Manual de Iniciación a

Juan Ángel Lorenzo del Castillo
jal@gui.uva.es
versión 2.0

Grupo Universitario de Informática (G.U.I.)
Universidad De Valladolid

Manual de Iniciación a Java

Juan Ángel Lorenzo del Castillo
(jal@gui.uva.es)

Grupo Universitario de Infomática (G.U.I.)
Universidad de Valladolid

21 de marzo de 2004

Prólogo

Este libro que tienes en tus manos pretende ser una colección ordenada de apuntes para facilitar el seguimiento del curso *Iniciación a Java* del *Grupo Universitario de Informática* (G.U.I.) de la Universidad de Valladolid. En ningún caso se trata de un manual exhaustivo sobre el lenguaje Java; ya existen libros muy completos sobre el tema, algunos de los cuales puedes encontrar referenciados en la bibliografía.

Este manual está pensado para impartir un curso de 20 horas, de forma que el alumno adquiera las bases y la soltura para poder continuar aprendiendo por su cuenta. Se trata, principalmente, de un **manual de programación**, aunque también aparecerán referencias al funcionamiento interno de la *Máquina Virtual de Java*. Espero que el número de páginas vaya aumentando paulatinamente, cubriendo otros aspectos que, si bien es poco probable que dé tiempo a explicar en clase, sirvan al alumno para profundizar en la programación en Java.

Ésta es la segunda versión del manual. Estoy convencido de que seguirá teniendo un montón de fallos, bastantes párrafos mal explicados, algún trozo de código que no funcione, y puede que una o dos faltas de ortografía. Con todo, ya se han subsanado unos cuantos errores que contenía la primera versión y se han reestructurado los contenidos, gracias a las sugerencias de los alumnos de cursos anteriores. Agradeceré que cualquier error que encuentres me lo notifiques, ya sea en persona o por correo electrónico (`jal@gui.uva.es`) para poder corregirlo.

Por cierto, este manual es **gratuito**. En el G.U.I. sólo te cobraremos los gastos de fotocopiarlo. También está disponible para su descarga en www.gui.uva.es/~jal/java/ (total, nadie va a pagar por él). Puedes fotocopiarlo, reproducirlo totalmente o en parte sin permiso del autor, e incluso utilizarlo para calzar una mesa coja. Lo que no puedes hacer es ganar dinero con él (si yo no lo hago, tú tampoco). Tampoco puedes modificarlo. Tanto los aciertos como los errores son sólo míos.

Por si te interesa, este libro ha sido escrito en formato texto[1] con el editor **xemacs** y formateado posteriormente en LaTeX con los editores **Ktexmaker2** y **Kate**. Todo el código fuente se programó con el editor **vi**, y las figuras han sido realizadas con el programa **Dia**. Y todo ello sobre un S.u.S.E. Linux 8.0. que *calza* un AMD Athlon a 1.2 Ghz. ¿Quién necesita Windows? ;-)

Nada más. Espero que este manual te sea útil y que descubras lo interesante y práctico que es ese lenguaje el cual, hace más de diez años, se diseñó para incluirlo en todo tipo de aparatos (desde teléfonos móviles hasta tostadoras) y que, gracias al crecimiento de Internet, se ha convertido en un estándar *de facto* para programar todo tipo de aplicaciones para la Red.

El Autor.

[1] O sea, a pelo, en un archivo de extensión `.txt`

Índice general

Capítulo 1

Introducción a Java

1.1. Introducción

En este primer capítulo comenzaremos hablando del origen e historia del lenguaje Java. A continuación, se explicarán cuáles son sus características, mostrando así en qué consiste y qué le hace diferente de otros lenguajes de programación. Seguidamente, se mostrará el modo de instalar y trabajar con la plataforma Java. Por último, se darán unos consejos para compilar y ejecutar programas.

1.2. Orígenes e historia del lenguaje Java

El lenguaje de programación Java fue desarrollado por Sun Microsystems en 1991, como parte de un proyecto de investigación para desarrollar software que pudiera ejecutarse en todo tipo de dispositivos electrónicos domésticos, como televisiones o frigoríficos. Dado que estos aparatos no tenían mucha potencia ni memoria (al menos, por entonces), se buscaba que fuera un lenguaje rápido, eficiente y que generara un código muy pequeño. Además, si se quería instalar en dispositivos tan heterogéneos y de distintos fabricantes, el lenguaje debía ser independiente de la plataforma en la que se ejecutase.

Para diseñarlo, los ingenieros de Sun se basaron en C++, y lo llamaron Java. Sus primeros prototipos de aparatos con chips que ejecutaban código Java, como un sistema para gestionar el vídeo bajo demanda, no tuvieron éxito. Nadie tenía interés en comprar el invento de Sun, por lo que el proyecto se disolvió en 1994.

Por entonces, la *World Wide Web* (WWW) estaba en auge. Los desarrolladores de Sun decidieron programar un navegador web escrito totalmente en Java, que mostrara las ventajas del lenguaje (independencia de la arquitectura, fiable, seguro, en tiempo real, etc.). Lo llamaron *HotJava*, e incluyeron una de las características más conocidas de Java: los ***applets***, o código capaz de ejecutarse dentro de una página web.

Todo esto, unido al hecho de que Netscape diera soporte en su navegador para ejecutar *applets*, proporcionaron a Java el empujón definitivo para ser ampliamente aceptado entre la comunidad de desarrolladores de software. El lenguaje pensado inicialmente para ejecutarse en electrodomésticos encontró su hueco en aplicaciones para Internet.

1.3. Características de Java

Los creadores de Java diseñaron el lenguaje con las siguientes ideas en mente:

- **Simplicidad:** Java está basado en C++, por lo que, si se ha programado en C o en C++, el aprendizaje de la sintaxis de Java es casi inmediato. Sin embargo, se modificaron o eliminaron ciertas características que en C++ son fuente de problemas, como la aritmética de punteros,

las estructuras, etc. Además, no debemos preocuparnos de la gestión de la memoria. Java se ocupa de descargar los objetos que no utilicemos. Es prácticamente imposible, por ejemplo, escribir en una posición de memoria de otro programa.

- **Orientación a Objetos (OO):** Cualquier lenguaje moderno está orientado a objetos, ya que su utilidad y ventajas con respecto a la programación tradicional orientada a procedimientos ha sido ampliamente demostrada en los últimos 30 años. El concepto de OO se explicará en el siguiente capítulo.

- **Distribuido:** Java posee una extensa colección de herramientas que proporcionan la capacidad de trabajar en red de forma simple y robusta.

- **Robusto:** Como ya se comentó antes, Java permite escribir programas fiables con mucho menor esfuerzo que en otros lenguajes. El compilador detecta problemas, como la sobreescritura de posiciones de memoria, que en otros lenguajes aparecerían en tiempo de ejecución. Además, la eliminación del uso de punteros en elementos como cadenas de caracteres o arrays evita muchos problemas que son comunes (y difíciles de depurar) en C o C++.

- **Seguro:** Java está pensado para ser utilizado en red, por lo que se ha cuidado mucho la seguridad. En principio, se supone que es capaz de evitar que se rebase la pila del sistema en tiempo de ejecución, que se corrompa la memoria externa a un proceso, o que se pueda acceder a ficheros locales de un ordenador que está ejecutando un *applet* en su navegador, por ejemplo.

- **Independencia de la plataforma:** He aquí una de las características más importantes y conocidas de Java. Un programa en Java sigue la **filosofía WORE** (*Write Once, Run Everywhere*), es decir, que una vez escrito, puede ejecutarse en cualquier plataforma hardware con cualquier sistema operativo sin recompilar el código.

  ¿Qué quiere decir esto?. En la compilación tradicional de programas escribimos nuestro código fuente pensando en el sistema operativo en el que va a ejecutarse, ya que cada S.O. tiene sus propias peculiaridades, librerías a las que es necesario invocar, etc. No puede escribirse con el mismo código un programa para Linux y para Windows, aunque corran en la misma máquina. Existe, por tanto, **dependencia a nivel de código fuente**.

  Una vez tenemos escrito nuestro programa, lo compilamos. Fijémonos en la figura 1.1. El compilador traduce el código fuente a código máquina capaz de ser entendido por el procesador de la máquina en la que va a correr ese programa. Es decir, que tenemos **dependencia a nivel de archivo binario**, ya que cada compilador es específico de cada arquitectura. Un programa compilado para una máquina *Intel* no funcionará en un *PowerPC*, ni en una *Sparc*.

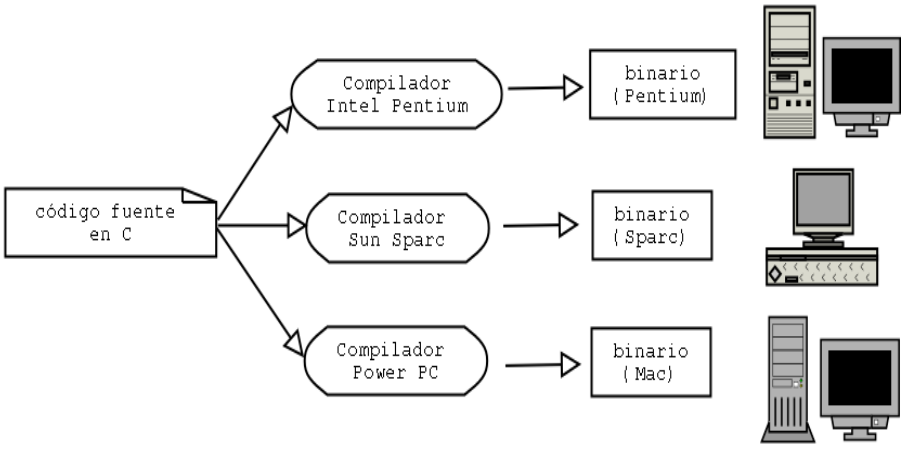

Figura 1.1: Compilación Tradicional de programas.

Java elimina estas dependencias. Una vez que escribamos y compilemos nuestro código fuente, podremos llevar el archivo binario a cualquier ordenador que tenga instalado una **Máquina Virtual de Java** (JVM, *Java Virtual Machine*) y se ejecutará exactamente igual, independientemente de la arquitectura software y hardware de ese ordenador.

¿Y qué es la JVM?. Observemos la figura 1.2, en la que se muestra la compilación de un programa Java. Comenzamos escribiendo nuestro código fuente Java. No nos tenemos que preocupar de las peculiaridades del S.O. ni del ordenador en el que se vaya a ejecutar ese código. Una vez escrito, lo compilamos con el compilador de Java, que nos genera un archivo de ***bytecodes***. Los *bytecode* son una especie de *código intermedio*, un conjunto de instrucciones en un lenguaje máquina independiente de la plataforma.

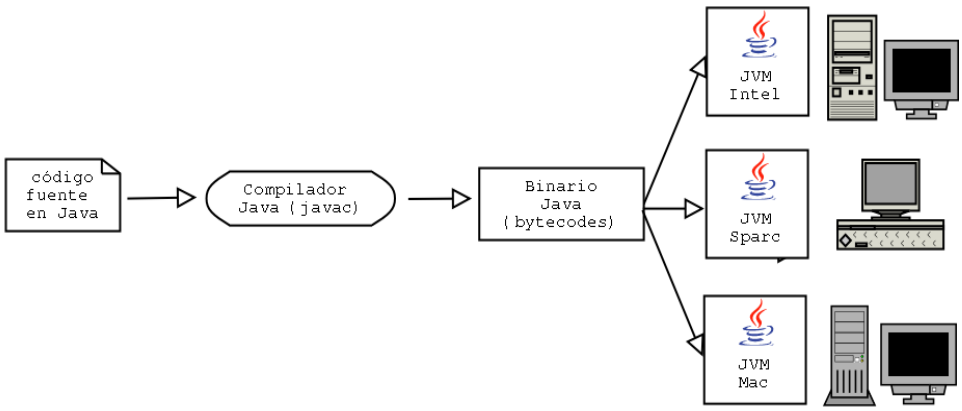

Figura 1.2: Compilación de programas en Java.

Cuando queramos ejecutar nuestro programa, la JVM instalada en el ordenador leerá el archivo de *bytecodes* y lo interpretará **en tiempo de ejecución**, traduciéndolo al código máquina nativo de la máquina en la que se está ejecutando en ese momento. Por tanto, la JVM es un intérprete de *bytecodes*.

En ocasiones, este sistema puede resultar ineficiente. Aunque, paulatinamente, está aumentando la velocidad del intérprete, la ejecución de programas en Java siempre será más lenta que la de programas compilados en código nativo de la plataforma. Es decir, que si queremos programar una aplicación con requisitos de ejecución en tiempo real, será mejor escribir un programa en C++ en vez de usar Java. Existen, sin embargo, compiladores JIT (*Just In Time*), que lo que hacen es interpretar los *bytecodes* la primera vez que se ejecuta el programa, guardando el código nativo resultante, y usando éste en las demás invocaciones del programa. De este modo, se puede llegar a incrementar entre 10 y 20 veces la velocidad respecto al intérprete estándar de Java.

- **Multithreaded:** Java, al igual que C o C++, permite trabajar con varios hilos de ejecución simultáneos, facilitando la programación en sistemas multiprocesador, y mejorando el funcionamiento en tiempo real.

1.4. El *Software Development Kit* de Java

El **JDK** (*Java Development Kit*), también llamado **SDK** (*Software Development Kit*, Kit de Desarrollo de Software) de Java[1] está compuesto por el conjunto de herramientas necesarias para compilar y ejecutar código escrito en Java. Comprende, principalmente, el compilador (*javac*), la JVM, y el conjunto de paquetes de clases[2] que forman una base sobre la que programar.

[1] Me refiero al SDK "oficial", el proporcionado por Sun Microsystems. Existen otras versiones "no oficiales".
[2] Las clases se verán en el siguiente capítulo.

Existen tres *ediciones* del SDK:

- **J2SE (*Java 2 Standard Edition*):** Versión estándar de Java, con la que trabajaremos. Lo de Java 2 es cosa del departamento de marketing de Sun: a partir de la versión 1.2 del SDK, Java pasó a llamarse Java 2, para denotar una importante evolución en la plataforma. La versión estable actual, en el momento de escribir este manual, es la 1.4.2, aunque ya está disponible una versión *beta* del J2SE 1.5.0.

- **J2ME (*Java 2 Mobile Edition*):** Versión de Java orientada a dispositivos móviles y pequeños, como PDAs o teléfonos móviles.

- **J2EE (*Java 2 Enterprise Edition*):** Versión orientada al entorno empresarial. Se utiliza, principalmente, en aplicaciones de servidor, como *servlets*, EJBs (*Enterprise Java Beans*) y JSPs (*Java Server Pages*).

1.5. Instalación y configuración del J2SE 1.4.2

La página oficial de Sun sobre Java es `http://java.sun.com`, y la relativa al SDK es `http://java.sun.com/j2se/1.4.2/index.jsp`. Concretamente, podemos descargarnos el J2SE de `http://java.sun.com/j2se/1.4.2/download.html`. Encontraremos versiones para varios sistemas operativos y plataformas. Nótese que, en algunos casos, existen dos modalidades del J2SE para la misma plataforma: con o sin ***NetBeans***. *NetBeans*[3] es un IDE, o Entorno de Desarrollo Integrado, que nos permite programar aplicaciones más cómodamente, incorporando un editor con resaltado de texto, herramientas para facilitar la programación gráfica y de depurado, etc. Sin embargo, no es necesario descargarlo para trabajar con el SDK. Por ello se nos da la opción de descargar el J2SE sin el IDE.

La instalación no se cubrirá aquí, pero consistirá en algo tan sencillo como hacer doble click sobre un icono (en Windows) o ejecutar un *script* desde la línea de comandos (en UNIX/Linux). Una vez terminada la instalación, sería conveniente poder ejecutar tanto el compilador como la JVM desde cualquier directorio, no sólo el de instalación del J2SE. Si, abriendo una ventana de línea de comandos (tanto en Windows como en UNIX) escribimos `java`, y nos da un error, indicando que no se reconoce el comando, será necesario configurar correctamente el `PATH` del sistema. Por ejemplo, para los siguientes casos:

-Windows 98/ME: En el `Autoexec.bat`, añadir la línea `SET PATH=c:\j2sdk1.4.2\bin;%PATH%` (suponiendo que ese es el directorio en el que está instalado el SDK, claro).

-Windows NT/2000/XP: En *Panel de Control/Sistema/Avanzado*, pulsar sobre el botón *Variables de Entorno* y, en la lista *Variables del Sistema* localizar la entrada `PATH` y añadir la ruta del SDK.

-Linux: Añadir en el archivo `$HOME/.bashrc` o `$HOME/.bash_ profile` la línea `export PATH=/usr/local/j2sdk1.4.2/bin:$PATH`

Ojo, que si instalamos el SDK con *NetBeans*, el directorio por defecto será `c:\j2sdk_nb\j2sdk1.4.2\bin`

1.6. El API del SDK

En cuanto tengamos la JVM funcionando, y queramos comenzar a programar código, nuestra primera pregunta será: ¿y cómo sé qué clases y funciones proporciona Java?. Es típico que queramos realizar tareas como, por ejemplo, ordenar un array de datos, y no sabemos si Java implementa esa función o tenemos que programarla nosotros mismos. Para ello, Sun proporciona la documentación del API (Interfaz de Programación de Aplicaciones) para consultarla online en

[3] Antes conocido como *Forte for Java*.

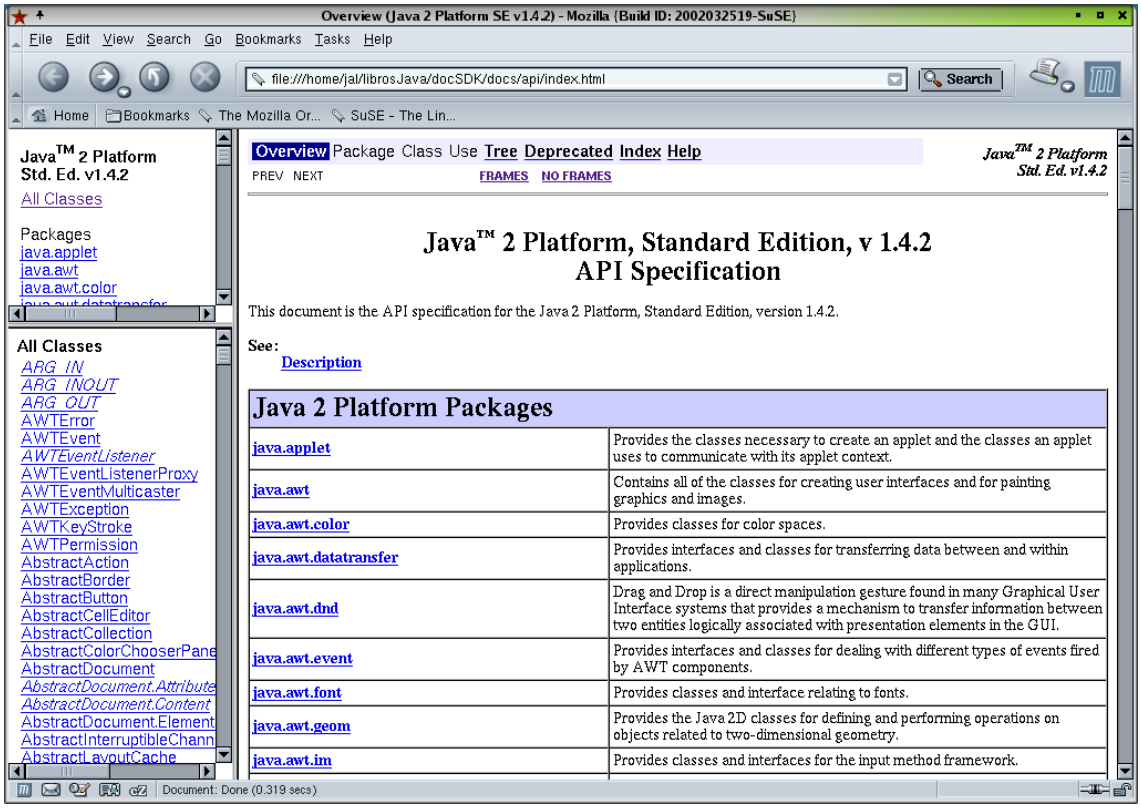

Figura 1.3: La documentación del API.

`http://java.sun.com/j2se/1.4.2/docs/api/`. Si queremos descargarla para disponer de ella en nuestro ordenador, podemos bajarla de la dirección donde se encuentra el SDK. Como se ve en la figura 1.3, el API[4] está en formato html, y muestra todas las clases, métodos y atributos que proporciona Java.

1.7. Cómo compilar archivos

Los archivos con código fuente en Java tienen siempre la extensión `java`. Compilarlos, suponiendo que nos encontramos en el mismo directorio que el fichero fuente, es tan sencillo[5] como:

```
javac archivo.java
```

que nos creará uno o varios archivos con extensión `class`. Esos archivos serán nuestro programa compilado, que podrá entender y ejecutar la JVM.

El compilador posee muchas opciones. Bajo Unix/Linux pueden consultarse con `man javac`. Algunas de las más utilizadas son:

```
javac -d directorio archivo.java
```

que nos permite compilar el fichero fuente y depositar la clase compilada en el directorio especificado. Esto es útil si la clase pertenece a un paquete[6] y queremos, por tanto, depositar la clase

[4] Nos referiremos al API aunque realmente estemos hablando de la documentación del API.

[5] Es un decir, porque `javac` será el encargado de informarnos de todos los errores que tengamos en nuestro código. La lista puede ser larguísima ;-)

[6] Los paquetes se explican en el capítulo 2.

en una estructura de directorios y subdirectorios acorde a la del paquete. Si el directorio no existe, no lo creará, sino que nos dará un error. El modo de que nos cree previamente el(los) directorio(s) es con la sentencia:

```
javac -d . archivo.java
```

que leerá de `archivo.java` la estructura de directorios que componen el paquete, creará esos directorios y depositará la clase compilada allí.

Otra opción interesante es:

```
javac -classpath classpath archivo.java
```

que permite redefinir el CLASSPATH, ignorando el definido por defecto para la máquina o usuario. ¿Y qué es el CLASSPATH?. Mira la sección siguiente.

1.8. El *Classpath*

Cuando compilamos una clase[7] Java, ésta necesitará importar otras clases, ya sean del propio j2sdk, o escritas por terceros. Para que pueda encontrarlas, es necesario definir una variable en el sistema que contenga las rutas en las que el compilador debe buscar las clases.

Al instalar el j2sdk se definirá un classpath por defecto. Sin embargo, podría suceder que necesitáramos redefinir esas rutas para buscar clases en otra parte de nuestra máquina. Usaremos la opción `-classpath`, como se especificó en el apartado anterior. Por ejemplo:

```
javac -classpath /ejemplos:.:/lib/milibreria.jar archivo.java
```

Nótese que las diferentes rutas se separan mediante dos puntos (“`:`”). Para especificar el directorio en el que nos encontremos al invocar a `javac`, se utiliza el punto (“`.`”).

1.9. Ejecución de Programas en Java

Si tenemos una clase llamada MiClase.class, la ejecutaremos escribiendo:

```
java MiClase
```

Fallos típicos al ejecutar un programa:

- Estamos intentando ejecutar una clase que no tiene definido un método `main()`.
- Hemos escrito `java MiClase.class`. El tipo (`class`) no se incluye.
- Al intentar ejecutar una clase llamada `MiClase.class`, `java` arroja un error de tipo `java.lang.NoClassDefFoundError`, a pesar de que estamos seguros de haber especificado correctamente el directorio en el que está la clase. Probablemente se deba a que esa clase pertenece a un paquete. Comprobemos el código fuente. Si incluye al principio una línea del estilo a:

  ```
  package otrasclases.misclases;
  ```

  entonces la clase debe encontrarse en una estructura de directorios con esos nombres. Podemos, por ejemplo, crear un subdirectorio del directorio actual al que llamaremos `otrasclases`. A continuación, dentro de ese, crearemos otro de nombre `misclases`. Y dentro de él copiaremos la clase que estábamos tratando de ejecutar. Si ahora ejecutamos, desde nuestro directorio actual, `java otrasclases.misclases.MiClase`, debería funcionar.

[7] Las clases se explican en el capítulo 2.

Es decir, que si una clase pertenece a un paquete, nos referiremos a ella por su ruta y nombre, separados por puntos. No hace falta que esa estructura de subdirectorios esté en nuestro directorio local. Basta con que se encuentre en alguno de los directorios especificados en el classpath. Lo que sí es obligatorio es respetar la estructura de subdirectorios en la que se encuentra la clase.

1.10. Resumen

En este primer capítulo hemos visto varios conceptos introductorios a Java, de entre los cuales es importante recordar:

- Java se diseñó para ser utilizado en electrodomésticos, pero, actualmente, su principal nicho de mercado se encuentra en aplicaciones del lado del servidor.
- Las principales características de Java son: simplicidad, orientación a objetos, distribución, robustez, seguridad, independencia de la plataforma, y capacidad multihilo.
- Un programa en Java es **independiente de la plataforma**, es decir, que se puede ejecutar en cualquier arquitectura y S.O. que tengan instalados una **Máquina Virtual de Java (JVM)** sin modificar ni recompilar el código. Cuando se compila código fuente en Java, se generan uno o varios ficheros de *bytecodes* que son interpretados, en tiempo de ejecución, por la JVM.
- El **Software Development Kit (SDK)** lo componen las herramientas necesarias para compilar y ejecutar código en Java. Exiten tres versiones: La estándar (J2SE), la orientada a aplicaciones empresariales y de servidor (J2EE) y una tercera para dispositivos móviles (J2ME).
- El SDK nos proporciona un conjunto de paquetes con clases para poder programar en Java. Se denomina API (Interfaz de Programación de Aplicaciones).
- El compilador Java se llama `javac`. El programa que los ejecuta (la JVM) es `java`. El *classpath* es una variable del sistema, que podemos redefinir si es necesario, y que contiene la ruta donde se encuentran las clases necesarias para la ejecución de un programa Java.

Capítulo 2

Programación Orientada a Objetos

2.1. Introducción

En este capítulo se explicará, en primer lugar, qué son las clases y los objetos, paso fundamental para poder programar en Java. Veremos, a continuación, que una clase puede contener atributos y métodos en su interior, por lo que también se introducirán aquí (como el tema de los métodos es muy extenso, se le dedicará más adelante un capítulo). Estudiaremos cómo se relacionan las clases mediante la herencia y, por último, explicaremos cómo se crean y utilizan los paquetes en Java.

2.2. Orientación a objetos

Si sabes algo sobre programación (y si no, te lo cuento yo), habrás oído hablar de lenguajes orientados y no orientados a objetos. El lenguaje C no está orientado a objetos. C++ sí lo está. Java también. Pensar en objetos supone tener que cambiar el chip para enfocar la resolución de problemas de una manera distinta a como se ha hecho tradicionalmente.

Vale, muy bonito. Pero, ¿qué es un objeto?. La mejor forma de entenderlo es mediante una analogía. Consideremos un ordenador, por ejemplo. Si lo abrimos y lo observamos detenidamente, podemos comprobar que está formado por la placa base, el procesador, la memoria, el disco duro, etc. Si, a su vez, examinamos por separado cada parte, veremos que el disco duro está compuesto por varios discos superpuestos, las cabezas lectoras, un circuito controlador, etc. Podemos ver también que cada módulo de memoria está construido a partir de circuitos integrados de memoria más pequeños interconectados entre sí, y lo mismo ocurre con todas las demás partes del ordenador: el todo está formado por piezas, y cada pieza está compuesta por partes más pequeñas.

Supongamos que se nos estropea el disco duro y necesitamos comprar otro. Si cada fabricante de PCs diseñara discos duros para sus ordenadores basándose en especificaciones propias, éstos serían incompatibles entre sí, y nos veríamos obligados a buscar el modelo de disco adecuado para nuestro ordenador.

Por suerte, existen en la industria estándares gracias a los cuales cada empresa puede fabricar internamente los discos duros como mejor les parezca, siempre y cuando la interfaz de conexión con el ordenador cumpla con un estándar determinado y aceptado por todos los fabricantes (IDE, SCSI, etc.). De este modo, tenemos un **objeto** (el disco duro) que realiza una **función** determinada (almacenar información) sobre unos **atributos** (los datos), y que se comunica con el resto del sistema mediante una **interfaz** determinada y bien conocida.

¿Nunca has abierto un ordenador y no sabes lo que hay dentro?. Bueno, usaré un ejemplo más sencillo. Cualquier juego de construcción como los de Lego o Tente está formado por elementos básicos (las piezas). Cada pieza, por sí sola, no tiene mucha utilidad, pero podemos juntarlas para construir lo que nos dé la gana. Si podemos construir cosas es porque cada pieza trae una serie de hendiduras que encajan en las de las demás. Así que tenemos una serie de objetos (las piezas) con una interfaz común (las hendiduras) y que nos permiten realizar una construcción (el programa). Solo que, en este caso, la construcción no tendrá ninguna utilidad práctica, salvo ocupar espacio

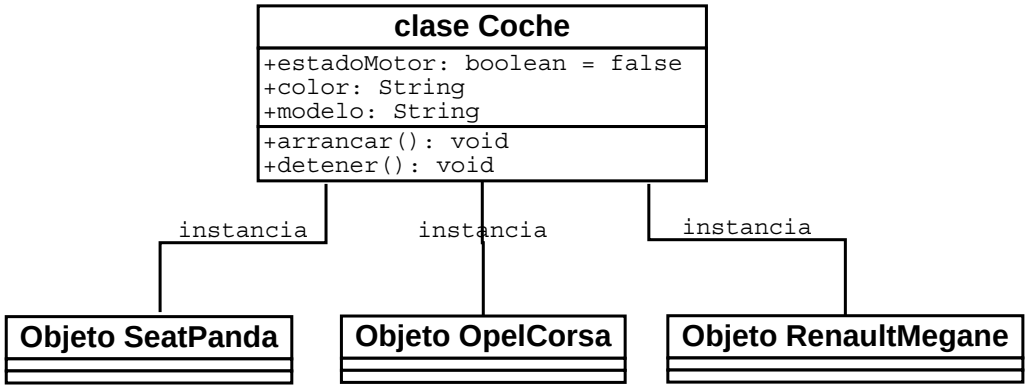

Figura 2.1: La clase Coche y tres objetos.

en la estantería de nuestra habitación, y que nuestras madres tengan un trasto más al que limpiar el polvo.

Mediante estos ejemplos ya podemos vislumbrar algunas de las características de los objetos:

- Realizan una tarea por sí solos.

- Proporcionan encapsulación: Es posible ocultar las partes internas de la implementación de un objeto, permitiendo el acceso sólo a través de una interfaz bien conocida y definida.

- Son reutilizables.

- Proporcionan escalabilidad (el programa puede crecer) y modularidad (el programa se puede dividir en bloques que faciliten su comprensión).

En Java ocurre lo mismo que en los ejemplos anteriores. Programaremos una serie de objetos independientes con una funcionalidad determinada, y los juntaremos para crear un programa. Pero, para crear objetos, primero debemos hablar de las clases.

2.3. Clases en Java. Atributos y Métodos

Cuando escribimos un programa en un lenguaje orientado a objetos, no estamos definiendo objetos, sino clases. Una clase es la unidad fundamental en programación, la pieza de Lego. El problema es... que no existe. Es una abstracción. Es la plantilla que utilizaremos posteriormente para crear un conjunto de objetos con características similares.

En estos momentos, cualquier manual decente de Java comienza a introducir ejemplos sobre figuras geométricas, especies de árboles que derivan de una idea abstracta, y otros símiles igual de esotéricos. En nuestro caso, vamos a utilizar el ejemplo de los coches.

Supongamos que definimos una clase `Coche`. No tiene entidad física. Hablamos de un coche en general, sin especificar de qué tipo de coche se trata. Podemos asignarle un comportamiento y una serie de características, como se muestra en la figura 2.1. A partir de esa clase `Coche`, podremos crear nuestros objetos (también llamados instancias), que serán las realizaciones "físicas" de la clase. En el ejemplo, se muestran un *Seat Panda*, un *Opel Corsa*, y un *Renault Megane*. Todos ellos comparten una serie de características comunes por las que podemos identificarlos como coches.

Vemos en la figura que, respecto al comportamiento, podemos `arrancar` y `detener` nuestro coche. Esos son los métodos de la clase. En cuanto a las características (llamadas atributos), tenemos las variables `estadoMotor`, `color`, y `modelo`. Para definir nuestra clase en Java, lo haremos de la siguiente manera (listado 2.3.1):

Programa 2.3.1 La clase Coche.

```
class Coche {

  boolean estadoMotor = false;
  String color;
  String modelo;

  void arrancar(){

    if(estadoMotor == true)
       System.out.println("El coche ya está arrancado");

    else{
       estadoMotor=true;
       System.out.println("Coche arrancado");
    }

  }


  void detener(){

    if(estadoMotor == false)
       System.out.println("El coche ya está detenido");

    else{
       estadoMotor=false;
       System.out.println("Coche detenido");
    }

  }

}  // fin de la clase Coche
```

Analicemos paso a paso cada parte de la clase:

- Las clases se definen con la palabra reservada `class`. Todo el código que pertenezca a esa clase se encierra entre dos llaves.
- Los nombres de las clases, por norma, comienzan con mayúscula.
- A continuación tenemos tres atributos, definidos por las variables `estadoMotor`, `color` y `modelo`. Los atributos nos definen las características que tendrá cada objeto de esa clase, y que podrán ser distintas para cada uno de ellos. Es decir, cada coche será de un modelo determinado, de un color, y su motor estará encendido o apagado. Por ahora no nos interesa que los tipos sean `String` o `boolean`. Eso se verá en el capítulo siguiente.
- Por último, tenemos dos métodos, `arrancar()` y `detener()`. Los métodos nos definen el comportamiento que tendrán los objetos de esa clase. En nuestro ejemplo, podremos arrancar o detener el motor de nuestros coches. Tampoco nos interesan por ahora las sentencias condicionales if-else. Se estudiarán en el capítulo 3.

Podemos escribir la clase anterior y guardarla en un archivo con el nombre *ej1.java*, por ejemplo. Como ya se explicó en la introducción, todos los archivos con código fuente de Java llevan

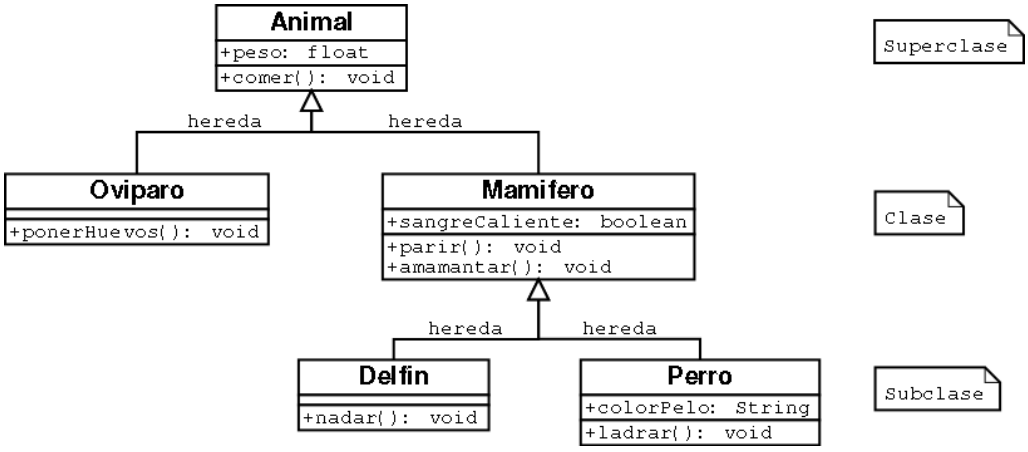

Figura 2.2: Herencia de Clases.

la extensión *java*. Si lo compilamos (`javac ej1.java`) obtendremos un archivo binario llamado `Coche.class` (nótese que, si hubiéramos definido varias clases en el mismo archivo, el compilador nos crearía tantos binarios con extensión `class` como clases hubiéramos definido). Si, a continuación, intentamos ejecutar esa clase (`java Coche`), nos dará un error, informándonos de que no tenemos creado un método *main*. Es decir, que una clase, por sí sola, no puede ejecutarse. O bien definimos un método *main* que nos cree un objeto de esa clase con el que trabajar, o bien llamamos a un objeto de esa clase desde otro objeto perteneciente a otra clase. Todo esto se verá en el capítulo 4.

2.4. Herencia

Por lo explicado hasta ahora, podría parecer que la programación orientada a objetos se basa simplemente en crear clases independientes que se relacionan entre sí. En realidad, existe una relación más compleja entre clases. Es la **herencia**.

Todas las clases en Java existen dentro de una jerarquía. Cada clase tiene una (y sólo una) clase por encima de ella, denominada *superclase*, y cualquier número de clases (o ninguna) por debajo. A estas últimas se las denomina *subclases*.

Una clase heredará los métodos y variables de su superclase. Del mismo modo, sus subclases heredarán los métodos y variables de esa clase.

Observemos la figura 2.2. En ella se muestran 5 clases y su relación de herencia. Por encima de todas tenemos una superclase `Animal`, con un atributo `peso` y un método `comer()` (todos los animales tienen peso y comen). Debajo de ésta aparecen otras dos clases con tipos de animales: `Oviparos` y `Mamiferos`[1] . Los Ovíparos pueden `ponerHuevos()`. Los Mamíferos pueden `parir()` y `amamantar()` a sus crias, y pueden tener la `sangreCaliente`, o no (el tipo `boolean` especifica que ese atributo puede ser cierto o falso. Se verá en el siguiente capítulo). Estas dos clases, aparte de tener su propio comportamiento y sus características, heredan también los métodos y atributos de su superclase. Por ello, tanto los Mamíferos como los Ovíparos pueden `comer()` y tienen un `peso`.

Asímismo, la clase `Mamífero` tiene dos subclases que heredan de ella. Son las subclases `Perro` y `Delfin`. El perro, por ejemplo, tendrá un color de pelo determinado, y podrá ladrar (el mío lo hace mucho; en ocasiones, demasiado :-). Y además, debido a la herencia, tendrá un peso, podrá comer, tendrá la sangre caliente y amamantará a sus crías.

¿Queda claro?. Cada subclase extiende y concreta la funcionalidad y características de su superclase. Es decir, se "especializa" más.

Bien, supongamos que tenemos una clase escrita en Java. La manera de especificar que una clase es subclase de otra se hace utilizando la palabra reservada *extends*. En el siguiente ejemplo

[1] Que nadie tome en serio este tipo de clasificación. Como puede comprobarse, no tengo ni idea de zoología :-)

definiremos las clases `Animal` y `Mamifero`.

Programa 2.4.1 La clase Mamifero y su superclase Animal.

```
class Animal{

    float peso;
    void comer(){ }

}

class Mamifero extends Animal{

    boolean sangreCaliente;
    void parir(){ }
    void amamantar(){ }

}
```

Vemos que, al definir la clase `Mamifero`, añadimos la coletilla `extends Animal`, para especificar que heredamos de la clase `Animal`. Aprovecho para comentar que, cuando se programa, no se utilizan acentos. Por eso definimos la clase como *Mamifero*, en vez de *Mamífero*.

Al principio de este apartado se explicó que todas las clases existen dentro de una jerarquía y que siempre heredan de su superclase. Seguro que, a estas alturas, el lector avispado se estará preguntando de quién hereda la clase `Animal`, puesto que no hemos añadido ningún *extends*. La respuesta es que todas las clases en las que no se especifique nada, heredarán de la clase `Object`. Esta clase es la superior en la jerarquía de clases de Java, y es la única que no hereda de nadie.

Un último apunte. El hecho de que una clase Java sólo pueda heredar de su superclase se denomina *herencia simple*. Todas las clases heredan de una, y sólo una clase (excepto la clase `Object`, claro, que no hereda de nadie). En otros lenguajes, como C++, existe el concepto de herencia múltiple, en el que una clase puede heredar de dos o más superclases. Pero no en Java, lo cual simplifica enormemente la programación sin restarle potencia.

2.5. Paquetes

Un paquete es un conjunto de clases e interfaces (que se estudiarán en el tema 6) relacionados, los cuales se agrupan juntos. Es una forma de crear librerías. Por ejemplo, supongamos que escribimos un conjunto de clases que, combinándolas, permiten calcular integrales. Podemos agruparlas en un paquete de modo que otro programador, que esté interesado en programar una calculadora científica, pueda utilizar. Cuando quiera que su calculadora calcule integrales, llamará a las clases contenidas en ese paquete.

Supongamos que tenemos, en nuestro paquete del ejemplo anterior, integrales de línea y de superficie. Podríamos crear un paquete `integrales`, y dentro de éste, otros dos, uno llamado `linea`, y otro, `superficie`. Imaginemos que, dentro del paquete `superficie`, tenemos, entre otras clases, la clase `Diferencial`. ¿Cómo llamamos a esa clase?. Referenciándola, al principio de nuestro fichero java, con la palabra reservada import:

```
import integrales.superficie.Diferencial;
```

Si quisiéramos utilizar todas las clases contenidas en el paquete de integrales de superficie, tendríamos que escribir:

```
import integrales.superficie.*;
```

Es importante indicar que, cuando usamos el asterisco, nos referimos sólo a las clases que estén en ese paquete, pero no las de los subpaquetes. Es decir, que `import integrales.*` importará las clases contenidas en el paquete integrales, pero no las del paquete superficie. Si necesitamos las clases de este último, tendremos que especificarlo directamente.

Todas las clases que nos proporciona el API de Java vienen empaquetadas. Y cada vez que queramos utilizar alguna de ellas, debemos referenciarla. Las clases del API están contenidas en un paquete llamado *java*. Por tanto, si queremos utilizar la clase `Button.class`, que pertenece al paquete `java.awt`, tendremos que especificarlo como:

```
import java.awt.Button;
```

o bien indicarlo en el momento de usar la clase para crear un objeto:

```
//código anterior...
java.awt.Button boton;
//código posterior...
```

Nótese que:

- Cuando tenemos unos paquetes dentro de otros, se especifican separándolos por puntos. Por ejemplo, `java.awt.Button`, para referirnos a la clase `Button.class`, contenida en el paquete `awt`, que a su vez está contenido en el paquete `java`.

- En los ejemplos propuestos de este tema no se ha llamado a `import` en ningún momento y, sin embargo, todos ellos se compilan correctamente. ¿Por qué?. Porque, de manera predeterminada, se tiene acceso al paquete `java.lang` sin tener que especificar el `import`. Este paquete contiene las clases necesarias para poder realizar las funciones más básicas en Java, como mostrar un texto por pantalla. Naturalmente, la clase `Object`, de la cual heredan todas las demás clases, pertenece al paquete `java.lang`. Por tanto, formalmente hablaremos de la clase `java.lang.Object`.

- ¿Qué ocurriría si importáramos dos paquetes distintos, y ambos contuviesen una clase con el mismo nombre?. Nada, el compilador de Java daría un error de conflicto de nombres, y nos obligaría a definir explícitamente la clase con la que queremos trabajar.

Por último, para terminar, aprenderemos a crear nuestros propios paquetes. Siguiendo con el ejemplo de las integrales, lo primero será especificar, en la primera línea de nuestro archivo fuente `diferencial.java`, que su contenido pertenece al paquete `integrales.superficie`. Esto se hace con la palabra reservada `package`:

```
package integrales.superficie;
```

Tras compilar nuestro código fuente, obtendremos una clase llamada `Diferencial.class`. Crearemos un directorio en nuestro disco duro con el nombre *integrales*. A continuación, dentro de éste, crearemos otro llamado *superficie* y, en su interior, copiaremos nuestra clase `Diferencial.class`. De este modo, cuando en otros programas incluyamos la sentencia `import integrales.superficie.Diferencial`, podremos utilizar esa clase [2]. Es importante ver que, si intentamos invocar a nuestra clase sin incluirla en la estructura de directorios correspondiente, el compilador de java nos dará un error, indicando que no encuentra la clase.

Esta estructura de directorios se puede comprimir en formato *zip* (el que utilizan programas como Winzip) y guardarlos con extensión *jar*, o utilizar el propio compresor *jar*, proporcionado por el J2sdk de Java El compilador podrá acceder a su contenido igualmente, y conseguiremos que los paquetes ocupen menos espacio y estén más organizados.

[2] Doy por sentado que el CLASSPATH está correctamente configurado para que el compilador de java pueda encontrar ese paquete. Te remito al tema 1 para aprender a configurar el CLASSPATH.

2.6. Resumen

En este capítulo hemos aprendido una serie de conceptos que es necesario recordar:

- Todos los programas en Java están formados por objetos independientes que pueden comunicarse entre sí.
- Los objetos tienen unas características o atributos, definidos por sus variables, y un comportamiento, determinado por sus métodos.
- Para crear objetos es necesario definir clases, que son una abstracción de esos objetos, una plantilla que utilizaremos posteriormente para crear un conjunto de objetos con características similares.
- Todas las clases heredan atributos y métodos de su superclase, extendiendo la funcionalidad de ésta. La única clase que no hereda de nadie es la clase `java.lang.Object`.
- Las clases se pueden agrupar en paquetes, para facilitar su utilización y organización.

Capítulo 3

Fundamentos del lenguaje Java

3.1. Introducción

Seguro que a estas alturas estarás deseando poner en práctica todos los conceptos teóricos aprendidos hasta ahora y empezar a escribir código en Java. En este tema introduciremos las herramientas necesarias para que puedas hacerlo, y en el siguiente podrás comenzar a escribir tus propios programas.

En el capítulo anterior estudiamos la programación orientada a objetos. Para ello, fue necesario mostrar algunos fragmentos de código en Java. Sin embargo, no fue posible explicarlos con detenimiento, debido a que aún no sabemos cómo trabajar con variables, o cuál es la sintaxis de una instrucción. Ese es el objetivo de este segundo capítulo. Estudiaremos, entre otros, las variables, los tipos de datos, los literales, explicaremos la forma de definir instrucciones, hablaremos de los operadores aritméticos y lógicos, y mostraremos el modo de controlar el flujo de un programa.

3.2. Comentarios

Los comentarios en java pueden hacerse de dos formas:

```
/* Esto es un comentario
de varias lineas */
int variable; //Esto es un comentario al final de línea.
```

Cuando queramos hacer comentarios que ocupen varias líneas, utilizaremos `/*` y `*/`. El segundo tipo de comentario, con las dos barras, `//`, considera como comentario todo lo que haya a partir de ellas hasta el final de línea.

Existe un tercer tipo de comentarios, utilizado por el sistema *javadoc* de documentación de Java. *Javadoc* permite crear documentación de nuestros programas en HTML con el mismo formato que el API de Java (elegante, ¿verdad?). Para aprender a utilizar *javadoc* te remito a la documentación del SDK, que se descarga junto con la del API.

3.3. Variables y Tipos de Datos

Las variables (lo que llamábamos atributos en el capítulo anterior) son zonas de memoria donde pueden almacenarse valores. Estos valores pueden ser un número, un carácter, un trozo de texto, etc. Cada variable tiene un tipo, un nombre y un valor. Para utilizar una variable, es preciso declararla primero. Por ejemplo:

```
int numeroEntero;
char caracter;
String cadenaTexto;
boolean motorEnMarcha;
```

(Nótese que en estos ejemplos sólo se muestra el tipo (`int`, `char`, etc.) y el nombre, no el valor).

3.3.1. El nombre de la variable

Las variables pueden definirse en cualquier parte de un método, aunque lo más ortodoxo es hacerlo al principio. El nombre de una variable sólo puede comenzar por una letra, un guión bajo ("`_`") o el signo del dólar ("`$`"). No puede comenzar por un número. A partir del segundo carácter, puede incluir los caracteres o números que queramos. Y mucho ojo con las mayúsculas: la variable `casa` es distinta de la variable `Casa` y de la variable `caSa`.

Aunque no es obligatorio, por costumbre suelen definirse las variables con la primera letra en minúscula. Si definimos una variable con varias palabras juntas, como `motorEnMarcha`, suelen ponerse en mayúsculas la primera letra de las restantes palabras que forman el nombre de la variable.

3.3.2. La asignación

Podemos asignar valores a las variables de las siguientes formas:

```
int numeroEntero, numero2Entero = 3;
int x = 4, y = 5, z = 6;
boolean motorEnMarcha = true;
String texto;
texto = "Esto es una cadena de texto";
```

Lo primero en lo que nos fijamos es que podemos definir varias variables en una misma línea, separándolas con comas. En el primer caso, definimos dos variables de tipo entero, pero sólo asignamos el valor 3 a `numero2Entero`. En el segundo, asignamos valores a las tres variables `x`, `y`, y `z`, en la misma línea. En el tercero, damos el valor `true` (verdadero) a una variable booleana, y en el último, definimos simplemente una variable de tipo `String`, a la cual se le asigna una cadena de texto más abajo.

3.3.3. El tipo

El tipo de una variable puede ser:

- Uno de los tipos primitivos.

- Una clase o interfaz.

- Un array de elementos.[1]

Los **tipos primitivos** son ocho:

[1] No confundir un array de elementos con la clase Array, definida en el API de Java. Los arrays se explican en el capítulo 4.

Tipo	Definición	Rango
boolean	Tipo de dato booleano	true o false
char	Caracter de 16 bits	Todos los caracteres Unicode
byte	Entero de 8 bits con signo	-128 a 127
short	Entero de 16 bits con signo	-32.768 a 32.767
int	Entero de 32 bits con signo	-2.147.483.648 a 2.147.483.647
long	Entero de 64 bits con signo	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
float	Punto Flotante de 32 bit	
double	Punto Flotante de 64 bit	

Tabla 3.3.1: Tipos Primitivos en Java.

Los **tipos de clase** son variables que almacenan una instancia (es decir, un objeto) de una clase. Por ejemplo:

```
String texto;
Font courier;
Mamifero mascota;
```

Es importante recordar que, si definimos una variable para almacenar una instancia de una clase determinada, nos servirá también para instancias de subclases de esa clase. Por ejemplo, la variable `mascota` nos permitirá almacenar una instancia de la clase `Mamifero`, pero también podrá contener instancias de las clases `Delfin`, `Perro` y `Gato`, que son subclases de `Mamifero`.

Quizá estés pensando que podrían definirse todas las variables de tipo `Object`, y así podrían contener cualquier instancia de cualquier clase, puesto que todas las clases heredan de `Object`. Pues sí, por poder, se puede, pero no es una buena costumbre en programación. Es mejor ceñirse al tipo de objetos con los que se esté trabajando.

Un array es un tipo de objeto que puede almacenar un conjunto ordenado de elementos. Por ahora no nos interesan demasiado, ya que aún no sabemos trabajar con objetos. Solo es importante saber que el tipo de una variable puede ser un array. A partir del capítulo 3 empezaremos a trabajar con ellos.

Existe un tipo especial de variables: las variables finales. Realmente no son variables, sino constantes. Una vez que definamos su valor, no podremos cambiarlo. Se define una variable final del siguiente modo:

```
final int a=6;
```

Evidentemente, puede ser `int`, `float`, `double`, etc.

3.4. Literales

Los literales se refieren a la forma en que especificamos un valor. Tendremos literales de enteros, de punto flotante, de caracteres, de cadenas, etc.

3.4.1. Literales de enteros

Podemos especificar el valor de un entero de distintas formas:

```
int entero = 65535;  // Valor en decimal
int entero = 0xFFFF; // El mismo valor, pero en hexadecimal
                     // (nótese el 0x al principio)
int entero = 0177777; // El mismo valor, en octal (se especifica
                      // mediante un 0 al principio)
```

```
long enteroLargo = 22L; // Entero de 64 bits (se utiliza L al final)
long enteroLargo = 22;  // 22 es un int, pero es convertido a long.
```

En este último ejemplo se ve que, si se asigna un valor de un tipo determinado (22, por defecto, es un `int`) a una variable de mayor rango (`enteroLargo` es un `long`), se realiza una conversión automática a ese tipo. Si queremos hacer lo contrario, es decir, una conversión a un tipo de menor rango, es necesario hacer la conversión (llamada *cast*) explícitamente:

```
int entero = 13;
byte enteroCorto = (byte) entero; // Forzamos la conversión de
                                  // entero a byte.
```

3.4.2. Literales de punto flotante

Los valores en punto flotante se pueden especificar en formato decimal o en notación científica. Por ejemplo:

```
double flotanteLargo = 5.34;      //Notación decimal.
double flotanteLargo = 2 {*} 3.14 //Otro ejemplo,notación decimal.
double flotanteLargo = 3.00e+8;   //Notación científica.
float flotanteCorto = 5.34F;      //Notación decimal para PF de 32 bits.
float flotanteCorto = 3.00e+8F;   //Notación científica para PF de 32 bits.
```

Por defecto, cualquier literal de punto flotante es de tipo `double`. Si queremos que sea de tipo `float`, debemos especificarlo añadiendo una `F` al final del valor, como se muestra en el ejemplo anterior para la variable `flotanteCorto`.

3.4.3. Literales de caracteres

Los caracteres se definen utilizando comillas simples. Se admiten también caracteres de escape **ASCII** o **Unicode**. Por ejemplo:

```
char caracter = 'a';
char saltoLinea = '\n'; // Carácter de escape de salto de línea.
```

Los caracteres de escape más utilizados son: `\n` (salto de línea), `\t` (tabulación), `\r` (retorno de carro) y `\b` (retroceso). Si queremos almacenar en la variable valores especiales como una comilla doble (`"`), lo haremos utilizando también la barra invertida:

```
char comilla = '\"';
```

3.4.4. Literales de cadena

Las cadenas las representaremos mediante un conjunto de caracteres y secuencias de escape entre comillas dobles. Normalmente, serán instancias de la clase `String`. Por ejemplo:

```
String cadena = "Esto es una cadena";
String cadena = ""; // Cadena vacía.
String cadena = "Cadena con \t tabulación en medio";
String cadena = "Cadena con \"texto entrecomillado\" en su interior";
```

El tratamiento que hace Java de las cadenas es muy completo. Se verá con más detalle en el capítulo 4, cuando estudiemos la creación de objetos.

3.5. Instrucciones

La sintaxis de las instrucciones en Java es muy similar a la de lenguajes como C o C++. Todas las instrucciones y expresiones aparecen dentro de un bloque de código. Un bloque de código es todo lo que esté contenido dentro de un par de llaves ("{" y "}").

Cuando escribimos una clase, delimitamos todo lo que pertenece a ésta dentro de un bloque de código. Igualmente con los métodos que tiene esa clase. Por ejemplo:

Programa 3.5.1 Ejemplo de bloques de código.

```
class ClaseEjemplo{

    int variable=1;
    void metodo(){
       if(variable == 1)
          System.out.println("La variable vale 1");
       else{
          System.out.println("La variable vale 0");
       }
    }
}
```

Vemos que:

- Todas las instrucciones y expresiones finalizan con un punto y coma (`;`), con excepción de las sentencias de control de flujo (`if` y `else`).

- El alcance de una variable se limita al par de llaves en el que está contenido. En el ejemplo, la variable sólo existe y tiene valor 1 dentro de la clase `ClaseEjemplo`.

- Quizá te estés preguntando por qué la instrucción que aparece debajo de la sentencia if no va entre llaves. Lo explicaremos enseguida, cuando hablemos de las sentencias de control de flujo.

3.6. Expresiones y Operadores. Preferencia

Una expresión es una instrucción que devuelve un valor. Ejemplos de expresiones son:

```
3 + 2
7 / 5
8 * 4
5.3 - 8.1
20  % 7
```

Los ejemplos anteriores son expresiones aritméticas, pero también puede tratarse de expresiones que devuelvan un objeto. Las expresiones utilizan operadores. Los símbolos `+`, `-`, `*`, `/` y `%` son operadores. Definen la operación a realizar entre los operandos.

Existe una relación de prioridad entre los operadores. Por ejemplo, si yo escribo: `3*2 + 5*4`, primero se evaluarán los productos, y después la suma, porque tiene mayor prioridad el operador `*` que `+`. Por tanto: `3*2 + 5*4 = 6 + 20 = 26`. Si quisiéramos cambiar la prioridad, para que primero se evaluara la suma, utilizaríamos paréntesis: `3*(2 + 5)*4= 84`.

Existe una gran cantidad de operadores en Java. En la tabla siguiente se muestran los más utilizados, su prioridad, los operandos a los que se aplican, y una breve descripción:

Prioridad	Operador	Operandos	Descripción
1	++,- -	Aritméticos	Incremento y Decremento
1	!	Booleano	Complemento Lógico
1	~	Entero	Complemento en binario
1	(tipo)	Cualquiera	Cast
2	*,/,%	Aritméticos	Multiplicación,división, módulo o residuo
3	+,-	Aritméticos	Adición y sustracción
3	+	String	Concatenación de cadenas
4	<<	Entero	Desplazamiento de bits hacia la izquierda
4	>>	Entero	Desplazamiento de bits hacia la derecha con extensión de signo
5	<,<=,>,>=	Aritméticos	Comparación numérica
5	instanceof	Objeto	Comparación de tipo
6	==, !=	Tipos primitivos	Igualdad o desigualdad de valor
6	==,!=	Objeto	Igualdad o desigualdad de referencia
7	&	Entero	AND de bits
7	&	Booleano	AND lógico
8	^	Entero	XOR de bits
8	^	Booleano	XOR lógico
9	|	Entero	OR de bits
9	|	Booleano	OR lógico
10	&&	Booleano	AND condicional
11	||	Booleano	OR condicional
12	=	Cualquiera	Asignación
12	*=,/=,%=, +=,-= ,<<=, >>=, >>>=,&=, ^=,|=	Cualquiera	Asignación con operación

Tabla 3.6.1: Operadores más utilizados en Java.

Nótese que los operadores lógicos utilizan operandos booleanos. Por ejemplo, supongamos que tengo cuatro variables `a=1`, `b=2`, `c=3` y `d=3,` y hago la comparación:

```
(a==b) || (c==d)
```

`a==b` y `c==d` devuelven valores booleanos, esto es, `true` o `false`. `a==b` devolverá `false`. `c==d` devolverá `true`. Como son booleanos, podemos aplicarlos al operador `||` (OR condicional). Tendremos, por tanto, `false OR true`, y el resultado será `true`.

También es necesario hacer hincapié en el operador lógico `&` y en el condicional `&&`. La expresión en la que los utilicemos será verdadera si y sólo si las dos expresiones que están a los lados del operador son verdaderas. Si no, el resultado será `false`. La diferencia entre `&` y `&&` es la forma de evaluación. `&` significa que ambos lados de la expresión serán evaluados sin importar su resultado. Utilizar `&&` significa que, si el lado izquierdo de la expresión es falso, la expresión completa es valorada como falsa, independientemente del lado derecho, que nunca se evaluará.

Una propiedad de Java que no existe en otros lenguajes como C es la posiblidad de concatenar cadenas. Por ejemplo:

```
String cadena1 = "Esta es la cadena 1";
String cadena2 = cadena1 + " y esta es la cadena 2";
```

El valor de `cadena2` será: *"Esta es la cadena 1 y esta es la cadena 2"*; Nótese el espacio entre las comillas y el texto en `cadena2`.

Se irá viendo la utilización de los operandos en ejemplos en los siguientes capítulos. Pero quiero comentar aquí que existe una clase `java.lang.Math` que permite realizar operaciones más complejas (exponenciales, trigonometría), por lo que no tendremos que implementarlas nosotros.

3.7. Control de Flujo

Una vez explicados los tipos y los bloques de código, y para terminar el tema de una puñetera vez, vamos a ver las sentencias que nos permiten modificar el flujo de un programa, es decir, la forma en que éste transcurre.

3.7.1. if-else

En primer lugar tenemos las **sentencias condicionales**. Su sintaxis es:

```
if(condicion)
   instrucción o bloque de instrucciones;
else
   instrucción o bloque de instrucciones;
```

Esto quiere decir que, si se cumple la condición, que será siempre una expresión booleana (`if` es nuestro *"si"* condicional pero en inglés, para los despistados), se ejecutará la instrucción o el bloque de instrucciones que hay a continuación. Si es un bloque de instrucciones, es obligatorio meterlo entre llaves, como se explicó en la sección 3.5 (Instrucciones). Si se trata de una sola instrucción, no es necesario. Ésta es la explicación de que la instrucción del programa 3.5.1, vista antes, no vaya entre llaves.

Si no se cumple la condición del `if`, se ejecutará la instrucción o instrucciones que aparezcan a continuación del `else`. Del mismo modo, si se trata de más de una intrucción, irán entre llaves. El `else` ES OPTATIVO. Puede que no necesitemos especificar una acción a realizar en caso de que no se cumpla la condición.

Un ejemplo de la sentencia condicional dentro de una clase:

Programa 3.7.1 Ejemplo de uso de la sentencia if.

```
class ControlFlujo{

      public static void main(String args[]){
         String cadena1="hola";
         String cadena2="hola";
         String cadena3;

         if(cadena1.equals(cadena2)){
            System.out.println("Las dos cadenas son iguales");
            cadena3 = cadena2;
            System.out.println("Cadena3: "+cadena3);
         }
         else
            System.out.println("Las dos cadenas son distintas");
      }
}
```

3.7.2. El condicional switch

Permite comparar una variable con un valor. Si no coincide, lo compara con otro valor siguiente, y así cierto número de veces. Por ejemplo:

Programa 3.7.2 Ejemplo de uso de la sentencia switch.

```
class UsoDeSwitch{

        public static void main(String args[]){
                int a=3;
                switch(a){
                   case 1:
                        System.out.println("El valor de a es 1");
                        break;
                   case 2:
                        System.out.println("El valor de a es 2");
                        break;
                   case 3:
                        System.out.println("El valor de a es 3");
                        break;
                   default:
                        System.out.println("a es distinto de 1, 2 o 3.");
                }
        }
}
```

Puntos importantes de esta sentencia:

- El valor de a es comparado con cada uno de los valores que hay a continuación del `case`. Si coincide con alguno de ellos, se ejecutan las sentencias que hay a continuación de ese case. Si no, sigue comprobando los siguientes `case`. Si no coincide con ninguno de ellos, se ejecutan las sentencias que hay a continuación de `default` (`default` es optativo).

- La condición a evaluar, es decir, lo que tiene `switch` entre paréntesis (`a`) sólo puede ser un tipo primitivo que pueda ser convertido a `int`, como, por ejemplo, `char`. No se pueden utilizar `float` ni `double` (si a=5.45, ¿a qué entero lo convertiríamos?. No tiene sentido.).

- Después de cada case se ponen dos puntos ("`:`"). Las sentencias que van a continuación de ese `case` no necesitan ir entre llaves.

- La sentencia `break` obliga al programa a salir del `switch` cuando llegue a ella. De ese modo, una vez que ejecutemos el `case` correspondiente, salimos del `switch`. Si no la pusiéramos, se seguirían ejecutando los siguientes `case` que hubiera hasta encontrar un `break`, o llegar al final del `switch`.

Los valores que hay a continuación de cada `case` son constantes. Si queremos utilizar variables, tendrán que ser de tipo `final` (es decir, constantes, al fin y al cabo):

Programa 3.7.3 Ejemplo de uso de switch con variables finales.

```
class UsoDeSwitch2{
        public static void main(String args[]){
                int a=3;
                final int b=1,c=2,d=3;
                switch(a){
                   case b:
                        System.out.println("El valor de a coincide con b");
                        break;
                   case c:
                        System.out.println("El valor de a coincide con c");
                        break;
                   case d: System.out.println("El valor de a coincide con d");
                        break;
                   default:
                        System.out.println("a es distinto de b, c y d.");
                }
        }
}
```

3.7.3. Bucles while y do-while

En ocasiones, necesitaremos que un conjunto de instrucciones se repitan hasta que se cumpla una determinada condición. En ese caso, utilizaremos bucles `while`. Por ejemplo:

Programa 3.7.4 Un bucle while.

```
class UsoDeWhile{
        public static void main(String args[]){
             int a=0,b=10;
             while(a<10 && b>0){
                 System.out.println("Valor de a: "+a+" .Valor de b: "+b);
                 a++; b--;
             }
        }
}
```

En este ejemplo se va incrementando el valor de la variable `a`, y decrementando el valor de `b`. Mostraremos el valor de las variables mientras se cumpla que `a<10` y que `b>0`.

También podemos utilizar un bucle `do-while`:

Programa 3.7.5 Un bucle do-while.

```
class UsoDeDoWhile{
        public static void main(String args[]){
           int a=0,b=10;
           do{
              System.out.println("Valor de a: "+a+" .Valor de b: "+b);
              a++; b--;
           } while(a<10 && b>0);
        }
}
```

Las dos clases, al ejecutarlas, devuelven el siguiente resultado:

```
Valor de a: 0 .Valor de b: 10
Valor de a: 1 .Valor de b: 9
Valor de a: 2 .Valor de b: 8
Valor de a: 3 .Valor de b: 7
Valor de a: 4 .Valor de b: 6
Valor de a: 5 .Valor de b: 5
Valor de a: 6 .Valor de b: 4
Valor de a: 7 .Valor de b: 3
Valor de a: 8 .Valor de b: 2
Valor de a: 9 .Valor de b: 1
```

¿Cuál es la diferencia entre los dos tipos de bucle?. En un bucle `while`, si no se cumple la condición, no se entra. Por ejemplo, si `a` hubiese valido 10 desde el principio, no se habría cumplido la condición, no habríamos entrado en el bucle, y no saldría nada por pantalla. Sin embargo, en un bucle `do-while`, siempre se ejecuta la primera iteración[2] antes de comprobar la condición, por lo que hubiésemos tenido en la salida una línea indicándonos que a valía 10.

3.7.4. Bucles for

Al igual que los anteriores, permite ejecutar un número determinado de iteraciones hasta que se cumple una condición.

Programa 3.7.6 Un bucle for.

```
class UsoDeFor{
    public static void main(String args[]){
            int a,b=10;
            for(a=0;a<10;a++){
              System.out.println("Valor de a: "+a+" .Valor de b: "+b);
              b--;
            }
    }
}
```

El resultado por pantalla al ejecutar esta clase es el mismo que el obtenido anteriormente con `while` y `do-while`.

La estructura de la sentencia `for` es: `for(inicialización; condición; incremento)`. Inicializamos la variable con un determinado valor, la incrementamos en cada iteración con el valor indicado en `incremento` (en el ejemplo, de uno en uno. Si el valor es negativo, decrementamos), y repetiremos el bucle hasta que lleguemos a la condición indicada (`a<10`);

Un último comentario a los bucles: Si queremos romper un bucle, es decir, salir de éste aunque no hayamos terminado de ejecutar todas las iteraciones, o porque se cumpla alguna condición, podemos utilizar la sentencia `break`, al igual que se hacía con la instrucción `switch`.

3.8. Resumen

Bueno, este capítulo ha resultado ser una chapa impresionante, incluso para el autor. La idea no es memorizar cuántos tipos de operadores existen, y a qué operandos se aplican, sino ir aprendiéndolos a medida que se utilicen, y usar este capítulo como una referencia cuando, por ejemplo, no recordemos la sintaxis de la sentencia `for`.

[2]Una iteración es cada una de las repeticiones que realiza la sentencia.

Los puntos más importantes que deben recordarse de este tema son:

- Toda variable tiene un tipo, un nombre y un valor.
- El tipo de una variable puede ser uno de los 8 tipos primitivos, una clase, o un array de elementos.
- Los literales definen la forma en que especificamos el valor de una variable. Por ejemplo: `int a=0xFFFF`, en hexadecimal.
- Las instrucciones se agrupan en bloques de código, delimitados por llaves.
- Una expresión es una instrucción que devuelve un valor. Las expresiones utilizan operadores y operandos. Por ejemplo: `3 + 2`.
- Es posible concatenar cadenas de texto mediante el símbolo `+`.
- Existen dos sentencias condicionales: `if-else` y `switch`.
- Existen tres sentencias para manejar bucles: `while`, `do-while`, y `for`.

Capítulo 4

Trabajando con Objetos

4.1. Introducción

En el primer tema se puso de manifiesto que los objetos tienen unas características o atributos, definidos por sus variables, y un comportamiento, determinado por sus métodos. También sabemos que los objetos se crean a partir de clases. En este tema aprenderemos a crear, destruir y trabajar con objetos. También aprenderemos a invocar a los métodos y atributos de un objeto, veremos qué es el *casting*, y estudiaremos los arrays y las cadenas de caracteres.

4.2. Creación y Destrucción de Objetos

En primer lugar, hay que aclarar el concepto de *instancia*. Una instancia es un objeto, Por tanto, cuando hablemos de *instanciar una clase*, nos estaremos refiriendo al proceso de crear un objeto a partir de esa clase.

Para crear un objeto utilizaremos el operador `new` con el nombre de la clase que se desea instanciar:

```
Coche ferrari = new Coche();
```

Vamos por partes:

- Definimos una variable cuyo tipo es la clase `Coche` (recuérdese del tema anterior que, de entre los tipos que puede tener una variable, están las clases). Le damos el nombre `ferrari`, y almacenaremos ahí una instancia de la clase `Coche`.

- A la clase que se desea instanciar se le añaden dos paréntesis, que pueden estar vacíos, como en el ejemplo, o contener unos parámetros de inicialización. Estos parámetros los encontraremos definidos en el API de Java, si trabajamos con clases del jdk, bajo el epígrafe *"Constructor Summary"*. Si son clases nuestras, o de terceros, deberán proporcionarnos documentación en la que se especifiquen esos parámetros.
  Lo que estamos haciendo realmente, al invocar a la clase con los dos paréntesis, es llamar a su constructor. El constructor no es más que un método que inicializa los parámetros que pueda necesitar la clase. En secciones posteriores se verá con detalle. Por ahora, sólo nos interesa el hecho de que la forma de crear una instancia es llamando al constructor de la clase.

Hemos creado un objeto, pero ¿qué pasa cuando ya no lo necesitemos más?, ¿existe alguna forma de destruirlo?. La respuesta es que sí es posible, pero no hace falta. En Java, la administración de memoria es dinámica y automática. Existe un *recolector de basura* que se ocupa, continuamente, de buscar objetos no utilizados y borrarlos, liberando memoria en el ordenador.

4.3. Invocación de Variables y Métodos

En el segundo tema se explicó que una clase, por sí sola, no puede ejecutarse. O bien definimos un método `main` que nos cree un objeto de esa clase, con el que podremos trabajar, o bien llamamos a un objeto de esa clase desde otro objeto perteneciente a otra clase.

Vamos a coger el ejemplo de la clase `Coche`, propuesto en el segundo tema, y vamos a definir un método `main` donde podamos instanciar la clase:

Programa 4.3.1 La clase Coche instanciada.

```
class Coche {

    boolean estadoMotor = false;
    String color;
    String modelo;

    void arrancar(){
        if(estadoMotor == true)
           System.out.println("El coche ya está arrancado");
        else{
           estadoMotor=true;
           System.out.println("Coche arrancado");
        }
    }

    void detener(){
        if(estadoMotor == false)
           System.out.println("El coche ya está detenido");
        else{
           estadoMotor=false;
           System.out.println("Coche detenido");
        }
    }

    public static void main(String args[]){

       Coche ferrari = new Coche();
       ferrari.color="rojo";
       ferrari.modelo="Diablo";

       System.out.println("El modelo del coche es " + ferrari.modelo
                             + " y es de color " + ferrari.color);

       System.out.println("Intentando detener el coche...");
       ferrari.detener();

       System.out.println("Intentando arrancar el coche...");
       ferrari.arrancar();
    }
} // fin de la clase Coche.
```

Analicemos el código anterior:

- Tenemos tres métodos: `arrancar()`, `detener()` y `main()`. Todos los métodos hay que definirlos dentro de la clase, incluso el `main`.

- El `main` no es obligatorio, como se comentó en el tema 1; pero si no lo definimos aquí, deberemos hacerlo dentro de otra clase que llame a ésta.

- Dentro del `main`, hemos creado una instancia llamada `ferrari`. La clase `Coche` y, por tanto, su instancia `ferrari`, poseen una variable llamada `color`. Podemos acceder a ella utilizando un punto (" . "), con la notación *nombre_objeto.variable*:

  ```
  ferrari.color="rojo";
  ```

  Lo que hemos hecho es asignar a la variable `color` de nuestro objeto `ferrari` el valor `"rojo"`. Del mismo modo, asignaremos el valor `"Diablo"` a la variable `modelo`.

  La manera de comprobar que ahora esas variables tienen el valor que les hemos asignado es mostrándolas por pantalla. Para ello, utilizamos el método `System.out.println`:

  ```
  System.out.println("El modelo del coche es " + ferrari.modelo
  + " y es de color " + ferrari.color);
  ```

  No nos interesa la sintaxis del comando, pero vemos que, en la sentencia anterior, volvemos a acceder a los valores de las variables mediante el nombre de la instancia, un punto, y el nombre de la variable.

- Para acceder a los métodos de ferrari, el proceso es idéntico al usado para acceder a las variables:

  ```
  ferrari.detener();
  ```

  Simplemente invocamos al método mediante *nombre_objeto.metodo(parámetros)*. En nuestro caso, no hay parámetros dentro de los métodos[1], porque hemos definido `arrancar()` y `detener()` sin ellos, pero lo habitual es que haya que llamar a un método introduciéndole algún parámetro. Bien, ¿qué pasa cuando llamamos a `detener()`?. Si miramos el código de ese método, vemos que comprueba el valor de la variable booleana `estadoMotor`. Si el motor ya estaba parado (es decir, si está a `false`), simplemente nos informa de ello por pantalla. En caso contrario, pone la variable `estadoMotor` a `false` y nos informa de que ha detenido el coche.

En ocasiones, puede ocurrir que una variable perteneciente a un objeto mantenga a su vez a otro objeto. Por ejemplo, supongamos una clase `Rueda` con tres variables:

Programa 4.3.2 La clase Rueda.

```
class Rueda{
     String llanta;
     String cubierta;
     String modelo;
}
```

Y modificamos el código del programa 4.3.1 para incluir una variable de tipo `Rueda`:

[1] Los parámetros son valores que pasamos a un método para que pueda trabajar con ellos. Por ejemplo, podríamos definir un método `mostrarMensaje(String mensaje)` que nos sacara un mensaje por pantalla. El parámetro sería la cadena de texto (`String mensaje`) que queremos que nos muestre.

Programa 4.3.3 Clase Coche que incluye una clase Rueda.

```
class Coche {
    boolean estadoMotor = false;
    String color;
    String modelo;
    Rueda neumatico;

    void arrancar(){
             ......
    }
    void detener(){
             ......
    }
    public static void main(String args[]){
       Coche ferrari = new Coche();
       ferrari.neumatico = new Rueda();

       ferrari.color = "rojo";
       ferrari.modelo = "Diablo";
       System.out.println("El modelo del coche es " + ferrari.modelo
                             + " y es de color " + ferrari.color);
       ferrari.neumatico.cubierta = "Michelín";
       System.out.println("La marca de la cubierta del neumático es
                                    "+ferrari.neumatico.cubierta);
    }
}
```

Vemos que:

- Nuestra clase `Coche` ahora tiene ruedas. Para ello, se ha definido una variable `neumatico`, de tipo `Rueda`, que se ha instanciado dentro del método `main`. Como el objeto `neumatico` es un atributo más de la clase `Coche`, para instanciarla debemos acceder a ella a partir del objeto de esa clase, escribiendo `ferrari.neumatico = new Rueda()`. Si pusiéramos `neumatico = new Rueda()` nos daría un error de compilación.

- Dentro del método `main`, queremos acceder a la variable `cubierta` de las ruedas del coche. Así que accedemos escribiendo `ferrari.neumatico.cubierta`. Fácil, ¿no?. Para descender en la jerarquía de objetos, vamos separando los objetos con puntos.

  Aunque no se haya mencionado, en realidad es lo que estamos haciendo cuando mostramos un texto en pantalla: `System.out.println()`. Accedemos a la parte de salida (`out`) del sistema (`System`), para llamar a la función `println()`.

Veamos, a continuación, un ejemplo más útil en el que podamos practicar el acceso a los métodos y atributos de una clase del API. Vamos a utilizar la clase `GregorianCalendar` para mostrar por pantalla un calendario del mes actual como el siguiente, en el que se ve que el día actual está marcado con un asterisco:

```
Fecha Actual: 7/3/2004
Dom Lun Mar Mie Jue Vie Sáb
      1   2   3   4   5   6
  7*  8   9  10  11  12  13
 14  15  16  17  18  19  20
 21  22  23  24  25  26  27
 28  29  30  31
```

El código del programa que realiza esto se muestra en el listado siguiente:

Programa 4.3.4 Clase Calendario.

```
import java.util.*;

public class Calendario {
    public static void main(String args []){
        //Creamos un calendario con la fecha actual
        GregorianCalendar miCal = new GregorianCalendar();

        int hoy = miCal.get(Calendar.DAY_OF_MONTH);
        int mes = miCal.get(Calendar.MONTH); // Los meses comienzan con ENERO = 0
        int año = miCal.get(Calendar.YEAR);

        //Ajustamos el calendario para que se inicie en el primer día del mes.
        miCal.set(Calendar.DAY_OF_MONTH,1);
        int diaSemana = miCal.get(Calendar.DAY_OF_WEEK);

        //Mostramos el encabezado del calendario
        System.out.println("Fecha Actual: "+hoy+"/"+(mes+1)+"/"+año);
        System.out.println();
        System.out.println("Dom Lun Mar Mie Jue Vie Sáb");

        //Sangramos la primera línea del calendario
        for(int i = Calendar.SUNDAY; i < diaSemana; i++)
            System.out.print("    "); //Cuatro espacios

        do{
            int dia = miCal.get(Calendar.DAY_OF_MONTH);  //Mostramos el día
            if(dia < 10) System.out.print(" "); //Un espacio
            System.out.print(dia);

            if(dia == hoy)
                System.out.print("* ");  // marcamos el dia actual con un *
            else
                System.out.print("  ");

            if(diaSemana == Calendar.SATURDAY) //Inicia una nueva línea cada sábado
                System.out.println();

            miCal.add(Calendar.DAY_OF_MONTH,1);  //Avanza al día siguiente
            diaSemana = miCal.get(Calendar.DAY_OF_WEEK);
        }
        while(miCal.get(Calendar.MONTH) == mes);
            // El bucle continúa mientras miCal no llegue al día 1 del siguiente mes.

        if(diaSemana != Calendar.SUNDAY) //Mostramos el final de línea si es necesario
            System.out.println();
    }
}
```

Bueno, para analizar este ejemplo, es necesario tener delante el API con la clase `GregorianCalendar` cargada. Vamos a comentar únicamente los aspectos más destacados del código:

- Nuestra clase `Calendario` no tiene definidos atributos, y el único método es el `main`. Las clases que necesite las importará del paquete `java.lang` (el utilizado siempre sin tener que indicarlo explícitamente) y del paquete `java.util`, en el cual se encuentra, entre otras, `GregorianCalendar`.

- Lo primero que hacemos es crear una instancia de `GregorianCalendar`, `miCal`, invocando a su constructor. Nuestro objeto miCal tendrá disponibles, por tanto, todos los métodos y atributos que veamos enumerados en el API para esa clase. Además, el constructor inicializa a `miCal` con la fecha y hora actuales.

- A continuación, llamamos al método `get` de `GregorianCalendar` para obtener el día, mes y año. Los métodos `get` y `set` permiten, respectivamente, acceder y actualizar los valores de los atributos `DAY_OF_MONTH`, `MONTH` y `YEAR`, entre otros. Esto es útil porque, aunque la llamada que hicimos al constructor inicializó nuestro objeto `miCal` con la fecha actual, podríamos querer cambiarla para, por ejemplo, mostrar otro mes.

- Si consultamos el API, comprobaremos que la clase `GregorianCalendar` no tiene definidos los métodos `get` ni `set`. Entonces, ¿de dónde vienen?. He aquí un ejemplo típico de herencia. Si miramos la parte superior del API, veremos que se indica que `GregorianCalendar` hereda de la clase `Calendar`, la cual sí contiene esos métodos `get` y `set`. La clase `GregorianCalendar`, por tanto, tiene disponibles los métodos de su superclase.

- Otro aspecto interesante es que, cuando invocamos al método `set`, el parámetro que le introducimos es `Calendar.DAY_OF_MONTH`. Queremos obtener el atributo que contiene el día, de la superclase `Calendar`. Sin embargo, en vez de llamar a un objeto de tipo `Calendar`, estamos llamando a la propia clase `Calendar`. La respuesta a esto es que estamos accediendo a un atributo estático de la clase, que no necesita instanciarla previamente. Ahora mismo no nos preocuparemos de eso. Los atributos y métodos estáticos se verán en el capítulo 5.

- Resumiendo: al llamar al constructor de `GregorianCalendar`, éste, internamente, invoca al constructor de su superclase, el cual actualiza todos sus atributos con la fecha y hora actual. De ese modo, cuando llamemos a `Calendar.MONTH`, por ejemplo, nos devolverá el valor de ese atributo. No se profundizará en el resto del código, ya que resulta bastante fácil entenderlo con lo que llevamos explicado en el manual.

4.4. Métodos Constructores de una Clase

Cuando hablamos, genéricamente, del constructor de una clase, nos estamos refiriendo a un método especial que inicializa un objeto nuevo. Cuando, al principio del tema, instanciábamos un objeto con la palabra reservada `new`, seguida del nombre de la clase y un par de paréntesis, lo que estábamos haciendo era:

- Reservar memoria para el objeto.

- Inicializar todas las variables de instancia del objeto, ya sea con los valores que hayamos especificado, o con valores por defecto (`0`, o `null`).

- Invocar al método constructor.

Hasta ahora, nunca hemos definido explícitamente un constructor, así que Java define uno por defecto sin parámetros, que es el que hemos estado utilizando. Sin embargo, puede que queramos inicializar explícitamente unos parámetros del objeto. Entonces definiremos nuestros propios constructores. Para ello debemos recordar dos reglas:

- Un constructor no devuelve ningún tipo. Ni siquiera `void`.

- Un constructor tiene el mismo nombre que su clase.

Veamos el siguiente programa:

Programa 4.4.1 Creación de constructores: clases Empleado y GestorEmpleado.

```
import java.util.*;

public class GestorEmpleados
{
   public static void main(String[] args)
   {
      Empleado[] plantilla = new Empleado[3];
      plantilla[0] = new Empleado("Juan López", 700, 2001, 12, 15);
      plantilla[1] = new Empleado("Luis García", 800, 2000, 10, 1);
      plantilla[2] = new Empleado("Tsutomu Shimomura", 850, 2004, 3, 15);

      for (int i = 0; i < plantilla.length; i++)  // Subimos a todos el sueldo un 5%
         plantilla[i].aumentoSueldo(5);

      for (int i = 0; i < plantilla.length; i++)  // Mostramos la información
      {                                           // sobre los empleados
         Empleado e = plantilla[i];
         System.out.println("Nombre=" + e.getNombre()+ ", Sueldo=" + e.getSueldo()
            + ", Fecha de Contratación=" + e.getFechaContratacion());
      }
   }
}

class Empleado
{
    private String nombre;
    private double sueldo;
    private Date fechaContratacion;

    public Empleado(String n, double s, int año, int mes, int dia){
      nombre = n;
      sueldo = s;
      GregorianCalendar calendario = new GregorianCalendar(año, mes - 1,dia);
      fechaContratacion = calendario.getTime();
    }

   public String getNombre(){
      return nombre;
   }
   public double getSueldo(){
      return sueldo;
   }
   public Date getFechaContratacion(){
      return fechaContratacion;
   }
   public void aumentoSueldo(double porcentaje){
      double aumento = sueldo * porcentaje / 100;
      sueldo += aumento;
   }
}
```

El código anterior muestra dos clases, `GestorEmpleados` y `Empleado`[2]. Cada instancia de la clase `Empleado` representa a un determinado empleado de una empresa. La clase `GestorEmpleados` los crea, les puede aumentar el sueldo, y los muestra por pantalla. Para crear un nuevo empleado, `GestorEmpleados` llama al constructor de `Empleado`, pasándole como parámetros el nombre, el sueldo y la fecha de contratación de ese empleado.

4.5. Conversión mediante *Casting*

El *casting*[3] es una técnica que permite cambiar el valor de un objeto o tipo primitivo a otro tipo diferente. Esto es útil si tenemos, por ejemplo, un valor `int` (entero) y queremos convertirlo a `float` para poder incluirlo en operaciones con otros `float`. Podemos realizar conversiones entre tipos primitivos, entre objetos, y de tipos primitivos a objetos y viceversa.

Para realizar un ***casting* entre tipos primitivos**, debemos considerar primero si queremos convertir nuestro valor a un tipo *"más grande"* o *"más pequeño"*. En el primer caso, la conversión está implícita, sin necesidad de hacer nada:

Programa 4.5.1 *Casting* de un tipo más pequeño a uno más grande (implícito).

```
int x = 3;
float y = x;
```

En el segundo caso, es necesario realizar explícitamente el *casting*:

Programa 4.5.2 *Casting* de un tipo más pequeño a uno más grande (explícito).

```
class Casting{

    public static void main(String args[]){
       float x = 100f;
       float y = 3f;
       float v = x/y;
       int z = (int)v;

       System.out.println("Valor de x/y= "+v+",valor de x/y tras el casting= "+z);
    }
}
```

Vemos que el *casting* se produce al anteponer a la variable el tipo al cual queremos hacer la conversión, entre paréntesis.

Habrá casos, además, en los que será imprescindible hacer un casting para poder trabajar con determinados valores. Por ejemplo, si dividimos dos enteros, y queremos almacenarlos como un float, deberemos escribir:

```
float x = (float) 3/7;
```

En caso de no hacer el casting, comprobaríamos que en x se almacena 0, no el valor de la división.

Si queremos hacer **conversiones entre objetos**, debemos tener en cuenta que sólo es posible hacerlo si entre esos objetos existe una relación de herencia. Debido a que las subclases contien la

[2] Ahora no nos preocuparemos de que una clase sea pública y la otra privada, eso se verá más adelante.

[3] Nada que ver con Operación Triunfo ;-)

misma información que sus superclases, se puede utilizar una instancia de una subclase en cualquier lugar donde se espere una superclase, y el *casting* se hará automáticamente[4].

Por ejemplo, en un método cuyo argumento sea de clase `Object`, podremos pasar como parámetro cualquier objeto, puesto que todos los objetos son subclases de `Object`.

Respecto a la **conversión entre tipos primitivos y objetos**, en teoría, no se puede hacer. Sin embargo, existen una serie de clases en el paquete `java.lang`, "reflejo" de los tipos primitivos, que nos permiten hacer operaciones sobre tipos primitivos como si de clases se tratara. Mirando el programa siguiente, se observa que estamos creando una instancia de la clase `Integer`. En su constructor incluimos el valor del entero que queremos que contenga. A partir de ahí, podremos realizar operaciones con ese valor como si fuera un objeto. Y entre esas operaciones estará la de volver a convertir ese valor en un tipo primitivo. Eso es lo que hacemos cuando queremos mostrarlo por pantalla, en el `println()`, gracias al método `intValue()`.

Programa 4.5.3 Conversión entre tipos primitivos y objetos.

```
class ConversiónTiposObjetos
{
       public static void main(String args[])
       {
           Integer entero = new Integer(3);
           System.out.println("Valor de entero= "+entero.intValue());
       }
}
```

Podemos comprobar las clases que tenemos para trabajar con tipos primitivos consultando el API de Java.

4.6. Arrays

Aunque se podrían haber explicado los arrays en el capítulo 3, me ha parecido mejor hacerlo aquí, después de mostrar cómo se trabaja con objetos. Nótese que, si consultamos el API de Java, encontraremos una clase llamada `Array` (con mayúscula). En este apartado no nos referimos a esa clase, sino al concepto de *array* (con minúscula). En algunos libros en castellano sobre Java, para evitar confusiones, se refieren a los arrays como *"arreglos"*. Sin embargo, prefiero seguir la nomenclatura tradicional, ya que, en este apartado, sólo hablaremos de arrays, y no de la clase `Array`.

Un array es una colección de elementos ordenados, que pueden ser tipos primitivos u objetos. Cada elemento está ubicado en una posición del array, de forma que sea fácil acceder a cualquiera de esas posiciones, para agregar o eliminar un elemento. La ventaja de Java es que los arrays son también objetos, por lo que podremos tratarlos como tal.

Un array puede contener cualquier elemento, pero deben ser todos del mismo tipo. Es decir, no puedo mezclar números enteros (`int`) con objetos `String`, por ejemplo.

Para crear un array, lo primero es definir una variable que contenga los elementos. El tipo de esa variable será el tipo de los elementos que contendrá:

```
String cadenas[];    // Un array de objetos de la clase String
int enteros[];       // Un array de enteros
Coche automoviles[]; // Un array de objetos de la clase Coche
```

Nótense los corchetes junto al nombre de la variable. Son los que indican que esa variable es un array. Sin embargo, existe una forma más utilizada de definir los arrays, consistente en poner los corchetes a continuación del tipo:

[4]En realidad, esto es un efecto secundario del polimorfismo, característica que se verá en el capítulo 5.

```
String[] cadenas;
int[] enteros;
Coche[] automoviles;
```

Ya tenemos un array. Ahora hay que llenarlo con algo. Podemos definir su contenido directamente, utilizando llaves y separando los elementos con comas:

```
String[] cadenas= {"ordenador","discoDuro","teclado"};
int[] enteros={5,3,6,9};
Coche[] automoviles={fiesta,panda,tigra};// Suponemos
        // que anteriormente se han creado esas
        // tres instancias de la clase Coche.
```

con lo cual hemos definido implícitamente el número de posiciones que contiene el array (3,en el primer caso, 4 en el segundo, y 3 en el tercero). O bien, podemos inicializar el array con el número de elementos que podrá contener como máximo, y ya lo llenaremos más tarde:

```
int[] enteros =new int[20];
String[] cadenas =new String[10];
```

Lo que estamos haciendo es reservar memoria en el ordenador para que puedan almacenarse más tarde los datos. En un principio, hasta que no definamos los valores del array, se inicializarán con `0` para los arrays numéricos, `false` para los booleanos, `'\0'` para los arrays de caracteres, y `null` para los objetos.

Las posiciones de un array se numeran de 0 al tamaño definido para el array menos uno. Es decir, que si nuestro array tiene 20 elementos, estos estarán numerados de 0 a 19.

Si queremos añadir un elemento, basta con indicar la posición en la que queremos hacerlo (se llama *índice*), y el valor a añadir:

```
enteros[3] = 5; // Metemos 5 en la 4ª posición del array.
enteros[3] = enteros[1];
```

Recordemos que un array es un objeto. Como objeto, posee variables. Una de ellas, útil para saber la longitud del array, es `length`:

```
int longitud = enteros.length
```

Nos devolverá, en la variable `longitud`, la longitud del array. De este modo, evitaremos inicializar posiciones del array superiores a su longitud, ya que no existen y nos daría un error de compilación o de ejecución.

Si queremos copiar un array, o un número determinado de elementos de éste, en otro, podemos utilizar el método:

```
System.arraycopy(origen,indiceorigen,destino,indicedestino,numelementosacopiar)
```

Para terminar este apartado, mencionaremos los arrays multidimensionales. Al igual que un array es una colección ordenada de elementos, con una sola dimensión (es decir, un elemento detrás de otro, en posiciones consecutivas), existen arrays de varias dimensiones. Usando un símil matemático, un array unidimensional sería el equivalente a un vector, y un array de dos dimensiones, el equivalente a una matriz.

Para definir un array de dos dimensiones, declaramos un array de arrays:

```
int matriz[][] =new int[10][10];
```

Ya tenemos una matriz de 10*10. Para acceder a los distintos elementos:

```
matriz[3][2] = 1;
```

Por supuesto, podemos definir arrays de cualquier otra dimensión.

4.7. Trabajando con cadenas de caracteres

Las cadenas son secuencias de caracteres, como "Hola". Java proporciona una clase predefinida, `String`, de forma que toda cadena de caracteres encerrada entre comillas es una instancia de esa clase. Por ejemplo:

```
String cadena = ""; //Cadena vacía
String saludo = "Hola";
```

Nótese que, para las cadenas, no es necesario llamar al operador `new`.
Para concatenar cadenas, utilizamos el caracter "+". Por ejemplo:

```
String cadena1 = "Hola, "; //Nótese el espacio después de la coma
String saludo = cadena1 + "Mundo";
```

Cualquier objeto se puede convertir a String. Eso es lo que hacemos cuado utilizamos `System.out.println()` para mostrar números por pantalla:

```
int edad ="27";
String informe = "La edad es "+edad;
```

Si consultamos el API, comprobaremos que existen muchas funciones útiles, como la selección de subcadenas dentro de un `String`:

```
String saludo = "Hola";
String subcadena = saludo.substring(0,3);
```

En `subcadena` tendremos "Hol". Se cogen 3 caracteres a partir del primero, que se cuenta como 0 (igual que los arrays).
Para saber la longitud de una cadena:

```
int longitud = saludo.length();
```

Por último, para comparar dos cadenas de caracteres, no se utiliza el -=çomo en C, sino que la clase `String` proporciona el método adecuado:

```
String saludo = "Hola";
String saludo2= "Adios";
saludo.equals(saludo2);
```

devolverá `false`, ya que las cadenas son distintas. También puede hacerse:

```
String saludo = "Hola";
"Adios".equals(saludo);
```

Si no queremos que se tengan en cuenta mayúsculas y minúsculas:

```
String saludo = "Hola";
String saludo2= "hola";
saludo.equalsIgnoreCase(saludo2);
```

4.8. Resumen

En este tema hemos aprendido uno de los aspectos más importantes de Java, el trabajo con objetos. Los conceptos que deben recordarse son:

- Para instanciar una clase se utiliza la palabra reservada `new` seguida del constructor de la clase.
  Por ejemplo: `Integer entero = new Integer(2);`

- Una vez instanciada la clase, tendremos un objeto. Para acceder a sus métodos y atributos, utilizaremos una notación de puntos. Por ejemplo:
  `coche.arrancar()`, `coche.color="rojo"`, `System.out.println("hola")`;

- El **constructor** de una clase es un método especial que inicializa un objeto nuevo.

- Podemos realizar conversiones entre tipos primitivos, entre objetos, o entre objetos y tipos. Es lo que se denomina *casting*.

- Un array es una colección de elementos ordenados (tipos primitivos u objetos) referenciados por su índice, que es la posición en la que se encuentra cada elemento. Un array es un objeto de Java, por lo que se instancia y trata como tal.

- Todas las cadenas de caracteres son instancias de la clase `String`, que propociona métodos para manejarlas.

Capítulo 5

Manejo de Clases, Métodos y Variables

5.1. Introducción

En el primer tema se explicó cómo definir clases en Java. También hemos visto cómo utilizar y definir atributos y métodos. Sin embargo, no se ha profundizado en ninguno de esos aspectos. Eso es lo que haremos en este capítulo. Descubriremos que, según su ubicación en el código, existen varios tipos de métodos y variables con un tipo u otro de alcance. Estudiaremos el control de acceso a clases, métodos y variables. Explicaremos formalmente la definición de métodos y otros conceptos relacionados. Y veremos un concepto importante relacionado con la orientación a objetos de Java: el polimorfismo.

5.2. Tipos de Variables

En los tres capítulos anteriores se han propuesto ejemplos en los que las variables aparecían definidas en diferentes partes del código, pero no se han explicado las diferencias. Es ahora cuando vamos a catalogar los tipos de variables que podemos encontrarnos:

-Variable local: La que están dentro de la definición de un método. Sólo es válida para el código contenido en ese método.

-Variable de instancia: Se declaran fuera de los métodos. Generalmente, al principio de la definición de la clase. El valor de esa variable es válido para toda la instancia, así que la pueden utilizar todos los métodos de esa instancia.

-Variable de clase: Se declara al principio de la clase, como las de instancia. Sin embargo, es válida para para todas las instancias de esa clase. Para diferenciarla de las variables de instancia, se utiliza la palabra reservada `static`.

Un ejemplo que muestra los tres tipos de variables se muestra en el siguiente programa. En él, utilizamos la clase `Random` para obtener un número entero aleatorio que nos sirva para generar una fecha aleatoria.

Programa 5.2.1 Ejemplo en el que se muestran los tipos de variables.

```
import java.util.Random;

class Fecha{
    static int año = 2003; // Variable de clase
    int mes = 0;           // Variable de instancia

    void obtenerFechaAleatoria(){
        int dia=0; // Variable local.

        System.out.println("Antes de generar la fecha tenemos: ");
        System.out.println("dia: "+dia+", mes: "+mes+", año: "+año);

        Random aleatorio = new Random();
        dia = aleatorio.nextInt(30) + 1; //num. aleatorio entre 1 y 30
        mes = aleatorio.nextInt(12) + 1; //num. aleatorio entre 1 y 12

        System.out.println("La fecha generada aleatoriamente es: "+dia+"/"+mes+"/"+año);
    }

    public static void main(String args[]){

        Fecha miFecha = new Fecha();
        System.out.println("Primera llamada al objeto miFecha:");
        System.out.println("---------------------------------");

        miFecha.obtenerFechaAleatoria();
        System.out.println("Segunda llamada al objeto miFecha:");
        System.out.println("---------------------------------");

        miFecha.obtenerFechaAleatoria(); // dos llamadas al método del mismo objeto
        Fecha otraFecha = new Fecha();

        System.out.println("Llamada al objeto otraFecha:");
        System.out.println("---------------------------------");

        otraFecha.obtenerFechaAleatoria();
    }
}
```

Si compilamos y ejecutamos este código, una de las posibles salidas que podríamos tener es la siguiente:

```
Primera llamada al objeto miFecha:
---------------------------------
  Antes de generar la fecha tenemos:
  dia: 0, mes: 0, año: 2003
  La fecha generada aleatoriamente es: 7/9/2003
Segunda llamada al objeto miFecha:
---------------------------------
  Antes de generar la fecha tenemos:
  dia: 0, mes: 9, año: 2003
  La fecha generada aleatoriamente es: 25/5/2003
```

```
Llamada al objeto otraFecha:
----------------------------
  Antes de generar la fecha tenemos:
  dia: 0, mes: 0, año: 2003
  La fecha generada aleatoriamente es: 10/1/2003
```

En primer lugar, instanciamos un objeto de la clase `Fecha`, y le damos el nombre `miFecha`. Mostramos el valor de las variables antes de obtener ningún valor aleatorio y vemos que el día y el mes están inicializados a 0, y el año a 2003. A continuación, obtenemos una fecha aleatoria, llamando al método `obtenerFechaAleatoria()`, y nos devuelve la fecha 7/9/2003. Se ha asignado el valor 7 al día, y el 9 al mes. Si volvemos a llamar a la función **de ese mismo objeto**, o sea, `miFecha`, comprobaremos que se mantiene el valor de la variable de instancia (`mes:9`) y el de la de clase (`año:2003`), a pesar de tratarse de **dos llamadas distintas** a un método de ese objeto.

Si ahora creamos un nuevo objeto de la clase `Fecha`, `otraFecha`, e invocamos a su método para generar fechas, comprobaremos que la única variable que se mantiene del caso anterior es la del `año:2003`. Como el año es una variable de clase, su valor se mantiene para todos los objetos de esa clase.

5.3. Alcance de las Variables

Se denomina *alcance de una variable* a la zona del código donde se puede usar esa variable. En realidad, el concepto estaba implícito en la seccion anterior cuando definimos los tipos de variables y dónde se podía usar cada una.

¿Qué ocurre cuando llamamos a una variable?. En principio, Java busca una definición de esa variable en el ámbito actual en que se encuentra, que puede ser un bucle, por ejemplo. Si no la encuentra ahí, va subiendo, hasta llegar a la definición del método actual. Si tampoco aparece, es que no se trata de una variable local, así que sale del método y busca una variable de instancia o de clase con ese nombre. En caso de que tampoco la encuentre, sube a la superclase a buscarla.

Supongamos que tenemos una variable local y otra de instancia, ambas con el mismo nombre. Diremos que la variable local es de ámbito o alcance más corto. Entonces, la variable local *enmascara* a la de instancia.

Programa 5.3.1 Alcance de una variable.

```
class Alcance{

      int variable = 10;

      void imprimeVar()
      {
          int variable = 20;
          System.out.println("El valor de la variable es: "+variable);
      }

      public static void main(String args[])
      {
          Alcance instancia = new Alcance();
          instancia.imprimeVar();
      }
}
```

Cuando imprimamos el valor de la variable, nos devolverá 20, ya que la variable local oculta a la de instancia.

Del mismo modo, si definimos en una clase una variable que ya existe en su superclase, enmascararemos a la de la superclase. Todo esto puede dar lugar a errores y confusiones en el código. Por ello, lo más aconsejable es no utilizar nunca el mismo nombre para referirnos a variables distintas.

5.4. Modificadores

Los modificadores son palabras reservadas que permiten modificar la definición y el comportamiento de las clases, métodos y variables. Los más importantes son:

- Modificadores de control de acceso a una clase, método o variable: `public`, `protected` y `private`.

- El modificador `static` para crear métodos y variables de clase.

- El modificador `abstract` para crear clases, métodos y variables abstractas.

- El modificador `final` para indicar que finaliza la implementación de una clase, método o variable.

5.5. Control de Acceso. Tipos de protección

En el segundo tema se habló de la *encapsulación*, o proceso de ocultar la implementación interna de un objeto, permitiendo su comunicación con el exterior sólo a través de una interfaz definida.

Por otra parte, el *control de acceso* se refiere a la visibilidad de una clase, variable o método. Cuando un método o variable es visible a otras clases, éstas pueden llamarlos y modificarlos. Para evitar o limitar ese uso, debemos protegerlos. Esa protección se consigue mediante la **encapsulación**.

Java proporciona cuatro niveles de protección para clases y para variables y métodos de clase o instancia: de paquete, público, privado y protegido.

5.5.1. Protección *Friendly* o de Paquete[1]

Las clases, métodos y variables con protección de paquete son visibles a todas las demás clases del mismo paquete, pero no fuera de él (los paquetes se explicaron en el capítulo 2). Es la protección que se aplica por defecto cuando no se especifica ninguna otra, y es con la que hemos estado trabajando hasta ahora.

Supongamos que creamos dos archivos, cada uno conteniendo una clase, ambas pertenecientes a un paquete llamado cocina:

Programa 5.5.1 Archivo `masa.java` conteniendo la clase Masa.

```
package cocina;

class Masa{
    void añadir()
    {
       System.out.println("Ingrediente añadido");
    }
}
```

[1] La definición *"Friendly"* o *"de paquete"*, para referirnos al tipo de protección, no es un término formal, y no existe un consenso entre los diferentes autores sobre cómo denominarlo. Yo me he decantado por esos dos términos por ser los más comunes.

Programa 5.5.2 Archivo `tarta.java` conteniendo la clase Tarta.

```
package cocina;

class Tarta{
    public static void main(String args[])
    {
         Masa ingrediente = new Masa();
         ingrediente.añadir();
    }
}
```

Vemos que la clase `Tarta` instancia un objeto de la clase `Masa`, e invoca a un método de ésta. Entonces, debemos compilar primero `masa.java` para disponer desde el principio de la clase `Masa.class`. Así, cuando compilemos `tarta.java`, el compilador leerá el código, verá que se hace referencia a una clase llamada Masa, y podrá disponer de ella, ya que la hemos creado previamente.

Nótese que estamos trabajando con paquetes. Como se explicaba en el tema 2, es necesario crear una estructura de directorios equivalente al paquete que se ha definido en el código, y meter las clases allí, para que el compilador pueda encontrarlas. Por tanto, los pasos necesarios para que el código anterior funcione son:

- Meter en el mismo directorio los archivos `tarta.java` y `masa.java`.
- Compilar en primer lugar `masa.java`. Obtendremos una clase llamada `Masa.class`. Realmente, lo que obtenemos es la clase `Masa` perteneciente al paquete `cocina` o, en nomenclatura de Java, la clase `cocina.Masa`.
- Crear un subdirectorio llamado `cocina`.
- Mover la clase `Masa.class` al subdirectorio `cocina`.
- Compilar `tarta.java`. El compilador buscará la clase `cocina.Masa`. Como en el *classpath* estará definida, entre otras rutas, nuestro directorio de trabajo (aparece como un punto, "`.`"), se buscará ahí el paquete `cocina` y, dentro de éste, la clase `Masa`.
- Si la compilación es correcta, tendremos una clase `Tarta.class`. Realmente, como también pertenece al paquete `cocina` (nótese en el programa 5.5.2 que hemos incluido la línea `package cocina`) tendremos que moverla a ese subdirectorio.
- Para ejecutarlo, debemos escribir en la línea de comandos, desde nuestro directorio de trabajo: `java cocina.Tarta`

Retomando el tema de la protección, fijémonos en que, al definir las dos clases, no se ha especificado ningún tipo de protección delante de la palabra reservada `class`. Por lo tanto, esas clases serán, por defecto, *Friendly*. De este modo, la clase `Tarta` ha podido instanciar a la clase `Masa`, porque ésta última tiene protección *Friendly* (la tienen las dos clases, pero nos preocupa la de la clase que es accedida, no la de la que accede), y ambas pertenecen al mismo paquete.

Alguien podría preguntarse por qué es posible acceder a la clase sin especificarlo con un `import`. Como ya se ha comentado, una clase *Friendly* sólo es visible para las clases de su mismo paquete. Luego las clases de otros paquetes no pueden utilizarlas, luego no podrán importarlas. Y las clases pertenecientes al mismo paquete no necesitan un `import` para invocarse entre ellas.

Un último apunte: Supongamos que compiláramos los dos archivos anteriores (programas 5.5.1 y 5.5.2) omitiendo en el código la línea en la que se definen como pertenecientes al paquete `cocina`. Compilarán sin problemas, y obtendremos dos clases *Friendly*. Sin moverlas a ningún subdirectorio, ejecutamos la clase `Tarta`. Funciona. ¿Cómo puede ser que una clase *Friendly* llame a otra, si no se ha especificado en ningún sitio que pertenezcan al mismo paquete?. Bien, cuando dos o más clases están en el mismo directorio y no tienen definido explícitamente un nombre de paquete, Java las trata como pertenecientes a un mismo paquete *"virtual"*.

5.5.2. Protección Pública

Las clases, métodos y variables con protección pública son visibles a todas las demás clases, dentro o fuera de la clase o paquete actual. Para especificar que una clase (o método o variable) es pública, se utiliza la palabra reservada `public`. Por ejemplo, retormando los dos ejemplos anteriores:

Programa 5.5.3 Archivo `MasaPublica.java` conteniendo la clase `MasaPublica`.

```
package ingredientes;

public class MasaPublica{

       public void añadir(){
           System.out.println("Ingrediente añadido");
       }
}
```

Programa 5.5.4 Archivo `tarta2.java` conteniendo la clase `Tarta`.

```
package postres;

import ingredientes.*;

class Tarta{

    public static void main(String args[]){
        MasaPublica ingrediente = new MasaPublica();
        ingrediente.añadir();
    }
}
```

Lo que estamos haciendo en estos códigos es crear dos clases pertenecientes a dos paquetes distintos: `postres` e `ingredientes`. La clase `ingredientes.MasaPublica` es pública, y es accedida por `postres.Tarta`, que es *Friendly*.

Los pasos a seguir para compilar y ejecutar el código anterior son:

- Compilar el archivo `MasaPublica.java`. Obtendremos una clase `MasaPublica.class` perteneciente al paquete `ingredientes` (es decir, la clase `ingredientes.MasaPublica`). Así que creamos un subdirectorio `ingredientes`, y movemos la clase ahí.

- Compilamos el archivo `tarta2.java`. Obtenemos una clase `Tarta.class`, que movemos a un subdirectorio `postres`.

- Ejecutamos `java postres.Tarta`, y vemos cómo, desde el paquete `postres`, la clase `Tarta` llama a la clase pública `MasaPublica`, contenida en el paquete `ingredientes`.

Algunos comentarios sobre el código:

- En el programa 5.5.4, al igual que pasaba en los programas 5.5.1 y 5.5.2, el nombre del fichero java es distinto del de la clase que contiene. Sin embargo, en el programa 5.5.3, el nombre del fichero y de la clase contenida coinciden. Es importante recordar que, **cuando tengamos más de una definición de clase en el mismo archivo java, sólamente una de ellas podrá ser pública**, y **el archivo java deberá llamarse igual que esa clase**.

- No sólo la clase `MasaPublica` debe ser de tipo `public`. También su método, `añadir()`, deber ser público para que pueda ser accedido desde fuera. De este modo, si tenemos más métodos, definiremos como `public` sólo aquellos a los que queramos permitir su acceso.

- Nótese que ahora, al pertenecer cada clase a un paquete diferente, la clase `Tarta` necesita especificar un `import` al principio, para cargar todas las clases (en este caso, sólo una) del paquete `ingredientes`.

- Nótese también que el método `main` siempre se define como un método público.

5.5.3. Protección Privada

Los métodos y variables con protección privada son accesibles únicamente por métodos de la misma clase. Para especificar que el método o variable es privado, se antepone la palabra `private`.

La protección privada es un buen método de encapsulación. Podemos definir una clase con varios métodos que realicen muchas funciones internas, todos ellos privados, y sólo uno accesible desde el exterior, para recoger datos y devolver resultados.

Evidentemente, la aplicación de la protección privada en una definición de clase, habitualmente no tendría sentido. ¿Para qué ibamos a querer restringir totalmente el acceso a esa clase?. Sin embargo, existe un tipo especial de clases que sí pueden ser privadas. Son las **clases internas**.

5.5.3.1. Clases Internas

Una clase interna es aquella que está definida dentro de otra clase. Esto permite que:

- Un objeto de una clase interna pueda acceder a la implementación del objeto que lo creó, incluyendo los datos que, de otra forma, serían privados.

- Las clases internas puedan ser ocultadas a otras clases del mismo paquete.

Por ejemplo:

Programa 5.5.5 Clase interna.

```
class ClasePrincipal{

   private atributo1;
   public metodo(){ }

   private class ClaseInterna{
       //desde aquí podemos acceder a atributo1
   }
}
```

5.5.4. Protección Protegida

La protección protegida proporciona cierto nivel de visibilidad menos restrictiva que la privada. Mediante la protección protegida, los métodos y variables de una clase estarán accesibles a todas las clases dentro del paquete y a sus subclases, aunque estén fuera de él. Para especificar que el método o variable es protegido, se antepone la palabra `protected`.

¿Qué utilidad tiene este tipo de protección?. Como se explicó en el tema 2, una subclase puede concretar y especificar más el comportamiento de su superclase. Puede que tengamos un *método B* en la subclase que necesite superponer un *método A* de su superclase. Sin embargo, no queremos que ese *método A* sea visible para el resto de clases. Ésto sólo será posible hacerlo mediante la protección protegida. Ahora mismo, esto no es importante. Más adelante se verá la superposición de métodos y se entenderá mejor la utilidad de `protected`.

5.6. Finalización de clases, métodos y variables

La finalización permite bloquear la implementación de una clase, variable o método. Se especifica utilizando la palabra reservada `final`, generalmente a continuación de un modificador de protección. Explicamos cada caso por separado.

5.6.1. Finalización de variable

Es el caso más sencillo. Ya se explicó en el tema 3. Consiste en anteponer la palabra `final` al nombre de una variable, convirtiéndola así en una constante:

```
final int a = 6;
```

5.6.2. Finalización de método

Evita que un método pueda ser superpuesto. Aunque se explicará más adelante detalladamente, la superposición de métodos significa que la definición de un método puede ser redefinida en una subclase.

```
public final void metodoInmutable(){ }
```

La utilización de métodos finales acelera la ejecución del programa, porque el compilador no tiene que buscar definiciones de ese método en las subclases.

5.6.3. Finalización de clase

Evita que una clase tenga subclases.

```
public final class ClaseFinal{ }
```

El API de Java tiene clases definidas como finales, para que no puedan extenderse. Sin embargo, salvo por razones de eficiencia, es poco probable que finalicemos nuestras clases, ya que puede interesarnos ampliar su funcionalidad mediante subclases.

5.7. Métodos

Veíamos en el tema 2 que los métodos nos definen la funcionalidad de un objeto. Llevamos cuatro capítulos poniendo ejemplos en los que aparecen métodos. Sin embargo, eran métodos muy simples que no podíam obtener ni devolver datos. En esta sección vamos a explicar formalmente cómo definirlos, los tipos de métodos existentes, y las operaciones que podemos realizar con ellos.

Un método consta de las siguientes partes:

- Uno o varios modificadores (`public`, `static`, `final`, etc.).
- El objeto o tipo primitivo que devuelve.
- El nombre del método.
- Un conjunto de parámetros.
- La palabra reservada throws, que permite arrojar excepciones, y que se verá en otro tema más adelante.
- El cuerpo del método.

El conjunto formado por el tipo de retorno, el nombre del método y el conjunto de parámetros constituyen lo que se denomina *firma del método*.

Un ejemplo de método:

Programa 5.7.1 Un método de ejemplo.

```
public int saludo(String nombre){

     int hayError=0; //Si 0,no error

     if(nombre==null)
        return (-1);

     else
     {
        System.out.println("Hola, "+nombre+", la prueba ha sido correcta");
        return 0;
     }
}
```

En el listado anterior, `public` es el modificador, `int` el tipo primitivo que devuelve el método, `saludo` el nombre del método, `nombre` el parámetro que se introduce, y todo lo que va entre las llaves del método es el *cuerpo del método*. Es un método muy sencillo, que se limita a mostrar por pantalla un saludo a la persona cuyo nombre hemos introducido como parámetro. Los valores que devuelve son: 0, si no ha habido ningún problema, o -1 en caso de que no se haya introducido ningún nombre. Nótese que los valores se devuelven mediante la palabra reservada `return`. Evidentemente, el tipo u objeto devuelto por `return` debe coincidir con el tipo de valor de vuelta especificado en el método (`int`, en este ejemplo) o el compilador nos dará un error.

Cuando no se devuelva nada en un método, debe especificarse el tipo de vuelta como `void`. Por ejemplo: `public static void main(String args[])`.

Un inciso: si no se introduce un nombre, el objeto `String nombre` estará vacío. Para comprobar si un objeto está vacío, utilizamos la palabra reservada `null`. Todo objeto que no esté inicializado estará a `null`.

Cuando se pasan parámetros, debe tenerse en cuenta que:

- Los tipos primitivos se pasan por valor. Es decir, que se crea una copia dentro del método, y es con esa copia con la que se trabaja. El tipo original mantiene el valor que tenía.

- Los objetos se pasan por referencia. Es decir, que lo que estamos pasando es una referencia (un apuntador) al objeto, de forma que todo lo que hagamos a ese objeto dentro del método afectará al objeto original. Concretamente, esto se refiere a los arrays y a los objetos que contienen.

La mejor forma de verlo es con un ejemplo:

Programa 5.7.2 Paso de argumentos a un método.

```
class UsoMetodos{

   public int modifica(String nombre, int num, String frases[]){
        int error = 0; // Si 0,no hay error.

        if (nombre ==null)
             return (-1);
        else{
            System.out.println("Entrando en el método...");
            System.out.println(-----------------------);
            System.out.println("Valor del entero: "+ num);
            System.out.println("Modificamos entero...");
            num=10;
            System.out.println("Entero modificado: "+ num);
            System.out.println("\n");

            System.out.println("Nombre introducido: "+ nombre);
            System.out.println("Modificamos el nombre...");
            nombre = "Juan Ángel";

            System.out.println("Nombre modificado: "+nombre); System.out.println("\n");
            System.out.println("Valor del array de frases= "+frases[0]+","+frases[1]);

            System.out.println("Modificamos las frases...");
            frases[0]="primera frase cambiada";
            frases[1]="segunda frase cambiada";

            System.out.println("Valor del array de frases cambiadas="+frases[0]+","+frases[1]);
            return 0;
        }
    }

   public static void main(String args[]){

            int hayError = 0;
            int numero = 2;

            String miNombre = "Paco";
            String sentencias[] = new String[2];

            sentencias[0] = "frase 0";
            sentencias[1] = "frase 1";

            UsoMetodos usoMetodo = new UsoMetodos();
            hayError = usoMetodo.modifica(miNombre,numero,sentencias);

            if(hayError == 0){
                System.out.println("\n");
                System.out.println("Saliendo del método...");
                System.out.println("----------------------");
                System.out.println("Numero devuelto: "+numero);
                System.out.println("Nombre introducido: "+miNombre);
                System.out.println("Frases: "+sentencias[0]+", "+sentencias[1]);
            }
            else
                System.out.println("Ha habido un error al ejecutar el método");
   }
}
```

Lo que estamos haciendo en el método main de este ejemplo es asignar unos valores a un `String`, un `int` y un array de Strings de longitud 2, y pasarlos como parámetros a una función. En la función se modifican esos valores. Posteriormente, volvemos al programa principal para comprobar si esas modificaciones se mantienen.

Al compilar y ejecutar esa clase, obtenemos la siguiente salida:

```
Entrando en el método...
------------------------
Valor del entero: 2
Modificamos entero...
Entero modificado: 10

Nombre introducido: Paco
Modificamos el nombre...
Nombre modificado: Juan Angel

Valor del array de frases= frase 0, frase 1
Modificamos las frases...
Valor del array de frases cambiadas= primera frase cambiada,
                                     segunda frase cambiada

Saliendo del método...
----------------------
Numero devuelto: 2
Nombre introducido: Paco
Frases: primera frase cambiada, segunda frase cambiada
```

Observamos que, dentro del método, se han cambiado los valores de los parámetros. Sin embargo, al volver a la función principal, sólo se mantienen los cambios hechos en el array de Strings. Como se comentó anteriormente, los arrays se pasan por referencia, no por valor.

¿Cómo podríamos devolver más de un valor en un método?. Bien, podemos hacer lo mismo que en el ejemplo anterior, introduciento un array como parámetro, y modificar sus valores. Sin embargo, es más elegante, y práctico, hacer lo mismo, pero en el objeto que se devuelve:

```
int[] devuelveArrayEnteros(int parametro){ }
String[] devuelveArrayStrings(int parametro){ }
```

También podemos utilizar *objetos contenedores*, que nos permiten devolver más de un objeto, al igual que hace el array, pero añadiendo funciones más avanzadas. Por ejemplo, la clase `Vector`, la clase `ArrayList`, la clase `Hashtable`... Te remito al API del j2sdk para aprender a utilizarlas.

5.8. Pasando argumentos desde la línea de comandos

En muchas ocasiones necesitaremos que el usuario, al invocar a nuestro programa desde la línea de comandos, introduzca una serie de parámetros. Para ello, se escribe el nombre del programa seguido de los argumentos separados por espacios. La forma en que el programa recibe esos parámetros es a través del argumento de la función `main`, como un array de Strings. Recordemos el aspecto de `main`:

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

El nombre del argumento, por costumbre, suele ser `args`, pero realmente podemos llamarlo como queramos, siempre y cuando lo definamos como un array de cadenas. En `args[0]` tendremos el primer argumento, en `args[1]` el segundo, y así sucesivamente. Un ejemplo se muestra en el programa ámetroslineacomandosparámetroslineacomandos.

Programa 5.8.1 Entrada de parámetros desde la línea de comandos.

```
class Argumentos{

      public static void main(String args[]){
         int i = 0;
         for(i=0;i<args.length;i++)
               System.out.println("argumento "+i+": "+args[i]);
      }
}
```

Este programa nos mostrará todos los argumentos que introduzcamos. Nótese que, en el bucle `for`, podemos averiguar el número de parámetros consultando el atributo `length` del array. ¡Cuidado!, es un atributo propio del array, no se trata de la función `length()` de la clase String.

Una entrada que podríamos hacer desde la línea de comandos, por ejemplo, sería:

```
java~Argumentos~uno~dos~tres~cuatro
```

Otro detalle a comentar es que, si queremos comparar los parámetros introducidos con determinados valores, no debemos usar el "`==`", sino que utilizaremos el método `equals` de la clase `String`:

```
if(args[0].equals("-version"))...
```

5.9. Métodos de clase e instancia

Al principio de este tema se explicaron las variables de clase y de instancia. El concepto de método de clase e instancia es similar y la diferencia entre ellos análoga a la de las variables. Los métodos de instancia son los métodos con que hemos trabajado hasta ahora: los que se utilizan dentro de una instancia y pueden ser accedidos desde otras clases (depediendo del modificador -`public`, `private`, etc-). Los métodos de clase están disponibles para cualquier instancia de esa clase y para otras clases, **independientemente de si existe, o no, una instancia de esa clase**.

¿Qué quiere decir eso?. Existen clases en Java que tienen definidos métodos de clase, a los que podemos acceder sin tener que instanciarla primero. Basta con utilizar el nombre de la propia clase. Por ejemplo, todos los métodos de la clase `Math` son estáticos (puedes comprobarlo consultando el API). Entre ellos, hay uno que permite hallar el coseno de un ángulo. Si quisiéramos utilizarlo, podríamos hacer:

```
double~coseno~=~Math.cos(Math.PI);
```

De paso, este ejemplo nos sirve para ilustrar el modo de acceder a variables de clase. La variable `PI` está definida, dentro de la clase `Math`, como `public static final double PI`. Accedemos a ella como hemos hecho con el método `cos()`, poniendo el nombre de la clase, un punto, y la variable.

Para definir nuestros propios métodos de clase, al igual que hacíamos con las variables, utilizaremos la palabra reservada `static`. Por ejemplo:

```
public~static~int~calculaDerivada(String~ecuacion){~
~~~~~~~~~~~~~~~~~~~~~...~
}
```

5.10. Análisis del método main

Siempre que creamos una aplicación en Java, como hemos visto en todos los ejemplos hasta ahora, necesitamos definir, como mínimo, una clase y, dentro de ésta, un método `main`, que será lo primero que se llama al ejecutar la aplicación[2].

[2]Esto no es totalmente cierto. Los applets no necesitan un método main para ejecutarse.

Recuérdese, del primer tema, que, para crear una aplicación, podemos tener tantas clases como deseemos con métodos normales y corrientes. Sin embargo, debe existir una, que contendrá el método `main`, que sea la que lance la aplicación. Desde ella se invocará a las demás clases de nuestro programa.

Con todo lo explicado hasta ahora, ya estamos en condiciones de entender por qué definimos el método `main` tal y como lo hacemos. Recordemos la firma y modificadores de `main`:

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

Vamos por partes:

public: Evidentemente, el método `main` debe estar accesible desde fuera de la clase, para poder invocarlo y *"arrancar"* la aplicación.

static: Es un método de clase. Como se comentó anteriormente, un método de clase está disponible independientemente de que esté, o no, instanciada esa clase. En el caso del `main` no existe previamente una instancia de la clase en la que está definido. Es por ello que, siempre que necesitemos un objeto de la propia clase, debemos instanciarlo dentro del propio `main`, como hemos venido haciendo hasta ahora en varios ejemplos (por ejemplo, el del programa 5.7.2).

void: El método `main` no devuelve nada.

main: Siempre se llama así. El paso de parámetros mediante un array de Strings ya se explicó en la sección anterior.

5.11. Polimorfismo

Una de las características más importantes de Java, con respecto a su orientación a objetos, junto con la herencia, es el *polimorfismo*. El polimorfismo es la capacidad de utilizar un objeto de una subclase como si fuera un objeto de la clase base. Esto nos permite escribir código que desconoce parcialmente las características de los objetos con los que trabaja y que es más fácil de escribir y entender. Se dice entonces que ese código está *desacoplado*.

La verdad es que el párrafo anterior, tal y como lo he escrito, es intragable. Se entiende mejor con un ejemplo. Supongamos que tenemos una clase llamada `Figura`, la cual extendemos mediante una subclase llamada `Circulo`. Si tenemos definido un método, en cualquier otro sitio, cuyo parámetro sea de tipo `Figura`, podremos pasarle una clase `Círculo`, y lo aceptará perfectamente.

Vamos a ver ese ejemplo en el programa siguiente. De paso, lo utilizaremos para repasar conceptos estudiados previamente.

Programa 5.11.1 Ejemplo de polimorfismo.

```
class Figura{
      private double area = 0;

      public void setArea(double cantidad){
              area = cantidad;
      }

      public double getArea(){
             return area;
      }
}

public class Circulo extends Figura{
       double getPerimetro(){
              return 2 {*} Math.PI {*} Math.sqrt(getArea()/Math.PI);
       }
}

class Polimorfismo{
       private double obtenerArea(Figura figura){
              return figura.getArea();
       }

       public static void main(String args[]){
              double areaCirculo = 0;
              double perimCirculo = 3;
              double areaIntroducida = 0;

              Polimorfismo poli = new Polimorfismo();
              Circulo c =new Circulo();

              areaIntroducida = (new Double(args[0])).doubleValue();
              c.setArea(areaIntroducida);

              areaCirculo = poli.obtenerArea(c);
              perimCirculo = c.getPerimetro();

              System.out.println("El área del círculo es:"+ areaCirculo);
              System.out.println("El perímetro del círculo es: "+ perimCirculo);
       }
}
```

El código anterior es un ejemplo medianamente realista. En primer lugar, definimos una clase `Figura`. Su único atributo es el área de la figura. Vemos que se trata de un atributo privado, por lo que no podremos acceder directamente a él. Para modificarlo u obtener su valor debemos utilizar los métodos públicos `setArea()` y `getArea()`, respectivamente.

A continuación, definimos la clase `Circulo`, que hereda de la anterior. Esta clase, además de poseer los métodos y atributos de su superclase, gracias a la herencia, especifica un método propio que permite obtener el perímetro del círculo. Si lo observamos detenidamente, comprobaremos que calcula el perímetro utilizando métodos de clase, disponibles en la clase `Math`, y que obtiene el área de la figura del método `getArea()`, perteneciente a su superclase, y que también ha heredado.

Por último, se define la clase `Polimorfismo`. En ella tenemos dos métodos: `obtenerArea()`,

cuyo parámetro es un objeto de la clase `Figura`, y `main()`. En `main()` instanciamos la propia clase `Polimorfismo` (para poder invocar más tarde a `obtenerArea()`) y, a continuación, creamos una instancia de la clase `Circulo`. Utilizaremos esa instancia para introducir el valor del área que nos habrán dado desde la línea de comandos, gracias al método `setArea()`.

Resumiento: en este momento tenemos un objeto `c`, instancia de la clase `Circulo`, que contiene un valor en su atributo `area`, heredado de su superclase `Figura`. A continuación, llamamos al método `obtenerArea()`, de nuestra clase `Polimorfismo`. Recordemos que ese método acepta como parámetro objetos de la clase `Figura`. Sin embargo, lo que le pasamos es la instancia de la clase `Circulo`...¡y la acepta!. Nuestro método llamará al método `getArea()` de la clase `Circulo`. Como `Circulo` lo posee, gracias a la herencia, no dará ningún tipo de problemas. En esto consiste el polimorfismo: escribir código (no sólo métodos) que trabaje con una clase y todas sus subclases.

La última acción que realizamos es obtener el perímetro llamando directamente al método de la instancia de la clase `Circulo`, mostrándolo por pantalla.

Aunque no tenga relación con el polimorfismo, puede aprovecharse el ejemplo anterior para explicar lo que es la *protección de variables de instancia* y los *métodos de acceso*. Si nos fijamos en la clase `Figura`, vemos que la variable de instancia `area` es privada. El único modo de acceder a ella o modificar su valor es mediante los métodos publicos `get()` y `set()`. Esto es una buena costumbre en programación orientada a objetos. De este modo, nos aseguramos de que todos los objetos externos que accedan a esa variable no lo harán directamente, sino a través de un *"entorno controlado"* que nos proporcionan esos métodos de acceso. Así, si quisiéramos hacer comprobaciones previas al acceso del valor de la variable (como pudiera ser la identidad del usuario, por ejemplo, para ver si está autorizado a acceder a esa variable) las incuiríamos en el cuerpo de esos métodos.

5.12. This y Super

Relacionado con la herencia y el alcance de las variables, existen dos palabras reservadas que pueden sernos de utilidad en algunas ocasiones. Son `this` y `super`.

`this` nos permite, dentro del cuerpo de una función contenida en un objeto, referirnos a las variables de instancia de ese objeto. También nos permite pasar el objeto actual como un argumento a otro método. En general, cada vez que necesitemos referirnos al objeto actual, utilizaremos `this`.

Por ejemplo, supongamos que estamos dentro de un método, y en la clase a la cual pertenece ese método existe una variable de instancia llamada `x`. Podemos referirnos a ella escribiendo `this.x`. En principio, esto parece una tontería, porque llevamos cuatro capítulos llamando a las variables de instancia sin utilizar `this`. Pues sí, es una tontería porque, en realidad, la palabra `this` está implícita en la llamada a la variable. Por tanto, podremos seguirla invocando escribiendo únicamente `x`.

Sin embargo, hay ocasiones en las que puede ser útil. Recordemos el ejemplo del programa 5.3.1. Teníamos un conflicto con dos variables que tenían el mismo nombre. Cuando imprimíamos por pantalla su valor, desde el método `imprimeVal()`, nos devolvía el valor de la variable local, que ocultaba al de la variable de instancia. Si deseáramos ver el valor de la variable de instancia, podríamos redefinir la sentencia que muestra la variable del siguiente modo:

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

Y nos devolvería 10, en lugar de 20 como nos ocurría antes.

El otro uso de this es pasar el objeto actual como parámetro de otro método. Por ejemplo:

```
miMetodo(this);
```

`super` es una palabra reservada similar a `this`. Mientras que this se refiere al objeto actual, super se refiere a la superclase del objeto en el que nos encontremos. Por ejemplo, `super.miMetodo()` llamará a ese método de la superclase. De nuevo, no parece servir para mucho. Sin embargo, en este caso será muy útil, como veremos cuando hablemos de la superposición de métodos.

5.13. Sobrecarga de Métodos

La sobrecarga de métodos consiste en crear varios métodos con el mismo nombre, pero con distintas firmas y definiciones. Por ejemplo, supongamos que tenemos un programa en el que necesitamos introducir unas coordenadas de una zona. Sin embargo, no sabemos a priori la forma que tendrá esa zona. Puede que sea una superficie rectangular, un círculo, o una superficie delimitada por líneas trazadas entre cuatro puntos. Así que definimos un método, sobrecargándolo, como se muestra a continuación:

```
void coordenadas(float alto, float ancho){...}
void coordenadas(float radio){...}
void coordenadas(float x1, float y1, float x2, float y2){...}
```

Entonces, recogeremos los datos que introduzca el usuario del programa, y llamaremos al método coordenadas[3] . ¿Cómo se las arregla Java para saber cuál de las tres definiciones del método tiene que utilizar?. Por el número y tipo de los argumentos. Por ejemplo, si invocamos al método de esta forma:

```
coordenadas(0.3);
```

Java sabrá que se refiere a la segunda definición del método, y que `0.3` es el radio de un círculo.

5.14. Superposición de Métodos

Hace mucho, mucho tiempo, algo así como al principio de este tema, se explicó en qué consistía el alcance de una variable. Decíamos que puede existir una variable con el mismo nombre en una clase y en su superclase. Lo mismo ocurre con los métodos. Puede que tengamos una clase con un método que realiza una función determinada. Si definimos una subclase, heredaremos ese método. Pero puede que no nos sirva tal y como está, o que queramos ampliar su funcionalidad. Lo que haremos, entonces, será definir un método en la subclase con la misma firma que el método original. Por ejemplo, veamos el programa siguiente:

Programa 5.14.1 Superposición de métodos.

```
class Clase{
      String ordenador1 = "PC Compatible";
      String ordenador2 = "Macintosh";
      void imprimeVariables(){
         System.out.println("Ordenador1: "+ordenador1);
         System.out.println("Ordenador2: "+ordenador2);
      }
}
class SubClase extends Clase{
      String ordenadorNuevo = "Sun Sparc";
      void imprimeVariables(){
           System.out.println("Ordenador1: "+ordenador1);
           System.out.println("Ordenador2: "+ordenador2);
           System.out.println("Ordenador Nuevo: "+ ordenadorNuevo);
      }
      public static void main(String args[]){
           SubClase subclase = new SubClase();
           subclase.imprimeVariables();
      }
}
```

[3]Sí, ya sé que podríamos definir una superficie rectangular con cuatro puntos, y nos ahorraríamos el método que utiliza la altura y la anchura, pero esto es un ejemplo, y hoy no estoy muy inspirado :-)

Java, al igual que hacía con las variables, buscará primero el método en la clase, y, si no lo encuentra, subirá a buscarlo en la superclase. En este caso, el método de la clase `SubClase` enmascarará al de su superclase, y será el que se ejecute. En eso consiste la superposición de métodos.

El problema en el ejemplo anterior es que, para superponer un método que sólo añade un `println` al método original, tenemos que reescribirlo entero. ¿No habría una forma de superponer el método *"cogiendo"* el método de la superclase, y añadiendo sólo la funcionalidad que queremos ampliar?. Sí. Menos mal, porque reescribir un método de 500 líneas de código puede ser muy aburrido. Miremos el programa siguiente:

Programa 5.14.2 Superponiendo un método sin reescribir todo el código.

```
class SubClase2 extends Clase{

      String ordenadorNuevo = "Sun Sparc";

      void imprimeVariables(){
           super.imprimeVariables();
           System.out.println("Ordenador Nuevo: "+ordenadorNuevo);
      }

      public static void main(String args[]){
           SubClase2 subclase2 =new SubClase2();
           subclase2.imprimeVariables();
      }
}
```

Ahora, el método `imprimeVariables()` aprovecha el contenido del método original de la superclase, y añade su propia funcionalidad (el `println()`). Como se advirtió en su momento, esto se consigue con la palabra reservada `super`. Con ella nos estamos refiriendo a la superclase, así que, con `super.imprimeVariables()`, invocamos al método de ésta.

5.15. Sobrecarga de constructores

Un constructor es un método. Por tanto, puede ser sobrecargado. Ello puede resultarnos útil si necesitamos inicializar un objeto con diversos tipos de parámetros, en función de los valores de entrada. Se realiza exactamente igual que la sobrecarga de métodos explicada anteriormente.

Además, es posible aprovechar el código escrito para un constructor en la definición de otro. Si tenemos dos constructores, y uno de ellos extiende la funcionalidad del otro (es decir, que va a hacer lo mismo que el otro, y algo más), utilizaremos en el segundo la palabra reservada `this`, refiriéndonos al objeto actual, y utilizando los parámetros del constructor que queremos extender:

```
this(argumento_1, argumento_2, ...);
```

Y, a continuación, añadiremos la nueva funcionalidad que queremos para ese constructor.

Un detalle importante: Cuando no definimos explícitamente un constructor para una clase, disponemos del constructor por defecto, sin parámetros. Sin embargo, en el momento que definamos un constructor con parámetros, perdemos el anterior. Si queremos tener también un constructor sin parámetros (por ejemplo, que inicialice parámetros por defecto en la clase) debemos programarlo nosotros.

5.16. Superposición de constructores

La superposición de métodos consistía en definir un método en una clase que enmascarase al método del mismo nombre perteneciente a la superclase. Como los constructores deben tener el mismo nombre que la clase, es imposible realizar la superposición.

Lo que sí que podemos hacer en un constructor es aprovechar el constructor de la superclase. Supongamos que tenemos un constructor, y, de entre los parámetros que queremos inicializar, algunos no los tenemos en nuestra clase, sino en la superclase. En ese caso, debemos llamar al constructor de la superclase dentro de nuestro constructor. Esa llamada se realiza con `super(arg1, arg2,...)`. Los argumentos serán los definidos en el constructor de la superclase.

5.17. Resumen

En este tema se han explicado bastantes conceptos, todos ellos muy importantes:

- Existen tres tipos de variables, en función de su alcance: variables **locales**, de **instancia** y de **clase** (definidas con el modificador `static`). Los métodos también pueden clasificase en métodos de instancia o de clase.

- Los **modificadores** son palabras reservadas que permiten modificar la definición y el comportamiento de las clases, métodos y variables. Entre ellos destacan los de control de acceso (`public`, `private`, `protected` y *`Friendly`*).

- La **finalización** permite bloquear la implementación de una clase, variable o método, evitando que tenga subclases, que se pueda modificar, o que se pueda superponer, respectivamente.

- La **firma** *de un método* lo constituyen el objeto o tipo primitivo que devuelve, el nombre del método, y sus parámetros.

- El **polimorfismo** es la capacidad de utilizar un objeto de una subclase como si fuera un objeto de la clase base. Ello nos permite escribir *código desacoplado*.

- La palabra reservada `this` se refiere siempre al objeto actual. La palabra reservada `super`, a la superclase.

- La **sobrecarga** de métodos y constructores consiste en crear varios métodos con el mismo nombre, pero con distintas firmas y definiciones.

- Mediante la **superposición** de métodos podemos redefinir o ampliar un método de una superclase. Se realiza definiendo un método con la misma firma que el método original.

Uf, ha sido un tema muy largo. Si aún no has cerrado (o quemado) el manual y echado a correr, debes saber que ya puedes programar en Java. Bueno, no, realmente aún queda muchísimo por aprender, y desde luego, no lo vas a encontrar todo en este libro. Sin embargo, los conceptos básicos que nos permiten realizar (casi) cualquier programa ya han sido explicados en estos cuatro temas.

En los capítulos siguientes se explicarán conceptos avanzados que nos permitan realizar una programación más estructurada y modular.

Capítulo 6

Conceptos Avanzados de Java

6.1. Introducción

Quizá el título de este capítulo sea demasiado pretencioso. Cuando hablamos de conceptos avanzados, nos referimos a aspectos menos básicos que nos facilitarán la escritura de nuestros programas, permitiendo escribir código más flexible, escalable y modular[1].

En este capítulo se estudiarán la abstracción de clases, las interfaces, el control y gestión de interrupciones y la entrada-salida (E/S).

6.2. Abstracción de clases y métodos

Existe un tipo de clases, ubicadas generalmente en la parte superior de la jerarquía de clases de Java, que sirven de *"referencia"* común para todas sus subclases. Se denominan **clases abstractas**, y no pueden tener instancias.

Las clases abstractas pueden contener métodos concretos (los de toda la vida), pero también métodos abstractos. Un método abstracto consiste, únicamente, en la firma del método, pero sin la implementación. No pueden existir métodos abstractos fuera de una clase abstracta, y si una clase contiene un método abstracto, aunque sea uno sólo, deberá ser definida como abstracta.

Al contrario que en la herencia, en la que obligamos a las subclases a utilizar los métodos de la superclase tal y como están (salvo que los superpongamos), con la abstracción estamos diciéndole a las subclases que implementen los métodos como quieran, pero les obligamos a incluirlos con la misma firma que la definida en la clase abstracta.

Para definir una clase o un método abstracto, se le antepone el modificador `abstract`.

Puesto que no se pueden instanciar, la forma de utilizar una clase abstracta será creando una subclase que la *"concrete"*. Veamos el siguiente ejemplo:

[1] En cristiano: que el programa puede ser modificado fácilmente, que se puede ampliar sin demasiados problemas, y que nos permite separarlo en partes independientes que se comunican entre sí (recuérdese el concepto de objeto del tema 2).

Programa 6.2.1 Una clase abstracta y dos subclases para concretarla.

```
import java.text.*;
import java.util.*;

public abstract class ClaseAbstracta{
       protected String texto =null;
       public abstract void setTexto(String dato);
       public abstract String getTexto();
}

class Concreta1 extends ClaseAbstracta{
       public void setTexto(String dato){
            texto = dato;
       }

       public String getTexto(){
            return texto;
       }

       public static void main(String args[]){
            Concreta1 concreta =new Concreta1();
            concreta.setTexto("hola");
            System.out.println("Cadena de texto en Concreta1: "+concreta.getTexto());
       }
}

class Concreta2 extends ClaseAbstracta{
       Date fecha =new Date();
       String fechaTexto = DateFormat.getDateInstance().format(fecha);

       public void setTexto(String dato){
             texto = dato + " (Cadena añadida el "+fechaTexto+")";
       }

       public String getTexto(){
             return texto;
       }

       public static void main(String args[]){
             Concreta2 concreta = new Concreta2();
             concreta.setTexto("hola");

             System.out.println("Cadena de texto en Concreta2: "+concreta.getTexto());
       }
}
```

Vemos que:

- La clase `ClaseAbstracta` se define con el modificador `abstract`. Contiene una variable protegida (e.d. accesible a sus subclases) y dos métodos, también abstractos. Como puede comprobarse, únicamente aparece la firma del método, terminada por un punto y coma (`;`).
- Como los métodos son abstractos, podemos implementarlos como queramos, siempre y cuando

respetemos la firma definida en la clase abstracta. La clase `Concreta1` se limita a utilizar los métodos para introducir y recoger una cadena en la variable. La clase `Concreta2` hace lo mismo, pero además, añade la fecha en la que se introdujo la cadena.

Es importante saber que, cuando definamos una subclase de una clase abstracta que contiene muchos métodos abstractos, no estamos obligados a implementarlos todos. Sin embargo, los métodos que no implementemos debemos definirlos como abstractos. Por tanto, al tener métodos abstractos, estaremos obligados a definir nuestra clase como abstracta, y no podremos instanciarla. Tendremos que crear más subclases de nuestra subclase que implementen los métodos restantes, hasta llegar a una en la que estén implementados todos. Sólo entonces podremos instanciar esa clase.

6.3. Excepciones

Cualquier programa, antes o después, sufrirá errores durante su ejecución. En ocasiones es culpa de la pereza del programador, que no ha tenido en cuenta situaciones que pueden producir fallos. Otras veces se debe a una entrada incorrecta de datos por parte del usuario, o simplemente, se dan problemas extraños que están fuera del alcance del programa (como fallos en la red, cortes de luz, aviones que se estrellan contra rascacielos, etc.).

Para minimizar en lo posible las situaciones en las que puede fallar un programa, lo primero es revisar concienzudamente el código y asegurarse de que se ha tenido en cuenta el mayor número de casos que pueden *"colgar"* nuestro programa o producir un comportamiento erróneo.

Y, en segundo lugar, tenemos las **excepciones**, que no solucionan los fallos pero nos permiten recuperarnos de éstos de una forma elegante. Es decir, nos ayudan a conseguir que nuestro programa sea tolerante a fallos.

6.3.1. Captura de excepciones

Las excepciones son clases de Java[2]. Si echamos un vistazo al API, veremos que todas heredan de `java.lang.Exception`, directa o indirectamente. Utilizaremos esas clases, por una parte, para *"capturar"* los errores que puedan producirse en nuestro programa y, por otra, para indicar en nuestro código el tipo de errores que es posible que genere, lanzando nuestras propias excepciones. Como siempre, se ve todo mejor con un ejemplo:

Programa 6.3.1 Manejo Básico de Excepciones.

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

    int[] numeros = new int[10];

    try{
        int extraigoNumero = numeros[10];
       } catch(java.lang.ArrayIndexOutOfBoundsException e)
       {
         System.out.println("Se ha producido una excepción.");
         System.out.println("Intento de acceder a una posición inexistente de un array.");
         System.out.println("Excepcion: "+e.getMessage());
       }
    }
}
```

[2]Concretamente, la clase `Exception`, junto con la clase `Error`, heredan de la superclase `Throwable` ("arrojable"). Por eso se habla de *"arrojar"* o *"lanzar"* excepciones en un programa.

Como se ve en el ejemplo, el código susceptible de generar excepciones se encierra entre una sentencia `try` y una sentencia `catch`. Es decir, le decimos que pruebe ese código y, si lanza alguna excepción, que la capture. El parámetro de `catch` es una variable de la clase de excepción que pretendemos capturar. En nuestro caso, `ArrayIndexOutOfBoundsException` es una excepción arrojada cuando se intenta acceder a una posición inexistente de un array, que es lo que hemos hecho intencionadamente.

A continuación, dentro del cuerpo del `catch`, ejecutamos nuestras sentencias de control de la excepción. El método `getMessage()` (consultar el API) suele ser muy útil para obtener detalles de por qué se ha producido la excepción.

Por cierto, que este es uno de esos casos que no deberían ocurrir nunca. Si el programador es un poco cuidadoso, debería preocuparse de no acceder a una posición inexistente de un vector, con lo que se ahorraría el manejo de esa excepción en particular.

El lector avezado estará pensando que, en vez de capturar un tipo de excepción en particular, podría usar la clase `Exception`, ya que todas las demás heredan de ella, y nos aseguraríamos de que no se nos pasara por alto ninguna excepción. Sí se puede, pero ocurre lo mismo que cuando hablábamos de utilizar la clase `Object` en todas partes: que es como matar moscas a cañonazos. Recogeremos una excepción, pero sin saber de qué tipo es, ni cómo actuar para cada clase de fallo. Sin embargo, de este modo, nos aseguramos de particularizar el código de gestión de excepciones para cada tipo de excepción que es lanzada. Es decir, podemos adecuar la acción a realizar según el problema que tengamos. Lo haremos concatenando varios `catch` para un solo `try`:

```
try{
    // Ponga sus sentencias aquí.
    // This space for rent ;-)
       ...
} catch(java.lang.ArrayIndexOutOfBoundsException e){
   //sentencias de control de la excepción
} catch(FileNotFoundException e2){
   //sentencias de control de la excepción
} catch(IOException e3){
   //sentencias de control de la excepción
}
```

6.3.2. Lanzando Excepciones

Si podemos capturar excepciones en nuestro código, es porque los métodos que estamos invocando tienen la capacidad de lanzarlas. ¿Cómo podemos lanzar nosotros excepciones desde nuestros métodos? Hay que seguir dos pasos:

1. En la firma del método debe especificarse la o las excepciones que puede lanzar. Para ello, utilizamos la palabra reservada `throws`:

```
public String leeFichero(String fichero) throws IOException,
                                        FileNotFoundException{

        // Cuerpo del método
}
```

Que indiquemos que el método puede lanzar esas excepciones no quiere decir necesariamente que vaya a hacerlo. Si no surge ningún problema, no las lanzará. Lo que se consigue con `throws` es que el compilador compruebe si el código que llama a nuestro método se ha preocupado de gestionar esas excepciones en el caso de que ocurran (es decir, si ha llamado al método desde un `try-catch`). Si no es así, nos dará un error y no nos dejará compilar. De este modo, Java consigue que los programas sean más robustos.

2. Una vez especificadas las excepciones podremos lanzarlas, desde el cuerpo de nuestro método, cuando nos convenga. Para ello, utilizaremos la palabra reservada `throw`:

```
throw new FileNotFoundException("No se ha encontrado el archivo");
```

Nótese que podemos añadir texto en el constructor de la excepción para aclarar el tipo de error que ha ocurrido. Esto es posible porque todas las excepciones heredan de `Exception`, y ésta posee, entre sus constructores, uno en el que se puede añadir un `String` con información adicional.
En ciertos casos no será necesario hacer un `throw`. Por ejemplo, si nues-tro método lee datos de un fichero, y se produce un error de lectura, automáticamente se lanzará una excepción `IOException`, aunque no lo hagamos nosotros.

6.3.3. Creando nuestras propias excepciones

Habrá ocasiones en las que queramos crear nuestras propias excepciones a medida. Para ello, debemos escribir una clase que herede de `java.lang.Exception` o de una de sus subclases. Por ejemplo, veamos el siguiente programa:

Programa 6.3.2 Creando nuestras propias excepciones.

```
public class MiExcepcion extends Exception{
         public MiExcepcion() { }

         public MiExcepcion(String mensaje){
             super(mensaje);
             System.out.println("Saliendo del sistema");

             //Otras acciones.
         }
}

class UsoDeExcepciones{

         void funcion(int a) throws MiExcepcion{
              if (a == 1) throw new MiExcepcion("Se ha producido una excepción propia.");
         }

         public static void main(String args[]){
              UsoDeExcepciones uso = new UsoDeExcepciones();

              try{
                  uso.funcion(1);

              }catch(MiExcepcion m){
                  System.out.println("Excepción: "+m.getMessage());
              }
         }
}
```

Analicemos el listado anterior:

- Se define una clase `MiExcepcion`, derivada de `Exception`. Por lo general, definiremos dos constructores. Uno, sin argumentos de ningún tipo. Y el otro, con la posibilidad de pasarle como parámetro el mensaje de error a mostrar. En este segundo constructor, el mensaje se enviará al constructor de la superclase `Exception`, utilizando `super()`. Esto sirve para que `Exception` coloque el mensaje donde corresponda, de forma que nos lo muestre cuando se solicite desde un programa (con `getMessage()`).

- A continuación, en otra clase distinta, definimos un método llamado `funcion` que puede lanzar una excepcion (nótese el `throws` de la firma) de clase `MiExcepcion` cuando se cumpla una condición determinada. Aprovechamos que tenemos un constructor en la excepción que admite mensajes de error para incluir nuestro propio mensaje.

- Por último, creamos un método `main` dentro de la misma clase, y llamamos al método `funcion()`. Si no lo hubiésemos hecho entre una pareja `try-catch`, el compilador nos lo indicaría, y no nos dejaría continuar. Dentro del `catch` accedemos al método `getMessage()` de la superclase de `MiExcepcion`, para que nos muestre por pantalla qué ha pasado.

6.3.4. Transfiriendo excepciones

¿Quién no se ha escaqueado de un marrón, pasándoselo a un compañero?. En Java ocurre lo mismo. Puede que estemos escribiendo un método susceptible de provocar una excepción, es decir, que llama a métodos que pueden arrojar excepciones. Quizá a nosotros no nos interese gestionar esas excepciones, y que quien lo haga sea el programa que llamó a nuestro método. El modo de hacerlo se muestra en el programa siguiente, que es una ampliación del programa 6.3.2:

Programa 6.3.3 Transferencia de excepciones.

```
class UsoDeExcepciones2{

      void funcion(int a) throws MiExcepcion{
           if (a == 1) throw new MiExcepcion("Se ha producido una excepción propia.");
      }

      void pasaMarrones(int variable) throws MiExcepcion{
                funcion(variable);

                //Otras acciones
      }

      public static void main(String args[]){

             UsoDeExcepciones2 uso = new UsoDeExcepciones2();

             try{
                 uso.pasaMarrones(1);

             }catch(MiExcepcion m){
                 System.out.println("Excepción: "+m.getMessage());
             }
       }
}
```

Nótese que se supone que ya tenemos la clase `MiExcepcion` compilada del listado anterior, así que no necesitamos volver a escribirla. ¿Cómo funciona la transferencia de excepciones?:

- Definimos una función `pasaMarrones()` que es, únicamente, una pasarela entre el `main` y el método `funcion`. Vemos que, en el código de `pasaMarrones()`, se está llamando a una función susceptible de devolvernos una excepción. En vez de capturarla, se incluye un `throws` en la firma del método, con el tipo (o los tipos) de excepción que puede arrojar, y ya será el `main` el que se ocupe de capturarlas.

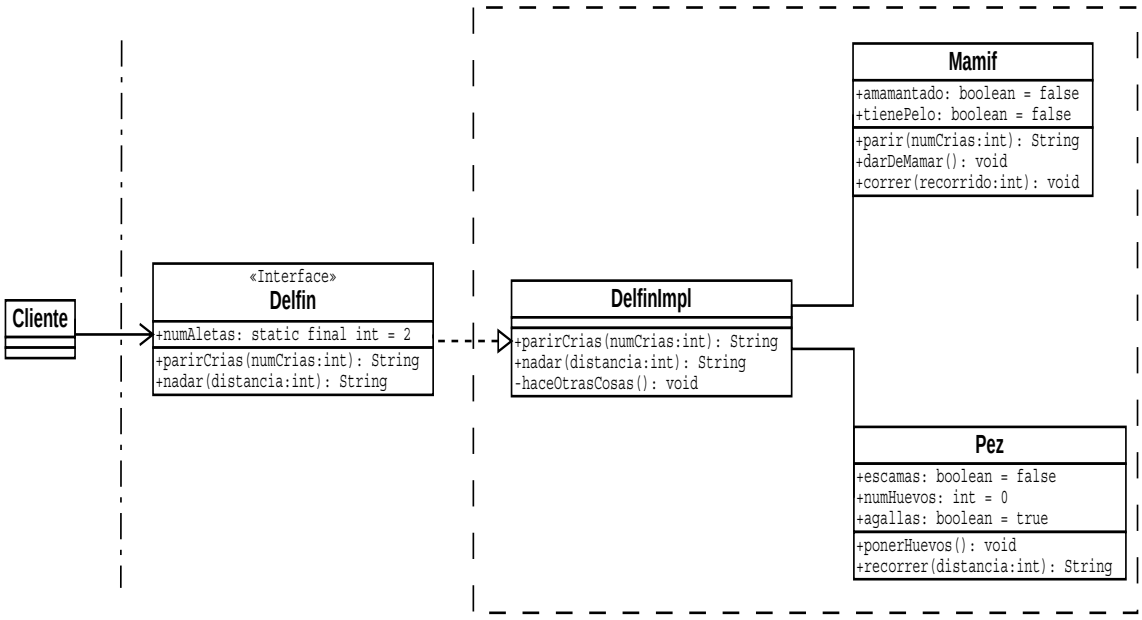

Figura 6.1: Implementación de una interfaz.

Será típico, cuando hagamos nuestros propios programas y capturemos excepciones, que incluyamos nuestro código de control dentro del `catch`, como se ha venido haciendo hasta ahora, y que, después, lancemos esa misma excepción fuera (con un `throw` dentro del `catch`), para que se le notifique el error al código que llamó al nuestro.

6.4. Interfaces

Sabemos que Java posee herencia simple. Por tanto, el comportamiento de una clase vendrá dado por los métodos que definamos en ésta o por los que heredemos de su superclase. En ocasiones, esto puede resultar restrictivo. Por ejemplo, supongamos que definimos una clase `Mamifero` con el comportamiento `parir()`. Y supongamos que también tenemos otra clase llamada `Pez` con el comportamiento `nadar()` y el atributo `agallas`. Estas dos clases pertenecen a jerarquías distintas. ¿Cómo podríamos definir entonces la clase `Delfin`, que pare a sus crías, pero también tiene agallas y nada?. O heredamos de una clase, o heredamos de la otra, o reescribimos todo el comportamiento en una nueva clase que no herede de ninguna de las dos.

Por otra parte, también podemos encontranos con que hemos escrito una clase con muchos métodos, pero sólo queremos *"desvelar"* a los demás un determinado número de ellos. ¿Cómo podríamos hacerlo?

La solución de Java a estos problemas son las *interfaces*. Una interfaz es una clase en la que sólo se especifica comportamiento, es decir, se definen únicamente firmas de métodos y constantes. Después, la implementación de esos métodos podrán realizarla varias clases (ojo, cada clase debe implementar todos los métodos de la interfaz). Se dice entonces que esas clases *implementan la interfaz*.

En nuestro ejemplo, definiríamos una interfaz `Delfin` con las firmas de los métodos que a nosotros nos interesen. Puede que la clase que implemente el interfaz tenga más métodos disponibles, pero quien acceda a la interfaz `Delfin` simplemente verá el comportamiento especificado en ésta, sin saber lo que hay detrás. Esto se ve gráficamente en la figura 6.1.

Por tanto, una interfaz nos permite establecer una jerarquía paralela a la herencia, ya que tenemos comportamientos no heredados de una superclase, y nos proporciona una capa de abstracción, ocultándonos lo que hay debajo de la interfaz. Esto puede ser muy útil, por ejemplo, cuando estemos desarrollando clases que van a interactuar con otras de las que no sabemos nada.

Basta con acordar entre los autores de las clases la interfaz con la que se van a comunicar, y luego, cada uno implementa por detrás esa interfaz como le venga en gana.

Nosotros vamos a escribir el ejemplo del delfín para ver, en la práctica, cómo se crean interfaces. Veamos los siguientes programas:

Programa 6.4.1 La interfaz `Delfin`, archivo `Delfin.java`.

```
public interface Delfin
{  public static final int numAletas = 2;
   String parirCrias(int numCrias);
   String nadar(int distancia);
}
```

Programa 6.4.2 Clase que implementa la interfaz, archivo `DelfinImpl.java`.

```
class DelfinImpl implements Delfin{

    public String parirCrias(int numCrias){
       Mamif mamifero = new Mamif();
       return mamifero.parir(numCrias);
    }

    public String nadar(int distancia){
        Pez pez = new Pez();
        return pez.recorrer(distancia);
    }

    private void haceOtrasCosas(){
       //no especificada por la interfaz
    }
}
```

Programa 6.4.3 La clase `Mamifero`, archivo `Mamif.java`.

```
class Mamif {
        boolean amamantado = false;
        boolean tienePelo = false;

        public String parir(int numCrias){
           return "El mamífero ha parido "+numCrias+" crías";
        }

        public void darDeMamar(){
           amamantado = true;
        }

        void correr(int recorrido){
            int distancia = 10;
            distancia += recorrido;
        }
}
```

Programa 6.4.4 La clase `Pez`, archivo `Pez.java`.

```
public class Pez{

       boolean escamas = false;
       int numHuevos = 0;
       boolean agallas = true;

       public String recorrer(int distancia){
          return "El pez ha nadado "+distancia+" metros";
       }

       public void ponerHuevos(){
           numHuevos++;
       }
}
```

Programa 6.4.5 Cliente de la interfaz.

```
class ClienteDelfin{

       public static void main(String args[]){

           Delfin delfin = new DelfinImpl();

           String mensaje = delfin.parirCrias(3);
           System.out.println(mensaje);

           mensaje = delfin.nadar(10);
           System.out.println(mensaje);
       }
}
```

Tenemos, como se mostró en la figura 6.1, una interfaz `Delfin`, una clase `DelfinImpl`, que implementa los métodos de esa interfaz (ya hemos dicho que está obligada a implementar **TODOS** los métodos), y dos clases independientes, `Mamif` y `Pez`, que son accedidas por `DelfinImpl`.

Para definir la interfaz se utiliza la palabra reservada `interface`. Una interfaz sólo puede ser pública o con protección de paquete, como la de nuestro ejemplo. No tendría sentido que fuera privada, o protegida (porque una interfaz no pertenece a ninguna jerarquía). Dentro de la interfaz sólo se definen las firmas de los métodos y/o variables. Los métodos, aunque no se especifique nada, son implícitamente `public` y `abstract`. Las variables son `public`, `static` y `final` (es decir, constantes).

Para implementar una interfaz, como se ve en la clase `DelfinImpl`, se utiliza la palabra reservada `implements` seguido del nombre de la interfaz que implementa. No tiene por qué ser una clase pública. Es más, la idea es que sea pública la interfaz, que es la que muestra a los demás los métodos disponibles, pero no la clase que implementa esos métodos.

La clase de implementación está obligada a implementar todos los métodos de la interfaz(creo que ya lo he dicho tres veces :-). Una clase no tiene por qué estar atada a una sola interfaz, sino que puede implementar varias simultáneamente. Por ejemplo, si tuviéramos una clase que contuviera un gran número de métodos, podríamos publicar unos pocos en una interfaz, otros en otra, un mismo método en las dos, etc. Entonces, cuando definiésemos esa clase, seguido de la palabra `implements`, pondríamos los nombres de las interfaces que implementa separados por comas.

En nuestro ejemplo, la clase de implementación instancia a las clases `Mamif` y `Pez`, *"cogiendo"* los métodos que le interesa de cada uno. Así, hemos definido, en la interfaz, un comportamiento personalizado a partir de dos clases que pertenecen a ramas distintas de la jerarquía de clases[3].

Por último, en el programa 6.4.5 se muestra un pequeño cliente que accede a la interfaz. Nótese que el operador `new` sólo puede utilizarse para crear una instancia de una clase, y no de una interfaz. Sin embargo, el nombre de una interfaz puede ser usado en lugar del nombre de cualquier clase que la implemente. Por eso, en nuestro cliente, instanciamos una clase `DelfinImpl`, y guardamos la instancia en un objeto de tipo interfaz. A continuación, trabajaremos con los métodos definidos por esa interfaz, invocándolos como si de una clase se tratara.

A estas alturas, y viendo el código del cliente, lo habitual es pensar que las interfaces no son tan útiles como parecen. ¿Cómo vamos a enmascarar lo que hay por detrás de la interfaz, si para obtener una instancia de ésta necesitamos saber el nombre de la clase que la implementa?. Pues para eso, instanciamos directamente la clase de implementación, trabajamos con todos los métodos de ésta, estén o no publicados en la interfaz, y nos quitamos de problemas.

Pues sí, visto así no tiene mucha lógica usar una interfaz. Sin embargo, cuando empecemos a trabajar un poco en serio con Java, y tengamos que utilizar clases de terceros, nos encontraremos, en el API que nos proporcionen, clases que devuelven interfaces directamente, sin tener que saber quién las implementa. Veamos el programa siguiente:

Programa 6.4.6 Una clase que proporciona una interfaz.

```
public class Generador{

          public Delfin Create(){
              return (new DelfinImpl());
         }
}
```

Nosotros no tendremos el código del programa 6.4.6. Sólo sabremos que, llamando al método `Create()` de esa clase, obtenemos una interfaz. Por tanto, nuestro nuevo cliente tendrá el siguiente aspecto:

Programa 6.4.7 Nuevo cliente de la interfaz.

```
class ClienteDelfin2{

      public static void main(String args[]){
             Generador gen =new Generador();
             Delfin delfin = gen.Create();

             String mensaje = delfin.parirCrias(3);
             System.out.println(mensaje);

             mensaje = delfin.nadar(10);
             System.out.println(mensaje);
      }
}
```

Y ya no podemos saber quién implementa la interfaz `Delfin`.

Por último, para terminar esta sección, dos comentarios:

[3]Es irrelevante en este apartado que `Mamif` tenga protección de paquete y `Pez` sea pública. Estamos suponiendo que ambas están en el mismo paquete que `DelfinImpl`, por lo que ésta va a encontrarlas sin problemas cuando quiera instanciarlas.

- Puede existir una jerarquía de interfaces paralela a la tradicional de clases de Java. Es decir, que una interfaz puede heredar de otra (`public interface Delfin extends Flipper`) o de otras (`extends SuperDelfin,SuperDelfina`). O sea, que tenemos **herencia múltiple para las interfaces**.

- La subclase de una clase que implementa una interfaz, también implementa esa interfaz. Puede darse el caso de que una clase herede de otra que implementa una interfaz, y a la vez implemente otra interfaz. Por tanto, estará implementando dos interfaces.

6.5. Entrada-Salida (E/S)

Lo último que se va a tratar en este tema es una pequeña introducción a la Entrada-Salida (E/S). Con E/S nos estamos refiriendo a trabajar con flujos de datos que se envían o reciben, generalmente, desde un periférico, como puede ser el teclado, la pantalla, un archivo en el disco duro, etc.

Si éste fuera un manual serio, explicaríamos teórica y detalladamente en qué consisten los flujos. Como no lo es, vamos a poner una serie de ejemplos prácticos de los casos más comunes de E/S, y, a continuación, los explicaremos.

6.5.1. Salida de datos por pantalla

Bueno, éste no necesita mucha explicación. El modo más usado de mostrar datos por pantalla es:

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

`System` es una clase final (no puede ser instanciada) del paquete `java.lang`. Posee tres campos:

in: flujo de entrada

err: flujo donde se redireccionan los errores

out: flujo de salida.

Este último devuelve una clase `java.io.PrintStream`, que es la que contiene el método `println()` con todas sus variantes. Por tanto, si queremos consultar los parámetros del método, debemos buscar en el API de Java esa clase.

6.5.2. Entrada de datos por teclado

Veamos el siguiente programa:

Programa 6.5.1 Programa que lee datos introducidos por teclado.

```
import java.io.*;
import java.text.*;

public class LeeDatos{

      public static void main(String args[]){

         String texto=null;
         int i = 0;

         System.out.println("Introduzca un entero:");

         try{
            InputStreamReader conversor = new InputStreamReader(System.in);

            BufferedReader entrada =new BufferedReader(conversor);
            texto = entrada.readLine();

            i = NumberFormat.getInstance().parse(texto).intValue();

         }
         catch (IOException e) { }
         catch (ParseException pe) { }

         System.out.println("Cadena: "+texto);
         System.out.println("Número: "+i);
      }
}
```

El código anterior permite leer un flujo de caracteres (una cadena de texto) del teclado hasta que el usuario presiona la tecla de retorno de carro (tecla *Enter*).

Analicemos el listado:

- La clase `InputStreamReader` lee un flujo de bytes del flujo de entrada especificado en su constructor (en nuestro caso, lo lee de `System.in`, la entrada por teclado), y los convierte en un flujo de caracteres.

- La clase `BufferedReader` admite en su constructor un flujo de caracteres y los *"formatea"* para que puedan ser leídos correctamente. De este modo, utilizando el método `readLine()`, obtendremos un `String` con la información introducida por teclado.

- Nótese que, en este ejemplo, estamos suponiendo que el usuario está introduciendo un número. Para convertir el `String` de entrada en un número, utilizamos la clase abstracta `java.text.NumberFormat`. El método `getInstance()`, típico en muchas clases abstractas, nos genera una implementación concreta por defecto de esa clase. Después llamamos al método `parse()`, que analiza el tipo de número que estamos manejando, y, por último, lo convierte en un entero.

- Necesitamos capturar dos tipos de excepciones. Por un lado, `IOException` que, aunque es una clase muy general, nos servirá para gestionar cualquier problema al introducir los datos. La clase `ParseException` se ha añadido para poder capturar las excepciones del método `NumberFormat.getInstance()`. Realmente, no la necesitaremos si sólo vamos a leer cadenas de texto.

6.5.3. Lectura de datos de un fichero

El programa siguiente muestra un programa que lee líneas de un fichero y las muestra por pantalla.

Programa 6.5.2 Programa que lee datos de un fichero.

```
import java.io.*;

class LeeFichero{

     public static void main(String args[]){

         File fichero = new File(args[0]);

         if(!fichero.exists() || !fichero.canRead()){
             System.out.println("No se puede leer el fichero "+fichero);
             return;
         }

         if(fichero.isDirectory()){
             String[] archivos = fichero.list();

             for(int i=0; i < archivos.length; i++)
                System.out.println(archivos[i]);
         }
         else
             try{
                FileReader fr = new FileReader(fichero);
                BufferedReader entrada =new BufferedReader(fr);
                String linea;

                while((linea = entrada.readLine()) !=null)
                      System.out.println(linea);
                }
                catch (FileNotFoundException e){
                      System.out.println("El archivo ha desaparecido!!");
                }
                catch (IOException ioex){
                      System.out.println("Fallo al leer el archivo.");
                }
             }
}
```

Veamos:

- El nombre del fichero se introduce desde la línea de comandos cuando se llama al programa. Así que lo primero es crear un objeto `File` con ese nombre de fichero o directorio. Nótese que el objeto se crea independientemente de que el fichero exista realmente. Por tanto, será responsablilidad nuestra comprobar que el nombre introducido pertenece a un fichero o directorio real.

- Para hacer esa comprobación, la clase `File` posee dos métodos (posee un montón más, muy útiles; consulta el API): `exists()` y `canRead()`, para asegurarse de que el fichero existe y tiene permisos de lectura, respectivamente.

- A continuación, se comprueba si el nombre pertenece a un directorio, ya que `exists()` sólo se asegura de la existencia, pero no del tipo, de fichero o directorio. Si se trata de un directorio, mediante el método `list()`, obtiene un array de Strings conteniendo, en cada posición del array, el nombre de cada uno de los ficheros que contiene el directorio. Y los muestra por pantalla.

- Si no es un directorio, es que estamos trabajando con un fichero. Creamos un objeto `FileReader`, clase hija de `InputStreamReader`, vista en el ejemplo anterior, y con una funcionalidad similar a ésta, sólo que lee flujos del fichero.

- El resto del código es conocido: creamos un objeto `BufferedReader` a partir del flujo leído por `FileReader`, y vamos leyendo línea a línea. Además, capturamos las posibles excepciones que puedan originarse.

6.5.4. Escribiendo datos en un fichero

El programa siguiente muestra un programa que lee una línea del teclado y la escribe en el fichero que hayamos especificado desde la línea de comandos.

Programa 6.5.3 Programa que escribe datos en un fichero.

```
import java.io.*;

class EscribeFichero{

        public static void main(String args[]){

           File fichero = new File(args[0]);

           try{
              String s = new BufferedReader(new InputStreamReader(System.in)).readLine();
              FileWriter fw = new FileWriter(fichero);

              PrintWriter pw = new PrintWriter(fw);
              pw.println(s);
              fw.close();
           }
           catch (FileNotFoundException e){
                System.out.println("El archivo ha desaparecido!!");
           }
           catch (IOException ioex){
                System.out.println("Fallo al leer el archivo.");
           }
        }
}
```

Analizamos el código:

- Tras crear un objeto `File` con el nombre de fichero introducido por teclado (por defecto, siempre se refiere al directorio actual), leemos una línea del teclado, tal y como se explicó en el programa 6.5.1.

- Para poder escribir, necesitamos un objeto de la clase `FileWriter`, opuesta a `FileReader`, y que crea un flujo que permite escribir caracteres en el archivo especificado en su constructor.

- `PrintWriter` permite formatear el texto que se va a mandar por el flujo (o sea, el archivo). A continuación, lo escribe mediante su método `println()`.

- Por último, se llama al método `close()`, para cerrar el flujo. Es una buena costumbre, aunque no lo hayamos hecho en los ejemplos anteriores.

6.6. Resumen

Los aspectos importantes a recordar de este tema son:

- Las clases y métodos abstractos no pueden instanciarse. Proporcionan una plantilla para que sus subclases implementen los métodos del modo que quieran, pero respetando la firma. Se distinguen por el modificador `abstract`.

- Cuando un programa en Java falle, lanzará excepciones. Capturándolas, mediante una pareja `try-catch`, podremos ejecutar código que permita al programa recuperarse del problema. De este modo, conseguiremos que nuestro código sea tolerante a fallos.

- Las interfaces son clases que únicamente especifican un comportamiento. Permiten publicar una serie de métodos de cara al cliente, ocultando la implementación subyacente.

- Java posee funciones de Entrada-Salida que nos permiten obtener datos por teclado, mostrar información por pantalla, y trabajar con ficheros.

Apéndice A

El Compresor `jar`

Cuando distribuyamos nuestros programas es probable que los hayamos incluido todos en un paquete. Por tanto, tendremos que haber creado una estructura de directorios en función del nombre del paquete (ver sección anterior). Es bastante incómodo tener que proporcionar nuestras clases con esa estructura de directorios. La solución es empaquetar nuestras clases en un archivo de extensión `jar`, que mantiene internamente la estructura de directorios del paquete.

Supongamos que nuestras clases pertenecen al paquete `otrasclases.misclases`. Estarán, por tanto dentro del subdirectorio `misclases`, el cual estará contenido en el subdirectorio `otrasclases`. Si nos colocamos en el directorio superior a `otrasclases`, y ejecutamos el comando

```
jar cf misclases.jar otrasclases/misclases/*.class
```

se comprimirán y empaquetarán automáticamente (opción `c`) todas las clases (`*.class`) del directorio `otrasclases/misclases` en el archivo (opción `f`) de nombre `misclases.jar`, respetándose la estructura de directorios.

De este modo, lo que nosotros distribuiremos será ese archivo `jar`. Cuando alguien desee ejecutar nuestras clases, deberá escribir:

```
java -classpath misclases.jar otrasclases.misclases.MiClase
```

indicando, mediante una redefinición del CLASSPATH, que busque las clases en misclases.jar, y que ejecute la clase indicada (respetando siempre la estructura de directorios, separados por puntos).

Para descomprimir un paquete `jar`, usaremos la opción `x` del comando `jar`[1]. Te remito al manual (`man jar`) para más información.

[1]La utilidad `jar` utiliza el mismo tipo de compresión que tienen los archivos de extensión `zip`. Por ello, si no tenemos el programa a mano, podemos renombrar la extensión del archivo a `zip`, y abrirlo con cualquier programa de compresión que soporte ese formato, como, por ejemplo, *Winzip* bajo Windows.

Bibliografía

[1] Eckel, B., Thinking in Java, tercera edición, revisión 4.0, *http://www.mindview.net/Books/TIJ/*. Última visita, 30 de Octubre de 2003.

[2] Lemay, L., Perkins, Charles L., Aprendiendo Java 1.1 en 21 días, segunda edición, Ed. Prentice-Hall Hispanoamericana S.A. 1998.

[3] Niemeyer, P., Learning Java. Ed. O'Reilly, 2000.

[4] Sun Microsystems web: *http://www.java.sun.com.* Última visita, 30 de Octubre de 2003.