2: Todo es un objeto

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

Tanto C++ como Java son lenguajes híbridos, pero en Java los diseñadores pensaban que esa "hibridación" no era tan importante como lo era en C++. Un lenguaje híbrido permite múltiples estilos de programación; la razón por la que C++ es híbrido es soportar la compatibilidad hacia atrás con el lenguaje C. Dado que C++ es un superconjunto del lenguaje C, incluye muchas de las características no deseables de ese lenguaje, lo que puede provocar que algunos aspectos de C++ sean demasiado complicados.

El lenguaje Java asume que se desea llevar a cabo exclusivamente programación orientada a objetos. Esto significa que antes de empezar es necesario cambiar la forma de pensar hacia el mundo de la orientación a objetos (a menos que ya esté en él). El beneficio de este esfuerzo inicial es la habilidad para programar en un lenguaje que es más fácil de aprender y usar que otros muchos lenguajes de POO. En este capítulo, veremos los componentes básicos de un programa Java y aprenderemos que todo en Java es un objeto, incluido un programa Java.

Los objetos se manipulan mediante referencias

Cada lenguaje de programación tiene sus propios medios de manipular datos. Algunas veces, el programador debe ser consciente constantemente del tipo de manipulación que se está produciendo. ¿Se está manipulando directamente un objeto, o se está tratando con algún tipo de representación indirecta (un puntero en C o C++) que debe ser tratada con alguna sintaxis especial?

Todo esto se simplifica en Java. Todo se trata como un objeto, de forma que hay una única sintaxis consistente que se utiliza en todas partes. Aunque se *trata* todo como un objeto, el identificador que se manipula es una "referencia" a un objeto[1]. Se podría imaginar esta escena como si se tratara de

---

[1] Esto puede suponer un tema de debate. Existe quien piensa que "claramente, es un puntero", pero esto presupone una implementación subyacente. Además, las referencias de Java son mucho más parecidas en su sintaxis a las referencias de C++ que a punteros. En la primera edición del presente libro, el autor decidió inventar un nuevo término, "empuñadura" porque las referencias C++ y las referencias Java tienen algunas diferencias importantes. El autor provenía de C++ y no deseaba confundir a los programadores de C++ que supuestamente serían la mejor audiencia para Java. En la 2ª edición, el autor decidió que el término más comúnmente usado era el término "referencia", y que cualquiera que proviniera de C++ tendría que lidiar con mucho más que con la terminología de las referencias, por lo que podrán incorporarse sin problemas. Sin embargo, hay personas que no están de acuerdo siquiera con el término "referencia". El autor leyó una vez un libro en el que "era incorrecto decir que Java soporta el paso por referencia", puesto que los identificadores de objetos en Java (en concordancia con el citado autor) son *de hecho* "referencias a objetos". "Y (continúa el citado texto), todo se pasa *de hecho* por valor. Por tanto, si no se pasan parámetros por referencia, se está pasando una referencia a un objeto por valor". Se podría discutir la precisión de semejantes explicaciones, pero el autor considera que su enfoque simplifica el entendimiento del concepto sin herir a nadie (bueno, los abogados del lenguaje podrían decir que el autor miente, pero creo que la abstracción que se presenta es bastante apropiada).

una televisión (el objeto) con su mando a distancia (la referencia). A medida que se hace uso de la referencia, se está conectado a la televisión, pero cuando alguien dice "cambia de canal" o "baja el volumen", lo que se manipula es la referencia, que será la que manipule el objeto. Si desea moverse por la habitación y seguir controlando la televisión, se toma el mando a distancia (la referencia), en vez de la televisión.

Además, el mando a distancia puede existir por sí mismo, aunque no haya televisión. Es decir, el mero hecho de tener una referencia no implica necesariamente la existencia de un objeto conectado al mismo. De esta forma si se desea tener una palabra o frase, se crea una referencia **String**:

```
String s;
```

Pero esta sentencia *solamente* crea la referencia, y no el objeto. Si se decide enviar un mensaje a **s** en este momento, se obtendrá un error (en tiempo de ejecución) porque **s** no se encuentra, de hecho, vinculado a nada (no hay televisión). Una práctica más segura, por consiguiente, es inicializar la referencia en el mismo momento de su creación:

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

Sin embargo, esta sentencia hace uso de una característica especial de Java: las cadenas de texto pueden inicializarse con texto entre comillas. Normalmente, es necesario usar un tipo de inicialización más general para los objetos.

Uno debe crear todos los objetos

Cuando se crea una referencia, se desea conectarla con un nuevo objeto. Así se hace, en general, con la palabra clave **new**, que dice "Créame un objeto nuevo de ésos". Por ello, en el ejemplo anterior se puede decir:

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

Esto no sólo significa "Créame un nuevo **String**", sino que también proporciona información sobre *cómo* crear el **String** proporcionando una cadena de caracteres inicial.

Por supuesto, **String** no es el único tipo que existe. Java viene con una plétora de tipos predefinidos. Lo más importante es que uno puede crear sus propios tipos. De hecho, ésa es la actividad fundamental de la programación en Java, y es precisamente lo que se irá aprendiendo en este libro.

Dónde reside el almacenamiento

Es útil visualizar algunos aspectos relativos a cómo se van disponiendo los elementos al ejecutar el programa, y en particular, sobre cómo se dispone la memoria. Hay seis lugares diferentes en los que almacenar información:

1. **Registros.** Son el elemento de almacenamiento más rápido porque existen en un lugar distinto al de cualquier otro almacenamiento: dentro del procesador. Sin embargo, el número de registros está severamente limitado, de forma que los registros los va asignando el compilador en función de sus necesidades. No se tiene control directo sobre ellos, y tampoco hay ninguna evidencia en los programas de que los registros siquiera existan.

2. **La pila.** Reside en la memoria RAM (memoria de acceso directo) general, pero tiene soporte directo del procesador a través del *puntero de pila*. Éste se mueve hacia abajo para crear más memoria y de nuevo hacia arriba para liberarla. Ésta es una manera extremadamente rápida y eficiente de asignar espacio de almacenamiento, antecedido sólo por los registros. El compilador de Java debe saber, mientras está creando el programa, el tamaño exacto y la vida de todos los datos almacenados en la pila, pues debe generar el código necesario para mover el puntero hacia arriba y hacia abajo. Esta limitación pone límites a la flexibilidad de nuestros programas, de forma que mientras existe algún espacio de almacenamiento en la pila —referencias a objetos en particular— los propios objetos Java no serán ubicados en la pila.

3. **El montículo.** Se trata de un espacio de memoria de propósito general (ubicado también en el área RAM) en el que residen los objetos Java. Lo mejor del montículo es que, a diferencia de la pila, el compilador no necesita conocer cuánto espacio de almacenamiento necesita asignar al montículo o durante cuánto tiempo debe permanecer ese espacio dentro del montículo. Por consiguiente, manejar este espacio de almacenamiento proporciona una gran flexibilidad. Cada vez que se desee crear un objeto, simplemente se escribe el código, se crea utilizando la palabra **new**, y se asigna el espacio de almacenamiento en el montículo en el momento en que este código se ejecuta. Por supuesto hay que pagar un precio a cambio de esta flexibilidad: lleva más tiempo asignar espacio de almacenamiento del montículo que lo que lleva hacerlo en la pila (es decir, si se *pudieran* crear objetos en la pila en Java, como se hace en C++).

4. **Almacenamiento estático.** El término "estático" se utiliza aquí con el sentido de "con una ubicación/posición fija" (aunque también sea en RAM). El almacenamiento estático contiene datos que están disponibles durante todo el tiempo que se esté ejecutando un programa. Podemos usar la palabra clave **static** para especificar que un elemento particular de un objeto sea estático, pero los objetos en sí nunca se sitúan en el espacio de almacenamiento estático.

5. **Almacenamiento constante.** Los valores constantes se suelen ubicar directamente en el código del programa, que es seguro, dado que estos valores no pueden cambiar. En ocasiones, las constantes suelen ser *acordonadas* por sí mismas, de forma que puedan ser opcionalmente ubicadas en memoria de sólo lectura (ROM).

6. **Almacenamiento no-RAM.** Si los datos residen completamente fuera de un programa, pueden existir mientras el programa no se esté ejecutando, fuera del control de dicho programa. Los dos ejemplos principales de esto son los objetos de flujo de datos *(stream)*, que se convierten en flujos o corrientes de bytes, generalmente para ser enviados a otra máquina, y los *objetos persistentes*, que son ubicados en el disco para que mantengan su estado incluso cuando el programa ha terminado. El truco con estos tipos de almacenamiento es convertir los objetos en algo que pueda existir en otro medio, y que pueda así recuperarse en forma de objeto basado en RAM cuando sea necesario. Java proporciona soporte para *persistencia ligera*, y

las versiones futuras de Java podrían proporcionar soluciones aún más complejas para la persistencia.

Un caso especial: los tipos primitivos

Hay un grupo de tipos que tiene un tratamiento especial: se trata de los tipos "primitivos", que se usarán frecuentemente en los programas. La razón para el tratamiento especial es que crear un objeto con **new** —especialmente variables pequeñas y simples— no es eficiente porque **new** coloca el objeto en el montículo. Para estos tipos, Java vuelve al enfoque de C y C++. Es decir, en vez de crear la variable utilizando **new**, se crea una variable "automática" que *no es una referencia*. La variable guarda el valor, y se coloca en la pila para que sea más eficiente.

Java determina el tamaño de cada tipo primitivo. Estos tamaños no varían de una plataforma a otra como ocurre en la mayoría de los lenguajes. La invariabilidad de tamaño es una de las razones por las que Java es tan llevadero.

Tipo primitivo	Tamaño	Mínimo	Máximo	Tipo de envoltura
boolean	–	–	–	**Boolean**
char	16 bits	Unicode 0	Unicode 2^{16}-1	**Character**
byte	8 bits	-1^{28}	+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, de forma que es inútil tratar de utilizar tipos sin signo.

El tamaño del tipo **boolean** no está explícitamente definido; sólo se especifica que debe ser capaz de tomar los valores **true** o **false**.

Los tipos de datos primitivos también tienen clases "envoltura". Esto quiere decir que si se desea hacer un objeto no primitivo en el montículo para representar ese tipo primitivo, se hace uso del envoltorio asociado. Por ejemplo:

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

O también se podría utilizar:

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

Las razones para hacer esto se mostrarán más adelante en este capítulo.

Números de alta precisión

Java incluye dos clases para llevar a cabo aritmética de alta precisión: **BigInteger** y **BigDecimal**. Aunque estos tipos vienen a encajar en la misma categoría que las clases "envoltorio", ninguna de ellas tiene un tipo primitivo.

Ambas clases tienen métodos que proporcionan operaciones análogas que se lleven a cabo con tipos primitivos. Es decir, uno puede hacer con **BigInteger** y **BigDecimal** cualquier cosa que pueda hacer con un **int** o un **float**, simplemente utilizando llamadas a métodos en vez de operadores. Además, las operaciones serán más lentas dado que hay más elementos involucrados. Se sacrifica la velocidad en favor de la exactitud.

BigInteger soporta enteros de precisión arbitraria. Esto significa que uno puede representar valores enteros exactos de cualquier tamaño y sin perder información en las distintas operaciones.

BigDecimal es para números de coma flotante de precisión arbitraria; pueden usarse, por ejemplo, para cálculos monetarios exactos.

Para conocer los detalles de los constructores y métodos que pueden invocarse para estas dos clases, puede recurrirse a la documentación existente en línea.

Arrays en Java

Virtualmente, todos los lenguajes de programación soportan arrays. Utilizar arrays en C y C++ es peligroso porque los arrays no son sino bloques de memoria. Si un programa accede al array fuera del rango de su bloque de memoria o hace uso de la memoria antes de la inicialización (errores de programación bastante frecuentes) los resultados pueden ser impredecibles.

Una de los principales objetos de Java es la seguridad, de forma que muchos de los problemas habituales en los programadores de C y C++ no se repiten en Java. Está garantizado que un array en Java estará siempre inicializado, y que no se podrá acceder más allá de su rango. La comprobación de rangos se resuelve con una pequeña sobrecarga de memoria en cada array, además de verificar el índice en tiempo de ejecución, pero se asume que la seguridad y el incremento de productividad logrados merecen este coste.

Cuando se crea un array de objetos, se está creando realmente un array de referencias a los objetos, y cada una de éstas se inicializa automáticamente con un valor especial representado por la palabra clave **null**. Cuando Java ve un **null**, reconoce que la referencia en cuestión no está señalando ningún objeto. Debe asignarse un objeto a cada referencia antes de utilizarla, y si se intenta hacer uso de una referencia que aún vale **null**, se informará de que se ha dado un problema en tiempo de ejecución. Por consiguiente, en Java se evitan los errores típicos de los arrays.

Uno también puede crear un array de tipos primitivos. De nuevo, es el compilador el que garantiza la inicialización al poner a cero la memoria que ocupará ese array.

Se hablará del resto de arrays más detalladamente 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 ocupa una parte importante del esfuerzo de programación. ¿Cuánto dura una variable? Si se supone que uno va a destruirla, ¿cuándo debe hacerse? La confusión relativa a la vida de las variables puede conducir a un montón de fallos, y esta sección muestra cómo Java simplifica enormemente esto al hacer el trabajo de limpieza por ti.

Ámbito

La mayoría de lenguajes procedurales tienen el concepto de *alcance* o *ámbito*. Éste determina tanto la visibilidad como la vida de los nombres definidos dentro de ese ámbito. En C, C++ y Java, el ámbito se determina por la ubicación de llaves **{}**. Así, por ejemplo:

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

Una variable definida dentro de un ámbito solamente está disponible hasta que finalice su ámbito.

Las tabulaciones hacen que el código Java sea más fácil de leer. Dado que Java es un lenguaje de formato libre, los espacios extra, tabuladores y retornos de carro no afectan al programa resultante.

Fíjese que uno *no puede* hacer lo siguiente, incluso aunque sea legal en C y C++:

```
{
      int x = 12;
      {
            int x = 96; /* ilegal */
      }
}
```

El compilador comunicará que la variable **x** ya ha sido definida. Por consiguiente, la capacidad de C y C++ para "esconder" una variable de un ámbito mayor no está permitida, ya que los diseñadores de Java pensaron que conducía a programas confusos.

Ámbito de los objetos

Los objetos en Java no tienen la misma vida que los tipos primitivos. Cuando se crea un objeto Java haciendo uso de **new**, éste perdura hasta el final del ámbito. Por consiguiente, si se escribe:

```
{
     String s = new String (''un string'');
}    /* Fin del ámbito */
```

la referencia **s** desaparece al final del ámbito. Sin embargo, el objeto **String** al que apunta **s** sigue ocupando memoria. En este fragmento de código, no hay forma de acceder al objeto, pues la única referencia al mismo se encuentra fuera del ámbito. En capítulos posteriores se verá cómo puede pasarse la referencia al objeto, y duplicarla durante el curso de un programa.

Resulta que, dado que los objetos creados con **new** se mantienen durante tanto tiempo como se desee, en Java desaparecen un montón de posibles problemas propios de C++. Los problemas mayores parecen darse en C++ puesto que uno no recibe ningún tipo de ayuda del lenguaje para asegurarse de que los objetos estén disponibles cuando sean necesarios. Y lo que es aún más importante, en C++ uno debe asegurarse de destruir los objetos cuando se ha acabado con ellos.

Esto nos conduce a una cuestión interesante. Si Java deja los objetos vivos por ahí, ¿qué evita que se llene la memoria provocando que se detenga la ejecución del programa? Éste es exactamente el tipo de problema que ocurriría en C++. Es en este punto en el que ocurren un montón de cosas "mágicas". Java tiene un *recolector de basura*, que recorre todos los objetos que fueron creados con **new** y averigua cuáles no serán referenciados más. Posteriormente, libera la memoria de los que han dejado de ser referenciados, de forma que la memoria pueda ser utilizada por otros objetos. Esto quiere decir que no es necesario que uno se preocupe de reivindicar ninguna memoria. Simplemente se crean objetos, y cuando posteriormente dejan de ser necesarios, desaparecen por sí mismos. Esto elimina cierta clase de problemas de programación: el denominado "agujero de memoria", que se da cuando a un programador se le olvida liberar memoria.

Crear nuevos tipos de datos: clases

Si todo es un objeto, ¿qué determina qué apariencia tiene y cómo se comporta cada clase de objetos? O dicho de otra forma, ¿qué establece el *tipo* de un objeto? Uno podría esperar que haya una palabra clave "type", lo cual ciertamente hubiera tenido sentido. Sin embargo, históricamente, la mayoría de lenguajes orientados a objetos han hecho uso de la palabra clave **class** para indicar "Voy a decirte qué apariencia tiene un nuevo tipo de objeto". La palabra clave **class** (que se utilizará tanto

que no se pondrá en negrita a lo largo del presente libro) siempre va seguida del nombre del nuevo tipo. Por ejemplo:

```
class UnNombreDeTipo { /* Aquí va el cuerpo de la clase */ }
```

Esto introduce un nuevo tipo, de forma que ahora es posible crear un objeto de este tipo haciendo uso de la palabra clave **new**:

```
UnNombreDeTipo u = new UnNombreDeTipo ();
```

En **UnNombreDeTipo**, el cuerpo de la clase sólo consiste en un comentario (los asteriscos, las barras inclinadas y lo que hay dentro, que se discutirán más adelante en este capítulo), con lo que no hay demasiado que hacer con él. De hecho, uno no puede indicar que se haga mucho de nada (es decir, no se le puede mandar ningún mensaje interesante) hasta que se definan métodos para ella.

Campos y métodos

Cuando se define una clase (y todo lo que se hace en Java es definir clases, se hacen objetos de esas clases y se envían mensajes a esos objetos), es posible poner dos tipos de elementos en la nueva clase: datos miembros (denominados generalmente *campos*), y funciones miembros (típicamente llamados *métodos*). Un dato miembro es un objeto de cualquier tipo con el que te puedes comunicar a través de su referencia. También puede ser algún tipo primitivo (que no sea una referencia). Si es una referencia a un objeto, hay que inicializar esa referencia para conectarla a algún objeto real (utilizando **new**, como se ha visto antes) en una función especial denominada *constructor* (descrita completamente en el Capítulo 4). Si se trata de un tipo primitivo es posible inicializarla directamente en el momento de definir la clase (como se verá después, también es posible inicializar las referencias en este punto de la definición).

Cada objeto mantiene el espacio de almacenamiento necesario para todos sus datos miembro; éstos no son compartido con otros objetos. He aquí un ejemplo de una clase y algunos de sus datos miembros:

```
class SoloDatos {
      int i;
      float f;
      boolean b;
}
```

Esta clase *no hace* nada, pero es posible crear un objeto:

```
SoloDatos s = new SoloDatos();
```

Es posible asignar valores a los datos miembros, pero primero es necesario saber cómo hacer referencia a un miembro de un objeto. Esto se logra escribiendo el nombre de la referencia al objeto, seguido de un punto, y a continuación el nombre del miembro del objeto:

```
ReferenciaAObjeto.miembro
```

Por ejemplo:

```
s.i = 47;
s.f. = 1.1f;
f.b = false;
```

También es posible que un objeto pueda contener otros datos que se quieran modificar. Para ello, hay que seguir "conectando los puntos". Por ejemplo:

```
miAvion.tanqueIzquierdo.capacidad = 100;
```

La clase **SoloDatos** no puede hacer nada que no sea guardar datos porque no tiene funciones miembro (métodos). Para entender cómo funcionan los métodos, es necesario entender los *parámetros* y *valores de retorno*, que se describirán en breve.

Valores por defecto para los miembros primitivos

Cuando un tipo de datos primitivo es un miembro de una clase, se garantiza que tenga un valor por defecto siempre que no se inicialice:

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

Debe destacarse que los valores por defecto son los que Java garantiza cuando se usa la variable *como miembro de una clase*. Esto asegura que las variables miembro de tipos primitivos siempre serán inicializadas (algo que no ocurre en C++), reduciendo una fuente de errores. Sin embargo, este valor inicial puede no ser correcto o incluso legal dentro del programa concreto en el que se esté trabajando. Es mejor inicializar siempre todas las variables explícitamente.

Esta garantía no se aplica a las variables "locales" —aquellas que no sean campos de clases. Por consiguiente, si dentro de una definición de función se tiene:

```
int x;
```

Entonces **x** tomará algún valor arbitrario (como en C y C++); no se inicializará automáticamente a cero. Cada uno es responsable de asignar un valor apropiado a la variable **x** antes de usarla. Si uno se olvida, Java seguro que será mejor que C++: se recibirá un error en tiempo de compilación indicando que la variable debería haber sido inicializada. (Muchos compiladores de C++ advertirán sobre variables sin inicializar, pero en Java éstos se presentarán como errores.)

Métodos, parámetros y valores de retorno

Hasta ahora, el término *función* se ha utilizado para describir una subrutina con nombre. El término que se ha usado más frecuentemente en Java es *método*, al ser "una manera de hacer algo". Si se desea, es posible seguir pensando en funciones. Verdaderamente sólo hay una diferencia sintáctica, pero de ahora en adelante se usará el término "método" en lugar del término "función".

Los métodos en Java determinan los mensajes que puede recibir un objeto. En esta sección se aprenderá lo simple que es definir un método.

Las partes fundamentales de un método son su nombre, sus parámetros, el tipo de retorno y el cuerpo. He aquí su forma básica:

```
tipoRetorno nombreMetodo ( /* lista de parámetros */ ) {
      /* Cuerpo del método */
}
```

El tipo de retorno es el tipo del valor que surge del método tras ser invocado. La lista de parámetros indica los tipos y nombres de las informaciones que es necesario pasar a ese método. Cada método se identifica unívocamente mediante el nombre del método y la lista de parámetros.

En Java los métodos pueden crearse como parte de una clase. Es posible que un método pueda ser invocado sólo por un objeto[2], y ese objeto debe ser capaz de llevar a cabo esa llamada al método. Si se invoca erróneamente a un método de un objeto, se generará un error en tiempo de compilación. Se invoca a un método de un objeto escribiendo el nombre del objeto seguido de un punto y el nombre del método con su lista de argumentos, como: **nombreObjeto.nombreMetodo(arg1, arg2, arg3)**. Por ejemplo, si se tiene un método **f( )** que no recibe ningún parámetro y devuelve un dato de tipo **int**, y si se tiene un objeto **a** para el que puede invocarse a **f( )**, es posible escribir:

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

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

---

[2] Los métodos **static**, que se verán más adelante, pueden ser invocados *por la clase*, sin necesidad de un objeto.

Este acto de invocar a un método suele denominarse *envío de un mensaje a un objeto*. En el ejemplo de arriba, el mensaje es **f( )** y el objeto es **a**. La programación orientada a objetos suele resumirse como un simple "envío de mensajes a objetos".

La lista de parámetros

La lista de parámetros de un método especifica la información que se le pasa. Como puede adivinarse, esta información —como todo lo demás en Java— tiene forma de objetos. Por tanto, lo que hay que especificar en la lista de parámetros son los tipos de objetos a pasar y el nombre a utilizar en cada uno. Como en cualquier situación en Java en la que parece que se estén manipulando directamente objetos, se están pasando referencias[3]. El tipo de referencia, sin embargo, tiene que ser correcto. Si se supone, por ejemplo, que un parámetro debe ser un **String**, lo que se le pase debe ser una cadena de caracteres.

Consideremos un método que reciba como parámetro un **String**, cuya definición, que debe ser ubicada dentro de la definición de la clase para que sea compilada, puede ser la siguiente:

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

Este método dice cuántos bytes son necesarios para almacenar la información de un **String** en particular (cada **carácter** de una **cadena** tiene 16 bits, o 2 bytes para soportar caracteres Unicode). El parámetro **s** es de tipo **String**. Una vez que se pasa **s** al método, es posible tratarlo como a cualquier otro objeto (se le pueden enviar mensajes). Aquí se invoca al método **length( )**, que es uno de los métodos para **String**; devuelve el número de caracteres que tiene la cadena.

También es posible ver el uso de la palabra clave **return**, que hace dos cosas. Primero, quiere decir, "abandona el método, que ya hemos acabado". En segundo lugar, si el método produce un valor, ese valor se ubica justo después de la sentencia **return**. En este caso, el valor de retorno se produce al evaluar la expresión **s.length( )*2**.

Se puede devolver el tipo que se desee, pero si no se desea devolver nada, hay que indicar que el método devuelve **void**. He aquí algunos ejemplos:

```
boolean indicador() { return true; }
float naturalLogBase() { return 2.718f; }
void nada() { return; }
void nada2() {}
```

Cuando el tipo de retorno es **void**, se utiliza la palabra clave **return** sólo para salir del método, y es, por consiguiente, innecesaria cuando se llega al final del mismo. Es posible salir de un método en cualquier punto, pero si se te da un valor de retorno distinto de **void**, el compilador te obligará (me-

[3] Con la excepción habitual de los ya mencionados tipos de datos "especiales" **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double**. Normalmente se pasan objetos, lo cual verdaderamente quiere decir que se pasan referencias a objetos.

diante mensajes de error) a devolver el tipo apropiado de datos independientemente de lo que devuelvas.

En este punto, puede parecer que un programa no es más que un montón de objetos con métodos que toman otros objetos como parámetros y envían mensajes a esos otros objetos. Esto es, sin duda, mucho de lo que está ocurriendo, pero en el capítulo siguiente se verá cómo hacer el trabajo de bajo nivel detallado, tomando decisiones dentro de un método. Para este capítulo, será suficiente con el envío de mensajes.

Construcción de un programa Java

Hay bastantes aspectos que se deben comprender antes de ver el primer programa Java.

Visibilidad de los nombres

Un problema de los lenguajes de programación es el control de nombres. Si se utiliza un nombre en un módulo del programa, y otro programador utiliza el mismo nombre en otro módulo ¿cómo se distingue un nombre del otro para evitar que ambos nombres "colisionen"? En C éste es un problema particular puesto que un programa es un mar de nombres inmanejable. En las clases de C++ (en las que se basan las clases de Java) anidan funciones dentro de las clases, de manera que no pueden colisionar con nombres de funciones anidadas dentro de otras clases. Sin embargo, C++ sigue permitiendo los datos y funciones globales, por lo que las colisiones siguen siendo posibles. Para solucionar este problema, C++ introdujo los *espacios de nombres* utilizando palabras clave adicionales.

Java pudo evitar todo esto siguiendo un nuevo enfoque. Para producir un nombre no ambiguo para una biblioteca, el identificador utilizado no difiere mucho de un nombre de dominio en Internet. De hecho, los creadores de Java utilizaron los nombres de dominio de Internet a la inversa, dado que es posible garantizar que éstos sean únicos. Dado que mi nombre de dominio es **BruceEckel.com**, mi biblioteca de utilidad de **manías** debería llamarse **com.bruceEckel.utilidad.manias**. Una vez que se da la vuelta al nombre de dominio, los nombres supuestamente representan subdirectorios.

En Java 1.0 y 1.1, las extensiones de dominio **com**, **edu**, **org**, **net**, etc. se ponían en mayúsculas por convención, de forma que la biblioteca aparecería como **COM.bruceEckel.utilidad.manias.** Sin embargo, a mitad de camino del desarrollo de Java 2, se descubrió que esto causaba problemas, por lo que de ahora en adelante se utilizarán minúsculas para todas las letras de los nombres de paquetes.

Este mecanismo hace posible que todos sus ficheros residan automáticamente en sus propios espacios de nombres, y cada clase de un fichero debe tener un identificador único. Por tanto, uno no necesita aprender ninguna característica especial del lenguaje para resolver el problema —el lenguaje lo hace por nosotros.

Utilización de otros componentes

Cada vez que se desee usar una clase predefinida en un programa, el compilador debe saber dónde localizarla. Por supuesto, la clase podría existir ya en el mismo fichero de código fuente que la está invocando. En ese caso, se puede usar simplemente la clase —incluso si la clase no se define hasta más adelante dentro del archivo. Java elimina el problema de las "referencias hacia delante" de forma que no hay que pensar en ellos.

¿Qué hay de las clases que ya existen en cualquier otro archivo? Uno podría pensar que el compilador debería ser lo suficientemente inteligente como para localizarlo por sí mismo, pero hay un problema. Imagínese que se quiere usar una clase de un nombre determinado, pero existe más de una definición de esa clase (y presumiblemente se trata de definiciones distintas). O peor, imagine que se está escribiendo un programa, y a medida que se está construyendo se añade una nueva clase a la biblioteca cuyo nombre choca con el de alguna clase ya existente.

Para resolver este problema, debe eliminarse cualquier ambigüedad potencial. Esto se logra diciéndole al compilador de Java exactamente qué clases se quieren utilizar mediante la palabra clave **import**. Esta palabra clave dice al compilador que traiga un *paquete*, que es una biblioteca de clases (en otros lenguajes, una biblioteca podría consistir en funciones y datos además de clases, pero debe recordarse que en Java todo código debe escribirse dentro de una clase).

La mayoría de las veces se utilizarán componentes de las bibliotecas de Java estándar que vienen con el propio compilador. Con ellas, no hay que preocuparse de los nombres de dominio largos y dados la vuelta; uno simplemente dice, por ejemplo:

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

para indicar al compilador que se desea utilizar la clase **ArrayList** de Java. Sin embargo, **util** contiene bastantes clases y uno podría querer utilizar varias de ellas sin tener que declararlas todas explícitamente. Esto se logra sencillamente utilizando el '*' que hace las veces de comodín:

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

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

La palabra clave **static**

Generalmente, al crear una clase se está describiendo qué apariencia tienen sus objetos y cómo se comportan. No se tiene nada hasta crear un objeto de esa clase con **new**, momento en el que se crea el espacio de almacenamiento y los métodos pasan a estar disponibles.

Pero hay dos situaciones en las que este enfoque no es suficiente. Una es cuando se desea tener solamente un fragmento de espacio de almacenamiento para una parte concreta de datos, independientemente de cuántos objetos se creen, o incluso aunque no se cree ninguno. La otra es si se necesita un método que no esté asociado con ningún objeto particular de esa clase. Es decir, se necesita un método al que se pueda invocar incluso si no se ha creado ningún objeto. Ambos efec-

tos se pueden lograr con la palabra clave **estático**. Al decir que algo es **estático** se está indicando que el dato o método no está atado a ninguna instancia de objeto de esa clase en particular. Por ello, incluso si nunca se creó un objeto de esa clase se puede invocar a un método **estático** o acceder a un fragmento de datos **estático**. Con los métodos y datos ordinarios no **estático**, es necesario crear un objeto y utilizarlo para acceder al dato o método, dado que los datos y métodos no **estático** deben conocer el objeto particular con el que está trabajando. Por supuesto, dado que los métodos **estático** no precisan de la creación de ningún objeto, no pueden acceder *directamente* a miembros o métodos no **estático** simplemente invocando a esos otros miembros sin referirse a un objeto con nombre (dado que los miembros y objetos no **estático** deben estar unidos a un objeto en particular).

Algunos lenguajes orientados a objetos utilizan los términos *datos a nivel de clase* y *métodos a nivel de clase*, para indicar que los datos y métodos solamente existen para la clase, y no para un objeto particular de la clase. En ocasiones, estos términos también se usan en los textos.

Para declarar un dato o un miembro a nivel de clase **estático**, basta con colocar la palabra clave **estático** antes de la definición. Por ejemplo, el siguiente fragmento produce un miembro de datos **estáticos** y lo inicializa:

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

Ahora, incluso si se construyen dos objetos de Tipo **PruebaEstatica**, sólo habrá un espacio de almacenamiento para **PruebaEstatica.i**. Ambos objetos compartirán la misma **i**. Considérese:

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

En este momento, tanto **st1.i** como **st2.i** tienen el valor 47, puesto que se refieren al mismo espacio de memoria.

Hay dos maneras de referirse a una variable **estática**. Como se indicó más arriba, es posible nombrarlas a través de un objeto, diciendo, por ejemplo, **st2.i**. También es posible referirse a ella directamente a través de su nombre de clase, algo que no se puede hacer con miembros no estáticos (ésta es la manera preferida de referirse a una variable **estática** puesto que pone especial énfasis en la naturaleza **estática** de esa variable).

```
PruebaEstatica.i++:
```

El operador ++ incrementa la variable. En este momento, tanto **st1.i** como **st2.i** valdrán 48.

Algo similar se aplica a los métodos estáticos. Es posible hacer referencia a ellos, bien a través de un objeto especificado al igual, que ocurre con cualquier método, o bien con la sintaxis adicional **NombreClase.método( )**. Un método estático se define de manera semejante:

```
class FunEstatico {
      static void incr() {PruebaEstatica.i++; }
}
```

Puede observarse que el método **incr( )** de **FunEstatico** incrementa la variable **estática i**. Se puede invocar a **incr( )** de la manea típica, a través de un objeto:

```
FunEstatico sf = new FunEstatico();
sf.incr();
```

O, dado que **incr( )** es un método estático, es posible invocarlo directamente a través de la clase:

```
FunEstatico.incr();
```

Mientras que **static** al ser aplicado a un miembro de datos, cambia definitivamente la manera de crear los datos (uno por cada clase en vez de uno por cada objeto no **estático**), al aplicarse a un método, su efecto no es tan drástico. Un uso importante de **estático** para los métodos es permitir invocar a un método sin tener que crear un objeto. Esto, como se verá, es esencial en la definición del método **main( )**, que es el punto de entrada para la ejecución de la aplicación.

Como cualquier método, un método **estático** puede crear o utilizar objetos con nombre de su propio tipo, de forma que un método **estático** se usa a menudo como un "pastor de ovejas" para un conjunto de instancias de su mismo tipo.

Tu primer programa Java

Finalmente, he aquí el programa[4]. Empieza imprimiendo una cadena de caracteres y posteriormente la fecha, haciendo uso de la clase **Date**, contenida en la biblioteca estándar de Java. Hay que tener en cuenta que se introduce un estilo de comentarios adicional: el '//', que permite insertar un comentario hasta el final de la línea:

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

public class HolaFecha {
      public static void main(String[] args) {
            System.out.println ("Hola, hoy es: ");
            System.out.println (new Date());
      }
}
```

Al principio de cada fichero de programa es necesario poner la sentencia **import** para incluir cualquier clase adicional que se necesite para el código contenido en ese fichero. Nótese que digo "adi-

[4] Algunos entornos de programación irán sacando programas en la pantalla, y luego los cerrarán antes de que uno tenga siquiera opción a ver los resultados. Para detener la salida, se puede escribir el siguiente fragmento de código al final de la función **main ( )**;

```
try {
  System.in.read();
  } catch(Exception e) {}
```

Esto hará que la salida se detenga hasta presionar "Intro" (o cualquier otra tecla). Este código implica algunos conceptos que no se verán hasta mucho más adelante, por lo que todavía no lo podemos entender, aunque el truco es válido igualmente.

cional"; se debe a que hay una cierta biblioteca de clases que se carga automáticamente en todos los ficheros Java: la **java.lang**. Arranque su navegador web y eche un vistazo a la documentación de Sun (si no la ha bajado de *java.sun.com* o no ha instalado la documentación de algún otro modo, será mejor hacerlo ahora). Si se echa un vistazo a la lista de paquetes, se verán todas las bibliotecas de clases que incluye Java. Si se selecciona **java.lang** aparecerá una lista de todas las clases que forman parte de esa biblioteca. Dado que **java.lang** está incluida implícitamente en todos los archivos de código Java, todas estas clases ya estarán disponibles. En **java.lang** no hay ninguna clase **Date**, lo que significa que será necesario importarla de alguna otra biblioteca. Si se desconoce en qué biblioteca en particular está una clase, o si se quieren ver todas las clases, es posible seleccionar "Tree" en la documentación de Java. En ese momento es posible encontrar todas y cada una de las clases que vienen con Java. Después, es posible hacer uso de la función "buscar" del navegador para encontrar **Date**. Al hacerlo, se verá que está listada como **java.util.Date**, lo que quiere decir que se encuentra en la biblioteca **util**, y que es necesario importar **java.util.*** para poder usar **Date**.

Si se vuelve al principio, se selecciona **java.lang** y después **System**, se verá que la clase **System** tiene varios campos, y si se selecciona **out**, se descubrirá que es un objeto **estático PrintStream**. Dado que es **estático**, no es necesario crear ningún objeto. El objeto **out** siempre está ahí y se puede usar directamente. Lo que se hace con el objeto **out** está determinado por su tipo: **PrintStream**. La descripción de este objeto se muestra, convenientemente, a través de un hipervínculo, por lo que si se hace clic en él se verá una lista de todos los métodos de **PrintStream** a los que se puede invocar. Hay unos cuantos, y se irán viendo según avancemos en la lectura del libro. Por ahora, todo lo que nos interesa es **println( )**, que significa "escribe lo que te estoy dando y finaliza con un retorno de carro". Por consiguiente, en cualquier programa Java que uno escriba se puede decir **System.out.println("cosas")** cuando se desee para escribir algo en la consola.

El nombre de la clase es el mismo que el nombre del archivo. Cuando se está creando un programa independiente como éste, una de las clases del archivo tiene que tener el mismo nombre que el archivo. (El compilador se queja si no se hace así.) Esa clase debe contener un método llamado **main( )**, de la forma:

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

La palabra clave **public** quiere decir que el método estará disponible para todo el mundo (como se describe en el Capítulo 5). El parámetro del método **main( )** es un array de objetos **String**. Este programa no usará **args**, pero el compilador Java obliga a que esté presente, pues son los que mantienen los parámetros que se invoquen en la línea de comandos.

La línea que muestra la fecha es bastante interesante:

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

Considérese su argumento: se está creando un objeto **Date** simplemente para enviar su valor a **println**. Tan pronto como haya acabado esta sentencia, ese **Date** deja de ser necesario, y en cualquier momento aparecerá el recolector de basura y se lo llevará. Uno no tiene por qué preocuparse de limpiarlo.

Compilación y ejecución

Para compilar y ejecutar este programa, y todos los demás programas de este libro, es necesario disponer, en primer lugar, de un entorno de programación Java. Hay bastantes entornos de desarrollo de terceros, pero en este libro asumiremos que se está usando el JDK de Sun, que es gratuito. Si se está utilizando otro sistema de desarrollo, será necesario echar un vistazo a la documentación de ese sistema para determinar cómo se compilan y ejecutan los programas.

Conéctese a Internet y acceda a *java.sun.com*. Ahí encontrará información y enlaces que muestran cómo descargar e instalar el JDK para cada plataforma en particular.

Una vez que se ha instalado el JDK, y una vez que se ha establecido la información de *path* en el computador, para que pueda encontrar **javac** y **java**, se puede descargar e instalar el código fuente de este libro (que se encuentra en el CD ROM que viene con el libro, o en *http://www.BruceEckel.com*). Al hacerlo, se creará un subdirectorio para cada capítulo del libro. Al ir al subdirectorio **c02** y escribir:

```
javac HolaFecha.java
```

no se obtendrá ninguna respuesta. Si se obtiene algún mensaje de error, se debe a que no se ha instalado el JDK correctamente, por lo que será necesario ir investigando los problemas que se muestren.

Por otro lado, si simplemente ha vuelto a aparecer el *prompt* del intérprete de comandos, basta con teclear:

```
java HolaFecha
```

y se obtendrá como salida el mensaje y la fecha.

Éste es el proceso a seguir para compilar y ejecutar cada uno de los programas de este libro. Sin embargo, se verá que el código fuente de este libro también tiene un archivo denominado **makefile** en cada capítulo, que contiene comandos "make" para construir automáticamente los archivos de ese capítulo. Puede verse la página web del libro en *http://www.BruceEckel.com* para ver los detalles de uso de los *makefiles*.

Comentarios y documentación empotrada

Hay dos tipos de comentarios en Java. El primero es el estilo de comentarios tradicional de C, que fue heredado por C++. Estos comentarios comienzan por /* y pueden extenderse incluso a lo largo de varias líneas hasta encontrar */. Téngase en cuenta que muchos programadores comienzan cada línea de un comentario continuo por el signo *, por lo que a menudo se verá:

```
/*    Esto es un comentario
*     que se extiende
*     a lo largo de varias líneas
*/
```

Hay que recordar, sin embargo, que todo lo que esté entre /* y */ se ignora, por lo que no hay ninguna diferencia con decir:

```
/* Éste es un comentario que
se extiende a lo largo de varias líneas */
```

La segunda forma de hacer comentarios viene de C++. Se trata del comentario en una sola línea que comienza por // y continúa hasta el final de la línea. Este tipo de comentario es muy conveniente y se utiliza muy frecuentemente debido a su facilidad de uso. Uno no tiene que buscar por el teclado donde está el / y el * (basta con pulsar dos veces la misma tecla), y no es necesario cerrar el comentario, por lo que a menudo se verá:

```
// esto es un comentario en una sola línea
```

Documentación en forma de comentarios

Una de las partes más interesantes del lenguaje Java es que los diseñadores no sólo tuvieron en cuenta que la escritura de código era la única actividad importante —sino que también pensaron en la documentación del código. Probablemente el mayor problema a la hora de documentar el código es el mantenimiento de esa documentación. Si la documentación y el código están separados, cambiar la documentación cada vez que se cambia el código se convierte en un problema. La solución parece bastante simple: unir el código a la documentación. La forma más fácil de hacer esto es poner todo en el mismo archivo. Para completar la estampa, sin embargo, es necesaria alguna sintaxis especial de comentarios para marcarlos como documentación especial, y una herramienta para extraer esos comentarios y ponerlos en la forma adecuada.

La herramienta para extraer los comentarios se denomina *javadoc*. Utiliza parte de la tecnología del compilador de Java para buscar etiquetas de comentario especiales que uno incluye en sus programas. No sólo extrae la información marcada por esas etiquetas, sino que también extrae el nombre de la clase o del método al que se adjunta el comentario. De esta manera es posible invertir la mínima cantidad de trabajo para generar una decente documentación para los programas.

La salida de javadoc es un archivo HTML que puede visualizarse a través del navegador Web. Esta herramienta permite la creación y mantenimiento de un único archivo fuente y genera automáticamente documentación útil. Gracias a javadoc se tiene incluso un estándar para la creación de documentación, tan sencillo que se puede incluso esperar o solicitar documentación con todas las bibliotecas Java.

Sintaxis

Todos los comandos de javadoc se dan únicamente en comentarios /**. Estos comentarios acaban, como siempre, con */. Hay dos formas principales de usar javadoc: empotrar HTML, o utilizar "etiquetas doc". Las etiquetas doc son comandos que comienzan por '@' y se sitúan al principio de una línea de comentarios (en la que se ignora un posible primer '*').

Hay tres "tipos" de documentación en forma de comentarios, que se corresponden con el elemento al que precede el comentario: una clase, una variable o un método. Es decir, el comentario relativo

a una clase aparece justo antes de la definición de la misma; el comentario relativo a una variable precede siempre a la definición de la variable, y un comentario de un método aparece inmediatamente antes de la definición de un método. Un simple ejemplo:

```
/** Un comentario de clase */
public class PruebaDoc {
     /** Un comentario de una variable */
     public int i;
     /** Un comentario de un método */
     public void f() {}
}
```

Nótese que javadoc procesará la documentación en forma de comentarios sólo de miembros **public** y **protected**. Los comentarios para miembros **private** y "friendly" (véase Capítulo 5) se ignoran, no mostrándose ninguna salida (sin embargo es posible usar el modificador **—private** para incluir los miembros **privados**). Esto tiene sentido, dado que sólo los miembros **públicos** y **protegidos** son visibles fuera del objeto, que será lo que constituya la perspectiva del programador cliente. Sin embargo, la salida incluirá todos los comentarios de la **clase**.

La salida del código anterior es un archivo HTML que tiene el mismo formato estándar que toda la documentación Java, de forma que los usuarios se sientan cómodos con el formato y puedan navegar de manera sencilla a través de sus clases. Merece la pena introducir estos códigos, pasarlos a través de javadoc y observar el fichero HTML resultante para ver los resultados.

HTML empotrado

Javadoc pasa comandos HTML al documento HTML generado. Esto permite un uso total de HTML; sin embargo, el motivo principal es permitir dar formato al código, como:

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

También puede usarse HTML como se haría en cualquier otro documento web para dar formato al propio texto de las descripciones:

```
/**
* Uno puede <em>incluso</em> insertar una lista:
* <ol>
* <li> Elemento uno
* <li> Elemento dos
* <li> Elemento tres
* </ol>
*/
```

Nótese que dentro de los comentarios de documentación, los asteriscos que aparezcan al principio de las líneas serán desechados por javadoc, junto con los espacios adicionales a éstos. Javadoc vuelve a dar formato a todo adaptándolo a la apariencia estándar de la documentación. No deben utilizarse encabezados como <**hl**> o <**hr**> como HTML empotrado porque javadoc inserta sus propios encabezados y éstos interferirían con ellos.

Todos los tipos de documentación en comentarios —de clases, variables y métodos— soportan HTML empotrado.

@see: referencias a otras clases

Los tres tipos de comentarios de documentación (de clase, variable y métodos) pueden contener etiquetas **@see**, que permiten hacer referencia a la documentación de otras clases. Javadoc generará HTML con las etiquetas **@see** en forma de vínculos a la otra documentación. Las formas son:

```
@see nombredeclase
@see nombredeclase-totalmente-cualificada
@see nombredeclase-totalmente-cualificada#nombre-metodo
```

Cada una añade un hipervínculo "Ver también" a la documentación generada. Javadoc no comprobará los hipervínculos que se le proporcionen para asegurarse de que sean válidos.

Etiquetas de documentación de clases

Junto con el HTML empotrado y las referencias **@see**, la documentación de clases puede incluir etiquetas de información de la versión y del nombre del autor. La documentación de clases también puede usarse para las *interfaces* (véase Capítulo 8).

@versión

Es de la forma:

```
@versión información-de-versión
```

en el que **información-de-versión** es cualquier información significativa que se desee incluir. Cuando se especifica el indicador **-versión** en la línea de comandos javadoc, se invocará especialmente a la información de versión en la documentación HTML generada.

@author

Es de la forma:

```
@autor información-del-autor
```

donde la **información-del-autor** suele ser el nombre, pero podría incluir también la dirección de correo electrónico u otra información apropiada. Al activar el parámetro -author en la línea de comandos javadoc, se invocará a la información relativa al autor en la documentación HTML generada.

Se pueden tener varias etiquetas de autor, en el caso de tratarse de una lista de autores, pero éstas deben ponerse consecutivamente. Toda la información del autor se agrupará en un único párrafo en el HTML generado.

@since

Esta etiqueta permite indicar la versión del código que comenzó a utilizar una característica concreta. Se verá que aparece en la documentación para ver la versión de JDK que se está utilizando.

Etiquetas de documentación de variables

La documentación de variables solamente puede incluir HTML empotrado y referencias **@see**.

Etiquetas de documentación de métodos

Además de documentación empotrada y referencias **@see**, los métodos permiten etiquetas de documentación para los parámetros, los valores de retorno y las excepciones.

@param

Es de la forma:

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

donde **nombre-parámetro** es el identificador de la lista de parámetros, y **descripción** es el texto que vendrá en las siguientes líneas. Se considera que la descripción ha acabado cuando se encuentra una nueva etiqueta de documentación. Se puede tener cualquier número de estas etiquetas, generalmente una por cada parámetro.

@return

Es de la forma:

```
@return descripción
```

donde **descripción** da el significado del valor de retorno. Puede ocupar varias líneas.

@throws

Las excepciones se verán en el Capítulo 10, pero sirva como adelanto que son objetos que pueden "lanzarse" fuera del método si éste falla. Aunque al invocar a un método sólo puede lanzarse una excepción, podría ocurrir que un método particular fuera capaz de producir distintos tipos de excepciones, necesitando cada una de ellas su propia descripción. Por ello, la etiqueta de excepciones es de la forma:

```
@throws nombre-de-clase-totalmente-cualificada descripción
```

donde **nombre-de-clase-totalmente-cualificada** proporciona un nombre sin ambigüedades de una clase de excepción definida en algún lugar, y **descripción** (que puede extenderse a lo largo de varias líneas) indica por qué podría levantarse este tipo particular de excepción al invocar al método.

@deprecated

Se utiliza para etiquetar aspectos que fueron mejorados. Esta etiqueta es una sugerencia para que no se utilice esa característica en particular nunca más, puesto que en algún momento del futuro puede que se elimine. Un método marcado como **@deprecated** hace que el compilador presente una advertencia cuando se use.

Ejemplo de documentación

He aquí el primer programa Java de nuevo, al que en esta ocasión se ha añadido documentación en forma de comentarios:

```
//: c02:HolaFecha.java
import java.util.*;

/** El primer ejemplo de Piensa en Java.
 *  Muestra una cadena de caracteres y la fecha de hoy.
 *  @author Bruce Eckel
 *  @author www.BruceEckel.com
 *  @version 2.0
*/
public class HolaFecha {
     /** Único punto de entrada para la clase y la aplicación
      * @param args array de cadenas de texto pasadas como
parámetros
      * @return No hay valor de retorno
      * @exception exceptions No se generarán excepciones
     */
public static void main (String[] args){
     System.out.printl("Hola, hoy es: ");
     System.out.println(new Date());
}
} ///:~
```

La primera línea del archivo utiliza mi propia técnica de poner "**:**" como marcador especial de la línea de comentarios que contiene el nombre del archivo fuente. Esa línea contiene la información de la trayectoria al fichero (en este caso, **c02** indica el Capítulo 2) seguido del nombre del archivo[5]. La última línea también acaba con un comentario, esta vez indicando la finalización del listado de código fuente, que permite que sea extraído automáticamente del texto de este libro y comprobado por un compilador.

[5] Una herramienta que he creado usando Python (ver http://www.Python.org) utiliza esta información para extraer esos ficheros de código, ponerlos en los subdirectorios apropiados y crear los "makefiles".

Estilo de codificación

El estándar no oficial de java dice que se ponga en mayúsculas la primera letra del nombre de una clase. Si el nombre de la clase consta de varias palabras, se ponen todas juntas (es decir, no se usan guiones bajos para separar los nombres) y se pone en mayúscula la primera letra de cada palabra, como por ejemplo:

```
class TodosLosColoresDelArcoiris { // ...
```

En casi todo lo demás: métodos, campos (variables miembro), y nombres de referencias a objeto, el estilo aceptado es el mismo que para las clases, con la *excepción* de que la primera letra del identificador debe ser minúscula. Por ejemplo:

```
class TodosLosColoresDelArcoiris {
      int unEnteroQueRepresentaUnColor;
      void cambiarElTonoDelColor (int nuevoTono) {
      // ...
      }
      // ...
}
```

Por supuesto, hay que recordar que un usuario tendría que teclear después todos estos nombres largos, por lo que se ruega a los programadores que lo tengan en cuenta.

El código Java de las bibliotecas de Sun también sigue la forma de apertura y cierre de las llaves que se utilizan en este libro.

Resumen

En este capítulo se ha visto lo suficiente de programación en Java como para entender cómo escribir un programa sencillo, y se ha realizado un repaso del lenguaje y algunas de sus ideas básicas. Sin embargo, los ejemplos hasta la fecha han sido de la forma "haz esto, después haz esto otro, y finalmente haz algo más". ¿Y qué ocurre si quieres que el programa presente alternativas, como "si el resultado de hacer esto es rojo, haz esto; sino, haz no se qué más?" El soporte que Java proporciona a esta actividad fundamental de programación se verá en el capítulo siguiente.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Siguiendo el ejemplo **HolaFecha.java** de este capítulo, crear un programa "Hola, mundo" que simplemente escriba esa frase. Sólo se necesita un método en la clase (la clase "main" que es la que se ejecuta al arrancar el programa). Recordar hacerla **static** e incluir la lista de parámetros, incluso aunque no se vaya a usar. Compilar el programa con **javac** y ejecutarlo

utilizando **java**. Si se utiliza un entorno de desarrollo distinto a JDK, aprender a compilar y ejecutar programas en ese entorno.

2. Encontrar los fragmentos de código involucrados en **UnNombreDeTipo** y convertirlos en un programa que se compile y ejecute.
3. Convertir los fragmentos de código de **SoloDatos** en un programa que se compile y ejecute.
4. Modificar el Ejercicio 3, de forma que los valores de los datos de **SoloDatos** se asignen e impriman en **main( )**.
5. Escribir un programa que incluya y llame al método **almacenamiento( )**, definido como fragmento de código en este capítulo.
6. Convertir los fragmentos de código de **FunEstatico** en un programa ejecutable.
7. Escribir un programa que imprima tres parámetros tomados de la línea de comandos. Para lograrlo, será necesario indexarlos en el array de **Strings** de línea de comandos.
8. Convertir el ejemplo **TodosLosColoresDelArcoiris** en un programa que se compile y ejecute.
9. Encontrar el código de la segunda versión de **HolaFecha.java**, que es el ejemplo de documentación en forma de comentarios. Ejecutar **javadoc** del fichero y observar los resultados con el navegador web.
10. Convertir **PruebaDoc** en un fichero que se compile y pasarlo por **javadoc**. Verificar la documentación resultante con el navegador web.
11. Añadir una lista de elementos HTML a la documentación del Ejercicio 10.
12. Tomar el programa del Ejercicio 10 y añadirle documentación en forma de comentarios. Extraer esta documentación en forma de comentarios a un fichero HTML utilizando **javadoc** y visualizarla con un navegador web.