Introducción a la Programación Orientada a Objetos con Java

Rafael Llobet Azpitarte
Pedro Alonso Jordá
Jaume Devesa Llinares
Emili Miedes De Elías
María Idoia Ruiz Fuertes
Francisco Torres Goterris

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Introducción a la Programación Orientada a Objetos con Java[1]

Primera edición
Marzo, 2009
Rev. 1.0.1

Rafael Llobet Azpitarte
Pedro Alonso Jordá
Jaume Devesa Llinares
Emili Miedes De Elías
María Idoia Ruiz Fuertes[2]
Francisco Torres Goterris

Departamento de Sistemas Informáticos y Computación

Universidad Politécnica de Valencia

ISBN: 978-84-613-0411-0

[1] Java es una marca registrada de Sun Microsystems.

[2] Con el apoyo del programa de ayudas FPI del Ministerio de Ciencia e Innovación (ref. BES-2007-17362).

Licencia

Índice general

Capítulo 1

Introducción a la Programación Orientada a Objetos

1.1. Introducción

Antes de empezar a desarrollar las características propias de la programación orientada a objeto, conviene hacer una revisión de alto nivel de la programación, sus fases y sus diferentes métodos.

En el proceso de desarrollo de un sistema de información (un programa, *software*, en general) hay una serie de etapas o fases en las que la programación como tal es una de ellas, ni tan siquiera la más importante. Hay diferentes modelos de desarrollo en los que se definen esas fases o etapas antes comentadas; uno de ellos es el método en cascada (*waterfall*) que nos servirá como guía en el desarrollo de estas ideas.

Según este modelo, a grandes rasgos, el desarrollo software consta de las siguientes fases:

- Análisis: Esta es una de las fases más importantes puesto que se trata de definir y analizar el problema en su totalidad. En general, se trata de entender el enunciado del problema. Evidentemente, para una resolución correcta (eficaz y eficiente) de un problema lo mejor es conocerlo.

- Diseño: Junto con la anterior, la más importante y consiste en, dado el análisis anterior, diseñar una solución del problema. Se trataría de definir los módulos, patrones, algoritmos, etc. que nos ayudaran a su solución. Entre esta fase y la anterior, se debería consumir un 70-80 % del tiempo del proyecto.

- Implementación: Sería un equivalente a la programación. En este caso, el diseño anterior se traduciría a un lenguaje de programación concreto; en esta fase es donde realmente se programa (codifica).

- Pruebas: Periodo en el que el producto se somete a diferentes tipos de pruebas: de sistema, de integración, etc.

- Implantación: Proceso de puesta en producción del producto.

- Mantenimiento: Realizar mejoras varias sobre el producto desde el punto de vista tecnológico, funcional, etc.

Normalmente, siempre nos centramos en la fase de codificación/implementación pero, como vemos, este proceso es mucho más complejo de lo que podríamos pensar.

Cuando se decide iniciar un desarrollo, lo primero que se debe decidir es el paradigma de trabajo. La elección del paradigma marca significativamente la forma de análisis y diseño de la solución del problema. Así un mismo problema se podría abordar usando un paradigma procedural clásico (es el que se ha usado en cursos anteriores de programación, cuando se ha programado en Pascal o C, por ejemplo) o bien un paradigma orientado a objetos (el que usaremos en este curso). Por ejemplo, el diseño de un software que implementara el juego del ajedrez en un paradigma procedural y en un paradigma orientado a objeto tendrían poco que ver.

La elección del paradigma marca la elección del lenguaje. Así, si hemos elegido un paradigma procedural para resolver el problema, lo normal es que lo implementemos en un lenguaje típicamente procedural (C o PASCAL, por ejemplo); por otro lado, si elegimos un paradigma orientado a objetos es normal elegir un lenguaje orientado a objetos (C++ o Java, por ejemplo).

Ocurre bastantes veces que se cree que un programador de *C* o *Pascal* o cualquier otro lenguaje de programación procedural puede convertirse en un programador de *C++*, *Object Pascal* o *Java* sin más que aprender una serie de nuevas estructuras e instrucciones. Por desgracia, esto no es así en absoluto; el estudiar un lenguaje orientado a objetos va más allá que "aprender otro lenguaje más". En realidad, el cambio es más profundo y se trata del estudio de un nuevo paradigma, esto es, una nueva forma de abordar los problemas y una manera de implementar esas soluciones mediante el lenguaje de programación Java.

Por tanto, el objetivo del curso se debería ver desde la perspectiva de este cambio de paradigma y el uso del lenguaje de programación Java considerarlo como un medio, más que como un fin. Una vez revisado el paradigma, el aprendizaje de cualquier lenguaje orientado a objetos sería bastante más simple. Analizaremos en los siguientes párrafos el porqué del estudio de un nuevo paradigma.

Por otro lado, el objetivo de este tema es introducir conceptos para que de manera intuitiva se vaya comprendiendo qué significa cambiar de paradigma; y concretamente usar el paradigma orientado a objetos. Cada uno de estos conceptos se verán desarrollados en temas específicos a lo largo del texto y del curso; este tema sólo pretende dar un repaso a nivel muy abstracto de cada una de las características fundamentales que diferencian un lenguaje orientado a objetos.

1.2. La Orientación a Objetos

La orientación a objetos promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo en el desarrollo de software:

- La falta de portabilidad del código y su escasa reusabilidad.
- Código que es difícil de modificar.
- Ciclos de desarrollo largos.
- Técnicas de codificación no intuitivas.

Un lenguaje orientado a objetos ataca estos problemas. Tiene tres características básicas: debe estar basado en objetos, basado en clases y capaz de tener herencia de clases. Muchos lenguajes cumplen uno o dos de estos puntos; muchos menos cumplen los tres. La barrera

más difícil de sortear es usualmente la herencia. El concepto de programación orientada a objetos (POO) no es nuevo, lenguajes clásicos como *SmallTalk* se basan en ella.

Dado que la POO se basa en la idea natural de la existencia de un mundo lleno de objetos y que la resolución del problema se realiza en términos de objetos, un lenguaje se dice que está basado en objetos si soporta objetos como una característica fundamental del mismo. No debemos confundir que esté basado en objetos con que sea orientado a objetos: para que sea orientado a objetos al margen que esté basado en objetos, necesita tener clases y relaciones de herencia entre ellas.

Hemos utilizado con mucha frecuencia la palabra *paradigma*, que convendría formalizar de alguna forma: Se entiende paradigma como un conjunto de teorías, estándares y métodos que juntos representan una forma de organizar el conocimiento, esto es, una forma de ver el mundo. La visión OO nos obliga a reconsiderar nuestras ideas a cerca de la computación, de lo que significa ponerla en práctica y de cómo debería estructurarse la información en los sistemas de información.

1.3. Los objetos

El elemento fundamental de la POO es, como su nombre indica, el **objeto**. Podemos definir un objeto como un conjunto complejo de datos y programas que poseen estructura y forman parte de una organización. En este caso las estructuras de datos y los algoritmos usados para manipularlas están encapsulados en una idea común llamada objeto.

Esta definición especifica dos propiedades características de los objetos:

- En primer lugar, un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados.
- En segundo lugar, cada objeto no es un ente aislado, sino que forma parte de una organización jerárquica o de otro tipo.

Para ilustrar mejor las diferencias entre la programación procedural clásica y la POO vamos a plantear, de forma intuitiva, un pequeño ejemplo escrito en C y después planteado en Java. Se trata de tener en un array una serie de figuras geométricas que tendrán lógicamente los subprogramas encargados de leer los datos, dibujarlos, etc...

La estructura de datos en C podría ser:

```
struct Figura {
  int Tipo; // 1.-hex 2.-rect
  int Color;
  int Borde;
  float x, y, z, t, w;
}

struct Figuras [N];

void function LeeHex (...........);

void function LeeRect (.......);

void function DibujaHex (.......);

void function DibujaRect (.....);
```

En el caso de que quisiéramos dibujar todas las figuras que aparecen en el array haríamos algo parecido a esto:

```
for (int i=1; i<n; i++) {
  switch (f[i].tipo) {
    case 1 : DibujaHex (....); break;
    case 2 : DibujaRect (...); break;
             .
             .
  }
}
```

En general, tendremos para cada figura sus correspondientes funciones que se encarguen de leer las coordenadas, de dibujarlos por pantalla, etc. ¿Qué ocurre si quiero añadir una nueva figura geométrica? Tendremos que modificar el programa añadiendo el tipo correspondiente y los subprogramas necesarios para manejarlos, además es más que probable que tengamos que revisar la mayor parte del código para actualizarlo a la nueva situación. Esta tarea será más o menos dificultosa en función de lo ordenado que se haya sido al crear el código.

Si resolviéramos este mismo problema de forma OO la solución a la hora de añadir una nueva figura geométrica no sería tan aparatosa. En primer lugar definiríamos un nuevo tipo de figura, por ejemplo, para representar los nuevos objetos "Círculos" que tendrán sus atributos propios (en este caso, por ejemplo, el punto que define el centro y el radio del círculo) y sus métodos, como por ejemplo `LeerCoordenadas` y `Dibujar`. De forma que el `for` anterior se podría reescribir de esta forma:

```
for (int i=1; i<n; i++) {
  v[i].Dibujar();
}
```

Como vemos, el hecho de añadir un nuevo tipo de objeto no implica ninguna modificación en la estructura. Lógicamente implicará añadir la nueva figura al código, pero si se ha sido cuidadoso a la hora de encapsular los datos propios (atributos) y los subprogramas encargados de manejarlos (métodos) será bastante más sencillo mantener y trabajar con el código. En la figura 1.1 se muestra un esquema de cómo se ven los objetos desde el punto de vista que acabamos de plantear.

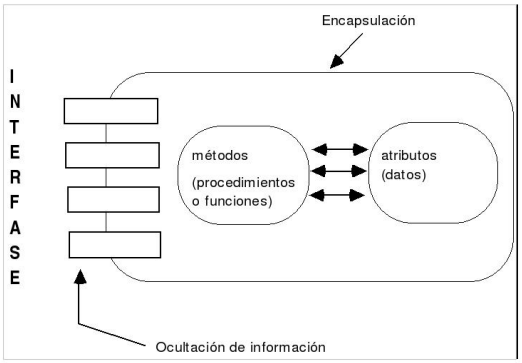

Figura 1.1: Encapsulación de métodos y atributos.

Usando como ejemplo el caso de un objeto círculo tendremos que, desde un punto de vista estructural, está compuesto de:

- Atributos: como podrían ser las coordenadas que definen el centro del círculo y el radio.
- Métodos: por ejemplo, el que nos permite leer o modificar los valores de estos atributos, dibujarlo o bien calcular su circunferencia o su área.

Son conceptos fundamentales en este paradigma la idea de encapsulación, en cuanto a que atributos y métodos forman el objeto como una única unidad y el de ocultación de información en cuanto que los atributos deben estar lo más ocultos posibles al exterior y sólo ser manipulados o consultados a través de los métodos pertinentes que son los que conforman la interfaz del objeto.

1.4. Las clases

En el punto anterior hemos definido de qué trata un objeto; pero en realidad el concepto más adecuado a la hora de definir las características de los objetos de nuestro sistema es el concepto de clase. Una clase es la descripción de una familia de objetos que tienen la misma estructura (atributos) y el mismo comportamiento (métodos). En el caso que estamos siguiendo tendríamos la clase `Circulo`en el que se definirían los atributos y los métodos que definen cualquier círculo. Cada ejemplar de círculo sería un objeto (o instancia) de la clase círculo en los que cada uno tendría sus propias coordenadas (su centro y su radio), y todos compartirían los mismos métodos.

De esta forma se organizan los conocimientos que tenemos del sistema; las clases son el patrón habitual que proporcionan los lenguajes orientados a objetos para definir la implementación de los objetos. Por ejemplo, supongamos que en nuestro sistema detectamos la necesidad de tener empleados, por tanto definimos las características comunes de todo empleado en un pseudo-código orientado a objetos:

```
Clase Empleado {
  Atributos
    DNI: Cadena;
    NRP: Cadena;
    Nombre: Cadena;
    Dirección: Cadena;
    AñoEntrada: Integer;
    AñoNacimiento: Integer;
    Complementos: Real;

  Metodos
          :
          :
    Integer AñosEmpresa();
          :
          :
    Integer Edad();
          :
          :
    Real Sueldo(Real Extras);
          :
          :
}
```

Todos los objetos son instancias de una clase, es decir, ejemplares o ejemplos concretos de la clase. Como vemos, la clase lo define todo, es una entidad autocontenida en cuanto que contiene todo lo necesario para definir la estructura o los datos (atributos) que cada objeto tendrá y la manera de manejarse esos objetos a través de los métodos.

Dada la clase anterior, podremos crear (ya veremos cómo) los empleados (objetos, instancias de la clase, ejemplares, etc..) que necesite de los que tendré una referencia, es decir, una variable que apunte a ese empleado. Así podré tener la referencia emp1 que coincide con el empleado con DNI 00001, con NRP 2345, de nombre José García, etc... Si quiero obtener el sueldo de este empleado tendré que invocar al método correspondiente en el objeto que lo representa a través de la referencia de la siguiente forma:

```
emp1.Sueldo(136000);
```

Esta invocación tendrá como contrapartida la ejecución del método `Sueldo` en el objeto especificado y no en otro.

Recordando lo comentado en el apartado anterior, y siendo más formales en la definición, cada clase posee una doble componente:

- Una componente estática, los datos, que son campos con nombres, que poseen ciertos valores. Estos campos caracterizan el estado del objeto. A los campos les llamaremos usualmente **atributos**.
- Una componente dinámica, los procedimientos o funciones, llamados **métodos**, que representan el comportamiento común de los objetos que pertenecen a la misma clase. Los métodos son los encargados de manipular los campos de los objetos (además, es una práctica muy mala el modificar atributos de forma directa: rompe la encapsulación y la ocultación de la información) y caracterizan las acciones que se pueden realizar sobre los mismos. Esta componente dinámica forma la llamada *interfaz* de la clase, y suministra la única forma de acceso a los objetos. Para efectuar una operación asociada con un objeto se debe mencionar el objeto implicado, que llamaremos el receptor del mensaje, el método a activar, y los argumentos sobre los que se aplica.

En cuanto a los métodos conviene comentar que no todos tienen por qué ser accesibles desde el exterior a cualquier usuario; puede que existan métodos que sólo sean visibles para un grupo concreto de clases o sólo para la clase en la que se definen. Una visión sencilla de un objeto es la combinación de estado y comportamiento. El estado se describe mediante los atributos, en tanto que el comportamiento se caracteriza por los métodos.

1.5. La iniciación de instancias

Como ya se ha comentado en el apartado anterior, una clase es una entidad conceptual que describe un conjunto de objetos. Su definición sirve como modelo para crear sus representantes físicos llamados **instancias** u **objetos**.

En la declaración de un objeto establecemos la referencia que identificará al objeto mediante un identificador. Por creación o iniciación nos referiremos a la asignación de espacio de memoria para un nuevo objeto y la ligadura de dicho espacio a su identificador; además, por iniciación aludiremos no sólo a la puesta de valores iniciales en el área de datos para un objeto, sino también al proceso más general de establecer las condiciones iniciales básicas para la manipulación de un objeto.

La iniciación se facilita usando *constructores de objetos*. Un constructor es un método que tiene el mismo nombre que la clase de la cual es constructor. El método se invoca siempre

que se crea un objeto de la clase asociada. Veamos un ejemplo de una clase que representa números complejos escrito esta vez ya en Java y no en pseudo-código:

```
class Complejo {

  private int real;
  private int imag;

  Complejo (int pr, int pi) {
    real = pr;
    imag = pi;
  }

  Complejo () {
    real = 0;
    imag = 0;
  }

  int real () {
    return (real);
  }

  int imag () {
    return (imag);
  }
}
```

En primer lugar nos fijamos en que hemos construido métodos para acceder a los atributos de la clase. Éstos los hemos declarado privados para enfatizar que sólo deben poder ser accedidos desde fuera de la clase a través de los métodos que creemos a tal fin. Si quisiéramos hacer uso de un número complejo, deberíamos tener una porción de código similar a la siguiente:

```
Complejo c;
   ...
c = new Complejo(34, 4);
```

Con lo que se crearía un objeto de tipo `Complejo` con los valores 34 y 4 para la parte real e imaginaria, respectivamente. Una instancia es un objeto particular, que se crea respetando los planes de construcción dados por la clase a la que pertenece. Todas las instancias de una misma clase tienen los mismos atributos con los valores que le corresponda. Todas las instancias poseen los mismos métodos, sólo que al invocarlos, sólo se ejecuta el del objeto de esa clase al que se le invoca.

Aunque todas las instancias de la misma clase posean los mismos atributos, éstos pueden contener valores diferentes correspondientes a la naturaleza particular de cada instancia. La lista de atributos es retenida por la clase, mientras que las instancias poseen los valores de esos campos.

El mecanismo de constructores se hace considerablemente más poderoso cuando se combina con la capacidad de *sobrecargar funciones* (aunque como veremos, esta cualidad no es exclusiva de los constructores). Se dice que una función está sobrecargada cuando hay dos o más cuerpos de función conocidos por el mismo nombre. Se rompe la ambigüedad de las funciones sobrecargadas por las listas de parámetros. En nuestro caso, el constructor está sobrecargado, de forma que si escribimos

```
Complejo d;
   ...
d = new Complejo();
```

Se creará un número `Complejo` en el que tanto la parte real como la imaginaria valen 0. El hecho de elegir uno de los dos métodos se realiza en función de los parámetros. En la figura 1.2 se muestra esta situación esquemáticamente.

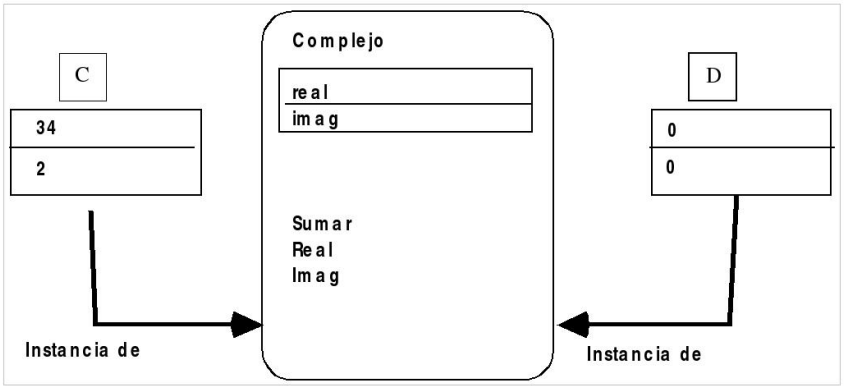

Figura 1.2: Ejemplo de instanciación.

Veamos cómo usaríamos la clase anteriormente presentada con alguna instrucción más. Algunas de las instrucciones que aparecen no las hemos explicado todavía pero no influyen en el objetivo del ejemplo:

```
Complejo c1 = new Complejo (5, 3);
Complejo c2 = new Complejo (2, 4);

System.out.println("El complejo" + c1 + " es: ");
System.out.println("               Parte Real: " + c1.real());
System.out.println("               Parte Imaginaria: " + c1.imag());
```

Físicamente, los atributos de una instancia se pueden considerar como variables (en algunos casos aparecen como variables de instancia); cada vez que el valor de un campo es modificado el nuevo valor se sustituye en la correspondiente instancia. Cada instancia tendrá su colección independiente de variables que sólo se podrán modificar usando los métodos dispuestos para ello (si se han declarado como privadas).

Existen otro tipo de atributos llamados atributos de clase (también llamados variables de clase) que se utilizan para definir campos que tienen un valor que es común a todas las instancias de una misma clase (en Java son los atributos *estáticos*). El valor de un atributo de clase reside en la propia clase y es compartido por todos los objetos de la citada clase.

Esta visión de objeto que acabamos de comentar tiene un término asociado de uso muy común en los diferentes paradigmas de programación que es la *encapsulación*.

El comportamiento general de un programa orientado a objeto se estructura en términos de responsabilidades. Es decir, se invocan métodos en objetos que tienen la responsabilidad de responder mediante la ejecución de los mismos.

Cuando en el ejemplo anterior escribíamos `c1.imag()` estábamos invocando el método `imag()` al objeto de tipo `Complejo` de referencia `c1` que provocaba la ejecución del método `imag()` definido en el mismo. Volvemos aquí al principio de ocultación de información en

cuanto a que el cliente que envía la petición no necesita conocer el medio real con el que ésta será atendida.

1.6. Herencia

Supongamos que estamos diseñando un programa orientado a objetos y nos damos cuenta de que aparece una *Clase B* que es idéntica a una *Clase A* excepto que tiene unos atributos y métodos más que ésta. Una manera muy sencilla y poco OO de solucionar la situación sería simplemente copiar el código de la *Clase A* en la *Clase B* y añadir las líneas de código propias de esta clase. El mecanismo ideal, sin embargo, es el concepto de herencia. En todo caso, sería un error pensar que el principal uso de la herencia es evitarnos escribir código, como veremos en el tema correspondiente, la herencia tiene que ver con la relación entre tipos de datos.

Una vez aceptada la filosofía básica de diseño OO, el siguiente paso para aprender a programar de forma OO implica adquirir conocimientos sobre cómo usar con efectividad clases organizadas bajo una estructura jerárquica basada en el concepto de herencia. Por *herencia* nos referimos a la propiedad por la que los ejemplares de una clase hija (subclase) pueden tener acceso tanto a los datos como al comportamiento (métodos) asociados con una clase paterna (superclase).

La herencia significa que el comportamiento y los datos asociados con las clases hijas son siempre una extensión de las propiedades asociadas con las clases paternas. Una subclase debe reunir todas las propiedades de la clase paterna y otras más. Esquemáticamente, la herencia la podemos representar como se muestra en la figura 1.3.

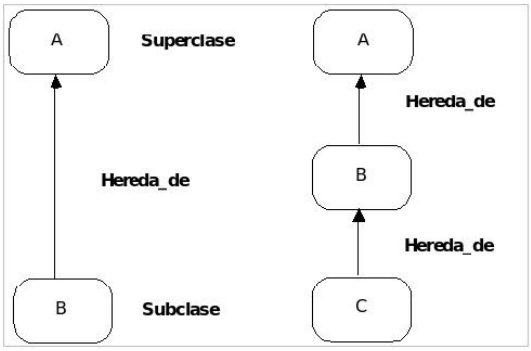

Figura 1.3: Esquema de herencia.

¿Qué beneficios nos aporta el uso de la herencia? Entre los muchos beneficios que se otorgan al uso de la herencia, los dos más relevantes en el punto en el que nos encontramos son:

- Reusabilidad del software: Cuando el comportamiento se hereda de otra clase, no necesita ser reescrito el código que proporciona ese comportamiento. Otros beneficios asociados a la reusabilidad son una mayor fiabilidad (cuanto más se use un código más pronto aparecerán los errores) y menor costo de mantenimiento gracias a la compartición de código.

- Consistencia de la interfaz: Cuando múltiples clases heredan de la misma superclase, nos aseguran que el comportamiento que heredan será el mismo en todos los casos. Es

más fácil garantizar que las interfaces para objetos similares sean en efecto similares y difieran lo menos posible.

Al principio es normal tener cierta inseguridad a la hora de reconocer las clases en nuestros sistemas y también el reconocer cuando una clase debe ser una subclase de otra. La regla más importante es que, para que una clase se relacione con otra por medio de la herencia, debe haber una relación de funcionalidad entre ambas. Al decir que una clase es una superclase de una subclase, estamos diciendo que la funcionalidad y los datos asociados con la clase hija (subclase) forman un superconjunto de la funcionalidad y los datos de la clase paterna (superclase).

El principio que dice que el conocimiento de una categoría más general es aplicable también a la categoría más específica se llama herencia. Por ejemplo, un estudiante se puede ver como una clase cuya categoría más general (superclase) es la clase Humano que a su vez tiene otra categoría más general (otra superclase) que son los mamíferos, formando una organización del conocimiento en forma de árbol, encontrándose en las hojas las instancias de las clases. Por ejemplo podemos encontrarnos con una estructura jerárquica como la mostrada en la figura 1.4 de una representación de una realidad.

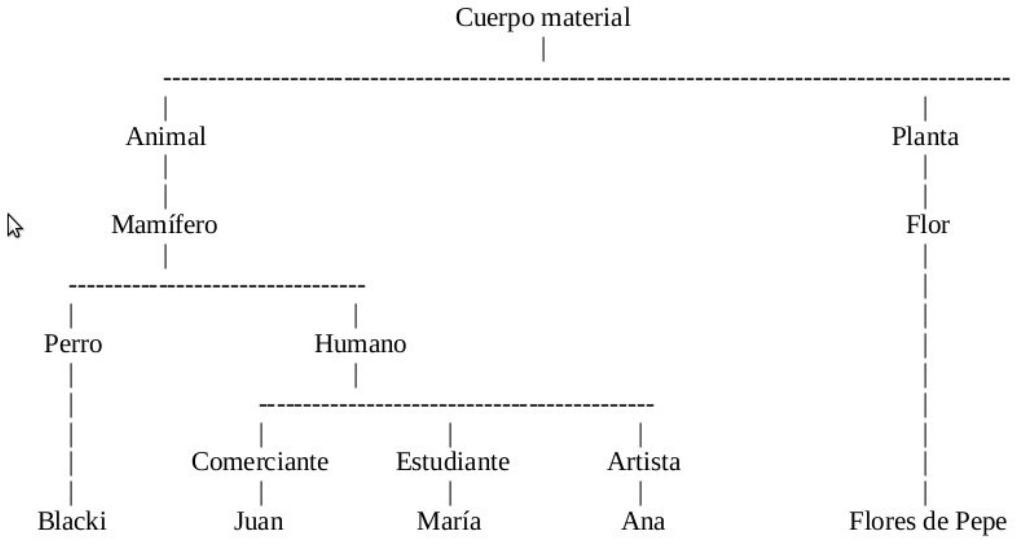

Figura 1.4: Ejemplo de una estructura jerárquica.

Dentro de este marco general hay lugar para variaciones en la forma en la que hacer uso de la herencia, la más habitual es la siguiente:

Especialización: Es el uso más común de la herencia. Si se consideran dos conceptos abstractos A y B, y tiene sentido el enunciado "A es un B", entonces es probablemente correcto hacer a A subclase de B. Por ejemplo, "un perro es un mamífero". Por otra parte, si tiene más sentido decir "A tiene un B" entonces quizá sea mejor hacer que los objetos de la clase B sean atributos de la clase A. Por ejemplo, "un número complejo tiene una parte imaginaria".

Por otro lado, existen dos enfoques alternativos en cuanto a cómo deben estructurarse las clases como un todo en los lenguajes OO. Un enfoque sostiene que todas las clases deben estar contenidas en una sola gran estructura de herencia (*Smalltalk, Objetive-C, Java*). La

ventaja de este enfoque es que la funcionalidad proporcionada en la raíz del árbol (clase `Object`) es heredada por todos los objetos. El otro enfoque sostiene que las clases que no están lógicamente relacionadas deben ser por completo distintas (*C++ y Object Pascal*).

1.7. Evolución histórica y diferencias entre la programación clásica y la POO

Iniciamos este punto de diferencias mediante la resolución de un sencillo ejemplo mediante la programación algorítmica clásica (en C) y mediante el paradigma orientado a objetos (en Java). No se trata en ningún caso de traducir una solución algorítmica clásica a una solución orientada a objetos. Resaltar una vez más que son maneras absolutamente distintas de ver una realidad.

El ejemplo es la representación de la estructura de datos para representar una lista enlazada y el subprograma necesario para insertar un nodo detrás de otro nodo dado:

```
struct reg {
  int px;
  struct reg *seg;
};

struct reg *lst ;

void insdetras (struct reg **l, int i) {
  struct reg *p;

  if (*l == NULL)
    inscab(&(*l),i);  // si está vacio se usa el insertar en cabeza
  else {
    p = (struct reg *) malloc(sizeof(struct reg)) ;
    p -> seg = (*l)->seg;
    p -> px = i ;
    (*l)-> seg = p ;
  }
  return;
}
```

Como estamos en un lenguaje procedural clásico, hemos tenido que definir por un lado la estructura de datos que define un nodo y por otro los procedimientos que sirven para trabajar con él. Se cumple la regla de estructuras de datos + algoritmos = programas.

En el caso de Java tendríamos que identificar los objetos de nuestro sistema, en este caso solo uno: el nodo.

```
class Nodo {
  int info;
  Nodo prox;

  Nodo (int i) {
    Info = i;
    prox = null;
  }

  public void InsDetras (Nodo e) {
```

```
      e.prox = prox;
      prox = e;
    }
  }
```

La diferencia podemos intuirla claramente con el ejemplo. En este caso, un nodo es una unidad auto-contenida en la que se define la estructura del nodo y las operaciones necesarias para trabajar con él.

A efectos de potenciar la idea de encapsulación y ocultación de la información se debería acceder a los atributos sólo a través de los métodos que se dispongan para ello. Para forzar esta situación, lo habitual es haber hecho que los atributos fueran privados a la clase:

```
class Nodo {
  private int info;
  private Nodo prox;

  Nodo (int i) {
    info = i;
    prox = null;
  }

  Nodo prox() {
    return prox;
  }

  Int info() {
    return info;
  }

  void ChangeInfo (Int i) {
    info = i;
  }

  void ChangeProx (Nodo p) {
    prox = p;
  }

  public void InsDetras (Nodo e) {
    e.ChangeProx (prox());
    ChangeProx(e);
  }
}
```

Desde un punto de vista histórico, la idea de la POO aparece por primera vez en *SmallTalk* (a principios de los 70), pero conceptos similares a los que se usan en la POO ya aparecían en *Simula* (1967). De cualquier forma se considera que *Smalltalk* fue el primer lenguaje orientado a objetos:

- Es uno de los lenguajes orientados a objetos más consistentes: todos sus tipos de datos son clases y todas sus operaciones son mensajes.
- No es adecuado para aplicaciones de gran envergadura: es ineficiente y no permite comprobación de tipos estática.

A mediados de los 80 aparecen lenguajes híbridos como son el *C++, Objetive-C, Object-Pascal* entre otros:

- Incluyen tipos de datos convencionales y clases.
- Incluyen procedimientos además de métodos.
- Permiten comprobaciones en tiempo de compilación (estáticas).
- Su eficiencia es equivalente a la de los convencionales.
- La transición de la programación algorítmica clásica a la POO es más fácil usando este tipo de lenguajes.

Existen muchas diferencias entre la programación convencional y la POO, aunque las más evidentes, es decir, las sintácticas son las menos importantes y en cambio, las diferencias más importantes son las menos evidentes. Nos vamos a centrar en este apartado en las menos evidentes, aunque a lo largo del curso vamos a intentar que se detecten con mayor claridad:

- La POO se centra en los datos en lugar de en los procedimientos.
- Especial énfasis en la reutilización:
  - El objetivo de los métodos de diseño tradicionales (diseño descendente) es encontrar una solución a medida para el problema específico. Los programas obtenidos son correctos y eficientes pero demasiado sensibles a cambios en los requisitos.
  - En la POO el objetivo no es ajustar las clases a los clientes que las vayan a usar, sino adaptarlas a su contexto y adaptar los clientes a las clases. Las clases deben ser más generales de lo necesario para una aplicación específica, pero el tiempo adicional de desarrollo se recupera con creces a medio plazo, cuando esas clases se reutilizan en otros programas para los que no se habían diseñado.
- Programación mediante extensión. POO significa normalmente extender software existente. Por ejemplo, se dispone de una serie de bibliotecas de clases como menús, ventanas, etc... y pueden extenderse para cumplir los requisitos de una aplicación específica.
- Estado distribuido y responsabilidades distribuidas:
  - En programas convencionales, el estado se almacena en las variables globales del programa principal. Aunque el programa principal suele invocar procedimientos, éstos no poseen estado, es decir, transforman datos de entrada en datos de salida pero no almacenan nada.
  - En POO el estado es distribuido entre múltiples objetos. Cada objeto posee su propio estado y un conjunto de procedimientos que trabajan sobre dicho estado.

La POO no es la solución a todos los problemas. Si somos conscientes de sus puntos fuertes y de sus debilidades y se usa de forma consciente, los beneficios compensan sobradamente los costes. Sin embargo, los costes pueden desbordarnos si las clases se usan de forma indiscriminada, particularmente en situaciones en las que no simplifican problemas, sino que añaden complejidad. Entre los beneficios más interesantes de la POO tenemos:

- Gestión de la complejidad: El principal problema de la programación es la complejidad. Para resolver la complejidad necesitamos mecanismos que nos permitan descomponer el problema en pequeños fragmentos fáciles de comprender por separado. Las clases son un buen mecanismo en esa línea porque:
  - Permiten la construcción de componentes manejables con interfaces simples, que abstraen los detalles de implementación.
    - Los datos y las operaciones forman una entidad que no está dispersa a lo largo del programa, como ocurre con las técnicas procedurales.
    - La localidad de código y datos facilita la legibilidad.
- La POO permite construir sistemas extensibles. Podemos conseguir que un sistema existente trabaje con nuevos componentes sin necesidad de modificación. Además, esos componentes se pueden añadir en tiempo de ejecución.
- La POO mejora la reutilización al permitir adaptar componentes a necesidades nuevas sin invalidar los clientes existentes.

Sin embargo, los beneficios vienen pagando una serie de costes, muchos de los cuales hacen que la POO no sea efectiva para muchas empresas y que siga estando relegada a la solución de problemas menores, aunque día a día va tomando mayor relevancia:

- Esfuerzo de aprendizaje: El cambio a la POO no es "conocer cuatro instrucciones nuevas". Se introducen conceptos importantísimos como clases, herencia y polimorfismo. La única manera de hacer efectivos todos estos conceptos es mediante la experiencia.
- Problemas de comprensión y flexibilidad. Las clases normalmente vienen en estructuras jerárquicas; si no se documenta y establece el porque de estas relaciones, al final las ventajas de establecer estas relaciones se convierte en una desventaja.
- Los dos problemas anteriores suelen llevar a soluciones que, en el mejor de los casos, son poco eficientes. Muchas veces se critica a los lenguajes orientados a objetos de ser poco eficaces, cuando en realidad lo que subyace es un problema de mal diseño y programación.

Capítulo 2

Introducción al lenguaje Java

2.1. Introducción

En este tema describiremos las características principales del lenguaje JAVA. En la sección 2.2, se trata la portabilidad de JAVA, uno de los principales motivos del éxito que ha tenido este lenguaje.

En las secciones siguientes, se tratan los fundamentos para escribir programas en JAVA:

- Normas generales de sintaxis: uso de comentarios, identificadores, separadores... (en la sección 2.3).
- Estructura básica de los diferentes tipos de programa (en la sección 2.4).
- Tipos de datos: tipos simples, objetos, vectores..., y operadores disponibles (en la sección 2.5).
- Estructuras de control: condicionales e iterativas (en la sección 2.6).

Finalmente, con el propósito de facilitar la comprensión de los conceptos descritos en este tema, se incluyen varios ejercicios resueltos (sección 2.7) y otros ejercicios propuestos (sección 2.8).

2.2. Portabilidad: la máquina virtual de Java

En JAVA, como en otros lenguajes, el código fuente ha de ser compilado y ejecutado, pero JAVA se definió buscando su independencia de la máquina donde se ejecutarían los programas. En la mayoría de los lenguajes, al compilar se genera un código máquina directamente ejecutable en una máquina determinada (y de uso limitado a ese tipo de máquina). En cambio, al compilar en JAVA, el código generado es independiente del hardware y no se puede ejecutar directamente en ninguna máquina, sino que ha de ser interpretado por una máquina virtual.

Al código generado por el compilador JAVA se le llama *bytecode*. Y la máquina que interpreta el *bytecode*, generando el código ejecutable para un procesador concreto, es la máquina virtual de JAVA o, en inglés, JVM (*Java Virtual Machine*). La portabilidad del código JAVA entre máquinas con distintos procesadores y/o sistemas operativos es posible si en esas máquinas se ha instalado la correspondiente versión de JVM. Entonces, el *bytecode* compilado en una máquina puede ser llevado a otra máquina diferente y ejecutado sin problemas.

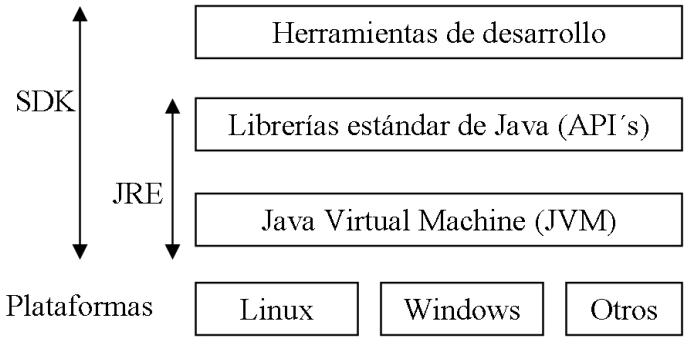

Figura 2.1: Plataforma JAVA 2

La JVM más el conjunto de clases necesarias para ejecutar los programas constituyen el entorno de ejecución de JAVA, o JRE (*Java Runtime Environment*). Para ejecutar código JAVA es suficiente con instalar el JRE. Si, además, se quiere escribir y compilar código fuente, se necesita el *kit* de desarrollo de JAVA, o SDK (*Software Development Kit*), que incluye el JRE.

En la figura 2.1 se muestra la jerarquía (simplificada) del software que constituye la plataforma JAVA 2. Se aprecia que la *Java Virtual Machine* es el nivel inmediato superior a los sistemas operativos, que el entorno JRE incluye la JVM y además todas las librerías estándar de JAVA (API's), y que el kit SDK incluye las herramientas de desarrollo.

Los programas o utilidades principales del SDK son las siguientes:

- *javac*. Es el compilador de JAVA. Traduce el código fuente a *bytecode*.
- *java*. Es el intérprete de JAVA. Ejecuta el código binario a partir del *bytecode*.
- *jdb*. Es el depurador de JAVA. Permite ejecución paso a paso, inserción de puntos de ruptura y rastreo de variables.
- *javadoc*. Es la herramienta para documentación automática de los programas.
- *appletviewer*. Otro intérprete de JAVA, específico para *applets* (aplicaciones gráficas para usar en navegadores *web*).

Estos programas se pueden ejecutar desde la línea de comandos del sistema operativo. Por ejemplo, si se ha escrito un programa en JAVA y el fichero que lo contiene se llama *MiPrograma.java*, se puede compilar mediante:

```
javac MiPrograma.java
```

Si el código fuente no contiene errores, se genera el correspondiente bytecode, y el programa se puede ejecutar mediante el siguiente comando, que invoca al método *main* (programa principal) de la clase *MiPrograma*:

```
java MiPrograma
```

Sin embargo, normalmente se dispondrá de herramientas visuales, kits de desarrollo con entorno de ventanas, y los programas se compilarán y ejecutarán accediendo a las correspondientes opciones de menú. En este texto, usaremos el entorno de desarrollo *JCreator*, disponible en www.jcreator.com.

2.3. Cuestiones sintácticas

En esta sección se describen los aspectos relativos a legibilidad y documentación del código (en las subsecciones de indentación y comentarios), las normas para definir identificadores, los usos de los caracteres separadores de código y, finalmente, se listan las palabras reservadas del lenguaje JAVA.

2.3.1. Indentación

Para separar las instrucciones del código fuente (por razones de legibilidad) se pueden usar espacios en blanco, tabuladores y saltos de línea.

Además, como en cualquier lenguaje de programación, se recomienda sangrar o indentar las líneas para facilitar la lectura del código e identificar rápidamente donde empiezan y terminan los bloques de código (las estructuras condicionales o iterativas que incluyan un cierto número de instrucciones o, a su vez, otros bloques de código).

El compilador no necesita que el código contenga estas separaciones, pero su uso es imprescindible para que el código sea comprensible para los programadores que lo lean.

2.3.2. Comentarios

Los comentarios en el código fuente son líneas que el compilador ignora y, por tanto, el programador puede escribir en el lenguaje que prefiera, que suele ser su propio lenguaje natural, por ejemplo, español, inglés, etc.

El texto incluido en los comentarios pretende facilitar la comprensión del código fuente a otros programadores (o a su mismo autor, tiempo después de haber terminado el desarrollo del programa). Se trata, pues, de explicaciones o aclaraciones a su funcionamiento y constituyen una documentación del programa.

Al igual que otros lenguajes, JAVA dispone de indicadores de comentario de una sola línea y de múltiples líneas:

- Comentarios de una línea. Si es de línea completa, ésta debe iniciarse con //. Si sólo parte de la línea es comentario, el separador // delimitará la frontera entre lo que es código a compilar (a la izquierda del separador) y lo que es comentario (a la derecha del separador y hasta el final de la línea).

```
// Programa principal que escribe "Hola" en la consola
public static void main (String[] args) {
  System.out.println( "Hola" ); // escribir "Hola"
}
```

- Comentarios de varias líneas. Deben escribirse dentro de los separadores /* (para marcar el inicio) y */ (para marcar el final del comentario).

```
/* El siguiente programa principal
   no tiene ninguna instrucción,
   y por tanto, no hace nada cuando se ejecuta */
```

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

Java, además, proporciona un tercer tipo de comentario, similar al segundo (/* ... */), que se denomina comentario de documentación (/** ... */). Java dispone de una herramienta de documentación, javadoc, que extrae este tipo de comentarios para generar automáticamente documentación de referencia de los programas. Se puede obtener toda la información relativa a esta herramienta en la siguiente página web: http://java.sun.com/j2se/javadoc.

2.3.3. Identificadores

Al escribir código fuente, los programadores tienen que dar nombres a diferentes elementos del programa: variables, constantes, objetos, métodos... Para ello se usan identificadores.

En Java, los identificadores se forman con letras mayúsculas y minúsculas (Java las distingue), dígitos y también los caracteres dólar y guión bajo, $_. Un identificador válido debe comenzar siempre con una letra, y no puede coincidir con una palabra reservada del lenguaje.

2.3.4. Separadores

En los lenguajes de programación existen caracteres separadores del código, como los paréntesis o las llaves, con funciones bien definidas.

En la tabla 2.1 se enumeran los separadores de Java y para qué se usan. Si se ha programado en otros lenguajes, como C, la mayoría de estos usos serán conocidos. Sin embargo, algunos usos de los separadores no se comprenderán en este momento. Ya se irán viendo a lo largo de este tema y los siguientes.

2.3.5. Palabras reservadas

Las palabras reservadas del lenguaje Java se listan en la tabla 2.2. El uso de la mayoría de estas palabras reservadas se irá viendo a lo largo del presente texto.

2.4. Estructura básica de un programa en Java

En esta sección se presentan los tres tipos de programas que se pueden escribir en Java: programas de consola, programas gráficos y *applets*. Se muestra un ejemplo muy sencillo de cada uno de estos tipos de programa.

En Java, las aplicaciones gráficas, en entorno de ventanas, pueden ejecutarse como programas independientes o como applets, programas integrados en un navegador web. En el último tema de este texto se tratarán los applets en Java.

2.4.1. Programa de consola

Se llama programas de consola a los programas que funcionan en modo texto, es decir, programas donde la entrada de datos y la salida de resultados se realizan mediante sucesivas líneas de texto.

Para ilustrar la estructura básica de un programa de consola, consideraremos el siguiente sencillo ejemplo:

Símbolo	*Nombre*	*Uso*
( )	paréntesis	• Delimitar la lista de argumentos formales en la definición de los métodos. • Delimitar la lista de argumentos actuales en las llamadas a métodos. • Establecer cambios de precedencia de operadores, en expresiones que incluyan varios operadores. • Realizar conversiones de tipo (*casting*). • Delimitar partes en las estructuras de control (por ejemplo, en las condiciones lógicas de las instrucciones *if* y *while*).
{ }	llaves	• Delimitar bloques de código (en las estructuras condicionales, en las iterativas, y en los métodos completos). • Dar valores iniciales a variables de tipo vector.
[ ]	corchetes	• Declaración de vectores. • Referencias a los elementos de los vectores.
;	punto y coma	• Separador de instrucciones.
,	coma	• Separar identificadores (por ejemplo, en una declaración de variables de un mismo tipo, o en una asignación de valores a un vector). • Separar los argumentos formales y actuales de los métodos.
.	punto	• Separador de paquetes, subpaquetes y clases. • Referenciar los atributos y métodos de los objetos.

Tabla 2.1: Separadores

abstract	boolean	break	byte	byvalue	case
cast	catch	char	class	const	continue
default	do	double	else	extends	false
final	finally	float	for	future	generic
goto	if	implements	import	inner	instanceof
int	interface	long	native	new	null
operator	outer	package	private	protected	public
rest	return	short	static	super	switch
synchronized	this	throw	throws	transient	true
try	var	void	volatile	while	

Tabla 2.2: Palabras reservadas

```
// HolaMundo.java (nombre del fichero de código fuente)
public class HolaMundo {
  public static void main (String[] args) {
    System.out.println( "Hola Mundo" );
  }
}
```

Se trata de un programa principal que se limita a mostrar por la consola un mensaje de saludo. En la figura 2.2 se muestra una captura de pantalla del entorno de desarrollo *JCreator* donde se aprecia la ventana de edición del código fuente y la ventana que muestra la salida obtenida al ejecutar el programa.

En JAVA cualquier programa forma parte de una clase. Las clases constan de datos (llamados atributos) que guardan la información y de funciones (llamadas métodos) que manipulan la información.

En este ejemplo, la clase (palabra reservada **class**) se llama *HolaMundo* y contiene el programa principal (palabra reservada **main**).

En el ejemplo, como la clase se ha identificado como *HolaMundo*, el código fuente se guarda en un fichero de nombre *HolaMundo.java*. Un fichero de código fuente (fichero con la **extensión java**) puede contener el código de una o varias clases:

- Si el fichero contiene sólo una clase, su nombre debe dar nombre al fichero y, cuando se compile, se generará un fichero con el mismo nombre y con la **extensión class** (en este caso, *HolaMundo.class*).

- Si el fichero contiene varias clases, una de ellas contendrá el programa principal *main* (que es una función o método dentro de esa clase) y su nombre será el nombre del fichero. Cuando se compile un fichero *java* con varias clases, se generarán varios ficheros *class*, uno por cada una de las clases existentes en el fichero *java*.

La estructura del programa principal (método *main*) del ejemplo puede servirnos para comentar algunos aspectos de sintaxis generales para cualquier método escrito en JAVA:

- Presencia de calificadores antes del nombre del método (en el ejemplo, aparecen tres palabras reservadas: **public**, **static** y **void**).Veremos el significado de estas palabras a lo largo del texto. Sólo anticiparemos que cuando el método no devuelve nada (como sucede aquí), se indica mediante la palabra reservada *void*.

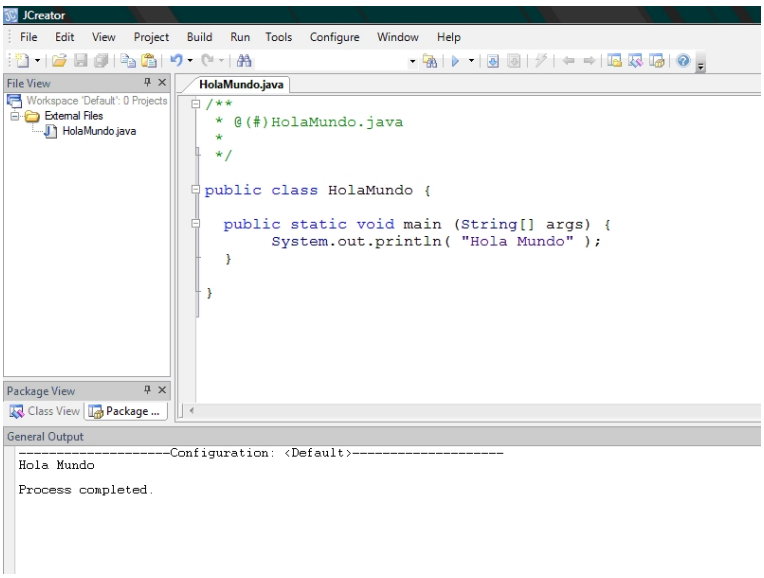

Figura 2.2: Programa de consola

- Presencia de la lista de parámetros o argumentos (son los datos que se suministran a un método en su llamada, al ser invocado). En el ejemplo, la lista es: `(String[] args)`, es decir, hay un parámetro, que es un *array* de objetos de la clase **String**.
- El bloque de código es el conjunto de instrucciones encerradas entre las correspondientes llaves. En este caso, sólo hay una instrucción:

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

  Esta instrucción es una llamada al método **println**. Este método permite mostrar en la consola los datos que se le pasen como argumento. En este caso, la cadena de texto "Hola Mundo".

2.4.2. Programa gráfico

Para ilustrar la estructura básica de un programa gráfico, de ventanas, consideraremos el siguiente sencillo ejemplo:

```
public class Grafico {
  public static void main (String[] args) {
    // Crear ventana
    Frame ventana = new Frame();
    // Crear área de texto
    TextArea texto = new TextArea( "Hola Mundo" );
    // Añadir área de texto a la ventana
    ventana.add(texto);
    // Mostrar ventana
    ventana.setVisible(true);
  }
}
```

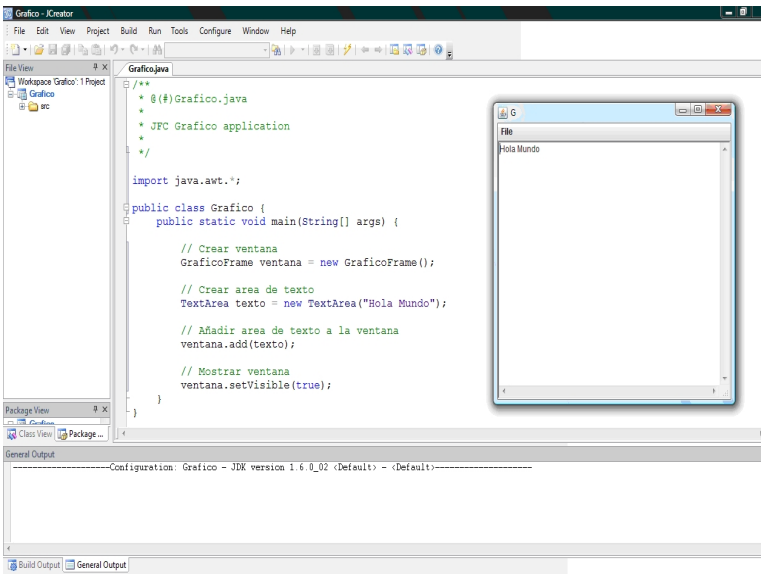

Figura 2.3: Programa gráfico

En este ejemplo, el programa principal crea una ventana y muestra ahí un mensaje de saludo. En la figura 2.3 se muestra una captura de pantalla donde se aprecia la ventana de edición del código fuente, y (superpuesta, a la derecha) la ventana generada al ejecutar el programa.

El programa principal contiene cuatro instrucciones, que realizan las siguientes cuatro acciones:

- Se crea un objeto (de la clase **Frame**), llamado `ventana`, con lo que se crea la ventana de la aplicación gráfica.
- Se crea un objeto (de la clase **TextArea**), llamado `texto`, y con el valor "Hola Mundo".
- Se añade el texto a la ventana.
- Se activa la visibilidad de la ventana.

En las clases de estas aplicaciones gráficas se incluirá todo el código necesario para que las ventanas contengan todos los elementos (menús, paneles gráficos, etc.) que se desee.

2.4.3. Applet

Un *applet* es un caso particular de aplicación gráfica de JAVA, pero que se ejecuta integrada en una página web (y es gestionado por el navegador web), en lugar de ejecutarse como aplicación independiente.

Para ilustrar la estructura básica de un applet, consideraremos el siguiente sencillo ejemplo:

```
public class AppletBasico extends Applet {
  public void paint (Graphics g) {
    g.drawString( "Hola Mundo" , 50, 50);
```

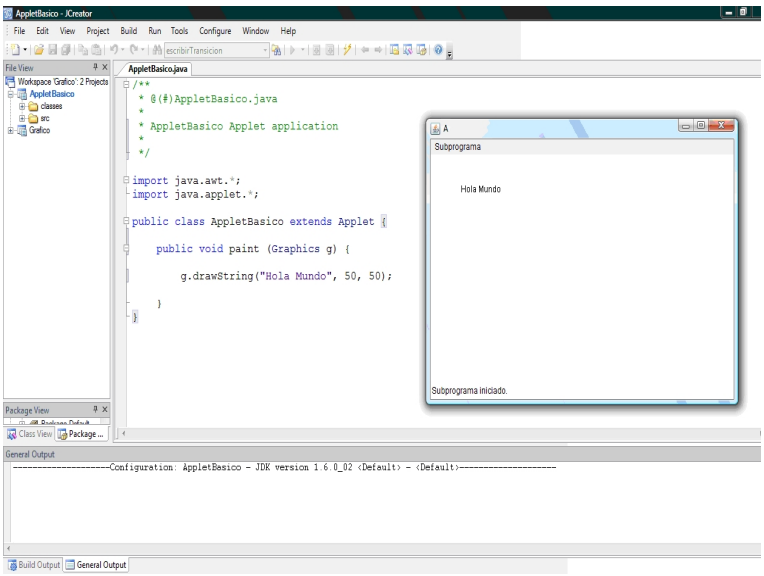

Figura 2.4: Applet

```
        }
    }
```

En este ejemplo, el applet muestra un mensaje de saludo en su ventana. En la figura 2.4 se muestra una captura de pantalla donde se aprecia la ventana de edición del código fuente, y (superpuesta, a la derecha) la ventana generada al ejecutar el programa.

Los applets se describirán en detalle en el último tema. Aquí sólo anticiparemos que los applets carecen de programa principal (método *main*) y disponen de una serie de métodos (**init, start, stop, destroy, paint**) para regular su interacción con el navegador web.

En el ejemplo, el método *paint* contiene una sola instrucción, la llamada al método **drawString**, que muestra la cadena de caracteres "Hola Mundo" en la posición de la ventana cuyas coordenadas son (50,50).

2.5. Tipos de datos

En esta sección, trataremos los diferentes tipos de datos (tipos simples, objetos, vectores) disponibles en Java, así como los operadores existentes para cada tipo y las reglas de conversión entre tipos diferentes.

Java, como la mayoría de los lenguajes de programación, dispone de tipos de datos para definir la naturaleza de la información que manipulan los programas. Para codificar la información más sencilla (como números enteros, números reales, caracteres, etc.), se dispone de los llamados tipos de datos simples.

Además, Java es un lenguaje orientado a objetos, lo que implica que existen tipos de datos objeto. Por tanto, los datos con los que operan los programas en Java también pueden almacenarse en entidades más complejas, llamadas objetos, y dichos objetos deben crearse como instancias de clases. En Java se hablará de objetos de una determinada clase de forma

tipo	*valores*	*codificación / tamaño*
byte	números enteros	8 bits
short	números enteros	16 bits
int	números enteros	32 bits
long	números enteros	64 bits
float	números reales	IEEE-754, de 32 bits
double	números reales	IEEE-754, de 64 bits
char	caracteres	Unicode, de 16 bits
boolean	lógicos: *true* / *false*	Dependiente de la JVM

Tabla 2.3: Tipos de datos simples

similar a como se habla de variables de un determinado tipo. El concepto de clase y objeto se estudiará en profundidad en el capítulo 3.

JAVA es un lenguaje fuertemente tipado, y esta característica es una de las razones de su seguridad y robustez. Cada variable tiene un tipo (o pertenece a una clase de objetos). Cada expresión tiene un tipo. Cada tipo o clase está definida estrictamente. En todas las asignaciones se comprueba la compatibilidad de tipos o clases. Sin embargo, como veremos en esta sección, se puede operar con datos de tipos diferentes siempre que se respeten determinadas reglas de conversión.

2.5.1. Tipos de datos simples

Los tipos primitivos disponibles en JAVA, los valores que pueden representar y su tamaño, se muestran en la tabla 2.3.

Para estos tipos de datos más simples o primitivos, JAVA proporciona un mecanismo sencillo para declarar variables: especificar el tipo de la variable y su nombre (identificador elegido para la variable). Por ejemplo, si en un programa se necesitan tres variables, una de tipo entero que llamaremos `contador` y las otras de tipo real y de nombres `dato` y `resultado`, se pueden declarar mediante:

```
int contador;
double dato, resultado;
```

2.5.2. Tipos de datos objeto

Cuando se escriben programas en JAVA, se pueden usar variables que sean objetos de determinadas clases (sea porque el mismo programador haya escrito el código de esas clases, sea porque dichas clases estén disponibles en librerías públicas). Es decir, se pueden declarar objetos de clases definidas por el programador o de clases estándar (ya definidas en librerías del lenguaje JAVA).

Los objetos declarados en nuestros programas pueden verse como variables del programa, al igual que las variables de los tipos simples. Las principales diferencias son las siguientes:

- Los objetos no suelen almacenar un solo dato simple, sino varios ítems de información (que llamamos atributos).
- Los objetos disponen además de una serie de funciones (que llamamos métodos) para manipular la información que almacenan.

Los objetos de cualquier clase se declaran de manera similar a las variables de tipos simples. Por ejemplo, si en un programa se necesitan tres objetos, uno de la clase **Date** que llamaremos `ayer` y dos de la clase *String* y de nombres `unaPalabra` y `otraPalabra`, se pueden declarar mediante:

```
Date ayer;
String unaPalabra, otraPalabra;
```

Sin embargo, y a diferencia de las variables de tipo simple, los objetos han de ser instanciados (es decir, debe reservarse explícitamente la memoria necesaria para guardar la información que contienen).

Los objetos se instancian mediante el operador **new** y la invocación de un método especial de la clase llamado constructor (método que tiene el mismo nombre de la clase). Por ejemplo, con la siguiente instrucción se construiría un objeto llamado `hoy` de la clase *Date*:

```
Date hoy = new Date();
```

El objeto ayer está declarado, el objeto hoy está declarado e instanciado.

Todo lo relativo a los tipos de dato objeto será tratado en profundidad en el siguiente tema.

2.5.3. Variables y constantes

Según las características del problema a resolver, el programador necesitará declarar ítems de información cuyo valor varíe durante la ejecución del programa y también ítems cuyo valor se mantenga constante. Por ejemplo, en un programa que calcule perímetros y áreas de círculos, se podrían declarar *radio*, *perímetro* y *área* como ítems variables, y Π, el número *pi*, como ítem constante.

Como ya se ha indicado, las variables se declaran precisando su tipo y eligiendo un nombre para las mismas. En el momento de la declaración de una variable, se puede asignar su valor inicial. Por ejemplo:

```
int contador = 0;  // variable entera con valor inicial
int i, j, k;       // 3 variables enteras, sin inicializar
```

Las constantes (también llamadas variables finales) se declaran de forma parecida, pero indicando que son constantes mediante la palabra reservada **final**, y asignándoles valores. Por ejemplo:

```
final int N = 100;           // constante entera, de valor 100
final char ULTIMA = 'Z';     // constante carácter, de valor Z
final double PI = 3.141592;  // constante real, de valor $\Pi$
```

Valores de las variables y constantes

Tanto para definir las constantes como para asignar valores a las variables, se debe tener en cuenta las formas válidas de especificar sus valores, según su tipo de dato. La tabla 2.4 resume las normas y muestra ejemplos para los tipos simples y para la clase *String*.

Tipo	*Norma*	*Ejemplos*

int	Valores en base decimal (la cifra tal cual), en octal (la cifra precedida de 0), o en hexadecimal (la cifra precedida de 0x o de 0X).	`int m = 75;`
		`int p = 024;`
		`int q = 0x36;`

long	Añadir a la cifra la letra L, mayúscula o minúscula.	`long r = 22L;` `long s = 0x33L;`
float	Valores en coma fija o en coma flotante. Añadir a la cifra la letra F, mayúscula o minúscula.	`float x = 82.5f;`
double	Valores en coma fija o en coma flotante. Añadir a la cifra la letra D, mayúscula o minúscula, o sin añadirla.	`double y = 3.24e1d;` `double z = 2.75e-2;`
boolean	Sólo dos valores posibles: *true* y *false*.	`boolean verdad = true;`

char	• Caracteres visibles: Escribir el carácter entre comillas simples (‘b’, ‘7’, ‘$’).
	• Caracteres especiales y no visibles: Usar las comillas simples con secuencia de escape (‘\”,‘\n’,. . .). O, también, usar la secuencia de escape seguida del código octal o hexadecimal que corresponda al carácter.
		`char letra = ‘A’;`
		
		`// salto de línea:`
		`char salto = ‘\n’;`
		
		`// á (código en octal):`
		`char a1 = ‘\141’;`
		
		`// á (en hexadecimal):`
		`char a2 = ‘\u0061’;`

String	Usar comillas dobles.	`String modelo = "HAL 9000";`

Tabla 2.4: Valores para variables y constantes

Ámbito de variables y constantes

Las variables tienen un ámbito (su lugar de existencia en el código) y un tiempo de vida asociado a la ejecución de las instrucciones de ese ámbito. Un bloque de instrucciones entre llaves define un ámbito. Las variables declaradas en un ámbito sólo son visibles en él. Se crean y destruyen durante la ejecución de las instrucciones de dicho ámbito.

Considérese el siguiente fragmento de código:

```
{                 // inicio del 1$^{er}$ ámbito
  int x1 = 10;    // x1 existe en el 1$^{er}$ ámbito
  {               // inicio del 2º ámbito
    int x2 = 20;  // x2 existe en el 2º ámbito
    x1 = x2 * 2;
  }               // final del 2º ámbito
  x2 = 100;       // incorrecto, x2 no existe en el 1$^{er}$ ámbito
  x1++;
}                 // final del 1$^{er}$ ámbito
```

Este código contiene un error. Se han definido dos ámbitos, y un ámbito está anidado en el otro. La variable `x1` se ha declarado en el ámbito externo y existe también en el ámbito interno (que es parte del ámbito externo). En cambio, la variable `x2` se ha declarado en el ámbito interno y deja de existir cuando la ejecución abandone ese ámbito. Por ello, la asignación a dicha variable fuera de su ámbito genera un error de compilación.

Un programa que declara y muestra variables

Como resumen de lo tratado en esta sección, considérese el siguiente programa principal:

```
public static void main (String[] args) {
  final double PI = 3.141592;
  int contador = 0;
  boolean verdad = true;
  char letra = 'A';
  String modelo = "HAL 9000" ;
  System.out.println( "PI = " + PI );
  System.out.println( "contador = " + contador );
  System.out.println( "verdad = " + verdad );
  System.out.println( "letra = " + letra );
  System.out.println( "modelo = " + modelo );
}
```

Este programa declara e inicializa variables de diferentes tipos y, luego, muestra en la consola los nombres y valores de las variables. En concreto, se obtiene la siguiente salida:

```
PI = 3.141592
contador = 0
verdad = true
letra = A
modelo = HAL 9000
```

En este programa, además, se ha usado el operador + como operador que concatena cadenas de caracteres (y no como operador aritmético). Así, por ejemplo, en la instrucción: `System.out.println( "PI = " + PI);` se ha unido la cadena "PI = " con el valor de la constante Π. El resultado de la concatenación es el dato que recibe el método *println* y, por tanto, lo que se muestra en la salida del programa.

operación	*operador normal*	*operador reducido*	*operador auto-*
suma	+	+ =	++
resta	−	− =	− −
multiplicación	*	* =	
división (cociente)	/	/ =	
módulo (resto)	%	% =	

Tabla 2.5: Operadores aritméticos

2.5.4. Operadores

JAVA proporciona los operadores básicos para manipular datos de los tipos simples. Así, se dispone de operadores aritméticos, operadores de bits, operadores relacionales, operadores lógicos (booleanos) y un operador ternario. En esta sección se enumeran y describen los mismos.

Operadores aritméticos

Para datos de tipo entero y de tipo real, se dispone de operadores aritméticos de suma, resta, multiplicación, división y módulo (obtener el resto de la división entera). De todos ellos, existe una versión reducida, donde se combina el operador aritmético y la asignación, que se puede usar cuando el resultado de la operación se almacena en uno de los operandos. Por último, existen los operadores de autoincremento y autodecremento para sumar y restar una unidad.

En la tabla 2.5 se muestran todos los operadores aritméticos.

El siguiente bloque de código ilustra el uso de los operadores aritméticos. En los comentarios de línea se ha indicado el valor de la variable modificada después de ejecutar la instrucción de cada línea.

```
{
  int i=2, j, k;
  j = i * 3;          // resultado: j = 2*3 = 6
  i++;                // resultado: i = 2+1 = 3
  j *= i;             // resultado: j = 6*3 = 18
  k = j \% i;         // resultado: k = 18\%3 = 0
  i++;                // resultado: i = 3+1 = 4
  j /= i;             // resultado: j = 18/4 = 4
  double x=3.5, y, z;
  y = x + 4.2;        // resultado: y = 3.5+4.2 = 7.7
  x--;                // resultado: x = 3.5-1.0 = 2.5
  z = y / x;          // resultado: z = 7.7/2.5 = 3.08
}
```

Operadores a nivel de bit

Los operadores a nivel de bit realizan operaciones con datos enteros, manipulando individualmente sus bits. Existe un operador unario para la negación (inversión bit a bit). Los demás son operadores binarios (requieren dos operandos) para realizar la conjunción, la disyunción, la disyunción exclusiva o el desplazamiento de bits a la izquierda o a la derecha,

operación	*operador normal*	*operador reducido*
negación (NOT)	~	
conjunción (AND)	&	&=
disyunción (OR)	|	|=
disyunción exclusiva (XOR)	^	^=
desplazamiento de bits a la izquierda	<<	<<=
desplazamiento de bits a la derecha con signo	>>	>>=
desplazamiento de bits a la derecha sin signo	>>>	>>>=

Tabla 2.6: Operadores a nivel de bit

operación	*operador*
operandos iguales	==
operandos distintos	!=
1^{er} operando mayor que 2^o operando	>
1^{er} operando menor que 2^o operando	<
1^{er} operando mayor o igual que 2^o operando	>=
1^{er} operando menor o igual que 2^o operando	<=

Tabla 2.7: Operadores relacionales

con signo o sin signo. También existe la posibilidad de combinar las operaciones bit a bit y la asignación (operadores reducidos).

En la tabla 2.6 se muestran los operadores a nivel de bit.

El siguiente bloque de código ilustra el uso de los operadores bit a bit. En los comentarios de línea se ha indicado el valor de cada variable, en binario, después de ejecutarse la instrucción de cada línea.

```
{
  int a = 18, b = 14, c;  // a = 10010 , b = 1110
  c = a & b;              // c = 10
  c = a | b;              // c = 11110
  c = a ^ b;              // c = 11100
  c <<= 2;                // c = 1110000
  c >>= 3;                // c = 1110
}
```

Operadores relacionales

En Java, como en otros lenguajes de programación, los operadores relacionales sirven para comparar dos valores de variables de un mismo tipo y el resultado de la comparación es un valor de tipo booleano.

En la tabla 2.7 se muestran los operadores relacionales.

Considérese el siguiente bloque de código como ejemplo del uso de los operadores relacionales:

```
{
  int a = 18, b = 14;
```

operación	*operador normal*	*operador reducido*
negación (NOT)	!	
conjunción (AND)	& &&	&=
disyunción (OR)	| ||	|=
disyunción exclusiva (XOR)	^	^=
igualdad	==	
desigualdad	!=	

Tabla 2.8: Operadores lógicos

a	*b*	*!a*	*a & b*	*a | b*	*a ^ b*	*a == b*	*a != b*
false	false	true	false	false	false	true	false
false	true	true	false	true	true	false	true
true	false	false	false	true	true	false	true
true	true	false	true	true	false	true	false

Tabla 2.9: Tablas de verdad de los operadores lógicos

```
    System.out.println( "a==b : " + (a==b) );
    System.out.println( "a!=b : " + (a!=b) );
    System.out.println( "a>b  : " + (a>b)  );
    System.out.println( "a<=b : " + (a<=b) );
  }
```

Al ejecutarse este bloque, se genera la siguiente salida:

```
a==b : false
a!=b : true
a>b  : true
a<=b : false
```

Operadores lógicos

Los operadores lógicos (booleanos) proporcionan las habituales operaciones lógicas (negación, conjunción, disyunción...) entre variables de tipo booleano o entre expresiones que se evalúen a booleano (verdadero o falso).

En la tabla 2.8 se muestran los operadores lógicos. En la tabla 2.9 se muestran sus tablas de verdad.

Todos los operadores lógicos son binarios, excepto la negación (que es unario, y se escribe delante de la variable o expresión a negar).

En el caso de las operaciones AND y OR, existen dos versiones del operador:

- Si se usan los símbolos &, |, siempre se han de evaluar los dos operandos para determinar el resultado de la operación.
- Si se usan los símbolos &&, ||, puede conocerse el resultado según el valor del primer operando (y sin calcular o evaluar el segundo operando). En la tabla 2.10 se muestra el comportamiento de estos operadores. Se ha indicado con "???" cuando no es necesario evaluar el 2º operando de la expresión lógica.

a	*b*	*a && b*	*a*	*b*	*a || b*
false	???	false	false	false	false
true	false	false	false	true	true
true	true	true	true	???	true

Tabla 2.10: Tablas de verdad de los operadores && y ||

a	*a ? b : c;*
false	c
true	b

Tabla 2.11: Operador ternario ?:

Operador ternario

El operador ternario ?: tiene la siguiente sintaxis:

a ? b : c;

Donde *a* es una expresión booleana, y *b* y *c* son valores o expresiones del mismo tipo. El operador ternario devuelve un valor (y, por tanto, puede usarse en una asignación). El valor devuelto será *b* si *a* se evalúa a verdadero, o será *c* si *a* se evalúa a falso, tal como indica la tabla 2.11.

Considérese el siguiente bloque de código como ejemplo de su uso:

```
{
  int a = 18, b = 14, c = 22, maximo;
  maximo = a>b ? a : b;               // maximo = 18
  maximo = maximo>c ? maximo : c;     // maximo = 22
}
```

El ejemplo calcula el mayor valor de tres variables enteras y el resultado se almacena en la variable `maximo`. Para ello se usa el operador ternario dos veces. La primera vez se determina el máximo de las dos primeras variables. La segunda vez se determina el máximo de las tres variables, comparando el valor provisional del máximo con el valor de la última variable.

Para ampliar la descripción del operador ternario y, además, la explicación del operador &&, consideremos el siguiente bloque de código:

```
{
  int a = 18, b = 14, c = 22;
  String s = "";
  s = ( a>b && a>c ) ? "maximo = " +a : s+"";
  s = ( b>a && b>c ) ? "maximo = " +b : s+"";
  s = ( c>a && c>b ) ? "maximo = " +c : s+"";
  System.out.println(s);
}
```

Este segundo ejemplo también calcula el mayor valor de tres variables enteras y el resultado lo muestra mediante un mensaje en la consola.

En esta ocasión, el resultado del operador ternario es una cadena de caracteres, que se almacena en el objeto `s` de la clase *String*. Cuando la expresión booleana del operador se evalúa a falso, no se añade nada a `s` (puesto que `s+"";` es concatenar la cadena vacía). Pero cuando se evalúa a verdadero, se asigna a `s` la cadena `"maximo = "+c`.

En cuanto a la evaluación de las expresiones booleanas, hay que indicar que:

- En la expresión `( a>b && a>c )` se evalúa `(a>b)` a verdadero y, entonces, es preciso evaluar `(a>c)`, que es falso, siendo falso el resultado de la expresión completa.
- En la expresión `( b>a && b>c )` se evalúa `(b>a)` a falso y, entonces, es falso el resultado de la expresión completa, sin necesidad de evaluar `(b>c)`.
- En la expresión `( c>a && c>b )` se evalúa `(c>a)` a verdadero y, entonces, es preciso evaluar `(c>b)`, que es verdadero, siendo verdadero el resultado de la expresión completa.

Por tanto, usando el operador &&, se han evaluado sólo cinco de las seis comparaciones. En cambio, si se hubiera usado el operador & se habrían evaluado todas las comparaciones. En consecuencia, los operadores && y || pueden ser más eficientes que & y |, si el programador elige con cuidado el orden de los componentes de una expresión booleana compleja. Además, los operadores && y || proporcionan mayor seguridad frente a errores en ejecución.

2.5.5. Conversión de tipos

Existen situaciones en las que se realiza una conversión del tipo de un dato para asignarlo a otro dato. Según los casos, puede tratarse de una conversión automática (implícita, hecha por el compilador) o explícita, si la tiene que indicar el programador.

La conversión automática de tipos (**promoción**) se hace cuando los dos tipos implicados en una asignación son compatibles y el tipo destino tiene mayor capacidad de almacenamiento que el tipo origen (valor o expresión evaluada en el lado derecho de la asignación). Por ejemplo, en una asignación de un valor *byte* a una variable *int*:

```
{
  int a;
  byte b = 7;
  a = b;
}
```

Si no se da la anterior situación, y se requiere una conversión, el programador debe indicarlo explícitamente mediante una proyección o **casting**. La conversión se indica anteponiendo el nombre del tipo, entre paréntesis, a la expresión a convertir. Así, por ejemplo, en una asignación de una expresión de tipo *int* a una variable *byte*:

```
{
  int a = 7, c = 5;
  byte b;
  b = (byte) (a + c);
}
```

El valor de la suma de las variables `a` y `c` es de tipo *int*, y se convierte a tipo *byte* antes de asignarlo a la variable `b`. Si no se hiciera el *casting* del resultado (es decir, si la instrucción fuera: `b = a + c`), el compilador genera el siguiente error:

```
possible loss of precision
found   : int
required: byte
```

También se generaría el mismo error si la instrucción fuera: `b = (byte) a + c`, ya que la conversión es más prioritaria que la suma, y en consecuencia se sumaría un valor *byte* y un valor *int*. Por tanto, son necesarios los paréntesis que encierran la expresión a convertir.

Consideremos otro ejemplo donde se usa el *casting*:

```
{
  char ch1 = 'a';
  char ch2 = (char) ((int)ch1 + 3);
  System.out.println(""+ch2+ch1);
}
```

Al ejecutarse el código de este ejemplo, se asigna el valor 'dá la variable `ch2`. Por ello, mediante *println*, se escribe en la consola la cadena "da".

Obsérvese la expresión que asigna valor a `ch2`. En el lenguaje C se podría haber escrito simplemente: `ch2 = ch1 + 3;` y la expresión se hubiera evaluado usando el código ASCII de la letra 'a'para obtener el de la letra 'd'. En JAVA, sin embargo, el compilador rechaza esa instrucción por incompatibilidad entre los tipos, y para llevarla a cabo son necesarias dos operaciones explícitas de conversión de tipos:

- *casting* de `ch1` a *int*, para poder sumarlo con el número 3.
- *casting* del resultado de la suma a *char*, para poder asignarlo a la variable `ch2`.

La conversión también se puede realizar con tipos no primitivos (clases). Este caso, que tiene implicaciones de gran trascendencia en la programación orientada a objetos, se verá con detalle en capítulos posteriores.

2.5.6. Vectores

Los vectores o *arrays* son estructuras de datos que almacenan un número fijo de elementos de información, siendo estos elementos del mismo tipo (o de la misma clase, si se trata de objetos).

Vectores unidimensionales

La declaración de vectores en JAVA tiene la siguiente sintaxis:

identificadorTipo [] nombreVector;

Donde:

- *identificadorTipo* ha de ser el nombre de un tipo conocido.
- *nombreVector* es el identificador o nombre que se le da al vector.
- Los elementos del vector son del tipo *identificadorTipo*.
- Los corchetes indican que se declara un vector, y no una variable de un solo elemento.

Por ejemplo, una declaración válida de un vector de enteros:

```
int [] diasMeses;
```

Existe una segunda forma de declarar vectores. En esta alternativa, los corchetes se sitúan después del nombre del vector: *identificadorTipo nombreVector []*;. Sin embargo, en este texto no usaremos esta segunda forma, por homogeneidad del código.

Para usar variables vector son necesarias dos acciones: la declaración y la instanciación de los vectores.

En la declaración de una variable vector simplemente se define una referencia, un nombre para el vector, pero no se reserva espacio de almacenamiento para sus elementos y, por tanto, no es obligatorio especificar su dimensión.

El tamaño del vector quedará fijado cuando se haga la instanciación de la variable, del siguiente modo:

nombreVector = **new** *identificadorTipo [tamaño];*

Donde:

- *new* es el operador de instanciación.
- *tamaño* es un número natural que fija el número de elementos.

Por ejemplo, la instanciación del anterior vector de enteros:

```
diasMeses = new int[12];
```

Declaración e instanciación pueden también realizarse simultáneamente. Siguiendo con el mismo ejemplo, se haría mediante la instrucción:

```
int [] diasMeses = new int[12];
```

También se puede declarar el vector con inicialización de sus valores, conforme a la siguiente sintaxis:

identificadorTipo [] nombreVector = listaValores ;

Para el vector `diasMeses`, la declaración e instanciación sería:

```
int[] diasMeses = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

La enumeración explícita de los valores del vector supone, implícitamente, la reserva de memoria necesaria para almacenarlos (del mismo modo que cuando se usa *new*, el operador de instanciación).

Los elementos de los vectores se numeran empezando por cero. Si el tamaño de un vector es N, los índices para acceder a sus elementos están en el rango [0..N-1]. En los vectores de JAVA, además, se dispone de un atributo, llamado **length**, que almacena el tamaño del vector.

Los elementos de los vectores se inicializan al valor por defecto del tipo correspondiente (cero para valores numéricos, el carácter nulo para *char*, *false* para *boolean*, *null* para *String* y para referencias).

Considérese el siguiente ejemplo de declaración, instanciación e inicialización de un vector:

```
char[] vocales = new char[5];
// inicializar componentes
vocales[0] = 'a';
vocales[1] = 'e';
vocales[2] = 'i';
vocales[3] = 'o';
vocales[4] = 'u';
// mostrar tamaño del vector (escribe: Tamaño = 5)
System.out.println( "Tamaño = " + vocales.length );
```

Vectores multidimensionales

Los vectores pueden tener varias dimensiones. La declaración de vectores o arrays multidimensionales (matrices) se hace de modo similar al caso de los vectores de una dimensión. Sólo hay que añadir un par de corchetes para cada dimensión que tenga la matriz o vector multidimensional.

Por ejemplo, una declaración e instanciación válida de una matriz de enteros de tres filas y tres columnas sería la siguiente:

```
int[][] mat3x3 = new int[3][3];
```

Otra forma equivalente de declarar e instanciar la misma matriz, pero reservando memoria independiente para cada fila, sería la siguiente:

```
int[][] mat3x3 = new int[3][];
mat3x3[0] = new int[3];
mat3x3[1] = new int[3];
mat3x3[2] = new int[3];
```

Por último, la declaración de la matriz con inicialización de sus valores se haría del siguiente modo:

```
int[][] mat3x3 = { {1,2,3}, {4,5,6}, {7,8,9} }
```

2.6. Estructuras de control

En esta sección, trataremos las diferentes estructuras de control disponibles en el lenguaje JAVA. En primer lugar, se describen las estructuras condicionales, que permiten la ejecución de bloques de código dependiendo de la evaluación de condiciones. A continuación, se tratan las estructuras de repetición o iteración, que permiten la ejecución repetida de bloques de código. Finalmente, se describe brevemente el uso de las sentencias de salto.

2.6.1. Estructuras condicionales

Las sentencias de selección permiten establecer bifurcaciones en el código, dando lugar a la ejecución de diferentes bloques de instrucciones en función de la evaluación de alguna condición lógica.

Sentencia if

La estructura condicional más importante en JAVA (y en otros lenguajes) es la sentencia **if**. La sintaxis de esta sentencia es la siguiente:

> **if** *(condición) sentencias1;*
> **else** *sentencias2;*

Donde:

- *condición* es una expresión lógica, que se evalúa a verdadero o falso.
- *sentencias1* es el bloque de instrucciones que se ejecuta si la condición es verdadera.
- *sentencias2* es el bloque de instrucciones que se ejecuta si la condición es falsa.

La presencia de la cláusula **else** es opcional (no se usará si no se requiere ejecutar nada cuando la condición del *if* no se cumple). Si el bloque *sentencias1* o *sentencias2* consta de una única sentencia, entonces las llaves del bloque correspondiente son opcionales.

El siguiente bloque de código muestra un posible uso de la sentencia de selección:

```
{
  int a = 18, b = 14, c = 22;
  int maximo, minimo;
  if ( a>b ) {
    maximo = a;
    minimo = b;
  } else {
    maximo = b;
    minimo = a;
  }                    // maximo = 18, minimo = 14
  if ( maximo<c ) {
    maximo = c;
  }                    // maximo = 22, minimo = 14
  if ( minimo>c ) {
   minimo = c;
  }                    // maximo = 22, minimo = 14
}
```

En este ejemplo, dados tres números enteros, se obtienen los valores máximo y mínimo. Se indican, mediante comentarios al código, los valores obtenidos después de ejecutarse cada sentencia *if*.

Se han utilizado tres sentencias condicionales:

- En la 1ª sentencia *if* se comparan los valores de `a` y `b`, ejecutándose el 1^{er} bloque de instrucciones al cumplirse la condición (esta sentencia *if* tiene cláusula *else* pero su bloque de instrucciones, para los valores de este ejemplo, no se ejecuta).
- Con la 2ª sentencia *if* se determina el valor máximo (esta vez no hay cláusula *else* porque no se necesita hacer nada si no se cumple la condición).
- Con la 3ª sentencia *if* se determina el valor mínimo (tampoco aquí se necesita cláusula *else*).

En JAVA, como en otros lenguajes, las sentencias *if-else* pueden concatenarse en una secuencia de verificación de condiciones. Como ejemplo, considérese el siguiente código:

```
{
  double nota = ... ; // llamar a un método que calcule la nota
  // mostrar la palabra correspondiente al valor de la nota
  if ( nota >= 9 )      System.out.println( "Sobresaliente" );
  else if ( nota >= 7 ) System.out.println( "Notable" );
  else if ( nota >= 5 ) System.out.println( "Aprobado" );
  else                  System.out.println( "Suspenso" );
}
```

En este ejemplo, se escribe por consola la palabra "Sobresaliente", "Notable", "Aprobado" o "Suspenso", según la variable `nota` tenga un valor en los rangos [9, 10], [7, 9[, [5, 7[, o [0, 5[, respectivamente.

La sentencia *if* permite, con construcciones como la anterior, discriminar entre un número amplio de condiciones y seleccionar los correspondientes bloques de instrucciones a ejecutar. Sin embargo, si el número de alternativas o valores a comprobar es extenso, resulta más recomendable usar la sentencia *switch* en su lugar.

Sentencia switch

La sentencia de selección **switch** evalúa una expresión que devuelve un valor en función del cual se selecciona un determinado bloque de instrucciones a ejecutar. La instrucción *switch* tiene la siguiente sintaxis:

switch *(expresión)* {
  case *valor1: sentencias1;* **break***;*
  case *valor2: sentencias2;* **break***;*
  ...
  case *valorN: sentenciasN;* **break***;*
  default*: sentencias_por_defecto;*
}

Donde:

- Los valores de la lista {*valor1*, *valor2*, ..., *valorN*} han de ser valores del mismo tipo que el devuelto por la expresión evaluada.
- Cada **case** señala un punto de entrada en el bloque de código de la sentencia *switch*.
- Cada **break** señala un punto de salida en el bloque de código de la sentencia *switch*.
- La cláusula **default** es opcional en la sentencia *switch*.

La expresión evaluada debe ser de un tipo simple enumerable (enteros, caracteres, booleanos). No se permite usar números reales.

Si la expresión del *switch* devuelve:

- Un cierto *valor-i* para el que existe un *case*, entonces se ejecuta su bloque de instrucciones *sentencias-i*. Si el *case* termina con una sentencia *break*, sólo se ejecuta su bloque. Pero si no hubiera *break*, se ejecutarían el bloque *sentencias-(i+1)* y siguientes, hasta encontrar algún *break* o acabar el bloque del *switch*.

- Un valor que no corresponde al de ningún *case*, entonces se ejecuta, si existe, el bloque de instrucciones *sentencias_por_defecto*, etiquetado como *default*.

Como ejemplo, consideramos el siguiente código *switch*:

```
{
  int contadorA ...   // declarar e inicializar contadores
  char vocal = ... ;  // llamar a un método que obtenga una vocal
  // incrementar el contador correspondiente a la vocal obtenida
  switch ( vocal ) {
    case 'A':
    case 'a': contadorA++; break;
    case 'E':
    case 'e': contadorE++; break;
    case 'I':
    case 'i': contadorI++; break;
    case 'O':
    case 'o': contadorO++; break;
    case 'U':
    case 'u': contadorU++; break;
    default:  contadorNoVocal++;
  }
}
```

En este ejemplo, se actualizan los valores de unas variables enteras (llamadas `contadorA`, `contadorE`, etc.) que llevan la cuenta de las vocales obtenidas mediante la ejecución de algún método. Supondremos que la llamada al método, previa a la sentencia *switch*, devuelve casi siempre vocales. Suponemos también que las variables enteras han sido correctamente declaradas e inicializadas en otro bloque de código (antes de ejecutarse el código mostrado).

La sentencia *switch* del ejemplo evalúa una variable de tipo *char*, llamada `vocal`. Si la letra es una vocal, se entra en el case correspondiente, incrementándose el contador apropiado. Obsérvese que el contador de cada vocal se incrementa tanto si la letra está en mayúsculas como en minúsculas. Para el caso de que la letra no sea vocal, se ha especificado un caso *default* donde se actualizaría un contador, `contadorNoVocal`, que lleva la cuenta de los caracteres que no sean vocales.

Hay que llamar la atención sobre la importancia de incluir la instrucción *break* al final de cada bloque *case*. Si se omite algún *break* en el ejemplo anterior, los contadores no se incrementan correctamente. Por ejemplo, si se omite el primer *break*, dejando el código de la siguiente forma:

```
switch ( vocal ) {
  case 'A':
  case 'a': contadorA++;
  case 'E':
  case 'e': contadorE++; break;
  case 'I':
  ...
}
```

El funcionamiento no sería el deseado. Al recibir una vocal 'E', se incrementa el `contadorE`. Pero al recibir una vocal 'A'se incrementan dos contadores: `contadorA` y `contadorE`, dado que se ejecutan todas las instrucciones hasta llegar a un *break*. En consecuencia, la variable `contadorE` almacenaría la cuenta de vocales 'A'y 'E', y no sólo las 'E'como se pretendía.

2.6.2. Estructuras de repetición

Las sentencias de repetición permiten la ejecución de un bloque de instrucciones de modo repetitivo. El bloque de instrucciones puede ejecutarse un número de veces fijado de antemano, o un número de veces variable en función del cumplimiento de alguna condición. En ambos casos, la estructura debe construirse garantizando que el número de ejecuciones del bloque sea finito. De lo contrario, se entraría en un bucle infinito y el programa en ejecución no saldría nunca del mismo, y nunca terminaría.

Sentencia while

La sintaxis de esta sentencia es la siguiente:

> **while** *(condición)* { *sentencias;* }

Donde:

- *condición* es una expresión lógica, que se evalúa a verdadero o falso.
- *sentencias* es el bloque de instrucciones que se ejecutará si la condición es verdadera. Si el bloque sólo contiene una instrucción, entonces las llaves son opcionales.

La condición se evalúa antes de entrar en el bloque. Por tanto, puede darse el caso de que el bloque no se ejecute ni una sola vez (cuando la condición sea falsa inicialmente). Cada vez que condición sea verdadera, se ejecutará una iteración del bloque sentencias, y se volverá a evaluar la condición.

Para que el bucle sea finito, ha de garantizarse que, en algún momento, la condición dejará de cumplirse. Para ello, se debe incluir en el bloque sentencias alguna instrucción que modifique alguna variable que forme parte de la condición a evaluar.

Como ejemplo de uso de la estructura *while*, considérese el siguiente bloque de código que calcula las sumas de los números pares e impares almacenados en un vector de enteros:

```
{
  int[] a = {3, 7, 12, 22, 9, 25, 18, 31, 21, 14, 45, 2};
  int sumaPares = 0, sumaImpares = 0;
  int i = 0;
  while ( i < a.length ) {
    if ( a[i] \% 2 == 0 ) sumaPares += a[i];
    else                  sumaImpares += a[i];
    i++;
  }
  System.out.println( "sumaPares = " + sumaPares);
  System.out.println( "sumaImpares = " + sumaImpares);
}
```

El número de iteraciones del bucle *while* de este ejemplo se controla con la variable `i`. Esta variable:

- Se inicializa a cero para acceder a la 1ª componente del vector a en la 1ª iteración.
- Se incrementa en una unidad (`i++`) al final de cada iteración (así se accede a componentes sucesivas del vector, y también se actualiza adecuadamente el número de repeticiones).

- Y se compara con la longitud del vector (`i < a.length`) de manera que el bucle concluye cuando la variable se iguala a esa longitud, porque entonces ya se han procesado todas las componentes del vector.

Otro ejemplo de estructura *while* se muestra en el siguiente método que comprueba si un vector es capicúa:

```
// método capicúa
public static boolean capicua (int[] v) {
  int i = 0;
  int j = v.length-1;
  while ( i<j && v[i]==v[j] ) {
    i++;
    j--;
  }
  return i>=j;
}

// programa principal que usa el método
public static void main (String[] args) {
  int[] v = { 1, 2, 3, 3, 2, 1 };
  System.out.println( "El vector  " +
                      ( capicua(v) ?  "si " :  "no  " ) +
                      "es capicua  " );
}
```

En el método capicua, el bucle *while* tiene una condición doble:

- Verifica que los dos índices de exploración del vector no se han cruzado o igualado (`i<j`).
- Verifica la igualdad de los componentes apuntados por estos índices (`v[i]==v[j]`).

En este caso, el bloque de instrucciones del *while* se limita a actualizar convenientemente las dos variables índice (`i++; j--;`). Si el vector es capicúa, el bucle *while* concluirá cuando los índices se crucen o igualen, es decir, cuando (`i>=j`) sea verdadera. Obsérvese que el resultado de esta expresión booleana es lo que devuelve el método, mediante la instrucción *return*.

En el método *main*, la invocación del método `capicua` se ha escrito dentro de una instrucción *println* y, concretamente, formando parte del operador ternario ?: al aprovechar el hecho de que el resultado del método es un valor *boolean*.

Sentencia do-while

Es una variante de la sentencia *while*. Tiene la siguiente sintaxis:

> **do** { *sentencias;* } **while** *(condición);*

La única diferencia respecto a la estructura *while* radica en cuando se verifica la condición lógica para continuar o abandonar el bucle. En la sentencia **do-while**, la condición se verifica después de ejecutar el bloque sentencias. Por tanto, el contenido del bucle se ejecuta por lo menos una vez (a diferencia del *while* donde podía no ejecutarse ninguna vez).

Como ejemplo de la estructura *do-while*, consideremos otra versión del método que comprueba si un vector es capicúa:

```
public static boolean capicua2 (int[] v) {
  int i = -1;
  int j = v.length;
  do {
    i++;
    j--;
  } while ( i<j && v[i]==v[j] );
  return i>=j;
}
```

La única diferencia se encuentra en la inicialización de las variables índice, al tener en cuenta que, dentro del *do-while*, esas variables se actualizan antes de verificar la condición del bucle.

Sentencia for

JAVA, como otros lenguajes, proporciona también una sentencia de repetición llamada **for**. Esta estructura tiene la siguiente sintaxis:

> **for** *(inicialización; condición; iteración)* { *sentencias;* }

Donde se usa una variable que habitualmente permite fijar el número de iteraciones, estableciendo:

- El valor inicial de dicha variable (en la asignación llamada *inicialización* en el esquema).
- La modificación de esa variable al final de cada iteración (en la expresión llamada *iteración*).
- El valor final de esa variable (en la expresión booleana llamada *condición*).

Para cada valor de la variable de control del bucle, desde su valor inicial hasta su valor final, se ejecuta una iteración del bloque de *sentencias*.

La sentencia *for* es la recomendada cuando se conoce de antemano el número de iteraciones del bucle. Esta situación es muy frecuente cuando se trabaja con vectores.

Considérese el siguiente bloque de código:

```
{
  int[] v1 = { 16,22,31,41,65,16,77,89,19 };
  int[] v2 = { 12,27,39,42,35,63,27,18,93 };
  int[] v3 = new int[v1.length];
  int[] v4 = new int[v1.length];
  for ( int i=0; i<v1.length; i++ ) {
    v3[i] = v1[i] + v2[i];
    v4[i] = v1[i] - v2[i];
  }
}
```

En este ejemplo, dados dos vectores, `v1` y `v2`, se obtienen otros dos vectores: `v3`, vector suma, componente a componente, de los anteriores; y `v4`, vector resta. La variable `i` del bucle for recorre el rango `[0..v1.length[`. En consecuencia, dadas las sentencias dentro del bucle *for*, todas las componentes de los vectores `v1` y `v2` se suman y restan, almacenándose en `v3` y `v4`, respectivamente.

Véase otro ejemplo, esta vez con matrices (vectores bidimensionales):

```
{
  int[][] m1 = { {16,22,31}, {41,65,16}, {77,89,19} };
  int[][] m2 = { {12,27,39}, {42,35,63}, {27,18,93} };
  int[][] m3 = new int[m1.length][m1[0].length];
  for (int i=0; i<m1.length; i++)
    for (int j=0; j<m1[0].length; j++)
      m3[i][j] = m1[i][j] + m2[i][j];
}
```

En este ejemplo, dadas dos matrices, `m1` y `m2`, se obtiene `m3`, matriz suma. Para sumar componente a componente las dos matrices, se requieren dos bucles *for* con anidamiento: el bucle controlado por la variable `j` es interno a (está anidado en) el bucle controlado por la variable `i`.

2.7. Sentencias de salto

En este apartado se tratan brevemente las instrucciones que JAVA proporciona para saltar de un punto del código a otro. Son las instrucciones **break**, **continue** y **return**.

Sentencia return

La sentencia **return** permite devolver el resultado obtenido al ejecutarse un método (como ya se ha visto en ejemplos anteriores). También puede situarse al final del programa principal, puesto que el *main* es también un método.

En el código de un método pueden aparecer varias instrucciones *return*. Sin embargo, en cada ejecución del método, sólo una de las instrucciones return se ejecutará, dado que la misma siempre supone la conclusión del método.

Como ejemplo, véase la siguiente versión del método que verifica si un vector es capicúa:

```
public static boolean capicua3 (int[] v) {
  int i = 0;
  int j = v.length-1;
  while ( i<j ) {
    if ( v[i]!=v[j] ) return false;
    i++;
    j--;
  }
  return true;
}
```

Sentencia break

La sentencia **break** ya se ha visto en su uso para interrumpir la ejecución de una sentencia *switch*. En esa situación, la ejecución del *break* supone saltar a la primera instrucción que esté justo después del final de la estructura *switch*.

Además, *break* puede usarse dentro de los bucles para salir de los mismos sin verificar la condición de finalización del bucle. Considérese el siguiente ejemplo:

```
public static int[] dividir_vectores (int[] v1, int[] v2) {
  int[] v3 = new int[v1.length];
  for (int i=0; i<v1.length; i++) {
    if ( v2[i]==0 ) {
```

```
        System.out.println( "Vectores no divisibles:" +
                            "un divisor es cero" );
        break;
      }
      v3[i] = v1[i] / v2[i];
    }
    return v3;
  }
```

Este método obtiene un vector de enteros resultante de dividir, componente a componente, los dos vectores de enteros que recibe como argumentos.

Para evitar una división por cero, en el caso de que algún elemento del segundo vector sea cero, se ha incluido una sentencia *break* para cancelar la operación. La ejecución del *break* causa la salida del bucle *for*, con independencia del valor de `i`, la variable del bucle. Esta solución dista de ser la mejor, pero se presenta para ilustrar este posible uso de la sentencia *break*.

Sentencia continue

La sentencia **continue** sirve, dentro de bucles, para salir anticipadamente de la iteración del bucle y continuar con la ejecución de la siguiente iteración del mismo bucle.

Veamos su uso con una variante del anterior ejemplo de método que divide dos matrices:

```
 public static int[][] dividir2_matrices (int[][] m1, int[][] m2) {
   int[][] m3 = new int[m1.length][m1[0].length];
   for (int i=0; i<m1.length; i++) {
     for (int j=0; j<m1[0].length; j++) {
       if ( m2[i][j]==0 ) continue;
       m3[i][j] = m1[i][j] / m2[i][j];
     }
   }
   return m3;
 }
```

En este ejemplo, cuando se encuentra una componente nula de la matriz `m2`, se interrumpe la iteración actual (de modo que no se ejecuta la instrucción `m3[i][j]=m1[i][j]/m2[i][j];` evitándose el error de la división por cero) y se pasa a la siguiente iteración del bucle (es decir, se incrementa la variable `j`).

Las instrucciones *break* y *continue* se pueden usar también con etiquetas que indican los lugares del código a donde saltar. Sin embargo, no se recomiendan estos usos dado que dificultan seriamente la legibilidad y comprensión del código fuente.

2.8. Ejercicios resueltos

2.8.1. Enunciados

1. Dado el siguiente programa principal:

```
 public static void main (String[] args) {
   int[] a = { 3, 7, 12, 22, 9, 25, 18, 31, 21, 14, 45, 2  };
   estadistica_pares_impares(a);
 }
```

Escribir el código del método `estadistica_pares_impares` para que realice los siguientes cálculos con las componentes del vector de números enteros que recibe como argumento:

- Suma de las componentes cuyo valor sea un número par.
- Suma de las componentes cuyo valor sea un número impar.
- Conteo de las componentes cuyo valor sea un número par.
- Conteo de las componentes cuyo valor sea un número impar.
- Valor máximo de las componentes cuyo valor sea un número par.
- Valor máximo de las componentes cuyo valor sea un número impar.

Además, el método debe mostrar, mediante mensajes en la consola, los resultados obtenidos.

2. Dado el siguiente programa principal:

```
public static void main (String[] args) {
  char[] v = { 'a', 'e', 'A', 'q', 'i', 'U', 'a',
               'E', 'p', 'O', 'u', 'A', 'i', 'e' };
  estadistica_vocales(v);
}
```

Escribir el código del método `estadistica_vocales` para que realice los siguientes cálculos con las componentes del vector de caracteres que recibe como argumento:

- Número de apariciones de cada vocal (sin distinguir mayúsculas y minúsculas).
- Número de apariciones de letras que no sean vocales.

Además, el método debe mostrar, mediante mensajes en la consola, los resultados obtenidos.

3. Implementación de métodos para realizar operaciones con matrices de números enteros.

   a) Escribir el código de un método que devuelva una matriz de números aleatorios, y que reciba como argumentos: el número de filas, el número de columnas, y el rango de valores permitidos (rango al que deben pertenecer los números aleatorios que se elijan como valores).

   b) Implementar dos métodos que devuelvan, respectivamente, el número de filas y el número de columnas de una matriz dada.

   c) Implementar tres métodos que verifiquen:

   - Si dos matrices tienen las mismas dimensiones.
   - Si una matriz es cuadrada.
   - Si una matriz es simétrica.

   Se recomienda usar los métodos definidos en el ejercicio anterior.

   d) Escribir el código de un método que devuelva una matriz que sea la suma de dos matrices dadas, previa comprobación de que son sumables (en caso contrario, se devolverá una matriz con todos los valores a cero).

e) Escribir el código de un método que devuelva una matriz que sea el producto de dos matrices dadas, previa comprobación de que son multiplicables (en caso contrario, devolver una matriz con todos los valores a cero).

f) Implementar un método que muestre en la consola los valores de una matriz dada.

g) Escribir un método main donde se invoquen todos los métodos anteriores de modo que se compruebe su funcionamiento.

2.8.2. Soluciones

1.
```
public class Estadistica {
  public static void estadistica_pares_impares (int[] a) {
    int sumaPares = 0, contPares = 0, mayorPar = 0;
    int sumaImpares = 0, contImpares = 0, mayorImpar = 0;
    boolean pp = true, pi = true;
    int i = 0;
    while ( i < a.length ) {
      if ( a[i] \% 2 == 0 ) {
        sumaPares += a[i];
        contPares++;
        if ( pp || a[i] > mayorPar ) mayorPar = a[i];
        pp = false;
      } else {
        sumaImpares += a[i];
        contImpares++;
        if ( pi || a[i] > mayorImpar ) mayorImpar = a[i];
        pi = false;
      }
      i++;
    }
    System.out.println("sumaPares= " + sumaPares);
    System.out.println("sumaImpares= " + sumaImpares);
    System.out.println("contPares= " + contPares);
    System.out.println("contImpares= " + contImpares);
    System.out.println("mayorPar= " + mayorPar);
    System.out.println("mayorImpar= " + mayorImpar);
  }
}
```

2.
```
public class Estadistica {
  public static void estadistica_vocales (char[] vocales) {
    int[] contadores = { 0, 0, 0, 0, 0, 0 };
    char[] letras = { 'a', 'e', 'i', 'o', 'u', 'x' };
    for (int i=0; i<vocales.length; i++) {
      switch ( vocales[i] ) {
        case 'a':
        case 'A': contadores[0]++; break;
        case 'e':
        case 'E': contadores[1]++; break;
        case 'i':
        case 'I': contadores[2]++; break;
        case 'o':
        case 'O': contadores[3]++; break;
        case 'u':
```

```
            case 'U': contadores[4]++; break;
            default: contadores[5]++;
          }
        }
        for (int i=0; i<contadores.length; i++)
          System.out.println("contador[" + letras[i] + " ]= "
                             + contadores[i]);
      }
    }
```

3. *a*)

```
public class Matriz {
  ...
  public static int generar_aleatorio (int min, int max) {
    return min+(int)(Math.random()*(max-min));
  }
  public static int[][] crear_matriz (int f, int c, int min,
                                      int max) {
    int[][] m = new int[f][c];
    for (int i=0; i<f; i++)
      for (int j=0; j<c; j++)
        m[i][j] = generar_aleatorio(min, max);
    return m;
  }
  ...
}
```

b)

```
public static int filas (int[][] m) {
  return m.length;
}
public static int columnas (int[][] m) {
  return m[0].length;
}
```

c)

```
public static boolean mismas_dimensiones (int[][] m1,
                                          int[][] m2) {
  return filas(m1)==filas(m2) &&
         columnas(m1)==columnas(m2);
}
public static boolean es_cuadrada (int[][] m) {
  return filas(m)==columnas(m);
}
public static boolean es_simetrica (int[][] m) {
  if ( !es_cuadrada(m) ) return false;
  for (int i=0; i<filas(m); i++)
    for (int j=0; j<i; j++)
      if ( m[i][j]!=m[j][i] ) return false;
  return true;
}
```

d)

```
public static int[][] sumar_matrices (int[][] m1,
                                      int[][] m2) {
  int[][] m3 = new int[filas(m1)][columnas(m1)];
  if ( mismas_dimensiones(m1,m2) ) {
    for (int i=0; i<filas(m1); i++)
      for (int j=0; j<columnas(m1); j++)
        m3[i][j] = m1[i][j] + m2[i][j];
```

```
    } else {
      System.out.print("distintas dimensiones: " );
      System.out.println("matrices no sumables" );
    }
    return m3;
  }
```

e)

```
  public static int[][] producto_matrices (int[][] m1,
                                           int[][] m2) {
    int[][] m3 = new int[filas(m1)][columnas(m2)];
    if ( columnas(m1)==filas(m2) ) {
      for (int i=0; i<filas(m1); i++)
        for (int j=0; j<columnas(m2); j++)
          for (int k=0; k<columnas(m1); k++)
            m3[i][j] += m1[i][k] * m2[k][j];
    } else {
      System.out.println( "matrices no multiplicables" );
    }
    return m3;
  }
```

f)

```
  public static void mostrar_matriz (int[][] m) {
    for (int i=0; i<filas(m); i++) {
      for (int j=0; j<columnas(m); j++)
        System.out.print(m[i][j]+" ");
      System.out.println("");
    }
    System.out.println("");
  }
```

g)

```
  public static void main (String[] args) {
    int[][] m1 = crear_matriz(5,4,1,100);
    int[][] m2 = crear_matriz(5,4,1,100);
    int[][] m3 = crear_matriz(4,7,100,1000);
    mostrar_matriz(m1);
    mostrar_matriz(m2);
    mostrar_matriz(m3);
    int[][] m4 = sumar_matrices(m1,m2);
    mostrar_matriz(m4);
    int[][] m5 = sumar_matrices(m2,m3);
    mostrar_matriz(m5);
    int[][] m6 = crear_matriz(3,4,1,10);
    int[][] m7 = crear_matriz(4,3,1,10);
    int[][] m8 = producto_matrices(m6,m7);
    mostrar_matriz(m6);
    mostrar_matriz(m7);
    mostrar_matriz(m8);
    int[][] m9 = crear_matriz(5,5,1,10);
    mostrar_matriz(m9);
    if ( es_cuadrada(m9) )
      System.out.println("matriz cuadrada");
    int[][] m10 = { {1,2,3,5}, {2,5,7,8},
                    {3,7,4,9}, {5,8,9,1} };
    mostrar_matriz(m10);
    if ( es_simetrica(m10) )
      System.out.println("matriz simétrica");
```

```
    }
```

2.9. Ejercicios propuestos

Implementación de métodos para realizar operaciones de búsqueda de palabras (vectores de caracteres) en una sopa de letras (matriz de caracteres).

1. Escribir el código de un método que devuelva una "palabra" (un vector de caracteres).

   El método debe generar un vector de caracteres aleatorios, recibiendo como argumentos: la longitud del vector y el rango de valores permitidos (rango al que deben pertenecer los caracteres aleatorios que se elijan como valores).

2. Escribir el código de un método que devuelva una "sopa de letras" (una matriz de caracteres).

   El método debe generar una matriz de caracteres aleatorios, recibiendo como argumentos: el número de filas, el número de columnas, y el rango de valores permitidos (rango al que deben pertenecer los caracteres aleatorios que se elijan como valores).

3. Escribir el código de métodos que devuelvan:

   - El número de filas de una sopa de letras dada.
   - El número de columnas de una sopa de letras dada.

4. Escribir el código de métodos para:

   - Obtener (leer) el carácter de una casilla de una sopa de letras dada.
   - Establecer (escribir) el carácter de una casilla de una sopa de letras dada.

5. Escribir el código de un método que compruebe si unas coordenadas dadas (números de fila y columna) corresponden a una casilla válida de una sopa de letras.

6. Escribir el código de un método que compruebe si una palabra dada está en una determinada posición (descrita por las coordenadas de inicio de búsqueda y las coordenadas de sentido o dirección de la búsqueda) de una sopa de letras.

7. Escribir el código de un método que muestre en la consola los valores de una sopa de letras.

8. Escribir el código de un método main donde se genere una sopa de letras, se genere una palabra a buscar, y se realice la búsqueda de todas las apariciones de la palabra en la sopa de letras, listando por pantalla las coordenadas de inicio y de dirección de cada una de las apariciones.

Capítulo 3

Fundamentos de la Programación Orientada a Objetos con Java

3.1. Clases y objetos

El concepto de *clase* es lo primero que distingue a un lenguaje orientado a objetos de otros lenguajes no orientados a objetos tales como C o Pascal. Una *clase* define un nuevo *tipo de dato*. Una vez definido dicho tipo de dato, éste se utilizará para crear o instanciar *objetos* de dicho tipo. El *objeto* o *instancia* es el segundo concepto que distingue a los lenguajes orientados a objetos del resto. El programador que utiliza un lenguaje orientado a objetos centra su trabajo en el diseño de clases que servirán después para crear objetos o instancias. Mediante el diseño una cada clase, el programador describe los atributos y el comportamiento que tendrán todos los objetos pertenecientes a dicha clase, es decir, que sean instancias de la misma. También se pueden crear atributos y definir un comportamiento en una clase que sean propios de la misma e independientes de la existencia o no de objetos instanciados (apartado 3.3).

La forma sintáctica general de una clase es la siguiente:

```
class Nombre_de_clase {
  tipo_de_dato atributo1, atributo2, ...;
  tipo_de_dato ...;

  tipo_de_dato metodo1 ( lista_de_argumentos ) {
    // cuerpo del metodo1
  }

  tipo_de_dato metodo2 ( lista_de_argumentos ) {
    // cuerpo del metodo2
  }

  ...
}
```

La estructura anterior permite observar dos partes claramente diferenciadas en las que se debe dividir una clase. En primer lugar, la declaración de atributos de la clase y, en segundo

lugar, la definición de los métodos.

Los atributos: Sintácticamente son exactamente iguales a las declaraciones de variables vistas en el capítulo anterior. Sin embargo, su significado en el diseño de programas orientados a objetos es distinto. Las clases sirven para instanciar objetos y, durante todo el tiempo de vida de un objeto, sus atributos prevalecen y su valor representa el "estado" de dicho objeto en un momento dado. Conviene tener presente dos aspectos importantes de los atributos:

- Sintácticamente pueden ser declarados en cualquier parte dentro de la clase, sin embargo, es conveniente hacerlo siempre al principio por estilo de programación.
- Los atributos pueden tener *modificadores de acceso*:

```
modificador_de_acceso tipo_de_dato nombre_de_atributo;
```

  siendo este modificador una palabra reservada (o nada) que caracteriza el modo de acceso a dicho atributo desde fuera de la clase. Es conveniente utilizar el modificador que hace privado este acceso desde fuera aunque por simplicidad omitiremos esto hasta que veamos dichos modificadores (3.4.1).

Los métodos: Sintácticamente se parecen mucho a funciones o procedimientos de otros lenguajes, utilizados para encapsular partes de código que tienen entidad propia como para diseñarlas de manera independiente del resto del código. Estas subrutinas (procedimientos o métodos) reciben una serie de argumentos como entrada y pueden devolver un valor. La diferencia con respecto a subrutinas de otros lenguajes es fundamental dado que los métodos representan el comportamiento de los objetos. Los métodos, por regla general, acceden a los atributos de la instancia para:

- consultar los atributos del objeto, tanto para simplemente devolver el valor de un atributo (dado que lo usual es que sólo sean accesibles desde dentro de la propia clase) como para devolver valores derivados de algún cálculo que involucra a los atributos; y,
- modificar los atributos del objeto (los atributos no deberían nunca modificarse de otra forma que no fuera a través de un método de la clase 3.4).

Así pues, los métodos constituyen un mecanismo para acceder a los atributos de un objeto. La ejecución de una aplicación orientada a objetos puede verse como un conjunto de objetos que se crean y destruyen. Una vez creados los objetos éstos van cambiando su estado (a través de sus métodos) mientras dura su tiempo de vida.

A continuación se irán desarrollando las ideas anteriores a través de la descripción sintáctica y semántica de los elementos que proporciona el lenguaje Java.

La definición de las clases comienza por la palabra reservada `class` seguida de un identificador válido para el nombre de la clase. Es una práctica bastante aceptada que este nombre comience con una letra mayúscula por lo que se recomienda seguir esta costumbre. Todo el cuerpo de la clase está englobado en un bloque de código entre llaves. La declaración de atributos suele estar al comienzo de la misma aunque la sintaxis de Java da libertad para declararlas donde se desee. Un ejemplo de definición de clase, en el cual no se han declarado métodos, es el siguiente:

```
class Esfera {
  double radio;
}
```

Una vez declarada la clase, se puede utilizar como un nuevo tipo de dato para realizar dos cosas:

- declarar *variables referencia* o simplemente *referencias*, y
- crear objetos de dicha clase.

Mediante una clase se define un nuevo tipo de dato. A los objetos también se les suele denominar *instancias* de la clase, *instancias* del tipo de dato o simplemente *instancias*. Por eso, también se suele decir que crear objetos es instanciar la clase.

La declaración de una referencia del tipo definido por la clase se realiza de la misma manera que se declaran variables simples, por ejemplo,

```
Esfera miesfera;
```

donde `miesfera` es una variable referencia que servirá para apuntar o tener acceso a un objeto o instancia de la clase `Esfera`. El concepto de referencia es sumamente importante para comprender el resto del material. Una referencia es lo más parecido a un puntero en otros lenguajes como el lenguaje `C`, es decir, es una variable que se puede considerar como simple, ya que su tamaño es de 32 bits[1] y contendrá la dirección de memoria de comienzo del objeto o instancia al que haga referencia. De la misma manera que una variable simple como `int`, `double`, etc. contiene bits que representan números enteros, reales, etc., una referencia es una variable simple cuyos 32 bits representan direcciones de memoria. Cuando se declara una variable simple, se reserva un espacio de memoria de 8 hasta 64 bits para almacenar un dato de tipo simple. Cuando se declara una variable referencia a una cierta clase, se reserva espacio de memoria para esos 32 bits que necesita una dirección de memoria.

El concepto de referencia es importante puesto que el acceso a los objetos que se vayan creando se realiza siempre a través de este tipo de variables. Al contrario que en otros lenguajes, la aritmética con las variables tipo referencia es muy limitada en Java. Esto ofrece gran robustez a los programas escritos en Java dado que la posibilidad de equivocarse al utilizar aritmética de punteros en otros lenguajes suele ser bastante alta. En Java se pueden realizar asignaciones entre referencias siempre que los tipos sean compatibles. La compatibilidad de tipos en el caso de tipos definidos por el usuario, es decir, de clases, se verá en el apartado 4.2.6. Existe una constante denominada `null` que se puede asignar a cualquier referencia para indicar que la variable no apunta a ningún objeto. Con una referencia se pueden utilizar los siguientes operadores: `==`, `!=`, `=`, `.`, `instanceof`.

3.1.1. Instanciación de clases

La instanciación de una clase tiene por objetivo la creación de un objeto. Crear un objeto significa reservar espacio de memoria para el mismo y establecer una ligadura con una variable referencia. Es importante no confundir la declaración de una variable referencia con la creación de un objeto. La declaración de una variable referencia **no implica la creación de un objeto o instanciación de una clase**, sólo implica la declaración de una variable que será necesaria para acceder a los *miembros* (atributos y métodos) de un objeto. La instanciación de una clase se realiza mediante el operador `new`, el mismo operador que se vio para la declaración de vectores o matrices.

Por ejemplo, la declaración de una variable referencia de tipo `Esfera` y la instanciación de un objeto del mismo tipo se realizaría de la siguiente manera:

[1] Para procesadores de 32 bits es de 32 mientras que para procesadores de 64 bits es del doble.

```
Esfera miesfera;
miesfera = new Esfera();
```

La primera línea del ejemplo constituye la declaración de la variable referencia `miesfera` de tipo `Esfera`. A partir de este momento la variable `miesfera` solo podrá hacer referencia o apuntar a objetos que pertenezcan a la clase `Esfera`.

La segunda línea del ejemplo constituye la instanciación del objeto. La ejecución del operador `new` supone la reserva dinámica de memoria para un objeto nuevo de una clase determinada, clase que viene indicada por el nombre que viene dado a continuación del operador `new`. La memoria reservada por el operador `new` tiene una dirección de comienzo que es guardada en la variable referencia `miesfera`. Esta variable será utilizada a partir de este momento para acceder a los miembros del objeto instanciado. La figura 3.1 muestra la ejecución de estas dos líneas de código de manera gráfica.

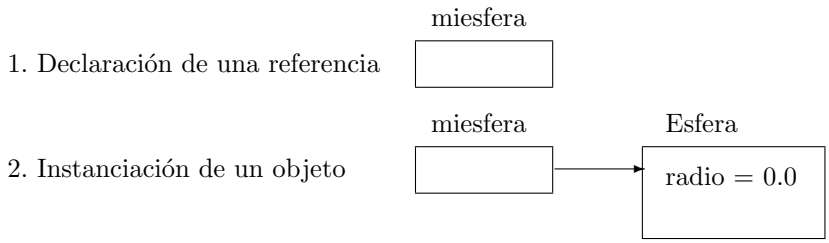

Figura 3.1: *Declaración de una variable referencia e instanciación de un objeto.*

Se entiende por *memoria dinámica* a la memoria que es reservada en tiempo de ejecución. La memoria dinámica es muy cómoda en el desarrollo de aplicaciones pero su utilización consume tiempo de ejecución. En algunos lenguajes de programación orientados a objetos es posible elegir el tipo de memoria a utilizar mientras que en el caso de JAVA no se pueden instanciar objetos de otra forma que no sea la que acaba de explicarse, al igual que sucede con la declaración de vectores.

De manera equivalente, puede realizarse la declaración de la referencia y la instanciación del objeto en una misma línea de código:

```
Esfera miesfera = new Esfera();
```

El acceso a los atributos y métodos del objeto se realiza a través de la referencia y del operador de acceso (`.`). Por ejemplo, para asignar el valor 10 al atributo `radio`,

```
miesfera.radio = 10;
```

o para obtener el valor de dicho atributo,

```
int a = miesfera.radio;
```

Los objetos son entidades completamente independientes. Cada objeto tiene su propio espacio de memoria, y cada cual debe tener una referencia distinta para su acceso. Así pues, en el siguiente código

```
Esfera miesfera1 = new Esfera();
Esfera miesfera2 = new Esfera();
```

se han instanciado dos objetos de la misma clase, cada uno con sus propios atributos. Se puede, por ejemplo, asignar valores a los atributos de cada uno de los objetos y calcular su área:

```
class Ejemplo {
  public static void main( String args[] ) {
    Esfera miesfera1 = new Esfera();
    Esfera miesfera2 = new Esfera();
    miesfera1.radio = 1.0;
    miesfera2.radio = 2.0;

    System.out.println(" Área de la esfera 1: " +
      4.0 * 3.14159 * miesfera1.radio * miesfera1.radio );

    System.out.println(" Área de la esfera 2: " +
      4.0 * 3.14159 * miesfera2.radio * miesfera2.radio );
  }
}
```

ofreciendo como resultado la siguiente salida:

```
Área de la esfera 1: 12.56636
Área de la esfera 2: 50.26544
```

Aunque en los ejemplos anteriores para cada objeto instanciado existía una variable referencia que lo apuntaba, la utilización de las referencias para apuntar a los objetos es versátil e independiente. Por ejemplo, se declara la referencia `miesfera1` de tipo `Esfera` y se instancia un objeto de tipo `Esfera` que será apuntado por `miesfera1`. Seguidamente, se declara la referencia `miesfera2` de tipo `Esfera` y se le asigna el valor de la referencia `miesfera1` (líneas 1-3 del código siguiente).

```
1: Esfera miesfera1 = new Esfera();
2: Esfera miesfera2;
3: miesfera2 = miesfera1;
4: miesfera1.radio = 3.0;
5: System.out.println(" Radio de la esfera: " + miesfera2.radio );
```

Ahora, la referencia `miesfera2` contiene la misma dirección de memoria que la referencia `miesfera1` y, por tanto, apunta al mismo objeto, es decir, que ahora habrá dos referencias apuntando al mismo objeto. El acceso a un miembro del objeto puede ser realizado utilizando cualquiera de las dos referencias. Las líneas 4 y 5 del código anterior muestran el acceso al atributo `radio` del único objeto existente (obsérvese que sólo se ha instanciado un objeto, sólo puede encontrarse un operador `new`) por medio de las dos referencias. Mediante `miesfera1` se accede a `radio` para asignarle un valor (línea 4) y mediante `miesfera2` se accede a `radio` para consultar el mismo valor (línea 5). El resultado de la ejecución es:

```
Radio de la esfera: 3.0
```

Esquemáticamente esto puede observarse en la figura 3.2.

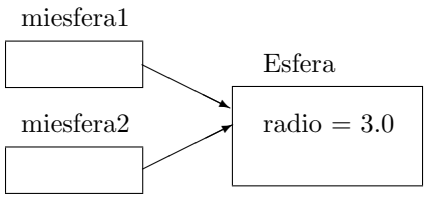

Figura 3.2: *Dos referencias apuntando al mismo objeto.*

A continuación se muestra un ejemplo que deja clara la idea de dos objetos iguales en el sentido de que poseen el mismo valor para sus atributos y de dos referencias que apuntan al mismo objeto.

```
class Ejemplo {
  public static void main( String args[] ) {
    Esfera miesfera1 = new Esfera();
    Esfera miesfera2;
    Esfera miesfera3 = new Esfera();

    miesfera1.radio = 1.0;
    miesfera2 = miesfera1;
    miesfera3.radio = 1.0;

    System.out.println("Las referencias miesfera1 y miesfera2 " +
      (miesfera1==miesfera2?"si":"no")+" apuntan al mismo objeto");
    System.out.println("Las referencias miesfera3 y miesfera1 " +
      (miesfera3==miesfera1?"si":"no")+" apuntan al mismo objeto");
  }
}
```

La salida del programa anterior es la siguiente:

```
Las referencias miesfera1 y miesfera2 si apuntan al mismo objeto
Las referencias miesfera3 y miesfera1 no apuntan al mismo objeto
```

donde se muestra que las referencias `miesfera1` y `miesfera2` apuntan al mismo objeto mientras que las referencias `miesfera3` y `miesfera1` apuntan a objetos distintos, aunque éstos tengan el mismo valor para su atributo.

3.1.2. Destrucción de objetos

Dado que la memoria que utilizan los objetos es dinámica, cabe suponer que será necesario algún mecanismo para poder recuperar ese espacio de memoria cuando ya no sea necesario, es decir, algo parecido al operador `delete` de *C++*, por ejemplo. Sin embargo, la liberación de memoria en JAVA es automática. JAVA posee un mecanismo denominado *Garbage Collection* mediante el cual, y de forma periódica, localiza objetos que no tengan ninguna referencia que les apunte para liberar la memoria que están utilizando. Un objeto sin referencia es, seguro, un objeto que no se utilizará más dado que no existe forma alguna de tener acceso a él. De esta manera, la liberación de memoria asignada a un objeto se puede forzar simplemente quitando todas las referencias al mismo, por ejemplo, asignando el valor `null` a dichas referencias.

3.2. Métodos

Como se ha mencionado anteriormente, un método es un "trozo" de código que se encapsula de manera independiente del resto del código. Desde este punto de vista, coincide con el concepto de función, procedimiento o subrutina de otros lenguajes. Esto es así dado que puede recibir unos parámetros o argumentos, y puede devolver un valor como resultado. Sin embargo, al contrario que en los lenguajes no orientados a objetos, los métodos no son completamente independientes, pertenecen siempre a una clase y sólo tienen sentido dentro de la misma dado que tienen como cometido acceder a sus atributos.

La estructura de un método es la siguiente:

```
tipo_de_dato nombre_de_metodo ( lista_de_argumentos ) {
  // Cuerpo del método
}
```

donde `tipo_de_dato` especifica, bien un tipo de dato simple, bien un tipo de dato definido por el usuario o bien nada, en cuyo caso se utiliza la palabra reservada `void`. Después, figura un identificador válido para el nombre del método y, entre paréntesis, una lista de argumentos separados por comas que puede estar vacía. Los argumentos se componen de un nombre precedido de su tipo de dato. Aunque existan varios argumentos con el mismo tipo de dato, cada uno ha de llevar especificado su tipo. El cuerpo del método es un bloque de código englobado entre llaves aun cuando esté constituido por una sola instrucción.

El ejemplo siguiente muestra la clase `Esfera` a la que se ha incorporado el método `area`.

```
class Esfera {
  double radio;

  double area( ) {
    return 4.0 * 3.14159 * radio * radio;
  }
}
```

Cabe mencionar que, desde el punto de vista del diseño orientado a objetos, es más apropiado incluir un método como éste dentro de la clase que averiguar el área del objeto desde fuera de la misma tal y como se ha hecho en ejemplos anteriores. En general y como estrategia de diseño orientado a objetos, se pretende dotar al objeto de toda aquella funcionalidad que pueda llevar a cabo, es decir, encapsular en la clase que define su tipo de dato todo lo que le atañe en cuanto a atributos y comportamiento.

Los métodos pueden verse de alguna manera como "solicitudes de servicio" que se realizan sobre el objeto sobre el que es llamado el método desde fuera del mismo. El código siguiente sería la versión correcta del ejemplo de la página 52. En lugar de realizar cálculos externos sobre atributos de un objeto (cálculo del área en el ejemplo), desde el punto de vista de la programación orientada a objetos resulta más adecuado solicitar que el objeto realice por sí mismo dichos cálculos en el método `area` y ofrezca el resultado tal como muestra el código siguiente:

```
class Ejemplo {
  public static void main( String args[] ) {
   Esfera miesfera1 = new Esfera();
   Esfera miesfera2 = new Esfera();
   miesfera1.radio = 1.0;
   miesfera2.radio = 2.0;

   System.out.println(" Área de la esfera 1: " + miesfera1.area() );

   double a =  miesfera2.area();
   System.out.println(" Área de la esfera 2: " + a );
  }
}
```

ofreciendo como resultado la misma salida que en el caso ejemplo de la página 52.

Como se ha observado en el ejemplo anterior, el método `area` no tiene argumentos pero sí devuelve un valor de tipo `double` que se ha impreso por la salida estándar en el primer caso y se ha asignado a la variable `a` en el segundo.

Un ejemplo de método con argumentos podría ser el método `setRadio` que se define de la siguiente manera:

```
void setRadio( double r ) {
  radio = r;
}
```

La inclusión de un método como `setRadio` en la clase `Esfera` permite asignar valores a los atributos del objeto de manera segura dado que el código del metodo `setRadio` puede comprobar que el valor pasado como argumento es válido antes de asignarlo al atributo `radio`. Desde este punto de vista, una implementación más correcta podría ser la siguiente:

```
void setRadio( double r ) {
  if( r>=0.0 ) {
    radio = r;
  }
}
```

de manera que ahora se puede estar seguro de que, aunque el método reciba un valor negativo como argumento, el radio de la esfera no tendrá nunca un valor negativo. A continuación se muestra la utilización del método `setRadio`:

```
Esfera miesfera1 = new Esfera(), miesfera2 = new Esfera();
miesfera1.setRadio( 1.0 );
miesfera2.setRadio( -4.0 );
miesfera2.setRadio( 2.0 );
...
```

Según la implementación del método `setRadio`, una llamada al mismo con un valor negativo no modificará el valor del radio de la esfera.

Conviene recordar en este punto que los argumentos de un método y las variables definidas dentro del mismo son variables locales al método y, por lo tanto, no son visibles fuera de éste. Su tiempo de vida está limitado al tiempo en el que se está ejecutando el método (por ejemplo, el argumento `r` del método `setRadio`). Por el contrario, los atributos declarados en el ámbito de la clase (`radio`), son perfectamente visibles dentro de cualquier método definido en la misma y cualquier modificación que se realice a través de un método permanecerá mientras no se cambie explícitamente y el objeto siga en ejecución.

Los métodos pueden no devolver nada (`void`) o devolver algún valor. En este segundo caso es preceptivo que exista al menos una instrucción de tipo `return`.

Antes de pasar a describir con detalle las particularidades de los métodos en JAVA, cabe mencionar que existe la *recursividad* en JAVA, es decir, la posibilidad de llamar al mismo método desde dentro del cuál se está realizando la llamada.

3.2.1. Constructores

Como se ha venido observando, en la instanciación de una clase para crear un objeto, se utiliza el nombre de la misma seguido de un paréntesis abierto y otro cerrado. En esta instanciación de la clase tiene lugar la ejecución de un *constructor*. El constructor, aunque sintácticamente es semejante a un método, no es un método. El constructor puede recibir argumentos, precisamente se utilizan los mencionados paréntesis para hacerlo, de la misma manera que se pasan argumentos en las llamadas a los métodos. Aunque el constructor no haya sido definido explícitamente, en JAVA siempre existe un constructor por defecto que posee el nombre de la clase y no recibe ningún argumento. Esa es la razón por la que

puede utilizarse el constructor `Esfera()` de la clase `Esfera` aunque no se haya definido explícitamente.

En el apartado anterior se utilizó el método `setRadio` para asignar valores al atributo `radio`. Sin embargo, existe un mecanismo más adecuado para asignar valores a los atributos de una clase cuando esta asignación se realiza en el momento de la creación del objeto. Este mecanismo consistente precisamente en la utilización de los "constructores". Los constructores son métodos especiales que tienen el mismo nombre que la clase en la que se definen, no devuelven ningún valor y son llamados en la instanciación de la misma para crear un objeto nuevo. El siguiente ejemplo muestra la implementación de un constructor para la clase `Esfera`:

```
class Esfera {
  double radio;

  Esfera( double r ) {
    radio = r;
  }
  ...
}
```

La instanciación de objetos de esta clase se ha de hacer ahora de la siguiente manera:

```
Esfera miesfera1 = new Esfera( 1.0 );
Esfera miesfera2 = new Esfera( 2.0 );
...
```

Varias cosas hay que tener en cuenta:

- Es perfectamente válido definir un método como el siguiente:

  ```
  void Esfera( double r ) {
    ...
  }
  ```

  Sin embargo, este método es precisamente eso, un método y no un constructor. El constructor no debe llevar especificador de tipo de retorno, ni siquiera `void`, dado que en este caso no podría utilizarse como constructor en la instanciación de la clase.

- Cuando se implementa un constructor, el constructor por defecto del que se disponía, es decir, el constructor sin argumentos, se pierde y ya no puede ser utilizado a no ser que se implemente explícitamente. Por ejemplo, según la última clase `Esfera` mostrada, lo siguiente sería un error,

  ```
  Esfera miesfera = new Esfera();
  ```

  dado que ahora el constructor `Esfera()` no existe.

3.2.2. Ocultación de atributos

Puede ocurrir que el nombre de los argumentos de un método o constructor coincida con el nombre de los atributos de la clase, como sucede en el siguiente ejemplo:

```
class Esfera {
  double radio;
  Esfera( double radio ) {
```

```
      this.radio = radio;
    }
  }
```

Esto no origina ningún conflicto para el compilador. Sin embargo, la utilización del mismo nombre oculta el atributo `radio` dentro del método dado que las variables más internas, es decir, las declaradas dentro del método y los argumentos tienen prioridad sobre los atributos de la clase. Tal como se ha mostrado en el ejemplo, mediante la utilización de la palabra reservada `this` es posible acceder a los atributos de clase distinguiéndolos así de las variables declaradas en el método y/o los argumentos con el mismo nombre.

3.2.3. Sobrecarga de métodos

La sobrecarga de métodos consiste en la posibilidad de definir dos o más métodos con el mismo nombre dentro de una misma clase, diferentes en su número y/o tipo de dato de los argumentos. El tipo de dato que devuelven dos o más métodos es insuficiente para diferenciarlos, por lo tanto, no se pueden definir métodos con el mismo nombre y número y tipo de argumentos que sólo se diferencien en el tipo de retorno.

Los constructores también pueden ser sobrecargados de la misma manera. Así, por ejemplo, se pueden sobrecargar los constructores de la clase `Esfera`,

```
class Esfera {
  double radio;

  Esfera( ) {
    radio = 0.0;
  }

  Esfera( double r ) {
    radio = r;
  }

  ...
}
```

y tener así la posibilidad de instanciar esferas con y sin dimensión.

```
Esfera miesfera1 = new Esfera( );
Esfera miesfera2 = new Esfera( 3.0 );
```

Un constructor puede ser llamado desde otro constructor. La forma de hacerlo consiste en utilizar la palabra reservada `this` en lugar del nombre del constructor. El constructor que será llamado vendrá determinado por el número y tipo de argumentos. El siguiente ejemplo muestra el constructor `Esfera` sobrecargado 4 veces.

```
class Esfera {
  double radio;
  Color color;
  Esfera( ) {
    color = Color.white;
    radio = 0.0;
  }
  Esfera( double radio ) {
    this();
```

```
      this.radio = radio;
    }
    Esfera( Color color ) {
      this();
      this.color = color;
    }
    Esfera( Color color, double radio ) {
      this.color = color;
      this.radio = radio;
    }
  }
```

La clase `Esfera` incluye ahora dos atributos: el radio y el color. Este segundo atributo pertenece a una clase definida en el API de JAVA denominada `Color`. La instanciación de un objeto de tipo `Esfera` sin argumentos crea una esfera blanca y de radio 0.0. El segundo constructor permite crear una esfera con un determinado radio y el color por defecto (el blanco). En este último caso se hace uso de `this` para llamar al constructor sin argumentos que asigna el color blanco al atributo `color`. El tercer constructor es análogo al anterior y se utiliza para asignar un color determinado a la nueva esfera sin que importe el radio. Estos dos últimos constructores son un ejemplo de sobrecarga con el mismo número de argumentos. El cuarto constructor permite la inicialización de los atributos con los valores pasados como argumentos.

Puede llamarse a cualquier constructor con y sin argumentos utilizando `this`. En el caso de este último constructor podría haberse pensado en implementarlo de la siguiente forma:

```
  Esfera( Color color, double radio ) {
    this(radio);
    this(color);
  }
```

Sin embargo, en este caso es incorrecto debido a que la utilización de `this` para llamar a un constructor sólo puede hacerse en la primera instrucción. La segunda llamada (`this(color)`) produce, por tanto, un error de compilación.

Por último, decir que desde un método no es posible utilizar `this()` para ejecutar el código de un constructor.

3.2.4. Objetos como argumentos de un método

En primer lugar conviene entender la manera en que se pasa un argumento cualquiera a un método. Otros lenguajes como *Pascal* diferencian entre paso de parámetros por *valor* o *referencia* en llamadas a subrutinas. En *C*, sin embargo, se dice que todos los argumentos se pasan por valor y que, para pasar un argumento por referencia ha de pasarse su dirección de memoria o puntero. En realidad, puede omitirse esta diferenciación entre el concepto de paso de argumentos por valor o por referencia en cualquier lenguaje si se analiza detenidamente el funcionamiento del paso de argumentos a una subrutina y ésto puede comprenderse fácilmente analizando el caso concreto de JAVA.

Puede afirmarse que un método sólo puede recibir datos de tipo simple, entre los cuales, se incluyen las referencias a objetos. Desde este punto de vista, en la llamada a un método, los argumentos pueden ser datos de tipo simple que representan números enteros (`byte`, `short`, `int` o `long`), caracteres (`char`), números reales (`float` o `double`), booleanos (`boolean`) o referencias. El argumento es una variable que recibe el valor de la variable pasada al método en la llamada, de manera que su modificación dentro del método no afecta a la variable

pasada a dicho método, es decir, se pasa únicamente su valor. El siguiente ejemplo muestra este concepto.

```
class A {
  void metodo( double argumento ) {
    argumento = 4.0;
  }
}

public class Ejemplo {
  public static void main(String args[]) {
    A ref;
    double variable;
    variable = 2.5;
    ref = new A();
    ref.metodo(variable);
    System.out.println("variable = "+variable);
  }
}
```

Según la explicación dada, la salida por pantalla debería quedar claro que debe ser la siguiente:

```
variable = 2.5
```

La asignación del valor 4.0 al argumento `argumento` no afecta en absoluto al valor de la variable `variable`. Además, tal como se ha dicho, este es el comportamiento común a la mayoría de lenguajes de programación. En lenguaje C, por ejemplo, el funcionamiento es idéntico. Si se desea cambiar el valor de una variable pasada como argumento a una función de C, lo que se pasa en realidad no es el valor de dicha variable sino su dirección. A través de su dirección, el método tiene acceso a dicha variable y puede modificar su valor real. En Pascal sucede lo mismo aunque con una sintaxis diferente.

Una diferencia fundamental entre estos lenguajes de programación y JAVA en relación a este tema es que, en el caso de JAVA, no existe ninguna posibilidad de pasar la dirección de una variable perteneciente a un tipo simple. Esto permite decir que los argumentos en JAVA siempre se pasan por valor y no es posible pasarlos por referencia. También es parte de la razón por la que se suele decir falsamente que en JAVA no existen los punteros. Aunque lo puede parecer, la imposibilidad de modificar una variable pasada como argumento a través de un método no constituye una limitación del lenguaje; sin embargo, simplifica considerablemente el lenguaje.

Si consideramos las referencias a objetos como variables o datos de tipo simple, las consideraciones anteriores pueden aplicarse inmediatamente para comprender el "paso de objetos como argumentos". Al pasar como argumento la referencia a un objeto, éste se puede utilizar para tener acceso a los miembros del mismo. Este acceso no tiene ninguna limitación, es decir, se puede utilizar la referencia para acceder a los atributos del objeto para leerlos e incluso para modificarlos tal como se ha venido haciendo hasta el momento. En definitiva, si se desea pasar un objeto a un método, se pasará una referencia al mismo. De todo lo anterior, debería deducirse fácilmente que la modificación del argumento referencia a un objeto dentro del método (asignarle, por ejemplo, `null`) no tiene efecto sobre la variable referencia que se pasó en la llamada al método.

En el ejemplo siguiente, se ha añadido un nuevo constructor a la clase `Esfera` cuya función es crear una esfera con la misma dimensión que otra dada.

```
class Esfera {
  double radio;

  Esfera( Esfera e ) {
    radio = e.radio;
  }
  ...
}
```

de manera que se tendrá la posibilidad de definir esferas con los mismos atributos de manera sencilla. Las siguientes dos esferas tienen exactamente la misma dimensión:

```
Esfera miesfera1 = new Esfera( 2.0 );
Esfera miesfera2 = new Esfera( miesfera1 );
```

Como se ha podido observar, dentro del nuevo constructor se ha accedido a los elementos de la clase `Esfera` a través de su referencia.

El siguiente ejemplo muestra un método que modifica el valor del atributo de un objeto:

```
void escalar( Esfera esfera, double escalado ) {
  esfera.radio *= escalado;
}
```

lo que muestra la manera en la que se pueden modificar atributos de un objeto desde un método.

3.2.5. Devolución de objetos desde un método

De la misma manera que un método devuelve datos de tipo simple, también puede devolver referencias a objetos. En el ejemplo siguiente se incorpora a la clase `Esfera` un método para mostrar la manera en la que se lleva a cabo la devolución de objetos por parte de un método.

```
class Esfera {
  double radio;

  ...
  Esfera doble( ) {
    Esfera c = new Esfera( 2*radio );
    return c;
  }
}
```

El método `doble` tiene por cometido crear una esfera nueva que tenga el doble de radio. Una vez creada esta nueva esfera, el método devuelve la variable referencia que contiene la dirección de memoria del objeto creado dentro del método. Esta referencia puede ser utilizada fuera del método para tener acceso a dicha esfera tal como muestra el ejemplo siguiente,

```
Esfera miesfera1 = new Esfera( 2.0 );
Esfera miesfera2 = miesfera1.doble( );
```

Después de la ejecución del código anterior, se dispone de dos objetos de tipo `Esfera`: el objeto apuntado por la variable referencia `miesfera1` con radio 2.0 y el objeto apuntado por la variable referencia `miesfera2` con radio 4.0.

Los argumentos y las variables locales a un método son variables simples y referencias. Las primeras se utilizan para representar números, caracteres y booleanos mientras que las

segundas se utilizan para representar direcciones de memoria donde comienza el espacio asignado a un objeto. Su dimensión es conocida en tiempo de compilación y su longitud puede ir desde los 8 hasta los 64 bits. Todas ellas, como argumentos de un método o variables locales del mismo, son temporales, es decir, son creadas cuando se ejecuta el método y se destruyen al acabar la ejecución del mismo. Sin embargo, el espacio de memoria asignado a un objeto cuando se crea (`new`) es dinámico, es decir, se reserva en tiempo de ejecución. Este espacio de memoria no sigue las misma reglas de tiempo de vida que las variables simples y las referencias, ya que no se destruye cuando acaba la ejecución del método donde se creó. El objeto creado en el método `doble()` del ejemplo anterior no se destruye a causa de la finalización de la ejecución del método. En dicho ejemplo, la referencia `c` de tipo `Esfera` es local al método y se destruye cuando acaba la ejecución del mismo. Esto no sucede con el objeto creado dentro del método con el operador `new`. El método `doble()` devuelve el valor de la referencia `c` que es copiado en la referencia `miesfera2` en la llamada al método `miesfera1.doble()`.

3.3. Miembros de instancia y de clase

Los atributos y métodos que se han visto hasta ahora en la clase `Esfera` se denominan *de instancia*. Esto quiere decir que sólo pueden ser utilizados cuando se ha instanciado un objeto de la clase. Mientras el objeto no se haya instanciado mediante el operador `new` no es posible su acceso. Dicho de otra forma, los atributos y métodos de instancia pertenecen a las instancias, si no existe la instancia no existen dichos miembros.

En contraposición existen los miembros llamados *de clase* o *estáticos*. Estos miembros existen siempre y, por lo tanto, pueden ser utilizados sin necesidad de haber instanciado un objeto de la clase. La introducción de los miembros de clase dota a éstas de una funcionalidad adicional a la de ser una especificación de tipo de dato de usuario para crear objetos de dicho tipo. Sintácticamente, se introduce la palabra reservada `static` precediendo la declaración de *atributos y métodos de clase*. Las implicaciones que tienen tanto la declaración de un atributo de clase como la definición de un método de clase son algo distintas en cada caso.

Los atributos de clase son accesibles sin necesidad de instanciar ningún objeto de la clase puesto que "pertenecen a la clase". Pero es importante saber también que son accesibles por las instancias de la clase teniendo en cuenta que el atributo de clase es **único** y **común** a todas ellas. Si una instancia modifica dicho atributo, cualquier otra instancia de la clase "verá" dicho atributo modificado. Sea el ejemplo siguiente,

```
class Atristatic {
  static int atributo;
}

public class Ejemplo {
  public static void main(String args[]) {
    Atristatic instancia1 = new Atristatic();
    Atristatic instancia2 = new Atristatic();
    instancia1.atributo = 3;
    System.out.println("El atributo \"atributo\" desde " +
          "la instancia2 vale: " + instancia2.atributo);
  }
}
```

la salida de este programa es

```
El atributo "atributo" desde la instancia2 vale: 3
```

Dado que no es necesaria la instanciación de la clase para el acceso a atributos y métodos de clase, es de esperar que exista un mecanismo para poder acceder a dichos elementos en caso de que no exista una referencia válida. Los atributos y métodos de clase (también llamados estáticos) pueden ser accedidos con el nombre de la clase. La tercera línea del método `main` de la clase `Ejemplo` anterior es equivalente a la siguiente:

```
Atristatic.atributo = 3;
```

En el caso de los métodos, las implicaciones son más sencillas ya que el código de un método no puede cambiar durante la ejecución de un programa tal y como lo hace una variable. Lo usual en la utilización de los métodos estáticos es acceder a ellos por el nombre de la clase a la que pertenecen.

Hay que tener en cuenta que los métodos y atributos de clase pueden ser utilizados siempre dado que no es necesario que se haya instanciado una clase para poder acceder a ellos. Sin embargo, constituye un problema el acceso a un método o atributo de instancia desde un método estático. En el ejemplo siguiente

```
class Atristatic {
  static int atributo_estatico;
  int atributo_de_instancia;

  static void metodo_estatico( Atristatic objeto ) {
    atributo_estatico = 1;
    atributo_de_instancia = 1; // línea errónea
    objeto.atributo_de_instancia = 1;
  }
}
```

el acceso al atributo `atributo_de_instancia` desde el método estático `metodo_estatico` produce un error de compilación. El compilador no permite el acceso a través de `this` a miembros de la clase (`atributo_de_instancia = 1;` y `this.atributo_de_instancia = 1;` son equivalentes). Otra cosa bien distinta la constituye el acceso a cualquier miembro a través de la referencia a un objeto existente tal como muestra la última línea de código del ejemplo.

El siguiente código es un ejemplo completo de una posible implementación del concepto de número complejo (clase `Complejo`) que muestra la utilización de los métodos de instancia y de clase para llevar a cabo parte de la aritmética de complejos.

```
class Complejo {
  double re, im;

  // Constructores
  Complejo ( double r, double i ) { ... }
  Complejo ( Complejo a )         { ... }

  // Métodos de instancia
  void     inc( Complejo a ) { ... }
  void     dec( Complejo a ) { ... }
  void     por( Complejo a ) { ... }
  void     por( double d )   { ... }
  void     div( double d )   { ... }
  double   modulo( )         { ... }
  Complejo conjugado( )      { ... }
```

```
    // Métodos de clase
    static Complejo suma     ( Complejo a, Complejo b ) { ... }
    static Complejo resta    ( Complejo a, Complejo b ) { ... }
    static Complejo producto ( Complejo a, Complejo b ) { ... }
    static Complejo division ( Complejo a, Complejo b ) { ... }
}
```

En la clase `Complejo`, existen siete métodos de instancia (no estáticos) cuya definición es la que figura en la Tabla 3.1. Se observa que el método `por` está sobrecargado para permitir la multiplicación de un número complejo por otro complejo o un escalar.

`inc`	$c \leftarrow c + a$
`dec`	$c \leftarrow c - a$
`por`	$c \leftarrow c \times a$
	$c \leftarrow c \times d$
`div`	$c \leftarrow c \div d$
`modulo`	`c`
`conjugado`	c^*

Tabla 3.1: *Métodos de instancia de la clase* `Complejo`*, donde* $a, c \in C$*, siendo* c *el complejo representado por la instancia sobre la que se invoca el método, y* $d \in R$*.*

Desde el punto de vista del diseño de una aplicación es importante notar cómo se han utilizado y por qué los métodos de instancia. Todos los métodos de instancia trabajan sobre el estado de un objeto concreto, bien para obtener alguna información del mismo (`modulo` y `conjugado`), o bien para modificar su estado (`inc`, `dec`, `por` y `div`). Por ejemplo,

```
Complejo conjugado( ) {
  return new Complejo( re, -im );
}
void por( double d ) {
  re *= d;
  im *= d;
}
```

Los métodos que modifican el estado de una instancia de la clase `Complejo` pretenden simular operaciones como `c = c + a`, `c = c * a`, ... que pueden realizarse sobre variables simples pero no sobre objetos de una clase.

A raíz de este hecho, cabe mencionar que, en otros lenguajes de programación orientados a objetos como *C++*, existe la posibilidad de sobrecargar operadores, lo que permitiría dotar a los operadores aritméticos de la posibilidad de operar sobre objetos de determinadas clases tales como la clase `Complejo`.

En la Tabla 3.2 se muestran los cuatro métodos de clase de la clase `Complejo` y un ejemplo de implementación es el siguiente:

```
static Complejo suma ( Complejo a, Complejo b ) {
  return new Complejo( a.re+b.re, a.im+b.im );
}
```

Estos métodos se implementan para ofrecer servicios que se considera adecuado que proporcione la clase y de manera independiente de las instancias concretas que se hayan creado.

`suma`	$c \leftarrow a + b$
`resta`	$c \leftarrow a - b$
`producto`	$c \leftarrow a \times b$
`division`	$c \leftarrow a \div b$

Tabla 3.2: *Métodos de clase de la clase* `Complejo`*, donde* $a, b, c \in C$ *siendo* c *el complejo resultado de la operación.*

Dicho de otra forma, los métodos de clase son la manera más parecida en la que se implementaría dicha operación en otros lenguajes de programación no orientados a objetos. Los métodos `suma`, `resta`, `producto` y `division` permiten realizar cualquiera de estas operaciones aritméticas sobre dos objetos de tipo `Complejo` que son pasados como argumentos y ofrecer un resultado del mismo tipo. Hay que notar que no se modifica el estado de ninguna instancia al realizar llamadas sobre estos métodos, su función consiste en recibir dos objetos como argumento, crear uno nuevo que sea resultado de la operación que implementan y devolverlo.

En muchos casos, una misma operación puede implementarse como método de instancia o como método de clase. Es una decisión de estilo de programación la que hay que tomar para decantarse por una u otra solución. La que se ha mostrado aquí a través de la clase ejemplo `Complejo` se ha justificado convenientemente y resulta bastante adecuada aunque no siempre es la que se sigue. Por ejemplo, en el API[2] de JAVA existe una clase denominada `Object` (apartado 4.1.5) con un método `equals` que devuelve cierto si el objeto pasado como argumento es igual que el objeto sobre el que es invocado el método. Según la estrategia de programación planteada hasta el momento este método debería haberse implementado como un método de clase.

Si se necesita ejecutar instrucciones para inicializar las variables estáticas en el proceso de "carga de la clase" éstas se pueden especificar en un *bloque estático*. JAVA utiliza lo que se denomina *carga dinámica* (dynamic loading) de código. Éste es un mecanismo mediante el cual la aplicación JAVA carga o lee en memoria un fichero ejecutable JAVA (extensión `.class`) en tiempo de ejecución en el momento en que lo "necesita". Este momento es la primera vez que se utiliza la clase para instanciarla o para acceder a un miembro estático.

La utilización de un bloque estático tiene la forma que muestra el siguiente ejemplo:

```
class UsoStatic {
  static int a = 3;
  static int b;

  // Bloque estatico
  static {
    System.out.println("Entra en el bloque estático.");
    b = a * 4;
  }

  static void metodo( int x ) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
```

[2]API (Application Programming Interface): Conjunto de componentes sofware (funciones, procedimientos, clases, métodos, ...) proporcionados para poder ser utilizados por otro software.

```
      System.out.println("b = " + b);
    }

  }
```

La ejecución del código siguiente:

```
public class Ejemplo {
  public static void main(String args[]) {
    UsoStatic s = new UsoStatic();
    UsoStatic.metodo(42);
  }
}
```

muestra la siguiente salida por pantalla:

```
Entra en el bloque estático.
x = 42
a = 3
b = 12
```

La carga de la clase `UsoStatic` se produce con la instanciación de la misma y, por tanto, tiene lugar la ejecución del bloque estático. A partir de ese momento, las variables `a` y `b` tienen los respectivos valores `3` y `12`. La subsiguiente ejecución del método estático `metodo` muestra los valores concretos de las variables estáticas y del argumento.

El siguiente es otro ejemplo de utilización de bloque estático que muestra un uso más práctico de los bloques estáticos.

```
public class Ejemplo {
  static int potencias2[];
  static {
    potencias2 = new int[10];
    potencias2[0] = 1;
    for( int i=1; i<10; i++ ) {
      potencias2[i] = 2*potencias2[i-1];
    }
  }
}
```

Cuando se carga la clase del ejemplo se calculan las 10 primeras potencias de 2. Este cálculo se lleva a cabo una sola vez mientras dure la ejecución de la aplicación.

3.4. Encapsulación de código

Por un lado, las clases en un lenguaje orientado a objetos constituyen la solución que ofrece este paradigma de programación al concepto de Tipo Abstracto de Datos (TAD) perseguido durante gran parte de la historia del software con objeto de simplificar el proceso de programación y de hacer abordables aplicaciones cada vez más grandes y complejas. Por otro lado, el concepto subyacente a los TAD's es el de la *encapsulación*.

La *programación estructurada* introduce las subrutinas (funciones y procedimientos) con objeto de abordar problemas grandes subdividiendo o particionando dicho problema en problemas más pequeños. Pero esta subdivisión se realiza desde el punto de vista funcional, es decir, se describe un algoritmo grande en función de un conjunto de algoritmos más pequeños

que, a ser posible, tengan entidad propia. Una rutina tiene entidad propia cuando es independiente del resto y reutilizable. La reutilización de código es una característica altamente deseable dado que no solo supone reducción de costes de programación sino que también proporciona

- fiabilidad: en la medida en que las rutinas son más utilizadas son más fiables, y
- eficiencia: al preocuparse de una subrutina de manera independiente del resto (incluso ésta puede ser implementada por un equipo distinto de programadores) es más fácil dedicar un esfuerzo adicional para que sea lo más eficiente posible.

La programación orientada a objetos da un salto de gigante en esta dirección reuniendo o encapsulando tanto atributos como funcionalidad en una sola entidad de código. El particionado o descomposición del problema ya no se realiza sólo desde el punto de vista funcional sino que va más allá. Los objetos surgen de la necesidad de modelizar el problema del diseño de software desde el punto de vista de las entidades que lo componen y de su clasificación en distintos tipos o clases. Básicamente ésta es la idea bajo la que se define un TAD y las clases son, hoy en día, una de sus mejores aproximaciones.

La utilización de rutinas en la implementación de un programa permite cierto grado de abstracción dado que para utilizar una subrutina no es necesario conocer cómo dicha rutina resuelve un problema sino únicamente saber qué problema resuelve y cual es su *interfaz*. La interfaz de una rutina viene dada por sus argumentos de entrada y de salida, es decir, qué datos y de qué tipo son los que hay que proporcionar a dicha rutina para que resuelva el problema y qué datos y de qué tipo son los que devuelve. Se puede decir que las rutinas "encapsulan" en su interior el algoritmo de resolución. Asociado al concepto de encapsulación se encuentra el de "visibilidad", que viene dado por su interfaz o aquello que se desea dar a conocer y que sea accesible desde otra parte del código.

La programación orientada a objetos, evidentemente, incluye el mismo tipo de encapsulación a nivel de métodos que el proporcionado por la programación estructurada con las subrutinas. Sin embargo, eleva la encapsulación de código a un nivel de abstracción superior, el nivel de abstracción proporcionado por las clases, las cuales pueden encapsular tanto los atributos de las entidades como el comportamiento asociado a las mismas. La interfaz en el caso de las clases puede venir dada por el prototipo de algunos de sus métodos y por algunos de sus atributos. La interfaz de una clase debería estar bien definida previamente a su implementación únicamente por ese subconjunto de métodos que la clase "quiere" ofrecer al resto de entidades y que definen los servicios que cualquiera de sus instancias ofrece.

El conjunto de atributos y métodos que se desea permanezcan "ocultos" es, por supuesto, decisión del programador. Sin embargo, existen mecanismos, como es el caso de Java, para dotar de distintos niveles de accesibilidad a los miembros de una clase y, de esta manera, construir una interfaz apropiada con la que otros objetos de la aplicación puedan interactuar con los objetos de dicha clase permitiendo la accesibilidad a unos miembros e impidiéndosela a otros. Como regla general y con el objetivo de llevar a cabo una programación adecuada y lo más próxima posible a la programación orientada a objetos la interfaz debería estar constituida únicamente por el prototipo de un subconjunto de métodos. No deberían estar incluidos ciertos métodos que se han implementado a la hora de subdividir un algoritmo complejo en pequeños algoritmos (programación estructurada), y nunca por los atributos, los cuales deberían quedar "encapsulados" en la clase (figura 3.3).

El siguiente subapartado describe los mecanismos de Java para proporcionar accesibilidad a los miembros de la interfaz e impedirla a aquellos que no forman parte de ella.

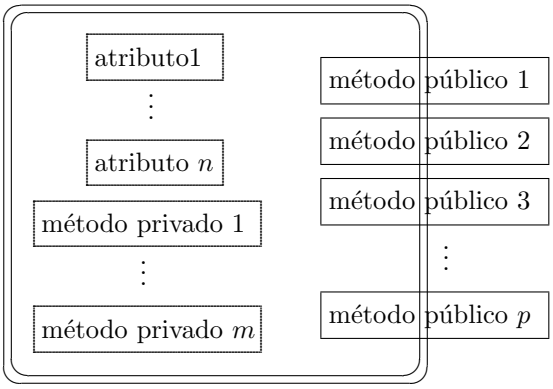

Figura 3.3: *Encapsulación de código en una clase. Los métodos públicos constituyen la interfaz de la clase con el exterior.*

3.4.1. Modificadores de acceso

Los miembros de una clase tienen una propiedad denominada *accesibilidad*. Esta propiedad se representa mediante un *modificador de acceso*. En principio, un miembro puede ser básicamente *público* o *privado*. La cuestión que hay que abordar consiste en saber desde dónde se entiende esa accesibilidad, es decir, los miembros de una clase pueden ser accedidos desde dentro de la propia clase, desde otra clase que pertenezca al mismo paquete o una clase que pertenezca a distinto paquete, desde una subclase, ... Dado lo cual, la caracterización pública o privada de un miembro ha de ser desarrollada en cada punto explicativo en el que tenga sentido hablar de ello y no se puede resumir hasta haber planteado los conceptos de herencia y de paquetes.

Sin embargo, en este punto, es posible diferenciar dos modos de acceso a los miembros de una clase: acceso público y acceso privado, haciendo la salvedad de que el concepto completo de la accesibilidad de los miembros de una clase no podrá ser conocida hasta comprender otros términos básicos. Por el momento se distinguen:

- **atributos y métodos privados** como miembros de una clase que no pueden ser accedidos desde otra clase, solo pueden serlo dentro de la propia clase. Estos miembros poseen el modificador de acceso `private`; y
- **atributos y métodos públicos** como miembros de una clase que pueden ser accedidos desde otra clase. Estos miembros poseen el modificador de acceso `public` o *friendly*. Se considera el acceso como *friendly* cuando no se posee ningún modificador explícito (modificador de acceso por defecto).

La diferencia entre poseer el atributo `public` y el atributo de acceso por defecto (*friendly*) está relacionada con la utilización de paquetes por lo que se verá su diferencia en ese contexto.

Un ejemplo de utilización de modificadores de acceso es el siguiente:

```
class A {
  int i;
  private int j;
}

class B {
  A a = new A();
```

```
    a.i = 10;
    // a.j = 10 no está permitido
    // System.out.println("Variable j: " + a.j); tampoco está permitido
  }
```

En el ejemplo, la variable `i` es accesible desde fuera de la clase `A`. De hecho, desde la clase `B` se accede a dicha variable a través de la referencia `a` de tipo `A` a la misma para modificar su valor. Sería equivalente haber declarado dicha variable como `public int i`. Sin embargo, el acceso a la variable `j` de la clase `A` no está permitido desde fuera de la clase `A`, ni para escritura ni para lectura, es decir, no se puede consultar su valor aunque no se modifique.

A continuación se muestra un ejemplo en el que se construye la clase `Pila`. Esta clase sirve para almacenar números enteros siguiendo esta estructura de datos en la que el último elemento introducido en la pila es el primer elemento en salir de la misma. La interfaz de la estructura de datos `Pila` viene dada por las siguientes operaciones:

`public void add( int valor ):` Mediante esta operación se añade el nuevo número entero proporcionado como argumento. La operación no devuelve nada.

`public int get():` Mediante esta operación se extrae el último número entero introducido en la estructura. Cada extracción devuelve los números de la estructura en orden inverso al que se introdujeron. La operación devuelve el número entero extraído.

`public int top():` Esta operación devuelve el último número entero introducido en la pila. La operación no modifica la información de pila.

```
class Pila {
  private int n;
  private int vector[];

  public Pila( int talla ) {
    vector = new int[talla];
    n = 0;
  }

  public void add( int valor ) {
    if( n<vector.length ) {
      vector[n++] = valor;
    }
  }

  public int get( ) {
    int a = -11111;
    if( n>0 ) {
      a = vector[--n];
    }
    return a;
  }

  public int top( ) {
    int a = -11111;
    if( n>0 ) {
      a = vector[n-1];
    }
```

```
      return a;
    }
  }
```

La clase `Pila` puede albergar números enteros hasta un máximo de `talla` definido por el usuario de la clase en la instanciación de la misma. Se utiliza un vector para almacenar los números de la pila y una variable para almacenar la cantidad de elementos almacenados. Según la implementación propuesta la introducción de un nuevo elemento cuando la pila está llena no tiene efecto. La extracción de un elemento de una pila vacía devuelve un valor de -11111. Se verá en el Capítulo 5 la forma correcta de realizar el control necesario para operaciones que pueden desencadenar errores como introducir datos en una pila llena o extraerlos de una pila vacía. Nótese, además, que los atributos de la clase se han declarado como privados mientras que los métodos que forman parte de la interfaz (en este caso todos) se han declarados como públicos.

3.5. Clases anidadas y clases internas

Las *clases anidadas* son clases definidas dentro de otras. La clase anidada sólo es visible para la clase que la contiene. En otras palabras, la clase anidada es un miembro privado por defecto de la clase externa que la contiene. La clase externa no tiene acceso a los miembros de la clase anidada.

Existen dos tipos de clases anidadas: las *estáticas* y las *no estáticas*. Las clases anidadas y estáticas poseen el modificador `static`. Estas clases poseen ciertas restricciones que las hacen poco útiles respecto de las clases no estáticas por lo que su utilización es escasa. Las clases anidadas no estáticas reciben también el nombre de *internas*. Las clases internas tienen acceso a todos los miembros (incluso a aquellos privados) de la clase en la que se ha definido.

Un ejemplo de aplicación es el siguiente (en el apartado 3.8 se explica el programa principal que utiliza esta clase):

```
class Sistema_Solar {
  private Estrella estrella;
  private Planeta planeta[];

  class Estrella {
    private String nombre;
    Estrella( String nombre ) {
      this.nombre = nombre;
    }
    public String getNombre() {
      return nombre;
    }
  }

  class Planeta {
    private String nombre;
    Planeta( String nombre ) {
      this.nombre = nombre;
    }
    public String getNombre() {
      return nombre;
    }
  }
```

```
    Sistema_Solar( String astro[] ) {
      estrella = new Estrella(astro[0]);
      planeta = new Planeta[astro.length-1];
      for( int i=1; i<astro.length; i++ ) {
        planeta[i-1] = new Planeta(astro[i]);
      }
    }

    void mostrar_planetas() {
      System.out.println("Estrella del sistema solar = "+
                                                 estrella.getNombre());
      for( int i=0; i<planeta.length; i++ ) {
        System.out.println("Planeta "+(i+1)+" = "+
                                                 planeta[i].getNombre());
      }
    }
  }
```

En este ejemplo, el constructor de la clase `Sistema_Solar` recibe como argumento un vector de referencias de tipo `String` (la clase `String` se explica en el apartado siguiente) y utiliza estas cadenas o *strings* para construir un sistema solar con un nombre para la estrella y nombres para los planetas. Obsérvese que en el constructor también se utiliza un vector de referencias a objetos de tipo `Planeta`. En general, un vector de referencias no es diferente a un vector de otros datos de tipo simple (números reales y enteros, caracteres y booleanos) como los estudiados en el Capítulo 2, simplemente cada elemento del vector en este caso servirá para referenciar a un objeto del mismo tipo del cual se ha declarado el vector de referencias.

Las clases internas pueden definirse en otros ámbitos o bloques como, por ejemplo, los métodos o el cuerpo de un bucle. A este tipo de clases se les denomina clases *anónimas* dado que no se les asigna un nombre. No darle un nombre tiene sentido si sólo se va a instanciar una vez. Actualmente, su utilización es muy frecuente y casi exclusiva en la implementación de "manejadores de eventos" para el desarrollo de interfaces gráficas de usuario, cuestión que se verá en el Capítulo 6.

3.6. La clase `String`

Para terminar este capítulo conviene hacer mención de la clase `String` de JAVA. La clase `String` es una clase que proporciona el propio lenguaje para el manejo de cadenas de caracteres. Se trata de una clase especial. Puede utilizarse como cualquier otra clase para instanciar objetos que almacenen cadenas de caracteres de la siguiente manera,

```
String st = new String("Esto es una cadena");
```

Sin embargo, un código equivalente al anterior es el siguiente,

```
String st = "Esto es una cadena";
```

Dado que `String` es una clase especial, los dos códigos anteriores son equivalentes.

Otra particularidad de la clase `String` es que tiene el operador `+` sobrecargado, es decir, se pueden "sumar" objetos de tipo `String` dando como resultado una cadena nueva formada por ambas cadenas concatenadas. Por ejemplo, la salida del código siguiente,

```
public class Ejemplo {
  public static void main(String args[]) {
    String string1 = "Esto es ";
    String string2 = "la cadena resultado";
    String string3 = string1 + string2;
    System.out.println(string3);
  }
}
```

sería,

```
Esto es la cadena resultado
```

Esto explica la utilización del método `System.out.println` donde el argumento puede estar formado por una concatenación de cadenas para formar la cadena resultado final que se desea mostrar en pantalla. Los tipos simples, además, promocionan a un objeto de tipo `String` cuando son concatenados con una cadena. De manera que, por ejemplo, la forma de mostrar en pantalla un mensaje como el siguiente:

```
x = 100
```

donde 100 es el valor de una variable entera declarada de esta manera: `int x=100;`, sería

```
System.out.println("x = "+x);
```

La clase `String` es el mecanismo que proporciona JAVA para la manipulación de cadenas de la misma manera que otros lenguajes no orientados a objetos lo hacen mediante librerías de subrutinas. A través de los métodos de la clase es posible realizar aquellas operaciones más comunes sobre las cadenas de caracteres. Destacan los siguientes métodos:

`length()`: Devuelve la longitud de una cadena. Nota: obsérvese que `length()` es un método de la clase `String` que no hay que confundir con el atributo `length` de los vectores.

`equals( String st )`: Devuelve un valor lógico indicando si la cadena `st` pasada como argumento es igual o no a aquella sobre la que se llama el método.

`charAt( int i )`: Devuelve el carácter (tipo `char`) correspondiente a la posición `i`, comenzando desde 0 como en las matrices.

Para el ejemplo siguiente:

```
public class Ejemplo {
  public static void main(String args[]) {
    String animal1 = "elefante";
    String animal2 = "león";
    String animal3 = "león";
    System.out.println("El primer y el segundo animal "+
      (animal1.equals(animal2)?"si":"no")+" son iguales" );
    System.out.println("El segundo y el tercer animal "+
      (animal2.equals(animal3)?"si":"no")+" son iguales" );
    System.out.println("La tercera letra de "+
      animal1+" es la \""+animal1.charAt(2)+"\"");
    System.out.println("El número de letras de \""+
      animal2+"\" es "+animal2.length());
  }
}
```

la salida por pantalla sería:

```
El primer y el segundo animal no son iguales
El segundo y el tercer animal si son iguales
La tercera letra de elefante es la "e"
El número de letras de "león" es 5
```

3.7. Paquetes

Los *paquetes* de JAVA son contenedores de clases que se utilizan para mantener el espacio de nombres dividido en compartimentos. Las clases se almacenan en dichos paquetes y tienen un nombre único dentro de dicho paquete de manera que ninguna clase pueda entrar en conflicto con otra clase con el mismo nombre pero en distinto paquete, ya que el nombre del paquete las diferencia. Los paquetes pueden estar a su vez dentro de otros paquetes estructurándose así de manera jerárquica.

La definición de un paquete consiste simplemente en la inclusión de la palabra reservada `package` como primera sentencia de una archivo JAVA. Por ejemplo, cualquier clase que se defina dentro del archivo que posee como primera línea la siguiente:

```
package mi_paquete;
```

pertenecerá al paquete `mi_paquete`.

Si se omite la sentencia *package*, las clases se asocian al paquete por defecto, que no tiene nombre.

JAVA utiliza los directorios del sistema de archivos para almacenar los paquetes. El paquete `mi_paquete` declarado más arriba ha de encontrarse en el directorio `mi_paquete`. Se pueden crear subpaquetes dentro de otros paquetes respetando la jerarquía de directorios, siendo la forma general de declaración la siguiente:

```
package paq1[.paq2[.paq3]];
```

hasta el nivel de profundidad necesario y, por supuesto, coincidiendo con la jerarquía de directorios apropiada. Esta jerarquía de la que se ha hablado hacer referencia a los ficheros objeto JAVA, es decir, los `.class`. La organización de los códigos fuente (`.java`) es responsabilidad del programador y no tiene porqué seguir esta estructura jerárquica.

Dentro de un paquete, sólo son visibles aquellas clases que pertenecen a dicho paquete. La utilización de clases que se encuentran en otros paquetes diferentes al que pertenecen las clases que se están implementando en el fichero actual se lleva a cabo mediante su importación. Por ejemplo, si se desea utilizar la clase `mi_clase` definida en el paquete `mi_paquete` en el fichero actual, se ha de incluir al principio de dicho fichero (siempre después de la declaración de paquete) la siguiente instrucción:

```
import mi_paquete.mi_clase;
```

Asimismo, es posible importar todas las clases del paquete mediante el comodín `*`,

```
import mi_paquete.*;
```

para poder así tener acceso a todas las clases de dicho paquete.

La importación de clases mediante el mecanismo explicado no es necesaria, por ejemplo, si van a ser sólo unas pocas las clases importadas. Siempre es posible denotar una clase de otro paquete utilizando su nombre completo, por ejemplo,

```
java.util.GregorianCalendar cal = new java.util.GregorianCalendar();
```

Tampoco es necesaria para el caso particular del paquete `java.util` del JDK de JAVA.

3.7.1. La variable de entorno CLASSPATH

La variable de entorno CLASSPATH contiene la raíz de la jerarquía de directorios que utiliza el compilador de JAVA para localizar los paquetes.

Para que funcione adecuadamente la clasificación de clases dentro de paquetes es necesario tener en cuenta algunas cuestiones:

- Crear el directorio con el mismo nombre que el paquete que se desea crear, por ejemplo, `mi_paquete`.
- Asegurarse que el fichero con el bytecode correspondiente a las clases que pertenecen a dicho paquete (contienen la sentencia `package.mi_paquete;`) se encuentran en dicho directorio.

  Esto se puede omitir si se utiliza la opción `-d` del compilador de JAVA. Mediante esta opción, el compilador coloca las clases (`.class`) en el directorio y subdirectorios correspondientes, creando incluso las carpetas necesarias si éstas no existen, para almacenar las clases y reflejar la estructura elegida según los paquetes creados.
- Para ejecutar una clase (por ejemplo, `mi_clase`) de dicho paquete es necesario indicar al intérprete JAVA el paquete y la clase que se desea ejecutar,

  ```
  java mi_paquete.mi_clase
  ```
- Introducir en la variable CLASSPATH el directorio raíz del paquete donde se encuentran las clases que se van a utilizar para que el intérprete de JAVA las encuentre.

Ejemplos de valores de la variable CLASSPATH son:

- Windows. Para darle valores a dicha variable abriríamos una ventana del intérprete de comandos de MS-DOS:

  ```
  C:> set CLASSPATH=C:MisClasesDeJava;C:MisOtrasClasesDeJava
  ```

  Si se desea añadir valores conservando los anteriores:

  ```
  C:> set CLASSPATH=C:MisClasesDeJava;%CLASSPATH%
  ```
- Linux. Desde una consola haríamos

  ```
  > set classpath="/home/usuario/MisClasesDeJava;$classpath"
  ```

  Hay que tener en cuenta que el comando exacto depende del tipo de shell de linux.

La ejecución de aplicaciones JAVA que utilizan paquetes definidos por el usuario tiene algunos detalles a tener en cuenta por lo que se recomienda ver el **Ejercicio** 2.

3.8. Paso de argumentos al programa

Las aplicaciones JAVA que se ejecutan en el entorno de ejecución del sistema operativo pueden recibir argumentos. Los argumentos son cadenas de texto, es decir, secuencias de caracteres. Cada cadena se diferencia de las adyacentes por caracteres separadores (espacios en blanco, tabuladores, ...). Estos argumentos son pasados a la aplicación JAVA en un vector de caracteres en la llamada al método `main`. Por ejemplo, en el siguiente código

```
public class Planetas {
  public static void main(String[] args) {
    Sistema_Solar s = new Sistema_Solar(args);
    s.mostrar_planetas();
  }
}
```

el argumento `args` del método `main` es un vector de referencias a objetos de tipo `String`. En la llamada al método `main`:

1. se instancia automáticamente este vector con un tamaño igual al número de argumentos,
2. para cada argumento se crea un objeto de tipo `String` con el argumento, y
3. dicho objeto pasa a ser referenciado por el elemento del vector `args` correspondiente.

Teniendo en cuenta la clase `Sistema_Solar` definida en el apartado 3.5, si ejecutamos la clase `Planetas` con el siguiente comando y argumentos:

```
> java Planetas Sol Mercurio Venus Tierra Marte Júpiter Saturno Urano
Neptuno Plutón
```

la salida por pantalla sería la siguiente:

```
Estrella del sistema solar = Sol
Planeta 1 = Mercurio
Planeta 2 = Venus
Planeta 3 = Tierra
Planeta 4 = Marte
Planeta 5 = Júpiter
Planeta 6 = Saturno
Planeta 7 = Urano
Planeta 8 = Neptuno
Planeta 9 = Plutón
```

Los argumentos pasados a una aplicación JAVA siempre son cadenas de caracteres. Si se desea pasar otros tipos de datos como números es necesario realizar una conversión de tipo `String` al tipo de dato deseado. El paquete `java.util` contiene clases que proporcionan esta funcionalidad. Por ejemplo, mediante

```
int n = Integer.parseInt(args[0]);
```

se obtiene el valor entero del primer argumento[3].

3.9. Ejercicios resueltos

Ejercicio 1 Implementación de una lista simple (simplemente enlazada) de objetos de clase `Elemento` que añade nuevos elementos en la cabeza (`void add(Elemento e)` y los extrae del final `Elemento get()`. La clase `ListaSimple` debe tener un método para obtener el número de elementos en la lista actual.

Solución:

[3]Esto es así suponiendo que la cadena de texto corresponde a un número entero. En realidad, este código no puede ejecutarse aisladamente, es necesario realizarlo bajo el control de excepciones que se ve en el Capítulo 5.

```
class Elemento {
}

class Elem {
  Elem sig;
  private Elemento elemento;

  public Elem( Elemento elemento, Elem e ) {
    this.elemento = elemento;
    sig = e;
  }

  public Elemento get() {
    return elemento;
  }
}

class ListaSimple {
  private Elem primero;
  private int elementos;

  public ListaSimple() {
    primero = null;
    elementos = 0;
  }

  public void add( Elemento e ) {
    primero = new Elem( e, primero );
    elementos++;
  }

  public Elemento get( ) {
    Elemento elemento = null;
    if( primero!=null ) {
      if( primero.sig==null ) {
        elemento = primero.get();
        primero = null;
      } else {
        Elem ant = primero;
        Elem e = primero;
        while( e.sig!=null ) {
          ant = e;
          e = e.sig;
        }
        elemento = e.get();
        ant.sig = null;
      }
      elementos--;
    }
    return elemento;
  }

  public int size() {
    return elementos;
```

```
    }
}
```

Ejercicio 2 Seguir los siguientes pasos para crear una clase dentro de un paquete definido por el usuario y utilizarla desde otra (las instrucciones proporcionadas son para un terminal de usuario linux).

- Crear un directorio para el paquete:

```
mkdir mi_paquete
```

- Entrar en dicho directorio

```
cd mi_paquete
```

y crear la clase siguiente:

```
package mi_paquete;

public class Mi_clase {
}
```

Compilarla de manera que el fichero `Mi_clase.class` se quede en dicho directorio.

```
javac Mi_clase.java
```

- Volver al directorio anterior

```
cd ..
```

y crear la clase

```
import mi_paquete.Mi_clase;

class Clase_principal {
  public static void main(String[] args) {
    Mi_clase a = new Mi_clase();
  }
}
```

- Compilar la clase principal y ejecutarla:

```
javac Clase_principal.java
java Clase_principal
```

Si no aparecen errores en la compilación y la ejecución es que los pasos se han seguido correctamente. El resultado ha sido la creación de un paquete con una clase que puede ser utilizada desde el paquete por defecto (paquete sin nombre) al que pertenece la clase principal de la aplicación JAVA. La variable `CLASSPATH` debe contener un punto en la lista de directorios.

Ejercicio 3 Implementar una estructura tipo *pila* de números enteros sin talla máxima. Utilizar una lista enlazada y clases internas.

Solución:

```
class Pila {
  private Elemento primero;
  private int n;

  class Elemento {
    Elemento next;
    int valor;
    Elemento( int valor ) {
      next = primero;
      this.valor = valor;
    }
    int get() {
      primero = next;
      return valor;
    }
    int getValor() {
      return valor;
    }
  }

  public Pila( ) {
    primero = null;
    n = 0;
  }

  public void add( int valor ) {
    primero = new Elemento( valor );
    n++;
  }

  public int get( ) {
    int a = -11111;
    if( n>0 ) {
      a = primero.get();
      n--;
    }
    return a;
  }

  public int top( ) {
    int a = -11111;
    if( n>0 ) {
      a = primero.getValor();
    }
    return a;
  }
}
```

Puede observarse que la clase `Pila` definida ahora posee la misma interfaz que la clase `Pila` de la página 69, sólo que el constructor ahora no recibe argumentos. La estructura definida ahora utiliza una lista enlazada de objetos que son creados cada vez que se añade

un nuevo número entero. Los elementos se eliminan de la lista cuando se extrae un elemento. Utilizando una lista enlazada se elimina la restricción de tener una talla máxima de elementos para la pila. Obsérvese que desde la clase interna `Elemento` se accede al atributo `primero` de la clase envolvente `Pila`.

3.10. Ejercicios propuestos

Ejercicio 1 Tomando como ejemplo la clase `Pila` mostrada en el apartado 3.4.1 (página 69), implementar la clase `Cola` para formar una cola de tipo FIFO (*First In First Out*) de números enteros. Esta estructura de datos se caracteriza porque el elemento que se extrae (método `get()`) debe ser el más antiguo de la cola.

Ejercicio 2 Implementar una clase `Lista` con la misma interfaz que la clase `Lista` del ejercicio resuelto 3.9 pero que añada elementos al final y los extraiga de la cabeza.

Ejercicio 3 Utilizando el ejercicio anterior, modificar la clase `Elemento` para que almacene objetos de tipo `String` y crear un constructor de `ListaSimple` a partir de los argumentos pasados. En el programa principal crear una lista con la clase `Lista` implementada con los argumentos pasados por linea de comandos.

Ejercicio 4 Crear una clase que almacene en una variable el número de veces que es instanciada.

Ejercicio 5 Tomando como base el Ejercicio 2 crear un subpaquete de `mi_paquete` que contenga al menos una clase y utilizar la misma en la clase principal.

Capítulo 4

Herencia y Polimorfismo

Tal y como se ha visto en capítulos anteriores, una clase representa un conjunto de objetos que comparten una misma estructura y comportamiento. La estructura se determina mediante un conjunto de variables denominadas *atributos*, mientras que el comportamiento viene determinado por los *métodos*. Ya se ha visto en capítulos anteriores que esta forma de estructurar los programas ofrece grandes ventajas, pero lo que realmente distingue la programación orientada a objetos (POO) de otros paradigmas de programación es la capacidad que tiene la primera de definir clases que representen objetos que comparten **sólo una parte** de su estructura y comportamiento, esto es, que representen objetos con ciertas similitudes, pero no iguales. Estas similitudes pueden expresarse mediante el uso de la ***herencia*** y el ***polimorfismo***. Estos dos mecanismos son, sin lugar a dudas, pilares básicos de la POO.

4.1. Herencia

La herencia es el mecanismo que permite derivar una o varias clases, denominadas *subclases*, de otra más genérica denominada *superclase*. La superclase reúne aquellos atributos y métodos comunes a una serie de subclases, mientras que estas últimas únicamente definen aquellos atributos y métodos propios que no hayan sido definidos en la superclase. Se dice que una subclase **extiende** el comportamiento de la superclase. En ocasiones se utiliza el término *clase base* para referirse a la superclase y *clase derivada* para referirse a la subclase. En la figura 4.1 se representa gráficamente el concepto de herencia.

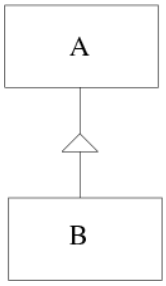

Figura 4.1: La clase B (subclase o clase derivada) hereda los atributos y métodos de la clase A (superclase o clase base).

4.1.1. Conceptos básicos

La herencia en JAVA se implementa simplemente especificando la palabra reservada `extends` en la definición de la subclase:

```
class nombreSubclase extends nombreSuperclase {
  // Atributos y métodos específicos de la subclase
}
```

Por ejemplo:

```
class Esfera {
  private double radio;

  public double setRadio( double r ) { radio = r; }

  public double superficie() {
    return 4*Math.PI*radio*radio;
  }
}

class Planeta extends Esfera {
  private int numSatelites;

  public int setNumSatelites( int ns ) { numSatelites = ns; }
}
```

En este ejemplo la clase `Planeta` hereda todos los métodos y atributos definidos en `Esfera` que no sean privados. Al fin y al cabo, un planeta también es una esfera (despreciaremos las pequeñas deformaciones que pueda tener). En el siguiente fragmento de código vemos cómo es posible acceder a los atributos y métodos heredados.

```
Planeta tierra = new Planeta();
tierra.setRadio( 6378 ); // (Km.)
tierra.setNumSatelites( 1 );
System.out.println("Superficie = " + tierra.superficie());
```

En el ejemplo anterior puede observarse cómo los objetos de la clase `Planeta` utilizan los métodos heredados de `Esfera` (`setRadio` y `superficie`) exactamente del mismo modo a como utiliza los definidos en su misma clase (`setNumSatelites`).

Cuando se define un esquema de herencia, la subclase hereda todos los miembros (métodos y atributos) de la superclase que no hayan sido definidos como privados. Debe observarse que en el ejemplo anterior la clase `Planeta` no hereda el atributo `radio`. Sin embargo, esto no quiere decir que los planetas no tengan un radio asociado, sino que no es posible acceder directamente a ese atributo. En realidad los objetos de tipo `Planeta` pueden establecer su radio a través del método `setRadio`. Debe remarcarse que es una práctica muy habitual definir como privados los atributos de una clase e incluir métodos públicos que den acceso a dichos atributos, tal y como se explicó en el apartado 3.4.

Veamos un nuevo ejemplo sobre el uso de la herencia. Supongamos que queremos implementar un juego en el que una nave debe disparar a ciertas amenazas que se le aproximan. De forma muy simplista, podríamos definir el comportamiento de nuestra nave mediante el siguiente código:

```
class Nave {
  private int posX, posY;  // Posición de la nave en la pantalla
  private int municion;
  Nave() {
    posX = 400;
    posY = 50;
    municion = 100;
  }
  void moverDerecha( int dist ) {
    posX += dist;
  }
  void moverIzquierda( int dist ) {
    posX -= dist;
  }
  void disparar() {
    if( municion > 0 ) {
      municion--;
      // Código necesario para realizar disparo
    }
  }
  void dibujar() {
    // Obviaremos la parte gráfica y nos limitaremos a mostrar
    // un texto por la consola
    System.out.println("Nave en " + posX + "," + posY);
    System.out.println("Munición restante: " + municion);
  }
}
```

Supongamos ahora que tras superar cierto nivel en el juego, podemos disponer de otra nave con mayores prestaciones (por ejemplo, un escudo protector). En este caso podemos tomar como base la nave anterior e implementar una segunda clase que contenga únicamente la nueva funcionalidad.

```
class NaveConEscudo extends Nave {
  private boolean escudo;
  NaveConEscudo() {
    escudo = false;
  }
  void activarEscudo() {
    escudo = true;
  }
  void desactivarEscudo() {
    escudo = false;
  }
}
```

Ahora sería posible hacer uso de esta nueva nave en nuestro juego.

```
class Juego {
  public static void main(String[] args) {
    NaveConEscudo miNave = new NaveConEscudo();
    miNave.moverDerecha(20);
    miNave.disparar();
    miNave.activarEscudo();
    miNave.dibujar();
```

```
    }
}
```

La salida del programa anterior sería:

```
Nave en posición 420 50
Municion restante: 99
```

El aspecto interesante del ejemplo anterior es observar que la subclase `NaveConEscudo` incluye todos los elementos de la clase `Nave` tal y como si hubiesen sido definidos en la propia clase `NaveConEscudo`. Esta característica permite definir con un mínimo esfuerzo nuevas clases basadas en otras ya implementadas, lo que redunda en la **reutilización de código**. Además, si en el uso de la herencia nos basamos en clases cuyo código ya ha sido verificado y el cual se supone exento de errores, entonces será más fácil crear programas robustos, ya que únicamente deberemos preocuparnos por validar la nueva funcionalidad añadida en la subclase.

Por supuesto sería posible crear objetos de tipo `Nave` en lugar de `NaveConEscudo`, pero en este caso no se podría ejecutar el método `nave.activarEscudo()`. En el ejemplo mostrado, los objetos de la clase `NaveConEscudo` disparan y se dibujan exactamente igual a como lo hacen los objetos de la clase `Nave`, ya que estos métodos han sido heredados.

Podríamos pensar que a nuestra nueva nave no se le permita disparar cuando tiene activado el escudo, o que se dibuje de forma distinta a como lo hacía la nave básica. Ello implica que la clase `NaveConEscudo` no sólo introduce nuevas prestaciones respecto de la clase `Nave`, sino que además debe modificar parte del comportamiento ya establecido. En el apartado 4.2 veremos qué mecanismos nos permiten abordar este problema.

Hay que mencionar que en Java no existe la herencia múltiple debido a los problemas que ello genera. Si se permitiera un esquema de herencia como el mostrado en la figura 4.2 implicaría que la clase `C` heredaría tanto los métodos definidos en `A` como los definidos en `B`. Podría ocurrir que tanto `A` como `B` contuvieran un método con el mismo nombre y parámetros pero distinta implementación (distinto código), lo que provocaría un conflicto acerca de cuál de las dos implementaciones heredar.

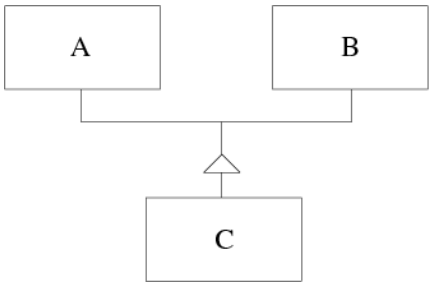

Figura 4.2: La herencia múltiple NO está permitida en Java.

4.1.2. Uso de la palabra reservada `super`

Una subclase puede referirse a la superclase inmediata mediante el uso de la palabra reservada `super`. Esta palabra reservada puede utilizarse de dos formas:

- Para llamar al constructor de la superclase.

```
super( lista_de_argumentos_del_constructor );
```

- Para acceder a un miembro de la superclase que ha sido ocultado. Por ejemplo, si declaramos en la subclase un miembro con el mismo nombre que tiene en la superclase, el miembro de la superclase quedará ocultado. En este caso sería posible invocarlo mediante el uso de `super`.

```
super.atributo = . . . ;
super.metodo( lista_de_argumentos );
```

Estos dos modos de usar la palabra `super` se verán con más detalle en los apartados 4.1.3 y 4.2.1 respectivamente.

4.1.3. Constructores y herencia

A diferencia de lo que ocurre con los métodos y atributos no privados, los constructores no se heredan. Además de esta característica, deben tenerse en cuenta algunos aspectos sobre el comportamiento de los constructores dentro del contexto de la herencia, ya que dicho comportamiento es sensiblemente distinto al del resto de métodos.

Orden de ejecución de los constructores

Cuando existe una relación de herencia entre diversas clases y se crea un objeto de una subclase S, se ejecuta no sólo el constructor de S sino también el de todas las superclases de S. Para ello se ejecuta en primer lugar el constructor de la clase que ocupa el nivel más alto en la jerarquía de herencia y se continúa de forma ordenada con el resto de las subclases[1]. El siguiente ejemplo ilustra este comportamiento:

```
class A {
  A() { System.out.println("En A"); }
}

class B extends A {
  B() { System.out.println("En B"); }
}

class C extends B {
  C() { System.out.println("En C"); }
}

class Constructores_y_Herencia {
  public static void main(String[] args) {
    C obj = new C();
  }
}
```

La salida de este programa sería:

```
En A
En B
En C
```

[1]Para ser más preciso, tal y como se verá más adelante, en primer lugar se examinan los constructores comenzando por las subclases que ocupan un nivel jerárquico más bajo. Con ello se determina el constructor que debe ejecutarse en la superclase inmediata (en caso de que haya más de un constructor en dicha superclase). Finalmente, una vez decidido qué constructor debe utilizarse en cada clase, se ejecutan comenzando por la clase de mayor nivel jerárquico.

¿Qué constructor se ejecuta en la superclase? Uso de super()

Como ya se ha visto en capítulos anteriores, es posible que una misma clase tenga más de un constructor (sobrecarga del constructor), tal y como se muestra en el siguiente ejemplo:

```
class A {
  A() { System.out.println("En A"); }
  A(int i) { System.out.println("En A(i)"); }
}

class B extends A {
  B() { System.out.println("En B"); }
  B(int j) { System.out.println("En B(j)"); }
}
```

La cuestión que se plantea es: ¿qué constructores se invocarán cuando se ejecuta la sentencia `B obj = new B(5);` ? Puesto que hemos creado un objeto de tipo `B` al que le pasamos un entero como parámetro, parece claro que en la clase `B` se ejecutará el constructor `B(int j)`. Sin embargo, puede haber confusión acerca de qué constructor se ejecutará en `A`. La regla en este sentido es clara: mientras no se diga explícitamente lo contrario, en la superclase se ejecutará siempre el constructor sin parámetros. Por tanto, ante la sentencia `B obj = new B(5)`, se mostraría:

```
En A
En B(j)
```

Hay, sin embargo, un modo de cambiar este comportamiento por defecto para permitir ejecutar en la superclase un constructor diferente. Para ello debemos hacer uso de la sentencia `super()`. Esta sentencia invoca uno de los constructores de la superclase, el cual se escogerá en función de los parámetros que contenga `super()`.

En el siguiente ejemplo se especifica de forma explícita el constructor que deseamos ejecutar en `A`:

```
class A {
  A() {
    System.out.println("En A");
  }
  A(int i) {
    System.out.println("En A(i)");
  }
}

class B extends A {
  B() {
    System.out.println("En B");
  }
  B(int j) {
    super(j); // Ejecutar en la superclase un constructor
              // que acepte un entero
    System.out.println("En B(j)");
  }
}
```

En este caso la sentencia `B obj = new B(5)` mostraría:

```
En A(i)
En B(j)
```

Mientras que la sentencia `B obj = new B()` mostraría:

```
En A()
En B()
```

Ha de tenerse en cuenta que en el caso de usar la sentencia `super()`, ésta deberá ser obligatoriamente la primera sentencia del constructor. Esto es así porque se debe respetar el orden de ejecución de los constructores comentado anteriormente.

En resumen, cuando se crea una instancia de una clase, para determinar el constructor que debe ejecutarse en cada una de las superclases, en primer lugar se exploran los constructores en orden jerárquico ascendente (desde la subclase hacia las superclase). Con este proceso se decide el constructor que debe ejecutarse en cada una de las clases que componen la jerarquía. Si en algún constructor no aparece explícitamente una llamada a `super(lista_de_argumentos)` se entiende que de forma implícita se está invocando a `super()` (constructor sin parámetros de la superclase). Finalmente, una vez se conoce el constructor que debe ejecutarse en cada una de las clases que componen la jerarquía, éstos se ejecutan en orden jerárquico descendente (desde la superclase hacia las subclases).

Pérdida del constructor por defecto

Podemos considerar que todas las clases en Java tienen de forma implícita un constructor por defecto sin parámetros y sin código. Ello permite crear objetos de dicha clase sin necesidad de incluir explícitamente un constructor. Por ejemplo, dada la clase

```
class A {
  int i;
}
```

no hay ningún problema en crear un objeto

```
A obj = new A();
```

ya que, aunque no lo escribamos, la clase `A` lleva implícito un constructor por defecto:

```
class A {
  int i;
  A(){} // Constructor por defecto
}
```

Sin embargo, es importante saber que dicho constructor por defecto se pierde si escribimos cualquier otro constructor. Por ejemplo, dada la clase

```
class A {
  int i;
  A( int valor) {
    i=valor;
  }
}
```

La sentencia `A obj = new A()` generará un error de compilación, ya que en este caso no existe ningún constructor en `A` sin parámetros. Hemos perdido el constructor por defecto. Lo correcto sería, por ejemplo, `A obj = new A(5)`.

Esta situación debe tenerse en cuenta igualmente cuando exista un esquema de herencia. Imaginemos la siguiente jerarquía de clases:

```
class A {
  int i;
  A( int valor) {
    i=valor;
  }
}
class B extends A {
  int j;
  B( int valor) {
    j=valor;
  }
}
```

En este caso la sentencia `B obj = new B(5)` generará igualmente un error ya que, puesto que no hemos especificado un comportamiento distinto mediante `super()`, en A debería ejecutarse el constructor sin parámetros, sin embargo, tal constructor no existe puesto que se ha perdido el constructor por defecto. La solución pasaría por sobrecargar el constructor de `A` añadiendo un constructor sin parámetros,

```
class A {
  int i;
  A() {
    i=0;
  }
  A( int valor) {
    i=valor;
  }
}
class B extends A {
  int j;
  B( int valor) {
    j=valor;
  }
}
```

o bien indicar explícitamente en el constructor de `B` que se desea ejecutar en `A` un constructor que recibe un entero como parámetro, tal y como se muestra a continuación:

```
class A {
  int i;
  A( int valor) {
    i=valor;
  }
}
class B extends A {
  int j;
  B( int valor) {
    super(0);
    j=valor;
  }
}
```

4.1.4. Modificadores de acceso

Los modificadores de acceso `public` y `private` descritos en el apartado 3.4.1 del capítulo anterior se utilizan para controlar el acceso a los miembros de una clase. Existe un tercer modificador de acceso, `protected` que tiene sentido utilizar cuando entra en juego la herencia. Cuando se declara un atributo o método `protected`, dicho elemento no será visible desde aquellas clases que no deriven de la clase donde fue definido y que, además, se encuentren en un paquete diferente.

En la tabla 4.1 se resume la visibilidad de los atributos y métodos en función de sus modificadores de acceso.

	`private`	Sin modificador (*friendly*)	`protected`	`public`
Misma clase	Sí	Sí	Sí	Sí
Subclase del mismo paquete	No	Sí	Sí	Sí
No subclase del mismo paquete	No	Sí	Sí	Sí
Subclase de diferente paquete	No	No	Sí	Sí
No subclase de diferente paquete	No	No	No	Sí

Tabla 4.1: Acceso a miembros de una clase.

4.1.5. La clase `Object`

La clase `Object` es una clase especial definida por JAVA. `Object` es la clase base de todas las demás clases, de modo que cualquier clase en JAVA deriva directa o indirectamente de ésta, sin necesidad de indicarlo explícitamente mediante la palabra `extends`. Esto significa que en JAVA existe un único árbol jerárquico de clases en el que están incluidas todas, donde la clase `Object` figura en la raíz de dicho árbol.

A continuación se muestra un ejemplo en el que se observa que las clases que no llevan la cláusula `extends` heredan implícitamente de `Object`. La figura 4.3 muestra el esquema de herencia de este ejemplo.

```
class A {  // Al no poner extends, deriva implícitamente de Object
  . . .
}
class B extends A {  // Deriva de A y, por tanto, de Object
  . . .
}
```

En la clase `Object` existen varios métodos definidos, entre los cuales figuran:

`boolean equals(Object o)`: compara dos objetos para ver si son equivalentes y devuelve un valor booleano.

`String toString()`: devuelve una cadena de tipo `String` que conteniene una descripción del objeto. Este método se llama implícitamente cuando se trata de imprimir un objeto con `println()` o cuando se opera con el objeto en una "suma" (concatenación) de cadenas.

`void finalize()`: este método es invocado automáticamente por el *garbage collector* cuando se determina que ya no existen más referencias al objeto.

Para que estos métodos sean de utilidad, las subclases de `Object` deberán *sobreescribirlos*, tal y como se explica en el apartado 4.2.7.

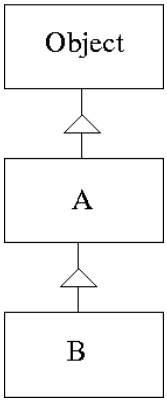

Figura 4.3: En Java existe un único árbol jerárquico, en el que la clase `Object` ocupa el nivel más alto.

4.2. Polimorfismo

Una de las características de la herencia es que una referencia de una clase base puede utilizarse para referenciar objetos de cualquir otra clase heredada (este concepto se explica en el apartado 4.2.3). Esta característica, tan simple y a la vez tan potente, permite implementar lo que se conoce como **polimorfismo**.

El término *polimorfismo* hace referencia a la capacidad que tienen algunos métodos de presentar "múltiples formas", en el sentido de que una llamada a un mismo método puede ofrecer comportamientos distintos en función del contexto en el que se ejecute. El polimorfismo, como se verá más adelante, dota a los lenguajes orientados a objetos de una potencia de la que carecen otros lenguajes, ya que permite realizar ciertas abstracciones sobre los tipos de datos con los que se trabaja.

Es imposible tratar y entender el polimorfismo sin antes abordar una serie de conceptos relacionados. Estos conceptos son independientes del tema del polimorfismo, pero necesarios para su comprensión.

4.2.1. Sobreescritura de métodos

La herencia permite crear subclases que contienen todos los atributos y métodos no privados de la clase superior. Habitualmente en la subclase se definirán, además, atributos y métodos propios que no existían en la superclase. En ocasiones, además interesa que la subclase modifique alguno de los métodos heredados para que tengan un comportamiento distinto. Esto se realiza mediante la *sobreescritura de métodos*. Simplemente consiste en implementar en la subclase un método que ya existía en la superclase.

```
class Esfera {
  double radio;

  void mostrarValores() {
    System.out.println("R=" + radio);
  }
}

class Planeta extends Esfera {
```

```
    int numSatelites;

    void mostrarValores() {
      System.out.println("R=" + radio + "Num. Sat.=" + numSatelites);
    }
  }
```

En este ejemplo la clase `Planeta` hereda el método `mostrarValores` de `Esfera`, pero luego lo sobreescribe para poder mostrar tanto el radio como el número de satélites. En esta situación el método `mostrarValores` ejecutará un código distinto dependiendo del objeto sobre el que se invoque.

```
public class EjemploSobreescritura {
  public static void main(String[] args) {
    Esfera e = new Esfera();
    e.radio=10;
    Planeta p = new Planeta();
    p.radio=6378;
    p.numSatelites=1;
    e.mostrarValores();  // Muestra los valores de la esfera
    p.mostrarValores();  // Muestra los valores del planeta
  }
}
```

El resultado de este programa sería

```
R=10
R=6378 Num. Sat.=1
```

En el ejemplo dado en el apartado 4.1.1 se definía la clase `NaveConEscudo` a partir de la clase `Nave`. Nuestra nueva nave permitía definir un escudo tal y como se muestra a continuación:

```
class Nave {
  int posX, posY;
  int municion;
  Nave() { . . . }
  void moverDerecha( int dist ) { . . . }
  void moverIzquierda( int dist ) { . . . }
  void disparar() { if( municion > 0) municion--; }
  void dibujar() { . . . }
}

class NaveConEscudo extends Nave {
  boolean escudo;
  NaveConEscudo() { escudo = false; }
  void activarEscudo() { escudo = true; }
  void desactivarEscudo() { escudo = false; }
}
```

En este ejemplo podría ser interesante que `NaveConEscudo` modificase el comportamiento de alguno de los métodos heredados de `Nave`. Por ejemplo, que no pueda disparar si tiene el escudo activado. Para ello, tal y como se ha visto, bastaría con *sobreescribir* el método correspondiente en la subclase:

```
class NaveConEscudo extends Nave {
  boolean escudo;
  NaveConEscudo() { escudo = false; }
  void activarEscudo() { escudo = true; }
  void desactivarEscudo() { escudo = false; }
  // Sobreescribimos el método disparar
  void disparar() {
    if( escudo == false && municion > 0 )
      municion--;
  }
}
```

De este modo cada nave tiene su propia versión del método `disparar`. De forma similar podría sobreescribirse el método `dibujar` para que cada tipo de nave tuviese una representación gráfica distinta.

Reemplazo y refinamiento

La sobreescritura de un método podemos enfocarla desde dos puntos de vista: reescribir el método completamente ignorando el código que hubiese en la superclase, o ampliar el código existente en la superclase con nuevas instrucciones. En el primer caso hablaremos de **reemplazo** y en el segundo de **refinamiento**. Los ejemplos vistos anteriormente corresponden a sobreescritura mediante reemplazo. Para implementar la sobreescritura mediante refinamiento se debe invocar desde el método sobreescrito de la subclase el método correspondiente de la superclase mediante el uso de **super**. Por ejemplo podríamos pensar que una `NaveConEscudo` se dibuja como una `Nave` con algunos elementos extras. En este caso sería más interesante el refinamiento que el reemplazo, ya que aprovecharíamos el código del método `dibuja` escrito para la clase `Nave`.

```
class Nave {
  . . .
  void dibujar() {
    // Código para dibujar una nave básica
  }
}

class NaveConEscudo extends Nave {
  . . .
  void dibujar() {
    super.dibujar(); // Primero dibujamos una nave básica.

    // Código para dibujar los elementos extra
    . . .
  }
}
```

Uso del modificador *final* para evitar la sobreescritura

Ya se vio que el modificador `final` puede aplicarse a la declaración de una variable para asegurarnos de que el valor de la misma no cambie tras su inicialización. Cuando entra en juego la herencia, puede tener sentido además aplicar el modificador `final` a la declaración de métodos. En este caso, un método declarado como `final` no puede ser sobreescrito en las subclases que hereden el método.

```
class A {
  final void metodo1() {
    . . .
  }
}
class B extends A {
  // Esta clase no puede sobreescribir metodo1()
}
```

4.2.2. Tipo estático y tipo dinámico

En este punto conviene realizar una distinción importante entre lo que es la declaración de una referencia y la instanciación de un objeto al cual apuntará dicha referencia. En todos los ejemplos vistos hasta ahora se declaraban referencias de un determinado tipo (una clase) y después se instanciaba un objeto de esa misma clase, el cual quedaba apuntado por la referencia. Por ejemplo:

```
Esfera e = new Esfera();
```

En este caso vemos que el tipo (clase) de la referencia coincide con el tipo de objeto que se crea, sin embargo, esto no tiene por qué ser siempre así. De hecho es posible declarar una referencia de una clase determinada e instanciar un objeto, no de dicha clase, sino de una subclase de la misma. Por ejemplo, dada la siguiente relación de clases

```
class Esfera {
  double radio;

  double superficie() {
    return 4*Math.PI*radio*radio;
  }
}

class Planeta extends Esfera {
  int numSatelites;
}
```

la siguiente declaración sería válida

```
Esfera e = new Planeta();
```

Conviene diferenciar el *tipo estático* de una variable del *tipo dinámico*. Denominamos *tipo estático* al tipo con el que se declara una variable referencia, y *tipo dinámico* al tipo del objeto al que apunta dicha referencia. En el primer ejemplo mostrado, tanto el tipo estático como el tipo dinámico de la variable `e` es `Esfera`. Por el contrario, en el segundo ejemplo se ha declarado una variable `e` de tipo `Esfera` (tipo estático), pero, el objeto creado es de tipo `Planeta` (tipo dinámico).

4.2.3. La conversión hacia arriba

Tal y como se ha visto en el apartado anterior, es posible declarar una referencia de un tipo estático determinado e instanciar un objeto, no de dicha clase, sino de una subclase de la misma. Esto es, es posible que el tipo dinámico sea una subclase del tipo estático. Este modo de referenciar objetos en los que el tipo referencia corresponde a una superclase del objeto referenciado es lo que denominamos *conversión hacia arriba*. El término *hacia arriba* viene

motivado por la forma en que se representa gráficamente la herencia, donde las superclases aparecen arriba de las subclases. En la *conversión hacia arriba* se referencia un objeto dado mediante una referencia perteneciente a un tipo o clase que jerárquicamente está "arriba" de la clase correspondiente al objeto creado.

Cuando se utiliza la *conversión hacia arriba* es importante tener en cuenta una limitación: si creamos un objeto de un tipo determinado y empleamos una referencia de una superclase para acceder al mismo, a través de dicha referencia únicamente podremos "ver" la parte del objeto que se definió en la superclase. Dicho de otro modo, el tipo estático limita la interfaz, esto es, la visibilidad que se tiene de los atributos y métodos de un objeto determinado.

Tomemos como ejemplo las siguientes clases:

```
class A {
  public void m1() {
    . . .
  }
}

class B extends A {
  public void m2() {
    . . .
  }
}
```

Supongamos ahora que en algún punto de nuestro programa se crean los siguientes objetos:

```
B obj1 = new B();  // Tipo estático y dinámico de obj1: B
A obj2 = new B();  // Tipo estático A y tipo dinámico B
B obj3 = new A();  // Tipo estático B y tipo dinámico A. ¡ERROR!
```

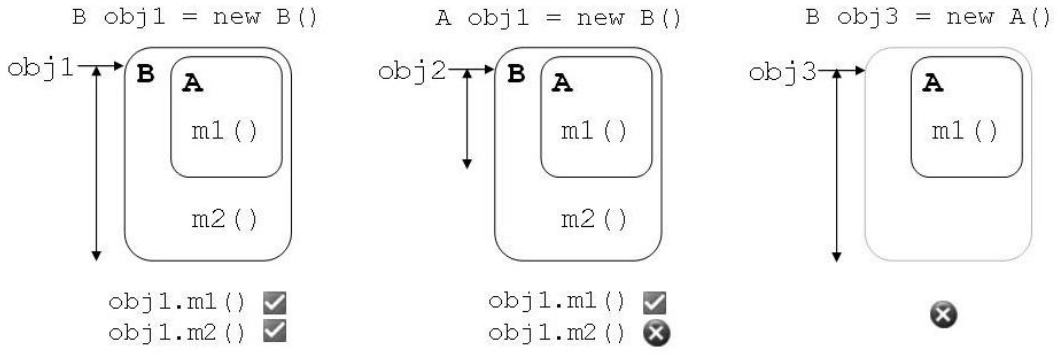

Figura 4.4: Dadas una superclase `A` y una subclase `B`, se crean objetos de distinta manera. (a) Tipo estático=`B`, tipo dinámico=`B`: se tiene acceso a todo el objeto. (b) Tipo estático=`A`, tipo dinámico=`B` (conversión hacia arriba): se tiene acceso únicamente a la parte de `A`. (c) Tipo estático=`B`, tipo dinámico=`A`: Error. Si se permitiera, se debería tener acceso a elementos que realmente no han sido creados.

En la figura 4.4 se muestra cómo, a partir de la referencia `obj1` se tiene acceso tanto al método `m1()` como a `m2()`. Sin embargo, con la referencia `obj2`, a pesar de que referencia a

un objeto de tipo `B`, únicamente se tiene acceso a los métodos definidos en `A`, ya que el tipo estático impone esta limitación.

Retomando el ejemplo de la `Esfera` del apartado anterior, sobreescribamos ahora alguno de los métodos definidos en la superclase. Por ejemplo, supongamos que se desea medir la superficie de un planeta con mayor precisión, teniendo en cuenta el achatamiento existente en los polos. Para ello podríamos redefinir la clase `Planeta` como sigue:

```
class Esfera {
  double radio;

  double superficie() {
    return 4*Math.PI*radio*radio;
  }
}

class Planeta extends Esfera {
  int numSatelites;
  double coeficienteAchatamiento;

  Planeta(double r, double c) {
    radio = r;
    coeficienteAchatamiento = c;
  }
  // Sobreescribimos el método superficie
  double superficie() {
    // Cálculo preciso de la superficie teniendo en cuenta
    // el coeficiente de achatamiento
  }
}
```

En esta situación, ¿qué código ejecutaría el siguiente fragmento de programa?

```
Esfera e = new Planeta(6378, 35.3);
double s = e.superficie();
```

El tipo estático `Esfera` contiene el método `superficie`, por lo tanto dicho método es visible a través de la referencia `e`, sin embargo es importante entender que se ejecutará el método del objeto que hemos creado (en este caso de tipo `Planeta`). Puesto que el método `superficie` ha sido sobreescrito en la clase `Planeta` se ejecutará esta nueva versión del método.

A modo de resumen diremos que el tipo estático dictamina QUÉ operaciones pueden realizarse, pero es el tipo dinámico el que determina CÓMO se realizan. Esta característica tiene una enorme transcendencia en la programación orientada a objetos ya que, como se verá en apartados posteriores, permite realizar ciertas abstracciones sobre los tipos de datos (clases) con los que se trabaja.

Existen algunas restricciones lógicas en la utilización de tipos estáticos y dinámicos distintos. Una variable referencia con un tipo estático determinado puede tener, como se ha visto, un tipo dinámico que sea derivado del tipo estático, pero no a la inversa. Esto es, la siguiente declaración no sería correcta:

```
Planeta p = new Esfera();
```

La restricción impuesta es lógica por varios motivos. Si se declara una referencia con el tipo estático `Esfera`, ésta puede ser utilizada para referenciar un `Planeta`. Es como decir

que "un planeta también es una esfera". Sin embargo, no tiene tanto sentido declarar una referencia de tipo `Planeta` e intentar referirse con ella a un objeto de tipo `Esfera`, ya que sería como decir que "cualquier esfera es un planeta".

Desde el punto de vista semántico del lenguaje existe una razón todavía más contundente. Si declaramos una referencia de tipo estático `Planeta` estamos indicando que con dicha referencia podemos acceder a cualquier miembro de la clase `Planeta` (el tipo estático indica qué se puede hacer), esto es, deberíamos poder hacer

```
Planeta p;
. . .
p.numSatelites=1;
```

Sin embargo, si dicha referencia la utilizamos para instanciar un objeto de tipo `Esfera`

```
Planeta p;
p = new Esfera();
```

en dicho objeto no existirá el atributo `numSatelites`, por lo que la sentencia

```
p.numSatelites=1;
```

resultaría incorrecta. De algún modo estaríamos dando al tipo estático un alcance o visibilidad mayor de lo que realmente existe en el tipo dinámico, tal y como se muestra en la figura 4.4(c), lo que generaría inconsistencias como la que acabamos de exponer.

¿Por qué es interesante la conversión hacia arriba?

La conversión hacia arriba no es un capricho del lenguaje, sino que tiene una razón de ser. De hecho es una de las características fundamentales de la POO, ya que es la base de cuestiones tan importantes como el polimorfismo o la abstracción. Aunque estos conceptos se estudiarán en detalle en los siguientes apartados, podemos avanzar que en ocasiones resulta muy interesante poder llevar a cabo ciertas acciones sin necesidad de saber exactamente con qué tipo de objeto estamos trabajando. Con la conversión hacia arriba ignoramos el tipo real del objeto y lo tratamos de un modo más genérico. Ello permite realizar ciertas abstracciones sobre los objetos con los que trabajamos, yendo de lo concreto a lo general.

Tomemos el siguiente ejemplo: imaginemos que tenemos una serie de *Dispositivos* (filtros, amplificadores, conversores A/D, codificadores, etc.) los cuales pueden conectarse en serie para procesar una señal eléctrica. Supongamos que la *cadena de proceso* permite cualquier secuencia de este tipo de dispositivos. Todos los dispositivos tienen en común que poseen una salida a la cual puede conectarse cualquier otro dispositivo. Por otro lado, cada dispositivo tendrá ciertas características particulares que lo diferenciarán del resto. Podríamos pensar en una clase base o superclase denominada `Dispositivo` de la que heredasen las clases `Filtro`, `Amplificador`, `Conversor` o `Decodificador`.

Puesto que todos ellos tienen una salida a la que conectar otro dispositivo, podríamos poner un atributo que represente esta conexión en la clase base:

```
class Dispositivo {
  Dispositivo salida;

  void conectar( Dispositivo d ) {
    salida = d;
  }

  // Resto de atributos y métodos
```

```
    . . .
  }
```

Lo interesante de este ejemplo es que el atributo `salida` es del tipo genérico `Dispositivo`, sin embargo, gracias a la conversión hacia arriba, esta referencia puede utilizarse para referenciar objetos de cualquier tipo de dispositivo (filtro, amplificador, etc.). De modo similar, el método `conectar` admite **cualquier tipo** de dispositivo. Podría parecer extraño que un método que acepta como parámetro objetos de un tipo determinado pueda llegar a aceptar objetos de otro tipo, sin embargo, tomando el ejemplo anterior, hemos de pensar que al fin y al cabo, un filtro también es un tipo de dispositivo.

4.2.4. Enlace dinámico y polimorfismo

Con todo lo expuesto en los apartados anteriores, ahora estamos en condiciones de entender el mecanismo que permite implementar el polimorfismo en tiempo de ejecución: el *enlace dinámico* o *selección dinámica de método*. Tomemos como ejemplo la siguiente jerarquía de clases en la que se definen distintas figuras geométricas:

```
class Figura {
  double area() {
    return 0;  // No sabemos qué área tiene una figura genérica
  }
}
class Rectangulo extends Figura {
  double alto, ancho;
  Rectangulo( double al, double an) {
    alto = al;
    ancho = an;
  }
  double area() {
    return alto*ancho;
  }
}
class Circulo extends Figura {
  double radio;
  Circulo(double r) {
    radio = r;
  }
  double area() {
    return Math.PI*radio*radio;
  }
}
```

Supongamos ahora que creamos el siguiente programa:

```
public class EnlaceDinamico {
  public static void main(String[] args) {
    // Creamos 10 referencias de tipo Figura
    Figura [] v = new Figura[10];

    // Rellenamos aleatoriamnente con rectángulos y círculos
    for(int i=0; i<10; i++) {
      double aleatorio = Math.random();
      if( aleatorio > 0.5 )
```

```
          v[i] = new Rectangulo(10,10);
        else
          v[i] = new Circulo(5);
      }

      // Mostramos las áreas de las figuras creadas
      for(int i=0; i<10; i++) {
        double a = v[i].area(); // Selección dinámica de método
        System.out.println("Area="+a);
      }
    }
  }
```

En este programa se crea un vector con 10 referencias de tipo `Figura`, esto es, el tipo estático de cada referencia `v[i]` es `Figura` (la sentencia `v = new Figura[10]` crea un vector con 10 referencias, pero éstas todavía no referencian ningún objeto). A continuación se crean objetos de tipo `Rectangulo` y `Circulo` los cuales quedan referenciados por las referencias contenidas en `v`. Por tanto, el tipo dinámico de cada referencia `v[i]` es o bien `Rectangulo` o bien `Circulo` (nótese el uso de la *conversión hacia arriba* estudiada en el apartado 4.2.3). Observemos que realmente no tenemos certeza sobre el tipo dinámico concreto ya que dicho tipo se ha elegido de forma aleatoria, lo único que sabemos es que, de un modo más genérico, todos ellos son `Figuras`. Finalmente se calcula el área de cada figura mediante la sentencia `v[i].area()`. En esta situación se tiene que el tipo estático de `v[i]` es `Figura`, y puesto que el método `area` está definido en dicho tipo, la sentencia `v[i].area()` es correcta (el tipo estático indica QUÉ podemos hacer). Sin embargo, ya se ha explicado que realmente se ejecutará el método `area` definido en el tipo dinámico (el tipo dinámico indica CÓMO se hace). Se ejecutará por tanto el método `area` del objeto que se haya creado en cada caso.

Llegados a este punto no es difícil observar que es imposible conocer en tiempo de compilación cuál de los métodos `area` se va a ejecutar en cada caso, ya que hasta que no se ejecute el programa no se van a conocer los tipos dinámicos de cada figura. En consecuencia debe existir un mecanismo que en tiempo de ejecución, y no en tiempo de compilación, sea capaz de establecer qué método ejecutar en función del tipo dinámico del objeto. Dicho mecanismo existe en la máquina virtual de Java y se conoce como *enlace dinámico* o *selección dinámica de método* y es el que permite llevar a cabo el polimorfismo en tiempo de ejecución.

El polimorfismo es una característica fundamental de los lenguajes orientados a objetos, que permite realizar ciertas abstracciones sobre los tipos de datos. Lo sorprendente del ejemplo anterior es que el programa principal no necesita saber qué tipo de objetos hay almacenados en el vector `v` para poder realizar ciertas operaciones con ellos (calcular el área en este caso), es suficiente con saber que, de forma genérica, son figuras. Es más, sería posible crear nuevos tipos de figuras, por ejemplo triángulos, y añadirlas en el vector `v`; el bucle que calcula el área de cada una de ellas seguiría funcionando sin necesidad de ningún cambio, a pesar de que cuando se escribió ni siquiera existía la clase `Triangulo`. Esto es así porque realmente a la hora de escribir el bucle que calcula las áreas no estamos concretando el método `area` que deseamos ejecutar, sino que lo estamos posponiendo hasta el último momento (la ejecución del programa) lo que permite en cualquier momento crear nuevas figuras, cada una con su propio método `area`.

Supongamos ahora que necesitamos un método que calcule si dos figuras cualesquiera tienen o no la misma área. Para ello podríamos añadir a la clase `Figura` el método `mismaArea`.

```
class Figura {
  double area() {
    return 0;  // No sabemos qué área tiene una figura genérica
```

```
    }
    boolean mismaArea(Figura otra) {
      double a1 = area();
      double a2 = otra.area();
      if( a1 == a2 )
        return true;
      else
        return false;
      // O simplemente: return ( a1==a2 );
    }
  }
  class Rectangulo extends Figura {
    // Igual que en el ejemplo anterior
  }
  class Circulo extends Figura {
    // Igual que en el ejemplo anterior
  }
```

Ahora la clase `Figura` implementa dos métodos, uno de los cuales (`area`) se sobreescribe en las subclases y otro (`mismaArea`) que se hereda tal cual. Observemos ahora el siguiente fragmento de código:

```
Figura f1 = new Rectangulo(10,10);
Figura f2 = new Circulo(5);
if( f1.mismaArea(f2) )
  System.out.println("Figuras con la misma área");
```

El método `mismaArea` definido en la clase `Figura` invoca a su vez al método `area`, pero ¿qué método `area` se ejecuta en este caso? Tratemos de averiguar qué método `area` se ejecuta en la llamada `double a1 = area()`. Un razonamiento erróneo podría llevar a pensar que, puesto que estamos ejecutando el método `mismaArea` que se encuentra en la clase `Figura` (ya que no ha sido sobreescrito en `Rectangulo` ni en `Circulo`) entonces cuando este mismo método invoca a su vez a `area` se estará ejecutando igualmente el método `area` definido en `Figura`.

El error de este razonamiento radica en pensar que se está ejecutando el método `mismaArea` de la clase `Figura`. Realmente dicho método se está ejecutando sobre un objeto de tipo `Rectangulo` ya que el tipo dinámico de `f1` corresponde a esta clase y, si bien es cierto que `Rectangulo` no sobreescribe el método `mismaArea`, sí que lo hereda. En otras palabras, `Rectangulo` tiene su propio método `mismaArea`, cuyo código es idéntico al método `mismaArea` definido en `Figura`. Una vez aclarado que se ejecuta el método `mismaArea` de `Rectangulo`, parece lógico pensar que la llamada al método `area` se ejecutará también sobre la clase `Rectangulo`. Ello podría verse más claro si en lugar de escribir

```
boolean mismaArea(Figura otra) {
  double a1 = area();
  . . .
}
```

escribimos el código equivalente

```
boolean mismaArea(Figura otra) {
  double a1 = this.area();
  . . .
}
```

Sabemos que `this` siempre hace referencia al objeto que invocó al método. Puesto que en nuestro ejemplo el método `mismaArea` fue invocado por `f1`, mentalmente podemos substituir `this.area()` por `f1.area()`, donde es obvio que `f1` referencia a un objeto de tipo `Rectangulo`. Por otro lado, en la llamada

```
double a2 = otra.area();
```

la referencia `otra` toma su valor de `f2`, la cual a su vez referencia a un objeto de tipo `Circulo`. En este caso, por tanto, se ejecutará el método `area` de la clase `Circulo`.

Nuevamente, lo sorprendente del método `mismaArea` es que puede comparar las áreas de cualquier tipo de figura, aunque todavía no hayan sido definidas. Si en un futuro definimos la clase `Dodecaedro` el método `mismaArea` podrá trabajar con este tipo de objetos sin necesidad de cambiar una sola línea de su código. Esto es así porque el método `area` es un método *polimórfico* (cambia de forma, o de código, en función del contexto en el que se ejecuta). El único requerimiento será que `Dodecaedro` herede de `Figura` y que sobreescriba su propia implementación del método `area`.

4.2.5. Clases abstractas

Si observamos con detenimiento el ejemplo anterior podríamos pensar que no tiene mucha lógica incluir el método `area` en la clase `Figura`, ya que difícilmente puede calcularse el área de una figura genérica. En el ejemplo se ha adoptado el criterio de devolver 0. La razón de haber incluido el método `area` en la clase `Figura` es únicamente para poder implementar el *enlace dinámico*. De no haber incluido este método, la sentencia `v[i].area()` hubiera generado un error de compilación puesto que el tipo estático (`Figura` en este caso) limita la visibilidad de los métodos y atributos a los que se tiene acceso. Vemos, por tanto, que es necesario incluir el método `area` en la clase `Figura`, aunque realmente nunca se va a ejecutar. Nótese que no sería válido dejar el método sin código

```
double area() {}
```

ya que, al menos, debe devolver un valor de tipo `double`, tal y como indica el tipo del método. ¿No sería posible poder definir métodos sin código para este tipo de situaciones? La respuesta es sí, y estos métodos especiales son los denominados métodos *abstractos*.

Un método abstracto es un método del que únicamente existe la cabecera y que carece de código. El objetivo de incluir este tipo de métodos en una clase es permitir el polimorfismo en tiempo de ejecución. La idea, tal y como se verá más adelante, es obligar a que las subclases que extiendan a la clase donde está definido lo sobreescriban.

En Java se define un método abstracto simplemente anteponiendo la palabra reservada *abstract* delante del método, y finalizando el mismo con un punto y coma tras el paréntesis que cierra la lista de parámetros.

```
abstract tipo nombre_metodo(lista_argumentos);
```

En el ejemplo anterior hubiéramos podido definir el método `area` abstracto del siguiente modo:

```
abstract double area();
```

El hecho de definir un método abstracto convierte en abstracta a la clase en la cual se define, lo que obliga a poner también la palabra *abstract* delante del nombre de la clase. Nuestra nueva clase `Figura` quedaría, por tanto,

```
abstract class Figura {
  abstract double area();
  boolean mismaArea(Figura otra) {
    double a1 = area();
    double a2 = otra.area();
    return ( a1==a2 );
  }
}
```

y el resto de subclases (`Rectangulo`, `Circulo`) no sufrirían modificación alguna.

Ante esta nueva implementación de la clase `Figura` cabría preguntarse qué efectos tendría el siguiente fragmento de código:

```
Figura f = new Figura();
System.out.println( "Area=" + f.area() );
```

Simplemente esto generaría un error de compilación. Ha de saberse que Java no permite instanciar objetos de clases abstractas. Esta restricción es lógica, ya que el objetivo de las clases abstractas es que sirvan de base para definir nuevas subclases y no el de utilizarlas de forma aislada. De algún modo, una clase abstracta es una clase inacabada, de la que se espera que futuras subclases terminen por implementar los métodos abstractos que tienen definidos. No parece lógico, por tanto, crear objetos de algo que todavía no está completamente definido. Básicamente la clase `Figura` representa la idea abstracta de una figura, y no hay manera de calcular el área de algo que todavía no sabemos lo que es. Únicamente es posible calcular el área de figuras concretas como rectángulos o círculos.

Debido a que puede llegar a crear confusión, en este punto vale la pena aclarar la diferencia entre la sentencia

```
Figura f = new Figura();
```

y

```
Figura [] v = new Figura[100];
```

La primera de ellas generaría un error de compilación ya que, como acaba de explicarse, no es posible instanciar objetos de una clase abstracta. En el segundo caso, sin embargo, es importante entender que no se están creando 100 objetos de tipo `Figura` sino 100 referencias de dicho tipo, lo cual es perfectamente válido. Estas referencias, por otro lado, deberán emplearse necesariamente para instanciar objetos de alguna subclase no abstracta de `Figura`. Por ejemplo,

```
v[i] = new Rectangulo(10,10);
```

Es importante tener en cuenta ciertas implicaciones que tiene la utilización de clases abstractas:

- Una clase que posea al menos un método abstracto debe ser declarada como abstracta.
- Los métodos abstractos no tienen cuerpo y deben ser implementados por las subclases de la clase abstracta.
- Si una subclase que extiende una clase abstracta no implementa alguno de los métodos abstractos declarados en la superclase, entonces debe ser declarada también como abstracta, ya que hereda el método abstracto tal y como se había definido en la superclase.

- Una clase abstracta no puede ser instanciada.
- Una clase abstracta puede tener métodos no abstractos. Es más, es posible crear clases abstractas en las que ninguno de sus métodos sea abstracto. Esto último puede resultar interesante cuando se desea evitar que una clase sea instanciada.
- En relación a los atributos, una clase abstracta no incorpora ninguna modificación con respecto a las clases no abstractas.
- Se pueden declarar variables referencia cuyo tipo sea una clase abstracta (por ejemplo, `Figura f;`). Estas referencias podrán ser utilizadas para referenciar objetos de alguna subclase no abstracta, haciendo uso de la *conversión hacia arriba* vista en el apartado 4.2.3. Por ejemplo, se puede crear un vector con `N` referencias del tipo abstracto `Figura` y posteriormente utilizar dicho vector para referenciar objetos de tipo `Rectangulo`, `Circulo`, etc.

Constructores en clases abstractas

Si bien es cierto que no se pueden instanciar objetos de una clase abstracta, ello no quiere decir que carezca de sentido incluir constructores en este tipo de clases. De hecho es habitual que las clases abstractas tengan constructor. El motivo de ello es que cuando se crea un objeto de una subclase, también se ejecutan los constructores de todas sus superclases (aunque éstas sean abstractas) tal y como se ha visto en el apartado 4.1.3.

Por ejemplo, podríamos definir la clase `Figura` y sus subclases como sigue:

```
abstract class Figura {
  int x, y;  // Posición de la figura

  Figura() {
    x=0;
    y=0;
  }

  Figura(int posX, int posY) {
    x = posX;
    y = posY;
  }

  // Resto de métodos
     . . .
}
class Circulo extends Figura {
  int radio;

  Circulo(int r) {  // Como no se especifica lo contrario mediante
    radio = r;     // super, se invocará al constructor sin
  }                 // parámetros en la superclase

  Circulo(int r, int posX, int posY) {
    super(posX, posY); // Se invoca al constructor con parámetros
    radio = r;         // en la superclase
  }

  // Resto de métodos
```

```
        . . .
}
```

La instrucción

```
Circulo c = new Circulo(5);
```

dará lugar a que se invoque el constructor sin parámetros en la superclase, lo que inicializará los atributos con valores `x=0`, `y=0` y `radio=5`, mientras que la instrucción

```
Circulo c = new Circulo(5,10,15);
```

dará lugar a que se invoque el constructor con parámetros en la superclase, lo que inicializará los atributos con valores `x=10`, `y=15` y `radio=5`.

4.2.6. La conversión hacia abajo: `instanceof`

Tal y como se ha visto en el apartado 4.2.3, en la conversión hacia arriba se gana generalidad pero se pierde información acerca del tipo concreto con el que se trabaja y, consecuentemente, se reduce la interfaz. Si se quiere recuperar toda la información del tipo con el que se trabaja (esto es, recuperar el acceso a todos los miembros del objeto) será necesario *moverse hacia abajo* en la jerarquía mediante un cambio de tipo. Para ello se debe cambiar el tipo de la referencia (tipo estático) para que coincida con el tipo real del objeto (tipo dinámico) o al menos con algún otro tipo que permita el acceso al miembro del objeto deseado. Esta conversión a un tipo más específico es lo que se conoce como *conversión hacia abajo* y se realiza mediante una operación de *casting*.

```
Figura f = new Circulo(); // Conversión hacia arriba
Circulo c = (Circulo)f;   // Conversión hacia abajo mediante
                          // cambio de tipo (casting)
// Ahora mediante la referencia c podemos acceder a TODOS los miembros
// de la clase Circulo. Con la referencia f SÓLO podemos acceder a los
// miembros declarados en la clase Figura
```

También se puede hacer un cambio de tipo *temporal* sin necesidad de almacenar el nuevo tipo en otra variable referencia.

```
Figura f = new Circulo(); // Conversión hacia arriba
((Circulo)f).radio=1;     // Acceso a miembros de la subclase
// A continuación f sigue siendo de tipo Figura
```

En este caso mediante la sentencia `((Circulo)f)` se cambia el tipo de `f` únicamente para esta instrucción, lo que permite acceder a cualquier miembro definido en `Circulo` (por ejemplo al atributo `radio`). En sentencias posteriores `f` seguirá siendo de tipo `Figura`.

Las conversiones de tipo, sin embargo, deben realizarse con precaución: la conversión hacia arriba siempre es segura (por ejemplo un círculo, con toda certeza, es una figura, por lo que este cambio de tipo no generará problemas), pero la conversión hacia abajo puede no ser segura (una figura puede ser un círculo, pero también un rectángulo o cualquier otra figura). Dicho de otro modo, en la conversión hacia arriba se reduce la "visibilidad" que se tiene del objeto (se reduce la interfaz) mientras que en la conversión hacia abajo se amplía. Debemos estar seguros de que esta ampliación de la parte visible se corresponde realmente con el objeto que hemos creado, de lo contrario se producirá un error en tiempo de ejecución.

Imaginemos que tenemos una jerarquía de clases como la que se muestra en la figura 4.5. En el siguiente ejemplo se muestran distintas conversiones dentro de esta jerarquía, alguna de ellas incorrecta.

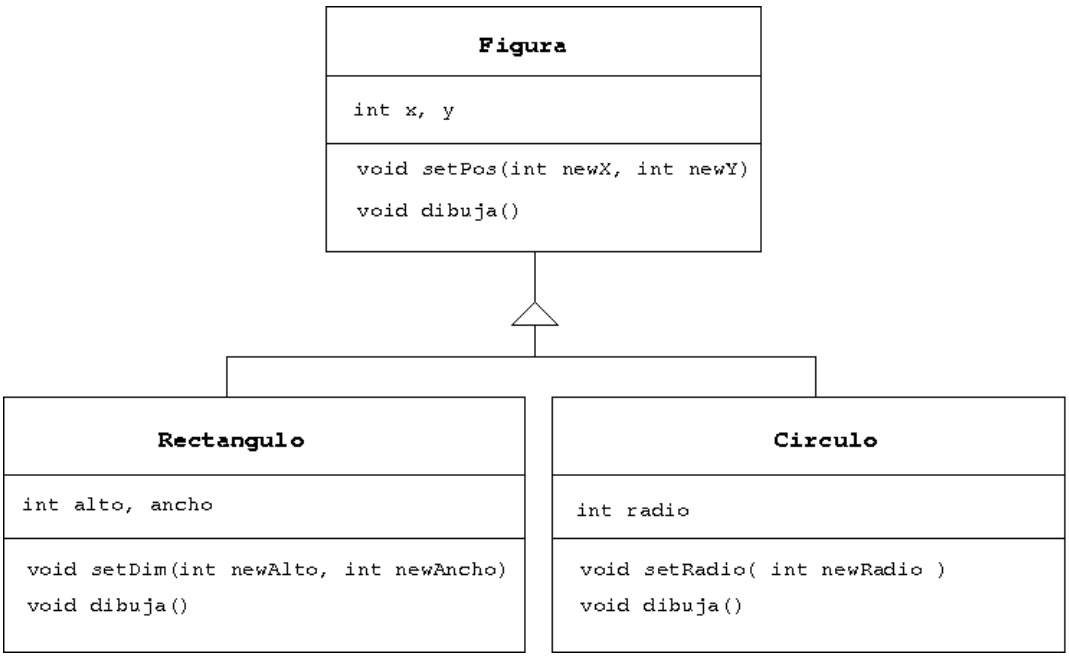

Figura 4.5: Jerarquía entre distintos tipos de figuras.

```
public class ConversionHaciaAbajo {
  public static void main(String[] args) {
    Figura[] v = new Figura[2];
    v[0] = new Rectangulo();       // Conversión hacia arriba
    v[1] = new Circulo();          // Conversión hacia arriba

    v[0].dibuja();                 // Llamada a método polimórfico
    v[1].setRadio(10);             // Provoca un error de compilación.
                                   // El método setRadio no se
                                   // encuentra en la clase Figura
    ((Circulo)v[1]).setRadio(10); // Conversión hacia abajo correcta
    ((Circulo)v[0]).setRadio(10); // Conversión hacia abajo incorrecta.
  }
}
```

En el ejemplo anterior, para acceder a la interfaz completa del objeto `Circulo` (por ejemplo para acceder al método `setRadio`) es necesaria la conversión hacia abajo. Ahora bien, hemos de tener la certeza de que el objeto que se encuentra referenciado es realmente del nuevo tipo al que queremos convertirlo, de lo contrario se producirá un error en tiempo de ejecución, tal y como ocurre en la última instrucción.

En este ejemplo es obvio que `v[0]` referencia un `Rectangulo` y `v[1]` un `Circulo`, por lo que es fácil darse cuenta del error cometido en la última instrucción, sin embargo, esto no siempre es así. Hagamos la siguiente modificación sobre el código anterior:

```
public class ConversionHaciaAbajo {
  public static void main(String[] args) {
    Figura[] v = new Figura[2];
```

```
      double aleatorio = Math.random();
      if( aleatorio > 0.5 ) {
        v[0] = new Rectangulo(); v[1] = new Circulo();
      }
      else {
        v[0] = new Circulo(); v[1] = new Rectangulo();
      }

      . . .

    }
  }
```

Si ahora pretendemos cambiar el radio de cualquier círculo que tengamos en el vector `v` nos encontramos con que no sería seguro hacer una conversión hacia abajo, ya que realmente desconocemos el tipo de objeto que hay almacenado en cada componente del vector. Sería interesante disponer de algún mecanismo que nos permitiera conocer el tipo de un objeto (tipo dinámico) para poder hacer la conversión hacia abajo con total seguridad. En Java, este mecanismo nos lo proporciona el operador **`instanceof`**.

El operador `instanceof` toma como primer operando una variable referencia y como segundo un tipo, según la siguiente sintaxis:

```
referencia instanceof Tipo
```

El resultado de la expresión anterior es `true` en caso de que el objeto asociado a la referencia (tipo dinámico) coincida con el tipo del segundo operador, y `false` en caso contrario.

Por ejemplo, para cambiar el radio de todos los círculos que se encuentran almacenados en un vector de figuras podría emplearse el siguiente código:

```
for( int i=0; i<v.length; i++ )
  if( v[i] instanceof Circulo )    // Comprobación del tipo dinámico
    ((Circulo)v[i]).setRadio(10);  // Conversión hacia abajo segura
```

Como recomendación final cabría decir que el uso del operador `instanceof` debería restringirse a los casos en que sea estrictamente necesario. En la medida de lo posible es preferible emplear métodos polimórficos que permitan realizar las operaciones requeridas sin necesidad de conocer con exactitud el tipo concreto del objeto con el que se trabaja.

4.2.7. Sobreescribiendo la clase `Object`

Es habitual que las clases sobreescriban algunos de los métodos definidos en `Object`.

La sobreescritura del método `finalize` es útil cuando el objeto utiliza ciertos recursos como ficheros y conexiones de red que conviene sean cerrados cuando termina su utilización. También es habitual sobreescribir los métodos `equals` y `toString` mencionados en el apartado 4.1.5, tal y como se muestra en el siguiente ejemplo:

```
class Complejo {
  double real, imag;

  Complejo(double r, double i) {
    real = r;
    imag = i;
  }
```

```
  Complejo() {   // Sobrecargamos el constructor
    this(0,0);  // Se invoca al constructor con dos parámetros
  }
  // Sobrecarga de equals
  public boolean equals( Complejo c ) {
    return (real == c.real && imag == c.imag);
  }
  // Sobreescritura de equals
  public boolean equals( Object o ) {
    if( o instanceof Complejo )
      return (real==((Complejo)o).real && imag==((Complejo)o).imag);
    else
      return false;
  }
  // Sobreescritura de toString
  public String toString() {
    return real + "+" + imag + "i";
  }
}

class P {
  public static void main(String [] args) {
    // Creamos dos números complejos iguales
    Complejo c1 = new Complejo(4, 3);
    Complejo c2 = new Complejo(4, 3);

    if( c1 == c2 )
      System.out.println("Objetos con la misma referencia");
    else
      System.out.println("Objetos con distintas referencias");
    if( c1.equals(c2) )
      System.out.println("Objetos con la misma información");
    else
      System.out.println("Objetos con distinta información");

    System.out.println("c1 = " + c1);
    System.out.println("c2 = " + c2);
  }
}
```

Este ejemplo produciría la siguiente salida:

```
Objetos con distintas referencias
Objetos con la misma información
c1 = 4+3i
c2 = 4+3i
```

Hay que observar que el método `equals` del ejemplo anterior se ha sobrecargado en un caso y sobreescrito en otro. El método con cabecera

```
public boolean equals( Complejo c )
```

corresponde realmente a una sobrecarga y no a una sobreescritura del método con el mismo nombre definido en `Object`. Esto es así porque el método `equals` de `Object` recibe como parámetro una referencia de tipo `Object` y no de tipo `Complejo`.

Por otro lado, el método

```
public boolean equals( Object o )
```

corresponde realmente a una sobreescritura, ya que recibe como parámetro una referencia de tipo `Object`. Tal y como se ha visto en apartados anteriores, es perfectamente válido referenciar un objeto de tipo `Complejo` mediante una referencia de tipo estático `Object`, sin embargo, será necesario cambiar el tipo estático mediante una *conversión hacia abajo* para poder acceder a los atributos propios de la clase `Complejo`.

En cuanto a la sobreescritura de `toString`, debe entenderse que cuando se utiliza cualquier objeto como si fuera un `String`, internamente se invoca el método `toString` de dicho objeto. En este sentido, la instrucción

```
System.out.println("c1 = "+ c1)
```

sería equivalente a

```
System.out.println("c1 = " +c1.toString());
```

donde `c1.toString())` se sustituye, lógicamente, por la cadena que devuelva el método `toString`.

4.3. Interfaces

Se entiende por interfaz de una clase el conjunto de operaciones (métodos) que se pueden realizar con los objetos de dicha clase. Aunque dicha interfaz puede deducirse fácilmente observando los métodos públicos implementados en una clase determinada, el lenguaje JAVA dispone de un mecanismo para definir de forma explícita las interfaces de las clases. Este mecanismo, más allá de detallar simplemente la interfaz de las clases, aporta mayor nivel de abstracción al lenguaje, tal y como se irá viendo en los siguientes apartados.

4.3.1. El problema de la herencia múltiple

En JAVA no existe la herencia múltiple. De esta manera, se evitan los problemas que genera la utilización de la herencia múltiple que fueron comentados en el apartado 4.1.1. Sin embargo, tampoco se tienen las ventajas de la herencia múltiple. La solución en JAVA ha sido la incorporación de un mecanismo, las *interfaces*, que proveen de algunas de las ventajas de la herencia múltiple sin sus inconvenientes.

En la figura 4.6 se presenta un diseño (no permitido en JAVA) en el que la clase `C` hereda tanto de `A` como de `B`. Si, como ocurre en el ejemplo, las dos superclases `A` y `B` contienen el método `m1()` y dicho método está implementado de forma distinta en cada una de ellas, existiría un conflicto respecto a cuál de las dos implementaciones de `m1()` debería heredar `C`. Este conflicto, por otro lado, no se daría si `m1()` se hubiese definido abstracto tanto en `A` como en `B`, ya que en este caso el método no estaría implementado y por tanto no cabría esperar conflicto alguno. Podría pensarse, por tanto, que si `A` y `B` fuesen clases abstractas no debería haber inconveniente en permitir la herencia múltiple. Sin embargo, tal y como se ha visto en apartados anteriores, las clases abstractas pueden contener métodos no abstractos, con lo que la exigencia propuesta de que las superclases sean abstractas no garantizaría la ausencia de conflictos en un esquema de herencia múltiple.

Para que un diseño como el propuesto fuera fácilmente implementable debería existir algún mecanismo que exigiera tanto a `A` como a `B` que todos sus métodos fuesen abstractos. Además, por motivos similares, dicho mecanismo no debería permitir la declaración de atributos de instancia en `A` o en `B`, ya que podría darse el caso de que dos atributos con el mismo nombre tuviesen valores distintos. Pues bien, este mecanismo existe en JAVA y se lleva a la práctica mediante lo que se conoce como *interfaz*.

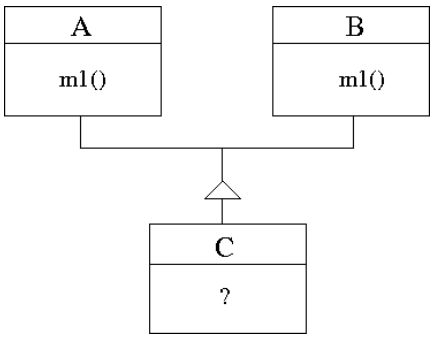

Figura 4.6: Conflictos en un esquema de herencia múltiple.

El uso de interfaces, más allá de tratar de resolver únicamente el problema de la herencia múltiple, guarda una estrecha relación con el concepto de polimorfismo estudiado en el apartado 4.2, tal y como se verá más adelante.

4.3.2. Declaración e implementación de interfaces

La interfaz de una clase viene definida de forma implícita en función de los miembros públicos[2] que contenga dicha clase. Además de esta definición implícita, JAVA dispone de un mecanismo para definir explícitamente la interfaz (o parte de la interfaz) de una clase, lo que permite separarla completamente de su implementación. Ello se hace mediante la declaración de lo que en JAVA se denomina una interfaz. Sintácticamente, las interfaces son como las clases pero sin variables de instancia y con métodos declarados sin cuerpo.

La definición de una interfaz tiene la forma siguiente:

```
acceso interfaz nombre_interfaz {
  tipo var1;
  tipo var2;
  ...
  tipo metodo1( ... ) ;
  tipo metodo2( ... ) ;
}
```

El acceso de una interfaz debe ser público, bien con la palabra reservada `public` o sin modificador de acceso (*friendly*). Después se especifica que la siguiente definición pertenece a una interfaz utilizando la palabra reservada `interface`, seguida del nombre de la interfaz y del bloque de código correspondiente a la definición de la interfaz. Las variables declaradas en la interfaz son implícitamente estáticas (`static`) y finales (`final`), lo que significa que han de ser inicializadas con un valor constante y su valor no puede cambiarse posteriormente. Los métodos no tienen cuerpo puesto que son básicamente métodos abstractos que han de ser implementados por las clases que implementan la interfaz.

Como se irá viendo a lo largo de esta sección, las interfeces tienen ciertas similitudes con las clases abstractas. De hecho, una interfaz podría verse como una clase abstracta sin atributos de instancia y en la que todos sus métodos son abstractos.

[2]En este contexto el concepto *público* no se restringe únicamente a aquellos miembros declarados con la palabra reservada public sino a aquellos miembros que sean visibles desde otros objetos. El que sean o no visibles dependerá del modificador de acceso (public, protected, private o *friendly*), de la relación de herencia entre las distintas clases y del paquete donde se encuentren.

Como ejemplo se podría definir la interfaz de cierta colección de objetos del siguiente modo:

```
interface Coleccion {
  void añadirElemento( Object o );
  int getNumElementos();
  void mostrar();
}
```

Se puede observar que la interfaz no hace suposiciones acerca de los detalles de implementación. Básicamente una interfaz define **qué** operaciones se pueden realizar, pero no **cómo** se realizan.

Una vez definida una interfaz, cualquier número de clases puede implementarla. Implementar una interfaz implica implementar cada uno de los métodos definidos en la misma. Para indicar que una clase implementa una interfaz determinada, se utiliza la palabra reservada ***implements*** con la siguiente sintaxis:

```
acceso class nombreClase implements NombreInterfaz1[, NombreInterfaz2] {
  ...
}
```

Básicamente, todo lo que necesita una clase para implementar una interfaz es sobreescribir el conjunto completo de métodos declarados en dicha interfaz. En caso de que la clase no sobreescriba todos los métodos, dicha clase deberá declararse abstracta, ya que contiene métodos sin implementar. Los métodos definidos en una interfaz deben ser declarados con `public` en la clase que los implementa.

Por ejemplo, podríamos realizar la siguiente implementación de la interfaz `Coleccion` utilizando un vector para almacenar los distintos elementos del conjunto:

```
class Conjunto implements Coleccion {
  private Object[] v;
  private int numElementos;

  Conjunto( int maxElementos ) {
    v = new Object[maxElementos];
    numElementos = 0;
  }

  // Implementar los métodos de la interfaz Coleccion
  public void añadirElemento( Object o ) {
    if( numElementos < v.length ) {
      v[numElementos] = o;
      numElementos++;
    }
  }
  public int getNumElementos() {
    return numElementos;
  }
  public void mostrar() {
    for(int i=0; i<numElementos; i++)
      System.out.println(o);  // Tendremos que haber sobreescrito el
                              // método toString de nuestros objetos
  }
}
```

Podría realizarse, por otro lado, una implementación alternativa basada en una lista enlazada:

```
class ListaEnlazada implements Coleccion {
  private Object cabeza;
  private int numElementos;

  Conjunto() {
    cabeza = null;
    numElementos = 0;
  }

  // Implementar los métodos de la interfaz Coleccion
  public void añadirElemento( Object o ) {
    // Operaciones para añadir el objeto o a la lista enlazada
              . . .
  }
  public int getNumElementos() {
    return numElementos;
  }
  public void mostrar() {
    // Recorrer la lista enlazada y mostrar cada uno de los elementos
              . . .
  }
}
```

Con este esquema, independientemente de que utilicemos objetos de tipo `Conjunto` o de tipo `ListaEnlazada`, sabremos que en cualquier caso tendrán implementados los métodos `añadirElemento`, `getNumElementos` y `mostrar` ya que todos ellos fueron definidos en la interfaz `Coleccion`.

Podría dar la sensación de que definir la interfaz en un lugar separado e independiente de la clase que lo implementa no aporte ninguna ventaja, ya que se puede "averiguar" fácilmente la interfaz de cualquier clase con tan solo fijarnos en sus métodos públicos, sin necesidad de definirla explícitamente en un lugar independiente de la clase. Por otro lado, este esquema tiene muchas similitudes con el concepto de clase abstracta visto en el apartado 4.2.5, con lo que nuevamente la utilidad de las interfaces parece diluirse. De hecho, implementar una interfaz tiene implicaciones muy similares a las que tendría extender una clase abstracta. Existe, sin embargo, una diferencia fundamental: las clases abstractas se reutilizan e implementan mediante el esquema de herencia. Ello quiere decir que sólo se puede heredar de una clase abstracta. Sin embargo, una clase puede implementar múltiples interfaces, e incluso heredar a la vez de alguna otra clase.

Las interfaces están en una jerarquía distinta de la jerarquía de clases, por lo que es posible que varias clases que no tengan la más mínima relación de herencia implementen la misma interfaz. En la figura 4.7 se muestran dos interfaces (`I1` e `I2`) que son implementadas por clases que no en todos los casos tienen una relación de herencia. Concretamente, la clase `A` (y por extensión `C` y `D`) implementan la interfaz `I1`. Por otro lado, la clase `E`, que no guarda relación de herencia con las anteriores, también implementa `I1`. Adicionalmente, `E` implementa `I2` (una clase puede implementar más de una interfaz).

En el próximo apartado se explotará el mecanismo de las interfaces en el contexto del polimorfismo y se verá la potencia que realmente aportan. Antes de ello concluiremos este apartado con algunas considereciones que deben tenerse en cuenta acerca de las interfaces:

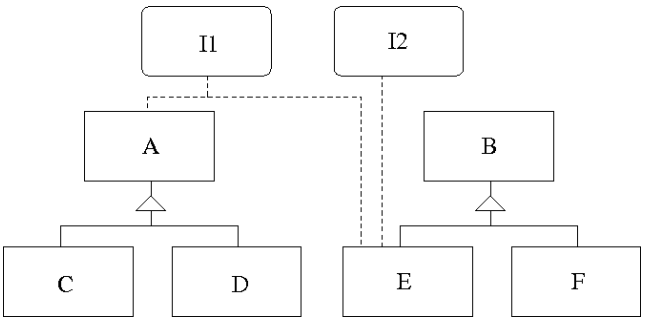

Figura 4.7: Combinación de un esquema de herencia e implementación de interfaces.

- Una interfaz únicamente puede contener atributos estáticos constantes y cabeceras de métodos.
- No es posible instanciar una interfaz.
- Una misma interfaz puede ser implementada por más de una clase.
- Una misma clase puede implementar más de una interfaz.
- Cualquier método definido en una interfaz se sobreentiende que es público, aunque no se ponga explícitamente la palabra `public` delante del mismo.
- Cualquier atributo definido en una interfaz se sobreentiende que es público, estático y final, aunque no se pongan explícitamente las palabras `public static final` delante del mismo.
- Cualquier clase que implemente una interfaz deberá implementar todos los métodos definidos en la misma. Si no lo hace se convierte en una clase abstracta (por no tener todos sus métodos implementados) y deberá declararse como tal mediante la palabra reservada `abstract`.

4.3.3. Implementación de polimorfismo mediante interfaces

Una interfaz es un tipo de dato y, por tanto, puede utilizarse para declarar referencias o como tipo de dato en los argumentos de un método.

En el apartado 4.2.3 se vio cómo una referencia de una superclase puede referenciar a objetos pertenecientes a alguna de sus subclases (conversión hacia arriba). De forma similar, una referencia de una interfaz puede emplearse para referenciar objetos pertenecientes a cualquier clase que implemente dicha interfaz. Ello permite implementar el polimorfismo en tiempo de ejecución de un modo similar a como se implementa haciendo uso de clases abstractas.

Por ejemplo, dada la interfaz `Coleccion` del ejemplo anterior y sus dos implementaciones `Conjunto` y `ListaEnlazada`, es posible declarar referencias de tipo `Coleccion` y sobre ellas instanciar objetos de tipo `Conjunto` o `ListaEnlazada`.

```
Coleccion c;
if( cierta_condicion )
  c = new Conjunto();
else
```

```
    c = new ListaEnlazada();

  // Añadir elementos
  for( int i=0; i<10; i++ )
    c.añadirElemento( new Integer(i) );

  // Mostrar el contenido
  System.out.print("La colección contiene los siguientes ");
  System.out.println( c.getNumElementos() + "elementos");
  c.mostrar();
```

En este ejemplo, según ciertas condiciones, se crea un objeto de tipo `Conjunto` o de tipo `ListaEnlazada`. Lo interesante del código anterior es que nuestro programa puede trabajar perfectamente con la referencia `c` sin necesidad de conocer qué tipo de objeto se ha creado realmente bajo dicha referencia. Será en tiempo de ejecución cuando, al invocar cualquiera de los métodos de `c`, se decida qué método debe ejecutarse (en nuestro caso un método de la clase `Conjunto` o de la clase `ListaEnlazada`). Ya se vio en el apartado 4.2.4 que este mecanismo que permite posponer hasta el último momento la decisión del método que debe ejecutarse se conoce como *selección dinámica de método* o *enlace dinámico*[3] y, a la postre, permite implementar el polimorfismo en tiempo de ejecución.

A continuación se muestra un ejemplo completo en el que se ha definido la interfaz `ConjuntoOrdenable` y una clase `Ordena` con una serie de métodos que permiten ordenar cualquier conjunto ordenable. Lógicamente, para que un conjunto sea considerado *ordenable* debe implementar la interfaz `ConjuntoOrdenable`. Nuevamente, lo interesante del ejemplo que se muestra a continuación radica en que los métodos de la clase `Ordena` no necesitan conocer el tipo real de los objetos que se están ordenando.

```
  interface ConjuntoOrdenable {
    public int getNumElementos();
    public boolean menor(int i, int j);
    public void intercambiar(int i, int j);
  }

  class Ordena {
    // Este método puede ordenar cualquier conjunto
    // que implemente la interfaz ConjuntoOrdenable
    static void seleccionDirecta( ConjuntoOrdenable c ) {
      int pos_min, N = c.getNumElementos();
      for( int i = 0; i <= N-2; i++ ) {
        pos_min = i;
        for( int j = i+1; j < N; j++ ) {
          if( c.menor(j, pos_min) )
            pos_min = j;
        }
        c.intercambiar(i, pos_min);
      }
    }
    static void burbuja( ConjuntoOrdenable c ) {
       // Otros métodos de ordenación
       // . . .
    }
    static void quickSort( ConjuntoOrdenable c ) {
```

[3] Algunos autores utilizan el término *ligadura tardía* para referirse a este mismo concepto.

```
      // Otros métodos de ordenación
      // . . .
  }
}

// Una implementación concreta de un ConjuntoOrdenable
class ConjuntoPalabras implements ConjuntoOrdenable {
  private LinkedList lista;

  // Métodos propios
  ConjuntoPalabras() {
     lista = new LinkedList();
  }
  public void inserta(String s) {
    lista.add(s);
  }
  public String toString() {
    String s = "";
    for( int i=0; i<lista.size(); i++ )
      s = s + lista.get(i) + " ";
    return s;
  }
  // Métodos exigidos por la interfaz ConjuntoOrdenable
  public int getNumElementos() {
    return lista.size();
  }
  public boolean menor( int i, int j ) {
    String s1 = (String)lista.get(i);
    String s2 = (String)lista.get(j);
    if( s1.compareTo(s2) < 0 )
      return true;
    else
      return false;
  }
  public void intercambiar(int i, int j) {
    Object pal1 = lista.get(i);
    Object pal2 = lista.get(j);
    lista.set(j, pal1);
    lista.set(i, pal2);
  }
}

class EjemploInterfazOrdena {
  public static void main( String[] args ) {
    ConjuntoPalabras cp = new ConjuntoPalabras();
    cp.inserta("las");
    cp.inserta("interfaces");
    cp.inserta("son");
    cp.inserta("muy");
    cp.inserta("utiles");

    System.out.println(cp);
    Ordena.seleccionDirecta(cp);
    System.out.println(cp);
```

```
    }
  }
```

Sobre el ejemplo anterior conviene resaltar que:

- `seleccionDirecta` es un método estático. Por eso se puede invocar sin necesidad de crear objetos de tipo `Ordena` (se invoca anteponiendo al método el nombre de la clase).
- El método `seleccionDirecta` admite cualquier objeto de tipo `ConjuntoOrdenable`.
- Los objetos de tipo `ConjuntoPalabras` también son de tipo `ConjuntoOrdenable`, ya que implementa dicha interfaz.
- El método `seleccionDirecta` espera recibir como parámetro algún objeto de tipo `ConjuntoOrdenable` sin embargo, cuando se invoca se le está pasando un objeto de tipo `ConjuntoPalabras`. Esto es posible gracias a la conversión hacia arriba y a que los objetos de tipo `ConjuntoPalabras`, de algún modo, también son de tipo `ConjuntoOrdenable`.

Podríamos ampliar el ejemplo anterior para ilustrar cómo una clase puede implementar más de una interfaz al mismo tiempo.

```
interface ConjuntoDesordenable {
  public int getNumElementos();
  public void intercambiar(int i, int j);
}
class Desordena {
  // Este método puede desordenar cualquier conjunto que
  // implemente la interfaz ConjuntoDesordenable
  static void desordenar( ConjuntoDesordenable c ) {
    Random rnd = new Random();
    int n = c.getNumElementos();
    for(int i=0; i<n*2; i++) {
      // Selecciono dos elementos aleatorios y los
      // cambio de lugar
      int elem1 = Math.abs(rnd.nextInt()) % n;
      int elem2 = Math.abs(rnd.nextInt()) % n;
      c.intercambiar(elem1, elem2);
    }
  }
}
class ConjuntoPalabras implements ConjuntoOrdenable,
                                  ConjuntoDesordenable {
   . . .
}
```

Ahora, los objetos de tipo `ConjuntoPalabras` no sólo pueden ordenarse sino también desordenarse. Esto es así porque `ConjuntoPalabras` implementa tanto la interfaz `ConjuntoOrdenable` como `ConjuntoDesordenable` y, por tanto, cualquier método que requiera como parámetro objetos de alguno de estos tipos, podrá recibir sin problemas objetos de tipo `ConjuntoPalabras`.

Por otro lado, el cuerpo de la clase `ConjuntoPalabras` será igual al del ejemplo anterior ya que, aunque ahora implementa también la interfaz `ConjuntoDesordenable`, todos los métodos requeridos por esta interfaz ya existían en la versión anterior y, por tanto, ya están implementados. De no ser así, hubiese sido necesario implementar los nuevos métodos requeridos por la interfaz `ConjuntoDesordenable`.

4.3.4. Definición de constantes

Dado que, por definición, todos los atributos que se definen en una interfaz son `static` y `final`, las interfaces pueden utilizarse también para definir grupos de constantes.

Por ejemplo

```
interface DiasSemana {
  int LUNES=1, MARTES=2, MIERCOLES=3, JUEVES=4, VIERNES=5,
      SABADO=6, DOMINGO=7;
}
```

Aunque no se haya indicado explícitamente, se considera que LUNES, MARTES, etc. son estáticas y finales. Por otro lado, si bien no es un requerimiento del lenguaje, por convenio las constantes se suelen escribir en mayúsculas.

Un posible uso de la interfaz anterior sería:

```
for( int i = DiasSemana.LUNES; i <= DiasSemana.DOMINGO; i++ ) {
   . . .
}
```

4.3.5. Herencia en las interfaces

Las interfaces pueden extender otras interfaces. Además, a diferencia de lo que ocurre con las clases, entre interfaces se permite la herencia múltiple. Esto es así porque, aun en el caso en el que se heredasen dos métodos con el mismo nombre y atributos, no podrían entrar en conflicto por carecer dichos métodos de cuerpo.

```
interface I1 {
  void metodo1();
  void metodo2();
}
interface I2 {
  void metodo3();
}
interface I3 extends I1, I2 {  // Herencia múltiple entre interfaces
  void metodo4();
  // metodo1, metodo2 y metodo3 se heredan de I1 e I2
}
class C implements I3 {
  // La clase C deberá implementar
  //  metodo1,  metodo2,  metodo3 y metodo4
}
```

4.4. Ejercicios resueltos

4.4.1. Enunciados

1. Indica lo que mostraría por pantalla el siguiente programa o, en caso de que sea incorrecto, explica cuál es el error:

```
class A {
}
class B extends A {
  void metodo() {
```

```
      System.out.println("Método de B");
    }
  }
  public class Clase {
    public static void main( String[] args ) {
      A obj = new B();
      obj.metodo();
    }
  }
```

2. Indica lo que mostraría por pantalla el siguiente programa o, en caso de que sea incorrecto, explica cuál es el error:

```
class A {
  void metodo() {
    System.out.println("Método de A");
  }
}
class B extends A {
  void metodo() {
    System.out.println("Método de B");
  }
}
public class Clase {
  public static void main( String[] args ) {
    A obj = new B();
    obj.metodo();
  }
}
```

3. En la siguiente jerarquía de herencia existen llamadas a métodos en los que se debe tener claro el concepto de polimorfismo. Escribe cuál sería el resultado de la ejecución del siguiente código:

```
abstract class Flujo {
   abstract public void escribe(char c);
   public void escribe (String s) {
     for (int i=0; i<s.length(); i++) {
       System.out.println("Escribe de Flujo ....");
       escribe(s.charAt(i));
     }
   }
   public void escribe (int i) { escribe(""+i); }
}

class Disco extends Flujo {
  public void escribe(char c) {
    System.out.println("Escribe de disco " + c);
  }
}

class DiscoFlexible extends Disco {
  public void escribe (String s) {
    System.out.println ("Escribe de Disco Flexible...");
```

```
      super.escribe(s);
    }
  }

  public class TestFlujo {
    public static void main (String a[]) {
      DiscoFlexible dc = new DiscoFlexible();
      Flujo f = dc;
      f.escribe("ab");
    }
  }
```

4. Dadas las siguientes clases:

```
class A {
  int at1=-1, at2=-1;
  A(int i, int j) { at1=i; at2=j; }
  A(int i) { this(i,0); }
  A() { this(0); }
  public String toString() { return "at1=" + at1 + " at2="+at2; }
}
class B extends A {
  int at3=-1;
  B(int i, int j, int k) { super(i,j); at3=k; }
  B(int i) { at3=i; }
  public String toString() { return super.toString() + " at3=" + at3; }
}
class C extends B {
  int at4=-1;
  C(int i, int j, int r, int s) { super(i,j,r); at4=s; }
  C(int i, int j, int k) { at4=0; }
  public String toString() { return super.toString() + " at4=" + at4; }
}
```

Indicar la salida por pantalla que generaría cada una de las siguientes sentencias o, en caso de que alguna de ellas no sea correcta, explicar el error.

```
1.- A obj = new B(1,2,3); System.out.println(obj);
2.- B obj = new B(1); System.out.println(obj);
3.- C obj = new C(1,2,3,4); System.out.println(obj);
4.- C obj = new C(1,2,3); System.out.println(obj);
```

5. Razona si el siguiente código es o no correcto.

```
abstract class A {
  int i;
  A(int i) {
    this.i = i;
  }
}
class B extends A {
  void metodo() { System.out.println(i); }
}

class Cuestion {
```

```
  public static void main(String[] args){
    A[] v = new A[10];
    for ( int i = 0; i < 10; i++ ) {
      v[i] = new B();
      v[i].metodo();
    }
  }
}
```

6. Reescribe el siguiente código para que realice la misma operación sin utilizar la instrucción instanceof y sin utilizar *casting*. En la solución aportada se debe usar la técnica de enlace dinámico.

```
interface Procesable {}
class A implements Procesable{
  void procesarA() { System.out.println("Procesando A") };
}
class B implements Procesable {
  void procesarB() { System.out.println("Procesando B") };
}
class Cuestion {
  public static void main(String[] args){
    Procesable p;
    if( cierta_condicion )
      p = new A();
    else
      p = new B();
    if( p instanceof A )
      ((A)p).procesarA();
    else
      ((B)p).procesarB();
  }
}
```

7. Dado el siguiente código, indica qué es incorrecto y por qué.

```
interface I {
  int i = 10;
  void mostrar();
}
class A implements I {
  A(int i) {
    this.i = i;
  }
  void mostrar() {
    System.out.println(i);
  }
}
```

8. Indica si el siguiente código es correcto. Razona la respuesta.

```
abstract class A {
  abstract void m1();
  void m2() { System.out.println("Soy el método 2 de A");
}
```

```
class B extends A {
  void m2() { System.out.println("Soy el método 2 de B");
}
```

4.4.2. Soluciones

1. La referencia `obj` tiene tipo estático `A` y tipo dinámico `B`. El tipo estático indicá QUÉ se puede hacer y el dinámico CÓMO se hace. A pesar de que `metodo()` está definido en el tipo dinámico, como no se encuentra en el tipo estático se generará un error de compilación.

2. El programa es correcto. Mostraría por pantalla

   ```
   Método de B
   ```

3. ```
   Escribe de Disco Flexible...
   Escribe de Flujo ....
   Escribe de disco a
   Escribe de Flujo ....
   Escribe de disco b
   ```

4. ```
   1.- at1=1 at2=2 at3=3
   2.- at1=0 at2=0 at3=1
   3.- at1=1 at2=2 at3=3 at4=4
   4.- Error. En la clase B no existe un constructor vacío, y en el constructor
       de C no se especifica explícitamente qué constructor debe ejecutarse en B
   ```

5. - La instrucción `v[i].metodo()` es incorrecta, ya que aunque dicho método existe en el tipo dinámico de v[i] (clase B), no existe en el tipo estático (clase A). Podría solucionarse añadiendo `abstract void metodo();` en la clase A.
   - Al haber escrito el constructor `A(int i)` en la clase A, se pierde el constructor sin parámetros. Puesto que al crear objetos de la clase B se debe ejecutar no sólo el constructor de B sino también el de A, sería necesario especificar en el constructor de B, mediante el uso de `super`, que en A deseamos ejecutar el constructor `A(int i)`. Una segunda alternativa consistiría en añadir un constructor sin parámetros en A.

6. ```
   interface Procesable {
     void procesar();
   }
   class A implements Procesable{
     void procesar() { System.out.println("Procesando A") };
   }
   class B implements Procesable {
     void procesar() { System.out.println("Procesando B") };
   }
   class Cuestion {
     public static void main(String[] args){
       Procesable p;
       if( cierta_condicion )
         p = new A();
       else
         p = new B();
       p.procesar();
   ```
   ```

```
    }
  }
```

7. - La instrucción `this.i = i` es incorrecta, ya que `this.i` es una variable de tipo `final static` (por haberse definido dentro de una interfaz) y por tanto no es posible modificar su valor.
   - El método `mostrar()` debería ser `public` por estar definido en una interfaz.

8. El código es incorrecto ya que el método `m1()` definido en la clase A debería haberse sobreescrito en la clase B, o bien debería haberse declarado la clase B como abstracta.

4.5. Ejercicios propuestos

1. Indicar cuál sería el resultado de ejecutar el siguiente programa:

```
class C1 {
  int i;
  C1() { i = 1;}
  void m1() { System.out.println( i ); }
  void m2() { m1(); m1(); }
}

class C2 extends C1 {
  C2() { i = 2; }
  void m1() { System.out.println( -i ); }
}

class C3 extends C2 {
  C3() { i++; }
  void m2() { m1(); }
}

class Herencia_y_Sobreescritura {
  public static void main( String[] args ) {
    C1 a = new C1();
    a.m2();
    C1 b = new C2();
    b.m2();
    C2 c = new C3();
    c.m2();
  }
}
```

2. Se pretende implementar el conjunto de clases necesarias para simular conexiones sencillas entre redes de ordenadores. Nuestro sistema de conexiones estará formado por redes, encaminadores y hosts. Para realizar la conexión de los distintos elementos que forman el sistema debe tenerse en cuenta que:

   - Todos los elementos (redes, encaminadores y hosts) tienes una dirección asociada formada por dos números enteros. Las redes y los encaminadores tienen una dirección del tipo X.0, mientras que los hosts tienen una dirección de tipo X.Y.

- Cada host se conecta a una red determinada. Las direcciones de todos los host de una misma red tendrán el primer campo en común, que coincidirá a su vez con el primer campo de la dirección de la red a la que están conectados (por ejemplo, los hosts con direcciones 128.1, 128.2, etc. deberán pertenecer a la red 128.0).
- Las redes deben de mantener una lista de los hosts que están presentes en ella. Para ello se aconseja que la clase red tenga un atributo privado de tipo `LinkedList`. Además, las redes pueden estar conectadas a un encaminador.
- Cada encaminador tiene N puertos, a los que pueden conectarse tanto redes como otros encaminadores. Cuando se crea un encaminador se debe indicar el número de puertos que tiene.

La implementación a realizar debe de ceñirse al uso que de ella se hace en el siguiente programa principal:

```
class Redes {
  public static void main(String[] args){
    Red r1,r2,r3;
    Encaminador e1,e2;
    Host h11,h12,h21,h22,h31,h32;

    // Creamos las redes
    r1=new Red(new Direccion(1,0));
    r2=new Red(new Direccion(2,0));
    r3=new Red(new Direccion(3,0));

    // Creamos los hosts
    h11=new Host(new Direccion(1,1));
    h12=new Host(new Direccion(1,2));
    h21=new Host(new Direccion(2,1));
    h22=new Host(new Direccion(2,2));
    h31=new Host(new Direccion(3,1));
    h32=new Host(new Direccion(3,2));

    // Conectamos los hosts a las redes
    r1.conectar(h11);  h11.conectar(r1);
    r1.conectar(h12);  h12.conectar(r1);
    r2.conectar(h21);  h21.conectar(r2);
    r2.conectar(h22);  h22.conectar(r2);
    r3.conectar(h31);  h31.conectar(r3);
    r3.conectar(h32);  h32.conectar(r3);

    // Creamos los encaminadores
    e1=new Encaminador(new Direccion(4,0),3);
    e2=new Encaminador(new Direccion(5,0),3);

    // Conectamos los encaminadores y las redes
    e1.conectar(r1);  r1.conectar(e1);
    e1.conectar(r2);  r2.conectar(e1);
    e1.conectar(e2);  e2.conectar(e1);
    e2.conectar(r3);  r3.conectar(e2);
  }
}
```

3. Ampliar el programa del ejercicio anterior para que permita el envío de mensajes (en nuestro caso objetos de tipo `String`) entre dos hosts. Para mandar un mensaje del host X1.Y1 al host X2.Y2 se sigue el siguiente procedimiento:

   - El host X1.Y1 envía el mensaje a su red (X1.0).
   - La red X1.0 comprueba la dirección destino y
     - Si la dirección destino es de la misma red, envía el mensaje directamente al host destino.
     - Si no, envía el mensaje al encaminador, para que este lo redirija.
   - Si un encaminador recibe un mensaje, comprueba si alguna de las redes que tiene conectadas a él es la que tiene el host destino (red con dirección X2.0).
     - Si encuentra la red destino, envía a ésta el mensaje.
     - Si no, reenvía el mensaje a todos los encaminadores que tenga conectados.

   El envío de mensajes debería funcionar correctamente con el siguiente código (suponiendo que se ha creado una red como la del ejercicio anterior).

```
h11.envia("MENSAJE_1",new Direccion(3,2));
h32.envia("MENSAJE_2",new Direccion(3,1));
h21.envia("MENSAJE_3",new Direccion(1,2));
```

Capítulo 5

Manejo de excepciones

5.1. Introducción a las excepciones

Una *excepción* es una situación anormal ocurrida durante la ejecución de un programa. En un programa Java, pueden producirse excepciones por distintos motivos. Por ejemplo, en el caso de que un programa ejecute una división entre cero, se producirá una excepción indicando que no es posible ejecutar tal operación. En un programa que intenta escribir ciertos datos en un fichero, se produce una excepción en caso de que ocurra algún error en el sistema de ficheros que contiene el fichero. También se produce una excepción cuando un programa intenta acceder a un elemento de un array utilizando un índice incorrecto (por ejemplo, un valor negativo). Existen muchas otras situaciones en las que se puede producir una excepción. En algunos casos, las excepciones indican errores en el programa. En otros casos, las excepciones indican algún problema grave ocurrido en la máquina virtual, el sistema operativo subyacente o los dispositivos hardware utilizados.

La plataforma de desarrollo Java proporciona algunos mecanismos que permiten a un programa Java detectar las excepciones y recuperarse de los posibles problemas ocurridos. Los programas Java también pueden generar excepciones, por ejemplo para señalar alguna situación anómala detectada por el propio programa.

Entre estos mecanismos se incluyen distintas clases que permiten representar distintos tipos de excepciones que pueden producirse durante la ejecución de un programa Java. Todas esas clases se organizan de forma jerárquica a partir de una clase base llamada `Throwable`.

Esta clase tiene dos subclases, `Exception` y `Error`. La clase `Exception` permite representar aquellas excepciones que un programa Java convencional podría querer detectar y tratar. La clase `Error` permite representar distintos errores graves, que no se suelen tratar en los programas Java convencionales. En general, estas excepciones se deben a problemas importantes que impiden que el programa se siga ejecutando con normalidad.

A su vez, la clase `Exception` tiene muchas subclases. Algunas de ellas pueden indicar problemas concretos. Por ejemplo, la excepción `ClassNotFoundException` se produce cuando la máquina virtual no encuentra un fichero `.class` con la definición de una clase necesaria para la ejecución del programa actual.

También hay subclases de `Exception` que permiten agrupar distintos conjuntos de excepciones. Por ejemplo, la clase `IOException` permite representar distintos tipos de excepciones que pueden producirse al realizar operaciones de entrada/salida de datos. La clase `FileNotFoundException` se produce cuando un programa Java intenta abrir un fichero que no existe.

La clase `Error` también tiene diversas subclases, aunque en general, los programas Java no suelen estar preparados para detectar y manejar dichos errores.

En la figura 5.1 se representa una pequeña parte de la jerarquía de clases de la librería estándar de Java que representan excepciones.

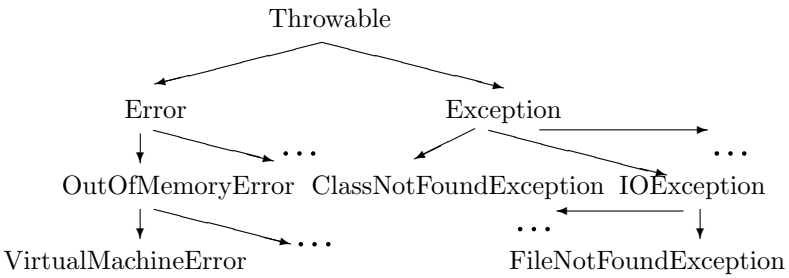

Figura 5.1: Jerarquía de excepciones en Java (fragmento)

En cualquier caso, cuando se produce una situación anómala durante la ejecución de un programa Java, sea cual sea el tipo de situación, la máquina virtual crea una instancia de `Throwable` o alguna de sus subclases. Este objeto contiene información acerca de la situación anómala y puede ser *capturado* por el programa para consultar dicha información y para realizar las acciones que se consideren adecuadas.

5.2. Captura de excepciones

Considérese el siguiente ejemplo, donde se intenta abrir el fichero `data.bin` para leer la información que contiene:

```
public class Exc0 {
  public static void main (String [] args) {
    int d1, d2, d3;
    FileInputStream f = new FileInputStream("data.bin");

    ... // Lectura de los datos

    ... // Uso de los datos
  }
}
```

La instrucción utilizada para abrir el fichero `data.bin` puede producir o *lanzar* una excepción, por ejemplo, en caso de que el fichero en cuestión no exista. Además, las operaciones utilizadas para leer los datos contenidos en el fichero (no indicadas en el ejemplo anterior) posiblemente también lancen excepciones, si se produce algún error leyendo el fichero. En caso de que esto ocurra, la ejecución del programa se interrumpe y en la salida de error del programa se escribe un resultado como el siguiente:

```
java.lang.FileNotFoundException: data.bin
at Exc0.main(Exc0.java:3).
```

En este caso, se dice que el programa *ha lanzado* una excepción. La salida anterior se conoce como *la traza* de la excepción y contiene información sobre el error producido (consultar el apartado 5.3.2).

El programa anterior presenta un importante problema. En caso de que se produzca una excepción, el programa es interrumpido y terminado, sin tener posibilidad alguna de tratar el error y recuperarse.

El lenguaje Java proporciona un mecanismo que permite a los programas Java detectar la ocurrencia de excepciones y realizar algunas operaciones que le permitan recuperarse de los errores ocurridos. Este mecanismo es la sentencia `try-catch`.

5.2.1. La sentencia `try-catch`

La sentencia `try-catch` permite *proteger* un cierto fragmento de código frente a la ocurrencia de determinados tipos de excepciones y en caso de que éstas ocurran, permite ejecutar un cierto código para recuperarse de los posibles errores.

De forma general, se utiliza de la siguiente forma:

```
try {
  ... // Código que puede generar excepciones
} catch (Tipos de excepciones a capturar) {
  ... // Código para tratar el error
}
```

Concretamente, el ejemplo anterior se podría modificar de la siguiente manera:

```
public class Exc0 {
  public static void main (String [] args) {
    int d1, d2, d3;
    try {
      FileInputStream f = new FileInputStream("data.bin");

      // Lectura de los datos
      d1 = ...
      d2 = ...
      d3 = ...

    } catch (Exception e) {
      System.out.println("No es posible abrir el fichero");
      return;
    }

    ... // Uso de los datos
  }
}
```

Las operaciones que pueden lanzar alguna excepción están *protegidas* mediante el bloque `try`. Si las operaciones protegidas se ejecutan correctamente y no se produce ninguna excepción, entonces la ejecución continúa a partir de la siguiente instrucción al bloque `try-catch`. En el ejemplo, una vez leídos los datos, se procedería a usarlos, según las necesidades del programa.

En caso de que alguna de las operaciones protegidas *lance* una excepción, ésta será *capturada* por el bloque `catch`. En este bloque se incluye el *código manejador* de la excepción. En el ejemplo anterior, el manejo de la excepción consiste únicamente en informar al usuario mediante un mensaje escrito en la salida estándar y ejecutar la sentencia `return`, que termina inmediatamente la ejecución del método actual. Como este método es el propio método `main`, el efecto de la sentencia `return` es el de terminar la ejecución del programa.

En la declaración del bloque `catch`, se utiliza la expresión `Exception e`, con dos objetivos. En primer lugar, esta expresión indica el tipo de excepciones que van a ser capturadas en caso de que se produzcan. En el ejemplo anterior, se captura cualquier excepción de tipo `Exception` o de cualquiera de sus subclases. Además, la expresión "Exception e" es también la declaración de una variable e que representa a la propia excepción y permite obtener información relativa a la misma (tal como se presenta en el apartado 5.3). El ámbito de esta variable es solo el bloque `catch`, por lo que no puede ser utilizada fuera de dicho bloque.

Una vez ejecutado el bloque `catch`, la ejecución del programa continúa en la instrucción siguiente al bloque `try-catch`, a menos que la ejecución sea *alterada* de alguna manera, por ejemplo, mediante una sentencia `return`, como en el ejemplo anterior. En el apartado 5.4.1 se introduce otra forma de alterar el flujo del programa dentro de un bloque `catch`. En el ejemplo anterior, otro posible tratamiento de la excepción podría consistir en asignar unos datos por defecto (y no ejecutar ninguna sentencia `return`), de manera que la ejecución del programa pueda continuar.

Las excepciones y el compilador

La gran mayoría de tipos de excepciones deben ser capturadas explícitamente mediante un bloque `try-catch`[1].

El compilador analiza el código fuente y comprueba si cada instrucción que puede lanzar alguna excepción está protegida mediante un bloque `try-catch` [2]. De lo contrario, el compilador interrumpirá la compilación y avisará del error, como en el siguiente ejemplo:

```
Excepciones.java:5: unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
                FileInputStream f= new FileInputStream("data.bin");
```

En el ejemplo anterior, el compilador indica que en la línea 5 del fichero `Excepciones.java` se realiza una acción que puede lanzar una excepción de tipo `FileNotFoundException` y que ésta línea debe ser protegida mediante un bloque `try-catch` (o una sentencia `throws` adecuada en el método que la contiene).

Por otra parte, el compilador también comprueba que los bloques `try-catch` se usen adecuadamente. Si se utiliza un bloque `try-catch` para proteger una determinada sección de código frente a un determinado tipo (o tipos) de excepciones pero dicho código no es capaz de generar dichas excepciones entonces el compilador interrumpe la compilación y avisa del error:

Por ejemplo, dado el siguiente fragmento de código:

```
 try {
   int i = 0;
 } catch (FileNotFoundException fnfe) {
   System.out.println("...");
 }
```

al compilarlo, el compilador produce el siguiente error:

```
Excepciones.java:7: exception java.io.FileNotFoundException is never
(/thrown in body of corresponding try statement
                } catch (FileNotFoundException fnfe) {
```

[1] O en su defecto, debe usarse una sentencia **`throws`** en la declaración del método correspondiente, tal como se indica en el apartado 5.4.

[2] O incluida en un método cuya declaración incluya una sentencia **`throws`** adecuada.

En el ejemplo anterior, se indica que el bloque `try-catch` está capturando una excepción (de tipo `FileNotFoundException` que nunca podrá ser lanzada por el código incluido en el bloque `try`.

Por último, conviene saber que existe un cierto subconjunto de tipos de excepciones que no necesitan ser explícitamente capturadas. Estos tipos de excepciones se conocen como *excepciones no capturadas* (se presentan en el apartado 5.2.7. En estos casos, el compilador no *obliga* a capturarlas mediante bloques `try-catch` (o cláusulas `throws`).

Asegurando el estado

Conviene tener en cuenta que, en caso de que se produzca una excepción al ejecutar un cierto conjunto de instrucciones contenidas en un bloque `try`, en general no se sabe con seguridad qué instrucciones se han ejecutado correctamente, cuáles han fallado y cuáles ni siquiera se han ejecutado. En general, en el bloque `catch` se debe asumir que todas las instrucciones del bloque `try` han fallado.

En el ejemplo anterior, se podría sustituir el bloque `catch` propuesto, por éste otro:

```
} catch (Exception e) {
  // Asignación de valores por defecto
  d1 = 0;
  d2 = -1;
  d3 = 1000;
}
```

Así, la ejecución del programa podría continuar, utilizando unos valores por defecto (en lugar de terminar la ejecución, como en el ejemplo inicial). En este caso, es necesario dar un valor por defecto a los tres datos leídos en el bloque `try`, porque no se sabe con seguridad qué datos han sido leídos correctamente y cuáles no.

5.2.2. Refinando la captura de excepciones

En el ejemplo propuesto, se utiliza un bloque `catch` que permite capturar excepciones de tipo `Exception`:

```
} catch (Exception e) {
  ...
}
```

De esta manera, se captura cualquier excepción de tipo `Exception` *o de cualquiera de sus subtipos*. En general, esta es una buena forma de *proteger* un cierto fragmento de código fuente frente a un conjunto bastante amplio de excepciones.

De forma alternativa, es posible *refinar* un poco el tipo de las excepciones capturadas. En el ejemplo propuesto, las instrucciones incluidas en el bloque `try` solamente pueden lanzar un subconjunto concreto de excepciones, como `FileNotFoundException` y otras subclases de `IOException`. En este caso sería posible cambiar la declaración anterior por ésta otra:

```
} catch (IOException e) {
  ...
}
```

De esta manera, el bloque `catch` solo captura aquellas excepciones que sean de tipo `IOException` o de alguna de sus subclases (por ejemplo, `FileNotFoundException`).

Este mecanismo de *captura selectiva* de excepciones resulta muy útil en combinación con el mecanismo de *captura múltiple* de excepciones (descrito en el apartado 5.2.3), que permite capturar distintos tipos de excepciones y manejar de forma distinta cada uno de ellos.

5.2.3. Capturando más de una excepción

En ocasiones, interesa proteger un determinado bloque de código frente a varias excepciones y manejar cada una de ellas de forma distinta. En estos casos, es posible indicar varios bloques `catch` alternativos, cada uno de los cuales se utilizará para capturar un tipo distinto de excepciones.

En el siguiente ejemplo, las excepciones relacionadas con la entrada/salida se manejan de una forma y el resto de excepciones se manejan de otra forma:

```
try {
  FileInputStream f = new FileInputStream("data.bin");

  // Lectura de los datos
  ...
} catch (IOException ioe) {
  // Asignación de valores por defecto
  d1 = 0;
  d2 = -1;
  d3 = 1000;
} catch (Exception e) {
  System.out.println("Se ha producido una excepción");
  return;
}
```

Así, en caso de que se produzca una excepción de entrada/salida, el programa continuará (con unos valores por defecto). En caso de que se produzca cualquier otra excepción, la ejecución del método actual será interrumpida, mediante la sentencia `return`.

En el ejemplo anterior, cuando se produce una excepción, su tipo es comparado con los tipos indicados en los distintos bloques `catch`, en el orden en que aparecen en el código fuente, hasta que se encuentra un tipo *compatible*. Por ejemplo, si en el caso anterior ocurre una excepción de tipo `FileNotFoundException`, en primer lugar dicho tipo será comparado con `IOException`. Como `FileNotFoundException` es un subtipo de `IOException`, entonces se ejecutará el primer bloque `catch`. En caso de que se produzca una excepción de tipo `IndexOutOfBoundsException` (u otro tipo distinto a `IOException` o a alguno de sus subtipos), entonces se comparará dicho tipo con el tipo `Exception` declarado en el segundo bloque `catch`. Como `IndexOutOfBoundsException` es un subtipo de `Exception`, la excepción será manejada en el segundo bloque `catch`.

Por ello, el orden en que se indican los distintos bloques `catch` es muy importante, pues determina qué bloque `catch` se ejecutará en cada caso. Además, hay que tener en cuenta que no todas las combinaciones posibles son correctas. Si, en el ejemplo anterior, el orden de los bloques `catch` fuera el contrario (primero un catch capturando `Exception` y luego el otro capturando `IOException`), obtendríamos un error de compilación, debido a que el tipo `Exception` es más general que el tipo `IOException` y éste a su vez es más general que `FileNotFoundException`.

Por eso, conviene elegir cuidadosamente el orden de los bloques `catch`. Los bloques `catch` que capturan excepciones de tipos concretos (por ejemplo, `FileNotFoundException`) deben ser los primeros. A continuación se indican los bloques para excepciones algo más generales (por ejemplo, `IOException`). Por último se indican los bloques con excepciones muy generales (por ejemplo, `Exception`).

En cualquier caso, *nunca* se ejecuta más de un bloque `catch`. En el ejemplo anterior, si se produce una excepción de tipo `FileNotFoundException`, ésta será manejada *únicamente* en el bloque `catch` que captura `IOException`. Una vez terminado dicho bloque `catch`, *no*

se ejecutará ningún otro bloque, sino que la ejecución del programa continuará a partir de la sentencia siguiente al bloque `try-catch`.

Por último, conviene tener en cuenta que los nombres de las variables utilizados en las cláusulas `catch` no deben *colisionar* entre sí ni con ninguna otra variable o atributo. En el ejemplo anterior, se han utilizado las variables `ioe`, para el bloque que captura `IOException` y `e`, para el bloque que captura `Exception` (asumiendo que no hay ningún atributo ni variable llamados `ioe` o `e`). Es una buena práctica utilizar algún convenio para nombrar las variables utilizadas en las cláusulas `catch`. Un convenio conocido es utilizar las iniciales de cada tipo de excepción (`ioe` para `IOException`, `fnfe` para `FileNotFoundException`, etc.). Puede ocurrir que en un mismo ámbito (por ejemplo, en un mismo método, haya varios bloques `try-catch`, que capturen el mismo tipo de excepción. En este caso, se recomienda numerar las variables (por ejemplo, `ioe`, `ioe2`, etc., para distinguir distintas excepciones de tipo `IOException`).

5.2.4. `try-catch` anidados

En algunos casos, el código manejador de una excepción puede a su vez lanzar una excepción. En el siguiente ejemplo, se intenta abrir un determinado fichero y leer algunos datos. Como estas operaciones pueden lanzar una excepción, se protegen mediante un bloque `try-catch`. El manejo de las posibles excepciones consiste en abrir un segundo fichero y guardar unos ciertos datos. Como estas operaciones también pueden lanzar excepciones, es necesario protegerlas mediante su correspondiente bloque `try-catch`.

```
try {
  f = new FileInputStream("data.bin");
  a = f.read();
  b = f.read();
  ...
} catch (Exception e) {
  try {
    FileOutputStream f2 = new FileOutputStream("incidencias.log")
    f2.write(...);
  } catch (Exception e2) {
    ...
  }
}
```

Aunque el lenguaje permite el uso de varios niveles de anidamiento de bloques `try-catch`, en general es recomendable evitar su uso, pues dificultan la lectura y el mantenimiento del código. En su lugar, es preferible una solución alternativa mediante sentencias de control convencionales. El ejemplo anterior podría reescribirse de la siguiente forma:

```
boolean datosLeidos;
try {
  f = new FileInputStream("data.bin");
  a = f.read();
  b = f.read();
  ...
  datosLeidos = true;
} catch (Exception e) {
  datosLeidos = false;
}
```

```
if (datosLeidos == false) {
  try {
    FileOutputStream f2 = new FileOutputStream("incidencias.log")
    f2.write(...);
  } catch (Exception e2) {
    ...
  }
}
```

En el apartado 5.4 se relacionan los bloques `try-catch` anidados con otras características del tratamiento de excepciones en Java.

5.2.5. La sentencia *finally*

En ocasiones, a la hora de escribir un bloque `try-catch`, es conveniente incluir algún fragmento de código que se ejecute tanto si el bloque `try` se ejecuta correctamente por completo como en el caso de que se produzca una excepción. Para ello, se puede añadir una cláusula `finally` a un bloque `try-catch`, tal como se ilustra en el siguiente ejemplo:

```
FileInputStream f = null;
int a, b;
try {
  f = new FileInputStream(fichero);
  a = f.read();
  b = f.read();
  ...
} catch (IOException e) {
  ...
} finally {
  if (f != null)
    f.close();
}
```

En el ejemplo anterior, se utiliza un bloque `finally` para cerrar el flujo de datos, en caso de que haya sido abierto.

En el apartado 5.4 se relaciona la cláusula `finally` con otras características del tratamiento de excepciones en Java y los bloques `try-catch`.

5.2.6. La forma general de `try-catch`

A modo de resumen de los conceptos relacionados con el uso de excepciones en Java presentados hasta el momento, conviene presentar la forma general de las sentencias `try-catch`:

```
try {
  // Fragmento de código que puede generar una excepción
} catch (tipo_de_excepción variable) {
  // Fragmento de código manejador de la excepción
} catch (tipo_de_excepción variable) {
  // Fragmento de código manejador de la excepción
} ... {
} finally {
  // Código que se ejecuta siempre
}
```

Como se ha indicado anteriormente es posible utilizar varios bloques `catch`, aunque es posible utilizar varios, para manejar de forma diferente distintos tipos de excepciones. Además, es posible incluir un bloque `finally` opcional. En caso de utilizarlo, este bloque se ejecutará en cualquier caso, tanto si las instrucciones del bloque `try` se ejecutan correctamente como si se produce alguna excepción.

5.2.7. Excepciones no capturadas

Existe un subconjunto de excepciones que pueden producirse durante la ejecución de operaciones muy comunes. Por ejemplo, caulquier instrucción del tipo

```
objeto.metodo(param1, param2, ...);
```

puede lanzar una excepción de tipo `NullPointerException` en caso de que la referencia objeto no haya sido inicializada y por tanto apunte a `null`.

Otro ejemplo común es la excepción `IndexOutOfBoundsException`, que se puede lanzar en una instrucción como la siguiente

```
int v = enteros[i];
```

en caso de que el valor de `i` no sea válido (cuando es menor que 0 o mayor o igual que `enteros.length`).

Existen otros casos muy comunes como las excepciones de tipo `ArithmeticException`, que se lanza, por ejemplo, al hacer una división entre cero, o las de tipo `ClassCastException`, que se lanzan cuando se intenta hacer una operación de *casting* incorrecta, como la siguiente:

```
Linea linea = new Linea(punto1, punto2);
Cuadrado cuadrado = (Cuadrado)linea;
```

Si se quisiera proteger el código que puede lanzar estas excepciones, prácticamente habría que proteger cualquier línea de código dentro de algún bloque `try-catch`. El resultado sería un programa poco legible e *inmantenible*.

Para evitar este problema, éstas y otras excepciones *no necesitan ser capturadas*. En su lugar, *se confía* en que el programador tomará las medidas necesarias para evitar que pasen.

Por ejemplo, para evitar las excepciones de tipo `NullPointerException`, el programador debe asegurarse de inicializar correctamente sus objetos. Además, puede hacer comprobaciones del tipo `if (objeto == null) ...`, para evitar que ocurran dichas excepciones. Para evitar excepciones de tipo `IndexOutOfBoundsException`, el programador debe asegurarse de que los índices utilizados en los accesos a arrays son correctos. En el caso de las excepciones de tipo `ArithmeticException`, se deben hacer las comparaciones sobre los operandos de las expresiones matemáticas a evaluar y en el caso de las excepciones de tipo `ClassCastException`, se puede hacer uso del operador `instanceof` para comprobar previamente que el *cast* a realizar es correcto. En otros casos, el programador deberá hacer las comprobaciones correspondientes.

Todos los tipos de excepciones que no necesitan ser capturadas son subclases (directas o indirectas) de `RuntimeException`, que a su vez es una subclase de `Exception`, como se indica en la figura 5.2.

A este tipo de excepciones se les suele llamar *excepciones no capturadas*[3], porque no suelen capturarse explícitamente.

Conviene notar que, en caso de que sea necesario por algún motivo, estas excepciones pueden ser capturadas como cualquier otra. En el siguiente ejemplo se manejan de forma distinta las excepciones debidas a la evaluación matemática y el resto de excepciones:

[3] En los libros y manuales escritos en inglés suelen llamarse *non-checked exceptions*.

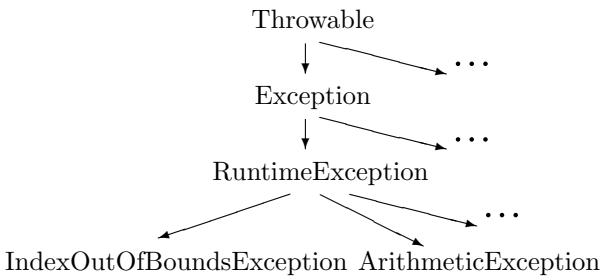

Figura 5.2: Jerarquía de excepciones no capturadas en Java (fragmento)

```
int resultado;
try {
  dato1 = ...
  dato2 = ...
  resultado = ...
} catch (ArithmeticException ae) {
  System.out.println("Los operandos no son correctos");
  resultado = 0;
} catch (Exception e) {
  return;
}
```

Conviene notar que este uso de los bloques `try-catch` es, en cierta manera, una forma de *controlar el flujo* del programa. Por ejemplo, el ejemplo anterior es similar al siguiente:

```
int resultado;
dato1 = ...
dato2 = ...
if (correctos(dato1, dato2) == true) {
  resultado = ...
} else {
  ...
}
```

Aunque es posible utilizar los bloques `try-catch` para realizar el control de flujo de un programa, en general no es recomendable . En su lugar, es preferible utilizar sentencias de control de flujo convencionales (por ejemplo, condiciones `if`).

5.3. La clase `Throwable`

Tal como se ha indicado en el apartado 5.1, la clase `Throwable` es la clase base que representa a todas las excepciones que pueden ocurrir en un programa Java.

En esta clase existen varios métodos útiles, que son heredados por las distintas subclases (directas e indirectas) de `Throwable`. En los siguientes subapartados se presentan dos de los métodos más utilizados. En la documentación de la clase `Throwable` es posible encontrar métodos adicionales que pueden resultar útiles en las fases de depuración de un programa Java.

5.3.1. El método getMessage

Uno de los métodos más útiles definido en la clase `Throwable` y heredado por todas sus subclases es el método `getMessage`. Este método permite obtener información de una excepción.

El siguiente ejemplo ilustra su uso:

```
try {
  C c = new C();
  c.m1();
} catch (FileNotFoundException fnfe) {
  System.out.println(fnfe.getMessage());
}
```

En el bloque `catch` se capturan las excepciones de tipo `FileNotFoundException`, que se producen cuando se intenta abrir un fichero que no existe. En caso de que se produzca una excepción de dicho tipo, en la salida estándar del programa se escribirá algo como:

```
noexiste.dat (No such file or directory)
```

(suponiendo que el fichero que se intenta abrir se llame `noexiste.dat`).

Como se puede observar, el método `getMessage` permite obtener información útil sobre la excepción. El usuario puede utilizar esta información para intentar solucionar el error. Por ejemplo, en este caso, se muestra el nombre del fichero que se intenta abrir. Esta indicación podría servir para que el usuario recuerde que es necesario que el fichero de datos utilizado por el programa debe tener un nombre concreto, deber estar ubicado en algún directorio concreto, etc.

5.3.2. El método printStackTrace

Otro de los métodos más útiles es el método `printStackTrace`. Este método se utiliza normalmente dentro de un bloque `catch`, como parte del código manejador de una excepción, para mostrar la secuencia de operaciones que han provocado la excepción:

```
try {
  FileInputStream f;
  f = new FileInputStream("fichero.dat");
  ...
} catch (FileNotFoundException fnfe) {
  fnfe.printStackTrace();
}
```

Este método produce una salida como la siguiente:

```
java.io.FileNotFoundException: noexiste.dat (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:106)
        at java.io.FileInputStream.<init>(FileInputStream.java:66)
        at C.m2(EjemploTraza.java:11)
        at C.m1(EjemploTraza.java:6)
        at EjemploTraza.main(EjemploTraza.java:19)
```

En la primera línea de la traza se indica el tipo de excepción que se ha producido (en el ejemplo, `java.io.FileNotFoundException`) y un mensaje con más información. Nótese que este mensaje es exactamente el mismo devuelto por el método `getMessage`, según se indicó en el apartado anterior.

Además, el método `printStackTrace` escribe en la salida estándar del programa una *traza* de la excepción. La traza está formada por varias líneas que permiten conocer cuál ha sido el flujo del programa, justo hasta el momento en que se produjo la excepción. Si se observan las líneas, es posible saber que la excepción se produjo en el método open de la clase `FileInputStream`. Dicho método fue invocado por el constructor de `FileInputStream`[4], en la línea 106 del fichero `FileInputStream.java`[5], que a su vez fue invocado por otro constructor distinto de `FileInputStream`, en la línea 66 del mismo fichero. Este otro constructor fue invocado por el método `m2` de la clase `C` (en la línea 11 del fichero `EjemploTraza.java`). Dicho método fue invocado por el método `m1` de la clase `C` (en la línea 6 del mismo fichero) y éste a su vez fue invocado por el método `main` de la clase `EjemploTraza` (en la línea 19 del mismo fichero).

Esta información es importante para depurar el error producido y corregirlo. La forma habitual de depurar un programa a partir de una traza consiste en inspeccionar todas las líneas de la traza y averiguar cuál es la primera que pertenece a alguna clase del programa que se está escribiendo. En este caso, esa primera línea sería la cuarta línea de la traza anterior (la línea 11 de `EjemploTraza.java`). Una vez identificado dicho punto, se debe revisar con detenimiento la línea del fichero fuente indicado, en busca del error.

5.4. Captura vs. propagación de excepciones: la sentencia `throws`

En los apartados anteriores se ha presentado la forma en que un programa Java puede capturar y manejar excepciones, de manera que las excepciones son manejadas lo antes posible a partir del momento en que se producen.

Sin embargo, en ocasiones conviene *retrasar* el manejo de una excepción por diversos motivos. Considérese el siguiente método:

```
private void leerDatos (String fichero) {
  FileInputStream f = new FileInputStream(fichero);
  f.read(...);
  f.close();
}
```

Como en casos anteriores, es necesario proteger mediante un bloque `try-catch` las instrucciones anteriores, frente a las excepciones de tipo `IOException` que puedan producirse. El método modificado sería el siguiente:

```
private void leerDatos (String fichero) {
  try {
    FileInputStream f = new FileInputStream(fichero);
    f.read(...);
    f.close();
  } catch (IOException ioe) {
    ...
```

[4]Se sabe que el invocador fue un constructor, por la cadena **`init`** que aparece en la línea correspondiente.
[5]Este fichero pertenece a las librerías estándar de Java.

```
    }
  }
```

Una alternativa sería *propagar* las excepciones, lo que permitiría *retrasar* su manejo. En lugar de utilizar un bloque `try-catch` que contenga las instrucciones que pueden lanzar excepciones, esta alternativa consiste en utilizar una *cláusula* **`throws`**, en la declaración del método que contiene dichas instrucciones, de la siguiente manera:

```
private void leerDatos (String fichero) throws IOException {
  FileInputStream f = new FileInputStream(fichero);
  f.read(...);
  f.close();
}
```

Mediante la cláusula `throws`, se está indicando que el código del método `leerDatos` *puede* lanzar excepciones de tipo `IOException`. Esto obliga a que cualquier invocación del método `leerDatos` esté protegida mediante un `try-catch` donde se capturen las excepciones de tipo `IOException` (o alguna clase padre, como `Exception`):

```
private void leerDatosCompras () {
  try {
    leerDatos("compras1.dat");
    leerDatos("compras2.dat");
  } catch (IOException ioe) {
    System.out.println("No es posible leer los datos de compras");
    return;
  }
}
```

De esta manera, es posible hacer un manejo de excepciones *globalizado*. En el ejemplo anterior, no interesa hacer un tratamiento particular en función del fichero cuya lectura haya fallado, sino un tratamiento global.

En cualquier caso, siempre es necesario usar alguna de las dos alternativas. Si una instrucción puede lanzar excepciones de un determinado tipo, entonces éstas deben ser capturadas mediante un bloque `try-catch` que contenga a dicha instrucción o, alternativamente, esta instrucción debe pertenecer a un método en cuya declaración se incluya una cláusula `throws` con la excepción correspondiente.

5.4.1. Otra forma de propagar excepciones: la sentencia `throw`

En ocasiones interesa utilizar las dos alternativas para capturar excepciones presentadas hasta el momento: la captura *inmediata* de excepciones y la *propagación* y captura *tardía* de las excepciones.

Existen dos casos donde conviene combinar ambas técnicas. El primer caso se da cuando se pretende manejar dos (o más) veces una misma excepción, desde distintos lugares del programa. Típicamente uno de los lugares es el método donde se lanza la excepción y el otro es un segundo método, que invoca al primero.

Existe un segundo caso, que se da cuando se prefiere capturar, en el momento en que se producen, solamente algunas excepciones. El resto de excepciones que se pudieran lanzar serán manejadas por el método invocante.

En los siguientes apartados se exploran ambos casos.

Caso 1: capturando varias veces la misma excepción

En el siguiente ejemplo se captura una excepción justo en el momento en que se produce, luego se *relanza* y posteriormente se vuelve a capturar, en el método invocante.

```
public void calcularBalance () {
  float nuevoBalance;
  try {
    ...
    float ingresos = calcularIngresos();
    float gastos = calcularGastos();
    float variaciones = calcularVariaciones();
    nuevoBalance = ingresos - gastos + variaciones;
  } catch (Exception e) {
    System.out.println("No es posible calcular el balance");
    this.balance = 0;
    return;
  }
  this.balance = nuevoBalance;
}

public float calcularIngresos () throws ArithmeticException {
  try {
    float ingresos = ...
    return ingresos;
  } catch (ArithmeticException ae) {
    System.out.println("No es posible calcular los ingresos");
    // Relanzar la excepción que se acaba de capturar
    throw ae;
  }
}
```

En este ejemplo, la captura inmediata de las excepciones de tipo `ArithmeticException` permite hacer un primer manejo de las excepciones de este tipo. La sentencia `throw` que hay en el bloque `catch` del método calcularIngresos permite *relanzar* la excepción. Si se produce una `ArithmeticException` en el método `calcularIngresos`, se ejecuta el bloque `catch` correspondiente. Cuando la ejecución del método `calcularIngresos` alcanza la sentencia `throw`, la ejecución del bloque `catch` se interrumpe y la ejecución *salta* al bloque `catch` del método `calcularBalance`, donde dicha excepción (en forma de `Exception`) es manejada.

En el método `calcularBalance`, es posible hacer un segundo tratamiento de la excepción. Sin embargo, nótese que en el método `calcularBalance` se está capturando *cualquier excepción (subclase de* `Exception`*)*, en lugar de capturar específicamente las excepciones de tipo `ArithmeticException`. Esto permite capturar, con el mismo bloque `catch`, las excepciones de tipo `ArithmeticException` producidas por los métodos `calcularIngresos` y `calcularGastos` y además, cualquier excepción de otro tipo producida en algún otro método y por tanto manejar de la misma manera todas las excepciones.

Caso 2: Eligiendo entre *captura inmediata* y propagación de excepciones

En este segundo caso se estudia la posibilidad de capturar solo algunas excepciones de forma *inmediata* y propagar el resto de excepciones al método invocante.

Por ejemplo, el método `calcularIngresos` podría ser el siguiente:

```
public float calcularIngresos () throws IOException {
  try {
    FileInputStream f = new FileInputStream(...);
    float ingresos = ...
    return ingresos;
  } catch (ArithmeticException ae) {
    ...
  }
}
```

En este caso, el bloque `catch` solo captura las excepciones de tipo `ArithmeticException`. Las posibles excepciones de tipo `IOException` son propagadas al método invocante (el método `calcularBalance`, del ejemplo anterior) y capturadas por éste.

5.4.2. Sentencias `throw` y bloques `finally`

Tal como se ha indicado en el apartado 5.2.5, en un bloque `try-catch`-finally, el bloque `finally` se ejecuta en cualquier caso.

Existen algunos casos particulares en los que el flujo de ejecución no es evidente:

Bloque `catch` y sentencia `throw` Considérese el siguiente ejemplo:

```
try {
  ...
} catch (Exception e) {
  ...
  throw e;
} finally {
  ...
}
```

Si se produce una excepción, se ejecutará el bloque `catch`. Antes de ejecutar la sentencia `throw`, se ejecutará el bloque `finally`. Terminado este bloque, se realizará la propagación de la excepción, por lo que esta será *entregada* al método invocante.

Cláusula `throws` Considérese el siguiente ejemplo:

```
public void m () throws Exception {
  try {
    FileInputStream f = new FileInputStream(...);
    ...
  } catch (IOException ioe) {
    ...
  } finally {
    ...
  }
}
```

Si se produce una excepción de un tipo distinto a `IOException`, por ejemplo una excepción de tipo `NullPointerException`, el bloque `catch` *no se ejecutará*, pero *sí* se ejecutará el bloque `finally`. Por último, se propagará la excepción al método invocante.

5.5. Lanzando nuevas excepciones

En los apartados anteriores se ha presentado el soporte de excepciones ofrecido por la plataforma de desarrollo Java y se ha hecho un especial énfasis en la captura de las excepciones. También se ha introducido la forma en que un programa puede *propagar* y *relanzar* excepciones.

Otra posibilidad ofrecida por el lenguaje Java es la de lanzar nuevas excepciones. Estas pueden ser de algún tipo existente (como IOException) o incluso pueden ser instancias de nuevos tipos de excepciones. Estas nuevas posibilidades se presentan en los siguientes apartados.

5.5.1. Lanzando excepciones de tipos existentes

Considérese el siguiente ejemplo:

```
public void calcularBalance (float ingresos, float gastos,
  float variaciones) {
  boolean correcto = validar(ingresos, gastos, variaciones);
  if (correcto == true)
    balance = f(ingresos, gastos, variaciones);
  else
    balance = -1;
  return balance;
}
```

En este método, se comprueba la validez de determinados parámetros y en su caso, se calcula un cierto balance. En caso de que algún parámetro no sea correcto, se devolverá como resultado un determinado valor especial (-1). Esta solución presenta dos problemas. El primer problema es la elección del valor especial. Éste no puede ser un valor que pueda ser considerado como un resultado correcto, pues entonces no habría forma de saber si el valor devuelto es el valor especial que indica un error en los parámetros o en cambio es el valor del balance. En el ejemplo anterior, el valor -1 no parece un valor adecuado, pues no permite distinguir aquellos casos en que realmente el balance tiene un valor igual a -1. El segundo problema es que el método que invoca al método `calcularBalance` *debe conocer* qué valor se ha decidido utilizar en caso de error. Si, en algún momento, se decide cambiar el valor especial -1 a otro valor (como -123456789), entonces habrá que comprobar y posiblemente cambiar también todos aquellos fragmentos de código donde se invoque al método `calcularBalance`.

Estos problemas se podrían solucionar sustituyendo el método `calcularBalance` por esta otra versión:

```
public void calcularBalance (float ingresos, float gastos,
  float variaciones) throws IllegalArgumentException {
{
  boolean correcto = validar(ingresos, gastos, variaciones);
  if (correcto == false)
    throw new IllegalArgumentException("Los datos no son correctos");
  else {
    balance = f(ingresos, gastos, variaciones);
    return balance;
  }
}
```

En esta nueva versión, se lanza una excepción en caso de que los parámetros no sean correctos. Esta excepción es de tipo `IllegalArgumentException`, un subtipo de `Exception`, utilizado para señalar precisamente aquellas situaciones en que los parámetros o datos de entrada proporcionados no son correctos por algún motivo.

De esta forma, no es necesario utilizar ningún valor especial para señalar una situación anormal y por tanto el método que invoca al método `calcularBalance` no tiene por qué conocer ningún valor especial.

5.5.2. Lanzando excepciones de nuevos tipos

Como segunda mejora al ejemplo anterior, se podría utilizar un nuevo tipo de excepciones, creado específicamente para señalar tal condición errónea.

En primer lugar, es posible crear un nuevo tipo de excepciones, simplemente definiendo la siguiente clase:

```
public class ParametrosException
extends Exception {
  private float gastos;
  private float ingresos;
  private float variaciones;
  public ParametrosException (float gastos, float ingresos,
    float variaciones) {
    this.gastos = gastos;
    this.ingresos = ingresos;
    this.variaciones = variaciones;
  }
  public String getMessage () {
    return "Parámetros incorrectos: gastos=" + gastos +
      ", ingresos=" + ingresos + ", variaciones=" + variaciones;
  }
}
```

La clase `ParametrosException` es considerada una excepción ya que extiende la clase `Exception`[6]. Por ello, podemos capturarla en bloques `try-catch`, lanzarla mediante sentencias `throw`, declararla en métodos mediante cláusulas `throws`, etc.

En la clase `ParametrosException` se sobreescribe el método `getMessage` (heredado de `Throwable`), para que devuelva una cadena indicando los valores de los atributos cuya validación ha producido la excepción.

Una vez definida la clase, es posible crear un nueva instancia, mediante el operador `new` y lanzar la excepción creada, mediante una sentencia `throw`, como en el siguiente ejemplo:

```
public void otroMetodo (float gastos, float ingresos,
  float variaciones) {
  boolean correctos = comprobar(gastos, ingresos, variaciones);
  if (correctos == false)
    throw new ParametrosException(gastos, ingresos, variaciones);
  else
    ...
}
```

[6]En la práctica, en ocasiones conviene utilizar como clase base una excepción de un tipo más concreto. En el ejemplo anterior, en lugar de utilizar la clase **`Exception`** como clase base se podría utilizar la clase **`IllegalArgumentExcepion`**, que en general se utiliza para representar excepciones producidas por operaciones cuando reciben parámetros incorrectos.

Esta excepción se podría capturar igual que el resto de excepciones, como se ha indicado en apartados anteriores. Por ejemplo, es posible utilizar el siguiente código:

```
try {
  otroMetodo(1.0, 2.0, 3.0);
} catch (Exception e) {
  e.printStackTrace();
}
```

La traza de una excepción de tipo `ParametrosException` sería como la siguiente:

```
Exception in thread "main" ParametrosException: Parámetros incorrectos:
gastos=1.0, ingresos=2.0, variaciones=3.0
        at OtraClase.otroMetodo(ParametrosException.java:21)
        at EjemploParametros.metodo(ParametrosException.java:27)
        at EjemploParametros.main(ParametrosException.java:31)
```

Nótese que en la primera línea de la traza se incluye la cadena devuelta por el método `getMessage` sobreescrito en `ParametrosException`.

5.6. Cuestiones

1. Indica cuál es la clase base de la jerarquía de excepciones de Java. Indica cuáles son sus subclases directas y las diferencias entre ellas. Consulta la ayuda de la librería estándar para obtener más información.

2. Indica el nombre de tres tipos de excepciones que no se hayan nombrado en este capítulo. Para ello, consulta la ayuda de la librería estándar de Java.

3. Explica qué ventajas e inconvenientes presentan las excepciones no capturadas frente el resto de excepciones.

4. Explica en qué casos es necesario hacer una captura múltiple de excepciones (mediante varios bloques `catch`).

5. Explica la utilidad de los bloques `finally`. Indica en qué casos son obligatorios.

5.7. Ejercicios resueltos

1. Dado el siguiente código, indica los errores que contiene

```
public void metodo (int arg1, int arg2) {
  try
    float auxiliar = 0;
    auxiliar = arg1 / arg2;
  catch Exception e {
    System.out.println("auxiliar = " + auxiliar);
    throws e;
  } finally {
    System.out.println("Terminando el try-catch...");
    throw e;
  }
  } catch (ArithmeticException e) {
```

```
      System.out.println("Los parámetros no son correctos");
    } catch (IOException e) {
      System.out.println("Excepción de entrada/salida");
    }
    System.out.println("auxiliar es " + auxiliar);
  }
```

Solución

a) El bloque `try` debe ir encerrado entre llaves (`{` y `}`).

b) En el primer bloque `catch`, la expresión `Exception e` debe escribirse entre paréntesis.

c) La variable `e` se está utilizando en los tres bloques `catch` (solo puede ser utilizada en uno de ellos).

d) La sentencia `throws e` debe ser `throw e` (la palabra reservada `throws` solo se utiliza en la declaración de los métodos).

e) Si se quiere permitir relanzar excepciones de tipo `Exception`, mediante una sentencia `throw e`, es necesario añadir la cláusula `throws Exception` en la declaración del método.

f) El orden de los bloques `catch` es incorrecto. En primer lugar debe indicarse el `catch` para las excepciones de tipos más concretos (`ArithmeticException` e `IOException`) y posteriormente deben indicarse los de tipos más generales (`Exception`).

g) El bloque `finally` debe estar después de todos los bloques `catch`.

h) En un bloque `finally` no es posible ejecutar una sentencia `throw`.

i) La variable `auxiliar` solo es visible en el bloque `try` (no es visible en los bloques `catch` ni fuera del bloque `try`). Debe declararse fuera del bloque `try-catch`, para que sea visible en todo el método.

j) Las sentencias del bloque `try` no pueden lanzar nunca una `IOException`, por lo que el bloque `catch` que captura `IOException` debe ser eliminado.

2. Dado el siguiente fragmento de código:

```
LinkedList lista = new LinkedList();
lista.add("uno");
lista.add("dos");
String p = (String)lista.get(0);
```

averigua, mediante la ayuda de la librería estándar de Java, qué tipos de excepciones pueden lanzarse. Añade el *mínimo código imprescindible* para capturar y manejar *por separado* cada uno de dichos tipos.

Solución Ni el constructor utilizado ni el método `add` lanzan ninguna excepción. El método `get` puede lanzar una excepción de tipo `IndexOutOfBoundsException`, pero ésta es una *excepción no capturada*, por lo que no es necesario capturarla ni propagarla.

3. Modifica el siguiente código para que propague las excepciones, en lugar de capturarlas:

```
public void metodo () {
  int i;

  try {
    FileInputStream f = new FileInputStream("f.txt");
    i = f.read(...);
    ...
  } catch (IOException e) {
    System.out.println("Error de entrada/salida: " + e.getMessage());
    return;
  } catch (Exception e) {
    e.printStackTrace();
    return;
  }

  if (i == 0)
    ...
}
```

Solución La solución más sencilla consiste en eliminar la cláusula `try` y los bloques `catch` y añadir la cláusula `throws`, como se indica a continuación:

```
public void metodo () throws Exception {
  int i;

  FileInputStream f = new FileInputStream("f.txt");
  i = f.read(...);

  if (i == 0)
    ...
}
```

Sin embargo, conviene tener que con esta solución se elimina la posibilidad de manejar de forma diferente excepciones de distinto tipo.

5.8. Ejercicios propuestos

1. Dado el siguiente código, indica qué problema existe (si es necesario, completa el ejemplo con el código necesario para que pueda compilarse y ejecutarse, compílalo e interpreta los mensajes de error proporcionados por el compilador):

```
public void m1 () {
  try {
    m2(1, 0);
  } catch (ArithmeticException ae) {
    System.out.println("Error aritmético al ejecutar m2");
    ae.printStackTrace();
  }
}

public void m2 (int a, int b) throws Exception {
  try {
```

```
      float f = a/b;
      System.out.println("f = " + f);
    } catch (ArithmeticException ae) {
      ae.printStackTrace();
      throw ae;
    } catch (Exception e) {
      System.out.println("Error de otro tipo");
      e.printStackTrace();
    }
  }
```

2. Dadas las siguientes clases Java, indicar qué se escribe en la salida estándar cuando se ejecuta el método `main`:

```
class C {
  private int a;
  public void m1 () {
    a = 0;
    m2();
    System.out.println(a);
  }
  public void m2 ()
  throws NullPointerException {
    try {
      m3();
    } finally {
      System.out.println(a);
      a = 1;
    }
    System.out.println(a);
  }
  private void m3 () {
    if (a == 0)
      throw new NullPointerException();
  }
}
public class TryFinally {
  public static void main (String [] args) {
    new C().m1();
  }
}
```

Capítulo 6

Interfaz gráfica de usuario y applets

6.1. Introducción

Un *applet* es una aplicación JAVA diseñada para ejecutarse en el contexto de un navegador web. De la misma forma que un navegador carga texto, gráficos y sonido a partir de las indicaciones contenidas en un fichero HTML, puede cargar el código de un applet JAVA y lanzarlo a ejecución en respuesta a una etiqueta (tag) específica contenida en dicho fichero HTML.

El applet es responsable de la gestión de lo que se denomina *área de interacción del applet*, que es una parte del documento web. Este esquema obliga a abandonar la entrada/salida orientada a consola y a trabajar en modo gráfico. Una interfaz gráfica facilita la interacción y permite mejorar la cantidad o calidad de la información que se muestra, pero desde el punto de vista del programador presenta nuevas necesidades e introduce cierto grado de complejidad:

- Se rompe la noción de secuencialidad: en una interfaz basada en texto hay una secuencia clara de introducción de datos y visualización de resultados (esquema rígido pero simple); una interfaz gráfica permite que el usuario interaccione con mucha mayor libertad, sin imponer un orden estricto (por ejemplo, al rellenar un formulario los campos pueden cumplimentarse en cualquier orden). Para abordar este tipo de problemas se utiliza la *programación dirigida por eventos*. Se entiende por *evento* cualquier acción externa a la aplicación que pueda afectar a la misma (en este caso cualquier acción del usuario, tal como pulsación de una tecla o utilización del ratón). El sistema registra los eventos y los almacena en una cola (*cola de eventos*). La aplicación extrae los eventos de la cola y determina la acción a realizar en respuesta a cada uno.
- Se necesita un conjunto de primitivas gráficas que permitan manipular la pantalla, así como bibliotecas que soporten los distintos componentes de interfaz (menús, botones, áreas de texto, barras de desplazamiento, etc.). Además, debe gestionarse la ubicación de dichos componentes dentro del área de interacción del applet mediante potentes esquemas de ubicación que permitan especificar relaciones entre las posiciones de los componentes en lugar de posiciones absolutas. Esto es necesario ya que un applet debe integrarse en un documento HTML que no se muestra con un formato fijo sino que se presentará de forma ligeramente distinta en navegadores distintos.

En consecuencia, un applet JAVA posee las siguientes características:

- Se utiliza intensivamente la biblioteca AWT (*Abstract Window Toolkit*), donde se definen los componentes de interfaz (o *componentes de interacción de usuario*: botones, menús, campos de edición de texto...), primitivas gráficas, y eventos generados por el usuario; así como la biblioteca Swing, que extiende a la anterior para crear componentes *ligeros* (*lightweight*), incorporando nuevas clases (los `JComponents`) y mejorando el aspecto visual. Cada componente de interacción posee una representación gráfica y un conjunto de métodos para su uso desde el programa. Además, cada componente reconoce y reacciona ante determinados eventos (por ejemplo, un botón reacciona ante un *click* de ratón) propagando el evento a aquellas partes del programa que previamente han informado de su interés en gestionar los mismos.
- La entrada no se basa en métodos orientados a consola, como `System.in.read()`. Como ya se ha mencionado, las aplicaciones gráficas se estructuran según el modelo denominado *programación dirigida por eventos*. El programa fija una interfaz y especifica las respuestas adecuadas (acciones) ante los posibles eventos (por ejemplo, generados por el ratón o teclado). Durante la ejecución, el programa se limita a responder ante los eventos disparando la acción adecuada.
- De la misma forma, la salida no se basa en métodos orientados a consola como por ejemplo `System.out.print()`, sino que se utiliza en su lugar el concepto de contexto gráfico y un conjunto de primitivas gráficas para: gestionar texto (distintas fuentes, tamaños, etc.); crear líneas, círculos, rectángulos, etc., con o sin relleno; gestionar colores de dibujo, relleno y fondo...

Una aplicación JAVA autónoma (lanzada directamente desde el sistema operativo) típicamente funciona o en modo consola o en modo gráfico. Un applet únicamente puede funcionar en modo gráfico. Una aplicación autónoma en modo gráfico es tan parecida a un applet que se pueden construir fácilmente programas que se ejecuten indistintamente desde el sistema operativo (como aplicación autónoma) o en el contexto de un navegador (como applet).

6.2. Applets

6.2.1. Concepto de applet

Un navegador web es un programa diseñado para descargar un fichero HTML (a partir del almacenamiento local o a través de la red), interpretar su contenido, y crear una imagen del mismo. Un fichero HTML contiene un conjunto de etiquetas (*tags*) que delimitan sus contenidos. El estándar HTML define distintas etiquetas para propósitos diferentes: delimitar un enlace dinámico, indicar un tipo de letra específico, indicar la inclusión de un fichero de imagen o de sonido, etc. Además, define también una etiqueta específica para incluir referencias a programas JAVA (applets).

Supongamos que se ejecuta un navegador web para acceder a un fichero HTML que contiene la siguiente información (los puntos suspensivos indican información adicional que no afecta a la discusión),

```
... <APPLET CODE="X.class" HEIGHT="400" WIDTH="700"></APPLET> ...
```

Cuando el navegador descarga el fichero HTML debe interpretar su contenido (a fin de formatearlo y mostrarlo en pantalla). Al encontrar la etiqueta `<APPLET...>` descarga de la

máquina B el fichero denominado `X.class` que corresponde a la clase principal de un applet JAVA. El navegador utiliza la máquina virtual JAVA disponible en la máquina, sobre la que crea una instancia de la clase `X`, a la que lanza una secuencia de mensajes determinados (*ciclo de vida del applet*).

Al visualizar el documento en pantalla, el navegador debe reservar la zona de visualización del applet. En el ejemplo, se trata de una zona de 700 píxels de anchura y 400 de altura. Dicha zona forma parte del documento que se visualiza (la ubicación exacta dentro del mismo depende de los restantes contenidos del fichero).

Así pues, los atributos `CODE`, `HEIGHT` y `WIDTH` son los mínimos y necesarios para definir una etiqueta `APPLET`, ya que el navegador necesita conocer tanto la clase a ejecutar como el tamaño de la zona de visualización.

Mientras el navegador muestre el documento en pantalla, el usuario puede interactuar con el applet mediante eventos sobre el área de interacción, y el applet utiliza el área de interacción para mostrar cualquier información pertinente al usuario. El applet termina cuando el navegador descarta el documento actual.

Un applet se implementa como una extensión de la clase `java.applet.Applet`, si queremos trabajar con AWT, o, en el caso de emplear Swing, como extensión de la clase `javax.swing.JApplet`, que es una subclase de `java.applet.Applet`. A su vez, la clase `java.applet.Applet` extiende a `java.awt.Panel`, que extiende a `java.awt.Container` y éste a `java.awt.Component`. El resultado es que las clases `Applet` y `JApplet` heredan los métodos de dibujo (primitivas gráficas, componentes de interfaz, etc.) y gestión de eventos definidos en esas clases. Cuando se indica que la clase principal de un programa extiende `Applet` o `JApplet`, dicho programa puede trabajar en modo gráfico y ser dirigido por eventos. Como la biblioteca Swing es más avanzada que AWT, a partir de ahora sólo se empleará la clase `JApplet` para implementar los applets.

La respuesta por defecto de la clase `JApplet` ante los eventos de usuario y los mensajes procedentes del navegador es *no hacer nada*. En consecuencia, la tarea del programador es extender la clase `JApplet` (definir una nueva clase derivada de `JApplet`) para introducir el código necesario ante los eventos o mensajes a los que se desee responder.

Ejemplo: Se desea desarrollar un applet que muestre un saludo en pantalla. Además de editar y compilar el programa `Hola.java`, se debe crear un documento HTML (`hola.html`) que incluya una etiqueta específica para lanzar la ejecución del applet.

Código del fichero `Hola.java`:

```
import java.awt.Graphics;
import javax.swing.JApplet;

public class Hola extends JApplet {
  public void paint (Graphics g) {
    g.drawString("Hola", 20, 20);
  }
}
```

Código del fichero `hola.html`:

```
<HTML>
<HEAD><TITLE>Saludo</TITLE></HEAD>
<BODY>
<APPLET CODE="Hola.class" WIDTH="100" HEIGHT="50"></APPLET>
</BODY>
</HTML>
```

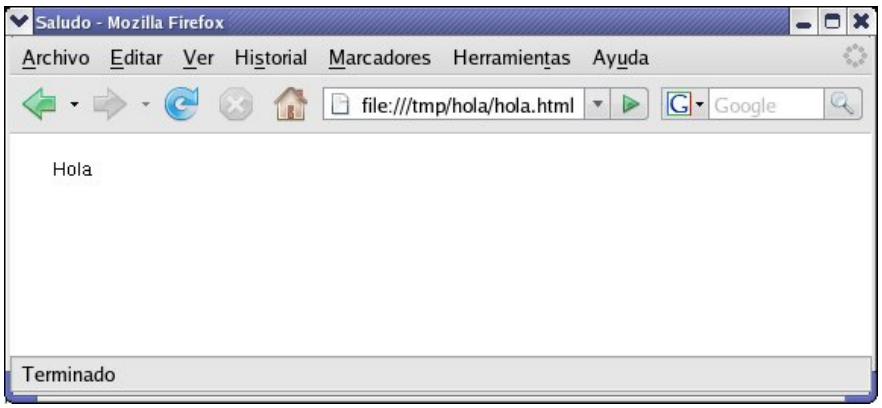

Figura 6.1: Aspecto de un applet en un navegador

Al igual que cualquier otro programa JAVA, el fichero `.java` contiene sentencias de importación (en este ejemplo `import java.awt.Graphics`, donde se definen las primitivas gráficas; e `import javax.swing.JApplet`, clase base para crear los applets en Swing), y una secuencia formada por una o más clases, de las cuales, una es la principal (en este ejemplo sólo hay una, y se denomina `Hola`). La diferencia estriba en que en un applet la clase principal es una extensión de la clase `JApplet`.

Para ejecutar el applet, el navegador crea una instancia de la clase principal (en el ejemplo, de la clase `Hola`) y provoca la invocación de distintos métodos del applet en respuesta a distintas situaciones (por ejemplo, inicio o fin de la ejecución, el área de interacción ha quedado oculta, etc). Todo applet debe ajustarse a un marco estricto que define su ciclo de vida (mecanismos para iniciar, parar, reanudar y terminar un applet).

Cuando el navegador ha cargado los *bytecodes* correspondientes a la clase principal (la extensión de `JApplet`), crea una instancia de la misma sobre la que invoca el método `init()`. El método `init()` es el punto de entrada al applet y normalmente se utiliza para inicializar las variables necesarias y dibujar los componentes de interacción. En este caso no se necesita código alguno en `init()`, ya que no existen variables en la clase, ni se utilizan componentes de interacción, por lo que no se sobreescribe la versión por defecto proporcionada por la clase `JApplet`.

El navegador invoca el método `paint()` del applet cada vez que necesita redibujar el área de interacción (por ejemplo, al visualizar el documento por vez primera, o cada vez que la zona vuelve a ser visible después de haber permanecido oculta –por ejemplo, estaba tras otra ventana, la ventana del navegador estaba minimizada, etc.–). En el ejemplo, `paint()` dibuja en pantalla el saludo, para lo cual utiliza el método `drawString()` definido en la clase `Graphics`. El parámetro de `paint()` es de tipo `Graphics`, y corresponde al contexto gráfico asociado al área de interacción del applet, por lo que los métodos invocados sobre dicho parámetro afectan al área de interacción. El aspecto visual de este applet en un navegador es el que se muestra en la figura 6.1.

6.2.2. Ciclo de vida de un applet

El navegador controla el comportamiento del applet invocando cinco métodos estándar del mismo: `paint()`, `init()`, `start()`, `stop()` y `destroy()`. El método `paint()` se encarga, como ya hemos visto, de redibujar el applet. Los otros cuatro métodos definen el ciclo de

vida del applet. En la tabla 6.1 se definen las acciones que deben realizar los métodos y cuándo son invocados.

Mensaje	**Evento(s) del navegador**	**Comportamiento que se debe definir en el applet**
`init()`	Carga documento HTML con etiqueta `<APPLET>`, y la clase principal correspondiente.	Código de inicialización (inicialización de los atributos, añadir componentes de interacción, etc.).
`start()`	Inmediatamente después de `init()`. Cada vez que se vuelva a visitar el applet con el navegador.	Iniciar animaciones y otras tareas.
`paint()`	Cada vez que se detecta la necesidad de *pintar* el área de pantalla del applet.	Redibujar lo necesario en el área de interacción.
`stop()`	Cuando la página web que contiene al applet se reemplaza por otra. Inmediatamente antes de invocar a `destroy()`.	Suspender animaciones y otras tareas (para no consumir recursos innecesariamente).
`destroy()`	Termina el applet (cuando se descarta el documento o se abandona el navegador).	Limpieza de los recursos del sistema.

Tabla 6.1: Ciclo de vida de un applet

Cuando se diseña un applet, se deben sobreescribir únicamente aquellos métodos que se desea realicen alguna tarea (por ejemplo, `Hola` no debe responder a los mensajes que afectan al ciclo de vida, por lo que no se sobreescribe ninguno de los métodos restantes `init()`, `start()`, `stop()` o `destroy()`).

6.3. Gráficos

JAVA proporciona distintas primitivas para dibujar figuras, mostrar texto, gestionar colores, etc.

6.3.1. Sistema de coordenadas

Se utiliza el sistema de coordenadas estándar bidimensional común a la mayor parte de las interfaces gráficas. La esquina superior izquierda de la zona de visualización se considera la posición (0,0). Los valores de la componente x crecen hacia la derecha, mientras que los valores de la componente y crecen hacia abajo.

6.3.2. Figuras

Toda acción de dibujo de figuras (líneas, rectángulos, óvalos, polígonos y sus rellenos) se lleva a cabo a través de la clase `Graphics` (o *contexto gráfico*), que define los distintos métodos para dibujar figuras. Cada objeto de tipo `Graphics` posee su propio origen de coordenadas, colores de dibujo y fondo, etc.

El código de dibujo debe aparecer en el método `paint(Graphics g)` del applet, de forma que se pueda reconstruir el contenido dibujado en el applet cuando lo solicita el navegador.

El parámetro `g` hace referencia a un contexto gráfico preexistente (creado por el entorno y asociado a la zona de interacción del applet) sobre el que se pueden aplicar las operaciones de dibujo.

Las diferentes operaciones de dibujo definidas en `Graphics` son las que se muestran en la tabla 6.2. Se asume que `x`, `y`, `ancho`, `alto`, `x1`, `y1`, `x2`, `y2`, `angIni` y `angArc` son enteros, y `p` un objeto de tipo `Polygon` creado a partir de dos vectores de enteros para las coordenadas `x` e `y`, y el número de puntos a dibujar.

Acción	**Método**
Dibujar línea	`drawLine(x1, y1, x2, y2)`
Dibujar rectángulo	`drawRect(x, y, ancho, alto)`
Dibujar rectángulo relleno	`fillRect(x, y, ancho, alto)`
Borrar rectángulo	`clearRect(x, y, ancho, alto)`
Dibujar óvalo	`drawOval(x, y, ancho, alto)`
Dibujar óvalo relleno	`fillOval(x, y, ancho, alto)`
Dibujar arco	`drawArc(x, y, ancho, alto, angIni, angArc)`
Dibujar arco relleno	`fillArc(x, y, ancho, alto, angIni, angArc)`
Dibujar polígono	`drawPolygon(p)`

Tabla 6.2: Operaciones de dibujo en `Graphics`

Ejemplos de utilización de los métodos de la tabla 6.2 son los siguientes (`g` es un objeto de tipo `Graphics`):

```
g.drawLine(0, 0, 99, 99);
g.drawRect(20, 20, 20, 20);
g.fillRect(40, 40, 15, 25);
g.clearRect(30, 30, 5, 5);
g.drawOval(25, 0, 50, 25);
g.fillOval(25, 75, 50, 25);
g.drawArc(57, 37, 30, 20, 45, 90);
g.fillArc(57, 42, 30, 20, 45, 90);
int[] x = {25, 40, 10}, y = {60, 70, 90};
Polygon p = new Polygon(x, y, 3);
g.drawPolygon(p);
```

6.3.3. Color

JAVA define la clase `Color`, que declara diferentes constantes estáticas que corresponden a otros tantos colores predefinidos: `Color.black`, `Color.white`, `Color.red`, `Color.blue`, `Color.green`, etc.). Además, la clase `Color` permite construir cualquier otro color especificando en el constructor sus valores **RGB**.

Una vez definidos los colores, es posible especificar el color de dibujo y el color de fondo (`setForeground(color)` y `setBackground(color)`, respectivamente). El color de dibujo se utiliza en las primitivas gráficas (las descritas en el apartado anterior y la correspondiente al texto, que se introduce en el apartado siguiente). El color de fondo corresponde al fondo del componente. También se definen primitivas para obtener los colores de dibujo y fondo (`getForeground()` y `getBackground()`, respectivamente).

```
setForeground(Color.red);
g.drawLine(0, 0, 99, 99);  // se pinta en rojo
```

```
g.drawOval(25, 0, 50, 25); // se pinta en rojo
setForeground(new Color(0, 0, 255));
// a partir de aquí se pintará en el color recién definido (azul)
```

6.3.4. Texto

La primitiva básica de `Graphics` para mostrar texto en la zona de interacción del applet es `drawString(texto, x, y)` (similar a los otros métodos de dibujo de figuras). Pero, además, se pueden controlar aspectos tales como el tipo de letra que se utiliza para mostrar el mensaje, o el tamaño o atributos del mismo.

La clase `Font` describe tipos de letra. Se puede construir un tipo de letra a partir de la denominación (por ejemplo, `Serif`, `Helvetica`, `Courier`), atributo (la clase `Font` define las constantes `ITALIC`, `BOLD`, `PLAIN`) y tamaño (por ejemplo, 12, 14, 18). Todo componente JAVA (por ejemplo, un applet) define el método `setFont(tipoLetra)`, mediante el que se puede asociar un tipo determinado de letra a dicho componente. La clase `Graphics` también define este método para el dibujado de cadenas de caracteres.

```
import java.awt.*;
import javax.swing.*;

public class PruebaTexto extends JApplet {
  public void paint (Graphics g) {
    g.drawString("Mensaje de prueba", 10, 10);
    Font f = new Font("Courier", Font.BOLD, 14);
    g.setFont(f);
    g.drawString("Mensaje de prueba", 10, 50);
  }
}
```

6.3.5. Imágenes y sonido

La clase `Applet` tiene métodos que se encargan de cargar ficheros de imagen (formatos **gif**, **jpg**, **png**) y sonido (formatos **mid**, **au**) a partir de una URL. En ambos casos, si el fichero se encuentra en la misma ubicación que el documento HTML, se puede utilizar el método `getDocumentBase()`, que proporciona acceso a la URL desde la que se ha descargado el documento.

El resultado de `getImage(url, fichero)` es de tipo `Image`. Cada vez que se desea mostrar la imagen (por ejemplo, en `paint()`) se invoca uno de los métodos `drawImage()` de `Graphics`.

Por su parte, el resultado de `getAudioClip(url, fichero)` es de tipo `AudioClip`, que ofrece los métodos `loop()`, `play()` y `stop()`.

```
import java.applet.*;
import java.awt.*;
import javax.swing.*;

public class Multimedia extends JApplet {

  private Image imagen;
  private AudioClip audio;

  public void init () {
```

```
      imagen = getImage(getDocumentBase(), "dibujo.jgp");
      audio = getAudioClip(getDocumentBase(), "sonido.au");
    }

    public void start () {
      audio.play();
    }

    public void stop () {
      audio.stop();
    }

    public void paint (Graphics g) {
      g.drawImage(imagen, 0, 0, this);
    }

  }
```

6.4. Componentes de interfaz de usuario

La elaboración de interfaces de usuario se va a ver tremendamente simplificada gracias a la programación orientada a objetos, ya que esta proporciona los elementos básicos necesarios para construir interfaces y sólo es necesario reutilizar dicho código (ya sea mediante instanciación o mediante herencia). De esta forma, la programación orientada a objetos permite crear fácilmente interfaces gráficas complejas.

Swing proporciona distintos componentes de interfaz (botones, campos editables, etc.) que permiten la interacción entre usuario y applet. Dichos componentes poseen una representación gráfica, un conjunto de atributos y un conjunto de métodos. Cada componente puede responder a distintos eventos de usuario (por ejemplo, hacer click en un botón). Para organizar dichos componentes se emplean los contenedores. Todo componente de interfaz debe incluirse en un contenedor o en el propio applet, que lo añadirá a su `contentPane` (todo contenedor dispone del método `add()` para añadir componentes). Finalmente, pueden emplearse gestores de ubicación para controlar la posición de los componentes dentro del contenedor.

6.4.1. Componentes principales

Etiqueta (`JLabel`) Es una cadena de caracteres que se visualiza en pantalla. Puede justificarse a la derecha, izquierda (por defecto), o centrada.

```
JLabel etiqueta1 = new JLabel("Etiqueta justificada a la izquierda");
JLabel etiqueta2 = new JLabel("Etiqueta justificada a la derecha",
                              JLabel.RIGHT);
```

Ya hemos visto anteriormente que la clase `Graphics` permite dibujar cadenas de texto. Las diferencias entre una etiqueta creada mediante `drawString()` y otra creada mediante un `JLabel` son las siguientes:

- la dibujada debe especificar las coordenadas dentro de la zona de visualización, mientras que la localización de un `JLabel` se maneja automáticamente con el gestor de ubicación asociado al contenedor al que se añade.

- la generada con `drawString()` debe redibujarse explícitamente en el método `paint()`, mientras que un `JLabel` se redibuja automáticamente.
- un `JLabel` dispone, además, de métodos diversos (por ejemplo, para asociar un icono a la etiqueta).

Botón (`JButton`) Es un recuadro con etiqueta que responde a pulsaciones de ratón o de la tecla **Intro** si tiene el foco.

```
JButton boton = new JButton("Un botón");
```

Campo de edición de texto (`JTextField`) Permite visualizar una cadena editable de caracteres en una única línea. Durante la construcción se puede indicar la anchura del campo y una cadena inicial. Si el contenido del campo es mayor que el área visible se puede desplazar el texto a derecha e izquierda.

```
// Un campo de texto vacío
JTextField ct1 = new JTextField();
// Un campo de texto vacío de 20 columnas
JTextField ct2 = new JTextField(20);
// Un campo de texto con contenido específico
JTextField ct3 = new JTextField("Un campo de texto");
// Con contenido y número de columnas
JTextField ct4 = new JTextField("Un campo de texto", 20);
```

Como métodos más usados, se pueden destacar los que permiten obtener y modificar el texto contenido en el campo:

```
String leer = campoTexto.getText();
campoTexto.setText("Contenido modificado");
```

Área de edición de texto (`JTextArea`) Es un campo compuesto por varias líneas (permite visualizar una cadena de caracteres única que incluye caracteres de salto de línea). Durante la construcción se fijan las dimensiones iniciales (altura y anchura) del campo. Si el texto que contiene ocupa más que dicho tamaño inicial, el área aumenta su tamaño. Si lo que queremos es que mantenga el tamaño inicial y se cree una barra de desplazamiento (scroll) de ser necesario, el componente deberá mostrarse dentro de un `JScrollPane`.

```
// Un área de texto vacía
JTextArea at1 = new JTextArea();
// Vacía pero con tamaño especificado
JTextArea at2 = new JTextArea(3, 10);
// Con contenido específico
JTextArea at3 = new JTextArea("cadena de caracteres que\n" +
                              "ocupa varias líneas");
// Con contenido y tamaño específicos
JTextArea at4 = new JTextArea("cadena de caracteres que\n" +
                              "ocupa varias líneas", 3, 10);
```

Igual que en el caso del `JTextField`, `JTextArea` dispone de los métodos `getText()` y `setText()`, con los que se obtiene y modifica el contenido del área de texto.

Lienzo (`JPanel`) Aunque se trata de un contenedor genérico (como veremos en la sección 6.5.1), un `JPanel` también puede emplearse como base para crear lienzos en los que dibujar figuras y texto. Para ello, se debe crear una clase que herede de `JPanel` y que sobreescriba el método `paintComponent()` para que contenga el código personalizado de pintado (ver sección 6.9). De esta forma, y a diferencia del resto de componentes mencionados, el código del `JPanel` se reutiliza en nuestras interfaces para crear nuevos lienzos de dibujo mediante la herencia y sobreescritura (en vez de mediante la instanciación directa, como en ejemplos anteriores).

```
class ZonaDibujo extends JPanel {
  public ZonaDibujo () {
    // modificamos el tamaño del panel
    setPreferredSize(new Dimension(100, 100));
  }

  public void paintComponent (Graphics g) {
    // heredamos el pintado por defecto del panel
    super.paintComponent(g);
    // realizamos nuestro propio pintado
    g.drawRect(0, 0, 99, 99); // dibuja un borde alrededor del panel
  }
}
```

Elección (`JComboBox`) Son menús desplegables (*pop-up*) cuya etiqueta es la entrada actualmente seleccionada en el menú. Cuando se selecciona una entrada genera un evento que indica la entrada seleccionada. Se pueden hacer editables, de forma que el usuario puede introducir un nuevo valor.

```
String[] colores = {"Rojo", "Verde", "Azul"};
JComboBox eleccion = new JComboBox(colores);
```

Dispone de los métodos `getSelectedIndex()` y `getSelectedItem()` que permiten obtener, respectivamente, el índice actualmente seleccionado y el elemento correspondiente.

Conmutador (`JCheckBox` y `JRadioButton`) Es una etiqueta con una pequeña zona sensible a pulsaciones de ratón y que posee dos estados: cierto (marcado) y falso. Cada pulsación conmuta el estado (por defecto está a falso).

```
JCheckBox conmutador1 = new JCheckBox("Un conmutador");
JRadioButton conmutador2 = new JRadioButton("Otro conmutador");
```

Ambos componentes poseen el método `isSelected()` que devuelve el estado del conmutador: `true` si está marcado y `false` en otro caso.

Grupo de conmutadores (`ButtonGroup`) Permite controlar el comportamiento de un grupo de conmutadores, de forma que sólo uno de ellos puede estar a cierto en cada instante (al marcar uno se garantiza que los demás quedan a falso).

```
ButtonGroup grupoConmutadores1 = new ButtonGroup();
JCheckBox conmutadorGrupo1 = new JCheckBox("Opción 1");
JCheckBox conmutadorGrupo2 = new JCheckBox("Opción 2");
// los conmutadores se deben añadir al ButtonGroup
```

```
grupoConmutadores1.add(conmutadorGrupo1);
grupoConmutadores1.add(conmutadorGrupo2);

ButtonGroup grupoConmutadores2 = new ButtonGroup();
JRadioButton conmutadorGrupo3 = new JRadioButton("Opción 3");
JRadioButton conmutadorGrupo4 = new JRadioButton("Opción 4");
grupoConmutadores2.add(conmutadorGrupo3);
grupoConmutadores2.add(conmutadorGrupo4);
```

Lista (`JList`) Es una lista de opciones entre las cuales, dependiendo de un atributo, podemos seleccionar una (al seleccionar una se deselecciona la previamente seleccionada) o varias (se pueden mantener tantas entradas seleccionadas como se desee).

El método `getSelectedValue()` devuelve el ítem seleccionado y `getSelectedIndex()`, el índice correspondiente a dicho ítem (el primer ítem ocupa la posición 0). Cuando se permite selección múltiple se deben utilizar `getSelectedValues()` y `getSelectedIndices()` respectivamente (devuelven como respuesta un vector en lugar de un valor único).

```
String[] ciudades = {"Murcia", "Castellón", "Valencia",
                     "Alicante", "Barcelona", "Tarragona"};
JList lista = new JList(ciudades);
// sólo se permite seleccionar un único elemento
lista.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

Deslizador (`JSlider`) Es un control deslizante horizontal o vertical que puede tomar un valor entero entre dos límites (mínimo y máximo) definidos en la construcción.

```
JSlider slider = new JSlider (JSlider.HORIZONTAL, 0, 10, 3);
```

Para obtener el valor actual del control, se dispone del método `getValue()`.

Menú (`JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem`, `JRadioButtonMenuItem`) El lenguaje JAVA dispone de mecanismos para añadir menús a una aplicación gráfica (`JApplet`, `JDialog`, `JFrame`...). Una aplicación posee un objeto `JMenuBar` (barra de menú), que contiene objetos de tipo `JMenu` compuestos a su vez de objetos `JMenuItem` (pueden ser cadenas de caracteres, menús, conmutadores o separadores).

Para añadir menús a una aplicación gráfica, se deben seguir los siguientes pasos:

1. Crear un objeto de tipo `JMenuBar`.
2. Crear los objetos de tipo `JMenu` necesarios.
3. Crear para cada `JMenu` las entradas necesarias (`JMenuItem`, `JCheckBoxMenuItem`, `JRadioButtonMenuItem` o separadores), y añadirlas al mismo.
4. Crear nuevos `JMenu` para incorporarlos como submenús dentro de otro `JMenu`, de ser necesario.
5. Incorporar los objetos `JMenu` a la `JMenuBar` (en el orden en que deben aparecer de izquierda a derecha).
6. Incorporar la `JMenuBar` a la aplicación gráfica (`JApplet`, por ejemplo) mediante el método `setJMenuBar`.

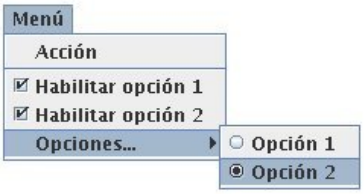

Figura 6.2: Un menú

```
JMenuBar menuBar = new JMenuBar(); // paso 1
JMenu menu = new JMenu("Menú");    // paso 2
menu.add(new JMenuItem("Acción")); // paso 3
menu.addSeparator();
menu.add(new JCheckBoxMenuItem("Habilitar opción 1"));
menu.add(new JCheckBoxMenuItem("Habilitar opción 2"));
JMenu submenu = new JMenu("Opciones..."); // paso 4
ButtonGroup grupo = new ButtonGroup();
JRadioButtonMenuItem radio1 = new JRadioButtonMenuItem("Opción 1");
JRadioButtonMenuItem radio2 = new JRadioButtonMenuItem("Opción 2");
grupo.add(radio1);
grupo.add(radio2);
submenu.add(radio1);
submenu.add(radio2);
menu.add(submenu);    // paso 4
menuBar.add(menu);    // paso 5
setJMenuBar(menuBar); // paso 6
```

El resultado visual del menú del código anterior es el mostrado en la figura 6.2.

6.4.2. Un ejemplo completo

El siguiente código declara todos los componentes estudiados en un mismo `JApplet`, cuya representación visual se muestra en la figura 6.3.

```
import java.awt.*;
import javax.swing.*;

class MiPanelDeDibujo extends JPanel {
  public MiPanelDeDibujo () {
    setPreferredSize(new Dimension(100, 100));
    setBackground(Color.white);
  }
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    g.drawLine(0, 0, 99, 99);
    g.drawRect(20, 20, 20, 20);
    g.fillRect(40, 40, 15, 25);
    g.clearRect(30, 30, 5, 5);
    g.drawOval(25, 0, 50, 25);
    g.fillOval(25, 75, 50, 25);
    g.drawArc(57, 37, 30, 20, 45, 90);
```

```
      g.fillArc(57, 42, 30, 20, 45, 90);
      int[] x = {25, 40, 10}, y = {60, 70, 90};
      Polygon p = new Polygon(x, y, 3);
      g.drawPolygon(p);
    }
  }

  public class EjemploCompleto extends JApplet {

    public void init () {
      // cambiamos el gestor de ubicación por defecto
      setLayout(new FlowLayout());

      // creamos los componentes y los añadimos al applet
      JLabel etiqueta = new JLabel("Una etiqueta", JLabel.RIGHT);
      add(etiqueta);

      JButton boton = new JButton("Un botón");
      add(boton);

      JTextField campoTexto = new JTextField("Un campo de texto", 20);
      add(campoTexto);

      JTextArea areaTexto = new JTextArea("cadena de caracteres que\n" +
                                          "ocupa varias líneas", 3, 10);
      add(areaTexto);

      String[] colores = {"Rojo", "Verde", "Azul"};
      JComboBox eleccion = new JComboBox(colores);
      add(eleccion);

      JCheckBox conmutador1 = new JCheckBox("Un conmutador");
      add(conmutador1);

      JRadioButton conmutador2 = new JRadioButton("Otro conmutador");
      add(conmutador2);

      ButtonGroup grupoConmutadores1 = new ButtonGroup();
      JCheckBox conmutadorGrupo1 = new JCheckBox("Opción 1");
      JCheckBox conmutadorGrupo2 = new JCheckBox("Opción 2");
      grupoConmutadores1.add(conmutadorGrupo1);
      grupoConmutadores1.add(conmutadorGrupo2);
      add(conmutadorGrupo1);
      add(conmutadorGrupo2);

      ButtonGroup grupoConmutadores2 = new ButtonGroup();
      JRadioButton conmutadorGrupo3 = new JRadioButton("Opción 3");
      JRadioButton conmutadorGrupo4 = new JRadioButton("Opción 4");
      grupoConmutadores2.add(conmutadorGrupo3);
      grupoConmutadores2.add(conmutadorGrupo4);
      add(conmutadorGrupo3);
      add(conmutadorGrupo4);

      String[] ciudades = {"Murcia", "Castellón", "Valencia",
```

```
                          "Alicante", "Barcelona", "Tarragona"};
        JList lista = new JList(ciudades);
        lista.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        add(lista);

        JSlider slider = new JSlider (JSlider.HORIZONTAL, 0, 10, 3);
        add(slider);

        // creamos el menú y lo añadimos al applet
        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("Menú");
        menu.add(new JMenuItem("Acción"));
        menuBar.add(menu);
        setJMenuBar(menuBar);

        // añadimos el lienzo de dibujo
        add(new MiPanelDeDibujo());
      }

    }
```

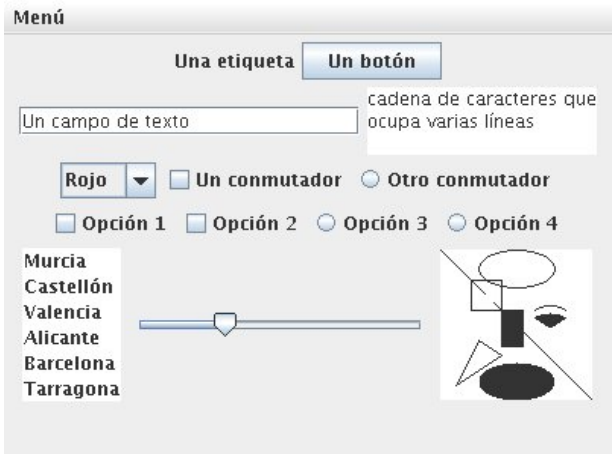

Figura 6.3: Componentes en un `JApplet`

6.5. Contenedores y gestores de ubicación

Los componentes de interfaz no se encuentran aislados, sino agrupados dentro de contenedores. Los contenedores agrupan varios componentes y se encargan de distribuirlos (ubicarlos) de la forma más adecuada (mantienen la funcionalidad de adición, sustracción, recuperación, control y organización de los componentes). Los contenedores son a su vez componentes, lo que permite situar contenedores dentro de otros contenedores. También disponen del código necesario para controlar eventos, cambiar la forma del cursor, modificar el tipo de fuente, etc. Todos los contenedores son instancias de la clase `Container` o una de sus extensiones.

6.5.1. Contenedores

En este libro nos centraremos en un único tipo de contenedor: el panel. De esta forma, todos los componentes los añadiremos a un panel existente. Hay que recordar que todo `JApplet` incorpora por defecto un `contentPane` como panel principal, de tal forma que cuando invocamos el método `add()` sobre un `JApplet` para incorporar un componente, en realidad se está añadiendo dicho componente al `contentPane` por defecto. De la misma forma, y como veremos más adelante, cuando invocamos `setLayout()` sobre un `JApplet`, lo que estamos cambiando es el gestor de ubicación de su `contentPane`. Este comportamiento permite tratar a un `JApplet` como si se tratara de un panel a estos efectos.

A continuación se describen los contenedores proporcionados por Swing.

Panel (`JPanel`) Es un contenedor genérico de componentes. Normalmente no tiene representación visual alguna (únicamente puede ser visualizado cambiando su color de fondo). Suele utilizarse para distribuir los componentes de interfaz de una forma adecuada, asignándole un gestor de ubicación determinado; o como lienzo, para realizar dibujos mediante la sobreescritura de su método `paintComponent()`.

Ventana (`JWindow`) Representa el concepto típico de ventana (zona de pantalla desplazable y redimensionable, con contenido específico, y que no puede incluirse en otro contenedor). Por defecto no posee título ni borde ni barra de menú, pero dispone de métodos específicos para alterar la posición, el tamaño, forma del cursor, etc. Normalmente no se crean objetos de tipo `JWindow`, sino de tipo `JFrame` o `JDialog`.

Marco (`JFrame`) Un `JFrame` es lo que visualmente conocemos como ventana. Posee borde y título y puede incluir una barra de menú. Los objetos de tipo `JFrame` son capaces de generar varios tipos de eventos (por ejemplo, `WindowClosing` cuando el usuario pincha con el ratón sobre el icono *cerrar* en la barra de título del `JFrame`).

Diálogo (`JDialog`) Es una ventana con borde y título que permite recoger entradas del usuario. El diálogo se lanza desde otra ventana, y puede ser modal (todas las entradas del usuario las recoge este diálogo, por lo que no se puede proseguir con la aplicación hasta concluir el diálogo) o no modal (se puede mantener el diálogo abierto e interactuar con diálogo y aplicación simultáneamente), según se indique en el constructor (también puede conmutarse a modal una vez creado). Se puede mover (libremente, sin restringirlo a las dimensiones del padre) y redimensionar, pero no minimizar o maximizar.

6.5.2. Gestores de ubicación

En lugar de especificar las coordenadas de pantalla en las que ubicar los distintos componentes de interfaz, se utilizan gestores de ubicación que distribuyen (ubican) los componentes de un contenedor según una política determinada. Así, cada contenedor tiene asociado un gestor de ubicación que se utiliza para procesar las llamadas a `micontenedor.add()`. El método `micontenedor.setLayout(migestor)` permite asociar un gestor de ubicación a un contenedor. Existen distintos gestores de ubicación, pensados para necesidades diferentes (por ejemplo, para ubicar las teclas de una calculadora se utiliza uno que ubique los elementos según una rejilla).

Todo componente posee un tamaño mínimo (por ejemplo, en un botón el necesario para mostrar la etiqueta) y un tamaño preferido (cuando se extiende un componente se puede sobreescribir el método `getPreferredSize()`, indicando de este modo el tamaño ideal para

ese componente). Los contenedores también pueden indicar el espacio alrededor de su borde interior que debe quedar libre (no se situará ningún componente) mediante el método `getInsets()`. El gestor de ubicación parte de estas indicaciones para cada componente del conjunto de componentes a gestionar y, según la política establecida (por ejemplo, situar en una rejilla), intenta obtener el mejor resultado visual posible.

Existe un conjunto de gestores de ubicación ya definidos y, además, se pueden crear otros propios. Los gestores de ubicación ya definidos más comunes se describen a continuación.

Flujo (`FlowLayout`) Ubica los componentes de izquierda a derecha hasta agotar la línea, y entonces pasa a la línea siguiente. En cada línea, los componentes están centrados. Es el que se utiliza por defecto en `JPanel`. El siguiente código mostraría el aspecto visual capturado en la figura 6.4[1].

```
import java.awt.*;
import javax.swing.*;

public class PruebaFlow extends JApplet {
  public void init () {
    setLayout(new FlowLayout());
    add(new JButton("Aceptar"));
    add(new JButton("Cancelar"));
    add(new JButton("Enviar"));
    add(new JButton("Borrar"));
    add(new JButton("Anterior"));
    add(new JButton("Siguiente"));
  }
}
```

Figura 6.4: Gestor de ubicación por flujo

Bordes (`BorderLayout`) Es el empleado por defecto en `JApplet`. Posee cinco zonas: norte, sur, este, oeste y centro. Cuando se añade un componente se indica su ubicación (por ejemplo, norte). No es necesario cubrir todas las zonas. Una característica importante de este gestor de ubicación es que redimensiona cada componente para que ocupe todo el espacio de la zona en la que se inserta. De esta forma, sólo se visualiza un componente por zona (el último, si se añade más de uno en la misma zona). Para ubicar varios componentes en la misma zona, se debe añadir un contenedor como único componente de la zona y colocar en él los componentes deseados.

El código siguiente ofrece el resultado visual de la figura 6.5.

[1] Otro ejemplo de uso de este gestor de ubicación está en la sección 6.4.2.

```
import java.awt.*;
import javax.swing.*;

public class PruebaBorder extends JApplet {
  public void init () {
    // no sería necesario, ya que es el gestor por defecto
    setLayout(new BorderLayout());
    add(new JLabel("Una cabecera"), BorderLayout.NORTH);
    add(new JTextField("Abajo, un mensaje"), BorderLayout.SOUTH);
    add(new JButton("SALIR"), BorderLayout.WEST);
    add(new JButton("AYUDA"), BorderLayout.EAST);
    add(new JTextArea("Ejemplo de texto en\n" +
                      "la parte central"), BorderLayout.CENTER);
  }
}
```

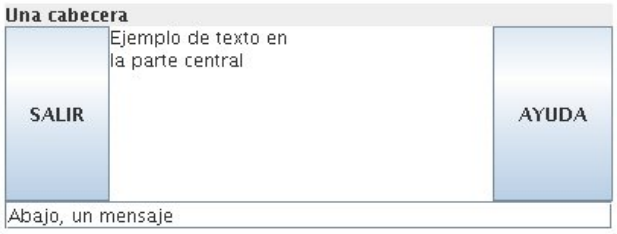

Figura 6.5: Gestor de ubicación por bordes

Rejilla (`GridLayout`) Distribuye los componentes en filas y columnas, proporcionando un aspecto de rejilla, que se rellena por filas. Cuando se proporcionan valores distintos de cero tanto para el número de filas como para el de columnas (ya sea en el constructor o en los métodos `setRows()` y `setColumns()`), el número de columnas especificado se ignora. En su lugar, se emplean tantas columnas como sea necesario según el número de filas y de componentes a ubicar (por ejemplo, si se especifican tres filas y dos columnas pero hay nueve componentes a ubicar, estos se muestran en una cuadrícula de tres filas y tres columnas). El número de columnas especificado sólo se tiene en cuenta cuando el número de filas se pone a cero.

Un código como el siguiente muestra un aspecto visual como el de la figura 6.6.

```
import java.awt.*;
import javax.swing.*;

public class PruebaGrid extends JApplet {
  public void init () {
    setLayout(new GridLayout(3, 2)); // rejilla de 3 filas y 2 columnas
    for (int i = 1; i < 7; i++)
      add(new JButton("" + i));
  }
}
```

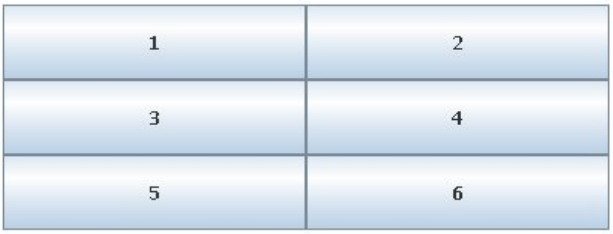

Figura 6.6: Gestor de ubicación en rejilla

Ubicaciones complejas: contenedores anidados o compuestos Como los contenedores también son componentes, pueden anidarse de forma arbitraria para mejorar el control. Así, podemos dividir el espacio visible en distintos paneles y aplicar un gestor de ubicación distinto para cada uno. De esta forma, se puede dividir la interfaz en zonas distintas, ubicando los componentes de cada una de ellas según diferentes políticas, así como respondiendo de forma distinta a los eventos de cada zona. Por ejemplo, si modelamos un teléfono móvil como un teclado y una pantalla, para el teclado interesará utilizar una rejilla pero la ubicación de los símbolos en pantalla requerirá otra estrategia.

```
import java.awt.*;
import javax.swing.*;

public class MultiPanel extends JApplet {

  private String[] teclas = {"1", "2", "3", "4", "5", "6",
                             "7", "8", "9", "*", "0", "#"};

  public void init () {
    // 2 secciones: pantalla y teclado
    setLayout(new GridLayout(2, 0));

    // creamos la pantalla con bordes
    JPanel pantalla = new JPanel();
    pantalla.setLayout(new BorderLayout());
    pantalla.add(new JLabel("movifone", JLabel.CENTER),
                 BorderLayout.NORTH);
    pantalla.add(new JLabel("Menú", JLabel.CENTER),
                 BorderLayout.SOUTH);

    // creamos el teclado con una rejilla
    JPanel teclado = new JPanel();
    teclado.setLayout(new GridLayout(5, 0));
    teclado.add(new JButton("^"));
    teclado.add(new JButton("OK"));
    teclado.add(new JButton("^"));
    for (int i = 0; i < teclas.length; i++) {
      JButton b = new JButton(teclas[i]);
      teclado.add(b);
    }

    // añadimos ambos paneles al applet
```

```
        add(pantalla);
        add(teclado);
    }

}
```

El código anterior muestra el aspecto visual recogido en la figura 6.7.

Figura 6.7: Gestores de ubicación compuestos

Finalmente, JAVA proporciona otros gestores de ubicación más especializados, como `BoxLayout`, `CardLayout`, `GridBagLayout`, `OverlayLayout`, etc., que no se abordarán en este libro.

6.6. Programación dirigida por eventos

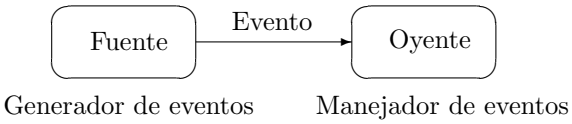

Figura 6.8: La arquitectura Fuente-Oyente

La arquitectura Fuente-Oyente (figura 6.8) define un esquema en el que un elemento denominado *fuente* genera un *evento* en un determinado instante de tiempo. Dicho evento puede ser tratado o *manejado* por otro elemento, denominado *oyente*. Para que el evento viaje de la fuente al oyente, el oyente debe haberse registrado como tal previamente en la fuente. De esta forma, la responsabilidad de la fuente se limita a generar los eventos y enviarlos a los oyentes que tenga registrados en cada momento[2]. Dicho envío se realiza en forma de invocación de un método concreto en cada oyente. Para asegurar que el oyente dispone de dicho método, se requiere la implementación de una cierta interfaz en el momento en el que el oyente se registra como tal.

Siguiendo esta arquitectura, los componentes de interacción de un applet se comportan como fuentes de eventos, generando eventos en respuesta a diversas acciones del usuario. Por

[2]En caso de que no haya ningún oyente registrado, no se llevará a cabo ninguna acción: el evento se perderá.

ejemplo, un botón genera un evento cada vez que el usuario lo pulsa. En este caso, el evento generado es la pulsación y la fuente del evento es el botón (la fuente no es el ratón ni el usuario, sino el componente de interacción). Adicionalmente, diversos elementos se pueden registrar como oyentes en las fuentes que sean de su interés, con el fin de ser invocados cada vez que se genera un evento, para realizar las acciones pertinentes. Este esquema de programación se conoce con el nombre de *programación dirigida por eventos*.

Siguiendo un esquema parecido, y como ya vimos al principio del tema, el navegador puede generar eventos de repintado cuando detecta que es necesario actualizar (repintar) la zona de interacción del applet (por ejemplo, cuando dicha zona vuelve a quedar visible después de haber permanecido parcialmente cubierta por otra ventana, o se recarga el documento HTML, o se ha minimizado y vuelto a maximizar la ventana del navegador, etc.). La gestión de dicha zona es responsabilidad exclusiva del applet, con lo que el applet se convierte en oyente de los eventos de repintado. La diferencia con la arquitectura general es que, para recibir eventos de repintado, el applet no necesita registrarse como oyente específicamente en ninguna fuente.

6.6.1. Eventos

Existen muchos tipos de eventos distintos. Clases de eventos diferentes se representan mediante distintas clases JAVA, todas ellas subclase de `java.util.EventObject`. En este texto analizamos los eventos generados por el usuario (ratón y teclado, y acción sobre los distintos componentes de interacción). Todos los eventos AWT derivan de la clase `java.awt.AWTEvent` (subclase de `java.util.EventObject`), y se incluyen en el paquete `java.awt.event`.

Cada evento es generado por un objeto fuente (puede obtenerse mediante `getSource()`) y posee un tipo (puede obtenerse con `getID()`). El tipo se utiliza para distinguir los distintos tipos de eventos representados por la misma clase de evento (por ejemplo, la clase `MouseEvent` define, entre otros, `MOUSE_CLICKED` y `MOUSE_DRAGGED`). Cada clase de evento posee el conjunto de métodos adecuado para los eventos que representa (por ejemplo, sobre los objetos de tipo `MouseEvent` se pueden invocar métodos tales como `getX()`, `getY()` y `getClickCount()`).

6.6.2. Fuente y oyente de eventos

Un generador o fuente de eventos mantiene una lista de oyentes interesados en recibir una notificación cuando se produzca el evento. Además, la fuente de eventos proporciona métodos que permiten a un oyente incorporarse a la lista de oyentes, o abandonarla. Así, si una fuente de eventos genera eventos de tipo `X`, posee un método denominado `addXListener()` (el oyente lo invoca para incorporarse a la lista de oyentes) y otro denominado `removeXListener()` (para eliminarse de esa lista).

Cuando el objeto fuente de eventos genera un evento (o cuando se produce un evento de usuario sobre ese objeto fuente), notifica tal evento a todos los objetos oyentes. Para ello, invoca un método específico sobre cada oyente, pasando como parámetro el objeto de tipo evento. La tabla 6.3 muestra los principales eventos generados por los distintos componentes de interacción.

Según el tipo de evento, el método invocado en el oyente será distinto. Para que este esquema funcione, todos los oyentes deben implementar el método adecuado. Este punto se garantiza obligando a que todos los oyentes que deseen ser notificados ante un evento concreto implementen una determinada interfaz (por ejemplo, los objetos que desean notificación de `ActionEvent` deben implementar la interfaz `ActionListener`. El paquete `java.awt.event`

Componente	**Eventos**	**Acción**
`JButton`	`ActionEvent`	Botón pulsado.
`JCheckBox`	`ItemEvent`	Ítem seleccionado/deseleccionado.
`JCheckBoxMenuItem`	`ItemEvent`	Ítem seleccionado/deseleccionado.
`JComboBox`	`ItemEvent`	Ítem seleccionado/deseleccionado.

`JComponent`	`ComponentEvent`
	`FocusEvent`
	`KeyEvent`
	`MouseEvent`
		Componente desplazado, redimensionado, ocultado o mostrado.
		El componente ha ganado/perdido el foco.
		Pulsación/liberación de tecla.
		Pulsación/liberación de botón, entrada/salida de componente, desplazamiento/arrastre.

`Container`	`ContainerEvent`	Incorporación/eliminación de un componente al contenedor.
`JList`	`ListSelectionEvent`	Ítem seleccionado/deseleccionado.
`JMenuItem`	`ActionEvent`	Ítem seleccionado.
`JRadioButton`	`ActionEvent`	Ítem seleccionado.
`JRadioButtonMenuItem`	`ActionEvent`	Ítem seleccionado.
`JSlider`	`ChangeEvent`	Desplazamiento del deslizador.
`JTextField`	`ActionEvent`	El texto ha terminado de ser editado.
`JWindow`	`WindowEvent`	Abrir, cerrar, minimizar, maximizar la ventana.

Tabla 6.3: Eventos generados por componentes de interacción

define una interfaz de oyente para cada tipo de evento que define (excepto para los eventos de ratón, sobre los que define dos oyentes, `MouseListener` y `MouseMotionListener`).

Cada interfaz de oyente puede definir varios métodos (por ejemplo, `MouseEvent` representa eventos tales como pulsar el boton del ratón, liberarlo, desplazar el ratón, etc., y cada uno de estos eventos supone invocar un método distinto). Todos estos métodos reciben como único parámetro el objeto evento, por lo que éste debe contener toda la información relativa al evento que el programa pueda necesitar. La tabla 6.4 recoge, para las principales clases de eventos, las interfaces que los oyentes deben implementar y los métodos de cada interfaz.

Clase de Evento	**Interfaz de Oyente**	**Métodos**
`ActionEvent`	`ActionListener`	`actionPerformed`
`ChangeEvent`	`ChangeListener`	`stateChanged`

`ComponentEvent`	`ComponentListener`	`componentHidden`
		`componentMoved`
		`componentResized`
		`componentShown`

`ContainerEvent`	`ContainerListener`	`componentAdded` `componentRemoved`
`FocusEvent`	`FocusListener`	`focusGained` `focusLost`
`ItemEvent`	`ItemListener`	`itemStateChanged`

`KeyEvent`	`KeyListener`	`keyPressed`
		`keyReleased`
		`keyTyped`

`ListSelectionEvent`	`ListSelectionListener`	`valueChanged`

`MouseEvent`	`MouseListener`	`mouseClicked`
		`mouseEntered`
		`mouseExited`
		`mousePressed`
		`mouseReleased`

	`MouseMotionListener`	`mouseDragged` `mouseMoved`

`WindowEvent`	`WindowListener`	`windowActivated`
		`windowClosed`
		`windowClosing`
		`windowDeactivated`
		`windowDeiconified`
		`windowIconified`
		`windowOpened`

Tabla 6.4: Eventos e interfaces de oyentes

Una vez creada la clase que implementa la correspondiente interfaz, se debe crear una instancia (objeto oyente) y registrarla en la fuente de eventos (siempre será algún componente de interacción), mediante el correspondiente método `addXListener()`.

6.6.3. Escuchando eventos

Cómo escuchar eventos

Para que un elemento esté en disposición de escuchar eventos, debe implementar la interfaz oyente correspondiente bien directamente o bien a través de la herencia a partir de una clase adaptadora.

Mediante la implementación de la interfaz oyente Cualquier clase que desee actuar como oyente del evento `XEvent`[3] debe implementar la interfaz `XListener`. Si dicha interfaz define únicamente un método, la tarea es simple. Si, por el contrario, define varios métodos, de los cuales sólo interesa responder a uno o dos, seguimos en la obligación de indicar el código de todos los métodos, aunque el código sea *no hacer nada*. Supongamos, por ejemplo, una clase que desea responder al evento `MOUSE_CLICKED` (pulsación de ratón):

```
class OyenteClickRaton implements MouseListener {
  public void mouseClicked (MouseEvent e) {
    responderEvento(e);
  }
  // para los restantes eventos no se hace nada
  public void mouseEntered (MouseEvent e) { }
  public void mouseExited (MouseEvent e) { }
  public void mousePressed (MouseEvent e) { }
  public void mouseReleased(MouseEvent e) { }
}
```

Mediante adaptadores Resulta bastante tedioso incorporar todo el conjunto de métodos que no hacen nada. Para simplificar la tarea, `java.awt.event` proporciona para cada clase de evento que posee más de un método una clase *adaptador*, que implementa la correspondiente interfaz simplemente con código vacío en los distintos métodos (para la clase `XEvent` se dispone de la interfaz `XListener` y el adaptador `XAdapter`). De esta forma, la clase oyente simplemente hereda del adaptador y sobreescribe los métodos correspondientes a los eventos que quiere tratar (para los restantes eventos ya se hereda del adaptador el comportamiento deseado de *no hacer nada*). El ejemplo anterior puede escribirse como:

```
class OyenteClickRaton extends MouseAdapter {
  public void mouseClicked (MouseEvent e) {
    responderEvento(e);
  }
}
```

Dónde escuchar eventos

Existen diversos modos de implementar la escucha de eventos: a) que el propio applet actúe como oyente; b) implementar el oyente en una clase independiente; c) implementar el oyente en una clase anidada; y d) mediante clases anónimas.

Implementación en el propio `JApplet` El siguiente ejemplo ilustra la estructura de un programa dirigido por eventos en el que el propio applet actúa como oyente de eventos de ratón, pero sólo está interesado en los eventos de presionar el ratón y de arrastrarlo:

[3] Donde `X` debe sustituirse por alguno de los diversos tipos de eventos: `Action`, `Mouse`, `Text`...

```
import java.awt.event.*;
import javax.swing.*;

public class Dibujo extends JApplet
  implements MouseListener, MouseMotionListener {

  private int x0, y0; // última coordenada

  public void init () {
    // el propio applet (this) es el oyente
    addMouseListener(this);
    addMouseMotionListener(this);
  }

  // método de la interfaz MouseListener
  public void mousePressed (MouseEvent e) {
    x0 = e.getX();
    y0 = e.getY();
  }

  // método de la interfaz MouseMotionListener
  public void mouseDragged (MouseEvent e) {
    int x = e.getX(), y = e.getY();
    // recta desde punto anterior
    getGraphics().drawLine(x0, y0, x, y);
    x0 = x;
    y0 = y;
  }

  // los otros métodos de la interfaz MouseListener
  public void mouseReleased (MouseEvent e) { }
  public void mouseClicked (MouseEvent e) { }
  public void mouseEntered (MouseEvent e) { }
  public void mouseExited (MouseEvent e) { }

  // el otro método de la interfaz MouseMotionListener
  public void mouseMoved (MouseEvent e) { }

}
```

El ejemplo anterior no puede explotar el uso de `MouseListenerAdapter` y `MouseAdapter` porque la clase principal ya extiende `JApplet` (JAVA no permite herencia múltiple). Además, resulta que el mismo `JApplet` se está comportando como fuente de eventos (ya que es un componente) y oyente (escucha eventos de ratón) a la vez, lo cual puede resultar confuso. Lo ideal es disponer de clases separadas para actuar como fuente y oyente.

Implementación mediante clases independientes Otra forma de realizar la escucha de eventos, más aconsejable que la anterior, es utilizando clases independientes (externas al applet) que extiendan el correspondiente adaptador o implementen la correspondiente interfaz. Hay que tener en cuenta que, al ser código externo al applet, necesitaremos proporcionarles una referencia al mismo para que puedan acceder a sus métodos de ser necesario. Dicha referencia al applet suele pasarse como parámetro al constructor de la clase oyente.

```
import java.awt.event.*;
import javax.swing.*;

class OyenteRaton extends MouseAdapter {

  private Dibujo elApplet;

  public OyenteRaton (Dibujo elApplet) {
    this.elApplet = elApplet;
  }

  public void mousePressed (MouseEvent e) {
    elApplet.setX0(e.getX());
    elApplet.setY0(e.getY());
  }

}

class OyenteMovRaton extends MouseMotionAdapter {

  private Dibujo elApplet;

  public OyenteMovRaton (Dibujo elApplet) {
    this.elApplet = elApplet;
  }

  public void mouseDragged (MouseEvent e) {
    int x = e.getX(), y = e.getY();
    // recta desde punto anterior
    elApplet.getGraphics().drawLine(elApplet.getX0(),
                                    elApplet.getY0(),
                                    x, y);
    elApplet.setX0(x);
    elApplet.setY0(y);
  }

}

public class Dibujo extends JApplet {

  private int x0, y0; // última coordenada

  public void init () {
    addMouseListener(new OyenteRaton(this));
    addMouseMotionListener(new OyenteMovRaton(this));
  }

  public int getX0 () {
    return x0;
  }

  public int getY0 () {
    return y0;
  }
```

```
    public void setX0 (int x) {
      x0 = x;
    }

    public void setY0 (int y) {
      y0 = y;
    }

  }
```

Implementación mediante clases anidadas En el apartado anterior, se han tenido que implementar métodos adicionales (`getX0()`, `getY0()`, `setX0()`, `setY0()`)[4] para acceder de forma controlada a los atributos desde las clases oyente externas. Esto puede resultar incómodo si dichos métodos no se necesitaban previamente. Además, el constructor de las clases externas ha debido *personalizarse* para aceptar un parámetro de tipo igual a la clase que extiende a JApplet (la referencia al applet, necesaria para invocar los métodos del mismo). Esto limita la reusabilidad del código.

JAVA dispone del concepto de clase anidada (clase definida dentro de otra clase), que suele explotarse precisamente en el contexto de la gestión de eventos.

El siguiente código equivale al anterior pero, en este caso, para actuar de oyente se crean dos nuevas clases dentro de la clase `Dibujo` (una como extensión de `MouseAdapter` y otra como extensión de `MouseMotionAdapter`). En este caso, al tratarse de código interno al applet, todos los atributos y métodos del mismo están disponibles desde las clases anidadas sin necesidad de emplear referencias ni implementar nuevos métodos de acceso controlado a los atributos del applet.

```
  import java.awt.event.*;
  import javax.swing.*;

  public class Dibujo extends JApplet {

    class OyenteRaton extends MouseAdapter {
      public void mousePressed (MouseEvent e) {
        x0 = e.getX();
        y0 = e.getY();
      }
    }

    class OyenteMovRaton extends MouseMotionAdapter {
      public void mouseDragged (MouseEvent e) {
        int x = e.getX(), y = e.getY();
        // recta desde punto anterior
        getGraphics().drawLine(x0, y0, x, y);
        x0 = x;
        y0 = y;
      }
    }

    private int x0, y0; // última coordenada
```

[4] Los métodos cuya única finalidad es proporcionar acceso de lectura y escritura a los atributos de una clase reciben el nombre de *getters* y *setters*, respectivamente.

```
  public void init () {
    addMouseListener(new OyenteRaton());
    addMouseMotionListener(new OyenteMovRaton());
  }

}
```

Implementación mediante clases anónimas De forma equivalente a la anterior, se pueden declarar las clases oyente como clases anónimas en lugar de como clases internas. Esto simplifica el código si dichas clases anónimas tienen pocas líneas, como suele ser el caso de los oyentes. Una clase anónima es una clase que se define (se proporciona el cuerpo de la clase) y se instancia (se crea un objeto de la clase) en la misma sentencia de código. Toda clase anónima se define como hija de otra clase, a la que sobreescribe los métodos necesarios; o como implementación de una interfaz, de la que proporciona el código de todos sus métodos. La sintaxis consiste en el operador `new` seguido del nombre de la clase padre o de la interfaz implementada[5] seguido de unos paréntesis. Si la clase anónima hereda de una superclase, entre los paréntesis se proporcionarán los parámetros que se quieran pasar al constructor de dicha superclase (en el caso de los adaptadores, los constructores no reciben ningún parámetro por lo que simplemente se pondrán dos paréntesis). Inmediatamente después, entre llaves, se define el cuerpo de la clase anónima igual que se haría para cualquier otra clase. Las clases anónimas tienen varias peculiaridades adicionales, pero no es necesario conocerlas para su empleo como clases oyente.

```
import java.awt.event.*;
import javax.swing.*;

public class Dibujo extends JApplet {

  private int x0, y0; // última coordenada

  public void init () {
    addMouseListener(new MouseAdapter() {
      public void mousePressed (MouseEvent e) {
        x0 = e.getX();
        y0 = e.getY();
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged (MouseEvent e) {
        int x = e.getX(), y = e.getY();
        // recta desde punto anterior
        getGraphics().drawLine(x0, y0, x, y);
        x0 = x;
        y0 = y;
      }
    });
  }

}
```

[5] Al proporcionar sólo el nombre de la clase padre o de la interfaz implementada, no se asigna un nombre propio a estas clases, razón por la que reciben el nombre de anónimas.

6.7. Paso de parámetros en ficheros HTML

Los applets se invocan mediante etiquetas incluidas en un documento HTML.

```
<APPLET CODE="Ejemplo.class" HEIGHT="100" WIDTH="50"></APPLET>
```

Además de los campos obligatorios (CODE, WIDTH, HEIGHT), se puede añadir el atributo CODEBASE para indicar la URL donde está el código (sólo es necesario si no coincide con la ubicación del fichero HTML):

```
<APPLET CODEBASE="una URL" CODE="Ejemplo.class"
   HEIGHT=100 WIDTH=50></APPLET>
```

En ocasiones se desea pasar parámetros al invocar el applet. La solución es utilizar la etiqueta <PARAM> como sigue:

```
<APPLET CODE="Ejemplo.class" HEIGHT="100" WIDTH="50">
  <PARAM NAME="valor" VALUE="230">
  <PARAM NAME="nombre" VALUE="Juan Perez">
  <PARAM NAME="dirCorreo" VALUE="jperez@nisesabe.es">
</APPLET>
```

Cada parámetro requiere su propia etiqueta <PARAM>, donde se indica el nombre del parámetro y su valor (en ambos casos se trata de cadenas de caracteres). Para acceder a los parámetros desde el applet se invoca `getParameter(nombreParametro)`, que devuelve una cadena de caracteres que corresponde al valor del parámetro indicado.

```
import javax.swing.*;

public class Ejemplo extends JApplet {

  private String nombre, dirCorreo;
  private int valor;

  public void init () {
    nombre = getParameter("nombre");
    dirCorreo = getParameter("dirCorreo");
    valor = Integer.parseInt(getParameter("valor"));
  }
}
```

Los aspectos a destacar de este esquema son:

- Se accede a los parámetros por nombre, no por posición (la posición relativa de los parámetros en el fichero HTML no es relevante).
- Si se intenta acceder a un parámetro que no existe el resultado es `null`. Si existe, el resultado es la cadena de caracteres que corresponde a su valor (en su caso el programa deberá interpretarla posteriormente).

El siguiente programa ilustra estos aspectos. Fichero HTML:

```
<APPLET CODE="EjemploParam.class" WIDTH="500" HEIGHT="450">
  <PARAM NAME="a1" VALUE="Leonardo Gimeno|leo@uno.es">
  <PARAM NAME="a2" VALUE="Ramon Martos|ram@dos.es">
  <PARAM NAME="a3" VALUE="Luis Perez|lp@tres.es">
</APPLET>
```

Fichero JAVA:

```
import java.awt.*;
import javax.swing.*;
import java.util.StringTokenizer;

public class EjemploParam extends JApplet {

  private static final int max = 30;

  public void paint (Graphics g) {
    for (int i = 1; i <= max; i++) {
      String s = getParameter("a" + i);
      if (s != null) {
        StringTokenizer st = new StringTokenizer(s, "|");
        g.drawString("nombre = " + st.nextToken() +
                     "; dirección = " + st.nextToken(), 10, 20*i);
      }
    }
  }

}
```

6.8. Restricciones y posibilidades de los applets

Desde el punto de vista de la seguridad, la ejecución en una máquina cliente de un programa cargado desde un servidor remoto es un riesgo. Los applets de JAVA tratan de minimizar estos riesgos:

- JAVA es un lenguaje *seguro*, con comprobaciones intensivas en ejecución, sin aritmética de punteros, etc.
- Se establece un contexto de ejecución rígido (la *sandbox*). La máquina virtual JAVA aísla a los applets y evita interferencias que pudieran comprometer la integridad del sistema:
  - Un applet no puede definir métodos nativos, cargar bibliotecas ni arrancar programas en la máquina cliente.
  - Un applet no puede leer/escribir ficheros sobre la máquina local, ni acceder a determinadas propiedades del sistema (por ejemplo, el nombre de usuario o la ruta de su directorio home).
  - Un applet sólo puede abrir conexiones de red con el servidor desde el que se ha descargado dicho applet.
  - Las ventanas abiertas por el applet poseen un aspecto distinto a las abiertas por un aplicación.

En algunos casos estas restricciones pueden parecer excesivas. Cada navegador puede definir distintos niveles de restricciones, de forma que el usuario puede relajar las restricciones si lo desea. Sin embargo, resulta más útil asociar una firma digital al fichero, de forma que si se carga un applet *firmado* por un servidor que ha sido etiquetado como fiable, se relajan las restricciones de ejecución.

Pero no todo son restricciones, ya que los applets disfrutan de un conjunto de posibilidades vedadas a una aplicación JAVA autónoma, como abrir conexiones de red con la máquina remota, visualizar documentos HTML, o invocar métodos públicos de otros applets en la misma máquina.

6.9. Guías de redibujado: el método `paint()`

Se ha dicho anteriormente que el navegador *invoca* el método `paint()` cuando detecta que la zona de interacción del applet debe repintarse. Esto es una simplificación, puesto que, en realidad, el navegador no invoca dicho método directamente, sino que lanza una petición de redibujado que lleva finalmente a tal invocación.

De esta forma, el navegador avisa al applet siempre que sea necesario repintar la pantalla, siendo el applet responsable de ejecutar el código adecuado.

El propio applet también puede iniciar una petición de repintado mediante una llamada a `repaint()` (no se debe invocar a `paint()` directamente), cuando necesite redibujar la interfaz para adecuarla a cambios en el estado.

Ya sea mediante un aviso del navegador, o por iniciativa del propio código del applet, finalmente se invoca el método `paint(Graphics g)` del applet, cuyo parámetro es el contexto gráfico asociado al área de visualización.

El método `paint()` merece atención especial debido a las diferentes implementaciones que han dado AWT y Swing. Sin embargo, no es nuestro objetivo conocer en profundidad las peculiaridades de este método en ambas bibliotecas. Por ello, a continuación se ofrecen unas directrices que aseguran el correcto funcionamiento de las interfaces gráficas:

- si se quieren dibujar figuras, estas deberían colocarse en componentes o paneles dedicados a ello, es decir, que no incluyan más componentes. Adicionalmente, el método donde colocar el código de pintado personalizado es distinto si dicho panel o componente pertenece a una u otra biblioteca. Así, tenemos que:
  - en AWT, se debe sobreescribir el método `paint()` incluyendo como primera línea una llamada a `super.paint()`.
  - en Swing, se debería sobreescribir el método `paintComponent()` incluyendo como primera línea una llamada a `super.paintComponent()`.
- en ningún caso debería sobreescribirse el método `paint()` de un applet o de otro contenedor principal (ventana, marco o diálogo –ver sección 6.5.1–).

Estas directrices, aunque quizás más laboriosas que simplemente colocar el código de pintado en el método `paint()` del applet (como se hace por simplicidad en algunos ejemplos del tema), son más seguras puesto que evitan ciertos comportamientos no deseados que suceden a veces al redibujar las interfaces.

6.10. Cuestiones

Cuestión 1 ¿En qué se diferencian una aplicación JAVA autónoma y un applet?
Cuestión 2 ¿Cuáles son los métodos que constituyen el ciclo de vida de un applet?
Cuestión 3 ¿Qué utilidad tienen los contenedores y los gestores de ubicación?
Cuestión 4 ¿Qué condiciones debe cumplir un elemento para poder escuchar un evento y realizar una acción en respuesta a dicho evento?

Cuestión 5 ¿Qué ocurre si, al dispararse un evento, no hay ningún oyente registrado en la fuente?
Cuestión 6 ¿Qué ventajas e inconvenientes presenta la implementación de oyentes en clases externas independientes frente a la implementación en el propio applet?

6.11. Ejercicios resueltos

Ejercicio 1 Crear un applet que convierta valores de temperatura expresados en grados Celsius a sus valores equivalentes en grados Farenheit. Implementar los oyentes necesarios mediante clases externas independientes. Todos los atributos del applet deben ser declarados como privados.

Solución: La interfaz requerida para la escucha de eventos de botón, `ActionListener`, sólo tiene un método. Debido a esto, no existe un adaptador para ella. De esta forma, la escucha de eventos se debe hacer necesariamente implementando la interfaz.

Por otra parte, como los atributos del applet deben ser privados, se deben implementar métodos getters y setters para permitir el acceso desde la clase oyente.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ConversorTemperatura extends JApplet {

  private JButton bConvertir;
  private JTextField tfCen, tfFar;

  public void init() {

    setLayout(new FlowLayout());

    // Crear componentes
    tfCen = new JTextField(5);
    tfFar = new JTextField(5);
    bConvertir = new JButton("Convertir");

    // Añadir componentes
    add(new JLabel("C"));
    add(tfCen);
    add(new JLabel("F"));
    add(tfFar);
    add(bConvertir);

    // Crear oyente y registrar con fuente (boton Convertir)
    MiOyente oy = new MiOyente(this);
    bConvertir.addActionListener(oy);
  }

  public JTextField getTfFar () {
    return tfFar;
  }
```

```
  public JTextField getTfCen () {
    return tfCen;
  }

}

class MiOyente implements ActionListener {

  private ConversorTemperatura elApplet;

  public MiOyente (ConversorTemperatura ct) {
    elApplet = ct;
  }

  public void actionPerformed (ActionEvent e) {
    try {
      float tempCel = Float.parseFloat(elApplet.getTfCen().getText());
      float tempFar = tempCel*9/5 + 32;
      elApplet.getTfFar().setText("" + tempFar);
    } catch(NumberFormatException ex) {
      elApplet.getTfFar().setText("Error");
    }
  }

}
```

Como se puede observar, sólo ha sido necesario implementar métodos getters para los atributos, ya que se trata de referencias a objetos y no de tipos simples. Adicionalmente, `tfFar` y `tfCen` son componentes de la interfaz del applet, más que atributos en el sentido tradicional (representantes del estado interno de la clase). Por ello, otra forma posible de resolver este ejercicio sería pasando las referencias a dichos componentes de interfaz como parámetros del constructor de la clase oyente. Esta solución no será siempre adecuada ya que el oyente podría necesitar acceder a muchos componentes de interfaz.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ConversorTemperatura extends JApplet {

  private JButton bConvertir;
  private JTextField tfCen, tfFar;

  public void init() {

    setLayout(new FlowLayout());

    // Crear componentes
    tfCen = new JTextField(5);
    tfFar = new JTextField(5);
    bConvertir = new JButton("Convertir");

    // Añadir componentes
    add(new JLabel("C"));
```

```
      add(tfCen);
      add(new JLabel("F"));
      add(tfFar);
      add(bConvertir);

      // Crear oyente y registrar con fuente (boton Convertir)
      MiOyente oy = new MiOyente(tfCen, tfFar);
      bConvertir.addActionListener(oy);
    }

  }

  class MiOyente implements ActionListener {

    private JTextField tfCen, tfFar;

    public MiOyente (JTextField c, JTextField f) {
      tfCen = c;
      tfFar = f;
    }

    public void actionPerformed (ActionEvent e) {
      try {
        float tempCel = Float.parseFloat(tfCen.getText());
        float tempFar = tempCel*9/5 + 32;
        tfFar.setText("" + tempFar);
      } catch(NumberFormatException ex) {
        tfFar.setText("Error");
      }
    }

  }
```

Ejercicio 2 Crear un applet que detecte cuando el ratón se sitúa sobre una zona especial resaltada en rojo e informe de ello al usuario. Implementar la escucha de los eventos de todas las formas estudiadas: en el applet, en clases externas, en clases internas y en clases anónimas.

Solución: En este caso, la interfaz correspondiente a los eventos de ratón, `MouseListener`, contiene más de un método, con lo que se dispone de un adaptador. De esta forma, la escucha de los eventos se puede hacer tanto mediante la implementación de la interfaz como mediante el adaptador.

- Escucha de eventos en el propio applet. Como el applet ya extiende de `JApplet`, la única forma disponible de escuchar eventos es implementando la interfaz correspondiente.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class PanelRojo extends JPanel {
  public PanelRojo () {
    super();
    setBackground(Color.red);
```

```
    }
  }

  public class Sensible extends JApplet implements MouseListener {

    private TextField tf;
    private PanelRojo pr;

    public void init () {
      tf = new TextField();
      pr = new PanelRojo();
      pr.addMouseListener(this);
      add(BorderLayout.NORTH, pr);
      add(BorderLayout.SOUTH, tf);
    }

    public void mouseEntered (MouseEvent e) {
      tf.setText("El ratón está sobre la zona roja");
    }

    public void mouseExited (MouseEvent e) {
      tf.setText("El ratón está fuera de la zona roja");
    }

    public void mouseClicked (MouseEvent e) {}
    public void mousePressed (MouseEvent e) {}
    public void mouseReleased (MouseEvent e) {}

  }
```

- Escucha de eventos en una clase externa independiente mediante la implementación de la interfaz. Como en este caso el oyente sólo necesita acceder al campo de texto, podemos pasarle dicha referencia en su constructor, en lugar de pasarle la referencia al applet.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class PanelRojo extends JPanel {
  public PanelRojo () {
    super();
    setBackground(Color.red);
  }
}

public class Sensible extends JApplet {

  private TextField tf;
  private PanelRojo pr;

  public void init () {
    tf = new TextField();
    pr = new PanelRojo();
```

```
    pr.addMouseListener(new OyenteExterno(tf));
    add(BorderLayout.NORTH, pr);
    add(BorderLayout.SOUTH, tf);
  }

}

class OyenteExterno implements MouseListener {

  private TextField tf;

  public OyenteExterno (TextField tf) {
    this.tf = tf;
  }

  public void mouseEntered (MouseEvent e) {
    tf.setText("El ratón está sobre la zona roja");
  }

  public void mouseExited (MouseEvent e) {
    tf.setText("El ratón está fuera de la zona roja");
  }

  public void mouseClicked (MouseEvent e) {}
  public void mousePressed (MouseEvent e) {}
  public void mouseReleased (MouseEvent e) {}

}
```

- Escucha de eventos en una clase externa independiente mediante el uso del adaptador.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class PanelRojo extends JPanel {
  public PanelRojo () {
    super();
    setBackground(Color.red);
  }
}

public class Sensible extends JApplet {

  private TextField tf;
  private PanelRojo pr;

  public void init () {
    tf = new TextField();
    pr = new PanelRojo();
    pr.addMouseListener(new OyenteExterno(tf));
    add(BorderLayout.NORTH, pr);
    add(BorderLayout.SOUTH, tf);
  }
```

```
}

class OyenteExterno extends MouseAdapter {

  private TextField tf;

  public OyenteExterno (TextField tf) {
    this.tf = tf;
  }

  public void mouseEntered (MouseEvent e) {
    tf.setText("El ratón está sobre la zona roja");
  }

  public void mouseExited (MouseEvent e) {
    tf.setText("El ratón está fuera de la zona roja");
  }

}
```

- Escucha de eventos en una clase interna mediante la implementación de la interfaz.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class PanelRojo extends JPanel {
  public PanelRojo () {
    super();
    setBackground(Color.red);
  }
}

public class Sensible extends JApplet {

  class OyenteInterno implements MouseListener {

    public void mouseEntered (MouseEvent e) {
      tf.setText("El ratón está sobre la zona roja");
    }

    public void mouseExited (MouseEvent e) {
      tf.setText("El ratón está fuera de la zona roja");
    }

    public void mouseClicked (MouseEvent e) {}
    public void mousePressed (MouseEvent e) {}
    public void mouseReleased (MouseEvent e) {}

  }
  private TextField tf;
  private PanelRojo pr;

  public void init () {
    tf = new TextField();
```

```
      pr = new PanelRojo();
      pr.addMouseListener(new OyenteInterno());
      add(BorderLayout.NORTH, pr);
      add(BorderLayout.SOUTH, tf);
    }

  }
```

- Escucha de eventos en una clase interna mediante el uso del adaptador.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class PanelRojo extends JPanel {
  public PanelRojo () {
    super();
    setBackground(Color.red);
  }
}

public class Sensible extends JApplet {

  class OyenteInterno extends MouseAdapter {

    public void mouseEntered (MouseEvent e) {
      tf.setText("El ratón está sobre la zona roja");
    }

    public void mouseExited (MouseEvent e) {
      tf.setText("El ratón está fuera de la zona roja");
    }

  }
  private TextField tf;
  private PanelRojo pr;

  public void init () {
    tf = new TextField();
    pr = new PanelRojo();
    pr.addMouseListener(new OyenteInterno());
    add(BorderLayout.NORTH, pr);
    add(BorderLayout.SOUTH, tf);
  }

}
```

- Escucha de eventos en una clase anónima mediante la implementación de la interfaz.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class PanelRojo extends JPanel {
  public PanelRojo () {
```

```
      super();
      setBackground(Color.red);
    }
  }

  public class Sensible extends JApplet {

    private TextField tf;
    private PanelRojo pr;

    public void init () {
      tf = new TextField();
      pr = new PanelRojo();
      pr.addMouseListener(
        new MouseListener() {
          public void mouseEntered (MouseEvent e) {
            tf.setText("El ratón está sobre la zona roja");
          }
          public void mouseExited (MouseEvent e) {
            tf.setText("El ratón está fuera de la zona roja");
          }
          public void mouseClicked (MouseEvent e) {}
          public void mousePressed (MouseEvent e) {}
          public void mouseReleased (MouseEvent e) {}
        });
      add(BorderLayout.NORTH, pr);
      add(BorderLayout.SOUTH, tf);
    }

  }
```

- Escucha de eventos en una clase anónima mediante el uso del adaptador.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class PanelRojo extends JPanel {
  public PanelRojo () {
    super();
    setBackground(Color.red);
  }
}

public class Sensible extends JApplet {

  private TextField tf;
  private PanelRojo pr;

  public void init () {
    tf = new TextField();
    pr = new PanelRojo();
    pr.addMouseListener(
      new MouseAdapter() {
        public void mouseEntered (MouseEvent e) {
```

```
            tf.setText("El ratón está sobre la zona roja");
          }
          public void mouseExited (MouseEvent e) {
            tf.setText("El ratón está fuera de la zona roja");
          }
        });
      add(BorderLayout.NORTH, pr);
      add(BorderLayout.SOUTH, tf);
    }

  }
```

Ejercicio 3 Un ordenador se halla conectado a un sensor que puede generar dos tipos de evento (`ACTION1` y `ACTION2`). Se desea controlar dicho sensor mediante el sistema de eventos de JAVA. Dado que no existe actualmente ningún evento que pueda responder a esta necesidad, será necesario incorporar al sistema de eventos de JAVA las herramientas oportunas que permitan manejar los eventos que produce dicho sensor. Para ello, se debe realizar lo siguiente:

- Definir un nuevo evento `SensorEvent`, teniendo en cuenta que todos los eventos en JAVA son subtipos de `EventObject`.
- Definir la clase `Sensor` para que se pueda utilizar como un componente de interfaz gráfica de usuario (todos los componentes en JAVA heredan de la clase `Component`). Dicho componente es una fuente del evento definido en el punto anterior y, por tanto, debe permitir la incorporación de un oyente (listener) para dicho evento (`addSensorListener`).
- Definir las interfaces y adaptadores necesarios para poder crear oyentes del evento.
- Definir un applet y añadirle un componente de tipo `Sensor`, incorporándole a continuación un oyente para responder únicamente a `ACTION1`.

Solución:

```
import java.awt.*;
import javax.swing.*;

class SensorEvent extends java.util.EventObject {
  ...
}

class Sensor extends Component {
   public void addSensorListener (SensorListener l) {
     ...
   }
}

interface SensorListener {
  public void action1Performed (SensorEvent e);
  public void action2Performed (SensorEvent e);
}

class SensorAdapter implements SensorListener {
  public void action1Performed (SensorEvent e) { }
```

```
    public void action2Performed (SensorEvent e) { }
  }

  public class PruebaSensor extends JApplet {
    public void init() {
      Sensor sensor = new Sensor();
      add(sensor);
      sensor.addSensorListener(new SensorAdapter() {
        public void action1Performed (SensorEvent e) {
          ...
        }
      });
    }
  }
```

Ejercicio 4 Implementar las clases y métodos necesarios para obtener el applet de la figura 6.9. Proporcionar también el fichero HTML necesario para cargar el applet en un navegador.

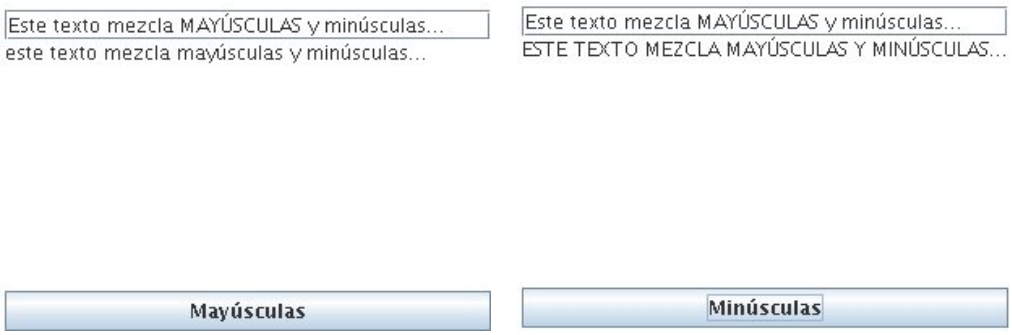

Figura 6.9: Applet antes y después de pulsar el botón

Como puede verse en la figura, el applet consta de tres partes: un campo de texto donde se permite la introducción de texto por parte del usuario, un área donde aparece reflejado dicho texto pero todo en mayúsculas o minúsculas, y un botón que sirve para cambiar el modo de transformación.

El comportamiento del applet debe ser el siguiente. A medida que el usuario escribe en el campo de texto, el área de texto se va actualizando automáticamente. El texto que aparece en dicha área es exactamente igual al del campo de texto excepto en que está todo en minúsculas o todo en mayúsculas. Pulsar el botón implica cambiar de modo. Así, cuando el texto se transforma en minúsculas, en el botón aparece "Mayúsculas" para cambiar a dicho modo y viceversa. Pulsar el botón hace que se cambie de modo de transformación y también transforma todo lo escrito hasta el momento.

Solución: Se introducen en este ejercicio varios aspectos:

- Los métodos `setEditable()` y `setLineWrap()` de `JTextArea` que transforman el área de texto para que sea o no editable, y para que haga o no ajuste de las líneas, respectivamente.

- Uso del método `getSource()` de un evento que permite obtener una referencia al componente de interacción que lanzó el evento. En este caso, nos permite obtener una referencia al botón para leer su *estado* y cambiarlo (dicho *estado* está representado por el texto del botón, que corresponde con el modo de transformación del texto).
- Un evento propio de Swing, `DocumentEvent` que permite escuchar eventos de transformación del texto contenido en un `JTextField` o en un `JTextArea` en *tiempo real*, es decir, a medida que se producen, letra a letra. Este evento tiene un funcionamiento similar a los vistos hasta el momento, aunque presenta ciertas diferencias. La más inmediata es que su fuente no es directamente el componente de interacción, sino un atributo del mismo: el `Document`. Esto es así porque en Swing se emplea el paradigma Modelo-Vista-Controlador, que separa los datos (Modelo) de la representación visual de los mismos (Vista) y del código que responde a eventos y realiza los cambios oportunos en los anteriores elementos (Controlador). Así pues, a través del componente de interacción, obtenemos una referencia al modelo y nos registramos como oyentes de los eventos que afecten a los datos. La interfaz de oyente correspondiente en este caso, `DocumentListener`, posee tres métodos: `changedUpdate()`, que informa de cambios en los atributos del texto; `insertUpdate()`, que informa de las inserciones en el texto; y `removeUpdate()`, que informa de los borrados en el texto. Para este ejercicio, sólo nos interesan los dos últimos.

Fichero HTML:

```
<HTML>
<HEAD><TITLE>MinusMayus</TITLE></HEAD>
<BODY>
<APPLET CODE="MinusMayus.class" WIDTH=300 HEIGHT=200></APPLET>
</BODY>
</HTML>
```

Código JAVA:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;

public class MinusMayus extends JApplet {

  private static final String sminus = "Minúsculas";
  private static final String smayus = "Mayúsculas";
  private JButton b;
  private JTextField t;
  private JTextArea a;
  private boolean pasarMinus = true;

  public void init () {
    b = new JButton(smayus);
    t = new JTextField();
    a = new JTextArea("");
    a.setEditable(false);
    a.setLineWrap(true);
```

```
      add(t, BorderLayout.NORTH);
      add(a, BorderLayout.CENTER);
      add(b, BorderLayout.SOUTH);

      b.addActionListener(new EscuchaBoton());
      t.getDocument().addDocumentListener(new DocumentListener() {
          public void changedUpdate (DocumentEvent e) { }
          public void insertUpdate (DocumentEvent e) {
            actualizar();
          }
          public void removeUpdate (DocumentEvent e) {
            actualizar();
          }
          });
    }

    private void actualizar () {
      if (pasarMinus)
        a.setText(t.getText().toLowerCase());
      else
        a.setText(t.getText().toUpperCase());
    }

    private void actualizar (boolean bminus) {
      pasarMinus = bminus;
      actualizar();
    }

    class EscuchaBoton implements ActionListener {

      public void actionPerformed (ActionEvent e) {
        boolean bminus;
        JButton source = (JButton) e.getSource();
        if (source.getText().equals(sminus)) {
          bminus = true;
          source.setText(smayus);
        } else {
          bminus = false;
          source.setText(sminus);
        }
        actualizar(bminus);
      }

    }

  }
```

Ejercicio 5 Disponemos de la clase `ConjuntoPalabras` que representa una estructura de datos tipo conjunto en la que se almacenan cadenas de texto (`String`) y el número de veces que cada cadena de texto está repetida en el conjunto. La interfaz de dicha clase es la siguiente:

- Constructor: `ConjuntoPalabras()`: crea un conjunto vacío.

- Métodos:
  - `boolean insertar (String s) throws ConjuntoLleno`: si la cadena `s` ya pertenece al conjunto, incrementa en uno su contador y devuelve `true`. Si la cadena no está en el conjunto, la inserta con el contador de repetición igual a 1 y devuelve `false`. Si el conjunto está lleno, se lanza la excepción `ConjuntoLleno`.
  - `int buscar (String s)`: devuelve 0 si la cadena pasada como argumento no está en el conjunto, o el número de repeticiones si dicha cadena sí pertenece al conjunto.

Se desea realizar un applet que proporcione una interfaz visual a un conjunto de este tipo. Inicialmente se debe crear un conjunto vacío y posteriormente el usuario podrá añadir cadenas de texto en el mismo. La apariencia visual del applet es la de la figura 6.10.

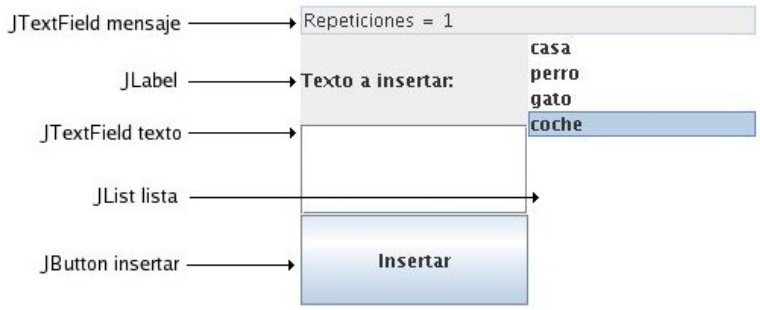

Figura 6.10: Applet para construir conjuntos de palabras

La cadena escrita en el campo de texto se inserta en el conjunto al presionar el botón o bien pulsando Enter cuando el cursor está en el campo de texto. En cualquier caso, si la cadena no estaba en el conjunto, se añade también a la lista de la derecha y aparece en el cuadro de texto de mensajes el mensaje "XXX insertado en el conjunto", siendo XXX el string escrito en el campo de texto. Si la cadena ya estaba en el conjunto, no debe insertarse en la lista de la derecha y debe aparecer en el cuadro de texto de mensajes el mensaje "XXX incrementado en el conjunto". Si al intentar insertar una cadena en el conjunto se genera la excepción `ConjuntoLleno` debe aparecer en el cuadro de texto de mensajes el texto "ERROR" y en la barra de estado debe especificarse el mensaje: "Conjunto lleno".

Por otra parte, si en la lista se selecciona cualquier ítem, en la caja de texto de mensajes aparecerá el número de repeticiones del mismo, por ejemplo: "Repeticiones = 4".

Solución: Aspectos a resaltar en este ejercicio:

- Para mostrar un mensaje en la barra de estado del navegador, se emplea el método `showStatus(String s)`.
- Siguiendo el paradigma Modelo-Vista-Controlador, `JList` emplea un modelo para gestionar los datos (en este caso, las entradas de la lista). Por ello, en lugar de añadir las entradas directamente al `JList`, se han de añadir al modelo. Para simplificar la tarea, JAVA proporciona un modelo básico para listas, `DefaultListModel`. De esta forma, basta con crear una instancia de este modelo y pasarla como parámetro en el

constructor de `JList`. Posteriormente, cuando se deseen añadir entradas a la lista, se empleará la referencia al modelo para hacer dichas inserciones.

Para disponer del código completo, se proporciona también la implementación de las clases `ConjuntoLleno` y `ConjuntoPalabras`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;

class ConjuntoLleno extends Exception {}

class ConjuntoPalabras {

  private static final int max = 10;
  private HashMap palabras = new HashMap();
  private int cont = 0;

  public boolean insertar (String s) throws ConjuntoLleno {
    Integer r = (Integer) palabras.get(s);
    if (r == null) {
      if (cont == max) throw new ConjuntoLleno();
      cont++;
      palabras.put(s, new Integer(1));
      return false;
    } else {
      palabras.put(s, new Integer(r.intValue()+1));
      return true;
    }
  }

  public int buscar (String s) {
    Integer r = (Integer) palabras.get(s);
    if (r == null) return 0;
    return r.intValue();
  }

}

public class CreadorConjunto extends JApplet {

  private JTextField mensaje;
  private JTextField texto;
  private JButton insertar;
  private ConjuntoPalabras c;
  private DefaultListModel modelo;
  private JList lista;

  public void init () {
    mensaje = new JTextField(5);
    mensaje.setEditable(false);
    texto = new JTextField(5);
    insertar = new JButton("Insertar");
```

```
      c = new ConjuntoPalabras();
      modelo = new DefaultListModel();
      lista = new JList(modelo);

      add(mensaje, BorderLayout.NORTH);

      JPanel p1 = new JPanel();
      p1.setLayout(new GridLayout(1, 2));
      add(p1, BorderLayout.CENTER);

      JPanel p2 = new JPanel();
      p2.setLayout(new GridLayout(3, 1));
      p2.add(new JLabel("Texto a insertar:"));
      p2.add(texto);
      p2.add(insertar);

      p1.add(p2);
      p1.add(lista);

      Oyente oy = new Oyente();
      insertar.addActionListener(oy);
      texto.addActionListener(oy);

      lista.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
      lista.addListSelectionListener(new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e) {
          int r = c.buscar((String) lista.getSelectedValue());
          mensaje.setText("Repeticiones = " + r);
        }
      });
    }

    class Oyente implements ActionListener {

      public void actionPerformed (ActionEvent e) {
        try {
          String s = texto.getText();
          if (s.length() == 0) return;
          texto.setText("");
          if (!c.insertar(s)) {
            modelo.addElement(s);
            mensaje.setText(s + " insertado en el conjunto");
          } else {
            mensaje.setText(s + " incrementado en el conjunto");
          }
        } catch (ConjuntoLleno ex) {
          mensaje.setText("ERROR");
          showStatus("Conjunto lleno");
        }
      }

    }

  }
```

6.12. Ejercicios propuestos

Ejercicio 1 Implementar el juego "Adivina el número". Este juego consiste en adivinar un número aleatorio en un máximo de 5 intentos. Tras cada intento, el applet responde si se ha acertado o, en caso contrario, si el número a adivinar es mayor o menor que el número propuesto por el jugador.
Ejercicio 2 Implementar el juego "Buscaminas".
Ejercicio 3 Implementar el juego "Hundir la Flota".

UNIVERSIDAD
POLITECNICA
DE VALENCIA