Todo es un objeto 2

"Si habláramos un lenguaje diferente, percibiríamos un mundo algo distinto".
Ludwig Wittgenstein (1889-1951)

Aunque está basado en C++, Java es un lenguaje orientado a objetos más "puro".

Tanto C++ como Java son lenguajes híbridos, pero en Java los diseñadores pensaron que esa hibridación no era tan importante con en C++. Un lenguaje híbrido permite utilizar múltiples estilos programación; la razón por la que C++ es capaz de soportar la compatibilidad descendente con el lenguaje C. Puesto que C++ es un superconjunto del lenguaje C, incluye muchas de las características menos deseables de ese lenguaje, lo que hace que algunos aspectos del C++ sean demasiado complicados.

El lenguaje Java presupone que el programador sólo quiere realizar programación orientada a objetos. Esto quiere decir que, antes de empezar, es preciso cambiar nuestro esquema mental al del mundo de la orientación a objetos (a menos que ya hayamos efectuado esa transición). La ventaja que se obtiene gracias a este esfuerzo adicional es la capacidad de programar en un lenguaje que es más fácil de aprender y de utilizar que muchos otros lenguajes orientados a objetos. En este capítulo veremos los componentes básicos de un programa Java y comprobaremos que (casi) todo en Java es un objeto.

Los objetos se manipulan mediante referencias

Cada lenguaje de programación dispone de sus propios mecanismos para manipular los elementos almacenados en memoria. En ocasiones, el programador debe ser continuamente consciente del tipo de manipulación que se está efectuando. ¿Estamos tratando con el elemento directamente o con algún tipo de representación indirecta (un puntero en C o C++), que haya que tratar con una sintaxis especial?

Todo esto se simplifica en Java. En Java, todo se trata como un objeto, utilizando una única sintaxis coherente. Aunque *tratamos* todo como un objeto, los identificadores que manipulamos son en realidad "referencias" a objetos.[1] Podríamos imaginarnos una TV (el objeto) y un mando a distancia (la referencia); mientras dispongamos de esta referencia tendremos una conexión con la televisión, pero cuando alguien nos dice "cambia de canal" o "baja el volumen", lo que hacemos es manipular la referencia, que a su vez modifica el objeto. Si queremos movernos por la habitación y continuar controlando la TV, llevamos con nosotros el mando a distancia/referencia, no la televisión.

---

[1] Este punto puede suscitar enconados debates. Hay personas que sostienen que "claramente se trata de un puntero", pero esto está presuponiendo una determinada implementación subyacente. Asimismo, las referencias en Java se parecen mucho más sintácticamente a las referencias C++ que a los punteros. En la primera edición de este libro decidí utilizar el término "descriptor" porque las referencias C++ y las referencias Java tienen diferencias notables. Yo mismo provenía del mundo del lenguaje C++ y no quería confundir a los programadores de C++, que suponía que constituirían la gran mayoría de personas interesadas en el lenguaje Java. En la segunda edición, decidí que "referencia" era el término más comúnmente utilizado, y que cualquiera que proviniera del mundo de C++ iba a enfrentarse a problemas mucho más graves que la terminología de las referencias, por lo que no tenía sentido usar una palabra distinta. Sin embargo, hay personas que están en desacuerdo incluso con el término "referencia". En un determinado libro, pude leer que "resulta completamente equivocado decir que Java soporta el paso por referencia", o que los identificadores de los objetos Java (de acuerdo con el autor del libro) son en realidad "referencias a objetos". Por lo que (continúa el autor) todo se pasa *en la práctica* por valor. Según este autor, no se efectúa un paso por referencia, sino que se "pasa una referencia a objeto por valor". Podríamos discutir acerca de la precisión de estas complicadas explicaciones, pero creo que el enfoque que he adoptado en este libro simplifica la compresión del concepto sin generar ningún tipo de problema (los puristas del lenguaje podrían sostener que estoy mintiendo, pero a eso respondería que lo que estoy haciendo es proporcionar una abstracción apropiada).

Asimismo, el mando a distancia puede existir de manera independiente, sin necesidad de que exista una televisión. En otras palabras, el hecho de que dispongamos de una referencia no implica necesariamente que haya un objeto conectado a la misma. De este modo, si queremos almacenar una palabra o una frase podemos crear una referencia de tipo **String**:

```
String s;
```

Pero con ello *sólo* habremos creado la referencia a un objeto. Si decidiéramos enviar un mensaje a **s** en este punto, obtendríamos un error porque **s** no está asociado a nada (no hay televisión). Por tanto, una práctica más segura consiste en inicializar siempre las referencias en el momento de crearlas:

```
String s = "asdf";
```

Sin embargo, aquí estamos utilizando una característica especial de Java. Las cadenas de caracteres pueden inicializarse con un texto entre comillas. Normalmente, será necesario emplear un tipo más general de inicialización para los restantes objetos.

Es necesario crear todos los objetos

Al crear una referencia, lo que se desea es conectarla con un nuevo objeto. Para ello, en general, se emplea el operador **new**. La palabra clave **new** significa: "Crea un nuevo ejemplar de este tipo de objeto". Por tanto, en el ejemplo anterior podríamos escribir:

```
String s = new String("asdf");
```

Esto no sólo dice: "Crea un nuevo objeto **String**", sino que también proporciona información acerca de *cómo* crear el objeto suministrando una cadena de caracteres inicial.

Por supuesto, Java incluye una plétora de tipos predefinidos, además de **String**. Lo más importante es que también podemos crear nuestros propios tipos. De hecho, la creación de nuevos tipos es la actividad fundamental en la programación Java, y eso es precisamente lo que aprenderemos a hacer en el resto del libro.

Los lugares de almacenamiento

Resulta útil tratar de visualizar la forma en que se dispone la información mientras se ejecuta el programa; en particular, es muy útil ver cómo está organizada la memoria. Disponemos de cinco lugares distintos en los que almacenar los datos:

1. **Registros**. Se trata del tipo de almacenamiento más rápido, porque se encuentra en un lugar distinto al de los demás tipos de almacenamiento: dentro del procesador. Sin embargo, el número de registros está muy limitado, por lo que los registros se asignan a medida que son necesarios. No disponemos de un control directo sobre los mismos, ni tampoco podremos encontrar en los programas que los registros ni siquiera existan (C y C++, por el contrario, permiten sugerir al compilador que el almacenamiento se haga en un registro).
2. **La pila**. Esta zona se encuentra en el área general de memoria de acceso aleatorio (RAM), pero el procesador proporciona un soporte directo para la pila, gracias al *puntero de pila*. El puntero de pila se desplaza hacia abajo para crear nueva memoria y hacia arriba para liberarla. Se trata de una forma extraordinariamente rápida y eficiente de asignar espacio de almacenamiento, sólo superada en rapidez por los registros. El sistema Java debe conocer, a la hora de crear el programa, el tiempo de vida exacto de todos los elementos que se almacenan en la pila. Esta restricción impone una serie de límites a la flexibilidad de los programas, por lo que aunque parte del almacenamiento dentro de Java se lleva a cabo en la pila (en concreto, las referencias a objetos), los propios objetos Java no se colocan nunca en la pila.
3. **El cúmulo**. Es un área de memoria de propósito general (también situada dentro de la RAM) en la que se almacenan todos los objetos Java. El aspecto más atractivo del cúmulo de memoria, a diferencia de la pila, es que el compilador no necesita conocer de antemano durante cuánto tiempo se va a ocupar ese espacio de almacenamiento dentro del cúmulo. Por tanto, disponemos de un alto grado de flexibilidad a la hora de utilizar el espacio de almacenamiento que el cúmulo de memoria proporciona. Cada vez que hace falta un objeto, simplemente se escribe el código para crearlo utilizando **new**, y el espacio de almacenamiento correspondiente en el cúmulo se asigna en el momento de ejecutar el código. Por supuesto, esa flexibilidad tiene su precio: puede que sea necesario un tiempo más largo para asignar y liberar el espacio de almacenamiento del cúmulo de memoria, si lo comparamos con el tiempo necesario

para el almacenamiento en la pila (eso suponiendo que *pudiéramos* crear objetos en la pila en Java, al igual que se hace en C++).

4. **Almacenamiento constante**. Los valores constantes se suelen situar directamente dentro del código de programa, lo que resulta bastante seguro, ya que nunca varían. En ocasiones, las constantes se almacenan por separado, de forma que se pueden guardar opcionalmente en la memoria de sólo lectura (ROM, *read only memory*), especialmente en los sistemas integrados.[2]
5. **Almacenamiento fuera de la RAM**. Si los datos residen fuera del programa, podrán continuar existiendo mientras el programa no se esté ejecutando, sin que el programa tenga control sobre los mismos. Los dos ejemplos principales son los *objetos stream*, en los que los objetos se transforman en flujos de bytes, generalmente para enviarlos a otra máquina, los *objetos persistentes* en los que los objetos se almacenan en disco para que puedan conservar su estado incluso después de terminar el programa. El truco con estos tipos de almacenamiento consiste en transformar los objetos en algo que pueda existir en el otro medio de almacenamiento, y que, sin embargo, pueda recuperarse para transformarlo en un objeto normal basado en RAM cuando sea necesario. Java proporciona soporte para lo que se denomina *persistencia ligera* y otros mecanismos tales como JDBC e Hibernate proporcionan soporte más sofisticado para almacenar y extraer objetos utilizando bases de datos.

Caso especial: tipos primitivos

Hay un grupo de tipos que se emplean muy a menudo en programación y que requieren un tratamiento especial. Podemos considerarlos como tipos "primitivos". La razón para ese tratamiento especial es que crear un objeto con **new**, una variable simple de pequeño tamaño, no resulta muy eficiente, porque **new** almacena los objetos en el cúmulo de memoria. Para estos tipos primitivos, Java utiliza la técnica empleada en C y C++; es decir, en lugar de crear la variable con **new**, se crea una variable "automática" que *no es una referencia*. La variable almacena el valor directamente y se coloca en la pila, por lo que resulta mucho más eficiente.

Java determina el tamaño de cada tipo primitivo. Estos tamaños no cambian de una arquitectura de máquina a otra, a diferencia de lo que sucede en la mayoría de los lenguajes. Esta invariabilidad de los tamaños es una de las razones por la que los programa Java son más portables que los programas escritos en la mayoría de los demás lenguajes.

Tipo primitivo	Tamaño	Mínimo	Máximo	Tipo envoltorio
boolean	—	—	—	**Boolean**
char	16 bits	Unicode 0	Unicode $2^{16}-1$	**Character**
byte	8 bits	−128	+127	**Byte**
short	16 bits	-2^{15}	$+2^{15}-1$	**Short**
int	32 bits	-2^{31}	$+2^{31}-1$	**Integer**
long	64 bits	-2^{63}	$+2^{63}-1$	**Long**
float	32 bits	IEEE754	IEEE754	**Float**
double	64 bits	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

Todos los tipos numéricos tienen signo, por lo que Java no podrá encontrar ningún tipo sin signo.

El tamaño del tipo **boolean** no está especificado de manera explícita; tan sólo se define para que sea capaz de aceptar los valores literales **true** o **false**.

Las clases "envoltorio" para los tipos de datos primitivos permiten definir un objeto no primitivo en el cúmulo de memoria que represente a ese tipo primitivo. Por ejemplo:

[2] Un ejemplo sería el conjunto de cadenas de caracteres. Todas las cadenas literales y constantes que tengan un valor de tipo carácter se toman de forma automática y se asignan a un almacenamiento estático especial.

```
char c = 'x';
Character ch = new Character(c);
```

O también podría emplear:

```
Character ch = new Character('x');
```

La característica de Java SE5 denominada *autoboxing* permite realizar automáticamente la conversión de un tipo primitivo a un tipo envoltorio:

```
Character ch = 'x';
```

Y a la inversa:

```
char c = ch;
```

En un capítulo posterior veremos las razones que existen para utilizar envoltorios con los tipos primitivos.

Arimética de alta precisión

Java incluye dos clases para realizar operaciones aritméticas de alta precisión: **BigInteger** y **BigDecimal**. Aunque estas clases caen aproximadamente en la misma categoría que las clases envoltorio, ninguna de las dos dispone del correspondiente tipo primitivo.

Ambas clases disponen de métodos que proporcionan operaciones análogas a las que se realizan con los tipos primitivos. Es decir, podemos hacer con un objeto **BigInteger** o **BigDecimal** cualquier cosa que podamos hacer con una variable **int** o **float**; simplemente deberemos utilizar llamadas a métodos en lugar de operadores. Asimismo, como son más complejas, las operaciones se ejecutarán más lentamente. En este caso, sacrificamos parte de la velocidad en aras de la precisión.

BigInteger soporta números enteros de precisión arbitraria. Esto quiere decir que podemos representar valores enteros de cualquier tamaño sin perder ninguna información en absoluto durante las operaciones.

BigDecimal se utiliza para números de coma fija y precisión arbitraria. Podemos utilizar estos números, por ejemplo, para realizar cálculos monetarios precisos.

Consulte la documentación del kit JDK para conocer más detalles acerca de los constructores y métodos que se pueden invocar para estas dos clases.

Matrices en Java

Casi todos los lenguajes de programación soportan algún tipo de matriz. La utilización de matrices en C y C++ es peligrosa, porque dichas matrices son sólo bloques de memoria. Si un programa accede a la matriz fuera de su correspondiente bloque de memoria o utiliza la memoria antes de la inicialización (lo cual son dos errores de programación comunes), los resultados serán impredecibles.

Uno de los principales objetivos de Java es la seguridad, por lo que en Java no se presentan muchos de los problemas que aquejan a los programadores en C y C++. Se garantiza que las matrices Java siempre se inicialicen y que no se pueda acceder a ellas fuera de su rango autorizado. Las comprobaciones de rango exigen pagar el precio de gastar una pequeña cantidad de memoria adicional para cada matriz, así como de verificar el índice en tiempo de ejecución, pero se supone que el incremento en productividad y la mejora de la seguridad compensan esas desventajas (y Java puede en ocasiones optimizar estas operaciones).

Cuando se crea una matriz de objetos, lo que se crea realmente es una matriz de referencias, y cada una de esas matrices se inicializa automáticamente con un valor especial que tiene su propia palabra clave: **null**. Cuando Java se encuentra un valor **null**, comprende que la referencia en cuestión no está apuntando a ningún objeto. Es necesario asignar un objeto a cada referencia antes de utilizarla, y si se intenta usar una referencia que siga teniendo el valor **null**, se informará del error en tiempo de ejecución. De este modo, en Java se evitan los errores comunes relacionados con las matrices.

También podemos crear una matriz de valores primitivos. De nuevo, el compilador se encarga de garantizar la inicialización, rellenando con ceros la memoria correspondiente a dicha matriz.

Hablaremos con detalle de las matrices en capítulos posteriores.

Nunca es necesario destruir un objeto

En la mayoría de los lenguajes de programación, el concepto de tiempo de vida de una variable representa una parte significativa del esfuerzo de programación. ¿Cuánto va a durar la variable? Si hay que destruirla, ¿cuándo debemos hacerlo? La confusión en lo que respecta al tiempo de vida de las variables puede generar una gran cantidad de errores de programación, y en esta sección vamos a ver que Java simplifica enormemente este problema al encargarse de realizar por nosotros todas las tareas de limpieza.

Ámbito

La mayoría de los lenguajes procedimentales incluyen el concepto de *ámbito*. El ámbito determina tanto la visibilidad como el tiempo de vida de los nombres definidos dentro del mismo. En C, C++ y Java, el ámbito está determinado por la colocación de las llaves {}, de modo que, por ejemplo:

```
{
  int x = 12;
  // x
  {
    int q = 96;
    // están disponibles tanto x como q
  }
  // sólo está disponible x
  // q está "fuera del ámbito"
}
```

Una variable definida dentro de un ámbito sólo está disponible hasta que ese ámbito termina.

Todo texto situado después de los caracteres '//' y hasta el final de la línea es un comentario.

El sangrado hace que el código Java sea más fácil de leer. Dado que Java es un lenguaje de formato libre, los espacios, tabuladores y retornos de carro adicionales no afectan al programa resultante.

No podemos hacer lo siguiente, a pesar de que sí es correcto en C y C++:

```
{
  int x = 12;
  {
    int x = 96; // Ilegal
  }
}
```

El compilador nos indicaría que la variable **x** ya ha sido definida. Por tanto, no está permitida la posibilidad en C y C++ de "ocultar" una variable en un ámbito más grande, porque los diseñadores de Java pensaron que esta característica hacía los programas más confusos.

Ámbito de los objetos

Los objetos Java no tienen el mismo tiempo de vida que las primitivas. Cuando se crea un objeto Java usando **new**, ese objeto continúa existiendo una vez que se ha alcanzado el final del ámbito. Por tanto, si usamos:

```
{
  String s = new String("a string");
} // Fin del ámbito
```

la referencia **s** desaparece al final del ámbito. Sin embargo, el objeto **String** al que **s** estaba apuntando continuará ocupando memoria. En este fragmento de código, no hay forma de acceder al objeto después de alcanzar el final del ámbito, porque la única referencia a ese objeto está ahora fuera de ámbito. En capítulos posteriores veremos cómo pasar y duplicar una referencia a un objeto durante la ejecución de un programa.

Como los objetos que creamos con **new** continuarán existiendo mientras queramos, hay toda una serie de problemas típicos de programación en C++ que en Java se desvanecen. En C++ no sólo hay que asegurarse de que los objetos permanezcan mientras sean necesarios, sino que también es preciso destruirlos una vez que se ha terminado de usarlos.

Esto suscita una cuestión interesante. Si los objetos continúan existiendo en Java perpetuamente, ¿qué es lo que evita llenar la memoria y hacer que el programa se detenga? Éste es exactamente el tipo de problema que podía ocurrir en C++, y para resolverlo Java recurre a una especie de varita mágica. Java dispone de lo que se denomina *depurador de memoria*, que examina todos los objetos que hayan sido creados con **new** y determina cuáles no tienen ya ninguna referencia que les apunte. A continuación, Java libera la memoria correspondiente a esos objetos, de forma que pueda ser utilizada para crear otros objetos nuevos. Esto significa que nunca es necesario preocuparse de reclamar explícitamente la memoria. Basta con crear los objetos y, cuando dejen de ser necesarios, se borrarán automáticamente. Esto elimina un cierto tipo de problema de programación: las denominadas "fugas de memoria" que se producen cuando, en otros lenguajes, un programador se olvida de liberar la memoria.

Creación de nuevos tipos de datos: class

Si todo es un objeto, ¿qué es lo que determina cómo se comporta y qué aspecto tiene una clase concreta de objeto? Dicho de otro modo, ¿qué es lo que establece el *tipo* de un objeto? Cabría esperar que existiera una palabra clave denominada "type" (tipo), y de hecho tendría bastante sentido. Históricamente, sin embargo, la mayoría de los lenguajes orientados a objetos han utilizado la palabra clave **class** para decir "voy a definir cuál es el aspecto de un nuevo tipo de objeto". La palabra clave **class** (que es tan común que la escribiremos en negrita normalmente a lo largo del libro) va seguida del nombre del nuevo tipo que queremos definir. Por ejemplo:

```
class ATypeName { /* Aquí iría el cuerpo de la clase */ }
```

Este código permite definir un nuevo tipo, aunque en este caso el cuerpo de la clase sólo incluye un comentario (las barras inclinadas y los asteriscos junto con el texto forman el comentario, lo que veremos en detalle más adelante en este capítulo), así que no es mucho lo que podemos hacer con esta clase que acabamos de definir. Sin embargo, sí que podemos crear un objeto de este tipo utilizando **new**:

```
ATypeName a = new ATypeName();
```

Si bien es verdad que no podemos hacer que ese objeto lleve a cabo ninguna tarea útil (es decir, no le podemos enviar ningún mensaje interesante) hasta que definamos algunos métodos para ese objeto.

Campos y métodos

Cuando se define una clase (y todo lo que se hace en Java es definir clases, crear objetos de esa clase y enviar mensajes a dichos objetos), podemos incluir dos tipos de elementos dentro de la clase: *campos* (algunas veces denominados *miembros de datos*) y *métodos* (en ocasiones denominados *funciones miembro*). Un campo es un objeto de cualquier tipo con el que podemos comunicarnos a través de su referencia o bien un tipo primitivo. Si es una referencia a un objeto, hay que inicializar esa referencia para conectarla con un objeto real (usando **new**, como hemos visto anteriormente).

Cada objeto mantiene su propio almacenamiento para sus campos; los campos normales no son compartidos entre los distintos objetos. Aquí tiene un ejemplo de una clase con algunos campos definidos:

```
class DataOnly {
  int i;
  double d;
  boolean b;
}
```

Esta clase no *hace* nada, salvo almacenar una serie de datos. Sin embargo, podemos crear un objeto de esta clase de la forma siguiente:

```
DataOnly data = new DataOnly();
```

Podemos asignar valores a los campos, pero primero es preciso saber cómo referirnos a un miembro de un objeto. Para referirnos a él, es necesario indicar el nombre de la referencia al objeto, seguida de un punto y seguida del nombre del miembro concreto dentro del objeto:

```
objectReference.member
```

Por ejemplo:

```
data.i = 47;
data.d = 1.1;
data.b = false;
```

También es posible que el objeto contenga otros objetos, que a su vez contengan los datos que queremos modificar. En este caso, basta con utilizar los puntos correspondientes, como por ejemplo:

```
myPlane.leftTank.capacity = 100;
```

La clase **DataOnly** no puede hacer nada más que almacenar datos, ya que no dispone de ningún método. Para comprender cómo funcionan los métodos es preciso entender primero los conceptos de *argumentos* y *valores de retorno*, que vamos a describir en breve.

Valores predeterminados de los miembros de tipo primitivo

Cuando hay un tipo de datos primitivo como miembro de una clase, Java garantiza que se le asignará un valor predeterminado en caso de que no se inicialice:

Tipo primitivo	Valor predeterminado
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Los valores predeterminados son sólo los valores que Java garantiza cuando se emplea la variable *como miembro de una clase*. Esto garantiza que las variables miembro de tipo primitivo siempre sean inicializadas (algo que C++ no hace), reduciendo así una fuente de posibles errores. Sin embargo, este valor inicial puede que no sea correcto o ni siquiera legal para el programa que se esté escribiendo. Lo mejor es inicializar las variables siempre de forma explícita.

Esta garantía de inicialización no se aplica a las *variables locales*, aquellas que no son campos de clase. Por tanto, si dentro de la definición de un método tuviéramos:

```
int x;
```

Entonces **x** adoptaría algún valor arbitrario (como en C y C++), no siendo automáticamente inicializada con el valor cero. El programador es el responsable de asignar un valor apropiado antes de usar **x**. Si nos olvidamos, Java representa de todos modos una mejora con respecto a C++, ya que se obtiene un error en tiempo de compilación que nos informa de que la variable puede no haber sido inicializa (muchos compiladores de C++ nos advierten acerca de la existencia de variables no inicializadas, pero en Java esta falta de inicialización constituye un error).

Métodos, argumentos y valores de retorno

En muchos lenguajes (como C y C++), el término *función* se usa para describir una subrutina con nombre. El término más comúnmente utilizado en Java es el de *método*, queriendo hacer referencia a una forma de "llevar algo a cabo". Si lo desea, puede continuar pensado en términos de funciones; se trata sólo de una diferencia sintáctica, pero en este libro adoptaremos el uso común del término "método" dentro del mundo de Java.

Los métodos de Java determinan los mensajes que un objeto puede recibir. Las partes fundamentales de un método son el nombre, los argumentos, el tipo de retorno y el cuerpo. Ésta sería la forma básica de un método:

```
TipoRetorno NombreMetodo( /* Lista de argumentos */ ) {
  /* Cuerpo del método */
}
```

El tipo de retorno describe el valor devuelto por el método después de la invocación. La lista de argumentos proporciona la lista de los tipos y nombres de la información que hayamos pasado al método. El nombre del método y la lista de argumentos (que forman lo que se denomina *signatura* del método) identifican de forma unívoca al método.

Los métodos pueden crearse en Java únicamente como parte de una clase. Los métodos sólo se pueden invocar para un objeto[3], y dicho objeto debe ser capaz de ejecutar esa invocación del método. Si tratamos de invocar un método correcto para un objeto obtendremos un mensaje de error en tiempo de compilación. Para invocar un método para un objeto, hay que nombrar el objeto seguido de un punto, seguido del nombre del método y de su lista de argumentos, como por ejemplo:

```
NombreObjeto.NombreMetodo(arg1, arg2, arg3);
```

Por ejemplo, suponga que tenemos un método **f( )** que no tiene ningún argumento y que devuelve una valor de tipo **int**. Entonces, si tuviéramos un objeto denominado **a** para el que pudiera invocarse **f( )** podríamos escribir:

```
int x = a.f();
```

El tipo del valor de retorno debe ser compatible con el tipo de **x**.

Este acto de invocar un método se denomina comúnmente *enviar un mensaje a un objeto*. En el ejemplo anterior, el mensaje es **f( )** y el objeto es **a**. A menudo, cuando se quiere resumir lo que es la programación orientada a objetos se suele decir que consiste simplemente en "enviar mensajes a objetos".

La lista de argumentos

La lista de argumentos del método especifica cuál es la información que se le pasa al método. Como puede suponerse, esta información (como todo lo demás en Java) adopta la forma de objetos. Por tanto, lo que hay que especificar en la lista de argumentos son los tipos de los objetos que hay pasar y el nombre que hay que utilizar para cada uno. Como en cualquier otro caso dentro de Java en el que parece que estuviéramos gestionando objetos, lo que en realidad estaremos pasando son referencias[4]. Sin embargo, el tipo de la referencia debe ser correcto. Si el argumento es de tipo **String**, será preciso pasar un objeto **String**, porque de lo contrario el compilador nos daría un error.

Supongamos un cierto método que admite un objeto **String** como argumento. Para que la compilación se realice correctamente, la definición que habría que incluir dentro de la definición de la clase sería como la siguiente:

```
int storage(String s) {
  return s.length() * 2;
}
```

Este método nos dice cuántos bytes se requieren para almacenar la información contenida en un objeto **String** concreto (el tamaño de cada **char** en un objeto **String** es de 16 bits, o dos bytes, para soportar los caracteres Unicode). El argumento es de tipo **String** y se denomina **s**. Una vez que se pasa **s** al método, podemos tratarlo como cualquier otro objeto (podemos enviarle mensajes). Aquí, lo que se hace es invocar el método **length( )**, que es uno de los métodos definidos para los objetos de tipo **String**; este método devuelve el número de caracteres que hay en una cadena.

También podemos ver en el ejemplo cómo se emplea la palabra clave **return**. Esta palabra clave se encarga de dos cosas: en primer lugar, quiere decir "sal del método, porque ya he terminado", en segundo lugar, si el método ha generado un valor, ese valor se indica justo a continuación de una instrucción **return**, en este caso, el valor de retorno se genera evaluando la expresión **s.length( ) * 2**.

Podemos devolver un valor de cualquier tipo que deseemos, pero si no queremos devolver nada, tenemos que especificarlo, indicando que el método devuelve un valor de tipo **void**. He aquí algunos ejemplos:

---

[3] Los métodos de tipo **static**, de los que pronto hablaremos, pueden invocarse *para la clase*, en lugar de para un objeto específico.

[4] Con la excepción usual de los tipos de datos "especiales" que antes hemos mencionado: **boolean, char, byte, short, int, long, float** y **double**. En general, sin embargo, lo que se pasan son objetos, lo que realmente quiere decir que se pasan referencias a objetos.

```
boolean flag() { return true; }
double naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

Cuando el tipo de retorno es **void**, la palabra clave **return** se utiliza sólo para salir del método, por lo que resulta necesaria una vez que se alcanza el final del método. Podemos volver de un método en cualquier punto, pero si el tipo de retorno es distinto de **void**, entonces el compilador nos obligará (mediante los apropiados mensajes de error) a devolver un valor del tipo apropiado, independientemente del lugar en el que salgamos del método.

Llegados a este punto, podría parecer que un programa consiste, simplemente, en una serie de objetos con métodos que aceptan otros objetos como argumentos y envían mensajes a esos otros objetos. Realmente, esa descripción es bastante precisa, pero en el siguiente capítulo veremos cómo llevar a cabo el trabajo detallado de bajo nivel, tomando decisiones dentro de un método. Para los propósitos de este capítulo, el envío de mensajes nos resultará más que suficiente.

Construcción de un programa Java

Hay otras cuestiones que tenemos que entender antes de pasar a diseñar nuestro primer programa Java.

Visibilidad de los nombres

Uno de los problemas en cualquier lenguaje de programación es el de control de los nombres. Si utilizamos un nombre en un módulo de un programa y otro programador emplea el mismo nombre en otro módulo, ¿cómo podemos distinguir un nombre del otro y cómo podemos evitar la "colisión" de los dos nombres? En C, este problema es especialmente significativo, porque cada programa es, a menudo, un conjunto manejable de nombres. Las clases C++ (en las que se basan las clases Java) anidan las funciones dentro de clases para que no puedan colisionar con los nombres de función anidados dentro de otras clases. Sin embargo, C++ sigue permitiendo utilizar datos globales y funciones globales, así que las colisiones continúan siendo posibles. Para resolver este problema, C++ introdujo los denominados *espacios de nombres* utilizando palabras clave adicionales.

Java consiguió evitar todos estos problemas adoptando un enfoque completamente nuevo. Para generar un nombre no ambiguo para una biblioteca, los creadores de Java emplean los nombres de dominio de Internet en orden inverso, ya que se garantiza que los nombres de dominio son unívocos. Puesto que mi nombre de dominio es **MindView.net**, una biblioteca de utilidades llamada foibles se denominaría, por ejemplo, **net.mindview.utility.foibles**. Después del nombre de dominio invertido, los puntos tratan de representar subdirectorios.

En Java 1.0 y Java 1.1, las extensiones de dominio **com**, **edu**, **org**, **net**, etc., se escribían en mayúsculas por convenio, por lo que el nombre de la biblioteca sería **NET.mindview.utility.foibles**. Sin embargo, durante el desarrollo de Java 2 se descubrió que esto producía problemas, por lo que ahora los nombres completos de paquetes se escriben en minúsculas.

Este mecanismo implica que todos los archivos se encuentran, automáticamente, en sus propios espacios de nombres y que cada clase dentro de un archivo tiene un identificador unívoco; el lenguaje se encarga de evitar automáticamente las colisiones de nombres.

Utilización de otros componentes

Cada vez que se quiera utilizar una clase predefinida en un programa, el compilador debería saber cómo localizarla. Por supuesto, puede que la clase ya exista en el mismo archivo de código fuente desde el que se la está invocando. En ese caso, basta con utilizar de manera directa la clase, aún cuando esa clase no esté definida hasta más adelante en el archivo (Java elimina los problemas denominados de "referencia anticipada").

¿Qué sucede con las clases almacenadas en algún otro archivo? Cabría esperar que el compilador fuera lo suficientemente inteligente como para localizar la clase, pero hay un pequeño problema. Imagine que queremos emplear una clase con un nombre concreto, pero que existe más de una definición para dicha clase (probablemente, definiciones distintas). O peor, imagine que está escribiendo un programa y que a medida que lo escribe añade una nueva clase a la biblioteca que entra en conflicto con el nombre de otra clase ya existente.

Para resolver este problema, es preciso eliminar todas las ambigüedades potenciales. Esto se consigue informando al compilador Java de exactamente qué clases se desea emplear, utilizando para ello la palabra clave **import**. **import** le dice al compilador que cargue un paquete, que es una biblioteca de clases (en otros lenguajes, una biblioteca podría estar compuesta por funciones y datos, así como clases, pero recuerde que todo el código Java debe escribirse dentro de una clase).

La mayor parte de las veces utilizaremos componentes de las bibliotecas estándar Java incluidas con el compilador. Con estos componentes, no es necesario preocuparse sobre los largos nombres de dominio invertidos, basta con decir, por ejemplo:

```
import java.util.ArrayList;
```

para informar al compilador que queremos utilizar la clase **ArrayList** de Java. Sin embargo, **util** contiene diversas clases, y podemos usar varias de ellas sin declararlas todas ellas explícitamente. Podemos conseguir esto fácilmente utilizando '*' como comodín:

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

Resulta más común importar una colección de clases de esta forma que importar las clases individualmente.

La palabra clave static

Normalmente, cuando creamos una clase, lo que estamos haciendo es describir el aspecto de los objetos de esa clase y el modo en que éstos van a comportarse. En la práctica, no obtenemos ningún objeto hasta que creamos uno empleando **new**, que es el momento en que se asigna el almacenamiento y los métodos del objeto pasan a estar disponibles.

Existen dos situaciones en las que esta técnica no resulta suficiente. Una de ellas es cuando deseamos disponer de un único espacio de almacenamiento para un campo concreto, independientemente del número de objetos de esa clase que se cree, o incluso si no se crea ningún objeto de esa clase. La otra situación es cuando hace falta un método que no esté asociado con ningún objeto concreto de esa clase; en otras palabras, cuando necesitamos un método al que podamos invocar incluso aunque no se cree ningún objeto.

Podemos conseguir ambos efectos utilizando la palabra clave **static**. Cuando decimos que algo es **static**, quiere decir que ese campo o método concretos no están ligados a ninguna instancia concreta de objeto de esa clase. Por tanto, aún cuando nunca creáramos un objeto de esa clase, podríamos de todos modos invocar el método **static** o acceder a un campo **static**. Con los campos y métodos normales, que no tienen el atributo **static**, es preciso crear un objeto y usar ese objeto para acceder al campo o al método, ya que los campos y métodos que no son de tipo **static** deben conocer el objeto en particular con el que estén trabajando.[5]

Algunos lenguajes orientados a objetos utilizan los términos *datos de la clase* y *métodos de la clase*, para indicar que esos datos y métodos sólo existen para la clase como un todo, y no para ningún objeto concreto de esa clase. En ocasiones, en la literatura técnica relacionada con Java también se utilizan esos términos.

Para hacer que un campo o un método sea **static**, basta con colocar dicha palabra clave antes de la definición. Por ejemplo, el siguiente código generaría un campo **static** y le inicializaría:

```
class StaticTest {
  static int i = 47;
}
```

Ahora, aunque creáramos dos objetos **StaticTest**, existiría un único espacio de almacenamiento para **StaticTest.i**. Ambos objetos compartirían el mismo campo **i**. Considere el fragmento de código siguiente:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

En este punto, tanto **st1.i** como **st2.i** tienen el mismo valor, 47, ya que ambos hacen referencia a la misma posición de memoria.

---

[5] Por supuesto, puesto que los métodos **static** no necesitan que se cree ningún objeto antes de utilizarlos, no pueden acceder *directamente* a ningún miembro o método que no sea **static**, por el procedimiento de invocar simplemente esos otros miembros sin hacer referencia a un objeto nominado (ya que los miembros y métodos que no son **static** deben estar asociados con un objeto concreto).

Hay dos formas de hacer referencia a una variable **static**. Como se indica en el ejemplo anterior, se la puede expresar a través de un objeto, escribiendo por ejemplo, **st2.i**. Podemos hacer referencia a ella directamente a través del nombre de la clase, lo que no puede hacerse con ningún nombre que no sea de tipo **static**.

```
StaticTest.i++;
```

El operador ++ incrementa en una unidad la variable. En este punto, tanto **st1.i** como **st2.i** tendrán el valor 48.

La forma preferida de hacer referencia a una variable **static** es utilizando el nombre de la clase. Esto no sólo permite enfatizar la naturaleza de tipo **static** de la variable, sino que en algunos casos proporciona al compilador mejores oportunidades para la optimización.

A los métodos **static** se les aplica una lógica similar. Podemos hacer referencia a un método **static** a través de un objeto, al igual que con cualquier otro método, o bien mediante la sintaxis adicional especial **NombreClase.metodo( )**. Podemos definir un método **static** de manera a similar:

```
class Incrementable {
  static void increment() { StaticTest.i++; }
}
```

Podemos ver que el método **increment( )** de **Incrementable** incrementa el dato **i** de tipo **static** utilizando el operador ++. Podemos invocar **increment( )** de la forma típica a través de un objeto:

```
Incrementable sf = new Incrementable();
sf.increment();
```

O, dado que **increment( )** es un método **static**, podemos invocarlo directamente a través de su clase:

```
Incrementable.increment();
```

Aunque **static**, cuando se aplica a un campo, modifica de manera evidente la forma en que se crean los datos (se crea un dato global para la clase, a diferencia de los campos que no son **static**, para los que se crea un campo para cada objeto), cuando se aplica esa palabra clave a un método no es tan dramático. Un uso importante de **static** para los métodos consiste en permitirnos invocar esos métodos sin necesidad de crear un objeto. Esta característica resulta esencial, como veremos, al definir el método **main( )**, que es el punto de entrada para ejecutar una aplicación.

Nuestro primer programa Java

Finalmente, vamos a ver nuestro primer programa completo. Comenzaremos imprimiendo una cadena de caracteres y a continuación la fecha, empleando la clase **Date** de la biblioteca estándar de Java.

```
// HelloDate.java
import java.util.*;

public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Al principio de cada archivo de programa, es necesario incluir las instrucciones **import** necesarias para cargar las clases adicionales que se necesiten para el código incluido en ese archivo. Observe que hemos dicho "clases adicionales". La razón es que existe una cierta biblioteca de clases que se carga automáticamente en todo archivo Java: **java.lang**. Inicie el explorador web y consulte la documentación de Sun (si no ha descargado la documentación del kit JDK de *http://java.sun.com*, hágalo ahora.[6] Observe que esta documentación no se suministra con el JDK; es necesario descargarla por separado). Si consulta la lista de paquetes, podrá ver todas las diferentes bibliotecas de clases incluidas con Java. Seleccione **java.lang**; esto hará que se muestre una lista de todas clases que forman parte de esa biblioteca. Puesto que **java.lang** está implícitamente

[6] El compilador Java y la documentación suministrados por Sun tienden a cambiar de manera frecuente, y el mejor lugar para obtenerlos es directamente en el sitio web de Sun. Descargándose esos archivos podrá disponer siempre de la versión más reciente.

en todo archivo de código Java, dichas clases estarán disponibles de manera automática. No existe ninguna clase **Date** en **java.lang**, lo que quiere decir que es preciso importar otra biblioteca para usar dicha clase. Si no sabe la biblioteca en la que se encuentra una clase concreta o si quiere ver todas las clases disponibles, puede seleccionar "Tree" en la documentación de Java. Con eso, podrá localizar todas las clases incluidas con Java. Entonces, utilice la función "Buscar" del explorador para encontrar **Date**. Cuando lo haga, podrá ver que aparece como **java.util.Date**, lo que nos permite deducir que se encuentra en la biblioteca **util** y que es necesario emplear la instrucción **import java.util.*** para usar **Date**.

Si vuelve atrás y selecciona **java.lang** y luego **System**, podrá ver que la clase **System** tiene varios campos; si selecciona **out**, descubrirá que se trata de un objeto **static PrintStream**. Puesto que es de tipo **static**, no es necesario crear ningún objeto empleando **new**. El objeto **out** está siempre disponible, por lo que podemos usarlo directamente. Lo que puede hacerse con este objeto **out** está determinado por su tipo: **PrintStream**. En la descripción se muestra **PrintStream** como un hipervínculo, por lo que al hacer clic sobre él podrá acceder a una lista de todos los métodos que se pueden invocar para **PrintStream**. Son bastantes métodos, y hablaremos de ellos más adelante en el libro. Por ahora, el único que nos interesa es **println( )**, que en la práctica significa "imprime en la consola lo que te voy a pasar y termina con un carácter de avance de línea". Así, en cualquier programa Java podemos escribir algo como esto:

```
System.out.println("Una cadena de caracteres");
```

cuando queramos mostrar información a través de la consola.

El nombre de la clase coincide con el nombre del archivo. Cuando estemos creando un programa independiente como éste, una de las clases del archivo deberá tener el mismo nombre que el propio archivo (el compilador se quejará si no lo hacemos así). Dicha clase debe contener un método denominado **main( )** con la siguiente signatura y tipo de retorno:

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

La palabra clave **public** quiere decir que el método está disponible para todo el mundo (lo que describiremos en detalle en el Capítulo 6, *Control de acceso*). El argumento de **main( )** es una matriz de objetos **String**. En este programa, no se emplean los argumentos (**args**), pero el compilador Java insiste en que se incluya ese parámetro, ya que en él se almacenarán los argumentos de la línea de comandos.

La línea que imprime la fecha es bastante interesante:

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

El argumento es un objeto **Date** que sólo se ha creado para enviar su valor (que se convierte automáticamente al tipo **String**) a **println( )**. Una vez finalizada esta instrucción, dicho objeto **Date** resulta innecesario, y el depurador de memoria podrá encargarse de él en cualquier momento. Nosotros no necesitamos preocuparnos de borrar el objeto.

Al examinar la documentación del JDK en *http://java.sun.com*, podemos ver que **System** tiene muchos métodos que nos permiten producir muchos efectos interesantes (una de las características más potentes de Java es su amplio conjunto de bibliotecas estándar). Por ejemplo:

```
//: object/ShowProperties.java

public class ShowProperties {
  public static void main(String[] args) {
    System.getProperties().list(System.out);
    System.out.println(System.getProperty("user.name"));
    System.out.println(
      System.getProperty("java.library.path"));
  }
} ///:~
```

La primera línea de **main( )** muestra todas las "propiedades" del sistema en que se está ejecutando el programa, por lo que nos proporciona la información del entorno. El método **list( )** envía los resultados a su argumento, **System.out**. Más adelante en el libro veremos que los resultados pueden enviarse a cualquier parte, como por ejemplo, a un archivo. También podemos consultar una propiedad específica, por ejemplo, en este caso, el nombre de usuario y **java.library.path** (un poco más adelante explicaremos los comentarios bastante poco usuales situados al principio y al final de este fragmento de código).

Compilación y ejecución

Para compilar y ejecutar este programa, y cualquier otro programa de este libro, en primer lugar hay que tener un entorno de programación Java. Hay disponibles diversos entornos de programación Java de otros fabricantes, pero en el libro vamos a suponer que el lector está utilizando el kit de desarrollo JDK (Java Developer's Kit) de Sun, el cual es gratuito. Si está utilizando otro sistema de desarrollo[7], tendrá que consultar la documentación de dicho sistema para ver cómo compilar y ejecutar los programas.

Acceda a Internet y vaya a *http://java.sun.com*. Allí podrá encontrar información y vínculos que le llevarán a través del proceso de descarga e instalación del JDK para su plataforma concreta.

Una vez instalado el JDK, y una vez modificada la información de ruta de la computadora para que ésta pueda encontrar **javac** y **java**, descargue y desempaquete el código fuente correspondiente a este libro (puede encontrarlo en *www.MindView.net*). Esto creará un subdirectorio para cada capítulo del libro. Acceda al subdirectorio **objects** y escriba:

```
javac HelloDate.java
```

Este comando no debería producir ninguna respuesta. Si obtiene algún tipo de mensaje de error querrá decir que no ha instalado el JDK correctamente y tendrá que investigar cuál es el problema.

Por el contrario, si lo único que puede ver después de ejecutar el comando es una nueva línea de comandos, podrá escribir:

```
java HelloDate
```

y obtendrá el mensaje y la fecha como salida.

Puede utilizar este proceso para compilar y ejecutar cada uno de los programas de este libro. Sin embargo, podrá ver que el código fuente de este libro también dispone de un archivo denominado **build.xml** en cada capítulo, que contiene comandos "Ant" para construir automáticamente los archivos correspondientes a ese capítulo. Los archivos de generación y Ant (incluyendo las instrucciones para descargarlos) se describen más en detalle en el suplemento que podrá encontrar en *http://MindView.net/Books/BetterJava*, pero una vez que haya instalado Ant (desde *http://jakarta.apache.org/ant*) basta con que escriba '**ant**' en la línea de comandos para compilar y ejecutar los programas de cada capítulo. Si no ha instalado Ant todavía, puede escribir los comandos **javac** y **java** a mano.

Comentarios y documentación embebida

Existen dos tipos de comentarios en Java. El primero de ellos es el comentario de estilo C heredado de C++. Estos comentarios comienzan con /* y continúan, posiblemente a lo largo de varias línea, hasta encontrar */. Muchos programadores empiezan cada línea de un comando multilínea con un *, por lo que resulta bastante común ver comentarios como éste:

```
/* Este comentario
 * ocupa varias
 * líneas
 */
```

Recuerde, sin embargo, que en todo lo que hay entre /* y */ se ignora, por lo que no habría ninguna diferencia si dijéramos:

```
/* Este comentario ocupa
varias líneas */
```

El segundo tipo de comentario proviene de C++. Se trata del comentario monolínea que comienza con // y continúa hasta el final de la línea. Este tipo de comentario es bastante cómodo y se suele utilizar a menudo debido a su sencillez. No es necesario desplazarse de un sitio a otro del teclado para encontrar el carácter / y luego el carácter * (en su lugar, se pulsa la misma tecla dos veces), y tampoco es necesario cerrar el comentario. Así que resulta bastante habitual encontrar comentarios de este estilo:

```
// Éste es un comentario monolínea.
```

[7] El compilador "jikes" de IBM es una alternativa bastante común, y es significativamente más rápido que Java C de Sun (aunque se están construyendo grupos de archivos utilizando *Ant*, la diferencia no es muy significativa). También hay diversos proyectos de código fuente abierto para crear compiladores Java, entornos de ejecución y bibliotecas.

Documentación mediante comentarios

Posiblemente el mayor problema de la tarea de documentar el código es el de mantener dicha documentación. Si la documentación y el código están separados, resulta bastante tedioso modificar la documentación cada vez que se cambia el código. La solución parece simple: integrar el código con la documentación. La forma más fácil de hacer esto es incluir todo en un mismo archivo. Sin embargo, para ello es necesario disponer de una sintaxis especial de comentarios que permita marcar la documentación, así como de una herramienta para extraer dichos comentarios y formatearlos de manera útil. Esto es precisamente lo que Java ha hecho.

La herramienta para extraer los comentarios se denomina *Javadoc*, y forma parte de la instalación del JDK. Utiliza parte de la tecnología del compilador Java para buscar los marcadores especiales de comentarios incluidos en el programa. No sólo extrae la información señalada por dichos marcadores, sino que también extrae el nombre de la clase o el nombre del método asociado al comentario. De esta forma, podemos generar una documentación de programa bastante digna con una cantidad mínima de trabajo.

La salida de Javadoc es un archivo HTML que se puede examinar con el explorador web. Javadoc permite crear y mantener un único archivo fuente y generar automáticamente una documentación bastante útil. Gracias a Javadoc, disponemos de un estándar bastante sencillo para la creación de documentación, por lo que podemos esperar, o incluso exigir, que todas las bibliotecas Java estén documentadas.

Además, podemos escribir nuestras propias rutinas de gestión Javadoc, denominadas *doclets*, si queremos realizar operaciones especiales con la información procesada por Javadoc (por ejemplo, para generar la salida en un formato distinto). Los *doclets* se explican en el suplemento que podrá encontrar en *http://MindView.net/Books/BetterJava*.

Lo que sigue es simplemente una introducción y una breve panorámica de los fundamentos de Javadoc. En la documentación del JDK podrá encontrar una descripción más exhaustiva. Cuando desempaquete la documentación, vaya al subdirectorio "tooldocs" (o haga clic en el vínculo "tooldocs").

Sintaxis

Todos los comandos de Javadoc están incluidos en comentarios que tienen el marcador /**. Esos comentarios terminan con */ como es habitual. Existen dos formas principales de utilizar Javadoc: mediante HTML embebido o mediante "marcadores de documentación". Los *marcadores de documentación autónomos* son comandos que comienzan con '@' y se incluyen al principio de una línea de comentarios (sin embargo, un carácter '*' inicial será ignorado). Los *marcadores de documentación en línea* pueden incluirse en cualquier punto de un comentario Javadoc y también comienzan con '@', pero están encerrados entre llaves.

Existen tres "tipos" de documentación de comentarios, que se corresponden con el elemento al que el comentario precede: una clase, un campo o un método. En otras palabras, el comentario de clase aparece justo antes de la definición de la clase, el comentario de campo aparece justo antes de la definición de un campo y el comentario de un método aparece justo antes de la definición del mismo. He aquí un sencillo ejemplo:

```
//: object/Documentation1.java
/** Comentario de clase */
public class Documentation1 {
  /** Comentario de campo */
  public int i;
  /** Comentario de método */
  public void f() {}
} ///:~
```

Observe que Javadoc sólo procesará la documentación de los comentarios para los miembros **public** y **protected**. Los comentarios para los miembros **private** y los miembros con acceso de paquete (*véase* el Capítulo 6, *Control de acceso*) se ignoran, no generándose ninguna salida para ellos (sin embargo, puede utilizar el indicador **private** para incluir también la documentación de los miembros **private**). Esto tiene bastante sentido, ya que sólo los miembros de tipo **public** y **protected** están disponibles fuera del archivo, por lo que esto se ajusta a la perspectiva del programador de clientes.

La salida del código anterior es un archivo HTML que tiene el mismo formato estándar que el resto de la documentación Java, por lo que los usuarios estarán acostumbrados a ese formato y podrán navegar fácilmente a través de las clases que

hayamos definido. Merece la pena introducir el ejemplo de código anterior, procesarlo mediante Javadoc y observar el archivo HTML resultante, para ver el tipo de salida que se genera.

HTML embebido

Javadoc pasa los comentarios HTML al documento HTML generado. Esto nos permite utilizar completamente las características HTML; sin embargo, el motivo principal de uso de ese lenguaje es dar formato al código, como por ejemplo en:

```
//: object/Documentation2.java
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
///:~
```

También podemos usar código HTML como en cualquier otro documento web, para dar formato al texto normal de las descripciones:

```
//: object/Documentation3.java
/**
* Se puede <em>incluso</em> insertar una lista:
* <ol>
* <li> Elemento uno
* <li> Elemento dos
* <li> Elemento tres
* </ol>
*/
///:~
```

Observe que, dentro del comentario de documentación, los asteriscos situados al principio de cada línea son ignorados por Javadoc, junto con los espacios iniciales. Javadoc reformatea todo para que se adapte al estilo estándar de la documentación. No utilice encabezados como **<h1>** o **<hr>** en el HTML embebido, porque Javadoc inserta sus propios encabezados y los que nosotros incluyamos interferirán con ellos.

Puede utilizarse HTML embebido en todos los tipos de comentarios de documentación: de clase, de campo y de método.

Algunos marcadores de ejemplo

He aquí algunos de los marcadores Javadoc disponibles para la documentación de código. Antes de tratar de utilizar Javadoc de forma seria, consulte la documentación de referencia de Javadoc dentro de la documentación del JDK, para ver las diferentes formas de uso de Javadoc.

@see

Este marcador permite hacer referencia a la documentación de otras clases. Javadoc generará el código HTML necesario, hipervinculando los marcadores **@see** a los otros fragmentos de documentación. Las formas posibles de uso de este marcador son:

```
@see nombreclase
@see nombreclase-completamente-cualificado
@see nombreclase-completamente-cualificado #nombre-método
```

Cada uno de estos marcadores añade una entrada "See Also" (Véase también) hipervinculada al archivo de documentación generado. Javadoc no comprueba los hipervínculos para ver si son válidos.

{@link *paquete.clase#miembro etiqueta* }

Muy similar a **@see**, excepto porque se puede utilizar en línea y emplea la *etiqueta* como texto del hipervínculo, en lugar de "See Also".

{@docRoot}

Genera la ruta relativa al directorio raíz de la documentación. Resulta útil para introducir hipervínculos explícitos a páginas del árbol de documentación.

{@inheritDoc}

Este indicador hereda la documentación de la clase base más próxima de esta clase, insertándola en el comentario del documento actual.

@version

Tiene la forma:

```
@version información-versión
```

en el que **información-versión** es cualquier información significativa que queramos incluir. Cuando se añade el indicador **@version** en la línea de comandos Javadoc, la información de versión será mostrada de manera especial en la documentación HTML generada.

@author

Tiene la forma:

```
@author información-autor
```

donde **información-autor** es, normalmente, nuestro nombre, aunque también podríamos incluir nuestra dirección de correo electrónico o cualquier otra información apropiada. Cuando se incluye el indicador **@author** en la línea de comandos Javadoc, se inserta la información de autor de manera especial en la documentación HTML generada.

Podemos incluir múltiples marcadores de autor para incluir una lista de autores, pero es preciso poner esos marcadores de forma consecutiva. Toda la información de autor se agrupará en un único párrafo dentro del código HTML generado.

@since

Este marcador permite indicar la versión del código en la que se empezó a utilizar una característica concreta. En la documentación HTML de Java, se emplea este marcador para indicar qué versión del JDK se está utilizando.

@param

Se utiliza para la documentación de métodos y tiene la forma:

```
@param nombre-parámetro descripción
```

donde **nombre-parámetro** es el identificador dentro de la lista de parámetros del método y **descripción** es un texto que puede continuar en las líneas siguientes. La descripción se considera terminada cuando se encuentra un nuevo marcador de documentación. Puede haber múltiples marcadores de este tipo, normalmente uno por cada parámetro.

@return

Se utiliza para documentación de métodos y su formato es el siguiente:

```
@return descripción
```

donde **descripción** indica el significado del valor de retorno. Puede continuar en las líneas siguientes.

@throws

Las excepciones se estudian en el Capítulo 12, *Tratamiento de errores mediante excepciones*. Por resumir, se trata de objetos que pueden ser "generados" en un método si dicho método falla. Aunque sólo puede generarse un objeto excepción cada vez que invocamos un método, cada método concreto puede generar diferentes tipos de excepciones, todas las cuales habrá que describir, por lo que el formato del marcador de excepción es:

```
@throws nombre-clase-completamente-cualificado descripción
```

donde *nombre-clase-completamente-cualificado* proporciona un nombre no ambiguo de una clase de excepción definida en alguna otra parte y *descripción* (que puede ocupar las siguientes líneas) indica la razón por la que puede generarse este tipo concreto de excepción después de la llamada al método.

@deprecated

Se utiliza para indicar características que han quedado obsoletas debido a la introducción de alguna otra característica mejorada. Este marcador indicativo de la obsolescencia recomienda que no se utilice ya esa característica concreta, ya que es probable que en el futuro sea eliminada. Un método marcado como **@deprecated** hace que el compilador genere una advertencia si se usa. En Java SE5, el marcador Javadoc **@deprecated** ha quedado sustituido por la *anotación* **@Deprecated** (hablaremos de este tema en el Capítulo 20, *Anotaciones*).

Ejemplo de documentación

He aquí de nuevo nuestro primer programa Java, pero esta vez con los comentarios de documentación incluidos:

```
//: object/HelloDate.java
import java.util.*;

/** El primer programa de ejemplo del libro.
 * Muestra una cadena de caracteres y la fecha actual.
 * @author Bruce Eckel
 * @author www.MindView.net
 * @version 4.0
*/
public class HelloDate {
  /** Punto de entrada a la clase y a la aplicación.
   * @param args matriz de argumentos de cadena
   * @throws exceptions No se generan excepciones
  */
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:36 MDT 2005
*///:~
```

La primera línea del archivo utiliza una técnica propia del autor del libro que consiste en incluir '//:' como marcador especial en la línea de comentarios que contiene el nombre del archivo fuente. Dicha línea contiene la información de ruta del archivo (**object** indica este capítulo) seguida del nombre del archivo. La última línea también termina con un comentario, y éste ('///:~') indica el final del listado de código fuente, lo que permite actualizarlo automáticamente dentro del texto de este libro después de comprobarlo con un compilador y ejecutarlo.

El marcador **/* Output:** indica el comienzo de la salida que generará este archivo. Con esta forma concreta, se puede comprobar automáticamente para verificar su precisión. En este caso, el texto **(55% match)** indica al sistema de pruebas que la salida será bastante distinta en cada sucesiva ejecución del programa, por lo que sólo cabe esperar un 55 por ciento de correlación con la salida que aquí se muestre. La mayoría de los ejemplos del libro que generan una salida contendrán dicha salida comentada de esta forma, para que el lector pueda ver la salida y saber que lo que ha obtenido es correcto.

Estilo de codificación

El estilo descrito en el manual de convenios de código para Java, *Code Conventions for the Java Programming Language*[8], consiste en poner en mayúscula la primera letra del nombre de una clase. Si el nombre de la clase está compuesto por varias

[8] Para ahorrar espacio tanto en el libro como en las presentaciones de los seminarios no hemos podido seguir todas las directrices que se marcan en ese texto, pero el lector podrá ver que el estilo empleado en el libro se ajusta lo máximo posible al estándar recomendado en Java.

palabras, éstas se escriben juntas (es decir, no se emplean guiones bajos para separarlas) y se pone en mayúscula la primera letra de cada una de las palabras integrantes, como en:

```
class AllTheColorsOfTheRainbow { // ...
```

Para casi todos los demás elementos, como por ejemplo nombres de métodos, campos (variables miembro) y referencias a objetos, el estilo aceptado es igual que para las clases, *salvo* porque la primera letra del identificador se escribe en minúscula, por ejemplo:

```
class AllTheColorsOfTheRainbow {
  int anIntegerRepresentingColors;
  void changeTheHueOfTheColor(int newHue) {
    // ...
  }
  // ...
}
```

Recuerde que el usuario se verá obligado a escribir estos nombres tan largos, así que procure que su longitud no sea excesiva.

El código Java que podrá encontrar en las bibliotecas Sun también se ajusta a las directrices de ubicación de llaves de apertura y cierre que hemos utilizado en este libro.

Resumen

El objetivo de este capítulo es explicar los conceptos mínimos sobre Java necesarios para entender cómo se escribe un programa sencillo. Hemos expuesto una panorámica del lenguaje y algunas de las ideas básicas en que se fundamenta. Sin embargo, los ejemplos incluidos hasta ahora tenían todos ellos la forma "haga esto, luego lo otro y después lo de más allá". En los dos capítulos siguientes vamos a presentar los operadores básicos utilizados en los programas Java, y luego mostraremos cómo controlar el flujo del programa.

Ejercicios

Normalmente, distribuiremos los ejercicios por todo el capítulo, pero en éste estábamos aprendiendo a escribir programas básicos, por lo que decidimos dejar los ejercicios para el final.

El número entre paréntesis incluido detrás del número de ejercicio es un indicador de la dificultad del ejercicio dentro de una escala de 1-10.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, a la venta en *www.MindView.net*.

Ejercicio 1: (2) Cree una clase que contenga una variable **int** y otra **char** que no estén inicializadas e imprima sus valores para verificar que Java se encarga de realizar una inicialización predeterminada.

Ejercicio 2: (1) A partir del ejemplo **HelloDate.java** de este capítulo, cree un programa "hello, world" que simplemente muestre dicha frase. Sólo necesitará un método dentro de la clase, (el método "main" que se ejecuta al arrancar el programa). Acuérdese de definir este método como **static** y de incluir la lista de argumentos, incluso aunque no la vaya a utilizar. Compile el programa con **javac** y ejecútelo con **java**. Si está utilizando otro entorno de desarrollo distinto de JDK, averigüe cómo compilar y ejecutar programas en dicho entorno.

Ejercicio 3: (1) Recopile los fragmentos de código relacionados con **ATypeName** y transfórmelos en un programa que se pueda compilar y ejecutar.

Ejercicio 4: (1) Transforme los fragmentos de código **DataOnly** en un programa que se pueda compilar y ejecutar.

Ejercicio 5: (1) Modifique el ejercicio anterior de modo que los valores de los datos en **DataOnly** se asignen e impriman en **main( )**.

Ejercicio 6: (2) Escriba un programa que incluya e invoque el método **storage( )** definido como fragmento de código en el capítulo.

Ejercicio 7: (1) Transforme los fragmentos de código **Incrementable** en un programa funcional.

Ejercicio 8: (3) Escriba un programa que demuestre que, independientemente de cuántos objetos se creen de una clase concreta, sólo hay una única instancia de un campo **static** concreto definido dentro de esa clase.

Ejercicio 9: (2) Escriba un programa que demuestre que el mecanismo automático de conversión de tipos funciona para todos los tipos primitivos y sus envoltorios.

Ejercicio 10: (2) Escriba un programa que imprima tres argumentos extraídos de la línea de comandos. Para hacer esto, necesitará acceder con el índice correspondiente a la matriz de objetos **String** extraída de la línea de comandos.

Ejercicio 11: (1) Transforme el ejemplo **AllTheColorsOfTheRainbow** en un programa que se pueda compilar y ejecutar.

Ejercicio 12: (2) Localice el código para la segunda versión de **HelloDate.java**, que es el ejemplo simple de documentación mediante comentarios. Ejecute **Javadoc** con ese archivo y compruebe los resultados con su explorador web.

Ejercicio 13: (1) Procese **Documentation1.java**, **Documentation2.java** y **Documentation3.java** con **Javadoc**. Verifique la documentación resultante con su explorador web.

Ejercicio 14: (1) Añada una lista HTML de elementos a la documentación del ejercicio anterior.

Ejercicio 15: (1) Tome el programa del Ejercicio 2 y añádale comentarios de documentación. Extraiga esos comentarios de documentación **Javadoc** para generar un archivo HTML y visualícelo con su explorador web.

Ejercicio 16: (1) En el Capítulo 5, *Inicialización y limpieza*, localice el ejemplo **Overloading.java** y añada documentación de tipo Javadoc. Extraiga los comentarios de documentación **Javadoc** para generar un archivo HTML y visualícelo con su explorador web.