. JDK viene con una implementación con el cliente de línea de comandos jdb y existen otras variantes visuales. Se puede parar/reanudar la ejecución de cualquier programa, se pueden colocar breakpoints condicionales, mostrar interbloqueos, modificar valores en ejecución. Además, permite hacer estudios de rendimiento.

Memoria

En Java no se puede manipular directamente la memoria. No hay operador * y todo lo que se manejan son punteros, a excepción de los tipos simples (`int`, `float`, etcétera).

Los objetos no se borran con delete. Cada clase puede incorporar opcionalmente un método que se debe llamar "public void finalize()..."que realiza tareas de cierre de recursos en clases que lo requieran (e.g. clases que realicen entrada/salida).

Existe un recolector de basura (*garbage collector* o GC) encargado de liberar toda la memoria utilizada por objetos que no son referenciados por ninguna variable. El GC se invoca por sí solo. No se puede forzar la recolección de basura, pero se puede cambiar la política de invocación.

Programando en Java

Estructura general de un fichero de código fuente

La estructura general de un fichero de código fuente de Java es la siguiente:

```
package nombre.del.paquete

// Sección de importación de otras clases y/o paquetes
import java.util.ArrayList;
import java.util.Date;

// Declaración e implementación de la clase
public class NombreClase {
```

```
    // ...
}
```

Donde:

- La primera línea indica en qué *paquete* se añade nuestra clase. Veremos más sobre esto a continuación.
- Los `import` vienen a sustituir los `#include` de C++ e indican qué clases o paquetes externos son utilizados por la clase implementada en el fichero.
- El nombre de la clase pública implementada *debe coincidir* con el nombre del fichero (incluyendo mayúsculas y minúsculas). Por convenio, los nombres de clases empezarán siempre en mayúscula.

Aplicaciones en Java

Las aplicaciones en Java no son más que un conjunto de clases (posiblemente en distintos paquetes) donde alguna de ellas implementa el *punto de entrada a la aplicación*, que Java fuerza a ser un método llamado `main` con la siguiente cabecera:

```
public static void main(String arg[]){
.... código a ejecutar...
}
```

es decir, un método que *no* devuelve ningún valor y que recibe como parámetro un array de cadenas[6].

Paquetes en Java

Los paquetes en Java son *agrupaciones de clases* que pueden, a su vez, contener otros paquetes. Se puede ver cada paquete como un directorio de clases. Los paquetes permiten organizar el código de una gran aplicación, agrupando clases relacionadas. Además, evitan problemas de colisión de nombres, pues puede haber clases con el mismo nombre en paquetes distintos.

Todas las clases deben pertenecer a un paquete concreto[7]. Dado que los paquetes permiten subpaquetes, existe una jerarquía de paquetes. Para nombrar un paquete lo habitual es indicar la ruta completa de paquetes separándola por un punto (“`.`”). Así, cuando decimos `java.lang` nos estamos refiriendo al paquete `lang` que está dentro del paquete `java`.

Cuando programamos una aplicación, la jerarquía de paquetes queda reflejada en el directorio de código fuente. De esta forma en el ejemplo anterior tendríamos un directorio `java`, dentro del cual aparecería un directorio `lang` donde estarían los ficheros `.java` de todas las clases de ese paquete.

En el fichero de código fuente la primera línea indica a qué paquete pertenece la clase. Así, si tenemos el paquete `tp` con todas las clases relacionadas con la asignatura, y dentro de él un paquete por cada práctica, tendremos por ejemplo la clase `MainPr0` que aparecerá en el paquete `tp.pr0` (y que estará en `src/tp/pr0`):

[6] El método además es estático (`static`), algo que se explicará más adelante.

[7] Existe el paquete `default` que contiene todas las clases no definidas explícitamente en un paquete, pero su uso está ampliamente desaconsejado.

```
package tp.pr0;

import ...

public class MainPr0 {
    ...
}
```

Durante el proceso de compilación la jerarquía de directorios de código fuente se replica en el directorio de ficheros compilados.

Si desde una clase queremos utilizar otra clase perteneciente a un paquete distinto al nuestro podemos:

- Poner el nombre completo de la clase: `java.util.ArrayList`.
- Utilizar `import` en la zona de importación con el nombre completo de la clase: `import java.util.ArrayList;`.
- Utilizar `import` importando *todas* las clases del paquete: `import java.util.*;`.

Tipos de datos

Tipos de datos primitivos

En Java existen unos pocos tipos de datos primitivos, similares a los de C++ (con la peculiaridad de que *no* existen las versiones sin signo). Se conservan incluso sus nombres, a excepción del tipo `bool` que pasa a ser `boolean`. Aparece además el tipo `byte` que representa un valor entero de 8 bits.

```
Enteros (con signo): MIN_VALUE      MAX_VALUE
byte  8 bits       -128         +127
short 16 bits      -32768          +32767
int   32 bits      -2147483648     +2147483647
long  64 bits      -2**63          2**63-1

Ejemplos: 0 1  -1 29 035   0x1d  29L

Reales (coma flotante, IEEE 754-1985):
float 32 bits
double   64 bits

Ejemplos: 24.  2.4e1 .24e2 0.0

Booleanos: 1 bit

Ejemplos: true false
```

Otras diferencias con respecto a C++ o cosas que merece la pena destacar:

- El rango de valores *se mantiene entre plataformas*. Los `int` ocupan 32 bits incluso en la versión de Java de 64 bits.
- Los enteros *no* pueden utilizarse como booleanos (no compila).
- El tipo `char` ocupa *2 bytes*, pues no utiliza la codificación ASCII, sino UNICODE.

- Una variable de tipo `boolean` ocupa 4 bytes, excepto cuando son arrays de `boolean` (en ese caso cada uno ocupa 1 byte).

La conversión entre tipos de datos funciona de forma similar a C++. En general los tipos numéricos se convierten automáticamente a tipos numéricos de rango superior (byte a short, short a int, int a long, long a float y float a double). La conversión inversa requiere que el programador lo indique explícitamente utilizando la conversión de tipos.

También se convierten automáticamente los `char` a `int` cuando es necesario, si bien la conversión contraria debe indicarse explícitamente.

Cadenas

Aunque no es un tipo primitivo (sino una clase), en Java también hay soporte para cadenas: `java.lang.String`. Dado que el paquete `java.lang` se importa *siempre*, se puede utilizar directamente `String`[8].

Para delimitar cadenas se utilizan dobles comillas, y pueden concatenarse utilizando el `+`; en la concatenación pueden también intervenir tipos numéricos.

```
String cad = "Esto es una cadena";
String cad2 = new String("Otra forma de crear cadenas");
int x = 3;

System.out.println("El valor de la variable x es " + x);
```

Dado que las cadenas son clases, *no* se puede utilizar el `==` para comparar cadenas. Hay que utilizar el método `compare`.

Declaración de variables

Aunque ya hemos visto algunos ejemplos en el apartado anterior, explicamos aquí que la declaración de variables se hace igual que en C++. Se puede inicializar en la misma instrucción.

```
int nombreVariable;
int nombreVariable=0;
char a, b, c;
String miCadena;
```

En Java *no* se puede utilizar una variable sin inicializarla antes; el compilador lo detecta y *da error*.

Se puede especificar una variable como *constante* utilizando la palabra `final`. En ese caso, su valor debe darse en el momento de la declaración.

```
final float PI = 3.141592;
final int MAX = 255;
```

Arrays

La declaración de arrays es algo distinta que en C++:

```
int[] nombreArray;
int[] nombreArray = {1,2,3};
int[] nombreVariable = new int[3];
```

[8] `String` *no* es una palabra reservada en Java, al igual que ocurre en C++ con `string`.

- Los `[]` aparecen *pegados al tipo base*, y no después del nombre de la variable.
- El tamaño *no* se indica en la declaración del tipo, sino en la inicialización.
- Se puede inicializar el array dando sus valores entre llaves. El compilador creará un array de tantos elementos como encuentre dentro de ellas.
- Si no se utiliza un inicializador, habrá que *reservar el espacio* para el array utilizando `new`, momento en el que indicaremos su tamaño.

Se puede averiguar el tamaño del array en tiempo de ejecución con `length`.

```
int[] nombreVariable = new int[3];

int size = nombreVariable.length;

// En este punto size==3
```

Los arrays bidimensionales se construyen con arrays de arrays:

```
int[][] A;

A = new int[2][3];

// También puede hacerse con
// A = new int[2][];
// for (int i = 0; i < A.length; i++)
//   A[i] = new int[3];
// Eso permitiría a los A[i] tener tamaños distintos.
```

Instrucciones de control y expresiones

Se mantienen y se utilizan igual las instrucciones `if`, `for`, `do` y `while`. El `switch` es también prácticamente igual[9]. Se mantienen también el `break` y el `continue`.

Los operadores también se mantienen y por tanto las expresiones también, `+`, `-`, `*`, `/`, `%`, `&&`, `||`, etc.

Existe también una versión extendida del `for` que permite recorrer cómodamente colecciones de elementos. Por ejemplo, para sumar todos los elementos de un array de enteros se puede utilizar:

```
int[] v;

... // Inicialización de v

int suma = 0;
// En cada vuelta, coge un elemento de v
// y guardalo en la variable entera valor
for (int valor : v)
   suma += valor;

...
```

Por último, el `return` funciona igual que en C++, permitiendo:

[9] Las diferencias son tan sutiles que ni siquiera tiene sentido mencionarlas aquí.

- Especificar el valor de salida de una función (nos referimos aquí a un método que devuelve un valor).
- Provocar la salida de un procedimiento ($\sim$ método que "devuelve" `void`).

Entrada y salida

Para imprimir por pantalla se utiliza `System.out` y `System.err` que hacen las veces de `std::cout` y `std::cerr`.

Para escribir una línea, se utiliza el método `println`:

```
System.out.println("Hola mundo");
```

Igual que ocurría en C++, cuando se mezcla el uso de la salida estándar (`System.out`) y la salida de error (`System.err`), *no* se garantiza el orden, por lo que el código:

```
System.out.println("Mensaje en System.out");
System.err.println("Mensaje en System.err");
```

puede escribir por pantalla las líneas en el orden contrario.

Para leer por teclado se puede utilizar `System.in`, que permite leer carácter a carácter:

```
System.in.read();
```

En vez de eso, también se puede utilizar la clase `java.util.Scanner` que permite leer cómodamente enteros, reales, cadenas, etc.

Enumerados

Java también tiene enumerados. A todos los efectos un enumerado debe verse como una clase que contiene símbolos[10]. Eso implica que los enumerados se crearán en ficheros independientes cuyo nombre coincidirá con el nombre del tipo enumerado que se define.

```
// Fichero Direcciones.java

public enum Direcciones {
   NORTE, ESTE, SUR, OESTE,
   INVALIDA
}
```

Por convenio, los símbolos de los enumerados van todos en mayúscula.

Comentarios

Los comentarios en Java son igual que en C++:

```
// Comentario de una sola línea

/*
  Comentario de varias líneas.
  Hasta el cierre.
*/

// Los comentarios de varias líneas NO
```

[10] De hecho, internamente el compilador crea una clase.

```
// pueden anidarse:

/*
  Un comentario
  // Esto es válido
  /*
     Pero esto no...
  */ // Esta linea cierra la primera apertura,
     // no la segunda...
*/ // Y este cierre no está asociado a nada, y falla
```

En Java existe una herramienta, *javadoc* que permite generar documentación en HTML de para qué sirven las clases, qué hacen los métodos y qué contienen los paquetes; puede aparecer el autor, explicación de los parámetros, etc.

Esa herramienta recoge los comentarios de los `.java` y genera los HTML correspondientes. Para eso, los comentarios deben seguir un cierto convenio. El primero y más importante: son comentarios multilínea que comienzan con `/**` en vez de `/*`[11]. Se puede poner un `*` al principio de cada línea (algunos entornos lo hacen automáticamente) y *javadoc* lo suprimirá en su salida.

```
/**
 * Clase con dos funciones matemáticas simples implementadas
 * como métodos estáticos.
 *
 * @author Walterio Malatesta
 */
public class FuncsMatematicas {

    /**
     * Devuelve el factorial de un número.
     *
     * @param n Número del que calculamos el factorial.
     * @return Factorial del número. Si el número es negativo
     * devolverá 0. El factorial de 0 es 1.
     */
     public static int factorial (int n) {
         ...
     }

    /**
     * Devuelve el número combinatorio de (n, k). El número
     * combinatorio se calcula con n! / ((n-k)! k!), con
     * la peculiaridad de que si no se cumple que 0 <= k <=n,
     * entonces el método devolverá 0.
     *
     * @param n Primer parámetro
     * @param k Segundo parámetro
     * @return Resultado del número combinatorio de los parámetros.
     */
    public static int combinatorio (int n, int k) {
        ...
    }
}
```

[11] No hace falta cerrarlos con `**/`; vale con el habitual `*/`.

Capítulo 3
Clases y objetos

C++: Donde los amigos tienen acceso a tus miembros privados.

Gavin Russell Baker

Resumen: Este capítulo explica dos conceptos fundamentales en la orientación a objetos: las clases y los objetos. No obstante se retrasa la exposición completa de ciertas características asociadas a ellos (como la visibilidad de la sección 2) a la espera de conocer más detalles sobre otros conceptos como la herencia.

Clases vs Objetos

La POO comienza entendiendo la diferencia entre clase y objeto. Según la RAE, se tienen las siguientes acepciones (señaladas con una flecha las elegidas):

```
Objeto
(Del lat. obiectus).
1. m. Todo lo que puede ser materia de conocimiento
o sensibilidad de parte del sujeto, incluso este mismo.
2. m. Aquello que sirve de materia o asunto al ejercicio
de las facultades mentales.
3. m. Término o fin de los actos de las potencias.
4. m. Fin o intento a que se dirige o encamina una acción
u operación.
5. m. Materia o asunto de que se ocupa una ciencia o
estudio.
6. m. cosa.<-----
7. m. ant. Objeción, tacha o reparo.

Clase.
(Del lat. classis).
1. f. Orden o número de personas del mismo grado,
calidad u oficio. La clase de los menestrales
2. f. Orden en que, con arreglo a determinadas
condiciones o calidades, se consideran comprendidas
diferentes personas o cosas. <---
3. f. En las universidades, cada división de
```

```
estudiantes que asisten a sus diferentes aulas.
4. f. En las escuelas, conjunto de niños que
reciben un mismo grado de enseñanza.
5. f. aula (en los centros docentes).
6. f. Lección que da el maestro a los discípulos
cada día.
7. f. En los establecimientos de enseñanza,
cada una de las asignaturas a que se destina
separadamente determinado tiempo.
8. f. Distinción, categoría.
9. f. clase social. Clase alta, baja Clases
dirigentes, trabajadoras
10. f. Bot. y Zool. Grupo taxonómico que
comprende varios órdenes de plantas o de
animales con muchos caracteres comunes. Clase
de las Angiospermas, de los Mamíferos
```

Un *objeto* es único. Se corresponde con una zona de memoria donde se almacena el valor de una serie de variables distinguidas, a las que llamaremos *atributos* sobre las que actúan un conjunto concreto de operaciones, a las que llamaremos *métodos* y que son características del objeto.

Al mismo tiempo, un *objeto* pertenece a una *clase* y se dice del objeto que es un *ejemplar* de la misma[1]. Todos los objetos que pertenecen a una clase tienen los mismos métodos y tipos de atributos. Una clase puede representar un *Tipo Abstracto de Datos* o TAD, aunque no necesariamente. La clase puede tener sus propios métodos y atributos. Son los *métodos de la clase* y *atributos de la clase*, como se verá más adelante (ver sección 5). En tales casos, se hablará de métodos de la clase y de atributos de la clase.

En términos de programación, uno empieza definiendo una clase y luego procede a obtener ejemplares de la misma. En ejecución, se tendrán variables que apuntarán a los objetos creados. Obtener un objeto requiere invocar una *constructora* de la clase deseada (ver sección 1.3). La consecuencia será la obtención de un nuevo ejemplar con un espacio de memoria dedicado a él donde se almacenarán valores de los atributos declarados en la clase original. Una vez creado, se invocará a métodos que define la clase sobre ese objeto, y no otros. El código de los métodos está declarado en la clase y actúa sobre los atributos concretos del objeto (ver sección 1.4).

Convenio de nombres

Ante la duda, se recomienda seguir un convenio de codificación. El de Oracle/Java se puede encontrar en esta dirección: `http://www.oracle.com/technetwork/java/codeconventions-135099.html`

La mayoría de estas convenciones se corresponde con el formateado del código y se pueden aplicar automáticamente con un programa. Al imprimir el código bien formateado se le llama *pretty printing*. En Eclipse se puede pedir al entorno que formatee el texto seleccionándolo y pulsando la combinación de teclas ctrl+shift+f.

Lo que sí debe preocupar es la forma en que se nombran las cosas:

- Nombres de clases. Debe ser un sustantivo, no un verbo. Intentar no usar abreviaturas a no ser que el nombre de la clase se vuelva excesivamente extenso. La primera letra

[1] También se suele hablar de *instancia*, aunque hay discusiones sobre si esta es una buena traducción del inglés *instance*

siempre en mayúscula. Si se trata de varias palabras, juntarlas y hacer que la primera letra de cada palabra esté en mayúscula.

- Nombres de métodos. Debe corresponderse con un verbo. La primera letra en minúscula. Si hay varias palabras en el nombre del método, hacer que vayan seguidas y que la primera letra de cada palabra esté en mayúsculas.
- Nombres de atributos y variables. La primera letra en minúscula. Si hay varias palabras, hacer que vayan seguidas y que la primera letra de cada palabra esté en mayúsculas. Intentar que el nombre sea significativo.
- Nombres de constantes. Todas las letras en mayúsculas. La separación entre palabras, si hubiera varias en el nombre, con un guión bajo.

Atributos

Los atributos son variables cuyo ámbito es el objeto o la clase:

- Si se trata de un *atributo de la clase*, el atributo será accesible desde los *métodos de la clase* y desde todos los objetos creados a partir de la definición de la clase en cualesquiera de sus métodos. En un sentido, estos atributos son compartidos por todos los objetos. Este tipo de atributos se verá en la sección 5.
- Si se trata de un atributo de un objeto, sólo será accesible dentro de aquellos métodos que se le presuman por ser instancia de una clase. Otros objetos no podrán ver o modificar los atributos de otro objeto a no ser que así se establezca en la definición de su clase. Es la visibilidad que se verá en la sección 2.

Es importante que los atributos, como cualquier otra variable, se inicialicen. Los atributos se pueden inicializar o bien en una constructora o bien durante su declaración.

Dada una declaración de clase en Java, se declararía un atributo como sigue:

```
public class EjemploDeClase {
  private TipoDelAtributo atributoDelObjeto=VALOR;
  ...
}
```

El *VALOR* dependerá del tipo del atributo. Este tipo puede ser alguno de los simples ya vistos o bien ser otra clase. Este último caso se verá en la sección 1.3.

En C++ la declaración es similar, aunque *no* puede dárseles valor inicial en la propia declaración (hay que hacerlo en los constructores que describimos en la sección 1.3). En un fichero `.h` se escribiría:

```
class EjemploDeClase {
  ....
  private:
  TipoDelAtributo atributoDelObjeto;
  ...
}
```

Y en el constructor (o constructores) se le dará el valor inicial deseado.

La palabra *private* que aparece tanto en la versión en Java como en la de C++ hace referencia a la visibilidad del atributo. Como norma general, se intentará que los atributos sean siempre privados; se darán más detalles sobre la visibilidad en la sección 2.

> Tener atributos públicos es en general una muy mala práctica y está justificado en muy pocos casos, como en el de la definición de constantes que veremos más adelante.

Los siguientes ejemplos en Java serían inicializaciones válidas:

```
public class EjemploDeClase {
  private int atributoEntero=1;
  private String atributoCadena="ejemplo de cadena";
  private boolean atributoBooleano=true;
}
```

El acceso a los atributos se hace dentro de los métodos (ver sección 1.4) y de las constructoras (ver sección 1.3). Es conveniente que este acceso se haga usando la palabra reservada *this*. Cuando se usa *this*, se pueden tener variables que se llamen igual que el atributo. Si no se usa *this*, el compilador decidirá, cuando vea el nombre de una variable, si se refiere a un atributo o no. El uso de *this* es ligeramente distinto en C++ y en Java. En C++, siempre aparece como `this->` y en java como `this.`.

El ejemplo siguiente en Java ilustra estas variantes:

```
public class EjemploDeClase {
  private int atributoEntero=1;

  private void ejemploDeMetodo1(){
    this.atributoEntero=2; // asigna el valor 2 al atributo
    atributoEntero=3; // asigna el valor 3 al atributo.
  }

  private void ejemploDeMetodo2(){
    int atributoEntero; // Define una variable que se llama igual
                        // que el atributo

    atributoEntero = 2; // Esta asignación no supone modificación
                        // del atributo, aunque se llamen igual
  }

  private void ejemploDeMetodo3(){
    int atributoEntero = 2; // Define una variable que se llama
                            // igual que el atributo.
    this.atributoEntero=atributoEntero; // asigna el valor local
                            // de la variable al atributo
  }
}
```

Como se puede apreciar, el usar variables que se llamen igual que los atributos da lugar a confusiones, por lo que es mejor adoptar la siguiente norma:

> Una variable local no debe tener el mismo nombre que un atributo.

Existen normas de codificación (especialmente en C++, pocas veces utilizado en Java) que sugieren comenzar los nombres de los atributos con el carácter de subrayado o guión bajo (_). De esta forma a la vista del código se sabe que si se hace referencia a un símbolo que comienza con él (por ejemplo `_numCuenta`) se está haciendo referencia a un atributo y

no a una variable local. De esta forma, el uso del `this` para evitar posibles ambigüedades no es necesario.

En Java existe un tipo especial de atributos que se corresponde semánticamente con una constante. Una constante es un atributo que tiene asociada la palabra reservada (modificador de tipo) *final*. En ese caso, el atributo conservará el valor dado durante la definición. En muchas ocasiones, cuando se definen este tipo de constantes, se querrá que ésta esté disponible públicamente y para todos los objetos que sean instancia de la clase. Para ello, además de *final* se utilizan otros modificadores de tipo, para terminar con *public static final*.

```
class EjemploDeClase {
 public static final TipoDeLaConstante NOMBRE_DE_LA_CONSTANTE=
   VALOR_DE_LA_CONSTANTE;
}
```

El acceso desde cualquier sitio *fuera* de la clase que contiene la constante se tendrá que hacer antecediento al nombre de la constante el de la propia clase:

```
System.out.println(EjemploDeClase.NOMBRE_DE_LA_CONSTANTE);
```

En C++ también se pueden especificar atributos constantes, utilizando el modificador `const`[2] que, al igual que en Java, suele implicar convertir el atributo en un atributo de clase con `static`. No obstante, en C++ para evitar consumir memoria las constantes de tipo entero suelen convertirse en enumerados. Otra alternativa es utilizar la opción de los `#define` heredada de C (aunque en ese caso la constante no pertenece a la clase, sino que es un "símbolo" global):

```
// Forma de definir "constantes" en C
#define NOMBRE_DE_LA_CONSTANTE VALOR_DE_LA_CONSTANTE1

class EjemploDeClase {
 public:
  // Atributo (de clase) constante; el valor se da en el .cpp
  static const TipoDeLaConstante NOMBRE_DE_LA_CONSTANTE2;

  // Definición de constante entera sin consumir memoria
  enum { NOMBRE_DE_LA_CONSTANTE3 = VALOR };

}
```

El acceso a estas constantes variaría, según fuera su declaración:

```
...
int main() {
  cout << NOMBRE_DE_LA_CONSTANTE;
  cout << EjemploDeClase::NOMBRE_DE_LA_CONSTANTE2;
  cout << EjemploDeClase::NOMBRE_DE_LA_CONSTANTE3;
 }
```

Constructores y destructores

Los constructores tienen el cometido de inicializar el objeto. Los destructores tienen el cometido de liberar los recursos asociados al objeto. Por recursos pueden entenderse

[2] El modificador `const` también puede utilizarse en otras declaraciones de variables, como variables locales y globales.

ficheros abiertos, segmentos de memoria, puertos de comunicación y similares.

La inicialización de un objeto consiste en asignar valores por defecto a todos los atributos, aunque también puede usarse para inicializar aquellos recursos del sistema que vaya a requerir el objeto. En Java el código del constructor se llama *después* de haber inicializado todos aquellos atributos a los que se les haya dado un valor de inicialización en su declaración.

> Como regla general, tras la invocación de un constructor no pueden quedar atributos sin inicializar.

La definición de los constructores se hace en la clase y se diferencian de los métodos regulares en que se llaman igual que la clase y en que no declaran un tipo para el valor de retorno. Una clase puede tener tantos constructores como se desee, aunque no puede haber dos constructores con los mismos parámetros (que coincida el número de parámetros y sus tipos).

Tanto en Java como en C++, el compilador garantiza que siempre existirá un constructor, aunque el programador no lo defina explícitamente. Cuando el programador no crea ningún constructor, el propio compilador añade el llamado *constructor por defecto*: un constructor público sin parámetros, que no hace nada y no tiene código asociado. Por lo tanto, si no se ve un constructor en el código, hay que asumir el *constructor por defecto*.

La invocación de los constructores es diferente que la de los métodos. En Java, implica el uso del operador `new`. Este operador `new` se aplica también a la creación de arrays de elementos, como se verá más tarde. Es coherente, pues un array se ve en Java como un objeto sobre el cual se pueden invocar métodos.

```
public class EjemploDeClase {
  private int atributoEntero=1;

  public EjemploDeClase(){
    this.atributoEntero=2; // asigna el valor 2 al atributo
  }

  public EjemploDeClase(int nuevoValor){
    this.atributoEntero=nuevoValor; // asigna el valor pasado
    //como parámetro al atributo
  }

  public static void main(String args[]){
   // crea un objeto con el atributo conteniendo el valor 2
   EjemploDeClase unaVariable=new EjemploDeClase();
   // crea un objeto con el atributo conteniendo el valor 5
   EjemploDeClase otraVariable=new EjemploDeClase(5);
  }
}
```

En C++ también puede haber `new`. Este operador lleva a crear objetos independientes en memoria que sean accesibles fuera del ámbito de la variable asociada, un tema que se verá en la sección 6.2. También hay invocaciones sin este operador. En tales casos, basta con indicar el nombre de la clase, el nombre de la variable y colocar unos paréntesis a la derecha del nombre de la variable, agregando los parámetros del constructor que se quiera utilizar[3].

[3] Si se va a llamar al constructor sin parámetros los paréntesis se pueden omitir.

```
class EjemploDeClase {
  public:
  EjemploDeClase(){
    this->atributoEntero=2; // asigna el valor 2 al atributo
  }

  EjemploDeClase(int nuevoValor){
    this->atributoEntero=nuevoValor; // asigna el valor
    //pasado como parámetro al atributo
  }

  private:
  int atributoEntero;
}
...
void main(){
  // crea un objeto con el atributo conteniendo el valor 2
  EjemploDeClase unaVariable();
  // lo mismo que antes, omitiendo los paréntesis
  EjemploDeClase unaVariableIgual;
  // crea un objeto con el atributo conteniendo el valor 4
  EjemploDeClase otraVariable(4);
  // crea un objeto con el atributo conteniendo el valor 2
  EjemploDeClase* otraVariableMas=new EjemploDeClase();
}
```

Los destructores también se declaran en la clase, pero cada clase sólo tiene un destructor. Su cometido principal es liberar los recursos que el objeto ha solicitado desde su construcción y que deben ser liberados. La mayoría de las veces los recursos que un objeto ha solicitado consisten en zonas de memoria para guardar otros atributos, y muy pocas veces para "devolver" otros recursos al sistema, como el cierre de ficheros, de puertos de comunicación o de conexiones a bases de datos por ejemplo. Eso hace que en Java, que tiene un recolector de basura que se encarga de liberar automáticamente los objetos no referenciados, los destructores pasen a estar en segundo plano y sea muy raro verlos en el código. En C++ al no existir recolección de basura, el peso de liberar toda la memoria solicitada cae en el lado del programador y por lo tanto la mayoría de las veces tenemos que crear destructores que se encarguen de ello.

La forma de implementar los destructores varía entre los dos lenguajes:

- En Java los destructores son métodos con la siguiente *signatura*:

  ```
  protected void finalize() throws Throwable
  ```

- En C++ el destructor se llama igual que la clase pero antecediéndolo con el carácter ~.

> Al contrario que con los constructores, tanto en Java como en C++, no se invoca a los destructores explícitamente.

Como ejemplo típico, veamos una clase que tiene como atributo un vector de enteros implementado como un array de enteros; el tamaño del array, si no se indica lo contrario,

es 10, pero el usuario de la clase puede dar otro tamaño utilizando para ello un constructor con un parámetro entero.

El código en Java podría ser:

```
public class EjemploDeClase {

  // El static quedará explicado más adelante en el tema
  public static final int TAM_POR_DEFECTO = 10;

  private int []vectorEnteros;

  public EjemploDeClase() {
    this.vectorEnteros = new int[TAM_POR_DEFECTO];
  }

  public EjemploDeClase(int tam){
    this.vectorEnteros = new int[tam];
  }

  protected void finalize() throws Throwable{
    // A pesar de haber pedido memoria para el atributo, en
    // Java no hace falta liberarlo, pues ya lo hará el
    // recolector de basura.
  }
}
```

Esto también se puede reescribir inicializando el atributo por defecto:

```
public class EjemploDeClase {

  public static final int TAM_POR_DEFECTO = 10;

  private int[] vectorEnteros=new int[TAM_POR_DEFECTO];

  public EjemploDeClase() {
  }

  public EjemploDeClase(int tam){
    // El array del tamaño por defecto se pierde
    this.vectorEnteros = new int[tam];
  }

  protected void finalize() throws Throwable{
    // A pesar de haber pedido memoria para el atributo, en
    // Java no hace falta liberarlo, pues ya lo hará el
    // recolector de basura.
  }
}
```

Como vemos, incluso aunque la clase haya pedido memoria para guardar información el destructor en Java ha quedado vacío. Esa situación será la tónica general; la definición anterior es equivalente a:

```
public class EjemploDeClase {

  public static final int TAM_POR_DEFECTO = 10;
```

```
  private int[] vectorEnteros=new int[TAM_POR_DEFECTO] ;

  public EjemploDeClase() {}

  public EjemploDeClase(int tam){
    this.vectorEnteros = new int[tam];
  }
}
```

El mismo ejemplo escrito en C++ quedaría así:

```
class EjemploDeClase {
  public:

  EjemploDeClase(){
    this->vectorEnteros = new int[TAM_POR_DEFECTO];
  }

  EjemploDeClase(int tam){
    this->vectorEnteros = new int[tam];
  }

  ~EjemploDeClase(){
    // Liberamos la memoria pedida
    delete []vectorEnteros;
  }

  private:

  // El array dinámico lo implementamos con punteros
  int *vectorEnteros;

  static const int TAM_POR_DEFECTO; // El valor 10 se da en el .cpp...
}
```

Como no hay recolector de basura, en este caso debemos encargarnos de eliminar todos aquellos recursos que hayamos pedido. Igual que ocurre en Java, si la clase no ha solicitado recursos y por tanto la implementación es vacía, se puede omitir.

Los constructores, como los métodos y atributos, pueden tener asociado una etiqueta que informe sobre su visibilidad. Para más información al respecto, consultar la sección 2.

Objetos dentro de objetos

Una clase puede declarar que sus instancias puedan contener otras instancias. Ello se consigue haciendo que los atributos definidos sean también otras clases. En tales casos, la inicialización de la clase en el constructor debe hacer referencia a los constructores de las clases agregadas. La contención puede ser por valor o por referencia y será estudiada en la secciones 6.1 y 6.2.

```
public class OtraClase {

 public OtraClase(){...};

 ...
}
```

```
...

public class EjemploDeClase {
  private OtraClase atributo;

  public EjemploDeClase(){
    this.atributo=new OtraClase(); // asigna una instancia de OtraClase
  }

  public EjemploDeClase(OtraClase nuevoValor){
    this.atributo=nuevoValor; // asigna el valor pasado
                              //como parámetro al atributo
  }

  public static void main(String args[]){
  // crea un objeto con el atributo conteniendo una instancia de OtraClase
   EjemploDeClase unaVariable=new EjemploDeClase();
   // crea un objeto con el atributo conteniendo una instancia de OtraClase
   EjemploDeClase otraVariable=new EjemploDeClase(new OtraClase());
  }
}
```

Como con otros atributos, los atributos que contendrán objetos deben inicializarse igualmente, llamando a sus constructores:

- En Java utilizando `new` o bien en la declaración del atributo o bien en el código del constructor. Si no se inicializan, contendrán por defecto el valor `null` por lo que no representan un objeto válido.

- En C++ si se utiliza un tipo valor (sección 6.1) el compilador llama automáticamente al constructor sin parámetros (si éste no existe, el programador deberá llamar a uno explícitamente en la lista de inicializadores del constructor). Si el atributo es un puntero, deberá pedirse memoria explícitamente (igual que se vió en el ejemplo del vector de enteros anteriormente); de no hacerse el puntero nos llevaría a una dirección aleatoria.

Podemos preguntarnos si tiene sentido que una clase se contenga a sí misma, es decir, tenga un atributo de la misma clase. Si lo pensamos con cuidado eso implicaría que un objeto de la clase tendría que tener como atributo un objeto de la misma clase que, a su vez, tendría otro objeto de la misma clase, etc, llegando a un bucle infinito.

Por lo tanto, la respuesta rápida es que *no* se puede: ¿cuánto ocuparía en memoria una instancia de la clase? Esa es la respuesta correcta en C++: no podemos tener dentro de una clase un atributo de la propia clase. Sin embargo en Java, al utilizar *tipos referencia* (y en C++ si utilizamos punteros), *sí* es posible aunque en la mayoría de los casos eso lleva, a largo plazo, a problemas. Dado el código siguiente en Java:

```
public class EjemploDeClase {
  private EjemploDeClase atributo;
  public EjemploDeClase(){
   // qué se pondría aquí?
   ...
  }
}
```

Se encuentra uno con que en el constructor hay que inicializar el atributo, pero para hacerlo, habría que invocar al constructor, otra vez. Se llegaría a un bucle infinito. Para romperlo, no queda más remedio que asignarle el valor *null*, o bien no asignarle nada (sabiendo que Java le pondrá el valor null por defecto), como en el ejemplo siguiente:

```
public class EjemploDeClase {
  private EjemploDeClase atributo;
  public EjemploDeClase(){
    atributo=null;
  }
}
```

Aunque este tipo de construcciones es habitual encontrarlas en las librerías que construyen estructuras de datos (para tener listas enlazadas, por ejemplo) deben utilizarse con cautela. Si bien en esos escenarios concretos es perfectamente justificable su uso, en otros escenarios es inadmisible. Cuando se consideran programas de mayor tamaño, se complica por momentos, haciendo el código más difícil de entender y mantener. Por ello, en la medida de lo posible, y salvo que los profesores lo indiquen, se debe cumplir lo siguiente:

> En la declaración de una clase, no puede haber atributos que, directa o indirectamente, hagan mención a la clase.

Siempre hay una forma de evitar este tipo de dependencias cíclicas.

Métodos

Los métodos son operaciones que se declaran en una clase y que operan sobre los atributos singulares que existirán en los objetos. Aunque hay métodos asociados a una clase, ver la sección 5, aquí se verán sólo aquellos que trabajan con los atributos de un objeto.

Un método contiene operaciones a realizar y puede devolver un resultado, o no, y tener parámetros, o no. Una declaración de método en Java tiene la siguiente forma:

```
public class EjemploDeClase {

    public TipoADevolver nombreDelMetodo1(
                            TipoDeParametro1 nombreParametro1,
                            TipoDeParametro2 nombreParametro2){
      TipoADevolver resultado;

      // código del método que inicializa la variable resultado

      return resultado
     }

    public void nombreDelMetodo2(
                            TipoDeParametro1 nombreParametro1,
                            TipoDeParametro2 nombreParametro2){

     // código del método. No hay que devolver nada
    }
}
```

En Java *no* es posible indicar si el parámetro es de entrada, salida o entrada/salida. Aunque detallaremos más adelante esto en las secciones 6.2 y 6.1 (y de hecho en cierto modo contradeciremos lo que decimos aquí), en Java los tipos primitivos (`boolean`, `int`, etc.) se pasan siempre por valor, y los tipos que son clases y arrays se pasan siempre por referencia.

En C++ la declaración de los métodos es similar a Java, con la particularidad de que pueden declararse métodos `const` para indicar que en la implementación *no se cambia* el `this`.

Por su parte, la definición/implementación puede hacerse:

- En la propia declaración de la clase, de forma similar a como se hace en Java. El compilador considerará el método *inline*.
- En el propio fichero de cabecera fuera de la declaración de la clase; para eso habrá que indicar explícitamente en la declaración que el método es `inline`.
- En un fichero distinto de implementación (normalmente extensión `.cpp`). Tanto en este caso como en el anterior el nombre del método habrá que antecederlo por el nombre de la clase seguida de `::`.

En el ejemplo siguiente utilizamos las dos primeras alternativas:

```
class EjemploDeClase {

  public:

    TipoADevolver nombreDelMetodo1(TipoDeParametro1 nombreParametro1,
                                   TipoDeParametro1 nombreParametro2) {

     TipoADevolver resultado;
     // código del método que inicializa la variable resultado
     return resultado
    }

    inline void nombreDelMetodo2(TipoDeParametro1 nombreParametro1,
                                 TipoDeParametro1 nombreParametro2);
};

inline void EjemploDeClase::nombreDelMetodo2(
                                 TipoDeParametro1 nombreParametro1,
                                 TipoDeParametro1 nombreParametro2) {
   // Implementación que no devuelve nada.
}
```

Los métodos tienen asociada una visibilidad, que se verá en la sección 2.

Hay que recordar que la convención de código acordada asume que el nombre de los métodos se corresponderá con un verbo que esté relacionado con la acción que se pretenda realizar.

Un método puede usar los atributos del objeto y variables creadas dentro del método.

Para ejecutar un método se parte de una variable que tiene el objeto sobre el que queremos ejecutar el método. En la nomenclatura habitual de POO se dice que lo que hacemos es "enviar un mensaje al objeto para que ejecute el método".

El operador de invocación (o envío del mensaje de invocación) es el `.` seguido del nombre del método que se quiere ejecutar. En C++ si la variable que manejamos es un *puntero a una clase* en vez de `.` se utiliza `->`.

A continuación aparece varios ejemplos de invocaciones en Java:

```
public class EjemploDeClase {
    ....
    public static void main(String args[]){
     EjemploDeClase ejemplo=new EjemploDeClase();
     TipoDeParametro1 nombreParametro1;
     ...
     TipoDeParametro2 nombreParametro2;
     ...
     // invoca y recoge el resultado
     TipoADevolver resultado= ejemplo.nombreDelMetodo1(
       nombreParametro1, nombreParametro2);
     // sólo invoca
     ejemplo.nombreDelMetodo2(nombreParametro1,
       nombreParametro2);
    }
}
```

Y en C++ (se ilustran dos formas de llamar al objeto, cuando manejamos objetos y cuando tenemos punteros a objetos):

```
int main() {
 EjemploDeClase ejemplo();
 TipoDeParametro1 nombreParametro1;
 ...
 TipoDeParametro2 nombreParametro2;
 ...
 // invoca y recoge el resultado
 TipoADevolver resultado= ejemplo.nombreDelMetodo1(
      nombreParametro1, nombreParametro2);
 // sólo invoca
 ejemplo.nombreDelMetodo2(nombreParametro1, nombreParametro2);

 EjemploDeClase* ejemplo2=new EjemploDeClase();
 // invoca y recoge el resultado
 TipoADevolver resultado= ejemplo2->nombreDelMetodo1(
      nombreParametro1, nombreParametro2);
 // sólo invoca
 ejemplo2->nombreDelMetodo2(nombreParametro1, nombreParametro2);
}
```

En la implementación de un método se puede llamar a otros métodos de la misma clase de forma similar a cómo accedemos a los atributos, utilizando el `this`.

Puede haber varios métodos que, teniendo el mismo tipo de retorno (o no teniendo ningún tipo de retorno), difieran en el número y tipo de los parámetros. En tales casos, se dirá que se tiene una *sobrecarga* de métodos. El compilador, en base a los parámetros reales utilizados en la invocación, decidirá a cuál de ellos llamar.

Podemos catalogar los métodos según su funcionalidad en los siguientes:

- Métodos inicializadores. Inicializan atributos y no se trata de constructores. Tienen interés cuando se quiere reinicializar los valores de los atributos sin invocar al cons-

tructor. Una vez invocado el constructor, no puede volver a ser invocada sin conllevar la creación de un nuevo objeto[4]. Por ello, si se quiere conservar el objeto y volver a los valores iniciales, no hay otra opción que encapsular el código de creación de la clase en un método. Este método debería ser reutilizado en los constructores, si las hubiere y si el constructor tuviese el mismo efecto de inicialización que el método en cuestión.

- Métodos accedentes. Devuelven el contenido de los atributos (cada accedente devuelve un atributo). Se conocen también como métodos *get*, pues se suelen tener como signatura la siguiente.

```
TipoDeAtributo getNombreDeAtributo()
```

  Se utilizan cuando queremos que el *usuario* de la clase tenga acceso a los valores de los atributos pues, como ya hemos dicho, los atributos deberían ser siempre declarados como no públicos.

  Este tipo de métodos hay que usarlos con cautela cuando el tipo del atributo es una clase. Si se devuelve el objeto, dependiendo de las circunstancias, se puede estar devolviendo una referencia al valor interno que guarda el objeto y que se suponía debía proteger. En casos sensibles, puede ser necesario copiar el objeto antes de devolverlo.

- Métodos mutadores. Establecen el contenido de los atributos (cada mutador, el contenido de un atributo). Se conocen también como métodos *set*, pues se suelen tener como signatura la siguiente.

```
void setNombreDeAtributo(TipoDeAtributo nuevoValor)
```

  De forma similar a los métodos accedentes, en este caso permiten al usuario cambiar el valor de un atributo a pesar de que ese atributo no sea público.

  El cambio en el atributo interno debe hacerse también con precaución, sobre todo en el caso de C++, cuando el valor a introducir se trata de un puntero. En ese caso, hay que establecer claramente si la clase *pasa a ser la propietaria del valor apuntado*, y por tanto si debe borrarlo en su destructor. En caso de no haber una transferencia de propiedad habrá que dejar claro cuál se espera que sea el tiempo de vida de ese objeto: si el invocante piensa eliminar el objeto inmediatamente entonces es posible que la clase necesite hacer una copia del mismo.

- Métodos computadores. Realizan cálculos y generan resultados. Uno de estos cálculos puede implicar convertir el objeto en una representación distinta, como la conversión de un objeto en una cadena. En Java, todas las clases tienen un método heredado[5] con esta signatura:

```
public String toString()
```

[4] Hay un caso especial que consiste en la llamada a un constructor de la clase padre dentro de un constructor de una clase hija, que veremos cuando tengamos herencia en un tema posterior.

[5] La herencia se verá en el capítulo siguiente.

Se asume que convierte el objeto entero en una cadena que puede ser mostrada por pantalla o usada para otros fines. Este es el método que se invoca al hacer un *System.out.println* pasándole como parámetro un objeto o bien cuando se concatena un objeto con una cadena. Por defecto, este método devuelve como valor el nombre de la clase, seguido de una arroba y el resultado de invocar al método *hashcode*, que también es un método heredado presente en todas las clases. Este método se puede sobrescribir como se verá en el capítulo siguiente.

Visibilidad

La encapsulación, una de las características principales de la OO, significa "meter en cápsulas", de forma que lo que guardamos queda protegido del exterior. Cuando creamos una clase, ésta "protege" los elementos que metemos en ella, métodos y atributos, de forma que oculta hacia fuera su funcionamiento interno (*information* o *data hiding*).

En una primera aproximación, podemos decir que una clase tiene *dos* tipos de usuarios:

- Los usuarios *externos* a la clase: aquellos que crean instancias (objetos) de la clase y quieren invocar a sus métodos o acceder a sus atributos.
- Los usuarios *internos* a la clase: o, lo que es lo mismo, la implementación de los métodos.

Como ejemplo, pensemos en una clase `Fecha` que tenga tres atributos enteros (`dia`, `mes`, `anyo`) y una colección de métodos como `escribe` o `resta`. Un usuario *externo* es el programador que, implementando el `main` de una aplicación, crea un objeto de la clase y llama a uno de sus métodos. El usuario *interno* es el programador del método `escribe` que terminará invocando a los `System.out` necesarios.

Esa división nos lleva a considerar dos categorías de visibilidad: visibilidad *pública* y visibilidad *privada*. Los elementos de una clase declarados *públicos* (`public`) podrán ser utilizados tanto por los usuarios externos como por los internos. Los elementos declarados como *privados* (`private`) serán sólo accesibles a los usuarios internos.

En Java se indica la visiblidad de cada uno de los elementos antecediendo a su declaración las palabras reservadas `public` y `private`; en C++ la declaración de la clase se divide en distintas secciones, marcando el comienzo de cada una con `public:` y `private:`.

Ejemplo en Java:

```
public class Fecha {

   ...

   // Método público; pueden utilizarlo los
   // usuarios externos.
   public void escribe() {
      // Implementación: usuario "interno".
      // Puede acceder a los elementos privados
      System.out.println("Día: " + this.dia);
      ...
   }

   public int resta(Fecha fecha) {
      // Implementación: usuario "interno".
      // Puede acceder a los elementos privados,
```

```
        // no sólo del parámetro this, sino también
        // del objeto fecha recibido como parámetro
        // (por ser de la misma clase).
        // Es decir, tanto a this.dia como a fecha.dia.
        ...
    }

    private int dia;
    private int mes;
    private int anyo;
}
```

Y su equivalencia en C++:

```
class Fecha {
public:
    ...

    // Método público; pueden utilizarlo los
    // usuarios externos.
    void escribe() {
        // Implementación: usuario "interno".
        // Puede acceder a los elementos privados
        std::cout << "Día: " + this->dia;
        ...
    }

    int resta(Fecha fecha) {
        // Implementación: usuario "interno".
        // Puede acceder a los elementos privados,
        // no sólo del parámetro this, sino también
        // del objeto fecha recibido como parámetro
        // (por ser de la misma clase).
        // Es decir, tanto a this->dia como a fecha.dia.
        ...
    }

private:
    int dia;
    int mes;
    int anyo;
};
```

En Java en realidad la división anterior de usuarios *internos* y *externos* puede subdividirse un poco más:

- Usuarios externos que *pertenecen al mismo paquete*.
- Usuarios externos que *no pertenecen al mismo paquete*.

Dado que los paquetes vienen a representar agrupaciones de clases relacionadas, puede tener sentido darles un tratamiento especial como usuarios de otras clases del mismo paquete. En este caso Java permite darles ciertos privilegios de acceso a los elementos que no están accesibles para los usuarios (clases) situados en otros paquetes.

En concreto, en Java se puede *omitir* el modificador de visilibidad (no poner ni `public` ni `private`). En ese caso:

- El elemento es accesible para los usuarios *internos* (implementación de la propia clase).
- El elemento es accesible ("público") para los usuarios externos del mismo paquete (para las clases del paquete).
- El elemento *no* es accesible para el resto (clases de otros paquetes).

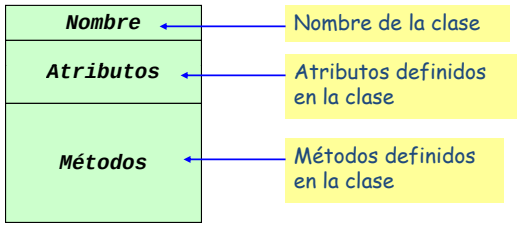
Dado que en C++ no existen reglas de visibilidad entre namespaces, se puede conseguir algo parecido utilizando las clases "amigas" (`friend`), que permiten al programador de una clase indicar al compilador el nombre de otras clases que podrán acceder (desde "fuera") a elementos declarados como no públicos.

Representación gráfica de las clases

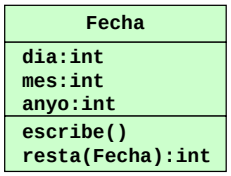
Es útil poder representar gráficamente los elementos de los programas orientados a objetos y sus relaciones. Para hacerlo, utilizaremos diagramas similares a los de UML (lenguaje unificado de modelado o *Unified Modeling Language*) que se utilizan en Ingeniería del Software.

A medida que vayamos estudiando los conceptos y los mecanismos de la POO iremos viendo cómo representarlos gráficamente.

De momento, comencemos con la representación de las clases. Se utiliza un rectángulo dividido en tres partes: uno que contiene el nombre de la clase, otro que contiene los atributos y un tercero para el nombre de los métodos:

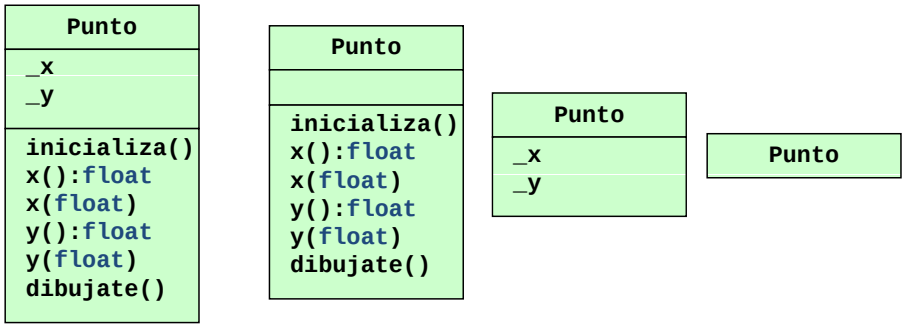


Un ejemplo para la clase `Fecha` (con únicamente los métodos `escribe` y `resta`) sería:

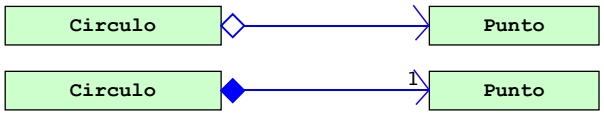


Para los atributos indicamos su tipo tras su nombre separado de `:`. Para los métodos utilizamos el nombre seguido de los tipos de los parámetros y si devuelve algo utilizamos `:` seguido del tipo devuelto.

A veces, dependiendo del objetivo del diagrama, no interesa mostrar tantos detalles, por lo que se pueden dejar secciones en blanco. Por ejemplo, para mostrar la forma en que se usan los objetos de una clase, la parte de atributos no es relevante, por lo que la podemos dejar en blanco. Si lo que se quiere es omitir la información sobre métodos, entonces se puede dejar en blanco o se puede quitar del todo. Se pueden incluso quitar las dos secciones de atributos y métodos, dejando simplemente el nombre de la clase; es el menor nivel de detalle.



También se pueden representar gráficamente las relaciones entre clases. La única que conocemos hasta ahora es la de *clientelismo*: cuando una clase contiene objetos de otra. Por ejemplo, si tenemos una clase *Circulo* que tiene atributos de la clase *Punto* (para guardar la posición del centro del círculo), conectamos una clase con otra mediante una línea con un rombo en el extremo de la clase que contiene objetos de la otra. Si queremos dar más detalle en el propio diagrama podemos indicar el número de objetos que contiene:



Igualdad entre objetos

Cuando manejamos objetos tiene sentido preguntarnos si dos objetos son iguales. Aunque veremos más detalles de esto más adelante, en Java se manejan siempre *referencias* a objetos, por lo que ante un código como el siguiente:

```
Fecha f1;
Fecha f2;

// ...Inicialización de f1 y f2...
```

podemos hacernos dos preguntas:

- ¿Son `f1` y `f2` *iguales*? (Es decir, ¿representan los dos objetos el mismo día?).
- ¿Son `f1` y `f2` *el mismo objeto*? (Es decir, si cambio el día al que equivale *f2*, ¿se cambia también el de `f1`?)

En Java el operador `==` sobre objetos contesta a la *segunda* pregunta, algo que normalmente *no* queremos. Si queremos saber si dos objetos son iguales debemos utilizar el método `equals`:

```
if (f1.equals(f2))
   System.out.println("Nacieron el mismo día!");
```

Por razones obvias, se cumple que cuando dos objetos son *el mismo*, también son iguales[6].

[6] En la práctica, no obstante, Java *no* obliga a que esto sea así. Un programador podría implementar el método `equals` de forma que siempre devuelva `false` y la propiedad anterior dejaría de ser cierta. Evidentemente esto no debería hacerse nunca, pues `equals` dejaría de funcionar como se espera.

Importante: Gran parte de las clases de la librería de Java implementan el método `equals`. Cuando creamos una clase, Java añade automáticamente una implementación por defecto del método para ella. Sin embargo es bastante probable que no sea válida.

Clonación de objetos

Por razones similares a las del apartado anterior, la *asignación* entre objetos en Java en realidad es una asignación de *referencias*. Siguiendo con el ejemplo de las fechas:

```
Fecha f1, f2;

// ... inicialización ...

f2 = f1; // Asignación de referencias

if (f1 == f2) // Esto será cierto
   System.out.println("El mismo");
```

Si lo que queremos es tener dos objetos iguales pero independientes debemos *copiar* o *clonar* el objeto original. Para eso existe el método `clone`, que crea un nuevo objeto estructuralmente igual al original:

```
Fecha f1, f2;

// ... inicialización ...

f2 = f1.clone(); // Copia del objeto

if (f1 == f2) // Esto NO será cierto
   System.out.println("El mismo");

if (f1.equals(f2)) // Esto sí
   System.out.println("Iguales");
```

Importante: Igual que ocurre con el método `equals`, gran parte de las clases de la librería de Java implementan el método `clone`. Cuando creamos una clase, Java añade automáticamente una implementación por defecto del método para ella. Sin embargo es bastante probable que no sea válida.

Atributos y métodos de clase (estáticos)

Los métodos y atributos vistos hasta ahora se suelen llamar métodos y atributos "de instancia" pues están asociados a una instancia de la clase (un objeto):

- Cada instancia/objeto tiene un conjunto de atributos (con el mismo nombre), de forma que cada objeto puede guardar en ellos valores distintos.
- Los métodos trabajan con una instancia/objeto particular (sobre el que se invoca el método), y pueden acceder a sus atributos utilizando el `this` (que, como hemos dicho, se puede omitir).

En contraposición a estos métodos y atributos, los lenguajes OO soportan también métodos y atributos "de clase" (llamados muchas veces "estáticos"):

- En un atributo de clase *sólo hay una copia* para *todas las instancias* de la clase. Es un dato único que se encuentra fuera de todos los objetos de la clase, y que puede ser accedido por todos ellos. Su valor es compartido por todos.
- Un método de clase *no* necesita para su ejecución un objeto al que acceder. Eso automáticamente implica que *no* podrá acceder a atributos de instancia (pues no tiene ningún objeto), sino únicamente a atributos de clase. Tampoco podrá invocar ningún método de instancia.

Los atributos y métodos de clase se marcan utilizando la palabra reservada `static` (tanto en Java como en C++). Igual que para el resto de métodos y atributos, el programador puede indicar la visibilidad que desea para ellos.

Como ejemplo, imaginemos que queremos llevar la cuenta de cuántos objetos de una clase se han construido. Podemos almacenar en un atributo estático un entero que comience en cero y que iremos incrementando en el constructor. Esto puede ser útil, por ejemplo, para una clase que representa una cuenta bancaria; el atributo estático puede interpretarse como el siguiente código de cuenta a utilizar (de forma que se garantiza que no habrá dos cuentas con el mismo número):

```
public class Cuenta {

   /** Constructor sin parámetros de una cuenta.*/
   public Cuenta() {
      cc = siguienteId;
      ++siguienteId;
   }

   ...

   /** Siguiente código de cuenta a utilizar. */
   private static int siguienteId = 0;

   /** Código de la cuenta. */
   private int cc;
}
```

Y en C++ (incluimos la implementación del constructor *inline*)[7]:

```
class Cuenta {
public:

   /// Constructor sin parámetros
   Cuenta() {
      cc = siguienteId;
      siguienteId++;
   }

   ...

private:

   /// Código de la cuenta
   int cc;
```

[7]En realidad en C++ hay que realizar un paso más, definiendo el atributo en el .cpp para reservar su espacio de memoria e inicializarlo.

```
    /// Siguiente id a utilizar
    static int siguienteId;
}
```

Los métodos estáticos son útiles, entre otras cosas, para la implementación de subprogramas que no necesitan ningún dato adicional más allá de lo que reciben por parámetro. Un uso habitual es para funciones matemáticas[8]. Por ejemplo una clase podría implementar las funciones trigonométricas recibiendo por parámetro el ángulo y devolviendo el valor. Esas implementaciones no necesitan acceder a ningún otro atributo, por lo que los métodos son estáticos.

De hecho, en estas clases con funciones matemáticas suele haber también atributos estáticos para las constantes como π o e[9]:

```
public class Math {

  public static final double PI = ...;
  public static final double E = ...;

  public static double cos(double angleInRad) { ... }

  public static double sin(double angleInRad) { ... }

}
```

y en C++:

```
class Math {
public:

  static const double PI;
  static const double E;

  static double cos(double angleInRad);

  static double sin(double angleInRad);
}
```

Para utilizar métodos y atributos estáticos *no* debe crearse un objeto, sino que se utiliza el nombre de la clase en el lugar en el que pondríamos el objeto.

```
// En Java:
Math.PI;
Math.sin(0);

// En C++:
Math::PI;
Math::sin(0);
```

[8] Ya hemos visto otro ejemplo de método estático en Java: el método `main`.

[9] La declaración siguiente de una supuesta clase `Math` contiene en realidad parte de la declaración de la clase `java.lang.Math` disponible en la librería de Java.

Tipos de tipos de datos

Utilizando la nomenclatura que se usa en la documentación de C# podemos decir que en lenguajes como Java y C# existen dos categorías de tipos de datos:

- Los *tipos valor* (*value-type*): las variables de estos tipos contienen directamente sus datos.
- Los *tipos referencia* (*reference-type*): las variables de estos tipos almacenan *referencias* a sus datos (que son normalmente *objetos*).

Tipos-valor

En Java los únicos tipos de datos de esta categoría son los conocidos como *tipos primitivos*: `char`, `boolean`, `int`, etc.[10]

Las variables de estos tipos almacenan directamente el valor al que representan. Como ejemplo, si en Java nos encontramos

```
long a;
```

la variable `a` representa directamente un valor de tipo entero largo.

Las variables de estos tipos-valor *no* pueden asignarse a `null`, y sus asignaciones *copian* los valores. Es decir, cada variable tiene su propia copia de los datos, por lo que las operaciones sobre una variable no afectarán nunca a los valores de otra:

```
long a = 7.0;
long b;

b = null; // Error
b = a; // b == 7.0
a = 2.0; // a == 2.0, b == 7.0
```

El operador de comparación (`==`) comprueba si los datos que guardan las variables son o no son iguales.

En Java estos tipos *no* se consideran objetos y por lo tanto tampoco tienen métodos[11], por lo que no podremos llamar al método `equals` que hacía la comparación de valores.

Los datos de estos tipos son destruidos cuando sus variables salen de ámbito. Si utilizamos una variable local de este tipo, su valor se almacena directamente en la pila y es eliminado de ella al terminar la función (o sección de código) en la que se encuentra. Cuando se utilizan como atributos hacen incrementar el tamaño en memoria del objeto en el tamaño de ese valor, y el espacio se mantiene durante toda la vida del objeto. Cuando el objeto es eliminado, automáticamente desaparece el espacio ocupado por ese dato.

Tipos-referencia

En Java *todas* las variables de clases y los arrays son tipos referencia. Como su nombre indica las variables de estos tipos *referencian* a una instancia del tipo al que pertenecen. Así por ejemplo:

```
Fecha f;
```

[10] En C# el programador puede crear otros tipos-valor distintos.
[11] En C# estos tipos sí pueden tener métodos.

la variable `f` permite almacenar una *referencia* a una instancia/objeto de la clase `Fecha`.

Para indicar que una variable de este tipo *no* referencia a ningún objeto se utiliza `null`; si se intenta acceder al valor referenciado, se generará un error en tiempo de ejecución.

Al contrario que los tipos-valor, la asignación entre dos variables en este caso copia *las referencias*, por lo que es posible que dos variables distintas representen al mismo objeto, lo que automáticamente implica que una operación realizada sobre una de las variables afecta al objeto referenciado por otra.

```
Fecha f = new Fecha(12, 10, 1492);
Fecha f2;

// ... inicializaciones varias ...

f2 = null; // OK
f2 = f; // f y f2 referencian lo mismo.
f.avanzaUnDia(); // Cambia también la instancia manejada con f2
f.getDia() == f2.getDia(); // true
```

El operador de comparación (`==`) comprueba si las dos variables *referencian* al mismo objeto (son el mismo), por lo que puede que dos objetos sean *iguales* pero `==` devuelva `false`.

La mera declaración de estas variables *no* construye instancias de clases. Para eso será necesario utilizar el `new` ya explicado.

En Java y C# los objetos referenciados por variables de este tipo *no* son destruidos cuando sus variables salen de ámbito, pues esas instancias pueden ser referenciadas por otras variables. Es el recolector de basura el encargado de su destrucción cuando ninguna variable los referencie.

Si utilizamos una variable local de este tipo, el objeto *no* se almacena directamente en la pila, por lo que el tamaño de ésta se ve incrementado únicamente en el tamaño de una referencia. Cuando se utilizan como atributos incrementan el tamaño ocupado por una instancia de esa clase en una referencia. Cuando el objeto es eliminado, la instancia a la que representa *no* se elimina automáticamente pues, igual que antes, podría estar referenciado por otras variables locales o atributos.

Tanto en Java como en C#, los arrays son tipos-referencia, por lo que deben construirse con `new` antes de utilizarse y la asignación de un array a otro hace que ambas variables trabajen sobre el mismo array, por lo que el cambio en uno de ellos afecta al otro:

```
int []v1 = new int[10];
int []v2;

v2 = v1;
v1[0] = 7; // v2[0] == 7 también
```

La construcción de un array bidimensional, por tanto, es en realidad un array de *referencias a arrays unidimensionales*. De ahí que haya que crear cada uno de forma individual:

```
int [][]m;

m = new int[3][];

m[0] = new int[2];
m[1] = new int[2];
m[2] = new int[2];
```

```
// La construcción m = new int[3][2]; es
// equivalente a lo anterior.

// Nada nos impide hacer algo como.
m[0] = new int[2];
m[1] = m[0];         // m[1] es EL MISMO array que m[0];
m[2] = new int[15]; // cambios en m[1] afectan a m[0]
```

Paso de parámetros a métodos

En Java los parámetros en las invocaciones a métodos se realiza *siempre* por valor. Es decir, los parámetros son *siempre* de entrada; de hecho, *no* se pueden pasar parámetros como entrada/salida, algo que sí puede hacerse en C#. Sin embargo, la naturaleza de los tipos-referencia hace que las operaciones realizadas sobre las variables de estos tipos dentro del método afecten a las instancias apuntadas por las variables de fuera, por lo que en la práctica podemos hablar de parámetros de entrada/salida cuando utilizamos tipos referencia: un cambio sobre el objeto o dato al que representa el parámetro es visible fuera, pero no lo es el cambio de la propia referencia (hacer que "apunte" a otro objeto).

Ejemplo: paso de parámetros de tipo-valor

```
// Este método NO funciona: los parámetros son
// de entrada.
void swap(int a, int b) {
  int aux;
  aux = a;
  a = b;
  b = aux;
}
```

Ejemplo: paso de parámetros de tipo-referencia. Los cambios sobre la instancia dentro del método son visibles fuera.

```
void hazAlgo(Fecha f) {
   f.avanzaUnDia();
}

... Y lo usamos ...

Fecha vble = new Fecha(12, 10, 1492);

hazAlgo(vble);

// En este momento, vble.getDia() devolverá 13,
// pues hazAlgo al modificar la instancia
// referenciada por f modificó también la
// referenciada por vble.
```

Ejemplo: paso de parámetros de tipo-referencia. La instancia puede cambiar, pero el valor referenciado no (paso *por valor*):

```
void hazAlgo(Fecha f) {
   f = new Fecha(1, 1, 2000);
}
```

```
... Y lo usamos ...

Fecha vble = new Fecha(12, 10, 1492);

hazAlgo(vble);

// En este momento, vble sigue haciendo referencia
// a la fecha 12/10/1492. La función cambió la
// instancia a la que hacía referencia la variable
// f, pero NO la instancia a la que referenciaba
// vble.
```

Boxing-unboxing

En Java para cada tipo primitivo (y por tanto para cada tipo-valor), existe una clase que almacena uno de esos valores. Así, tenemos `java.lang.Integer` para almacenar un `int`, `java.lang.Float` para `float`, etc.

El lenguaje hace automáticamente conversiones de unos a otros, algo conocido como *boxing* y *unboxing*, utilizando la metáfora de que cuando un `int` es convertido a un `Integer` se "mete en una caja" (la instancia de `Integer`) y cuando se convierte de `Integer` a `int`, se "saca de la caja". El proceso de *boxing* implica la creación de un objeto, que se realiza de forma transparente al programador.

```
Integer a;
int i = 0;

// Boxing
a = 3; // Equivalente a a = new Integer(3);
a = i; // o a = new Integer(i);

// Unboxing
i = a; // Equivalente a i = a.intValue();
```

Esta característica será especialmente útil cuando trabajemos con colecciones genéricas.

Hay que hacer notar que los objetos de las clases `Integer`, etc. son *inmutables*: una vez construido el objeto *no* puede cambiarse el valor entero que almacenan. Eso impide utilizar estas clases para pasar tipos primitivos como parámetros de entrada/salida.

Tipos en C++

En C++ todos los tipos son tipos-valor. Todas las variables almacenan directamente los valores a los que representan[12]. Entre las implicaciones de esto tenemos que:

- Si declaramos una variable local de una clase, el espacio reservado en la pila será el tamaño completo de ese objeto.
- Si una clase tiene un atributo que pertenece a otra clase, el tamaño de sus instancias se incrementa en base a los atributos de esa segunda clase.

[12]En realidad, los arrays de C++, heredados de C, tienen un comportamiento peculiar, que hace que unas veces se comporten como tipos valor y otras veces como tipos referencia; ignoraremos los arrays en esta discusión.

- Si declaramos una variable local de una clase, la instancia a la que representa esa variable será destruida cuando ésta salga de ámbito (al final de la función o de la sección de código). Eso provocará la llamada a su destructor.
- Si una clase `A` tiene un atributo de otra clase `B`, cuando se construye el objeto de la clase `A` se crea automáticamente el objeto de la clase `B` (y llama a su constructor). Cuando el objeto de la clase `A` se destruye, se destruye automáticamente el objeto de la clase `B` (y llama al destructor).
- El operador de comparación, `==`, entre dos instancias de una clase debe devolver `true` si los dos objetos sin *iguales* (no si son *el mismo*)[13].
- El operador de asignación, `=`, hace una *copia* de un objeto a otro.

A continuación aparecen algunos ejemplos de esto, asumiendo la existencia de la clase `Fecha`:

```
void func1() {
   Fecha f; // Provoca la llamada al constructor de Fecha
            // En la pila se reservan 3*sizeof(int) bytes.

   ...
} // Al final se llamará al destructor.

void func2() {

  Fecha f1(12, 10, 1492);
  Fecha f2(1, 1, 2000);

  f1 == f2; // Esto evalúa a false; representan días distintos

  f2 = f1; // Copiamos el valor de f1 en f2.
           // Pero son instancias distintas.

  f1 == f2; // Esto evalúa a true

  f1.avanzaUnDia();  // Cambiamos f1

  f1 == f2; // Ahora evalúa a false; f2 no cambia.
}
```

Es el programador el que, si quiere manejar tipos referencia, debe hacerlo explícitamente manejando o bien punteros o bien referencias. En ese caso:

- Igual que ocurre en Java y C#, tendrá que crear explícitamente las instancias con `new`.
- Tendrá que encargarse de liberarlas con `delete`.
- Si utiliza punteros, para invocar a los métodos tendrá que utilizar el operador `->`.
- El `==` con punteros tendrá el mismo significado que en Java (`true` si es *el mismo* objeto).

[13] Tanto en éste como en el caso siguiente para que esto funcione correctamente se requiere, a menudo, la colaboración del programador que creó la clase.

- El equivalente al `equals` se consigue haciendo el `==` sobre *los contenidos* de los punteros (operador `*`).

Veamos los ejemplos anteriores utilizando punteros:

```
void func1() {
   Fecha *f; // Variable sin inicializar.

   f = NULL; // Puntero que no hace referencia a ninguna instancia.

   f = new Fecha(12, 10, 1492); // Creamos la instancia (se invoca
                                // a su constructor)

   ...

   delete f; // Eliminamos la instancia (se invoca
             // a su destructor)
}

void func2() {

  // Dos instancias que representan el mismo día.
  Fecha *f1 = new Fecha(12, 10, 1492);
  Fecha *f2 = new Fecha(12, 10, 1492);

  f1 == f2; // Esto evalúa a false. Son objetos distintos

  *f1 == *f2; // true: los "contenidos" de los dos punteros
              // son iguales.

  f2 = f1; // De ahora en adelante, f1 y f2 representan a la misma
           // instancia.
           // Por el camino, hemos dejado una fuga (leak) de memoria
           // que ya nunca liberaremos: la instancia a la que
           // referenciaba f2 antes de la asignación.

  f1 == f2; // Esto evalúa a true: son la misma instancia.

  f1.avanzaUnDia();  // Cambiamos f1

  f1 == f2; // Ahora sigue evaluando a true.

  delete f1; // Borramos la instancia
  delete f2; // FALLA: esa instancia ya está borrada (era la misma que
             // f1).
}
```

Capítulo 4
Herencia

Desde el punto de vista de un programador, el usuario no es más que un periférico que teclea cuando se le envía una petición de lectura

P. Williams

Resumen: En este tema se presenta el concepto de *herencia* en orientación a objetos. Este mecanismo permite al programador crear jerarquías de clases que recuerdan a las taxonomías utilizadas en otras ramas de la ciencia y que ahorran tiempo de desarrollo y, bien utilizada, consiguen implementaciones más claras y concisas.

Primera aproximación a la herencia

La herencia es un mecanismo fundamental de la programación orientada a objetos. En este tema vemos los conceptos básicos de la misma aunque las verdaderas ventajas se consiguen cuando se combina con el polimorfismo y la vinculación dinámica que trataremos en el capítulo siguiente.

Imaginemos que tenemos una clase `Persona` capaz de representar una persona, con su DNI, nombre y apellidos. Tiene un método `mostrar` que escribe en la salida estandar la información[1]:

```
public class Persona {

   private long dni;
   private String nombre;

   public Persona() { dni = -1; nombre = ""; }

   public Persona(long nuevoDni, String nuevoNombre) {
     this.dni = nuevoDni;
     this.nombre = nuevoNombre;
   }

   // [ ... omitidos getNombre, setNombre, getDNI y setDNI ... ]
```

[1]Un buen programador de Java habría implementado el método `toString` en su lugar; uno de C++ habría sobreescrito el operador <<.

```
    // [ ... que tienen sentido especialmente cuando se usa ... ]
    // [ ... el constructor sin parámetros.                  ... ]

    public void mostrar() {
        System.out.println("Nombre: " + this.nombre);
        System.out.println("DNI: " + this.dni);
    }

}
```

La clase la hemos utilizado durante un tiempo en el desarrollo de una aplicación donde de repente nos surge la necesidad de manejar un tipo concreto de `Personas`: los `Alumnos`. La diferencia es que para los alumnos hay que añadir a lo que ya guarda la clase `Persona` otro tipo de información: el número de matrícula y el número de créditos aprobados.

En vez de crear una clase `Alumno` desde cero posiblemente copiando partes del código de `Persona`, y dado que un alumno *es* una persona, podemos utilizar el mecanismo de herencia[2]:

```
public class Alumno extends Persona {

    private long numMatricula;
    private int creditosAprobados;

    public Alumno() {
        numMatricula = -1;
        creditosAprobados = 0;
    }

    // [ ... omitido constructor con parámetros ... ]

    public long getNumMatricula() { return numMatricula; }
    public long getCreditosAprobados() { return creditosAprobados; }

    public void setNumMatricula(int m) {
        numMatricula = m;
    }

    public void setCreditosAprobados(int c) {
        creditosAprobados = c;
    }

}
```

La clase `Alumno` *hereda* de la clase `Persona` (debido al `extends Persona` utilizado en su declaración). Eso significa que las instancias de la clase `Alumno` son también instancias de la clase `Persona`, y por lo tanto un objeto de `Alumno` también tiene disponibles los métodos públicos de la clase `Persona`:

```
Alumno al = new Alumno();

al.setNombre("Walterio Malatesta");
al.setDNI(12312312);
al.setCreditosAprobados(9);
```

[2]El énfasis en el "es" no debe entenderse como muestra de sorpresa por parte de los autores :). El esquema "X es Y" suele utilizarse en diseño orientado a objetos para identificar herencia.

```
al.mostrar();
```

El objeto `al` anterior incorpora tanto los atributos privados declarados en la clase `Alumno` como los heredados de la clase `Persona`, por lo que en memoria almacena dos `long`, un `int` y un `String`.

Como ya detallaremos en la sección 4, desde `Alumno` no se puede acceder directamente a los atributos de `Persona` por el hecho de que son privados. Para leer/escribir en los atributos de `Persona`, hay que usar los métodos que `Persona` haya dispuesto para ello.

Se dice que `Persona` es la clase *padre*, *superclase* o clase *base* de la clase `Alumno`. De forma simétrica, `Alumno` es la clase *hija*, *subclase* o clase *derivada* de la clase `Persona`. También se dice que `Alumno` *especializa* `Persona` y que `Persona` *generaliza* `Alumno`.

La misma idea se aplica en C++. Mostramos primero la definición de la clase `Persona`:

```
class Persona {
public:

   Persona() { dni = -1; nombre = ""; }

   Persona(long dni, std::string nombre) {
     this->dni = dni;
     this->nombre = nombre;
   }

   // [ ... omitidos getNombre, setNombre, getDNI y setDNI ... ]
   // [ ... que tienen sentido especialmente cuando se usa ... ]
   // [ ... el constructor sin parámetros.                  ... ]

   void mostrar() const {
      std::cout << "Nombre: " << nombre << std::endl;
      std::cout << "DNI: " << dni << std::endl;
   }

private:
   long dni;
   std::string nombre;
};
```

y luego la definición de la clase `Alumno`. Esta vez, la herencia se indica con dos puntos y la palabra reservada *public*.

```
class Alumno : public Persona {
public:
   Persona() {
      numMatricula = -1;
      creditosAprobados = 0;
   }

   // [ ... omitido constructor con parámetros ... ]

   long getNumMatricula() { return numMatricula; }
   long getCretidosAprobados() { return creditosAprobados; }

   void setNumMatricula(int m) {
      numMatricula = m;
```

```
    }

    void setCreditosAprobados(int c) {
        creditosAprobados = c;
    }

private:
    long numMatricula;
    int creditosAprobados;
};
```

El uso de las instancias de Alumno es idéntico al ejemplo en Java:

```
Alumno al;

al.setNombre("Walterio Malatesta");
al.setDNI(12312312);
al.setCreditosAprobados(9);

al.mostrar();
```

El ejemplo muestra una pequeña parte del concepto de herencia, pero lo suficiente para una primera aproximación.

Herencia y jerarquía de clases

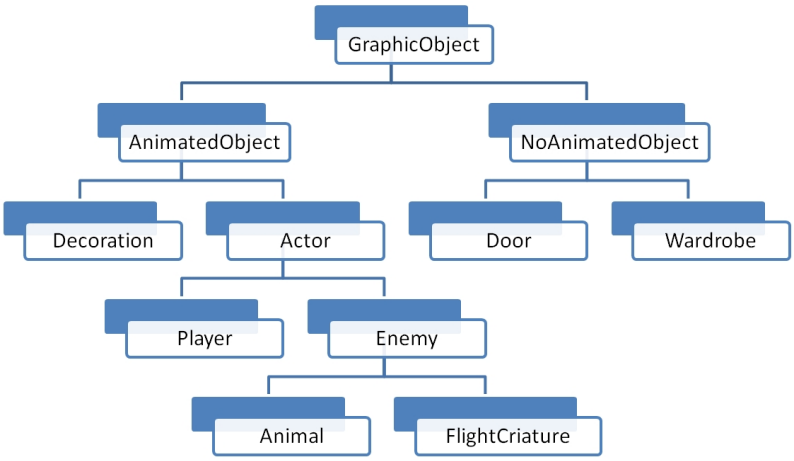
La herencia es el principal mecanismo de la OO para fomentar y facilitar la *reutilización* del software: si se necesita una nueva clase de objetos y se detectan *suficientes* similitudes con otra clase ya desarrollada, se toma esa clase existente como punto de partida para desarrollar la nueva de forma que:

- Se adoptan automáticamente características *ya implementadas* con el ahorro de tiempo y esfuerzo que ello supone.

- Se adoptan automáticamente características *ya probadas*, ahorrando tiempo de pruebas y depuración.

Aunque al crear una subclase podríamos hacer que ésta sea más general que la superclase, el enfoque más habitual (y más útil en general) es el contrario: la herencia se ve como una *especialización*. Los lenguajes de programación habituales están diseñados con esta visión de forma que los mecanismos de herencia que proporcionan están pensados para que las clases hijas representen conceptos más específicos que las clases padre.

Viéndolo de esta forma el mecanismo de clases permite definir una *jerarquía de clases* que podemos ver como una *taxonomía*. A continuación aparece lo que podría ser una jerarquía de clases parcial utilizada en un videojuego para representar todos los elementos visibles:



La organización taxonómica es una potente herramienta para la construcción de modelos por dos razones fundamentales: evita repeticiones y permite construir descripciones a distintos niveles de abstracción.

Evita las repeticiones porque permite extraer las características comunes de un conjunto de clases y definir con ellas una clase más general a la cual especialicen otras clases más específicas. Así, la descripción de una clase se completa automáticamente, a través de la herencia, con las descripciones de las superclases.

A medida que ascendemos por la jerarquía nos encontramos descripciones cada vez más abstractas, con menor nivel de detalle. Los clientes pueden ver a un objeto con el nivel de abstracción que les interese.

Podemos dar dos razones por las que utilizamos la herencia:

- Para reutilización de código: los métodos y atributos de los padres son heredados por los hijos. El ejemplo del apartado anterior se sitúa aquí.

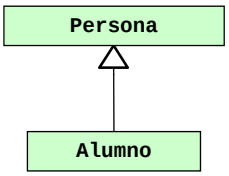
- Por la reutilización de conceptos: tiene que ver con la parte taxonómica. Desde un punto de vista conceptual, tanto la puerta (`Door`) como el armario (`Wardrobe`) de la figura anterior son objetos no animados. En este caso se aprovecha un mecanismo que veremos más tarde, la sobreescritura o redefinición de métodos (sección 6).

Cuando se crea una clase derivada de otra:

- Los atributos de la nueva clase serán la suma de los definidos en la clase padre junto con los nuevos aportados por la clase hija. Que sean accesibles o no dependerá de las reglas de visibilidad que veremos en la sección 4.

- Los métodos de la nueva clase son la suma de los métodos de la clase padre y los de la clase hija (aunque, como veremos, algunos métodos de la clase padre son sobreescritos). Igual que antes la visiblidad de esos métodos dependerá de las reglas explicadas después.

La relación de herencia puede representarse en los diagramas de clase uniendo la clase padre y la clase hija por una línea con el extremo de la clase padre terminado en un triángulo sin relleno:



Construcción y destrucción

Para que un objeto de una clase que pertenece a una jerarquía de clases esté completamente construido debe inicializarse tanto la parte específica de esa clase como la parte *heredada* desde la clase padre.

En el escenario más simple, ante un

```
al = new Alumno();
```

en el que los constructores de las clases padre e hija son sin parámetros, hay que inicializar los atributos de las dos clases y llamar a los dos constructores.

El orden en el que se realiza la tarea es distinto: varía según el lenguaje y en ciertos escenarios es importante conocerlo.

Tanto en Java como C++ el orden coincide: no se inicializa nada de la clase hija hasta que no se ha completado la inicialización de la clase padre. Esto es automático y verificado en tiempo de compilación.

Asumiendo que estamos utilizando los constructores sin parámetros (veremos más adelante el proceso con constructores con parámetros), el orden de creación de objetos es el siguiente[3]:

1. Localizar la clase padre. Si la clase padre tiene a su vez otra clase padre, continuar hasta encontrar una clase que no tenga padre.
2. Inicializar los atributos de la clase padre localizada (llamando a sus constructores si hiciera falta). Esto implica ejecutar la asignación que esté incluida en la declaración del atributo.
3. Ejecutar el constructor sin parámetros de la clase padre localizada.
4. Inicializar los atributos de la clase hija (llamando a sus constructores si hiciera falta) y continuar con todas las hijas localizadas. Esto implica ejecutar la asignación que esté incluida en la declaración del atributo.
5. Ejecutar el constructor sin parámetros de la clase hija y continuar con los de todas las hijas localizadas.

Cuando se destruye el proceso se invierte. En el siguiente ejemplo en C++ se pone de manifiesto el proceso completo[4]. Se definen dos clases que se utilizarán como atributos y que en su constructor y destructor escriben ciertas cadenas. El ejemplo lo completan una clase padre y una clase hija que poseen esos atributos y que también escriben tanto en sus constructores como en los destructores:

```
#include <iostream>
using namespace std;
```

[3]Como estamos en el proceso de creación de objetos no nos preocupamos de la inicialización de los atributos de clase, que se maneja de forma independiente.

[4]En Java es similar, pero las llamadas a los destructores es realizada por el recolector de basura.

```
class AtribA {
public:
  AtribA() {
    std::cout << "AtribA::AtribA()" << std::endl;
  }
  ~AtribA() {
    std::cout << "AtribA::~AtribA()" << std::endl;
  }
};

class AtribB {
public:
  AtribB() {
    std::cout << "AtribB::AtribB()" << std::endl;
  }
  ~AtribB() {
    std::cout << "AtribB::~AtribB()" << std::endl;
  }
};

class Padre {
public:
  Padre() {
    std::cout << "Padre::Padre()" << std::endl;
  }
  ~Padre() {
    std::cout << "Padre::~Padre()" << std::endl;
  }
private:
  AtribA a;
};

class Hija : public Padre {
public:
  Hija() {
    std::cout << "Hija::Hija()" << std::endl;
  }
  ~Hija() {
    std::cout << "Hija::~Hija()" << std::endl;
  }
private:
  AtribB b;
};

int main() {
  Hija h;
}
```

Al crear en el `main` un objeto de la clase `Hija` se pone en marcha todo el proceso de construcción, empezando por la clase Padre: primero su atributo y luego su constructor. Posteriormente se pasa a la clase hija, inicializando el atributo `AtribB` y luego su constructor. A la hora de la destrucción, cuando la variable sale de ámbito, el proceso es el contrario. El resultado de la ejecución es:

```
AtribA::AtribA()
```

```
Padre::Padre()
AtribB::AtribB()
Hija::Hija()
Hija::~Hija()
AtribB::~AtribB()
Padre::~Padre()
AtribA::~AtribA()
```

En C++ es prácticamente *imprescindible* que los destructores de las clases se declaren como *virtuales*. El concepto de método *virtual* es explicado en la sección 6 y la razón por la que los destructores deben ser virtuales se entenderá en el tema siguiente.

El mismo ejemplo en Java quedaría como sigue:

```
class AtribA {

  AtribA() {
    System.out.println("AtribA::AtribA()");
  }

  protected void finalize () throws Throwable {
    System.out.println("AtribA::~AtribA()");
  }
}

class AtribB {

  AtribB() {
    System.out.println("AtribB::AtribB()");
  }

  protected void finalize () throws Throwable {
    System.out.println("AtribB::~AtribB()");
  }
}

class Padre {

 private AtribA a = new AtribA();

  Padre() {
    System.out.println("Padre::Padre()");
  }

  protected void finalize () throws Throwable {
    System.out.println("Padre::~Padre()");
  }

}

class Hija extends Padre {

  private AtribB b = new AtribB();

  Hija() {
    System.out.println("Hija::Hija()");
```

```
  }

  protected void finalize () throws Throwable {
    System.out.println("Hija::~Hija()");
  }

};

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

    {
      Hija hija=new Hija();
    }
    // Pedimos al recolector de basura que
    // entre en funcionamiento...
    Runtime r = Runtime.getRuntime();
    r.gc(); // gc = Garbage Colletor
  }
}
```

En este código hay pequeñas diferencias con respecto al de C++:

- Los atributos hay que inicializarlos directamente en la definición de los mismos para conseguir el mismo efecto que en C++. Si no se usa el operador *new*, los atributos valdrán *null*.
- La declaración del destructor se ha convertido a su equivalente en Java. Hay que recordar que, en Java, la destrucción de los objetos no la gestiona el programador sino el recolector de basura.
- Para mostrar algo de recolección de basura, en el Main la creación de una instancia de *Hija* se hace dentro de un bloque. Ello hace que, al salir del bloque, se convierta la instancia en basura. Para forzar la recolección se invoca al método `gc` de `Runtime`. Ten en cuenta que esto habitualmente *no* hay que hacerlo; lo hemos añadido aquí a modo de ejemplo para intentar forzar al recolector de basura y que el funcionamiento del código en Java sea lo más parecido posible al de C++.

El resultado depende de las decisiones que haya tomado el recolector de basura. Dado que la especificación de la máquina virtual no determina cómo debe funcionar, dependerá de la implementación (de hecho la llamada a `r.gc()` debe interpretarse como una *sugerencia* dada por el programador al recolector para que intente eliminar objetos). En una ejecución concreta el resultado fue:

```
AtribA::AtribA()
Padre::Padre()
AtribB::AtribB()
Hija::Hija()
AtribB::~AtribB()
```

Se puede ver que la destrucción de los objetos en Java no llega a invocarse del todo. Ello se debe a que sólo el recolector de basura puede hacerlo y, en este contexto, no parece que llegue a ejecutarse por completo antes de terminar el programa. Sí que llega a invocarse para el atributo de Hija.

Hasta este punto se han usado constructores sin parámetros. Cuando se usan constructores con parámetros, el código de las constructoras hay que adecuarlo a las constructoras de las clases padre que se quieran reutilizar. En el ejemplo siguiente se ha modificado el código para crear una constructora nueva en *Padre* e *Hija* con parámetros para inicializar atributos. Igual que ocurría antes, el código es correcto y se ejecuta, aunque la salida será distinta:

```
class Padre {

  private AtribA a = new AtribA();

  Padre() {
    System.out.println("Padre::Padre()");
  }

  Padre(AtribA valorInicial){
   a =valorInicial;
  }
}

class Hija extends Padre {

 private AtribB b = new AtribB();

  Hija() {
    System.out.println("Hija::Hija()");
  }

  Hija(AtribB valorInicial){
    b = valorInicial;
  }

}

public class Main {
  public static void main(String args[]) {
     Hija hija = new Hija(new AtribB());
  }
}
```

Se ha visto antes que hay una llamada implícita a una constructora de la clase *Padre* cada vez que se invoca una constructora de la clase *Hija*. Si no se indica nada, la constructora de *Padre* invocada es la de la constructora sin parámetros. Por ello, si ahora se quita la constructora por defecto de *Padre*, como indica el código siguiente, se verá que ya no compila el código.

```
class Padre {

  private AtribA a;

  public Padre(AtribA valorInicial) {
    a = valorInicial;
  }
}

class Hija {
```

```
  private AtribB b = new AtribB();

  Hija() {    //  error: constructor Padre in class
              // Padre cannot be applied to given types;
    System.out.println("Hija::Hija()");
  }

  Hija(AtribB valorInicial) { // Mismo error que en Hija()
    b = valorInicial;
  }
}
```

Para entender mejor por qué falla (y automáticamente averiguar la solución), podemos ver qué es exáctamente lo que está haciendo el compilador para llamar al constructor de la clase padre. Lo que hace es introducir automáticamente una invocación a él como "primera instrucción" del constructor implementado en la clase hija. Esa invocación implícita podemos hacerla explícita utilizando la palabra reservada `super`, de forma que el código de la clase `Hija` es en realidad equivalente a:

```
class Hija extends Padre {

  private AtribB b = new AtribB();

  Hija() {
    super();
    System.out.println("Hija::Hija()");
  }

  Hija(AtribB valorInicial){
   super();
   b = valorInicial;
  }
}
```

Cada vez que se pone `super()` (o que el compilador lo añade por nosotros), se indica que hay que invocar la constructora sin parámetros de la clase padre[5]. Por ello, cuando la constructora sin parámetros no está, da el error de compilación.

La solución es utilizar `super` para llamar a otro constructor distinto, dándole los parámetros adecuados. El ejemplo siguiente sí es válido a pesar de no existir constructor sin parámetros en la clase Padre:

```
class Hija extends Padre {

  private AtribB b = new AtribB();

  Hija() {
    super(new AtribA());
    System.out.println("Hija::Hija()");
  }

  Hija(AtribA valorInicialA, AtribB valorInicialB){
   super(valorInicialA);
```

[5]En realidad, como es lógico, Java sólo permite poner esa invocación en la primera instrucción del constructor.

```
   b=valorInicialB;
  }

  Hija(AtribB valorInicialB){
    super(new AtribA());
    b = valorInicialB;
  }

};
```

Ahora el `main` puede sacar ventaja de la nueva constructora e indicar cómo se inicializa la parte del padre y la parte de la hija:

```
public class Main {

 public static void main(String args[]) {

   Hija hija=new Hija(new AtribA(), new AtribB());
 }
}
```

En C++ se puede hacer algo equivalente, pero utilizando la lista de inicializadores del constructor, que también puede utilizarse para inicializar atributos que no tengan constructores sin parámetros.

Imaginemos que tenemos la siguiente clase:

```
class Padre {
public:
  Padre(int valorInicial) : atrib(valorInicial) {
  }

private:
  int atrib;
};
```

Si implementamos otra clase que hereda de ésta, cuando se construya un objeto de la misma deberá llamarse al único constructor (con un parámetro) de la clase `Padre`, de forma que el siguiente código *no* compilará:

```
class Hija : public Padre {
public:

   Hija() { // Error: 'Padre' : no hay disponible un
            //  constructor predeterminado adecuado
      std::cout << "Instancia de Hija construida." << std::endl;
   }
};
```

Para hacerlo funcionar utilizaremos en `Hija` algo parecido al `super`: indicar en la lista de inicializadores a qué constructor (y con qué parámetros) queremos invocar de la clase `Padre`:

```
class Hija : public Padre {
public:

   // En este caso se llama al constructor de Padre
   // con el valor 0
```

```
    Hija() : Padre(0) {
        std::cout << "Instancia de Hija construida." << std::endl;
    }

    // Y en este con el valor recibido como parámetro
    Hija(int tam) : Padre(tam) {
        std::cout << "Instancia de Hija construida." << std::endl;
    }
};
```

Visibilidad

Una vez explicada la herencia debemos recuperar el tema de la visibilidad que ya introdujimos en la sección 2 del capítulo anterior.

Desde el punto de vista del usuario de la clase hija la regla básica es que todos los atributos de visibilidad de la clase padre se mantienen. Eso significa que los elementos (métodos y atributos) que eran públicos (y por tanto visibles y utilizables desde fuera) siguen siéndolo.

Por otro lado, al introducir la herencia se añade un nuevo "tipo de usuario": las implementaciones de métodos en las clases hija. O lo que es lo mismo ¿se puede acceder a un atributo no público de la clase padre en la implementación de los métodos de la clase hija?

Para permitir al programador de una clase decidir si permitirá o no a las clases hijas acceder a sus elementos, se añade un nuevo tipo de modificador de visibilidad: `protected`. Las clases derivadas podrán acceder a los elementos públicos y protegidos de las clases base pero *no* a sus atributos privados[6].

Recordemos que en Java, además, se puede omitir el modificador de visibilidad para utilizar la "visibilidad por defecto" que debe interpretarse como "público dentro del mismo paquete". Eso significa que una clase derivada podrá acceder a uno de esos elementos sólo si está en el mismo paquete.

Las reglas de visibilidad completas para Java aparecen en la tabla siguiente. En ella se muestra a qué elementos de una clase se puede acceder desde fuera en base a si se hace en el mismo paquete y como clase derivada o no.

	Miembros de la superclase			
	Públicos	Protegidos	Por defecto	Privados
Clase en el mismo paquete	Sí	Sí	Sí	No
Subclase en el mismo paquete	Sí	Sí	Sí	No
Clase en otro paquete	Sí	No	No	No
Subclase en otro paquete	Sí	Sí	No	No

Colisión de nombres en los atributos

Aunque desde un punto de vista de claridad de código no debería ocurrir nunca, podría darse el caso de que una clase definiera un atributo y una clase derivada definiera otro atributo con el mismo nombre (con tipos iguales o distintos).

[6]Esto significa que, aunque la clase `Alumno` tenga el atributo `nombre` debido a que lo hereda de `Persona` los métodos implementados en la clase `Alumno` *no* pueden acceder a él directamente, sino que tendría que hacerlo mediante los métodos accedentes o modificadores.

El resultado en ese caso sería que el objeto tendría *dos atributos distintos* ocupando zonas de memoria independientes. Cuando el código de un método de la clase padre referencia al atributo lo hace al declarado en su clase padre. Cuando se referencia el atributo en la clase hija, lo hace al propio y no al heredado.

En ese escenario (tan desafortunado) si desde la clase hija se quiere acceder al valor del atributo de la clase padre se utilizará la palabra reservada `super` en Java o el nombre de la clase padre en C++[7].

Ejemplo en Java:

```
class Padre {
   protected char comun;
   public Padre() {
      comun = 'c';
   }
}

class Hija {
   protected double comun;
   public Hija() {
      comun = 3.0;
      super.comun = 'd';
   }
}
```

Y en C++:

```
class Padre {
public:
   Padre() {
      comun = 'c';
   }

protected:
   char comun;
};

class Hija {
public:
   Hija() {
      comun = 3.0;
      Padre::comun = 'd';
   }

protected:
   double comun;
};
```

La colisión de nombres cuando el elemento en vez de ser un atributo es un método nos lleva al concepto de *sobreescritura* o *redefinición* de métodos que tratamos en la sección siguiente.

[7]Evidentemente, sólo se podrá hacer si el atributo es visible según las normas de visibilidad vistas anteriormente.

Sobreescritura

Recuperando el ejemplo de principio del tema de las clases `Persona` y `Alumno`, cuando invocamos el método `mostrar` sobre un `Alumno` la única información que se muestra es la de la clase `Persona`, pues es donde está implementado el método.

La *sobreescritura* nos permite *redefinir* la implementación del método dando una nueva y exige que la declaración del método, su signatura, sea idéntica en el caso del método que sobreescriba. En Java basta con volver a escribir el método con la misma signatura. En el ejemplo, la clase `Alumno` sobreescribe el método *mostrar* de la clase `Persona`[8].

```
public class Alumno extends Persona {

   [ ... ]

   public void mostrar() {
      // Código original (copiado, ya veremos como
      // hacerlo sin copiar)
      System.out.println("Nombre: " + this.nombre);
      System.out.println("DNI: " + this.dni);
      // Código nuevo
      System.out.println("Código de matrícula: " + this.numMatricula);
      System.out.println("Créditos ya aprobados: " + creditosAprobados);
   }

  [ ... ]

  public static void main(String args[]){
    Alumno alumno = new Alumno();
    Persona persona = new Persona();
    Persona personaAlumno = alumno;

    persona.mostrar(); // muestra nombre + dni
    alumno.mostrar(); // muestra nombre+dni+código de matrícula + créditos
    personaAlumno.mostrar(); // muestra nombre+dni+código de matrícula + créditos
  }
}
```

Cuando se ejecuta en el main el método *mostrar* del objeto almacenado en la variable *persona*, es intuitivo que lo que se imprimirá es el nombre y el dni. Ahora bien, cuando se invoca *mostrar* sobre un objeto de la clase *Alumno*, hay que pensar que hay dos métodos posibles que se pueden ejecutar al tener la misma signatura: el *mostrar* de *Alumno* y el *mostrar* de *Persona*. Cuando se dan estos casos, se elige el método que está en la clase más especializada, dentro de la parte de la jerarquía que se esté considerando. Así pues, se ejecutaría sólo el método *mostrar* de *Alumno*.

> Cuando un método se reescribe en dos o más clases siendo estas clases especializaciones unas de las otras, se elige siempre la implementación del método de la clase más especializada.

[8] En la implementación de la clase `Persona` que aparecía en la página 51 los atributos `nombre` y `dni` aparecían privados. Para hacerlos accesibles a las clases hijas, a partir de este momento asumiremos que se ha cambiado su visibilidad a `protected`.

Adelantando contenidos acerca del *polimorfismo* de la sección 7 y sabiendo que un *Alumno* es una *Persona* por la relación de herencia que los une, es posible meter un objeto de la clase *Alumno* en una variable que haya sido declarada de tipo *Persona*. En estos casos, también se cumple la misma regla, la de invocar el método de la clase más especializada, que aunque se esté en una variable de tipo *Persona*, sigue siendo de tipo *Alumno*.

También en Java, es posible que el entorno agregue automáticamente la anotación de sobreescritura `@Override`. Esta anotación sólo indica explícitamente que el método en cuestión sobreescribe un método de alguna clase padre en la jerarquía de herencia a la que pertenece la clase. No es necesaria y puede ser eliminada, pero ayuda al desarrollador para que vea, al imprimir el código o al incluirlo en una documentación, que el método sobreescribe a otros.

```
public class Alumno extends Persona {
   [ ... ]
   @Override
   public void mostrar() {
    [ ... ]
   }
  [ ... ]
}
```

En C++ el proceso es similar aunque es importante (por razones que entenderemos en el capítulo siguiente) que, al menos en la clase padre, los métodos que luego podrán ser sobreescritos sean declarados como *virtuales* (palabra clave `virtual`):

```
class Persona {
public:
   [ ... ]

   virtual void mostrar() const {
      [ ... ]
   }

   [ ... ]
};
```

```
class Alumno : public Persona {
public:

   [ ... ]

   virtual void mostrar() const {
      [ ... ]
   }
   [ ... ]
};
```

En muchos casos cuando sobreescribamos un método lo que querremos será *extender* su funcionalidad haciendo lo que se hacía en la clase padre y algo más. Para evitar copiar el código en la nueva implementación se puede invocar desde el método al mismo método de la clase padre de forma similar a como se accedía al atributo con el mismo nombre: utilizando `super` en Java y el nombre de la clase padre en C++. De esta forma, el nuevo método *mostrar* que se vio antes quedaría como sigue:

```java
// En Java
public void mostrar() {
   super.mostrar();
   // Código nuevo
   System.out.println("Código de matrícula: " + this.numMatricula);
   System.out.println("Créditos ya aprobados: " + creditosAprobados);
}
```

```cpp
// En C++
void mostrar() const {
   Persona::mostrar();
   // Código nuevo
   [ ... ]
}
```

En general en la sobreescritura *no* puede cambiarse la visibilidad del método, al menos no puede ponerse una visibilidad más restrictiva[9].

Un último aspecto que merece la pena mencionar es qué ocurre cuando dentro de un método de la clase padre se invoca a un método sobreescrito. Por poner un ejemplo concreto:

```java
class Persona {

   [ ... ]

   public void mostrar() {
      System.out.println("Nombre: " + this.nombre);
      System.out.println("DNI: " + this.dni);
   }

   public void mostrarConBordes() {
      System.out.println("==========================");
      this.mostrar();
      System.out.println("==========================");
   }
}
```

Y tenemos la clase `Alumno` anterior que sobreescribe el método `mostrar` escribiendo la información completa. Cuando sobre una instancia de `Alumno` se llama al método (heredado) `mostrarConBordes`:

```java
   [ ... ]

   Alumno walterio = new Alumno(12312312, "Walterio Malatesta");

   walterio.mostrarConBordes();

   [ ... ]
```

En la ejecución el método `mostrarConBordes` terminará invocando (enviando el mensaje de ejecución) al método `mostrar` *de la instancia con la que está trabajando*. En este

[9] En lenguajes como C# sí se permite degradar la visiblidad del método sobrecargado en algunas circunstancias.

caso esa instancia es un `Alumno` que ha sobreescrito el método por lo tanto el método `mostrar` que se ejecuta es el de el `Alumno` y no el de la `Persona`[10]. Este funcionamiento está relacionado con el polimorfismo y la vinculación dinámica que trataremos en el tema siguiente.

Principio de sustitución. Subtipos y subclases

Ahora que ya hemos visto más detalles sobre la herencia podemos recuperar lo que describíamos al principio del tema sobre jerarquías de clases y taxonomías.

Desde un punto de vista teórico podemos ver las clases como *tipos* o *conceptos* de forma que cuando establecemos una relación de herencia lo que estamos indicando es que la nueva clase es un *subtipo* o *subconcepto*. Igual que en una taxonomía todos los mamíferos son vertebrados (y por tanto tienen una serie de características y comportamientos comunes con el resto de vertebrados no mamíferos), en una relación de herencia todos los alumnos son personas, y se comportarán como tal.

No obstante, subtipo y subclase son dos cosas independientes, pues el lenguaje/compilador no puede garantizar que se mantienen: el programador de la clase hija podría sobreescribir un método de la clase padre de tal forma que su *semántica* no tenga nada que ver con el subconcepto al que pertenece. Podemos decir que:

- Una clase es subclase de otra si ésta ha sido construida usando el mecanismo de herencia.
- Una clase es subtipo de otra si preserva el propósito original, o si cumple el principio de sustitución enunciado más abajo.
- Todos los subtipos son subclases pero no todas las subclases son subtipos: puede que el programador haya reimplementado algún método que vulnere el principio de sustitución.

El principio de sustitución dice que si `B` es subclase de `A` entonces en *cualquier situación* se puede sustituir una instancia de `A` por una de `B` obteniéndose el mismo comportamiento observable.

Que se mantenga el principio de sustitución (o lo que es lo mismo, que la relación de subclase coincida con la de subtipo) es, en general, una cuestión de disciplina del programador. El principio de sustitución es de vital importancia con el polimorfismo tratado en el tema siguiente.

Otro aspecto relacionado con el principio de sustitución tiene que ver con la sobreescritura de métodos del apartado anterior. ¿Tendría sentido permitir cambiar los tipos de los parámetros del método sobreescrito?

Por ejemplo, al implementar la igualdad uno siente la tentación de especializar el tipo de los parámetros. Por poner un ejemplo concreto, podemos vernos tentados a implementar en Java algo así:

```
class Shape {
   public boolean igualA(Shape other) {
      return false;
   }
```

[10] En C++ el comportamiento es igual siempre y cuando el método haya sido sobreescrito declarándolo como `virtual`.

```
}

class Square extends Shape {
    public boolean igualA(Square other) {
        ...
    }
}

class Triangle extends Shape {
    public boolean igualA(Triangle other) {
        ...
    }
}
```

Donde intentamos sobreescribir un método cambiando los tipos de sus parámetros a otros más específicos. En general podemos ver dos aproximaciones:

- Covarianza: el tipo de los parámetros en la especialización es una especialización del tipo en la clase base. Hay problemas con el sistema de tipos.
- Contravarianza: el tipo de los parámetros en la especialización es una generalización del tipo en la clase base. No suele resultar de mucha utilidad.

En la mayoría de los lenguajes se opta por *no* permitir ninguna de las dos alternativas: los tipos de los parámetros deben coincidir.

El ejemplo Java anterior *no* hace lo que pensamos: los métodos `igualA` de `Square` y `Triangle` *no* sobreescriben al método con el mismo nombre de `Shape`, sino que son dos métodos nuevos e independientes por tener parámetros de tipos distintos.

Con el tipo devuelto podemos plantear una discusión similar. En este caso tanto Java como C++ permiten que el tipo devuelto se modifique de manera covariante, es decir que devuelva un tipo más específico.

En cuanto a determinar qué método ejecutar, se tomará aquel cuyos parámetros encajan mejor en los tipos devueltos y dentro de los que tengan la misma signatura, el que esté en la clase más especializada.

Clases abstractas

Cuando se usa la herencia como herramienta, se dan ocasiones en las que una clase padre no debe proporcionar un cuerpo a un método porque espera que sea la clase hija la que lo defina. En tales casos, se tiene una *clase abstracta* y al método o métodos por definir se les llama métodos *abstractos*. Una clase abstracta por definición no puede ser instanciada, no importa las constructoras que se declaren. El motivo es precisamente lo que la hace abstracta, ¿qué sentido tiene crear una instancia si no se sabe lo que hacen uno o más métodos?

En Java, una clase abstracta debe incluir la palabra *abstract* junto a la definición de clase, debe tener al menos un método que incluya *abstract* en su declaración y que este método no tenga un cuerpo de método asociado.

```
// Ejemplo en Java
abstract class Padre {
   public abstract boolean metodoAbstracto() ;
}

class Hija extends Padre {
   public abstract boolean metodoAbstracto(){
      return true;
   }

  public static void main(String args[]){
   Hija hija=new Hija();
   // Padre padre=new Padre(); // Esto no compilaría porque Padre es abstracta
  }
}
```

En C++, no se incluye la palabra *abstract* en la definición de la clase, pero el método abstracto se declara como *virtual* y debe estar asignado a 0.

```
// Ejemplo en C++

class Padre {
   public:
      virtual bool metodoAbstracto()=0 ;
};

class Hija: public Padre {
   public:

    bool metodoAbstracto(){
      return true;
   }
};

int main(){
   Hija hija();
  // Padre padre(); // Esto no compilaría porque Padre es abstracta
  }
```

El uso de las clases abstractas se verá cuando se intenten definir soluciones reutilizables con clases. La idea será que, dada una estructura de clases que te resuelve un problema, algunas de ellas sean abstractas y que reflejen lo particularizable de una solución.

Interfaces

Podemos llevar al extremo las clases abstractas y hacer una clase que tenga *todos los métodos abstractos*, es decir, que no implemente ningún método. Esa es la idea de una interfaz.

El concepto interfaz no existe como tal en C++ (aunque siempre pueden implementarse clases con todos los métodos virtuales puros). En Java las interfaces se pueden ver como clases que:

- no tienen atributos

- sólo declaran signaturas de métodos y nunca proporcionan un cuerpo de método
- todos los métodos declarados son públicos
- pueden heredar de otras interfaces y pueden hacerlo de más de una

Se usan con frecuencia en Java para indicar qué servicios son esperados sin entrar en detalles de implementación concretos. Al no declarar atributos ni cuerpo de métodos, se deja libertad absoluta para proporcionar una solución que satisfaga los requisitos de la interfaz. Ello permite hacer un diseño top-down de forma eficaz y asumiendo diferentes implementaciones en puntos concretos, que serán aquellos que usen las interfaces.

Para declararlas se utiliza la palabra clave `interface` en vez de `class` (y normalmente irá en un fichero `.java` independiente igual que una clase). Ejemplos de declaración de interfaces son los siguientes:

```
// Ejemplo en Java
interface EjemploInterfaz1 {
  public void metodo1();
  public Integer metodo2();
  public Integer metodo3();
}

interface EjemploInterfaz2 {
  public void metodo4();
  public Integer metodo5();
  public Integer metodo6();
}

interface EjemploInterfaz3 extends EjemploInterfaz1, EjemploInterfaz2 {
  public Integer metodo7();
}
```

Las interfaces se *implementan* en clases como indica el ejemplo. Implementar requiere usar la palabra reservada *implements* asociada a tantas interfaces como se quiera (se separa cada uno con una coma), y forzando a la clase a proporcionar un cuerpo a los métodos declarados (también puede dejar alguno sin implementar convirtiendo la clase en clase abstracta):

```
// Ejemplo en Java
class EjemploImplementacion implements EjemploInterfaz1, EjemploInterfaz2 {
  public void metodo1() { ....}
  public Integer metodo2(){ ....}
  public Integer metodo2(){ ....}
  public void metodo4(){ ....}
  public Integer metodo5(){ ....}
  public Integer metodo6(){ ....}
}
```

El uso de las interfaces se hace declarando variables del mismo tipo que la interfaz en cuestión y asignándoles instancias de clases que implementen la interfaz. En el ejemplo siguiente se definen dos variables del mismo tipo que dos interfaces distintas y se les asigna una clase que implementa ambas.

```
// Ejemplo en Java
class EjemploImplementacion implements EjemploInterfaz1, EjemploInterfaz2 {
    { ....}
```

```
  public static void main(String args[]){
   EjemploImplementacion ejemplo=new EjemploImplementacion();
   EjemploInterfaz1 i1=ejemplo;
   EjemploInterfaz2 i2=ejemplo;
   i1.metodo1();
  }
}
```

Es muy útil cuando se trata de ocultar partes de una aplicación. Por ejemplo, si paso como parámetro *i1* que en el ejemplo es de tipo *EjemploInterfaz1*, nadie tiene por qué saber que, debajo, hay en realidad una instancia de la clase *EjemploImplementacion*. Esta clase, originalmente, tiene seis métodos, pero el que use *i1* sólo podrá ver los tres que les corresponden.

Hay muchas interfaces que define Java que saldrán en la asignatura, como *Serializable* (clases que se pueden serializar), *Runnable* (para indicar código a ejecutar concurrentemente), o *ActionListener* (para indicar una acción a realizar cuando se produce un evento concreto). Cómo se implementen estas interfaces queda a nuestra discreción. Sólo se pide que se implementen los métodos que se piden, dejando libertad para determinar estructuras de datos y métodos adicionales que necesitemos.

Herencia múltiple

Hasta ahora todos los ejemplos que hemos visto constaban de una clase que heredaba de otra; es lo que se llama *herencia simple*: una clase hija tiene sólo un padre. ¿Tendría sentido heredar de más de una clase?

Lo cierto es que ha existido, y sigue existiendo, el debate sobre si un lenguaje orientado a objetos debe permitir que una clase hija pueda tener varias clases padre, que es lo que se conoce como *herencia múltiple*.

En un ejemplo de herencia múltiple lo que tendríamos es una clase derivada que hereda elementos (métodos y atributos) de dos (o más) clases distintas. Eso significa que una instancia de esa clase derivada tendrá *todos* los atributos de las clases superiores en la jerarquía así como los atributos que ella incluya; tendrá además todos los métodos de ambas ramas y podrá añadir nuevos o sobreescribir algunos.

```
// Ejemplo en C++

class Padre1 {
   public:
      [ ... ]
      void incrementa() { value++; }
   private:
      int value;
};

class Padre2 {
   public:
      [ ... ]
      void apaga() { encendido = false; }
   private:
      bool encendido;
};
```

```
class Hija : public Padre1, public Padre2 {
    [ ... ]
  private:
    std::string cadena;
};
```

Las instancias de `Hija` almacenarán los tres atributos y se podrá llamar a los métodos `incrementa`, `apaga` y el resto de métodos que haya en `Hija`.

La herencia múltiple tiene una serie de problemas que la hacen, cuando menos, polémica en el mundo de la orientación a objetos.

El primer problema (que de hecho tienen también lenguajes como Java que no soportan herencia múltiple de clases) es debido a los posibles conflictos de nombres que pueden surgir.

Por ejemplo, si una clase *Hija* hereda de *Padre1* y *Padre2* y ambas clases definen un mismo método, por ejemplo, *void mostrar()*, al invocar en una instancia de la clase *Hija* el método *mostrar*, ¿cuál de los dos métodos se invocará? Esto queda reflejado con el ejemplo siguiente:

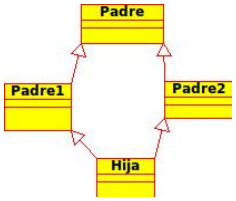
```
// Ejemplo en C++
#include <iostream>
using namespace std;
class Padre1 {
   public:
      void mostrar(){
         cout << "Padre1";
      };
};

class Padre2 {
   public:
      void mostrar(){
         cout << "Padre2";
      };
};

class Hija: public Padre1, public Padre2 {

};

int main(){
   Hija hija=Hija();
   hija.mostrar();  // ERROR: acceso ambiguo de 'mostrar', puede
                    // ser el mostrar en la base 'Padre1' o
                    // puede ser el 'mostrar' en la base 'Padre2'
  }
```

La clase `Hija` tiene *dos métodos* `mostrar` distintos, por lo que cuando el usuario de la clase lo invoca el compilador da un error debido a la ambigüedad: no sabe a cuál de las dos implementaciones quiere llamarse. Para solucionar el problema el lenguaje debe aportar algún mecanismo (sintaxis) que permita desambiguar, que en el caso de C++ es con:

```
hija.Padre1::mostrar() // Llama a la implementación de Padre1
hija.Padre2::mostrar() // Llama a la implementación de Padre2
```

Sin embargo el principal problema de la herencia múltiple es lo que se conoce como

herencia en diamante: una clase derivada hereda de dos clases distintas que, a su vez, heredan de una cuarta clase:



El problema en este caso es que los objetos de la clase Hija van a tener *duplicados* los atributos (y métodos) de `Padre`: una copia heredada vía `Padre1` y otra heredada vía `Padre2`. El siguiente ejemplo muestra el problema con el método `mostrar1` y el atributo público `problematico`[11]:

```
// Ejemplo en C++
#include <iostream>
using namespace std;

class Padre {
   public:
      void mostrar1(){
         cout << "Padre" << problematico << std::endl;
      };

      int problematico;
};

class Padre1: public Padre {
   public:
      Padre1() { problematico = 1; }
      void mostrar2(){
         cout << "Padre1";
      };
};

class Padre2: public Padre {
   public:
      Padre2() { problematico = 3; }
      void mostrar3(){
         cout << "Padre2";
      };
};

class Hija: public Padre1, public Padre2 {

};

int main(){
   Hija hija=Hija();

   hija.mostrar2();
   hija.mostrar3();
   hija.mostrar1(); // ERROR: ¿por qué vía llamamos?
```

[11] Los atributos nunca deberían ser públicos, pero en este caso ayuda para mostrar el problema.

```
   hija.problematico; // ¿Cuál de los dos?
  }
```

Para evitar la duplicación de los atributos de la clase por heredar dos veces de la misma clase C++ proporciona una solución conocida como "herencia virtual" que instruye al compilador a mantener una única copia de los atributos.

En Java y otros lenguajes la solución es simplemente *no* permitir herencia múltiple de forma que una clase *nunca* podrá heredar atributos e implementaciones de métodos de dos clases distintas.

En concreto, Java sigue el principio de *herencia simple* cuando se trata de clases y el de *herencia múltiple* cuando se trata de interfaces.

En Java, los ejemplos anteriores son reproducibles en parte. La herencia múltiple se limita a interfaces, que sólo pueden declarar operaciones. Por lo tanto, no hay colisión de posibles implementaciones del método ni de heredar varias veces un mismo atributo. Las redefiniciones de método, por lo tanto, dan igual. En el ejemplo se tiene una clase que implementa una interfaz que proviene de una herencia en diamante. Como se puede ver, no se requiere el dar varias implementaciones del método redefinido; la clase que implemente esa interfaz sólo "tiene que implementarlo una vez":

```
interface Padre {
 public void mostrar();
}

interface Padre1 extends Padre {
 public void mostrar();
}

interface Padre2 extends Padre {
 public void mostrar();
}

interface Hija extends Padre1, Padre2 {
 public void mostrar();
}

class Ejemplo implements Hija{
 public void mostrar(){
   System.out.println("Ejemplo");
 }
 public static void main(String args[]){
  Hija hija=new Ejemplo();
  Padre2 padre2=hija;
  Padre1 padre1=hija;
  Padre padre=hija;
  hija.mostrar();
  padre2.mostrar();
  padre1.mostrar();
  padre.mostrar();
 }
}
```

Usando el principio de sustitución, se crea una única instancia de *Ejemplo* y se hace pasar por *Hija*, *Padre2*, *Padre1*, *Padre* indistintamente.

Limitando la herencia en Java

La herencia en Java incorpora un mecanismo de control que no se da en C++: el poder impedir que se sobreescriban métodos o que se extiendan clases. Ambas cosas se consiguen con la palabra reservada *final*. Esta palabra puede anteceder la declaración de una clase o bien la de un método.

El siguiente ejemplo muestra una clase final. El ejemplo no compila porque ninguna clase puede especializar la clase final.

```
final class Padre {}; // Clase final, no puede extenderse
class Hija extends Padre {};  // Error
```

En el caso de un método final, no se permite sobreescribir el método. El ejemplo siguiente tampoco compila porque no se puede sobreescribir un método final.

```
class Padre {
 public final void mostrar(){...} // Método final, no puede
                                  // sobreescribirse
}

class Hija extends Padre {
 public  void mostrar(){...}  // Error.
}
```

Hay clases finales en Java, como la clase *String*, *StringBuilder*, o *URL*. Eso significa que no puedo extender estas clases para proveer otras cadenas.

La palabra *final* también se usa con variables y atributos para indicar inmutabilidad: una vez hecha la definición y asignación inicial, no se puede cambiar. En el ejemplo siguiente se ilustra este caso. Si se transcribe en código, se verá que no llega ni a compilar.

```
class Ejemplo {
 public void metodo(){
  final int indice=0;
  indice = indice + 1;
 }
}
```

Jerarquías múltiples y la clase Object

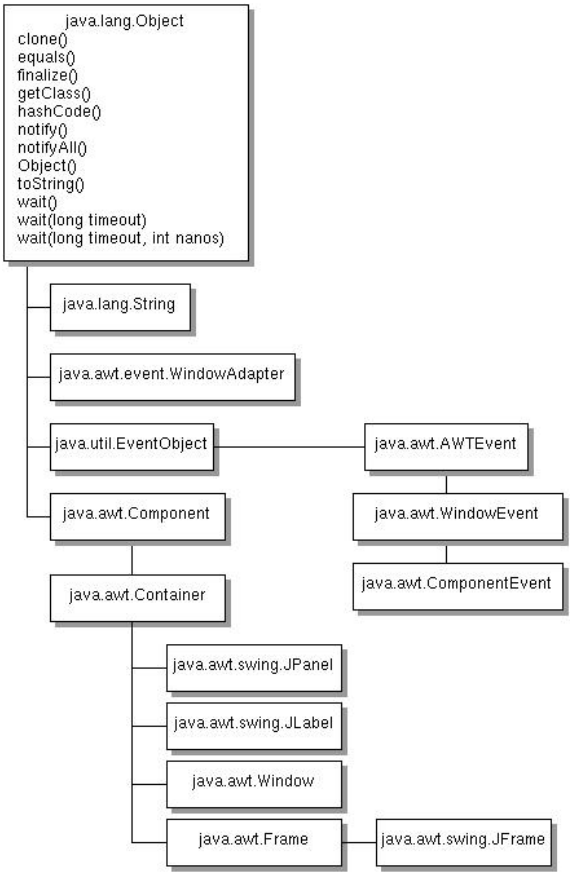
La herencia permite crear jerarquías de clases. En un lenguaje como C++ esas jerarquías pueden ser *independientes*: podemos tener dos jerarquías de clases que parten cada una de clases base distintas.

En lenguajes como Java y C# existe una única jerarquía de clases que parten de una clase común. En el caso de Java esa clase es `java.lang.Object`. Si un programador no indica ninguna superclase, el compilador añade automáticamente que la clase hereda de `Object`[12].

La clase `Object` tiene una serie de métodos útiles que automáticamente pasan a estar disponibles en todos los objetos de Java. Algunos de ellos irán apareciendo en la asignatura. Ejemplos de métodos de `Object` son *toString*, *clone*, *equals* o *hashCode*.

A continuación aparece una vista parcial de la jerarquía de Java.

[12]De ahí la importancia de eliminar la herencia múltiple en Java: si todas las clases heredan de `Object`, cualquier escenario de herencia múltiple se traduciría automáticamente en herencia en diamante.


java.lang.Object
clone()
equals()
finalize()
getClass()
hashCode()
notify()
notifyAll()
Object()
toString()
wait()
wait(long timeout)
wait(long timeout, int nanos)
java.lang.String
java.awt.event.WindowAdapter
java.util.EventObject
java.awt.AWTEvent
java.awt.Component
java.awt.WindowEvent
java.awt.ComponentEvent
java.awt.Container
java.awt.swing.JPanel
java.awt.swing.JLabel
java.awt.Window
java.awt.Frame
java.awt.swing.JFrame

Capítulo 5

Polimorfismo y vinculación dinámica

No hay lenguaje de programación, no importa su estructura, que impida que los programadores hagan malos programas.

Larry Flon

Resumen: En este tema se presenta el concepto de *polimorfismo*, que va de la mano de la vinculación dinámica. Aunque ya han aparecido ejemplos de polimorfismo y vinculación dinámica en el tema anterior (es complicado explicar la herencia sin utilizarlos), en este tema se explican todos los detalles de su funcionamiento.

Definición de polimorfismo

El término *polimorfismo* viene del griego y significa "múltiples formas". En programación se utiliza con diferentes significados, aunque en la mayoría de los casos cuando se utiliza el término se está pensando únicamente en la última acepción:

- Sobrecarga (*overloading*). También llamada *polimorfismo ad-hoc*. Se produce cuando tenemos varios métodos o funciones con el mismo nombre, el mismo tipo de retorno, pero que difieren en los parámetros dentro del mismo ámbito.
- Sobreescritura (*overwriting*) o reemplazo (*overriding*): un caso especial de sobrecarga que ocurre en el contexto de una relación de herencia entre clases padre e hija. Se produce cuando hay métodos con la misma signatura en las clases padre e hija.
- Variables polimórficas: se produce cuando una variable se declara de un tipo pero durante la ejecución puede contener valores (objetos) de distinto tipo. Es el que muchos llaman "polimorfismo puro". Dado que es el más importante, muchas veces nos referiremos a él con el nombre de "polimorfismo".

Sobrecarga

Se produce al tener varios métodos o funciones dentro del mismo ámbito que poseen el mismo nombre, el mismo tipo de retorno, pero que difieren en los parámetros.

No todos los lenguajes permiten la sobrecarga (por ejemplo C), aunque está presente en la mayoría de los lenguajes fuertemente tipados como C++ o Java.

```
public class Log {

   public void escribe(float dato) { ... }
   public void escribe(String cad) { ... }
}
```

La sobrecarga se resuelve *en tiempo de compilación*, basándose en los tipos estáticos de los valores de los argumentos utilizados. El compilador examina el contexto en el que el nombre sobrecargado se está usando y determina el método o función más adecuado.

```
Log l = new Log();
l.escribe(3.5f);  // versión escribe(float)
l.escribe("3.5"); // versión escribe(String)
```

Durante este examen, si no se encuentra ningún método candidato, el compilador prueba con conversiones de tipos automáticas promocionando los tipos. Así si utilizamos

```
l.escribe(7);
```

el compilador al no encontrar ningún método recibiendo un entero, hará la promoción del `7` a `7.0f` e invocará a la versión `escribe(float)`. Esto es la *promoción* de los tipos o conversión automática.

Es importante destacar que ni Java ni C++ tienen en cuenta toda la signatura del método/función para la sobrecarga. En particular, el tipo devuelto no debe variar en la sobrecarga. Eso significa que no es correcto lo siguiente:

```
public class Log {

   public void escribe(float dato) { ... }
   public boolean escribe(float cad) { ... }
}
```

ya que tenemos dos métodos con los mismos parámetros pero distinto tipo devuelto. Es lógico, pues si el programador ignora el valor devuelto el compilador no sabría a qué versión llamar:

```
l.escribe(7.0f); // ¿llamamos a void escribe(float)?
                 // ¿o a boolean escribe(float) ignorando
                 // el valor devuelto?
```

Existen otros modificadores de los métodos/funciones que tampoco se tienen en cuenta y que por tanto no pueden utilizarse como única variación para dos métodos sobrecargados. Por ejemplo, tampoco la declaración de las excepciones lanzadas (que veremos en un tema posterior) o la visibilidad.

Sobreescritura

Aunque ya hemos explicado la sobreescritura la vemos ahora desde el punto de vista del polimorfismo. La sobreescritura puede verse como un caso especial de sobrecarga que

ocurre en el contexto de una relación de herencia entre clases padre e hija, y se da cuando hay métodos con la misma signatura en las dos clases[1].

La principal diferencia con la sobrecarga es que en el caso de la sobreescritura se resuelve *en tiempo de ejecución*, dependiendo del tipo dinámico del objeto. Y en este punto debemos decir que la sobreescritura no sería de utilidad si no existiera el concepto de *variables polimórficas* que indicábamos como tercer tipo de polimorfismo.

Variables polimórficas

Se dice que una variable es polimórfica cuando en el código fuente se declara de un tipo pero durante la ejecución puede contener valores (objetos) de otros tipos. Como ya explicaremos con detalle más tarde, sólo los tipos referencia (punteros o referencias) permiten variables polimórficas.

```
// Java
...
Persona p;
p = new Alumno;
```

La variable `p` anterior tiene como *tipo estático* (o tipo en tiempo de compilación) la clase `Persona`. Sin embargo durante la ejecución la variable toma el valor de una instancia de la clase `Alumno`. El polimorfismo es posible gracias a la herencia: dado que un *Alumno es* una `Persona`, el compilador admite la asignación a la variable polimórfica.

El polimorfismo facilita la reutilización de código y la implementación de aplicaciones y *armazones* (o *frameworks*) extensibles. Como ejemplo, imaginemos que tenemos un método que carga el mapa de una aventura conversacional desde disco:

```
class Cargador {
   public static Map cargaMapa(java.io.FileReader file) {...}
}
```

para eso recibe un fichero ya abierto para lectura. La forma de invocar al método anterior será algo como[2]:

```
Map m;
m = Cargador.cargaMapa(new java.io.FileReader("MiMapa.map"));
```

En la práctica la implementación del método `cargaMapa` hará uso únicamente de la operación `read`. Un rápido vistazo a la documentación de Java nos descubre que la clase `java.io.FileReader` en realidad *hereda* de `java.io.Reader`. `java.io.Reader` representa a los *stream* de caracteres. Y resulta que `FileReader` sobreescribe la implementación de `read` haciendo la lectura efectiva desde fichero[3].

Debido a esto, se puede hacer una versión más general de `cargaMapa`:

```
class Cargador {
   public static Map cargaMapa(java.io.Reader file) {...}
}
```

[1] Las mismas reglas que para la sobrecarga se mantienen: el compilador no distingue dos métodos del mismo nombre y con los mismos parámetros pero distinto valor devuelto.

[2] Omitimos aspectos de comprobación de errores en la apertura del fichero.

[3] En realidad la clase `FileReader` es abstracta por lo que no proporciona implementación de `read`, pero para la discusión esto no es relevante.

que recibe `Reader`'s generales independientemente de dónde se realice la lectura. El parámetro pasa a ser una variable polimórfica que guardará en tiempo de ejecución un tipo u otro dependiendo del usuario.

De esta forma el usuario podrá utilizar el método para lectura del mapa desde fichero, desde un buffer guardado en memoria o incluso desde un servidor en la red utilizando un socket.

Vinculación estática y vinculación dinámica

Las variables polimórficas y el uso de la sobreescritura es posible gracias a lo que se llama vinculación dinámica. Para ponerla en contexto, lo primero es ver qué es la vinculación estática. Para eso, utilizamos un ejemplo simple en C++[4]

```
// C++
Persona p;

...

p.mostrar();
```

En el ejemplo, `p` es un objeto de la clase `Persona` y tanto su tipo estático como su tipo en tiempo de ejecución coinciden (recordemos que el ejemplo es en C++ donde todos los tipos son tipos valor). Como la variable `p` no es una variable polimórfica (porque no es de tipo puntero o referencia), se utiliza *vinculación estática*: en el momento de la compilación se determina qué método será el invocado. Es posible que de `Persona` hereden otras clases (la clase `Alumno`), pero dado que la variable no es polimórfica, se invoca directamente al método `Persona::mostrar`. Dado que la vinculación entre el nombre del método y la implementación utilizada se realiza en tiempo de compilación, también se llama *vinculación anticipada*. En C++ éste es el mecanismo utilizado por defecto.

La vinculación dinámica consiste en retrasar este paso al momento de la ejecución. Cuando se trata con variables polimórficas, la invocación a un método se traslada al momento de la ejecución y en vez de utilizar el tipo estático (o tipo declarado en el código) se utiliza el tipo dinámico.

Por mantener el ejemplo del cargador de mapas anterior, en Java:

```
class Cargador {
   public static Map cargaMapa(java.io.Reader file) {

      ...
      int next = file.read();
      ...
   }
}
```

En la codificación de *Reader*, el método *read()* invoca a un método abstracto *read(char[] cbuf, int off, int len)* que es el que implementan las clases que, como *FileReader* o *StringReader*, quieren leer de medios concretos. Estas clases, además, proveen nuevas implementaciones para el método *read*. Por todos estos motivos, en el código anterior no es posible saber, al invocar a *read*, qué consecuencias tendrá y por ello se posterga la decisión

[4]Usamos C++ porque es difícil explicar la vinculación estática en Java por el uso indiscriminado de tipos referencia.

al momento de ejecutar. El método ejecutado dependerá del tipo en tiempo de ejecución de la variable `file`:

```
m = Cargador.cargaMapa(new FileReader("MiMapa.map")); // 1
m = Cargador.cargaMapa(new StringReader("<map/>")); // 2
```

La línea 1 provocará que el `read` utilizado sea el implementado en la clase `FileReader` mientras que en el caso 2 se utilizará a lectura sobre la cadena en memoria implementada en `StringReader`.

Vinculación estática y dinámica en C++

Como ya hemos mencionado, la vinculación dinámica sólo se puede realizar sobre variables polimórficas. Por otro lado, los tipos valor *no* son variables polimórficas, por lo que en C++ por defecto se utiliza vinculación *estática*.

Para poder utilizar vinculación dinámica en C++, pues, tenemos que utilizar tipos referencia usando punteros o referencias:

```
Persona *p; // Variable polimórfica; p puede apuntar
            // a objetos de tipo Persona o de cualquier
            // otra subclase.

p = new Alumno(); // En ejecución el tipo del objeto apuntado
            // por p es Alumno

p->mostrar();     // Se llama a virtual void Alumno::mostrar()
```

Vinculación dinámica y sobrecarga

Imaginemos que tenemos la siguiente clase:

```
class A {
  public void hacerLoQueSea(Object a){
    System.out.println("El objeto NO es de tipo A");
  }

  public void hacerLoQueSea(A a){
    System.out.println("El objeto es de tipo A");
  }

  public static void main(String args[]){
    A a = new A(); // Variable polimórfica.
                   // Tipo estático y dinámico coinciden.

    Object b = a;  // Variable polimórfica.
                   // Tipo estático: Object
                   // Tipo dinámico: A
    Object c = new A(); // Variable polimórfica.
                        // Tipo estático: Object
                        // Tipo dinámico: A

    A comprobador=new A();
    comprobador.hacerLoQueSea(a);
    comprobador.hacerLoQueSea(b);
    comprobador.hacerLoQueSea(c);
```

```
 }

};
```

El método `main` presenta un ejemplo en el que se mezcla sobrecarga con vinculación dinámica: tenemos dos métodos sobrecargados y la diferencia de sus parámetros se da en el tipo, de forma que uno de ellos es descendiente del otro.

Dado que la sobrecarga *se resuelve en tiempo de compilación* y por tanto utiliza siempre *vinculación estática*, en este caso concreto el compilador invoca a uno u otro método dependiendo del *tipo estático*. Por tanto, el ejemplo escribe:

```
El objeto es de tipo A
El objeto NO es de tipo A
El objeto NO es de tipo A
```

> Si se están usando variables polimórficas como parámetros de métodos sobrecargados, se tomará como tipo de parámetro para esta variable polimórfica exactamente el mismo tipo con el que fue declarada y no el tipo real de la instancia que contiene.

Coste de la vinculación dinámica

La vinculación dinámica traslada tareas que antes realizaba el compilador al momento de la ejecución. En concreto, es durante la ejecución cuando se tiene que determinar *a qué método llamar* en base al tipo de datos de la variable polimórfica.

En Java no hay forma de salvar este coste, pero no es el caso de otros lenguajes. En lenguajes como C++ y C# la vinculación dinámica *no* es la opción por defecto. Si el programador desea que se utilice la vinculación dinámica cuando se llama a un método de una variable polimórfica debe marcar ese método de forma especial. En concreto, en C++ deberá utilizar la palabra `virtual`. Si no se especifica, la vinculación será estática.

Un ejemplo que pone de manifiesto todo esto es el siguiente:

```
class A {
public:
   virtual int f() { return 1; }
   int g() { return 2;}
};
class B : public A {
public:
   virtual int f() { return 4; }
   int g() { return 8; }
};
int main() {
  A a; // Variable no polimórfica
  B b; // Variable no polimórfica
  A *p = &b; // Variable polimórfica a objeto de B

  int c = a.f() + b.f() + p->g();

  return 0;
}
```

En el ejemplo, `a.f()` invoca a `A::f()`, pues la variable `a` no es polimórfica y su tipo en tiempo de compilación (tipo estático) es `A`. De forma similar, `b.f()` invoca a `B::f()`.

Por último, al llamar a `p->g()` se está utilizando una variable polimórfica para llamar al método `g()`. En este caso como ese método en la clase del tipo estático (`A`) *no* es virtual, no se utiliza vinculación dinámica y se llama por tanto a `A::g()`.

El valor final de la variable `c` es por tanto 7.

La vinculación dinámica durante la construcción y destrucción de objetos

En la implementación de los constructores se puede llamar a métodos de la propia clase. La pregunta evidente es ¿qué ocurre si esos métodos están sobreescritos?:

```
class Padre {
   public Padre() {
      metodoSobreEscrito();
   }

   public void metodoSobreEscrito() {
      System.out.println("Padre");
   }
}

class Hija {
   public void metodoSobreEscrito() {
      System.out.println("Hija");
   }

   public static void main(String []args) {
      Hija h = new Hija();
   }
}
```

En este ejemplo cuando se ejecuta el constructor de `Padre` se invoca a un método sobreescrito. En Java la vinculación dinámica entra en acción y, dado que en tiempo de ejecución el objeto es de la clase `Hija`, el método invocado es `metodoSobreEscrito()` de *Hija*. Eso puede provocar problemas pues, según las reglas de construcción de los objetos Java, la parte del objeto correspondiente a la clase `Hija` *aún no ha sido inicializada*, por lo que el método *no* debería utilizar ningún atributo propio.

Para evitar ese problema en C++ la vinculación dinámica *no está disponible* en los constructores y destructores.

Por otro lado, las variables polimórficas en C++ pueden generar fugas de memoria si no se implementa correctamente. Consideremos el siguiente código:

```
class CEntity {
};

class CBadGuy : public CEntity {
public:
   CBadGuy() {
      _weapon = new CWeapon;
   }
   ~CBadGuy() {
      delete _weapon;
   }
private:
   CWeapon* _weapon;
```

```
};

int main() {
   CEntity* e = new CBadGuy;
   delete e;
   return 0;
}
```

Cuando el compilador llega a `delete e;` comprueba cuál es el tipo de la variable `e` para llamar al destructor. En este caso la variable `e` es polimórfica pero como el destructor *no* es `virtual` se utiliza vinculación estática y se invoca directamente al destructor de la clase `CEntity`, lo que provoca fugas de memoria. Por eso:

> En C++ los destructores deben declararse virtuales a no ser que se esté completamente seguro de que nunca se implementarán clases derivadas.

Comprobando el tipo real

Llegado a este punto, hay que hablar de qué hacer cuando el tipo real de una instancia importa. A veces, se tiene un código que, en función del tipo que se esté considerando, tiene que ejecutar un código distinto. Para este tipo de situaciones, los lenguajes proveen formas de averiguar de qué tipo es realmente una variable polimórfica.

En Java, el operador `instanceof` indica si una instancia es del mismo tipo que de una clase concreta. Así *`instancia`* `instanceof` *`Clase`* querría decir si la variable polimórfica *`instancia`* contiene algo que sea de una clase que herede de *`Clase`*. En el código siguiente se ilustra una clase que tiene un método capaz de discernir cuándo se le pasa una instancia de la misma clase o de otra distinta.

```
class A {

  public void hacerLoqueSea(Object a){
    if (a instanceof A){
   System.out.println("El objeto es de tipo A");
    } else
   System.out.println("El objeto NO es de tipo A");
  }


 public static void main(String args[]){
  Object a = new A();
  Object b = "hola"; // new String("hola");

  A comprobador=new A();
  comprobador.hacerLoqueSea(a);
  comprobador.hacerLoqueSea(b);
 }
};
```

Sin embargo, el uso de `instanceof` es cuestionable en la mayoría de situaciones porque va en contra del polimorfismo. La ventaja del polimorfismo es que permite asumir comportamientos por defecto y reusar el código. El distinguir mediante `instanceof` hace que sí que importe el tipo y que no haya comportamientos por defecto. Además, siempre

existe un diseño alternativo que permite hacer lo mismo. Por lo tanto, antes de usarlo, hay que estar seguros de que esas otras opciones son peores.

> No se debe usar `instanceof` en nuestros programas a no ser que se indique lo contrario.

Para el caso anterior, por ejemplo, se recomendaría algo como lo que sigue, que saca partido de la vinculación dinámica:

```
abstract class A {
 public abstract void hacerLoqueSea();
}

class B extends A {
 public void hacerLoqueSea(){
   System.out.println("El objeto es B");
  }
}

class C extends A {
 public void hacerLoqueSea(){
   System.out.println("El objeto es C");
  }
}

class Main {
 public static void main(String args[]){
  A a=new B();
  A b=a;
  A c= new C();
  a.hacerLoqueSea();
  b.hacerLoqueSea();
  c.hacerLoqueSea();
 }
};
```

El comportamiento es el esperado porque, en función del tipo, se ejecuta un código u otro.

Conversión de tipos

Java y C++ tienen conversiones de tipos automatizadas para aquellos casos en que es seguro. Por ejemplo, cuando se habla de tipos valor, se puede asignar un *int* a un *float*. En el caso de Java se da un paso más allá, siendo válido el código siguiente donde se asignan tipos valor a variables que tienen como tipo una clase[5].

```
class A {
 public static void main(String args[]){
  Integer entero=5;
  int otroEntero=5;
  entero=otroEntero;
 }
```

[5]Se usa el concepto de *Boxing* y *Unboxing* que vimos en el tema 3.

```
};
```

Java, para cada tipo valor, provee una clase que lo representa. Así se tienen las clases *Integer*, *Float*, *Double*, *Boolean*, *Byte*, o *Char*, por citar algunas. Cualquier asignación de un tipo de valor a una variable de tipo la correspondiente clase de tipo de valor, provocará una conversión automática de tipos.

Para el caso de las variables polimórficas no es tan fácil. La forma de conversión en estos casos se hace con un *cast* al tipo que interese. Por ejemplo, en el código siguiente, se crea un objeto de tipo *A*, se guarda en una variable de tipo *Object* y después se vuelve a guardar en una variable de tipo *A*.

La conversión "hacia arriba" (desde `A` a `Object`) *siempre* es segura, pues son variables polimórficas y todos los objetos de la clase `A` *son* también `Object` gracias a la herencia. Por lo tanto el programador no necesita hacer explícita la conversión. Sin embargo el compilador *no* puede garantizar que la conversión contraria, de una clase base a una clase derivada (conocida como *downcast*), sea segura, por lo que requiere que el programador la marque explícitamente:

```
class A {
 public static void main(String args[]){
  Objet objetoInicial = new A();
  A ahoraNecesitoElTipoA = (A)objetoInicial;
 }
};
```

La conversión en tiempo de ejecución es especialmente delicada, pues no se sabe a ciencia cierta si la conversión es posible. El ejemplo siguiente compila pero da un error de ejecución (generando una excepción):

```
class B {
}

class A {

 public static void main(String args[]){
  Object a=new A();
  Object b=new B();
  A nueva=(A)b;
 }

};
```

Este tipo de construcciones son dudosas también desde el punto de vista de orientación a objetos. Si en un lugar se asume que sólo se necesita saber que una instancia válida de una clase, ¿por qué forzar la conversión a otro tipo? Muy probablemente, se trate de un problema de diseño del programa, por lo que el uso de este operador para *downcast* debe evitarse siempre.

> Salvo indicación explícita, no se va a usar la conversión manual de tipos.

En C++, el equivalente a esta conversión se llama *dynamic_cast* y su uso es parecido al casting de Java, solo que la sintaxis varía. Eso sí, para que tenga sentido, las variables involucradas deben ser polimórficas, por lo que serán punteros o referencias.

Si tenemos dos clases[6]:

```
class A
{
public:
        // Since RTTI is included in the virtual
        // method table there should be at least
        // one virtual function.
        virtual void foo();

        // other members...
};

class B : public A
{
public:
        void methodSpecificToB();

        // other members.
};
```

el uso de `dynamic_cast` con punteros es:

```
B *b = new B(); // Tipo estático y dinámico: B.
A *a1 = b;  // Tipo estático: A, tipo dinámico B.
A *a2 = new A(); // Tipo estático y dinámico: A.

B *ptr;

ptr = b; // Seguro siempre. Tipos estáticos coinciden.

ptr = a1; // ERROR al compilar. Conversión no segura (downcast).
          // Requiere que el programador la haga explícita

ptr = dynamic_cast<A*>(a1); // Conversión explícita. En
          // tiempo de ejecución se comprueba si puede hacerse.
          // Como es correcta, todo va bien.

ptr = dynamic_cast<A*>(a2); // Conversión explícita. Downcast
          // incorrecto. El dynamic_cast asigna NULL a ptr.
          // El programador puede comprobar if (!ptr) ...

ptr = (B*)a1; // NO DEBE USARSE. El compilador no garantiza
          // que el downcast se haga bien.
ptr = (B*)a2; // ¿Qué ocurre aquí? Cuando usemos ptr
          // tendremos un error en ejecución (this incorrecto).
```

Cuando se utilizan variables polimórficas de tipo referencia, sin embargo, no puede utilizarse el valor especial `NULL` para indicar error en la conversión, por lo que se utilizan excepciones[7]:

```
#include <typeinfo> // For std::bad_cast
#include <iostream> // For std::cerr, etc.

void my_function(A& my_a)
```

[6]Sacado del ejemplo de la wikipedia.
[7]Ejemplo extraido de la Wikipedia

```
{
        try
        {
                B& my_b = dynamic_cast<B&>(my_a);
                my_b.methodSpecificToB();
        }
        catch (const std::bad_cast& e)
        {
                std::cerr << e.what() << std::endl;
                std::cerr << "This object is not of type B"
                          << std::endl;
        }
}
```

Implementación de la vinculación dinámica

El objetivo de esta sección es ilustrar cómo se resuelve la vinculación dinámica en cada lenguaje. Normalmente, la vinculación dinámica no se controla a nivel de nuestros programas, siendo algo inherente al compilador.

La vinculación dinámica se implementa añadiendo información adicional a los objetos en tiempo de ejecución. Resulta evidente que durante la ejecución es necesario saber de qué tipo es el objeto para determinar a qué método llamar.

Aunque la forma de implementar la vinculación dinámica depende por completo del lenguaje y compilador/máquina virtual, la idea básica utilizada en todos ellos es la misma. En concreto cada objeto que puede ser referenciado por una variable polimórfica (todos los objetos en Java y aquellos que tienen algún método `virtual` en C++) tienen un campo oculto adicional que apunta a la "tabla de funciones virtuales". Y esa tabla contiene punteros a las funciones que hay que llamar. En toda la jerarquía de clases se utiliza la misma entrada para cada método de forma que en tiempo de ejecución simplemente hay que ir a la entrada correspondiente y ejecutar la función que ahí se indica. En el momento de construir el objeto el puntero apuntará a una u otra tabla dependiendo del tipo (en tiempo de ejecución).

Capítulo 6
Excepciones

Un buen programador es aquél que mira a los dos lados antes de cruzar una carretera con un único sentido.

Doug Linder

Resumen: En este capítulo se muestra el concepto de excepciones como mecanismo de gestión de errores. Se presenta la sintáxis utilizada en Java así como la jerarquía de excepciones de su librería.

Introducción

Hasta ahora, las excepciones han sido los errores del programa, pero van más allá. Citando el tutorial de Java de Oracle

> An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

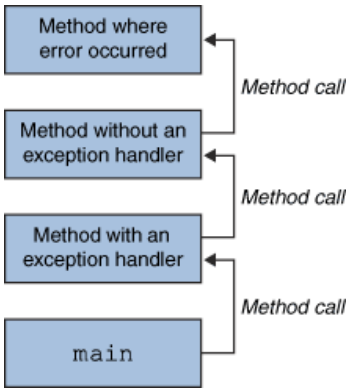
Hay que enteder que, dentro de esta definición, hay tres tipos fundamentales de eventos:

- Errores. Son eventos que indican un error irrecuperable. No se esperan mayores acciones tras la ocurrencia de un error y lo normal es que el programa se interrumpa.
- Excepciones no comprobables. Son las que se dan en tiempo de ejecución. Se corresponden con situaciones excepcionales derivadas normalmente de errores de programación. Por ejemplo, invocar un método usando una variable que tiene un *Null*, lo cual genera un *NullPointerException* o división por cero.
- Excepciones comprobables. Son aquellas que constituyen fallos menores, como intentar leer un fichero que no existe, lo cual produce un *IOException*.

Una excepción detiene la ejecución de un programa, pero para entender cómo funciona, hace falta hacer referencia a la pila de llamadas. Se sabe de primero que el hecho de hacer llamadas hace que se vayan apilando las direcciones de retorno de las llamadas[1]. Cuando

[1]Se hacen más cosas, pero quedémonos ahí de momento.

se produce una excepción, ésta debe ser tratada en el lugar de aparición. Si ahí no se consigue, la excepción se propaga hacia arriba en la cadena de llamadas hasta llegar a la llamada original. Si ésta tampoco indica qué hacer con la excepción, el programa termina. La figura siguiente, extraida del tutorial de Oracle ilustra este caso.



La figura da a entender que cada método elige entre pasar la excepción o tratarla. Pasar una excepción hacia atrás requiere avisar en el método correspondiente qué excepción puede lanzarse. Y dentro de este método deberá haber código que la genere, al menos en el caso de las excepciones comprobables. Las no comprobables se producen igualmente, pero no hace falta que el método declare que pueden darse.

Declarando y produciendo una excepción

Las excepciones comprobables se llaman así porque, en Java, se modifica la signatura del método para incluir la palabra `throws` seguida de tantas excepciones como se generen dentro del método.

```
class Ejemplo {
 public void metodoQueLanzaExcepcion() throws NombreDeLaExcepcion{
  // lo que sea
 }
}
```

Este tipo de excepciones aparecen con frecuencia en el API de Java, por ejemplo, en la clase File, alternándose con excepciones no comprobables. En el fragmento siguiente, se observa en la signatura que se puede capturar la excepción *IOException*, pero que, además, se puede lanzar una *SecurityException*, la cual es una excepción no-comprobable.

createNewFile

```
public boolean createNewFile()
                    throws IOException
```

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The FileLock facility should be used instead.

Returns:
`true` if the named file does not exist and was successfully created; `false` if the named file already exists

Throws:
IOException - If an I/O error occurred
SecurityException - If a security manager exists and its SecurityManager.checkWrite(java.lang.String) method denies write access to the file

Since:
1.2

La generación de la excepción se produce o bien cuando se recibe una excepción al hacer alguna llamada a algún método que tiene una cláusula `throws` en su signatura; o bien se crea y lanza la excepción. La creación de una excepción es igual que la construcción de una instancia de una clase. Para lanzarla, se usa la palabra clave `throw` seguida de una instancia de una excepción.

```
class NombreDeLaExcepcion extends Exception{};

class Ejemplo {
 public void metodoQueLanzaExcepcion() throws NombreDeLaExcepcion{
  throw new NombreDeLaExcepcion(); // se crea una
  //instancia de la excepción y se lanza
 }
}
```

El caso de las excepciones no comprobadas merece discusión aparte. No hace falta declararlas en la signatura del método. En el ejemplo siguiente, se crea y lanza una excepción no comprobable de tipo RuntimeException. Si se intenta hacer algo parecido con una excepción comprobable, como *Exception*, se puede comprobar que se genera un error en tiempo de compilación.

```
class Ejemplo {
 public void metodoQueLanzaExcepcion() {
  throw new RuntimeException("Esto no se tiene por qué declarar");
 }
 public static void main(String args[]){
  new Ejemplo().metodoQueLanzaExcepcion();
 }
}
```

Tratando la excepción

El responsable de tratar una excepción lo indica usando un `try/catch`. Esta construcción es común a C++ y Java. El siguiente código es Java:

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

```
  try {
   // aquí se ubica el código que lanza la excepción
  } catch (Exception e){
   // aquí se trata la excepción
   e.printStackTrace();
  }
 // por aquí continuaría ejecutándose
 }
}
```

Cuando se escribe un `try/catch` se espera que el código incluido entre llaves genere, o pueda generar una excepción. De lo contrario, en Java, se produce un error de compilación. En cada `try/catch`, se pueden usar tantos `catch` como se crea necesario. Cada `catch` aludirá a un tipo concreto de excepción.

Dentro del catch se espera que se trate la excepción. Esto implica también decidir si se continúa o no. Para no continuar, es necesario volver a lanzar la misma u otra excepción. Cualquier otra cosa implica seguir con la ejecución:

```
class Ejemplo {
 public static void main(String args[]){
  try {
   // aquí se ubica el código que lanza la excepción
  } catch (IOExcepcion e){
   // aquí se trata la excepción y se continua
   e.printStackTrace();
  } catch (NumberFormatException nf){
   throw nf; // vuelvo a lanzar la excepción capturada
  } catch (RuntimeException nf){
   throw new RuntimeException(nf); // vuelvo a lanzar
   // la excepción capturada
   // pero encapsulada en una nueva excepción
  }
  // por aquí continuaría ejecutándose
 }
}
```

Aunque la literatura dice que las *RuntimeException* no deberían capturarse, el caso es que corresponde al programador decidir qué hacer en cada caso, pues se pueden capturar igualmente las excepciones. Ahora bien, no parece correcto capturar un *NullPointerException* en ningún caso, aunque sí un *NumberFormatException* si se está procediendo a analizar una cadena y se están convirtiendo de cadena a un número.

Hay una diferencia sutil entre relanzar la excepción y encapsularla en otra e implica principalmente al *StrackTrace* o *traza*. Algo que se suele pedir a una excepción es que nos imprima su traza, lo cual se consigue invocando su método *printStackTrace*. El resultado es un texto volcado en la salida de error indicando qué excepción se ha producido y cuál es la cadena de llamadas que llevó a ella:

```
java.lang.NullPointerException
         at MyClass.mash(MyClass.java:9)
         at MyClass.crunch(MyClass.java:6)
         at MyClass.main(MyClass.java:3)
```

Cuando se relanza la excepción, se mantiene el stacktrace original. Sin embargo, cuando se crea una nueva excepción, se crea un nuevo stacktrace, lo cual puede no convenirnos si lo que queremos es saber dónde se produjo el error original. En tales casos, sigue siendo

posible obtenerlo porque una excepción puede indicar cuál fue la causa original que la provocó usando el método *getCause()*.

La traza también se puede usar como mecanismo de depuración. En cualquier momento, podemos crear una excepción y pedirle la traza. El código siguiente crea una excepción y ordena imprimir su traza. Esto no implica que se lance la excepción porque no se está invocando al *throw*, por lo que no hace falta alterar nada en la signatura del método correspondiente ni definir bloques *try/catch*

```
class Ejemplo {
 public static void main(String args[]){
   new Exception("Ejemplo de traza").printStackTrace();
 }
}
```

Opcionalmente se puede incluir una cláusula `finally`. Debe ir detrás de todos los bloques catch considerados. Sus sentencias se ejecutan siempre, sea cual sea la excepción que se produzca o si no se produce ninguna.

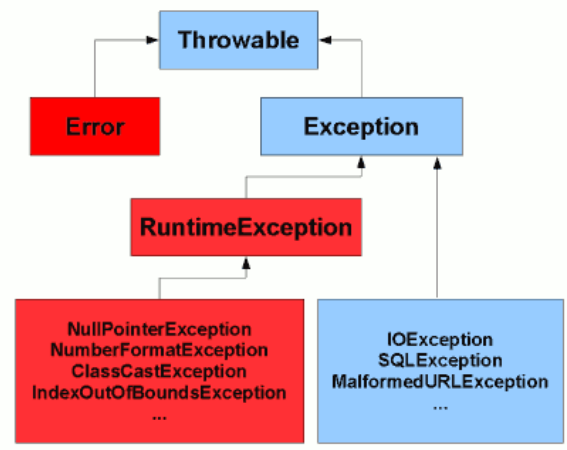
Como ejemplo de uso se podría pensar en un bloque try dentro del cual se abre un fichero para lectura y escritura de datos y se desea cerrar el fichero abierto. El fichero se debe cerrar tanto si produce una excepción como si no se produce, ya que dejar un fichero abierto puede provocar problemas posteriores. Para conseguir esto se deberá incluir las sentencias correspondientes al cierre del fichero dentro del bloque finally.

La forma general de una sección donde se controlan las excepciones es por tanto:

```
class Ejemplo {
 public static void main(String args[]){
  try {
   // aquí se ubica el código que lanza la excepción
  } catch (Excepcion e){
   // aquí se trata la excepción y se continua
   e.printStackTrace();
  } finally{
   // estas sentencias se ejecutan siempre
  }
  // por aquí continuaría ejecutándose
 }
}
```

Jerarquía de excepciones

Las excepciones en Java se tratan como una clase más. Existe una jerarquía de excepciones que refleja la distinción dada en la sección 1 sobre los tres tipos de excepciones: errores, excepciones no-comprobables y comprobables. El siguiente esquema, sacado de `http://www.javamex.com/tutorials/exceptions/exceptions_hierarchy.shtml` ilustra estos casos.



La clase básica es *Throwable* que representa el evento lanzado en ejecución. La clase *Error* engloba los errores fatales que hacen parar la ejecución. La clase *RuntimeException* se corresponde con las excepciones no comprobadas que también pueden implicar una parada de ejecución.

Nuestras excepciones, si las necesitamos, deberán extender una heredera de *Exception*. Es conveniente, además, recrear todos los constructores de la clase *Exception*. En el ejemplo siguiente, escrito en Java, se ejemplifican cada una de las variantes.

```
public class MiExcepcion extends Exception {

   public MiExcepcion() {
      // TODO Auto-generated constructor stub
   }

   public MiExcepcion(String arg0) {
      super(arg0);
      // TODO Auto-generated constructor stub
   }

   public MiExcepcion(Throwable arg0) {
      super(arg0);
      // TODO Auto-generated constructor stub
   }

   public MiExcepcion(String arg0, Throwable arg1) {
      super(arg0, arg1);
      // TODO Auto-generated constructor stub
   }

}
```

Es especialmente útil la segunda constructora y la tercera. La segunda porque permite crear excepciones personalizando el mensaje que se muestra en el stacktrace. La tercera porque permite crear excepciones que contienen excepciones. De cualquier forma, no siempre es necesario crear excepciones personalizadas. Se pueden usar las que existan adecuando el mensaje mostrado. En el ejemplo siguiente, se muestra la traza de una excepción personalizada:

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

```
  try {
   // aquí se ubica el código que lanza la excepción
   throw new Exception("Esta es mi excepcion");
  } catch (Exception e){e.printStackTrace();};
  // por aquí continuaría ejecutándose
 }
}
```

Lo que muestra el ejemplo es el texto siguiente:

```
java.lang.Exception: Esta es mi excepcion
   at A.main(A.java:5)
```

Capítulo 7

Colecciones y genéricos

First, solve the problem. Then, write the code.
John Johnson

Resumen: En este capítulo se hace una pequeña aproximación a la librería de Java relacionada con las colecciones y se introduce el mecanismo de clases *genéricas*.

Justificación

Las estructuras de datos tratadas hasta ahora se caracterizan por ser poco flexibles en diferentes aspectos. Las hemos utilizado para guardar colecciones de objetos, pero de forma poco eficaz desde el punto de vista de reutilización de código. Si se construía una instancia de un array de enteros, tenía que tener un tamaño concreto y ser sólo para enteros. Sin embargo es muy habitual *no* conocer a priori este tipo de circunstancias. Por eso, en el caso de array, se nos obliga a tener que *especular* y crear un array de un tamaño fijo esperando que entren ahí todos los elementos. En caso contrario tendremos que crear un array más grande y mover los datos a ese nuevo array. Esto implica alterar la estructura de datos, y la pregunta es si esto es necesario.

Este tipo de preguntas se extiende a cualesquiera estructuras programáticas que se creen y dado que estamos hablando de un lenguaje orientado a objetos, se plantea la cuestión de si hay mecanismos adicionales al de herencia, vinculación dinámica y estática de tal forma que pueda maximizar la reusabilidad. En Java se usan lo que se conocen como genéricos, *generics*. Su aplicación inmediata son estructuras de datos, pero veremos que tiene otras aplicaciones.

En términos semánticos, los genéricos de Java no aportan nada nuevo[1]. Se puede ver como azúcar sintáctico que hace más clara la programación evitando explicitar conversiones de tipos, pero realmente no permiten resolver ningún problema nuevo. Como ilustración del problema de partida, se considera el ejemplo del inventario.

El problema del inventario

La clase siguiente sirve para simular un "inventario" de un jugador en un juego en el que se pueden meter `Items`[2].

[1] No ocurre lo mismo con las *plantillas* de C++ que sí aportan más expresividad al lenguaje.

[2] La clase dista mucho de estar completa; se echa en falta, al menos, un método para *eliminar* objetos.

```
class Inventario {

   static final int TAM_INICIAL=10;

   Item []v;
   int next;

   public Inventario() {
      this(TAM_INICIAL);
   }

   public Inventario(int tamInicial) {

      if (tamInicial <= 0)
         tamInicial = 1;

      v = new Item[tamInicial];
      next = 0;
   }

   public int getNumItems() {
      return next;
   }

   public void addItem(Item i) {
      if (lleno())
         duplica();
      v[next] = i;
      next++;
   }

   public Item getItem(int i) {
      if ((i < 0) || (i >= next))
         return null;
      return v[i];
   }

   private void duplica() {
      Item [] nuevo = new Item[2 * v.length];

      for (int i = 0; i < v.length; ++i)
         nuevo[i] = v[i];

      // El recolector eliminará el v viejo...
      v = nuevo;
   }

   private boolean lleno() {
      return next == v.length;
   }

}
```

En este programa se ha invertido esfuerzo en crear una estructura basada en arrays que sea dinámica, esto es, que esté preparada para crecer si se queda pequeña. Este tipo

de estructuras es muy frecuente, como se verá. Partiendo de este programa, se plantea el reutilizar esta estructura dinámica y cómo hacer que este código saque partido de otras estructuras de datos existentes.

Mejorando el programa: `java.util.ArrayList`

Es tan habitual encontrarnos con la necesidad de almacenar una colección variable de objetos que la mayoría de los lenguajes de programación proporcionan, en sus bibliotecas, clases o estructuras para resolver el problema anterior. En Java esa labor es realizada, por ejemplo, por la clase `ArrayList` del paquete `java.util`.

Lo que hace *ArrayList* es similar a lo que hace nuestro *Inventario*. Si se agota el espacio, crea nuevos arrays y coloca dentro el contenido anterior.

Dada que es una clase de la librería y debe ser lo más general posible la pregunta obvia es ¿qué tipo de objetos permite almacenar? La respuesta es: *cualquier objeto*, o lo que es lo mismo, cualquier cosa que herede de `Object`[3].

La clase ArrayList usada de forma directa resulta poco elegante pues fuerza a hacer conversiones de tipos.

```
java.util.ArrayList inventario = new java.util.ArrayList();
inventario.add(new Item());
inventario.add(new Item());
Item i = (Item)inventario.get(0);
```

El método `get` es equivalente al `getItem` del ejemplo anterior. Sin embargo, dado que lo que devuelve es un `Object` necesitamos hacer explícita la conversión de tipos[4]. Si por un momento se almacenara en el vector algo que no fuera un *Item*, el código anterior daría un error en ejecución.

Dado que el `ArrayList` almacena objetos, en principio no es posible guardar valores de tipos primitivos, sino que hay que *envolverlos* en un objeto que lo guarde, en concreto en las clases asociadas (`Integer`, `Float`, etc.):

```
java.util.ArrayList enteros = new java.util.ArrayList();
inventario.add(new Integer(37));
...
Integer i = (Integer)inventario.get(0);
int j = i.intValue();
```

Desde Java 1.5 la conversión de los tipos primitivos a las clases concretas se hace automáticamente (como ya describimos en la sección 6.4 del capítulo 3), de forma que lo anterior es equivalente al código siguiente[5]:

```
java.util.ArrayList enteros = new java.util.ArrayList(); // Boxing
inventario.add(37);
...
int j = (Integer)inventario.get(0); // Unboxing
```

Con este nuevo conocimiento, se puede revisar el *Inventario* y comprobar que se puede reducir el código.

[3] La clase `Inventario` anterior permite guardar `Item`s o cualquier objeto que pertenezca a una clase que *hereda* de `Item`s. Revisa el código y compruebalo. Si en vez de `Item` hubieramos puesto `Object`, permitiría almacenar cualquier tipo de objeto.

[4] Y observa que esta conversión es un *down-casting*; algo, como sabemos, poco recomendable en general.

[5] El compilador automáticamente convierte este código en el anterior.

```
class Inventario {

   ArrayList v=new ArrayList();

   public Inventario() {}

   public int getNumItems() {
      return v.size();
   }

   public void addItem(Item i) {
      v.add(i);
   }

   public Item getItem(int i) {
      return (Item)v.get(i);
   }
}
```

Hasta la introducción de JDK 1.4 esta era la forma de codificar. Con la introducción de genéricos, se consiguió una mejor codificación, como se verá a continuación.

Mejorando el programa: Genéricos

La necesidad de las conversiones de tipos anterior no sólo es incómoda, sino también insegura. En Java 1.5 se introdujeron los *genéricos*, un concepto que permite abstraer tipos. Clases como `ArrayList` se pueden convertir en clases genéricas que están *parametrizadas* con un tipo. Usando genéricos, se puede reescribir el programa del inventario como sigue:

```
class Inventario {

   ArrayList<Item> v=new ArrayList<Item>();

   public Inventario() {}

   public int getNumItems() {
      return v.size();
   }

   public void addItem(Item i) {
      v.add(i);
   }

   public Item getItem(int i) {
      return v.get(i);
   }
}
```

El uso de genéricos permite obviar el cast y la agregación de entidades comprueba automáticamente si lo que le pasamos al vector es o no una instancia de *Item*.

Con la introducción de genéricos en Java 1.5 se crearon clases genéricas paralelas a todas las clases que permitían guardar colecciones en la librería. De esta forma, podemos usar el vector de `Item`s sin necesidad de todas las conversiones de tipos incómodas (y potencialmente inseguras):

```
java.util.ArrayList<Item> inventario = new java.util.ArrayList<Item>();
inventario.add(new Item());
inventario.add(new Item());
Item i = inventario.get(0); // Conversión no necesaria
```

Ante el código anterior el compilador comprueba que el tipo de `inventario` utiliza `Item` como valor para el parámetro de tipo y por tanto el `get` devolverá un `Item` por lo que no exige al usuario la conversión[6].

Es importante indicar que el tipo utilizado en los parámetros de los genéricos *no* puede ser un tipo básico: si queremos un vector que guarde `int`, deberemos utilizar `Integer` y confiar en el *boxing* y *unboxing*.

Pero se puede ir más allá. ¿Y si se quiere hacer que la clase no haga presunciones sobre un tipo concreto? Volviendo al ejemplo del Inventario, vamos a tratar que sea un inventario de cualquier cosa. Recordemos que, ahora mismo es un inventario de instancias de *Item*. la idea intuitiva (aunque no del todo cierta en Java) es que en el momento de crear una clase como la clase `Inventario` anterior, en vez de indicar que lo que guardaremos serán objetos de la clase `Item` le decimos al compilador que el tipo que guardamos *no lo sabemos* en el momento de crear la clase, sino que vendrá dado como parámetro en el momento de instanciar la clase `Inventario`:

```
class Inventario <Elem> {

   static final int TAM_INICIAL=10;

   Elem []v;
   int next;

   public Inventario() { ... }

   public Inventario(int tamInicial) { ... }

   public int getNumItems() { ... }

   public void addItem(Elem i) { ... }

   public Elem getItem(int i) { ... }

   ...
}
```

Toda referencia a `Item` ha sido sustituida por el tipo `Elem` que es en realidad un *parámetro* de la clase (por el `<Elem>` que aparece tras su nombre). De esta forma, podremos crear un inventario de `Item`s, pero también un inventario de `Command`s:

```
Inventario<Item> inv1 = new Inventario<Item>();
Inventario<Command> inv2 = new Inventario<Command>();
```

Aunque se haya escrito *Elem* y *Elem* como clase o interfaz no existan, eso no quiere decir que uno puede poner lo que quiera. El siguiente código no compila:

```
public class Prueba<T> {
```

[6] En la práctica, por el modo en el que funcionan los genéricos en Java, esa conversión sigue realizándose, pero la hace el compilador automáticamente.

```
 public void nuevoMetodo(T t){
  t.debeTenerEsteMetodo();
 }
}
```

Esto quiere decir que cuando uno usa genéricos, no se puede hacer lo que sea. Aunque no se indique, se asume en el ejemplo que T extiende *Object* y puedo invocar sus métodos.

```
public class Prueba<T> {
 public void nuevoMetodo(T t){
  t.hashCode();
 }
}
```

Hemos visto que dado un genérico, sólo se puede presumir de él los métodos de la clase Object. Sin embargo, existe una forma de asumir una clase básica más específica en el parámetro. Se logra usando la palabra reservada *extends*. En el ejemplo siguiente, se indica que el parámetro extiene la clase *ArrayList*, permitiendo así que sean accesibles los métodos de un vector desde una instancia del parámetro.

```
import java.util.*;
public class Prueba<T extends ArrayList> {
 public void nuevoMetodo(T t){
  System.out.println(t.size());
 }
}
```

En cualquier caso, hay algo que no se puede hacer: no se puede invocar la constructora. El código siguiente lo intenta y genera un fallo de compilación:

```
public class Prueba<T> {
 public void nuevoMetodo(){
   new T();
 }
}
```

Los genéricos se pueden aplicar también a las interfaces. Por eso, podría definir:

```
interface Inventario <Elem> {
   public int getNumItems();

   public void addItem(Elem i);

   public Elem getItem(int i);
}
```

Comodines

Imaginemos el siguiente método:

```
void printItems(java.util.ArrayList<Item> v) {
  for (Item i : v)
     System.out.println(i);
}
```

que recibe un vector de `Items` y lo recorre escribiendo uno a uno cada elemento.

Si tenemos una clase `Key` que hereda de `Item`:

```
public class Key extends Item { ... }
```

Nos podríamos plantear tener lo siguiente:

```
java.util.ArrayList<Key> llaves = new java.util.ArrayList<Key>();
llaves.add(new Key());
...
printItems(llaves);
```

Desde el punto de vista de orientación a objetos, dado que las llaves *son* items, el código anterior debería ser válido (al fin y al cabo, en un `java.util.ArrayList<Item>` puedo meter `Key`s). Sin embargo, el código anterior *no compila*. La razón es que ante la llamada a `printItems` el compilador *no puede garantizar* que éste no va a intentar utilizar el vector de `Item`s de otra forma. ¿Qué ocurriría si la implementación de `printItems` *añadiera* un `Item` nuevo? El método `printItems` sería válido, pues al fin y al cabo el vector es de `Item`s, pero como le estamos pasando un vector que sólo debería aceptar `Key`s, nos añadiría un elemento que estropearía el tipo. Para evitar ese peligro potencial, la llamada no es correcta.

La solución es utilizar *comodines*. El método `printItems` puede recibir un vector de la siguiente forma:

```
void printItems(java.util.ArrayList<? extends Item> v) {
  for (Item i : v)
     System.out.println(i);
}
```

es decir cualquier vector que tenga objetos del tipo `Item` o cualquier clase que herede de ella. Como en la práctica el tipo real no se conoce por `printItems` (está marcado como `?`) cualquier intento de añadir elementos al vector dará un error de compilación. Eso implica que el compilador puede asegurar que la llamada a `printItems` con un vector de *Key*s no romperá el tipo y la dará por válida.

Plantillas ("genéricos" en C++)

Mucho antes de que en Java añadieran los genéricos, C++ ya disponía de un mecanismo "similar": las plantillas o *templates*. La capacidad expresiva de las plantillas es superior a la de los genéricos principalmente a que la compilación de la clase parametrizada se retrasa hasta el momento de su uso o instanciación, lo que permite utilizar métodos y atributos del tipo parámetro e incluso construir nuevos objetos del tipo del parámetro (algo que veíamos que no puede hacerse en Java y sus genéricos[7]). Hay una comparativa completa en la wikipedia al respecto: `http://en.wikipedia.org/wiki/Comparison_of_Java_and_C`

Limitándonos a lo que afecta a la programación de colecciones, diremos que en C++ la sintáxis de una clase similar a la clase `Inventario` sería:

```
template<typename Elem>
class Inventario {
public:
   Inventario() { ... };

   Inventario(int tamInicial) { ... };
```

[7]Eso hace que el proceso de compilación sea más complicado y los errores de compilación provocados por un mal uso de las plantillas mucho más difíciles de entender.

```
    int getNumItems() const { ... };

    void addItem(Elem i) { ... }; // mejor const Elem &

    Elem getItem(int i) { ... }; // mejor const Elem &

private:

    ...
};
```

Es decir, en vez de indicar el parámetro de la plantilla/genérico inmediatamente después del nombre, se antecede el `class` con la palabra clave `template` seguido, entre los ángulos, del nombre del parámetro que se utilizará después en la declaración/definición.

Para utilizar la clase la sintáxis es similar a la de Java:

```
Inventario<Item> i1;
Inventario<int> i2;
```

Colecciones en Java

Aunque en las secciones anteriores hemos estado utilizando la clase `ArrayList` como ejemplo, la realidad es que esa clase se mantiene en la biblioteca por compatibilidad hacia atrás desde la versión 1.0 de Java. Se ha ido modificando para acomodar nuevos métodos y revisiones de las estructuras de datos.

En las últimas versiones, se ha definido lo que se denomina armazón de colecciones o *collections framework*. Forman parte de este framework dos jerarquías, una que tiene como raíz la interfaz *Collection* (no confundir con el framework) y otra que tiene como raíz la interfaz *Map*. Como interfaces que son todas, existen implementaciones de las mismas que se irán introduciendo brevemente.

Jerarquía Collection

`Collection` es una interfaz que representa una colección de objetos en la que se pueden añadir o eliminar, así como preguntar si un objeto está y algunas otras operaciones.

De `Collection` heredan dos interfaces:

- `List`: también una interfaz representa colecciones de objetos que tienen *orden* (en el sentido de que se distingue el primer elemento, etc.) y permite, por tanto, acceder al elemento i-ésimo de la colección. Hay dos implementaciones de esta interfaz (además del `ArrayList`): `LinkedList`, que utiliza listas enlazadas de nodos (con todas sus desventajas y, ocasionalmente, ventajas), y `Vector`, antecesora de `ArrayList` y preservada por motivos de compatibilidad.

- `Set`: igual que `List` es una interfaz que hereda de `Collection`. En este caso, sin embargo, estas colecciones deben verse, como su nombre indica, como *conjuntos*: si se añade a la colección un objeto que ya estaba insertado no se vuelve a añadir. Hay dos implementaciones distintas, `HashSet`[8] y `TreeSet`[9] que utilizan tablas y árboles

[8] Usa el método *equals* y el *hashCode* del objeto.

[9] Usa el método *equals*; y requiere que haya un comparador definido para ver si un objeto va antes o después de otro.

respectivamente[10].

Maps

Los vectores y arrays pueden verse como colecciones que permiten seleccionar un elemento concreto a partir de un entero que indica su posición. Esta idea puede generalizarse al uso de otro tipo distinto a un entero. Eso son lo que en estructuras de datos se conocen como tablas dispersas y en Java se llaman `map`s. La idea es tener algo así[11]:

```
//Ojo no compila en java
Map<String, String> diccionario = new ...;
diccionario["uno"] = "one";
diccionario["dos"] = "two";
diccionario["tres"] = "three";
...
System.out.println("El número dos en inglés es:" +
                   diccionario["dos"]);
```

Se deduce que usar un `Map` equivale a definir pares clave-valor. La clave actúa como índice ("uno", "dos" y "tres" en el ejemplo). El valor a insertar es lo que se asigna al array ("one", "two" y "three").

Vemos que la clase `Map` está parametrizada con dos tipos distintos: el tipo que hace de *clave* ("índice") y el que hace de valor. Para añadir elementos se utiliza `put(clave, valor)` y para obtener un valor a partir de una clave, `get(clave)`.

Existen dos implementaciones: `HashMap` y `TreeMap`. A nuestro nivel, se pueden usar indistintamente. Con esta nueva información, el ejemplo anterior quedaría así:

```
Map<String, String> diccionario = new HashMap<String,String>();
//Map<String, String> diccionario = new TreeMap<String,String>();
diccionario.put("uno","one");
diccionario.put("dos","two");
diccionario.put("tres","three");

System.out.println("El numero dos en "+
  "ingles es "+diccionario.get("dos"));
```

Iteradores

Una ventaja de las colecciones es que se puede generalizar su recorrido. De momento, se han recorrido los arrays usando un índice dentro de un bucle *for* o *while*. Existe una forma más reutilizable. La interfaz *Collection* incluye un método *iterator()* que proporciona un iterador. La interfaz *Map* no tiene un iterador al uso, pero permite obtener instancias de *Collection* para las claves y los valores.

Un iterador en Java sintetiza lo que se espera de un bucle usando la interfaz *Iterator*. La interfaz *Iterator* está parametrizada con el mismo tipo que estuviera la colección en cuestión. Tiene dos métodos: uno para saber si hay que continuar y otro para obtener el siguiente.

[10] Igual que con el `ArrayList` y el `LinkedList`, las diferencias entre `HashSet` y `TreeSet` radican en la complejidad de cada una de las operaciones, y es un tema que se estudia en las asignaturas relacionadas con estructuras de datos.

[11] Esta sintáxis no puede utilizarse en Java, aunque sí en las clases equivalentes en C++.

```
ArrayList<Item> is = new ArrayList<Item>();
Iterator<Item> it = is.iterator();
while (it.hasNext()) {
   Item i=it.next();
   System.err.println(i);
}
```

El bucle del iterador es igual sea cual sea la colección considerada.

Es mucho más cómodo, eso sí, iterar usando los bucles *foreach*, disponibles desde Java 5:

```
ArrayList<Item> is = new ArrayList<Item>();
for (Item i : is) {
   System.err.println(i);
}
```

Los bucles *foreach* están disponibles para todas las colecciones y para los arrays. Además, se puede hacer que una nueva clase sea compatible con foreach sin más que implementar la interfaz *Iterable*.

Colecciones y maps en C++

La librería de C++ (STL) también dispone de clases similares a las de Java para almacenar colecciones de elementos. Desde su diseño inicial todas estas clases están parametrizadas con el tipo de elementos que almacenan. Existen clases equivalentes al `ArrayList`/`ArrayList`, conjuntos y tablas dispersas. También hay implementaciones de multi-conjuntos, pilas, colas y colas de prioridad.

Capítulo 8
Componentes visuales

First, solve the problem. Then, write the code.
John Johnson

Resumen: En este capítulo se hace una pequeña introducción a la programación de interfaces visuales en general. Introduce también alguna característica del lenguaje Java que es muy utilizada en la programación de este tipo de aplicaciones. Por último muestra un pequeño ejemplo de cómo programar una aplicación con una ventana en Java.

Modelos de interacción usuario-programa

Las aplicaciones que hemos desarrollado hasta ahora seguían una interacción controlada por el programa:

- La aplicación comienza con la ejecución de la primera línea del `main` y va ejecutando todas las instrucciones hasta el final.
- En cada momento la aplicación solicita al usuario lo que se precisa (por ejemplo, una opción de un menú, la siguiente acción a ejecutar en un juego, etc.).
- Si la aplicación no necesita la intervención del usuario, se ejecutará de principio a fin sin interrupciones.

Este modelo de programación es *sencillo* (sólo cuando el programa cree que necesita algo del usuario le pregunta), pero es más rígido desde el punto de vista del usuario, por lo que da lugar a programas menos *usables*.

La contrapartida a este modelo de programación se tiene cuando la interacción es controlada *por el usuario*. Pensemos un momento, ¿qué está ejecutando el *Word* cuando el usuario no está haciendo nada con él?

Este modelo de interacción:

- Dota al usuario de total libertad. En cada momento éste es libre de realizar cualquier acción sobre todos los *controles* (botones, menús, barras de herramientas, etc.) ofrecidos por el programa.

- La aplicación ya *no* tiene el control sobre cuándo el usuario puede actuar, pues se intenta que el usuario pueda interactuar en todo momento.
- La consecuencia directa es que no existe un flujo de ejecución determinado, pues el orden en el que se ejecutarán las distintas funcionalidades dependerá siempre del usuario.

Este tipo de interacción, típico de las aplicaciones con interfaces gráficas, tiene como ventaja la usabilidad, al ser el usuario el eje central de ejecución. La desventaja es que su programación es más complicada, debido a la *inversión de control*: ahora el control no lo tiene el programa, sino el usuario.

Programación dirigida por eventos

El modelo de interacción utilizado con las interfaces gráficas se basa en la *programación dirigida por eventos*. Entendemos por evento cualquier suceso que puede ser relevante para la aplicación, como por ejemplo que el usuario pulse el botón *Aceptar* o seleccione la opción *Guardar* del menú *Archivo*.

La programación dirigida por eventos consiste en proporcionar los métodos que respondan a dichos mensajes.

Por ejemplo, imaginemos que tenemos una ventana que simplemente muestra un botón con el texto "`Suena`". Querríamos que cuando se pulse se ejecute el siguiente código:

```
java.awt.Toolkit.getDefaultToolkit().beep();
```

La idea detrás de los eventos consiste en que la clase que implementa el botón (pongamos que es la clase `Boton`) permite a otros *registrarse* para ser avisados cuando el usuario pulse el botón. Algo así:

```
// Interfaz que implementarán todos aquellos que quieran
// enterarse cuando un botón es pulsado por el usuario
public interface OyenteBoton {
   public void botonPulsado();
}
```

```
public class Boton {
   // Registra un nuevo oyente que será informado
   // cuando el usuario pulse el botón
   public void registraOyente(OyenteBoton o) {
      ...
   }
}
```

```
public class Suena implements OyenteBoton {
   public void botonPulsado() {
      java.awt.Toolkit.getDefaultToolkit().beep();
   }
}
```

En la práctica un método del tipo `botonPulsado` recibe como parámetro un objeto con información sobre el evento; por ejemplo qué botón se pulsó o la posición concreta donde estaba el ratón cuando el botón se pulsó.

En general la programación basada en eventos se basa en tres patas fundamentales:

- Hay un mecanismo de *delegación*: la responsabilidad de gestionar/reaccionar ante un evento en un objeto (por ejemplo un botón) la tiene otro objeto distinto (el *oyente*). La fuente del evento (el botón) no se preocupa en comprobar quién reacciona ante el evento.
- Existe el concepto de *evento*: el oyente recibe el evento en un objeto que encapsula la información del evento.
- Los eventos se crean en la fuente y se propagan a los oyentes para que tomen las acciones que crean oportunas.

Lo normal en las aplicaciones con interfaces visuales es que esos componentes (botones, menús, listas desplegables, ...) actúen como fuentes de generación de eventos; los oyentes serán clases creadas por el programador que también pueden ser otros componentes gráficos (por ejemplo, cuando se pulsa un botón de una calculadora en la "pantalla" aparece el número).

El modelo de eventos permite:

- Que varias clases distintas se registren como oyentes del mismo evento. Por ejemplo, en una calculadora podemos registrar como oyente de la pulsación de botones a la pantalla donde van apareciendo los números y a una clase que hace que suene un corto pitido.
- Varias clases pueden emitir el mismo evento. Por ejemplo, el evento de "Guardar documento" puede generarse cuando se pulsa la opción *Guardar* del menú *Fichero*, cuando se pulsa `Ctrl-S` o cuando el usuario pulsa el pequeño icono con forma de disco de la barra de herramientas.

La existencia de los *oyentes* anteriores implica la creación de distintas clases que implementan las interfaces exigidas por las fuentes (como el del ejemplo, `OyenteBoton`). Para evitar la proliferación de muchas clases en distintos ficheros, los lenguajes como Java permiten *anidamiento* de clases. Antes de introducirnos en la creación de interfaces de usuario con Java, veamos esta característica del lenguaje que hemos mantenido ignorada hasta ahora.

Clases dentro de clases en Java

Las clases (e interfaces) "normales" están declaradas en el nivel más externo (*top level*), es decir, a nivel *de paquete*. Sin embargo, existen otros cuatro tipos de clases (o interfaces). En la nomenclatura inglesa son:

- Top-level nested classes and interfaces.
- Non-static Inner Classes
- Local Classes
- Anonymous Classes

Las iremos viendo una a una.

Clases e interfaces anidados a nivel de paquete

También llamadas "top-level nested classes and interfaces", consiste en la declaración de clases *dentro* de otras clases y etiquetarlas como `static` (para el caso de las interfaces no hace falta el `static` pues se asume por defecto).

Una clase puede tener más de una clase interna que, a su vez, puede tener más clases o interfaces.

En el siguiente ejemplo la clase `TopLevelClass` contiene una clase interna que, a su vez, tiene una interfaz y una clase que implementa esa interfaz:

```
public class TopLevelClass {
  // ...
  static class NestedTopLevelClass {
    // ...
    interface NestedTopLevelInterface1 {
      // ...
    }

    static class NestedTopLevelClass1
       implements NestedTopLevelInterface1 {
         // ...
    }
  }
}
```

Las clases e interfaces internos pueden declararse con los tipos de visibilidad ya vistos (`public`, `protected`, etc.). Cuando se compila un fichero `.java` que tiene clases internas el compilador en vez de generar un único `.class`, generará uno *por cada clase* (o interfaz) que haya en el fichero. Los nombres de esos `.class` se formarán a partir de los nombres de las clases, separándolos por el `$`. Para el ejemplo anterior, tendremos:

```
TopLevelClass.class
TopLevelClass$NestedTopLevelClass.class
TopLevelClass$NestedTopLevelClass$
                        NestedTopLevelClass1.class
TopLevelClass$NestedTopLevelClass$
                        NestedTopLevelInterface1.class
```

Si la visibilidad lo permite, las clases internas se podrán utilizar desde otras clases. Para referirse a ellas se utiliza el `.` para separar el nombre de la clase externa y el de la interna. Por ejemplo:

```
TopLevelClass.NestedTopLevelClass.NestedTopLevelClass1 obj;
TopLevelClass.NestedTopLevelClass.NestedTopLevelInterface1 obj2;
```

Para ganar legibilidad y evitar repeticiones, se puede utilizar la cláusula `import` que hasta ahora habíamos vinculado únicamente a paquetes para importar las clases e interfaces internos a una clase:

```
import TopLevelClass.*;

public class Client {
  NestedTopLevelClass.NestedTopLevelClass1 objRef1 =
```

```
      new NestedTopLevelClass.NestedTopLevelClass1();
}
```

```
import TopLevelClass.NestedTopLevelClass.*;

public class Client2 {
  NestedTopLevelClass1 objRef1 = new NestedTopLevelClass1();
}

class SomeClass implements NestedTopLevelInterface1 {
  // ...
}
```

Existen, además, unos criterios de visibilidad especiales. Se entiende que una clase interna a otra es una clase "especial" que tiene acceso a atributos a los que no podría acceder en otras condiciones. En concreto, los métodos de una clase interna pueden tener acceso a los métodos y atributos *estáticos* de la clase externa. A continuación aparece un nuevo ejemplo de clases; las líneas marcadas con `// X` dan error de compilación.

```
public class MiClase {

  public void noEstatico() {
     System.out.println("A");
  }

  // Clase estática anidada
  public static class Anidada {

    private static int i;
    private int j;

    public static void estatico() {
       System.out.println("B");
    }

    // Interfaz interno
    interface IntfcAnidado {
       int Y2K = 2000;
    }

    public static class AnidadaIn1 implements IntfcAnidado {

      private int k = Y2K;

      public void otroNoEstatico() {
        int jj = j;  // X Error: intento de acceso a
                     // atributo no estático de clase externa
        int ii = i;
        int kk = k;
        noEstatico(); // X Error: intento de acceso a método
                     // no estático de clase externa
        estatico();
      }

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

```
            int ii = i;
            int kk = k; // X Error: intento de acceso atributo
                        // no estático dentro de método estático
            estatico();
        }
    }
  }
}
```

Como vemos en el ejemplo, las clases internas pueden contener también un método `main` por lo tanto pueden lanzarse desde fuera. Para indicar que se quiere ejecutar una clase interna se utiliza el `$` desde la línea de órdenes (no se puede utilizar el `.` para separar porque la JVM lo interpretaría como separador de paquetes y no encontraría la clase):

```
java MiClase$Anidada$AnidadaIn1
```

Clases internas no estáticas

Son parecidas a las anteriores pero sin utilizar el modificador `static` cuando se definen. La principal diferencia es que los objetos de estas clases *sí* pueden acceder a los atributos no estáticos de la clase externa. La pregunta obvia es ¿de qué objeto? Es decir, si desde un objeto de la clase `Interna` se puede acceder al atributo `k` de la clase `Externa`, ese atributo pertenecerá a un objeto concreto de la clase `Externa`; ¿a cuál?

La respuesta da otra propiedad de las clases internas no estáticas (también llamadas *non-static inner class*): los objetos de estas clases internas *sólo pueden existir en el contexto de una instancia de la clase externa*. Es decir, *no* podremos crear un objeto aislado de las clases, sino que debe siempre hacerse a partir de otro objeto de la clase externa. Si el objeto de la clase interna accede a los atributos no estáticos de la externa estará leyendo los atributos del objeto concreto sobre el que se creó.

Como ejemplo, podemos tener estas clases:

```
public class MiClase {

    public String msg = "Mensaje";

    public InternaNoEstatica crea() {
        return new InternaNoEstatica();
    }

    public class InternaNoEstatica {

        private String str;

        public InternaNoEstatica() {
            str = msg;
        }

        public void printMsg() {
            System.out.println(str);
            System.out.println(msg);
        }
    }
}
```

Vemos que `MiClase` tiene un método que crea (y devuelve) un objeto de la clase interna. Cuando ese objeto interno accede al atributo `msg` estará accediendo al atributo del objeto desde el que fue creado:

```
public class Cliente {
    public static void main(String args[]) {
        MiClase ref = new MiClase();

        MiClase.InternaNoEstatica interna = ref.crea();
        ref.msg = "Cambio";
        interna.printMsg();
    }
}
```

En el momento de llamar a `crea()`, el atributo `msg` de `ref` vale "`Mensaje`", por lo que el atributo `str` del objeto interno será esa misma cadena. Posteriormente cambiamos el valor del atributo del objeto de la clase externa. Al llamar a `printMsg` se escriben ambos atributos, por lo que la salida es:

```
Mensaje
Cambio
```

Existen otras características de estas clases que merece la pena mencionar:

- No pueden tener miembros `static`.
- Es decir: la clase no proporciona ningún servicio; únicamente *sus instancias* lo hacen.
- Los métodos de la clase interna *pueden acceder* a los miembros de la clase externa (incluidos los privados).
- Se puede definir su accesibilidad (públicas, protegidas...).
- Si la clase interna y la clase externa tienen atributos con el mismo nombre, el interno *tapa* al externo.
- Para acceder al atributo externo, se puede utilizar una *variación* del `this`, con la sintaxis `ClaseExterna.this.atributoExterno`.

Una situación típica en la que resulta natural utilizar una clase interna no estática es cuando implementamos una clase colección, como las que vimos en el Capítulo 7, para la que además queremos definir un iterador. Basándonos en la clase `Inventario` de dicho capítulo podemos definir fácilmente un inventario genérico, de la siguiente forma.

```
public class Inventario<T> {

   private static final int TAM_INICIAL = 10;

   private Object [] v;
   private int next;

   public Inventario(){
      this(TAM_INICIAL);
   }
```

```
    public Inventario(int tamInicial){
        if (tamInicial <= 0)
            tamInicial = 1;
        v = new Object[tamInicial];
        next = 0;
    }

    public int getNumElems() {
        return next;
    }

    public void addElem(T i) {
        if (lleno())
            duplica();
        // siempre podemos asignar un objeto de tipo T
        // a una variable de tipo Object
        v[next] = i;
        next++;
    }

    public T getElem(int i) {
        if ((i < 0) || (i >= next))
            return null;
        // "Type safety: Unchecked cast from Object to T"
        // Inseguridad relativa
        return (T) v[i];
    }

    private boolean lleno() {
        return next == v.length;
    }
    private void duplica() {
        Object [] nuevo = new Object[2 * v.length];
        for (int i = 0; i < v.length; ++i)
            nuevo[i] = v[i];
        v = nuevo;
    }
}
```

En este código ya podemos apreciar algo interesante, pues que en un principio resulta sorprendente que se utilice un array de `Object` en vez de un array de elementos de tipo `T` como tipo para la variable `v`. El motivo de esto es una limitación de los genéricos en Java, que de la misma manera que no permiten invocar la constructora de la variable parámetro, no nos permiten ejecutar la instrucción `new T[tamInicial]`. Esto nos obliga a realizar un casting en el retorno del método `getElem`, que sin embargo no es peligroso ya que el único método público con el que se pueden añadir elementos a `v` es `getElem`, que sólo acepta elementos de tipo `T` como argumento. Por ejemplo el siguiente código no compila porque no se nos permite añadir un objeto de tipo `String` a un inventario donde se almacenan objetos de tipo `Integer`.

```
Inventario<Integer> inventario = new Inventario<Integer>();
inventario.addElem("hola");
```

Ya tenemos un inventario genérico, veremos ahora cómo extender esta clase para que sea capaz de crear sus propios iteradores. La interfaz *Iterable* —que es extendido por

Collection— corresponde a las clases que se comprometen a devolver un iterador para poder recorrerlas, lo que se expresa en el método `Iterator<T> iterator()`, que es el único que contiene esta interfaz. El problema que se nos plantea ahora es que para implementar este método necesitamos devolver un objeto de tipo `Iterator<T>`, para lo que necesitamos una clase que implemente esa interfaz. Además dicha clase debería ser capaz de recorrer en orden los elementos almacenados en un objeto inventario. Aunque cualquier clase podría recorrer los elementos del inventario obteniendo el número de elementos con `getNumElems` y haciendo un bucle llamando a `getElem`, en general eso podría ser bastante ineficiente, dependiendo de las estructuras de datos que se utilicen en la implementación de la clase. Por ejemplo el método `get` de la clase `LinkedList`, que es el equivalente a `getElem`, tiene coste lineal, por lo que un recorrido de un objeto `LinkedList` realizado de esta manera tendría coste cuadrático. La mejor solución por tanto es definir una clase interna no estática `InventarioIterator`, que como tal tendrá acceso a la representación interna de la clase inventario, y cuyas instancias tendrán a su vez al valor concreto de las atributos del objeto inventario que las creó, lo que nos proporciona los ingredientes necesarios para recorrer los inventarios de forma eficiente. En el siguiente código llevamos a cabo estas ideas.

```
public class Inventario<T> implements Iterable<T>{
// el código de antes
...
   public Iterator<T> iterator() {
      return new InventarioIterator() ;
   }

   private class InventarioIterator implements Iterator<T> {
      private int index = 0;

      public boolean hasNext() {
         return index < next;
      }

      public T next() {
         T ret = (T) v[index];
         index++;
         return ret;
      }

      public void remove() {
         ...
      }
   }
}
```

Finalmente, al utilizar este diseño la estructura interna del inventario permanece encapsulada, ya que la clase `InventarioIterator` no es accesible desde fuera de la clase `Inventario`: otros objetos que utilicen el iterador devuelto por un inventario no sabrán a que clase concreta pertenece el iterador, sólo sabrán que es un iterador y que por tanto ofrece los métodos `hasNext` y `next`, lo que por otra parte es lo único que le puede intersar a un objeto que utilice una variable iterador.

```
Inventario<Integer> inventario = new Inventario<Integer>();
inventario.addElem(2);
inventario.addElem(1);
inventario.addElem(3);
```

```
Iterator<Integer> iterator = inventario.iterator();
while (iterator.hasNext())
   System.out.print(iterator.next() + ", ");
```

Igual que ocurre con las clases internas estáticas, en las no estáticas el compilador crea un `.class` independiente cuyo nombre sigue las mismas reglas que en el caso anterior. También se puede utilizar la clausula `import` para evitar declaraciones largas.

Clases locales

Las clases locales son parecidas a las clases no estáticas pero con la peculiaridad de que en vez de declararse en el ámbito de una clase más externa, se declaran en el ámbito de *un bloque* de código, como la implementación de un método o un constructor. Eso implicar que son visibles únicamente *dentro de ese código*.

Dado su carácter privado al bloque, no tiene sentido especificar criterios de visiblidad, pues nunca serán accesibles fuera de ese bloque. Eso tiene dos consecuencias inmediatas: no pueden implementar métodos estáticos y desde fuera del bloque no se podrán crear instancias suyas; únicamente desde dentro del bloque se podrá.

Este tipo de clases suele utilizarse junto con la herencia: la clase local implementa una interfaz o extiende una clase que sí es accesible desde fuera y el bloque devuelve un objeto de esa clase padre o interfaz.

Por ejemplo, podemos tener una interfaz:

```
public interface IDrawable {
  void draw();
}
```

De forma que un cliente puede almacenar una colección de objetos que implementan esta interfaz para dibujarlos todos en un recorrido:

```
public class Cliente {
  public static void main(String args[]) {
    ArrayList<IDrawable> v = new ArrayList<IDrawable>();

    for (IDrawable d : v)
      d.draw();
}
```

Un ejemplo de implementación de la interfaz anterior puede ser algo tan sencillo como esto:

```
public class Shape implements IDrawable {
  public void draw() {
    System.out.println("Dibujando una forma.");
  }
}
```

Ahora bien, podemos tener una clase `Pintor` que cree objetos que implementan la interfaz como un círculo unitario o un mapa. En vez de implementar esas clases de formas en ficheros independientes o clases internas, las podemos hacer locales a los métodos respectivos:

```
class Pintor {
  public Shape crearCirculoUnitario() {
```

```
    class Circle extends Shape {
      public void draw() {
        System.out.println("Dibujando un círculo de radio 1");
      }
    }
    return new Circle();
  }

  public static IDrawable crearMapa() {
    class Map implements IDrawable {
      public void draw() {
        System.out.println("Dibujo de un mapa");
      }
    }
    return new Map();
  }
}
```

Es cierto que en el ejemplo anterior las clases locales no aportan expresividad adicional al lenguaje, pues se podrían haber utilizado clases internas (no locales). Sin embargo, existe una característica de las clases locales que *no* está disponible en las clases internas no estáticas: las implementaciones de los métodos de las clases locales *pueden acceder a parámetros y variables locales del bloque*, siempre y cuando éstas hayan sido declaradas como `final`. En el ejemplo siguiente tenemos un método que crea un círculo cuyo radio se pasa como parámetro al propio método. La implementación del método `draw` accede al valor de ese parámetro. Eso significa que cuando el usuario llame posteriormente al método `draw`, éste accederá a un valor cuyo tiempo de vida estaba en principio limitada al momento de la invocación al método de creación del círculo:

```
class Pintor {
  public Shape crearCirculo(final float radio) {
    class Circle extends Shape {
      public void draw() {
        System.out.println(
                "Dibujando un círculo de radio " + radio);
      }
    }
    return new Circle();
  }
}
```

Con este tipo de clases el programador podría crear clases con el mismo nombre en dos bloques de código de la misam clase. Eso implica que el mecanismo de nombres dado a los `.class` por el compilador ya no es válido, pues habría conflicto de nombres. Para evitarlo, el compilador introduce un número que incrementa en caso de ser necesario. Las clases anteriores se verían reflejadas en los ficheros:

```
Pintor$1$Circle.class
Pintor$1$Map.class
Pintor$2$Circle.class
```

Clases anónimas

La última vuelta de tuerca de las clases locales es *quitarles el nombre*. Hemos visto que las clases locales anteriores sólo pueden utilizarse en el contexto del bloque donde están declaradas. En mayoría de los casos el uso que se hace de las clases locales es tan localizado que se declara la clase únicamente para crear un objeto de ella y utilizarlo. Para esos casos tan habituales pueden utilizarse las clases anónimas.

Una clase anónima es, pues, una clase sin nombre que, siguiendo las mismas reglas que en las clases locales, se definen e instancian a la vez. Eso implica que sólo se puede crear una instancia de la clase[1].

Las clases anónimas no pueden tener constructores (al fin y al cabo no tienen nombre...), y siempre extenderán de otra clase o implementarán una interfaz, de forma que la definición sobreescribe o implementa esos métodos que serán los que puedan llamarse desde fuera.

Para la definición de estas clases se utiliza una sintáxis extendida del operador `new`. El ejemplo anterior del `Pintor` haciendo uso de clases anónimas queda:

```
class Pintor {
  public Shape crearCirculoUnitario() {
    return new Shape() {
      public void draw() {
        System.out.println("Dibujando un círculo de radio 1");
      }
    }
  }

  public static IDrawable crearMapa() {
    return new IDrawable() {
      public void draw() {
        System.out.println("Dibujo de un mapa");
      }
    }
  }
}
```

Dado que no tienen nombre, los ficheros finales `.class` son:

```
Pintor$1
Pintor$2
```

Primeros pasos con Swing

En Swing las ventanas están representadas por la clase `JFrame` del paquete `javax.swing`[2]. Se puede crear una de estas ventanas indicando en el constructor el texto que aparecerá en la barra de título y, tras establecer su tamaño, ponerla visible:

```
import javax.swing.JFrame;

public class Ventana {
```

[1] Nos estamos refiriendo a una instancia *por invocación*; evidentemente si el método es llamado varias veces, se crearán más objetos.

[2] En general, todas las clases de Swing están en ese paquete o subpaquetes.

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

   JFrame ventana = new JFrame("Mi primera ventana");
   ventana.setSize(320, 200);

   ventana.setVisible(true);
    }
}
```

El ejemplo (que aunque funciona no es del todo correcto, como veremos enseguida) pone de manifiesto el cambio en la forma de programar con interfaces visuales: tras ejecutar la última línea del `main` (que provoca que la ventana aparezca), la aplicación *no termina*. La ejecución del `main` acaba pero la aplicación sigue ejecutándose, con la ventana visible. De hecho, la aplicación se mantiene en ejecución incluso después de cerrar la ventana. Para evitar este extraño comportamiento se debe configurar el `JFrame` de forma que su cierre provoce el final de la aplicación, utilizando el método `setDefaultCloseOperation`.

Aunque el código anterior funciona, es preferible realizar la llamada al `setVisible` utilizando la hebra de trabajo de Swing/AWT. La razón se entenderá más adelante.

Con estas dos modificaciones, el ejemplo queda:

```
import javax.swing.JFrame;
import java.awt.EventQueue;

public class Ventana {

    public static void main(String []args) {

   final JFrame ventana = new JFrame("Mi primera ventana");
   ventana.setSize(320, 200);
   ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   EventQueue.invokeLater(new Runnable() {
      public void run() {
          ventana.setVisible(true);
      }
       });
    }
}
```

Las principales diferencias son:

- El nuevo `import` al paquete `java.awt` para poder acceder a `EventQueue`.
- La llamada al método `setDefaultCloseOperation` para que cuando el `JFrame` se cierre, éste termine la aplicación.
- El cambio a `final` de la variable `ventana` para poder acceder a ella desde la clase anónima utilizada como parámetro de `invokeLater`.

El ejemplo anterior, como hemos visto, crea un objeto de la clase `JFrame`, cambia sus propiedades y lo hace visible. Otra alternativa muy utilizada es *crear una nueva clase* que hereda de `JFrame` de forma que en el constructor se configuran todas sus propiedades. En ese caso el `main` tiene simplemente que crear el objeto de la nueva clase y mostrarlo:

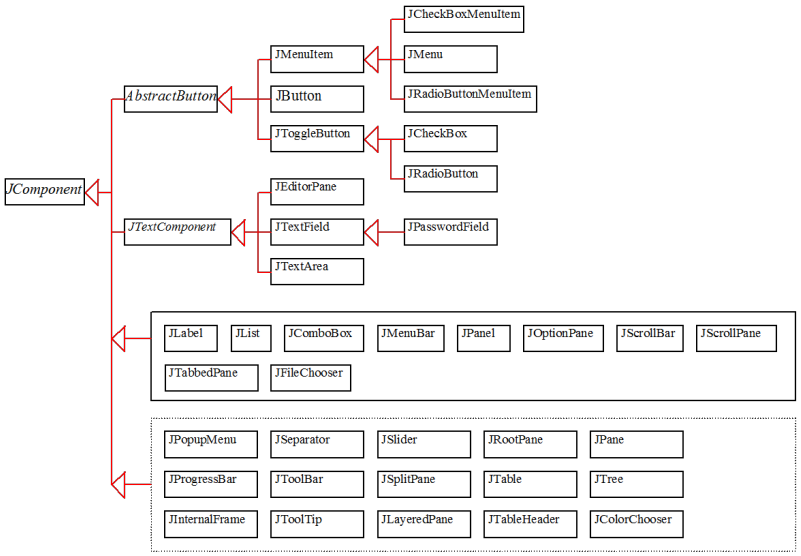
```
import javax.swing.JFrame;

public class MiPrimeraVentana extends JFrame {

    public MiPrimeraVentana () {
   super("Mi primera ventana");
   this.setSize(320, 200);
   this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Y el `main` de la aplicación (que podrían en realidad estar en la propia clase de la ventana):

```
import java.awt.EventQueue;

public class DemoMiPrimeraVentana {

    public static void main(String []args) {

   final MiPrimeraVentana v = new MiPrimeraVentana();

   EventQueue.invokeLater(new Runnable() {
      public void run() {
          v.setVisible(true);
      }
       });
    }

}
```

Añadiendo un componente

El siguiente paso en la creación de interfaces es añadir en la ventana *componentes*. Entendemos por componente aquellos elementos básicos que conforman la interfaz de usuario (o GUI), como botones, listas desplegables, check boxes, etc. Todos ellos heredan de la clase `JComponent`.



Por ejemplo podemos añadir un botón (clase `JButton`) con la idea de que cuando se pulse suene un pequeño pitido. La forma más fácil de añadir un componente a una ventana es utilizar el método `add`:

```
import javax.swing.*;

public class MiPrimeraVentana extends JFrame {

    public MiPrimeraVentana () {
   super("Mi primera ventana");
   this.setSize(320, 200);
   this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   JButton boton = new JButton("Pita!");
   this.add(boton);
    }
}
```

El último paso es asociar al componente un *oyente* que será avisado cuando el usuario pulse el botón. Aunque las clases involucradas dependen del tipo de evento, la mayoría de las veces el evento será del tipo `ActionEvent` y para capturarlo deberemos implementar la interfaz `ActionListener` (ambos están en el paquete `java.awt.event`):

```
import java.awt.event.*;

public class PitaCuandoSePulse implements ActionListener {
    public void actionPerformed(ActionEvent event) {
   java.awt.Toolkit.getDefaultToolkit().beep();
    }
}
```

En este ejemplo sencillo no utilizamos el parámetro `event`. En otros usos más sofisticados se puede obtener información sobre el tipo de evento que se ha producido, permitiendo que el mismo oyente capture eventos distintos a los que distingue gracias al parámetro.

La configuración del botón es tan simple como:

```
...
```

```
JButton boton = new JButton("Pita!");
boton.addActionListener(new PitaCuandoSePulse());
this.add(boton);
...
```

No obstante, en vez de crear una clase por cada tipo de evento que se quiere gestionar muchas veces se utilizan las clases anónimas explicadas anteriormente:

```
import javax.swing.*;
import java.awt.event.*;

public class MiPrimeraVentana extends JFrame {

    public MiPrimeraVentana () {
   super("Mi primera ventana");
   this.setSize(320, 200);
   this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   JButton boton = new JButton("Pita!");

   boton.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent event) {
           java.awt.Toolkit.getDefaultToolkit().beep();
            }
   });

   this.add(boton);
    }
}
```

Diseño y creación de la interfaz gráfica

En el ejemplo de la ventana con el botón anterior nos hemos limitado a añadir el botón y éste se ha colocado automáticamente. En particular, ha ocupado todo el espacio disponible. Para entender el por qué de este funcionamiento es necesario conocer los tres elementos esenciales de una interfaz gráfica:

- Componentes gráficos: ya los hemos explicado. Son los elementos que heredan de `JComponent` como botones, listas desplegables, etc.
- Contenedores (containers): tienen como tarea *agrupar* el resto de elementos gráficos. Un ejemplo de contenedor es la ventana principal de la aplicación. Un contenedor puede tener componentes o incluso otros contenedores internos. Por ejemplo una barra de herramientas puede verse como un contenedor (que agrupa a todos los botones de la barra) que está dentro de un contenedor más grande (el de la ventana principal).
- Administradores de distribución (o *layout managers*): determinan la estrategia con la que los distintos elementos de un contenedor se colocan en él. Los contenedores se configuran para que utilicen uno de estos administradores dependiendo de lo que queramos conseguir. La ventaja de utilizar administradores es que mantienen los componentes bien colocados (y con tamaños correctos) aún cuando el contenedor

cambie de tamaño (por ejemplo cuando ampliamos o disminuimos el tamaño de la ventana).

Administradores de distribución

Como ya hemos dicho son los responsables de organizar los componentes en el panel. De hecho, son los que eligen la mejor posición y tamaño de cada componente de acuerdo al espacio disponible. Para eso tienen en cuenta las *preferencias* de cada componente (gracias al método `getPreferredSize()` disponible en todos ellos). En el ejemplo que hemos visto el espacio disponible es el tamaño de la ventana que hemos establecido nosotros con

```
this.setSize(320, 200);
```

Sin embargo es posible utilizar el propio administrador para que determine cuál es el mejor tamaño de la ventana de acuerdo con los componentes contenidos en el contenedor.

Todos los contenedores tienen un método `setLayout` que permite configurar qué administrador utilizar. En Swing hay varios tipos, los más utilizados:

- `BorderLayout`: organiza el contenedor en cinco zonas: norte, sur, este, oeste y zona central. Cuando se añade un componente se puede indicar en qué zona queremos que aparezca.
- `FlowLayout`: coloca los componentes de izquierda a derecha y de arriba a abajo.
- `GridLayout`: sirve para distribuir los componentes en una cuadrícula. En el constructor se especifican el número de filas y de columnas en los que estará dividido el contenedor, todos ellos del mismo tamaño.

Para construir una ventana se pueden utilizar distintas combinaciones de paneles y gestores de distribución; por ejemplo la ventana principal puede utilizar el `BorderLayout` donde en la zona norte hay un contenedor (`JPanel`) con un `FlowLayout` y en la zona sur un `Grid Layout`.

Componentes

Existen muchos componentes que pueden ser utilizados para crear la ventana. Como usuarios estamos acostumbrados a verlos y utilizarlos. Cuando diseñamos una interfaz de usuario debemos colocarnos siempre en el lugar del usuario haciendo cómoda la interacción. A continuación aparecen algunos de los componentes disponibles en Swing:

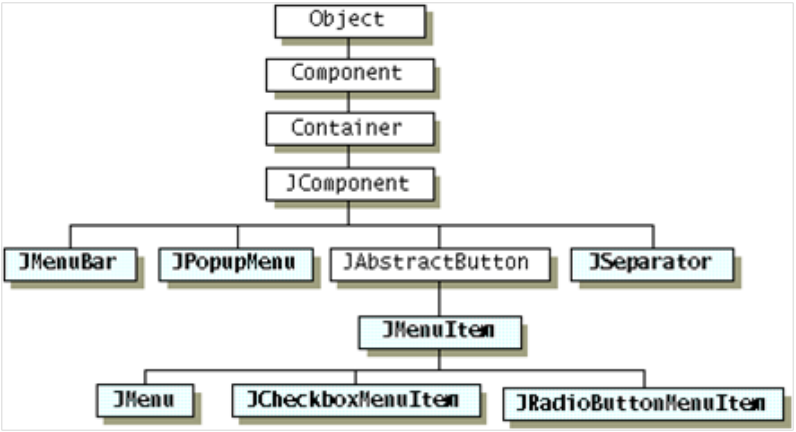
- `JButton`: los botones tradicionales.
- `JCheckBox`: representa una opción que puede aparecer activada o desactivada. Para eso utiliza un pequeño cuadro en el que aparece dentro un pequeño punto si la opción está activada.
- `JRadioButton`: similar al anterior pero utilizando un círculo. El usuario normalmente entiende que esa opción forma parte de un grupo excluyente de opciones, de forma que si una de ellas está activa, el resto estarán desactivadas.

- `JTextField`: un pequeño cuadro que permite al usuario añadir texto de una sola línea; se utiliza, por ejemplo, en navegadores para indicar la dirección URL.
- `JTextArea`: área de texto similar a la anterior pero que permite añadir más de una línea. El bloc de notas se puede ver como uno de estos componentes.
- `JLabel`: una simple etiqueta en una ventana. La ventana que muestra las propiedades de un archivo en el explorador de Windows está formada principalmente por este tipo de componentes.
- `JList`: lista de elementos.
- `JComboBox`: similar a la anterior, pero sólo una puede estar seleccionada. En condiciones normales sólo aparece la opción seleccionada, pero hay un botón para mostrar el resto y poder seleccionarlo. Un ejemplo es el que permite establecer el tamaño de la fuente a utilizar en programas como Word.
- `JTree`: permite presentar información en forma de árbol. Por ejemplo, el árbol de directorios de un explorador.

Menús

Los menús de la ventana son un tipo especial de componente que se maneja de forma distinta. A pesar de que las clases utilizadas para gestionar los menús heredan de `JComponent` *no* todos los contenedores pueden tener menús. En concreto, las barras de menú típicas que aparecen en las ventanas no las podemos asociar a cualquier contenedor sino únicamente a los `JFrame`[3].

La jerarquía de clases es la siguiente:



Los más utilizados son:

- `JMenuBar`: representa una barra de menú de una ventana; es un contenedor de menús. La clase `JFrame` tiene un método `setJMenuBar` para establecer la barra de menús que se utilizará en esa ventana (sólo puede haber una).
- `JMenu`: representa un menú (por ejemplo el "menú Archivo"). Contiene las distintas *opciones* del menú (objetos de la clase `JMenuItem`).

[3]Bueno, también a otro par de clases de Swing que no hemos tratado aquí, `JApplet` y `JDialog`.

- `JMenuItem`: representa una opción en el menú. Normalmente la opción será un simple texto que, al pulsarse, activa o ejecuta la opción. Sin embargo la opción también puede tener un *check box* (como el de Eclipse para seleccionar si el proyecto se compila automáticamente) o un *radio button* que permite tener activa en exclusión mutua una opción dentro de un grupo.

  Dado que la clase `JMenu` *hereda* de `JMenuItem`, las opciones pueden ser a su vez otros menús, para crear menús anidados (o submenús).

- `JSeparator`: es una opción especial de un menú que añade una línea de separación entre opciones. Útil para agrupar opciones comunes.

- `JPopupMenu`: sirve para implementar menús contextuales que se activan al pulsar el botón derecho sobre otro componente.

La forma de capturar los eventos es similar a las de los controles ya tratados. Basta con registrarse como oyente (método `addActionListener`) e implementar la interfaz `ActionListener`[4].

Cuadros de diálogo

En ocasiones las aplicaciones de ventanas renuncian al modelo de eventos liberal en el que el usuario tiene el completo control de qué acciones realizar y presentan al usuario una ventana de la que no puede escaparse y cuya interacción está muy limitada. Por ejemplo, cuando el usuario intenta salir de una aplicación sin haber guardado los últimos cambios aparece una ventana de advertencia para que confirme que desea salir; la aplicación no le permitirá hacer otra cosa más que contestar a la pregunta de confirmación. Lo mismo ocurre con otros escenarios, como una ventana que muestra un aviso y espera a que el usuario lo lea y pulse el botón "Aceptar", etc.

Esos son los que se conocen como *cuadros de diálogo* que son ventanas especiales que heredan de `JDialog` y que son mostradas por otra aplicación de ventana con el objetivo de tratar con algún aspecto específico sin "desordenar" la ventana original con esos detalles.

Dado que los cuadros de diálogo se utilizan mucho, existen en Swing varios "prefabricados". En concreto:

- `JOptionPane`: una clase con métodos estáticos para mostrar cuadros de diálogo sencillos, como cuadros para confirmar una acción (con botones como Si/No/Cancelar), cuadros para mostrar un mensaje con un botón de Aceptar, o para pequeñas entradas de datos, como un nombre.

- `JFileChooser`: permite al usuario seleccionar uno (o varios) archivos. Se puede configurar para indicar qué tipo de extensión se debe permitir seleccionar, etc.

- `JColorChooser`: similar al anterior, pero para seleccionar un color. No son tan utilizados como los anteriores.

- `PrinterJob`: cuadro de diálogo para impresión.

[4] En realidad hay una incostencia, porque los `JCheckboxMenuItems` utilizan una interfaz distinto, el `ItemListener`...

Capítulo 9

Diseño orientado a objetos[1]

Cuanto antes comiences a codificar, más tiempo tardarás en finalizar el programa

Roy Carlson

Resumen: En este capítulo se hace una pequeña incursión en el diseño orientado a objetos, dando algunas claves para la creación de una buena estructura de clases que ayude a programar la aplicación. La primera parte del tema repasa los conceptos ya conocidos de orientación a objetos desde un punto de vista general (completamente independiente del lenguaje), para pasar posteriormente a dar algunos aspectos clave a tener en cuenta en el momento de enfrentarnos a un nuevo diseño de una aplicación.

Orientación a objetos

La orientación a objetos es la unión de un conjunto de datos y de las funciones que operan sobre ese conjunto de datos. Esa separación consigue que el cliente utilice los datos a través de una interfaz de funciones que operan sobre ellos.

De esta forma, cuando se realiza un diseño orientado a objetos (*DOO*), manejamos las clases teniendo en cuenta *qué hacen* los objetos y no *cómo lo hacen*.

La orientación a objetos utiliza una filosofía distinta a la programación imperativa/modular tradicional. En la programación imperativa los datos y los procedimientos/funciones se consideran de forma separada, de forma que las funciones se transmiten datos entre ellas y operan sobre los mismos. Una aplicación se divide entonces en módulos de procedimientos con funcionalidad similar. En OO sin embargo los programas son agrupaciones de objetos que interactúan entre ellos transmitiéndose órdenes e información.

La orientación a objetos facilita por un lado la reutilización de código (para no "reinventar la rueda" en cada nueva aplicación) y la extensibilidad (pues el código generado es más resistente a los cambios).

Los conceptos básicos de la OO son (algunos de ellos aparecen también en la programación modular):

[1] El contenido de este tema está extraído casi totalmente de un material previo de Guillermo Jimenez Díaz.

- Abstracción: oculta los detalles del proceso con el fin de mostrar más claramente otros aspectos, en concreto las clases crean un modelo simplificado de la realidad de forma que en el momento del diseño no nos perdemos en los detalles de implementación pues trabajamos con los *conceptos* que representan esas clases.

- Encapsulación: restringe el acceso a los datos y al comportamiento, definiendo el conjunto de mensajes que un objeto es capaz de recibir. La encapsulación es la responsable de que la abstracción no se pueda "saltar"; la abstracción oculta los detalles de implementación y la encapsulación evita el acceso externo a ellos.

- Objetos: son las entidades que se responsabilizan de la ejecución de los "servicios". Es típico hacer una *antropomorfización* de ellos, y verlos como "agentes" capaces de proveer servicios a una comunidad (los usuarios de esa clase/objeto). Cada agente se especializa y acude a otros grupos de objetos para distribuir el trabajo.

  Los objetos, pues, tienen una serie de responsabilidades o servicios que ofrecen y que son solicitados por los clientes externos.

  Para realizar esos servicios, mantienen un estado interno, que viene dado por los valores de las propiedades (atributos) del objeto. Además de datos, el estado puede también incluir otros objetos a los que solicitar información o en los que delegar para completar el servicio solicitado por el cliente. Como es lógico, en el momento de ejecutar esos servicios, el estado interno puede cambiar.

- Clases: es la abstracción que describe la estructura y comportamiento de los objetos. Es en realidad la fuente para la creación de nuevos objetos que compartirán el mismo comportamiento aunque no el mismo estado.

  Existen distintas relaciones entre clases: la relación de generalización-especialización (herencia), la relación parte-todo (composición, donde una clase representa el "todo" y un conjunto de clases denotan sus partes), y asociación (relación semántica entre clases).

- Paso de mensajes: en la programación imperativa se habla de llamadas o invocaciones a procedimientos y funciones. En OO el paso de mensajes viene a simbolizar el mismo proceso: el proceso de comunicación formal por el que un objeto solicita a otro un servicio. Debido principalmente a la vinculación dinámica, sin embargo, es lícito establecer una separación (al menos conceptual) entre la solicitud del servicio y la invocación del método concreto. Hablamos entonces de que un objeto solicita un servicio enviando un mensaje y el receptor invoca la ejecución de un método que satisfaga la petición; el método concreto que se terminará llamando depende del tipo del objeto en tiempo de ejecución. Evidentemente, el mensaje permite devolver datos al solicitante.

  Cuando un objeto recibe un mensaje puede *delegar* parte de la realización del servicio en otros objetos distintos.

- Métodos: son los bloques de código ejecutados en respuesta a los mensajes. Son los que nos permiten acceder, fijar y manipular la información de un objeto.

- Herencia: es el mecanismo que permite que los datos y el comportamiento de una clase (la *superclase*) estén incluidos o sean usados como la base de otra clase (*subclase*). Es la base del polimorfismo, lo que hace posible un mayor nivel de abstracción pues posibilita que las subclases puedan ser tratadas de igual forma que las superclases. También favorece la reutilización y la extensibilidad.

No obstante la herencia (especialmente la herencia entre clases, en contraposición de la *implementación* de una interfaz) puede llegar a ser también un inconveniente en grandes proyectos pues impone una relación muy fuerte entre las clases. Una jerarquía de clases demasiado profunda puede terminar siendo inmanejable y provocar unos tiempos muy altos de compilación.

- Polimorfismo: la habilidad que presentan las distintas clases para responder al mismo mensaje, de modo que cada una de ellas implementa el método invocado por el mensaje de la manera apropiada de acuerdo al tipo en ejecución.

  El polimorfismo es en realidad el responsable de ocultar las distintas implementaciones bajo una interfaz común y hace que distintos objetos puedan responder al mismo mensaje produciendo un comportamiento distinto. La herencia sin polimorfismo se quedaría prácticamente en una forma de evitar el *copiar-pegar*. La base del polimorfismo es la vinculación dinámica (o en tiempo de ejecución) ya descrita en el tema correspondiente.

  Desde un punto de vista general, el receptor es el responsable de interpretar el mensaje que le llega, mientras que el emisor lo único que sabe es que ese receptor es capaz de responder al mensaje. En la práctica algunos lenguajes (el más importante C++) implementan el paso de mensajes y la vinculación dinámica utilizando mecanismos de bajo nivel que recaen *en el solicitante* del mensaje[2]; pero esto no debe confundirnos con las ideas de alto nivel de lo que significa polimorfismo, vinculación dinámica y paso de mensajes.

Diseño orientado a objetos

El diseño orientado a objetos consiste en la descomposición de un sistema en clases de objetos que se utilizarán para construir ese sistema. El proceso consiste en identificar qué clases hacen falta y organizarlas. Esa organización de las clases podrá hacer uso de las dos relaciones típicas: la jerarquía y la agregación. Por otro lado, en el momento de realizar esa organización se debe tener en cuenta tanto el acoplamiento como la cohesión.

Entendemos por acoplamiento a la relación existente entre las operaciones y datos de distintas clases. Un alto acomplamiento significa una fuerte dependencia entre clases, por lo que está claro que dificulta la reutilización: para poder utilizar la clase A necesitaré la clase B.

Por otro lado, la cohesión es la relación que existe entre las operaciones y los datos de una clase. Si las operaciones que realiza una clase tienen poca relación unas con otras, la cohesión es baja. Cuanto mayor sea la cohesión más relacionadas estarán y por tanto más fáciles serán de entender y de mantener.

Eso implica que el objetivo principal de un diseño orientado a objetos es *maximizar la cohesión y minimizar el acoplamiento*.

En general, hay que intentar realizar clases "minimalistas", de forma que una clase sea un concepto del mundo real ("si no eres capaz de pensar fácilmente en el nombre de una clase, quizá estés ante un diseño incorrecto"). Las clases más pequeñas son más fáciles de comprender, y por tanto de usar y reutilizar. También son más fáciles de probar. En el lado contrario una clase monolítica (grande y con mucha funcionalidad) diluye la encapsulación

[2]Nos estamos refiriendo a la forma en la que el envío del mensaje es traducido a instrucciones de bajo nivel.

pues muchas operaciones (poco cohesionadas) tienen visible la misma información (datos internos del objeto).

En los siguientes apartados revisaremos la encapsulación, herencia y polimorfismo/-vinculación dinámica desde el punto de vista del diseño orientado a objetos.

Encapsulación[3]

Ya hemos dicho que proporciona la protección de la implementación interna de una clase, de forma que ésta queda oculta tras una interfaz que es visible para el resto del mundo.

Cuando en DOO nos planteamos hacer un buen uso de la encapsulación el aspecto más importante es la definición de la interfaz u operaciones que la clase proporcionará al exterior. Todo lo demás (la forma en la que se implementarán esas operaciones, los datos que se almacenarán en el objeto, etc.) se puede fijar más adelante. Pero si la interfaz está mal pensado/diseñado, una vez en uso no habrá forma de arreglarlo sin romper el código de terceros (los usuarios de tu clase).

Como regla general, los datos de una clase han de ser privados. Pensemos en los siguientes ejemplos sencillos:

```
// Opción a
class Punto {
  public int x;
  public int y;
}
```

```
// Opción b
class Punto {
  private int x;
  private int y;
  public int usarX() {
    return x;
  }
  public int usarY() {
    return y;
  }
}
```

En principio ambas versiones tienen el mismo coste (asumiendo compiladores mínimamente inteligentes). Sin embargo la segunda opción es menos rígida que la primera. Por un lado, todo lo que se puede hacer con la primera opción puede también hacerse con la segunda (asumiendo formas de alterar las coordenadas). Además, si se empieza utilizando la primera opción, ya no podremos cambiarla más adelante, por ejemplo cambiando la forma de almacenar el punto de coordenadas cartesianas a coordenadas polares.

Por otro lado, no se debe abusar de los famosos *getters* y *setters* que habitualmente lo que terminan provocando es que hacen públicos atributos privados. Al hacer este tipo de métodos de forma sistemática para todos los atributos de la clase dejamos de ocultar su implementación, aumentando el acoplamiento con el cliente. Además es especialmente peligroso en lenguajes como Java o C# (en general en aquellos que hagan un uso mayoritario de tipos referencia), pues el cliente se puede quedar con referencias a objetos internos y puede en el mejor de los casos utilizarlo cuando ya está invalidado y en el peor de los casos

[3]En realidad esta palabra que tantas veces utilizamos en informática *no existe* según la R.A.E....

utilizarlo "sin previo aviso" cuando toda nuestra funcionalidad se basa en que su estado no cambie desde fuera (desde un punto de vista formal, se pueden romper los invariantes de la representación).

Estas ideas se resumen en la *ley de Demeter*, también conocida como principio de menor conocimiento o regla "no hables con extraños", que impone restricciones a los objetos a los cuales podemos enviar mensajes dentro de un método, con el objetivo de evitar conocer la estructura de objetos indirectos. Según esta ley:

- Cada clase debe tener un conocimiento limitado sobre otras clases y solo conocer aquellas clases estrechamente relacionadas a la clase actual.
- Cada clase debe hablar sólo a sus amigos y no hablar con extraños.
- Sólo hablar con los amigos más inmediatos.

Esto se concreta en la ley de que en la implementación de cualquier método sólo deberán enviarse mensajes (hacer llamadas a métodos) a:

- El objeto `this` y sus atributos, y elementos de colecciones que sean atributos de `this`.
- Objetos parámetros del método.
- Objetos creados en el interior del método.
- Otros objetos accesibles inmediatamente por `this` (para las clases anidadas).

En particular debe evitarse invocar métodos de un objeto miembro de otro objeto, que ha sido devuelto por una llamada a un método (por ejemplo un *getter*). Esto se puede resumir con el eslogan "usa un sólo punto", que se refiere al operador '`.`' que se utiliza para invocar métodos: por ejemplo la sentencia `a.m1().m2()` rompe esta regla mientras que `a.m()` la respeta.

La ley de Demeter reduce el acoplamiento ya que se evita la visibilidad temporal de objetos indirectos, como los objetos miembro de una clase que presta un servicio a los que la clase cliente no tenga acceso de otra manera. De esta forma la clase que presta el servicio puede cambiar su implementación (y por tanto sus atributos) sin que esto afecte al código de la clase cliente. Aunque no siempre es posible respetar la ley de Demeter, siempre hay que intentar hacerlo, y antes de saltárnosla debemos plantearnos las posibles consecuencias que resultarán para el acoplamiento del sistema, y si no es posible emplear otro diseño alternativo que sí que la respete.

Como resumen:

- La encapsulación da una medida de la flexibilidad de una clase: si la implementación cambia, ¿cuánto código externo a la clase cambia? En general, una clase con N funciones miembro está mejor encapsulada que una con $N + 1$.
- No implementes una funcionalidad *pidiendo* a un segundo objeto la información que necesitas para completar la tarea; pide a ese objeto que tiene la información que haga el trabajo por ti.
- Si finalmente tienes que tener acceso al estado del objeto, utiliza métodos accesores, que aislan posibles cambios de implementación (en el modo de representar internamente esos datos).
- No debes usar métodos accesores y observadores salvo que sean absolutamente necesarios.

Herencia

En OO la herencia define la relación *es-un* entre clases. Permite la especialización de código sin modificar el ya existente y favorece tanto la reutilización (pues una clase se define en términos de otra para no repetir código) como la extensibilidad y flexibilidad pues facilita que una subclase pueda redefinir, ampliar o restringir el comportamiento de la superclase y que pueda ser usada en cualqueir lugar en el que es usada ésta[4].

Ya comentamos en el tema de herencia la diferencia entre subclase y subtipo, que tenía que ver con el principio de sustitución: si B es una subclase de A entonces en cualquier situación se puede sustituir una instancia de A por una de B, obteniéndose el mismo comportamiento observable.

Cuando se utiliza la herencia, sin embargo, el programador puede *romper* el principio de sustitución haciendo que el comportamiento observado sea distinto (por ejemplo reimplementando una operación con un comportamiento incompatible con el de la clase padre). En un DOO debemos respetar siempre el principio de sustitución para que una subclase construida mediante herencia sea, además de subclase, subtipo de la clase.

Durante el DOO se plantea la pregunta de si utilizar herencia o composición: ¿B *es-un* A, o *tiene-un* A?

La respuesta a esta pregunta en algunos casos no es clara, y se admiten las dos respuestas. En ese caso en general se debe intentar utilizar siempre la relación *más débil*. En este sentido, deberá preferirse la composición frente a la herencia.

La razón es que la herencia impone una relación muy fuerte entre las dos clases. Ya lo hemos explicado antes brevemente, pero daremos dos razones más:

- El comportamiento heredado se *fija de manera estática*, de modo que éste no puede ser cambiado en tiempo de ejecución.

- Los cambios en las superclases (ya sea por un cambio de implementación o por la creación o destrucción de operaciones) obliga también a la revisión y modificación de todas sus subclases. Si la jerarquía es grande y profunda, por tanto, un cambio en una clase colocada en una zona superior de la jerarquía puede provocar profundas revisiones de todo el código.

Por todo esto, en principio la composición debe preferirse a la herencia, pues proporciona mayor flexibilidad sin afectar a los clientes y unas clases más robustas y seguras. Ahora bien, esta afirmación no debe llevarse hasta el extremo; la herencia *es útil y necesaria* en muchas ocasiones y tiene ventajas claras cuando viene de la mano del polimorfismo.

Una recomendación habitual cuando se utiliza el mecanismo de herencia es utilizar jerarquías de interfaces (o clases abstractas en la nomenclatura de C++). De esta forma hacemos uso de la herencia sólo para la generalización (y no para la herencia de código y atributos): las superclases son conceptos más abstractos para facilitar la reutilización mientras que las clases derivadas son las que implementan el estado y las operaciones de las superclases.

Con una jerarquía de interfaces, además, separamos la parte más inestable de un desarrollo (las implementaciones) de las más estables (abstracciones), de forma que la mayoría de los cambios están localizados y no terminan afectando a todo el sistema, conduciendo a diseños más flexibles y con una mayor modularidad.

[4] A excepción de que se utilicen mecanismos poco extendidos en los lenguajes OO como la herencia privada.

El uso de estas jerarquías abstractas es aún más recomendable en los casos de herencia múltiple. Es tan importante que lenguajes como Java y C# no permiten herencia múltiple a no ser que sea de interfaces, para evitar la complejidad de gestionar el estado de los objetos cuya codificación está distribuida en varias clases de distinta procedencia.

Polimorfismo y vinculación dinámica

El polimorfismo nos da la flexibilidad necesaria para poder modificar un comportamiento ya creado con la simple creación de nuevas clases sin tener que realizar modificaciones sobre otras ya existentes. Ya hablamos largo y tendido en el tema dedicado al polimorfismo. Debemos simplemente mencionar aquí que conviene hacer uso del polimorfismo en un DOO y delegar en él la extensibilidad del sistema.

No es polimorfismo consultar el tipo de un objeto (`instanceof`) para terminar invocando un método concreto de ese tipo que no teníamos disponible antes con la visión "sesgada" que nos daba tener como tipo una de sus clases antecesoras. En la mayoría de las ocasiones el uso de `instanceof` significa un mal DOO.

Capítulo 10

Patrón Modelo-Vista-Controlador

Codifica siempre como si la persona que finalmente mantendrá tu código fuera un psicópata violento que sabe dónde vives

Martin Golding

Introducción

El MVC (Modelo-Vista-Controlador o *Model-View-Controller*) es un patrón de arquitectura software que separa el manejo de los datos de su presentación al usuario y de la llamada "lógica de negocio".

MVC es utilizado en aplicaciones de distinta índole y con muchísimas variaciones. Además, podemos tener instanciaciones de MVC a distinto nivel de abstracción en distintas partes de la aplicación.

Hoy por hoy el MVC es ampliamente utilizado en aplicaciones Web. Tanto es así que en muchos casos se olvida que en sus orígenes (en 1979) el MVC fue un patrón utilizado para el desarrollo de aplicaciones visuales, que es en el que nos centraremos en este tema.

Conceptos

En MVC distinguimos tres componentes:

- *Modelo*: contiene los datos del dominio de la aplicación e implementa la funcionalidad que gestiona su comportamiento. Puede recibir consultas sobre su estado y solicitudes para cambiarlo. La implementación de esas solicitudes debe garantizar que las restricciones del dominio se siguen cumpliendo.
- *Vista*: consiste en una representación (visual) del modelo, por lo tanto es el componente que gestiona la salida gráfica y/o textual de la aplicación. La vista puede decidir destacar ciertos elementos del modelo e incluso ignorar y no presentar otros.
- *Controlador*: originalmente el controlador era el que interpretaba la entrada del usuario (en su forma más primitiva de pulsaciones de teclas y movimientos de ratón) y lo traducía a acciones sobre el modelo. Hoy por hoy el controlador realiza una tarea similar a la original pero sin lidiar normalmente con aspectos de tan bajo nivel. En ciertas instanciaciones del MVC, el controlador además de solicitar modificaciones al modelo también puede solicitarlas a las vistas.

Como ejemplo, en una aplicación para jugar al ajedrez, el modelo contiene el estado del tablero y permite mover las fichas. Para garantizar la consistencia interna, implementa las reglas del ajedrez de forma que echa para atrás movimientos inválidos; también es el responsable de detectar el final de la partida, etc. Por su parte, la vista es la ventana que dibuja el tablero, o la responsable de escribirlo por la consola, por ejemplo. Por último, el controlador es el que determina qué modificaciones hay que hacer en el modelo cuando el usuario interactúa con la vista, es decir, es el que realiza el control de los eventos que ocurren en la misma.

Interacción entre componentes

El flujo de ejecución en el MVC suele ser el siguiente[1]:

- El usuario interactúa con la vista pulsando botones, seleccionando elementos de una lista, etc.

- La vista informa al controlador sobre las acciones del usuario. En algunas instanciaciones del MVC estas notificaciones llegan debido a que el controlador es *observador* de la vista. En otras ocasiones la vista informa directamente al controlador del tipo de acción deseada por el usuario (realizando así una primera traducción de la acción haciendo, posiblemente, que el controlador sea más general y pueda ser reutilizado en varias vistas).

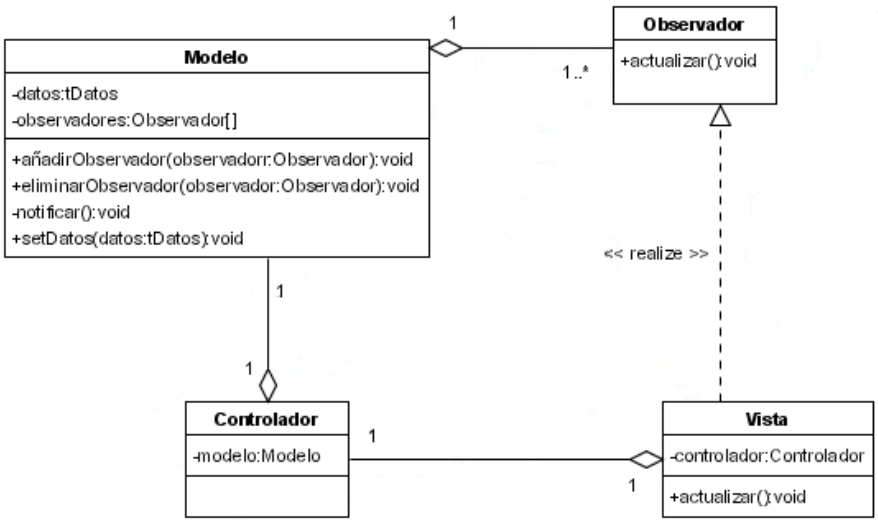
- El controlador realiza una traducción de la acción del usuario sobre la vista hacia el dominio de la aplicación e invoca a la función (o funciones) necesarias en el modelo para que se lleve a cabo la acción solicitada.

- El modelo realiza los cambios de estado internos de acuerdo con la lógica de negocio. Esos cambios son notificados a las vistas para que actualicen la visión mostrada al usuario. Para hacerlo, las vistas son *observadoras* del modelo. De esta forma el modelo es completamente independiente de las vistas y, a pesar de ello, invoca a sus métodos de actualización. Las notificaciones deberán enviar la información suficiente para que las vistas puedan reconstruir el estado interno del modelo para mostrarselo al usuario.

MVC

Con todas las piezas anteriores, el diagrama de MVC es el siguiente:

[1] Cuando la aplicación no es de ventana, el flujo es sensiblemente distinto, debido a que el modelo de interacción hombre-máquina es completamente distinto como ya se explicó en un tema anterior.



En él se ve que:

- El modelo *no* ve a ninguna clase de los otros grupos. Eso garantiza que se puede cambiar de vista y controlador sin alterar el modelo.
- Dada la restricción anterior, para que las vistas puedan enterarse de los cambios producidos en el modelo, se utiliza el patrón *Observer*: las vistas se registran como observadoras del modelo de forma que son notificadas cuando ocurre algún cambio en él (en el diagrama simple anterior, cuando se cambia el estado del modelo con `setDatos`, se llama al método notificar que termina llamando a los observadores).
- El controlador conoce al modelo y de hecho es el que provoca cambios en él invocando a los métodos necesarios. Como ya hemos dicho, en algunas instanciaciones de MVC se sitúa en el controlador los *listener* de la vista (por ejemplo, cuando el que gestiona la pulsación de un botón concreto de la vista) y por tanto es el que traduce ese evento en acciones sobre el modelo. En nuestro caso consideraremos los *listener* como parte de la vista (y podrán seguir siendo clases internas a ésta) de forma que esos *listener* llaman a métodos de más alto nivel del controlador que serán los que terminarán llamando al modelo. De esta forma un cambio de vista no afecta al controlador.

  Por último, en algunas instanciaciones del MVC se permite que los controladores tengan acceso a las vistas de forma que ante ciertos eventos en vez de cambiar el modelo, invocan a métodos de la vista directamente. Este tipo de comportamiento suele tener sentido para notificar errores. Por ejemplo, si el usuario pulsa un botón que permite añadir un nuevo elemento al modelo pero no ha completado todos los campos, el controlador puede notificar el error a la vista, en vez de trasladar esa responsabilidad al modelo.
- Las vistas no deben ver las clases del modelo. Eso implica que las vistas *no* pueden consultar el estado almacenado en él invocando a sus métodos, sino que tiene que reconstruir el estado interno del modelo de forma incremental en base a los cambios que se le han ido notificando.

  Para evitar esta duplicación de esfuerzo en algunas variantes se permite a la vista tener acceso al modelo pero únicamente *para consultarle* el estado y nunca para realizar cambios en él. En lenguajes como C++ eso puede conseguirse haciendo

que la vista tenga una *referencia constante* al modelo; en lenguajes como Java se requiere un mayor esfuerzo de programación garantizar que efectivamente las vistas no invocan a ningún método de cambio en el modelo, como el uso de interfaces con métodos únicamente de consulta.

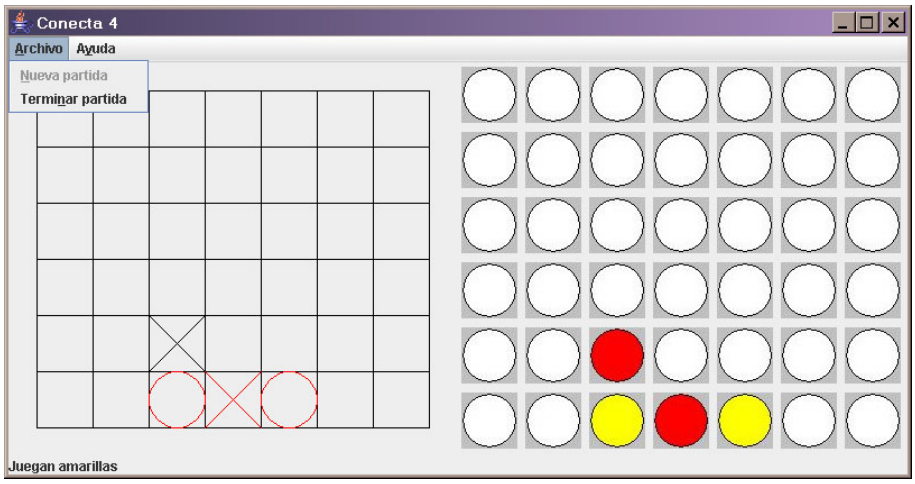
- Es posible tener varias vistas simultáneamente para el mismo modelo. Es más, esas vistas pueden crearse *a posteriori* sin necesidad de modificar el modelo subyacente.

La puesta en marcha de una aplicación que utiliza MVC puede ser algo así:

```
Modelo modelo = new Modelo();
Controlador controlador = new Controlador(modelo);
Vista vista = new Vista(controlador);
modelo.añadirObservador(vista);
```

Ejemplo en Swing

Como ejemplo, veamos cómo podríamos implementar una aplicación con Swing para jugar a las cuatro en raya. El objetivo es presentar al usuario una ventana como la siguiente:



En la ventana aparecen hasta *cuatro* vistas distintas del modelo:

- Dos tableros, que utilizan dos modos distintos de Swing para pintar el tablero.
- Una barra de estado que indica a quién le toca el turno.
- Un menú cuyas opciones se activan y desactivan de acuerdo al estado de la partida.

Veamos uno a uno cada componente.

Modelo

El modelo sería una clase `Partida` con algunos métodos como `iniciarPartida` o `ponFicha` para cambiar el estado y otros como `terminado`, `ganador` y `getTablero` para consultarlo.

La `Partida`, además, permite registrar observadores (`ObserverPartida`) a los que avisa cuando se inicia o termina una partida (ya sea por tablas o porque hay ganador) y cuando se pone una ficha.

```
package logica;

public class Partida {
  // ...
  public void iniciaPartida() {
    // ...
    emitPartidaEmpezada();
  }

  // ...

  private void emitPartidaEmpezada() {
    for (ObservadorPartida o : _observers)
       o.partidaEmpezada();
  }

  private ArrayList<ObservadorPartida> _observers;
}
```

Controlador

Ya sabemos que su tarea es determinar qué modificaciones hay que hacer en el modelo cuando se interacciona con la vista. Es decir, "traduce" las interacciones de la vista en el modelo. Dejaremos los *listener* implementados en las vistas de forma que desde ellos se llame a los métodos públicos que hay en el controlador. Hay dos: uno llamado cuando se selecciona una casilla en el tablero, y otro cuando se solicita empezar una partida nueva.

El controlador mantiene una referencia al modelo para trasladar esas solicitudes del usuario.

```
public class Controlador {

   logica.Partida _modelo;

   public Controlador(logica.Partida p) {
      _modelo = p;
   }

   public void casillaSeleccionada(int col, int fila) {
      if (!_modelo.terminado())
         _modelo.ponFicha(col);
   }

   public void nuevaPartidaSolicitada() {
      if (_modelo.terminado() ||
          _modelo.getTurno() == Ficha.VACIA)
         _modelo.iniciarPartida();
   }
}
```

Vemos que en este caso antes de invocar a esos métodos comprueba que tienen sentido preguntando al modelo su estado interno. Si la acción no puede realizarse el propio controlador podría notificar a las vistas que deben presentar un mensaje de error al usuario. En otras instanciaciones se puede directamente solicitar al modelo la acción y que sea éste el que notifique ese error.

Vistas

Las vistas son observadoras del modelo; cuando llega el evento del cambio de estado cambian su representación visual.

Un ejemplo simple podría ser la etiqueta que luego puede colocarse en la parte inferior de la ventana a modo de "Barra de estado". La etiqueta (`JLabel`) indicará "Partida no empezada", "Juegan rojas", "Juegan amarillas" o "Partida terminada" de acuerdo al estado:

```
public class EstadoPartidaGUI extends JLabel
                              implements ObservadorPartida {

   public EstadoPartidaGUI() {
     super("Partida no empezada");
   }

   public partidaEmpezada() {
     _turno = "amarillas"; ponTurno();
   }

   public partidaTerminada() {
     this.setText("Partida terminada");
   }

   public movimientoRealizado(Ficha c, int col) {
     _turno = _turno.equals("amarillas") ?
                          "rojas" :
                          "amarillas";
     ponTurno();
   }

   protected ponTurno() {
      this.setText(_turno);
   }

   protected String _turno;
}
```

Se puede ver que la vista debe inferir el estado del juego que le interesa en base a las notificaciones recibidas. En concreto, cuando comienza la partida se asume que comienzan las fichas amarillas. También se ve que es la propia vista la que va cambiando el color que tiene el turno en base a quién fue el último que puso ficha.

Este ejemplo nos sirve para ilustrar un aspecto que también explicábamos antes. ¿Qué ocurre si queremos mostrar quién ha ganado cuando acaba la partida? El modelo únicamente nos notifica que la partida ha acabado, pero no *quién es el ganador*.

En el MVC puro las vistas no tienen acceso al modelo por lo que la única solución sería que la vista *reprodujera* el estado interno del tablero y, llegado el momento, buscara quién es el ganador.

Para evitar esa duplicación de código, en ciertas circunstancias se permite que las vistas tengan acceso al propio modelo, siempre y cuando *no lo modifiquen*. En nuestro caso, eso implica permitir llamar a métodos como `getTurno` o `getCasilla` pero no a cosas como `ponerFicha` o `empezarPartida`.

El extracto de código siguiente muestra (parcialmente) la implementación de una de las dos vistas que presentan el tablero y que, por comodidad, mantienen una referencia al

modelo de forma que en el momento del dibujado le preguntan directamente por el estado:

```
public class TableroGUIJPanel extends JPanel
                              implements ObservadorPartida {
   public TableroGUIJPanel(Controlador c, Partida p) {
      _controlador = c;
      _partida = p;
      // Inicializamos los atributos del JPanel que somos.
      initComponent();
   }

   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      // Dibuja los cuadros
      drawGrid(g);
      // Dibuja las fichas (acceso al estado del modelo)
      drawFichas(g, _partida.getTablero());
   }

   // Métodos observadores de la partida
   public void movimientoRealizado(Ficha color, int columna) {
      // Simplemente nos repintamos
      repaint();
   }

   // ...

   private void initComponent() {
      // ...
      // Configuración del listener que avisa al
      // controlador
      addMouseListener(new MouseAdapter() {
         public void mouseReleased(MouseEvent e) {
            if (e.getButton() == MouseEvent.BUTTON1) {
               // Convertimos la posición
               int col = ...;
               int fila = ...;
               _controlador.casillaSeleccionada(col, fila);
            }
         }
      });
   }
}
```

Capítulo 11
Hebras

La física es el sistema operativo del universo.
Steven R Garman

Resumen: Este capítulo explica brevemente los conceptos de hebra y proceso para pasar directamente a explicar la forma en la que se puede conseguir la ejecución de varias hebras en Java. Después pasa a describir las peculiaridades que deben tenerse en cuenta cuando se programa una aplicación utilizando Swing con varias hebras.

Motivación

Vamos a desarrollar una aplicación de ventana muy simple que presente únicamente un botón *Start* y un área de texto. Cuando se pulsa el botón, el área de texto se vacía y se escriben todos los números primos menores que 100, escribiendo uno cada medio segundo aproximadamente. Sin ánimo de complicarnos en la implementación y asumiendo que tenemos una clase `Primos` con un método estático `nextPrime(n)` que te devuelve el primer número primo mayor que `n`, una posible implementación sería algo así:

```
import ...
...;

public class MainWindow extends JFrame {

   private static final long serialVersionUID = -5267471286080527448L;

   public MainWindow() {
      super();
      initialiceGUI();
      setSize(400, 300);
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
   }

   protected void initialiceGUI() {

      JPanel botonera = new JPanel();
      botonera.setLayout(new FlowLayout());
      startButton = new JButton("Start");
```

```
        botonera.add(startButton);

        this.add(botonera, BorderLayout.NORTH);
        startButton.addActionListener(new StartListener());
        primos = new JTextArea();
        this.add(new JScrollPane(primos), BorderLayout.CENTER);
    }

    JTextArea primos;
    JButton startButton;

    public static void main(String args[]) {

        MainWindow app = new MainWindow();
        app.setVisible(true);
    }

    class StartListener implements ActionListener {

        public void actionPerformed(ActionEvent arg0) {
            int n = 2;
            primos.setText("");
            do {
                primos.append("" + n + "\n");
                n = Primos.nextPrime(n);
                esperaMedioSegundo();
            } while (n < 1E7);
        }

        public void esperaMedioSegundo() {
             long ini = System.currentTimeMillis();
             while (ini + 500 > System.currentTimeMillis())
                 ;
        }
    }
}
```

Al ejecutar la aplicación, sin embargo, las cosas no funcionan como uno espera. Cuando se pulsa el botón *Start*, la aplicación deja de responder a los eventos del usuario, hasta que ha completado el cálculo de todos los primos. Por ejemplo, no se puede volver a pulsar el botón (se queda "pulsado") y si redimensionamos la ventana, el tamaño de los bordes aumenta pero no se recolocan sus elementos. Dependiendo de la plataforma, además, los números no irán apareciendo de uno en uno, sino que aparecerán de golpe al final del proceso.

La razón de este comportamiento está en la ejecución del método `actionPerformed`. Cuando Swing avisa al *listener* de la pulsación del evento, el método tarda en terminar. Durante ese tiempo la ejecución de la aplicación está centrada en el cálculo de los números primos, por lo que "no hay nadie" prestando atención a los eventos que vienen del usuario.

Como norma general de diseño de aplicaciones de ventana, los *listener* deben tratar el evento de forma rápida y volver (terminar) lo antes posible para que el usuario no perciba esta pérdida momentánea de control. Si un evento del usuario desencadena un cálculo complejo y potencialmente largo, éste debe hacerse en otro sitio distinto al método

`actionPerformed`. Ese otro sitio es una hebra auxiliar.

En las siguientes secciones detallaremos las ideas generales de las hebras sin volver a ejemplos visuales. Después retomaremos el ejemplo de la ventana que calcula primos para, teniendo ya las nociones necesarias sobre hebras, entender exactamente el funcionamiento interno de swing y cómo debemos trabajar para evitar el comportamiento anterior.

Procesos y hebras

Los procesos y las hebras (o *threads*) son las unidades básicas de ejecución de la programación concurrente. Ambos permiten la ejecución "en paralelo" de dos bloques de código, aunque con mecanismos distintos.

Antes de ver las diferencias entre ambas se debe hacer una matización. La ejecución no tiene por qué ser *simultánea* (de ahí las comillas en el "en paralelo" del párrafo anterior). En el modelo hardware subyacente se podrían ejecutar unas pocas instrucciones de un bloque de código para pasar a continuación a ejecutar unas pocas instrucciones del otro bloque y así sucesivamente. La percepción final para los humanos, que somos dispositivos "lentos", es que ambos bloques se están ejecutando a la vez, pero en realidad se están ejecutando secuencialmente pequeños trozos de cada uno de los dos bloques.

Dicho esto, la diferencia fundamental entre un proceso y una hebra es que los procesos son entidades estancas en cuanto a memoria se refiere. Es decir, cada proceso dispone de su propio espacio de memoria independiente del espacio del resto de procesos lanzados y un proceso no tiene acceso a la memoria de otro. Si un proceso se quiere comunicar con otro tendrá que contar con la ayuda del sistema operativo (el gestor de los procesos) para poder poner en marcha sistemas de comunicación como tuberías, sockets o esquemas de memoria compartida.

Por su parte, las hebras o threads deben verse como caminos de ejecución dentro de un mismo proceso. Es decir, las hebras se sitúan dentro de los procesos y todas tienen acceso a la memoria completa del proceso al que pertenecen. Eso significa que las hebras sí comparten memoria (las hebras del mismo proceso, claro).

Como ejemplo, cuando se lanza la ejecución de un programa implementado en Java:

```
$> java tp.pr5.Main
```

se crea el proceso que ejecutará la aplicación y que carga la máquina virtual de Java. En el momento de la creación se construye la hebra inicial (todos los procesos tienen una hebra inicial que es la que que ejecuta el punto de entrada a la aplicación).

Una vez cargada la JVM, la hebra principal comienza a ejecutar el `main` de la aplicación. Al terminar la ejecución del `main`, la JVM comprueba si quedan hebras por concluir. Si no queda ninguna activa, se termina la aplicación.

Durante la ejecución del `main`, éste puede crear otras hebras distintas para ejecutar "en paralelo" otros bloques de código. Si en el momento de terminar el `main` esas nuevas hebras no han terminado, la aplicación continuará hasta que todas ellas terminen su ejecución.

Aunque desde Java se pueden crear procesos, en realidad hay un mejor soporte para la gestión de hebras. En el siguiente apartado veremos cómo crear nuevas hebras.

Creación de hebras en Java

La clase principal para conseguir concurrencia en Java es la clase `Thread` del paquete `java.lang`. La clase representa una hebra que, al llamar al método `start`, comienza su

ejecución. El código concreto que se ejecuta aparece en el método `run` que es sobreescribible. Por lo tanto para implementar una hebra particular se puede crear una clase que herede de `Thread` y sobreescribir el método `run` con el código concreto. Para lanzar la ejecución en una hebra nueva, se crea un objeto de esa clase y se invoca a su método `start`. *No* se debe llamar a `run()`, pues el método se ejecutaría pero no en una hebra nueva, sino en la hebra que lo invoca.

El siguiente ejemplo implementa una hebra que simplemente escribe una y otra vez por la salida un valor constante configurado en el momento de su creación. La aplicación crea (y lanza) dos hebras con constantes distintas:

```
class MiHebra extends Thread {

    private int num;

    public MiHebra(int constante) {
        this.num = constante;
    }

    public void run() {
        while (true) {
            System.out.print(num);
        }
    }

    public static void main(String []args) {
        new MiHebra(0).start();
        new MiHebra(1).start();
    }
}
```

Como se ve, la aplicación crea y lanza las dos hebras y termina. Eso significa que la hebra principal de la aplicación (la que ejecuta el `main`) termina enseguida. Sin embargo la aplicación continúa su ejecución indefinidamente.

Si la clase hebra no se va a reutilizar, en vez de crear una clase específica, se puede utilizar una clase anónima:

```
class EjemploHebra extends Thread {

    public static void main(String []args) {
        new Thread() {
            public void run() {
                while (true)
                    System.out.print(0);
            }
        }.start();
    }
}
```

Existe otra alternativa para la creación de hebras. La vista anteriormente tiene un problema en algunos momentos: requiere el uso de herencia. Dada la limitación de herencia simple de Java eso puede ser un problema (si queremos que la clase, además, herede de otra clase distinta). Para solucionarlo, se puede utilizar la interfaz `Runnable`. Esta interfaz (que, de hecho, es implementado por la clase `Thread`) tiene un método `run` que hay que implementar. Una vez implementada la interfaz, se construye un objeto de la clase `Thread`

asociando el objeto `Runnable` y se lanza la hebra:

```
class MiHebra2 implements Runnable {

    private int num;

    public MiHebra2(int constante) {
        this.num = constante;
    }

    public void run() {
        while (true) {
            System.out.print(num);
        }
    }

    public static void main(String []args) {
        new Thread(new MiHebra2(0)).start();
        new Thread(new MiHebra2(1)).start();
    }
}
```

Internamente este mecanismo está implementado haciendo que el método `run` de la clase `Thread` invoque al `run()` de un atributo `Runnable` asignado en el constructor:

```
public class Thread implements Runnable {

    // ...

    protected Runnable target;

    public Thread(Runnable target) { this.target = target; }

    public void run() {
        if (target != null)
            target.run();
    }

    // ...

}
```

Aunque la herencia directa de la clase `Thread` es más sencilla, el uso de `Runnable` es más flexible y preferible en aplicaciones de tamaño medio o grande. El API de la librería `java.util.concurrent` de gestión de hebras trabaja directamente con objetos que implementan esta interfaz.

Cuando se programa utilizando hebras hay que tener muy presente que todas ellas comparten la misma memoria. Eso significa que dos hebras pueden estar trabajando con el mismo objeto a la vez. El acceso simultáneo a un mismo recurso puede provocar problemas que son tratados (y resueltos) en disciplinas de la informática como la programación concurrente. Aunque no entraremos en detalle aquí, sí diremos que Java tiene ciertos mecanismos de sincronización que permiten la programación segura ante hebras. De esta forma, por ejemplo, si tenemos un contenedor de elementos al que queremos que varias hebras accedan podremos, utilizando esos mecanismos de sincronización, garantizar que en ningún momento haya dos hebras leyendo o escribiendo a la vez en él. Estos meca-

nismos de sincronización tienen un coste asociado (en tiempo de ejecución) por lo que el programador de una librería debe decidir si quiere que su librería sea segura ante hebras (más versátil pero algo más lenta) o no (menos versátil, pero más rápida).

Planificación de hebras

Cuando una aplicación tiene varias hebras, hay que decidir en qué momento se ejecuta cada una: si tenemos varios procesadores o núcleos ¿se le da uno a cada hebra?. Si se usa un único núcleo para toda la aplicación, ¿cuántos milisegundos se concede a cada hebra y cuándo se la "*expropia*"?

Estas decisiones son responsabilidad del *planificador* de la JVM. La especificación no exige ningún algoritmo o funcionamiento concreto más allá de ciertas condiciones sobre las prioridades que veremos a continuación. Ni siquiera se establece si la JVM debe crear una hebra del sistema operativo por cada hebra o simular las hebras por ella misma.

Una manera de afectar al planificador es utilizando *prioridades*. Efectivamente, la clase `Thread` tiene dos métodos relacionados, uno para cambiar la prioridad, `setPriority`, y otro para acceder a ella, `getPriority`. La prioridad puede ser un valor entre `MIN_PRIORITY` (=1) y `MAX_PRIORITY` (=10), por lo tanto `setPriority` fallará (generando una excepción) si se intenta poner un valor más allá de ese rango. Si no se especifica prioridad, por defecto las hebras se crean con `NORM_PRIORITY` (=5). Además, los mecanismos de permisos de la JVM son comprobados, de forma que no cualquiera puede cambiar la prioridad de una hebra determinada; en caso de invocar al método y no tener permiso, también se genera una excepción.

El planificador está obligado a que si hay varias hebras con la misma prioridad, todas ellas se deben ejecutar en algún momento (ninguna muere por *inanición*). La JVM sin embargo no obliga a que deban ejecutarse hebras de menor prioridad si hay pendientes de ejecución hebras con mayor prioridad.

Cuando se lanza una hebra, ésta está automáticamente disponible para que el planificador pueda darle CPU. Durante su ejecución, la hebra puede pedir al planificador que la suspenda (pause) durante una cantidad de tiempo dada. De esta forma, dejará la CPU libre para otras hebras. Para eso, se utiliza el método `sleep(long msecs)` o `sleep(long msecs, int nanos)`. En ambos casos se recibe la cantidad de tiempo que se desea estar pausado; transcurrido ese tiempo (aproximado, la precisión dependerá del reloj *hardware* del sistema), la hebra pasa a estar de nuevo disponible y el planificador puede ponerla a ejecutar en el momento que desee.

El uso de `sleep` es más eficiente que dejar a la hebra "entretenida" con un bucle que no hace nada, pues se consumirá una CPU que podría haberse estado utilizando para otra cosa.

Existe otro método para pedir al planificador explícitamente que entre en acción por si quiere cambiar la hebra en ejecución; técnicamente lo que hace es pausar temporalmente la hebra pero dejándola preparada para la ejecución. El método se llama `yield` y se utiliza en contadas ocasiones pues si bien es una característica indispensable cuando el planificador no expropia hebras (multitarea colaborativa), cuando tenemos un planificador como el de Java raras veces se necesita. Eso sí, si se usa hay que tener presente que ante la solicitud, el planificador puede optar por seguir ejecutando la hebra, es decir `yield` no garantiza un cambio de contexto.

Java, por otra parte, no da ningún mecanismo para parar (abortar) desde fuera la

ejecución de una hebra ya lanzada[1]. Si queremos parar una hebra, tenemos que *solicitárselo* a la propia hebra y confiar en que ésta nos hará caso. Para realizar esa solicitud se utiliza el método `interrupt`:

```
// Hebra que multiplica 10000 matrices de 1000x1000
class HebraPesada implements Runnable {

    // ...

    public void run() {

        for (int i = 0; i < 10000; ++i) {
            this.multiplicaMatriz(i);
            if (Thread.interrupted()) {
                // Nos han pedido que acabemos...
                break;  // También vale return
            }
        }
    }
}
```

La hebra, por tanto, debe comprobar periódicamente si la han solicitado terminar, utilizando el método `Thread.interrupted` y en ese caso liberar los posibles recursos que tenga y terminar. Eso no siempre es fácil pues podemos estar ejecutando un método distinto de `run` y que está en ejecución tras una larga serie de llamadas a métodos. En esos casos puede ser recomendable, en vez de terminar devolviendo algún tipo de código indicando que la acción se ha detenido por la solicitud de terminación, generar la excepción `InterruptedException` para que el método `run` en última instancia la capture.

Algunos métodos del API de Java también hacen esa labor de comprobación periódica de la solicitud de terminación, y en ese caso generan la excepción anterior. El ejemplo más notable es el del método `sleep`:

```
class HebraMensajes implements Runnable {

    public void run() {
        for (int i = 0; i < importantInfo.length; ++i) {
            // Paramos 4 segundos
            try {
                Thread.sleep(4000);
            } catch (InterruptedException ex) {
                // Nos han pedido terminar...
                liberaRecursos();
                return;
            }

            // Escribimos el mensaje
            System.out.println(importantInfo[i]);
        }
    }

    // ...
}
```

[1] Bueno, en realidad sí la hay, pero está desaconsejada, mediante métodos `deprecated`.

Swing y hebras

Tanto swing como AWT crean una hebra nueva utilizada para procesar los eventos de entrada (teclado y ratón)[2]. Cuando se detectan eventos, la propia hebra de swing invoca los métodos de usuario de procesado de eventos.

Para que la interacción con el usuario sea fluida, la hebra de swing tiene más prioridad (6) que las hebras normales (5). De esta forma si hay algún evento de entrada, responde inmediatamente pues el planificador pondrá a ejecutar la hebra en detrimento de las hebras normales (a no ser que la aplicación tenga alguna hebra de más prioridad que la de swing, claro, pero esto no es lo normal). Cuando la hebra termina de procesar esos eventos, se suspende a la espera de que lleguen más, dejando de consumir recursos (CPU).

Lo anterior implica que el código incluido en los observadores de eventos (métodos `actionPerformed`, etc.) son ejecutados *en la hebra de swing*. Eso automáticamente nos lleva a que si ese procesado del evento consume mucho tiempo (ver apartado de motivación al principio del tema), los componentes de swing dejarán de responder automáticamente, pues la hebra responsable de sus respuestas está ocupada manejando el último evento que llegó.

Es por esto que si el procesado de un evento va a durar demasiado tiempo, el gestor de ese evento debería recopilar toda la información necesaria de los componentes swing adecuados y crear una hebra distinta que realice la tarea. Esa hebra será suspendida por el planificador en el momento en el que haya otro evento de swing.

Cuando se hace lo anterior, no obstante, hay que tener presente que swing *no es seguro ante hebras*. En efecto, la mayoría de los métodos del API de swing no se protegen ante la invocación concurrente a los mismos, para evitar pérdida de rendimiento.

Eso significa que una vez lanzada la primera ventana de swing (lo que provoca el lanzamiento de la hebra de swing), la invocación a cualquier método de swing (excepto la adición y borrado de *listener*) debe hacerse *en la propia hebra de swing*. En otro caso, podríamos tener a la hebra de swing leyendo las propiedades de un `JTextArea` para pintarlo y otra hebra cambiando ese texto para añadir una línea más.

Si una hebra distinta a la de swing quiere por tanto acceder a un componente de swing no puede hacerlo directamente, sino que debe solicitar a la hebra de swing que ejecute una "tarea". Para eso, se puede utilizar uno de los siguientes métodos estáticos de la clase `SwingUtilities`:

- `invokeLater(Runnable)`: la hebra de swing ejecuta la tarea pasada como parámetro (llama a su método `run`) cuando esté libre, es decir, cuando termine de procesar los eventos que tiene pendientes.

- `invokeAndWait(Runnable)`: igual que la anterior, pero la invocación del método no termina hasta que la hebra de swing no ha acabado la tarea.

Desde Java 1.6 existe, además, una clase que ayuda en la implementación de aplicaciones visuales con métodos de gestión de eventos de larga duración, la `SwingWorker` del paquete `javax.swing`. Nosotros no la usaremos, pero puedes consultar la documentación para más detalles.

[2]Por eso puede ser problemático leer de teclado desde la hebra principal con swing lanzado: dos hebras leen simultáneamente del mismo dispositivo.

Para terminar...

A continuación aparece la implementación de la aplicación con la que empezábamos el tema. En ella:

- Se ha añadido un nuevo botón, `Stop`.
- En un instante determinado únicamente uno de los dos botones está activo: el botón `Start` cuando no se ha empezado la generación de primos, y el botón `Stop` cuando se están generando.
- La generación y escritura de los primos se realiza en una hebra aparte. Antes de lanzarla se activa el `Stop` y la hebra, antes de terminar, lo desactiva, activando de nuevo el `Start`.
- La ventana guarda una referencia a la hebra que está haciendo el trabajo. Cuando el usuario pulsa `Stop` se le solicita que termine.
- La hebra comprueba periódicamente si se le ha pedido terminar, y en ese caso acaba antes de generar todos los primos.
- Siempre que necesita acceder a swing lo hace solicitando la ejecución de una tarea a la hebra de swing.
- Para escribir el texto utiliza una clase interna, mientras que para cambiar el estado de los botones al final de la ejecución utiliza una clase anónima.

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;

public class Version2 extends JFrame {

   private static final long serialVersionUID = 447250806821747625L;

   public Version2() {
      super();
      initialiceGUI();
      setSize(400, 300);
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
   }

   protected void initialiceGUI() {

      JPanel botonera = new JPanel();
      botonera.setLayout(new FlowLayout());
      startButton = new JButton("Start");
      stopButton = new JButton("Stop");

      botonera.add(startButton);
      botonera.add(stopButton);
      activaStart();
```

```
    this.add(botonera, BorderLayout.NORTH);
    startButton.addActionListener(new StartListener());
    stopButton.addActionListener(new StopListener());
    primos = new JTextArea();
    this.add(new JScrollPane(primos), BorderLayout.CENTER);
}

 protected void activaStart() {
     // Debería llamarse sólo desde la hebra de swing,
     // y desde el initialiceGUI().
     startButton.setEnabled(true);
     stopButton.setEnabled(false);
 }

 protected void activaStop() {
     // Debería llamarse sólo desde la hebra de swing.
     // Se podría averiguar si eso es así usando
     // SwingUtilities.isEventDispacthThread()
     startButton.setEnabled(false);
     stopButton.setEnabled(true);
 }

JTextArea primos;
JButton startButton;
JButton stopButton;
Thread worker;

public static void main(String args[]) {

    Version2 app = new Version2();
    app.setVisible(true);
}


class StartListener implements ActionListener {

    public void actionPerformed(ActionEvent arg0) {
       primos.setText("");
       activaStop();

       worker = new Thread(new EscribePrimos(10000000));
       worker.start();
    }
}

class StopListener implements ActionListener {

    public void actionPerformed(ActionEvent arg0) {
       worker.interrupt();
    }
}

 class EscribePrimos implements Runnable {

      private int max;
```

```
        EscribePrimos(int max) {
            this.max = max;
        }

        public void run() {
            int n = 2;
         do {
             if (Thread.interrupted())
                 break;
             // Alternativa 1: con clase interna
             SwingUtilities.invokeLater(new EscribeNumYTermina(n, primos));
             try {
                 Thread.sleep(500);
             } catch (InterruptedException ex) {
                 break;
             }
             n = Primos.nextPrime(n);
         } while (n < max);

         // Otra alternativa: con clase anónima
         SwingUtilities.invokeLater(new Runnable() {
                 public void run() {
                     activaStart();
                 }
             });

        }
    }

    private class EscribeNumYTermina implements Runnable {
        int n;
        JTextArea texto;
        public EscribeNumYTermina(int n, JTextArea where) {
            this.n = n;
            this.texto = where;
        }

        public void run() {
             texto.append("" + n + "\n");
        }
    }
}
```