CAPÍTULO

2 Referencias

En el Capítulo 1 hemos estudiado los tipos primitivos de Java. Todos los tipos que no sean ninguno de los ocho tipos primitivos de Java son tipos referencia, incluyéndose en estos últimos entidades tan importantes como las cadenas de caracteres, los vectores y los canales de E/S.

En este capítulo veremos:

- Qué es un tipo referencia y qué es un valor.
- En qué se distinguen los tipos referencia de los tipos primitivos.
- Ejemplos de tipos referencia, incluyendo las cadenas de caracteres, los vectores y los canales de E/S.
- Cómo se emplean las excepciones para detectar comportamientos erróneos.

2.1 ¿Qué es una referencia?

El Capítulo 1 describe los ocho tipos primitivos, junto con algunas de las operaciones que se pueden realizar con sus elementos. Todos los demás tipos de Java son tipos referencia, incluyendo las cadenas de caracteres, los vectores y los ficheros. Pero, ¿qué es una referencia? Una *variable referencia* (o simplemente *referencia*) en Java es una variable que guarda la dirección de memoria en la que se almacena un objeto.

Por ejemplo, en la Figura 2.1 hay dos objetos de tipo `Punto`. Supongamos que estos objetos están almacenados en las posiciones de memoria 1000 y 1024, respectivamente. Para ambos existen tres referencias: `punto1`, `punto2` y `punto3`. `punto1` y `punto3` referencian al objeto situado en la posición 1000 y `punto2` referencia al objeto situado en la posición 1024. Observe que las direcciones de los objetos, como 1000 y 1024, son asignadas por el compilador de forma arbitraria (siempre que encuentre memoria disponible), por lo que estos valores no son útiles externamente como números. Sin embargo, el hecho de que `punto1` y `punto3` almacenen valores idénticos es útil: significa que están referenciando al mismo objeto.

Una referencia siempre contiene la dirección de memoria en la que un objeto está guardado, a menos que no referencie a ningún objeto. En este caso, almacena la *referencia nula*, `null`. Java no permite referencias a variables primitivas.

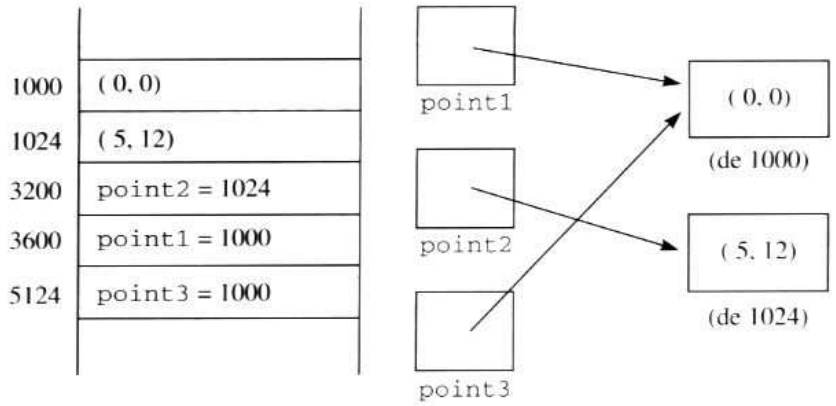

Figura 2.1 Representación gráfica de una referencia: el objeto de tipo `Punto` almacenado en la posición de memoria 1000 es referenciado por `punto1` y `punto3`. El objeto de tipo `Punto` almacenado en la posición de memoria 1024 es referenciado por `punto2`. Las posiciones de memoria donde se guardan las variables son arbitrarias.

Existen dos amplias categorías de operaciones que pueden aplicarse a variables referencia. Una de ellas nos permite examinar o manipular el valor referencia. Por ejemplo, si modificamos el valor almacenado en `punto1` (1000) pasaríamos a estar referenciando a otro objeto. También podemos comparar `punto1` y `punto3`, y determinar si están referenciando al mismo objeto. La otra categoría de operaciones se aplica al objeto referenciado; es posible examinar o cambiar el estado interno de uno de los objetos de tipo `Punto`. Por ejemplo, podemos averiguar las coordenadas *x* e *y* de algunos `Puntos`.

Antes de describir qué puede hacerse con referencias, vamos a ver lo que no está permitido. Considérese la expresión `punto1*punto2`. Como los valores almacenados por `punto1` y `punto2` son 1000 y 1024, respectivamente, su producto debería ser 1024000. Sin embargo, éste es un cálculo sin sentido que no tendría ninguna utilidad. Las variables referencia almacenan direcciones de memoria, y no puede asociarse ningún significado lógico al producto de dos direcciones de memoria.

Análogamente, `punto1++` no tiene sentido en Java; esta expresión sugiere que `punto1` —1000— debería incrementarse a 1001, pero en ese caso podría no referenciar a un objeto válido de tipo `Punto`. Muchos lenguajes de programación definen el *puntero*, que se comporta como una variable referencia. Por el contrario, los punteros de C++ son mucho más peligrosos, pues se permiten operaciones aritméticas sobre las direcciones de memoria almacenadas. Así, en C++, `punto1++` tiene sentido. Además, C++ permite punteros a tipos primitivos, por lo que se debe ser cuidadoso al distinguir entre cálculos con direcciones de memoria y cálculos con los objetos referenciados. Esto se hace explícitamente *desreferenciando* el puntero. En la práctica, los inseguros punteros de C++ causan numerosos errores de programación.

En Java los únicos operadores permitidos para manipular los tipos referencia (con una única excepción en el caso de los `Strings`) son las asignaciones vía `=`, y las comparaciones de igualdad, a través de `==` y `!=`.

La Figura 2.2 ilustra el operador de asignación para variables referencia. Asignando a `punto3` el valor almacenado en `punto2` se consigue que `punto3` referencie al mismo objeto que `punto2` estaba referenciando. Ahora `punto2==punto3` es `true` ya que `punto2` y `punto3` almacenan el valor 1024, y por lo tanto, referencian al mismo objeto. `punto1!=punto2` es también cierto ya que `punto1` y `punto2` referencian objetos distintos.

La otra categoría de operaciones maneja el objeto que está siendo referenciado. Sobre él sólo pueden realizarse tres acciones básicas:

1. Realizar una conversión de tipos (Sección 1.4.4).
2. Acceder a un campo interno o invocar a un método, empleando el operador punto (.) (Sección 2.2.1).
3. Emplear el operador `instanceof` para comprobar que el objeto almacenado es de un tipo determinado (Sección 3.6.3).

La siguiente sección muestra con más detalle las operaciones con referencias más comunes.

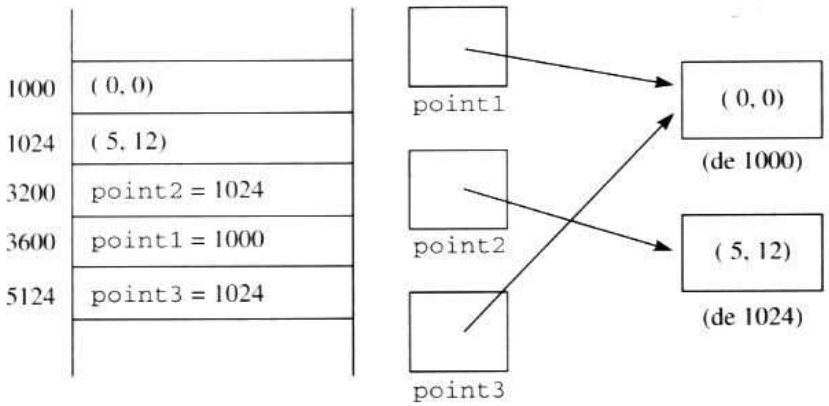

Figura 2.2 El resultado de `punto3=punto2`: ahora `punto3` referencia el mismo objeto que `punto2`.

2.2 Nociones básicas sobre objetos y referencias

En Java, un *objeto* es un elemento de cualquiera de los tipos no primitivos. Los objetos son tratados de forma diferente a los valores de los tipos primitivos. Los tipos primitivos, como hemos visto, se manipulan por *valor*, es decir, los valores supuestos para las variables primitivas se almacenan en dichas variables y son copiados de unas a otras en las asignaciones. Como se ha explicado en la Sección 2.1, las variables referencia guardan referencias a objetos. El objeto actual se almacena en algún lugar de la memoria, y la variable referencia guarda la posición de memoria del objeto. Así, una variable referencia representa simplemente un nombre de una zona de la memoria. Esto significa que las variables primitivas y

En Java, un *objeto* es un elemento de cualquiera de los tipos no primitivos.

las variables referencia se comportan de forma diferente. Esta sección estudia con más detalle dichas diferencias e ilustra las operaciones permitidas con variables referencia.

2.2.1 El operador punto (.)

El operador punto (.) se emplea para seleccionar un método y aplicarlo a un objeto. Por ejemplo, supongamos que se tiene un objeto de tipo `Circulo`, en el que se define el método `area`. Si `elCirculo` referencia a un `Circulo`, entonces podemos calcular el área del `Circulo` referenciado (y guardarla en una variable de tipo `double`) haciendo lo siguiente:

```
double elArea = elCirculo.area( );
```

Es posible que `elCirculo` contenga la referencia `null`. En este caso, el empleo del operador punto generará una `NullPointerException` cuando el programa se ejecute. Generalmente, esto causará la terminación anormal del programa.

El operador punto también puede emplearse para acceder a las componentes individuales de un objeto, suponiendo que se han hecho los preparativos necesarios para hacerlas observables. El Capítulo 3 muestra cómo se realizan dichos preparativos. Se explica además la razón de por qué es generalmente preferible no permitir acceso directo a las componentes individuales.

2.2.2 Declaración de objetos

Ya hemos visto la sintaxis adecuada para declarar variables primitivas. En el caso de los objetos existe una diferencia importante. Cuando se declara una variable referencia, simplemente estamos indicando un nombre que puede emplearse para referenciar un objeto almacenado en memoria. Sin embargo, la declaración, por sí misma, no genera ese objeto. Por ejemplo, supongamos que existe un objeto de tipo `Button` que deseamos añadir en un `Panel p` dado, empleando para ello el método `add` (todo ello definido en las librerías de Java). Para hacerlo utilizaríamos las instrucciones

```
Button b;              // b puede referenciar a un objeto de tipo Button
b.setLabel( "No" ); // La etiqueta del botón b es "No"
p.add( b );            // b se añade al panel p
```

Cuando se declara un tipo referencia no se asigna memoria a ningún objeto. En este momento la referencia es `null`. Para crear un nuevo objeto debe emplearse *new*.

Todo parece estar bien hasta que recordamos que `b` es el nombre de un objeto de tipo `Button`, pero que aún no se ha creado ninguno de ellos. Como resultado, después de la declaración de `b` el valor almacenado por la variable referencia `b` es `null`, indicando que `b` todavía no apunta a ningún objeto válido de tipo `Button`. Como consecuencia, la segunda línea es incorrecta ya que está intentando alterar un objeto inexistente. En este contexto, el compilador detectará el error, indicando que `b` no está inicializada. En otros casos, el compilador no detectará el fallo y se producirá un error durante la ejecución, obteniéndose el críptico mensaje `NullPointerException`.

Una forma de almacenar un objeto (la única común a todos los tipos referencia) es emplear la palabra reservada `new`. `new` se utiliza para construir un nuevo objeto. Un modo de hacerlo es el siguiente:

La palabra clave new se emplea para *crear* un objeto.

```
Button b;              // b puede referenciar a un objeto de tipo Button
b = new Button( );     // Ahora b referencia a un objeto creado
b.setLabel( "No" ); // La etiqueta del botón b es "No"
p.add( b );            // b se añade al panel p
```

Obsérvese que se necesitan los paréntesis tras el tipo del objeto.

Se necesitan los paréntesis cuando se emplea new.

También es posible combinar la declaración y la construcción de un objeto, como en

```
Button b = new Button( );
b.setLabel( "No" );  // La etiqueta de botón b es "No"
p.add( b );           // b se añade al panel p
```

Muchos objetos pueden construirse dándoles un valor inicial. Por ejemplo, el objeto de tipo `Button` puede construirse con un `String` que sea su etiqueta:

La construcción puede especificar el estado inicial del objeto.

```
Button b = new Button( "No" );
p.add( b );            // b se añade al panel p
```

2.2.3 Recogida de basura

Como todos los objetos deben construirse, podríamos suponer que cuando uno ya no se vaya a emplear más, debemos destruirlo explícitamente. En Java, cuando un objeto creado no es referenciado por ninguna variable referencia, la memoria que consume es reclamada automáticamente y pasa a estar disponible. Esta técnica se conoce como *recogida de basura*.

Java emplea la *recogida de basura*. Con la recogida de basura, la memoria no referenciada se reutiliza automáticamente.

2.2.4 El significado de =

Supongamos que se tienen dos variables primitivas `lizq` y `lder`, donde empleamos `lizq` y `lder` para denotar los *lados izquierdo* y *derecho*, respectivamente. Entonces, la instrucción de asignación

Se emplean `lizq` y `lder` para los *lados izquierdo* y *derecho*, respectivamente.

```
lizq = lder;
```

tiene un sencillo significado: el valor almacenado en `lder` es copiado en la variable primitiva `lizq`. Los cambios posteriores en alguna de las variables no afectan a la otra.

En los objetos, el significado de = es el mismo: los valores almacenados se copian. Si `lizq` y `lder` son referencias (de tipos compatibles), después de la instrucción de asignación, `lizq` señalará el mismo objeto al que `lder` apunta. Aquí lo que se copian son las direcciones. El objeto al que `lizq` referenciaba dejará de ser referenciado por esta variable. Si `lizq` era la única referencia a ese objeto,

En los objetos, = es la asignación de referencias en lugar de la copia de objetos.

entonces dicho objeto se encuentra ahora desreferenciado por lo que será incluido en la próxima recogida de basura. Observe que los objetos no se copian.

Veamos algunos ejemplos. En primer lugar, supongamos que se quiere tener dos objetos de tipo `Button`. Intentamos lograrlo comenzando por crear `noBoton`. Después creamos `siBoton` modificando `noBoton` del modo siguiente:

```
Button noBoton = new Button( "No" );
Button siBoton = noBoton;
siBoton.setLabel( "Si" );
p.add( noBoton );
p.add( siBoton );
```

Esto no funciona ya que sólo ha sido creado un objeto de tipo `Button`. De este modo, la segunda instrucción simplemente indica que `siBoton` es otro nombre para el botón creado en la primera línea. Dicho botón tiene ahora dos nombres. En la tercera línea, el objeto de tipo `Button` construido ha cambiado su etiqueta por `Si`, pero esto significa que el único objeto `Button`, identificado por dos nombres, se etiqueta ahora con `Si`. Las dos últimas líneas añaden al `Panel p` el objeto `Button` dos veces.

El hecho de que `siBoton` nunca consulte su propio objeto carece de importancia en este ejemplo. El problema es la asignación. Considere

```
Button noBoton = new Button( "No" );
Button siBoton = new Button( );
siBoton = noBoton;
siBoton.setLabel( "Si" );
p.add( noBoton );
p.add( siBoton );
```

Las consecuencias son exactamente las mismas. Ahora sí han sido generados dos objetos `Button`, pero al final de la secuencia el primer objeto está referenciado por `noBoton` y `siBoton`, mientras que el segundo objeto está sin referenciar.

A primera vista, el hecho de que los objetos no puedan copiarse parece una limitación importante. En realidad no lo es, aunque cuesta un poco acostumbrarse. (Para ser exactos algunos objetos sí necesitan ser copiados. Para ellos, si está disponible un método `clone`, debe usarse. `clone` invoca a `new` para obtener una réplica del objeto. Sin embargo, `clone` no se utiliza en este libro.)

2.2.5 Paso de parámetros

Paso de parámetros por valor significa que, para tipos referencia, el parámetro formal referencia al mismo objeto que el parámetro actual.

A causa del paso de parámetros por valor, los parámetros actuales se pasan en lugar de los parámetros formales empleando la asignación usual. Si el parámetro es un tipo referencia, entonces ya sabemos que la asignación significa que el parámetro formal ahora referencia al mismo objeto que el parámetro actual. Cualquier método aplicado al parámetro formal se aplica también al parámetro actual. En otros lenguajes esto se conoce como *paso de parámetros por referencia*. Emplear esta terminología en Java podría provocar equívocos, pues haría pensar que el

paso de parámetros es diferente. En realidad el paso de parámetros no ha cambiado; en su lugar, son los parámetros los que han variado, pasando de corresponder a tipos no referenciales a corresponder a tipos referencia.

2.2.6 El significado de ==

En los tipos primitivos, == es verdadero cuando los valores almacenados son idénticos. En los tipos referencia su significado es diferente, aunque totalmente consistente con la discusión previa.

En los tipos referencia, == es verdadero sólo si las dos referencias apuntan al *mismo objeto*.

Dos variables referencia son iguales vía == si ambas apuntan al mismo objeto (o bien son las dos `null`). Consideremos, por ejemplo, la siguiente secuencia de asignaciones:

```
Button a = new Button( "Si" );
Button b = new Button( "Si" );
Button c = b;
```

Aquí se han generado dos objetos. El primero es accesible mediante el nombre `a`, mientras que el segundo se accede con cualquiera de los dos nombres, `b` y `c`. En consecuencia `b==c` es `true`. Sin embargo, aunque `a` y `b` estén referenciando objetos que parecen tener el mismo valor, `a==b` es `false`, ya que apuntan a objetos distintos. Para `!=` se aplican reglas similares.

El método `equals` se emplea para saber si dos referencias apuntan a objetos con estados idénticos.

En algunas ocasiones es importante conocer si los objetos referenciados son idénticos. Todos los objetos pueden compararse mediante `equals`, pero para muchos (incluyendo los de tipo `Button`) `equals` devuelve `false` a menos que ambas referencias estén apuntando al mismo objeto (en otras palabras, para algunos objetos `equals` no es otra cosa que el test ==). En la Sección 2.3, donde se estudia el tipo `String`, veremos un ejemplo en el que `equals` resulta muy útil.

2.2.7 Sobrecarga de operadores para objetos

Salvo para la excepción descrita en la sección siguiente, los operadores no pueden definirse para ser aplicados sobre objetos. Así, no existe ningún operador < disponible sobre ningún tipo de objetos. En su lugar, deberíamos definir para realizar esta función un método con identificador, como por ejemplo, `menorQue`.

2.3 Cadenas de caracteres

El tipo `String` se comporta como cualquier otro tipo referencia.

Las cadenas de caracteres en Java se manipulan mediante el tipo `String`. El lenguaje hace aparecer a `String` como un tipo primitivo ya que define la + y el operador += para la concatenación de cadenas. Pero éste es el único tipo referencia para el que la sobrecarga de operadores está permitida. Por lo demás, el tipo `String` se comporta como cualquier otro tipo no primitivo.

2.3.1 Conceptos básicos de la manipulación de `Strings`

Los `Strings` son *inmutables*, es decir, un objeto de tipo `String` no puede modificarse.

Existen dos reglas fundamentales sobre el tipo `String`. En primer lugar, se comporta como cualquier otro tipo no primitivo, a excepción de lo que hace referencia a los operadores de concatenación. En segundo lugar, el tipo `String` es *inmutable*. Esto significa que, una vez que un objeto `String` se ha creado, su contenido no puede modificarse.

Como un `String` es inmutable, siempre es seguro emplear el operador == con él. Así, un `String` puede declararse de la forma siguiente:

```
String vacia      = " ";
String mensaje    = "Hola";
String repeticion = mensaje;
```

Después de estas declaraciones, se ha creado dos objetos `String`. El primero de ellos es la cadena vacía, referenciada por `vacia`. El segundo es el `String "Hola"`, referenciado por `mensaje` y `repeticion`. En otros casos la doble referencia mediante `mensaje` y `repeticion` podría generar algún problema. Sin embargo, como los `Strings` son inmutables, la compartición de objetos de tipo `String` es segura a la vez que eficiente. El único modo de modificar el valor que `repeticion` está referenciando es crear un nuevo `String` y hacer que `repeticion` lo referencie. Esto no tiene efecto alguno sobre el `String` que referencia `mensaje`.

2.3.2 Concatenación de cadenas

Java no permite la sobrecarga de operadores para tipos referencia. Sin embargo, tenemos una excepción en la concatenación de cadenas.

La concatenación de cadenas de caracteres se hace a través de los operadores + y +=.

El operador + realiza la concatenación cuando, al menos, uno de los operandos es un `String`. El resultado es una referencia al objeto de tipo `String` recién construido. Por ejemplo,

```
"este" + "aquel"    // Genera "esteaquel"
"abc" + 5           // Genera "abc5"
5 + "abc"           // Genera "5abc"
"a" + "b" + "c"     // Genera "abc"
```

Las cadenas de un solo carácter no deben reemplazarse por caracteres individuales; el Ejercicio 2.5 pide explicar el porqué. Observe que el operador + es asociativo por la izquierda, de modo que

```
"a" + 1 + 2      // Genera "a12"
1 + 2 + "a"      // Genera "3a"
1 + ( 2 + "a" )  // Genera "12a"
```

También el operador += está definido sobre el tipo `String`. El efecto de `str+=exp` es el mismo que el de `str=str+exp`. Más concretamente, esto significa que `str` referenciará al mismo objeto de tipo `String` generado por `str+exp`.

2.3.3 Comparando cadenas

Para realizar la comparación de cadenas de caracteres es preciso emplear los métodos *equals* y *compareTo*.

Como el operador básico de asignación funciona para los `Strings`, es lógico suponer que los operadores relacionales y de igualdad también funcionarán. Sin embargo, ello no es cierto.

De acuerdo con la prohibición de la sobrecarga de operadores, los operadores relacionales (<, >, <=, >=) no están definidos para el tipo `String`. Mas aún, == y != tienen el significado usual para las variables referencia. Así, para dos objetos de tipo `String`, `lizq` y `lder`, `lizq==lder` es `true` sólo cuando ambos referencian al mismo objeto `String`. De modo que, si referencian dos objetos distintos con el mismo contenido, `lizq==lder` es `false`. Para `!=` valen razonamientos similares.

Para comparar la igualdad de dos objetos `String` empleamos `equals`. `lizq.equals(lder)` es `true` si `lizq` y `lder` referencian `Strings` que almacenan valores idénticos.

Una comprobación más general puede realizarse con el método `compareTo`. `lizq.compareTo(lder)` compara dos objetos `String`, `lizq` y `lder`. Se devuelve un número negativo, cero o un número positivo, según `lizq` sea menor, igual o mayor que `lder`, respectivamente.

2.3.4 Otros métodos del tipo `String`

Para calcular la longitud de una cadena de caracteres, obtener un solo carácter y obtener una subcadena se emplean los métodos *length*, *charAt* y *substring*, respectivamente.

La longitud de un objeto `String` (la cadena vacía tiene longitud cero) puede obtenerse con el método `length`. Como `length` es un método, es necesario emplear paréntesis.

Se tienen definidos dos métodos para acceder a los caracteres individuales de un `String`. El método `charAt` obtiene un solo carácter, especificando su posición (la primera posición es la posición 0). El método `substring` devuelve una referencia a un nuevo `String`. La llamada se realiza especificando la posición de inicio y la primera posición no incluida.

Veamos a continuación un ejemplo de estos tres métodos:

```
String saludo = "hola";
int lon = saludo.length( );              // lon es 4
char ch = saludo.charAt( 1 );            // ch es 'o'
String sub = saludo.substring( 2, 4 );   // sub es "la"
```

2.3.5 Conversión entre cadenas y tipos primitivos

toString convierte valores de tipos primitivos (y objetos) en un objeto de tipo `String`.

El método `toString` puede emplearse para convertir cualquier valor de tipo primitivo en un objeto de tipo `String`. Por ejemplo, `toString(45)` devuelve una referencia al recién creado `String "45"`. Muchos tipos tienen implementado el método `toString`. De hecho, cuando el operador + tiene un solo argumento `String`, el argumento que no lo es se convierte automáticamente en una cadena aplicando `toString`. Para los tipos de enteros, una forma alternativa de `toString` permite la especificación de la base de la representación. Así

```
System.out.println( Integer.toString( 55,2 ) );
```

imprime la representación binaria de 55.

El valor de tipo `int` representado por un `String` puede obtenerse llamando al método `Integer.parseInt`. Este método genera una excepción si el `String` no representa un valor de tipo `int`. Las excepciones se estudian en la Sección 2.5. Para obtener un valor `double` a partir de un `String` es necesario más trabajo. Aquí se muestran algunos ejemplos:

```
int    x = Integer.parseInt( "75" );
double y = Double.valueOf( "3.14" ).doubleValue( );
```

```
int    x = Integer.valueOf( "75" ).intValue( );
double y = Double.parseDouble( "3.14" );
```

2.4 Vectores

Un vector almacena un conjunto de elementos del mismo tipo.

Un *agregado* es una colección de entidades almacenadas en una unidad. El *vector* es el mecanismo básico para almacenar una colección de entidades del mismo tipo. En Java el vector no es un tipo primitivo, pero se comporta de forma muy similar al resto de los objetos. Así, muchas de las reglas de los objetos pueden aplicarse también a los vectores.

El *operador de indexación de vectores* [ ] permite el acceso a cada elemento del vector.

Cada entidad en el vector puede ser accedida mediante el *operador de indexación de vectores* [ ]. Decimos que el operador [ ] *indexa* el vector en el sentido de que especifica qué objeto debe ser accedido. A diferencia de C y C++, el chequeo de los límites se realiza automáticamente.

Los vectores se indexan comenzando en cero. El número de elementos almacenados se obtiene mediante el atributo `length`. Los paréntesis no son necesarios.

En Java, los vectores se indexan comenzando siempre en cero. Así, un vector `a` de tres elementos almacena `a[0]`, `a[1]` y `a[2]`. El número de elementos que puede ser almacenado en un vector `a` se obtiene con `a.length`. Observe que en esta ocasión no hay paréntesis. Un bucle típico para recorrer un vector se basaría en

```
for ( int i = 0; i < a.length; i++ )
```

2.4.1 Declaración, asignación y métodos

Un vector es un objeto, así que dada una declaración de vector

```
int [ ] vector1;
```

Para asignar memoria a un vector se debe emplear `new`.

aún no se ha asignado memoria para guardar el vector. `vector1` es simplemente una referencia de un vector, por lo que en este momento es `null`. Para generar 100 valores de tipo `int`, por ejemplo, aplicaríamos la instrucción `new` como sigue

```
vector1 = new int [ 100 ];
```

Ahora `vector1` referencia un vector de 100 `int`s.

Existen otras formas de declarar vectores. Por ejemplo, en algunos contextos

```
int [ ] vector2 = new int [ 100 ];
```

es aceptable. También puede emplearse listas de inicialización, como en C o C++, para especificar valores iniciales. En el siguiente ejemplo, un vector de cuatro `int`s es almacenado y referenciado por `vector3`.

```
int [ ] vector3 = { 3, 4, 10, 6 };
```

Los corchetes pueden colocarse antes o después del nombre del vector. Situándolos antes es más sencillo ver que el nombre corresponde a un objeto de tipo vector, por lo que éste es el estilo que emplearemos en este texto. La declaración de un vector de objetos (en lugar de tipos primitivos) requiere la misma sintaxis. Obsérvese, sin embargo, que cuando se guarda un vector de objetos, cada objeto almacena inicialmente una referencia `null`. Además cada una de ellas debe redefinirse para que pase a referenciar a un objeto creado. Por ejemplo, un vector de cinco botones se construye como sigue

```
Button [ ] vectorDeBotones;
vectorDeBotones = new Button [ 5 ];
for ( int i = 0; i < vectorDeBotones.length; i++ )
    vectorDeBotones[ i ] = new Button( );
```

Debemos estar seguros de declarar el tamaño correcto del vector. Los errores por salida de rango son muy usuales.

La Figura 2.3 muestra el empleo de los vectores en Java, mediante su aplicación a la modelización del siguiente problema: en la lotería primitiva, cada semana se seleccionan seis números distintos entre 1 y 49 (ambos inclusive). El programa de la figura elige números repetidamente para 1.000 sorteos. La salida es la cantidad de veces que cada número ha sido escogido. La línea 15 declara un vector de enteros que lleva la cuenta de las apariciones de cada número. Como los vectores se indexan comenzando en el cero, el `+1` es crucial. Sin él, tendríamos un vector cuyo rango indexable se extendería desde el 0 hasta el 48, y como consecuencia el acceso al índice 49 estaría fuera de rango. El bucle de las líneas 16 y 17 inicializa las entradas del vector a cero. El resto del programa es relativamente sencillo. Se emplea el tipo `Random` definido en el Capítulo 9. Las llamadas al método `randomInt` producen repetidamente un número aleatorio dentro del rango especificado. La salida de los resultados se genera en las líneas 26 y 27.

Como el tipo vector es un tipo referencia, `=` no copia vectores. En cambio, si `lizq` y `lder` son vectores, el efecto de

```
int [ ] lizq = new int [ 100 ];
int [ ] lder = new int [ 100 ];
   ...
lizq = lder
```

es que el objeto vector referenciado por `lder` está ahora también referenciado por `lizq`. Como consecuencia, el cambio de `lizq[0]` también modifica `lder[0]`. (Para hacer que `lizq` sea una copia independiente de `lder` se debería emplear el método `clone`.)

Por último, un vector puede emplearse como parámetro de un método. El comportamiento que se produce se corresponde con el hecho de que un nombre de vector es una referencia. Consideramos un método `metodoLlamada` que tiene un vector de `int` como parámetro. Las vistas del invocador/invocado son

```
metodoLlamada( vectorActual );          // llamada al método
metodoLlamada( int [ ] vectorFormal )   // declaración del método
```

Los contenidos de un vector se pasan por referencia.

De acuerdo con los convenios sobre el paso de parámetros para los tipos referencia de Java, `vectorFormal` referencia al mismo objeto vector que `vectorActual`. Como consecuencia, `vectorFormal[i]` accede a `vectorActual[i]`.

```
1  import Soporte.Random;
2
3  public class Loteria
4  {
5      // Genera los números de la lotería (1-49)
6      // Imprime la cantidad de ocurrencias de cada número
7
8      public static final int NUMEROS_DIF    =   49;
9      public static final int NUMEROS_JUEGO  =    6;
10     public static final int JUEGOS         = 1000;
11
12     public static void main( String [ ] args )
13     {
14         // Generación de números
15         int [ ] numeros = new int [ NUMEROS_DIF + 1 ];
16         for( int i = 0; i < numeros.length; i++ )
17             numeros[ i ] = 0;
18
19         Random r = new Random( );
20
21         for( int i = 0; i < JUEGOS; i++ )
22             for( int j = 0; j < NUMEROS_JUEGO; j++ )
23                 numeros[ r.randomInt( 1, NUMEROS_DIF ) ]++;
24
25         // Salida de resultados
26         for( int k = 1; k <= NUMEROS_DIF; k++ )
27             System.out.println( k + ": " + numeros[ k ] );
28     }
29 }
```

Figura 2.3 Una simple muestra de vectores.

Esto significa que las variables representadas por el vector indexado son modificables. Éste será siempre el caso. Nótese también que una instrucción como

```
vectorFormal = new int [ 20 ];
```

no tiene ningún efecto sobre `vectorActual`. Finalmente, como los nombres de los vectores no son otra cosa que referencias, pueden devolverse como resultados.

2.4.2 Expansión dinámica de vectores

La *expansión dinámica de vectores* permite asignar memoria a vectores de tamaño arbitrario y hacerlos más grandes cuando sea necesario.

Supongamos que queremos leer una secuencia de números y almacenarlos en un vector para su procesamiento. La propiedad básica de un vector es que debemos especificar su tamaño para que el compilador pueda reservarle la cantidad de memoria adecuada. Además debemos realizar su declaración antes del primer acceso al vector. Si no tenemos idea a priori de cuántos elementos deberemos manejar, es difícil realizar una elección razonable de dicho tamaño. Esta sección muestra cómo expandir vectores si su tamaño inicial resulta ser al final demasiado pequeño. Esta técnica recibe el nombre de *expansión dinámica de vectores* y nos permite almacenar en memoria vectores de cualquier tamaño, agrandándolos o empequeñeciéndolos durante la ejecución del programa.

El método de almacenar vectores en memoria que hemos visto anteriormente es

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

Suponga que después de las declaraciones, observamos que necesitaremos 12 valores de tipo `int` en lugar de 10. En tal caso podemos emplear la siguiente estrategia:

```
int [ ] original = a;       // 1. Se guarda una referencia a a
a = new int [ 12 ];         // 2. Se crea una referencia con más memoria
for ( int i = 0; i < 10; i++ )  // 3. Se copian los datos antiguos
    a[ i ] = original[ i ];
```

Siempre se debe expandir un vector aumentando su tamaño en una constante multiplicativa. La duplicación es una buena decisión en estos casos.

Examinando detenidamente el código anterior llegamos a la conclusión de que ésta es una operación costosa. Esto es debido a que se copian todos los elementos de `original` en `a`. Si, por ejemplo, esta expansión se produce como respuesta a una entrada de datos, sería ineficiente aumentar el vector cada vez que se lean unos pocos datos. Por esta razón, cuando se implementa la expansión de vectores, lo hacemos de modo que el tamaño aumente según una constante multiplicativa. Por ejemplo, en cada paso podemos expandirlo duplicando su longitud. De esta forma, cuando alargamos un vector de N elementos a $2N$ elementos, el coste de N copias puede distribuirse sobre los siguientes N elementos, que pueden ser insertados en el vector sin necesidad de expandirlo.

Para mayor detalle, la Figura 2.4 muestra un programa que lee una cantidad ilimitada de enteros de la entrada estándar y almacena el resultado de la lectura en un vector que se expande dinámicamente. La rutina `reajustar` realiza la expansión del vector (o su contracción), devolviendo una referencia al nuevo vector. De forma análoga, el método `leerEnteros` devuelve una referencia al vector donde se almacenan los datos.

Al principio de `leerEnteros`, `elemLeidos` se inicializa a 0 y se comienza la lectura con un vector de cinco elementos. La lectura reiterada de los datos se realiza en las líneas 28 y 30. Si el vector está lleno, hecho que se comprueba en la línea 31, el vector se expande llamando a `reajustar`. En las líneas 44 a 50 el vector se agranda aplicando la estrategia comentada. En la línea 33, el dato que acaba de ser leído se inserta en el vector y el contador de datos introducidos se incrementa. Si se produce un error en la lectura, se detiene la ejecución del programa. Por último, en la línea 38 se reduce el tamaño del vector para ajustarlo al número de datos leídos.

2.4.3 Vectores multidimensionales

Un *vector multidimensional* es un vector al que se accede empleando varios índices.

En algunas ocasiones es necesario acceder a un vector a través de varios índices. Un ejemplo típico son las matrices. Un *vector multidimensional* es un vector en el que el acceso a un elemento se realiza empleando más de un índice. La cantidad de memoria que se le reserva se determina al especificar el tamaño de sus índices. Accedemos a cada elemento colocando cada índice entre su propio par de corchetes. Por ejemplo, la declaración

```
int [ ] [ ] x = new int [ 2 ] [ 3 ];
```

```
1  import java.io.*;
2  public class LecturaEnteros
3  {
4      public static void main( String [ ] args )
5      {
6          int [ ] vector = leerEnteros( );
7          for( int i = 0; i < vector.length; i++ )
8              System.out.println( vector[ i ] );
9      }
10
11     // Lee una cantidad ilimitada de ints sin recuperación
12     // de errores; devuelve un int [ ]
13     public static int [ ] leerEnteros( )
14     {
15     //BufferedReader se estudia en la Sección 2.6
16         BufferedReader in = new BufferedReader( new
17                                 InputStreamReader( System.in ) );
18         int entradaVal = 0;
19         int [ ] vector = new int[ 5 ];
20         int elemLeidos = 0;
21         String unaLinea;
22
23         System.out.println( "Introduce una cantidad cualquiera" +
24                             "de enteros, uno por línea: " );
25         try
26         {
27             while( ( unaLinea = in.readLine( ) ) != null )
28             {
29                 entradaVal = Integer.parseInt( unaLinea );
30                 if( elemLeidos == vector.length )
31                     vector = reajustar( vector, vector.length * 2 );
32                 vector[ elemLeidos++ ] = entradaVal;
33             }
34         }
35         catch( Exception e ) { } // No se procesan los errores
36         System.out.println( "Lectura finalizada" );
37         return reajustar( vector, elemLeidos );
38     }
39
40     // Reajustar el int[ ] vector; devuelve el nuevo vector
41     public static int [ ] reajustar( int [ ] vector, int nuevoTamanyo )
42     {
43         int [ ] original = vector;
44         int numCopia = Math.min( original.length, nuevoTamanyo );
45
46         vector = new int[ nuevoTamanyo ];
47         for( int i = 0; i < numCopia; i++ )
48             vector[ i ] = original[ i ];
49         return vector;
50     }
51 }
```

Figura 2.4 Código para leer una cantidad ilimitada de `ints` y devolverlos como salida.

define el vector bidimensional `x`, con el primer índice tomando valores entre 0 y 1 y el segundo moviéndose entre 0 y 2 (para guardar un total de seis objetos). El compilador asigna seis posiciones de memoria a dichos objetos.

2.4.4 Argumentos de la línea de comandos

Los argumentos de la línea de comandos se ajustan a los parámetros de `main`.

Los argumentos de la línea de comandos aparecen cuando examinamos el parámetro de `main`. El vector de cadenas de caracteres representa los argumentos adicionales de la línea de comandos. Por ejemplo, cuando se ejecuta el programa

```
java Eco este aquel
```

`args[0]` referencia al `String "este"` y `args[1]` referencia al `String "aquel"`. El programa de la Figura 2.5 implementa el programa `Eco`.

```
1  public class Eco
2  {
3      // Muestra los argumentos de la línea de comandos
4      public static void main( String [ ] args )
5      {
6          for( int i = 0; i < args.length - 1; i++ )
7              System.out.print( args[ i ] + " " );
8          if( args.length != 0 )
9              System.out.println( args[ args.length - 1 ] );
10         else
11             System.out.println( "No hay argumentos que mostrar" );
12     }
13 }
```

Figura 2.5 El comando `eco`.

2.5 Manejo de excepciones

Las excepciones se emplean para señalar eventos excepcionales como, por ejemplo, errores.

Las *excepciones* son elementos que almacenan información y la transmiten fuera de la secuencia normal de retorno de datos. Las excepciones se propagan hacia atrás a través de la secuencia habitual de llamada, hasta que sean *recogidas* por alguna rutina. Se emplean para detectar hechos excepcionales, como por ejemplo, errores.

2.5.1 Procesamiento de excepciones

Un bloque `try` contiene el código que puede generar una excepción.

El código de la Figura 2.6 muestra el empleo de excepciones. El código que eventualmente puede generar una excepción se encierra en el bloque `try`. El bloque `try` se extiende desde la línea 15 hasta la línea 19. Inmediatamente después del bloque `try` se encuentran los manejadores de excepciones. Esta parte del código se ejecuta sólo si se produce una excepción. En el momento en el que ésta se lanza, el bloque `try` del que procede se considera finalizado. Cada bloque `catch` se prueba en orden hasta encontrar un manejador de excepciones adecuado. Como

```
1  import java.io.*;
2
3  public class DividePorDos
4  {
5      public static void main( String [ ] args )
6      {
7          // BufferedReader se discute en la Sección 2.6
8          BufferedReader in = new BufferedReader( new
9                                  InputStreamReader( System.in ) );
10         int x;
11         String unaLinea;
12
13         System.out.println( "Introduce un entero: " );
14         try
15         {
16             unaLinea = in.readLine( );
17             x = Integer.parseInt( unaLinea );
18             System.out.println( "La mitad de x es " + ( x / 2 ) );
19         }
20         catch( Exception e )
21           { System.out.println( e ); }
22         }
23 }
```

Figura 2.6 Un programa sencillo para ilustrar las excepciones.

`Exception` incluye los tipos de todas las excepciones de nuestro interés, recoge cualquier excepción generada por el bloque `try`. Más concretamente, estas excepciones son del tipo `IOException`, generadas por `readLine` si se produce algún error inesperado de lectura, y del tipo `NumberFormatException`, generadas por `parseInt`, si `unaLinea` no es convertible a entero.

Un bloque `catch` procesa una excepción.

Una vez capturada la excepción correspondiente se ejecuta el código del bloque `catch` —en nuestro caso la línea 21—. Entonces este bloque y el que contiene la combinación `try/catch`[1] se consideran terminados. En el ejemplo se imprime un mensaje significativo para el objeto `e` de tipo `Exception`. Como alternativa, podría realizarse un procesamiento adicional o bien darse mensajes de error más detallados.

2.5.2 La cláusula `finally`

La cláusula `finally` siempre se ejecuta antes de la terminación de un bloque, haya o no haya excepciones.

Algunos objetos construidos en el bloque `try` deberían ser eliminados antes de concluir la ejecución del mismo. Por ejemplo, los ficheros que se abren en el bloque `try` necesitan cerrarse antes de abandonar dicho bloque. Uno de los problemas que aparece es que si se lanza una excepción dentro de un bloque `try`, la eliminación podría no producirse al causar la excepción la salida inmediata del bloque `try`. Aunque la eliminación de los objetos puede realizarse justo después del último bloque `catch`, esto sólo es efectivo si la excepción es recogida por alguna cláusula `catch`, y esto es difícil de garantizar con seguridad.

[1] Obsérvese que `try` y `catch` requieren un bloque de instrucciones en lugar de una instrucción única, por lo que las llaves no son opcionales. Algunos compiladores anteriores de Java aceptan, de forma errónea, código que no incluye las llaves necesarias.

La cláusula `finally` que sigue al último bloque `catch` (o al bloque `try` si no hay ningún bloque `catch`) se emplea en estas situaciones. La cláusula `finally` consiste en la palabra clave `finally` seguida del bloque `finally`. Existen tres situaciones básicas:

1. Si el bloque `try` se ejecuta sin generarse excepciones, el control pasa al bloque `finally`. Esto es así aún cuando el bloque `try` termine antes de alcanzar su última instrucción, mediante un `return`, `break` o `continue`.
2. Si se produce una excepción dentro del bloque `try` que no es recogida, el control pasa al bloque `finally`. Tras ejecutar éste la excepción se propaga.
3. Si se produce una excepción dentro del bloque `try` que es recogida en un `catch` posterior, el control pasa al bloque `catch` correspondiente. Tras ejecutarse, se procesa el bloque `finally`.

2.5.3 Excepciones más comunes

En Java existen muchos tipos de excepciones estándar. Las excepciones *run-time* (o en tiempo de ejecución) incluyen eventos como la división-por-cero de enteros y los accesos ilegales a vectores. Como estos eventos pueden producirse casi en cualquier punto, sería agotador incorporar constantemente manejadores de excepciones para ellos. En cualquier caso, si se especifica para ellos un bloque `catch`, estas excepciones se comportan como cualquier otra. Pero si no se especifica ninguno, se lanza una excepción estándar que se propaga de la forma usual, pudiendo sobrepasar `main`. En este caso el programa termina de forma anómala, presentando un mensaje de error. En la Figura 2.7 se muestran algunas de las excepciones *run-time* más usuales.

Las excepciones *standard checked* deben tener un manejador o listarse en la *cláusula* `throws`.

La mayoría de las excepciones son del tipo *standard checked*. Si un método puede lanzar una de estas excepciones directa o indirectamente, el programador debe definir el bloque `catch` adecuado para ella o indicar de forma explícita que

Excepciones Run-time	Significado
`ArithmeticException`	Desbordamiento o división entera por cero.
`NumberFormatException`	Conversión ilegal de un `String` a un tipo numérico.
`IndexOutOfBoundsException`	Acceso a un elemento inexistente de un vector o de un `String`.
`NegativeArraySizeException`	Intento de creación de un vector de longitud negativa.
`NullPointerException`	Intento de uso de una referencia nula.
`SecurityException`	Violación de la seguridad en tiempo de ejecución.

Figura 2.7 Excepciones *run-time* más comunes.

la excepción va a propagarse, incluyendo para esto último la cláusula `throws` en la declaración del método. Observe que antes o después deben definirse manejadores para ellas, ya que el método `main` no debería tener cláusula `throws`. En el Capítulo 4 se muestra cómo definir nuevos tipos de excepciones. En la Figura 2.8 se recogen algunas de las excepciones más usuales de este tipo.

Los *errores* son excepciones irrecuperables.

Los *errores* son excepciones, pero no se ajustan al tipo `Exception`. Típicamente son irrecuperables. El error más común es `OutOfMemoryError`. Para capturar cualquier excepción posible debe recogerse un objeto `Throwable`.

Excepciones Checked	Significado
`java.io.EOFException`	Final de fichero antes de lo esperado.
`java.io.FileNotFoundException`	Fichero inexistente.
`java.io.IOException`	Incluye muchas excepciones de E/S.
`InterruptedException`	Lanzada por el método `Thread.sleep`.

Figura 2.8 Excepciones *standard checked* más comunes.

2.5.4 Las cláusulas `throw` y `throws`

La *cláusula* `throw` se emplea para lanzar una excepción.

El programador puede lanzar una excepción empleando la cláusula `throw`. Por ejemplo, podemos lanzar una excepción de tipo `Exception` mediante

```
throw new Exception( "Malas Noticias" );
```

Normalmente, no se lanzan excepciones del tipo `Exception`, sino que se lanzan excepciones definidas por el usuario. Los detalles de esto se explican en el Capítulo 4.

La *cláusula* `throws` indica las excepciones que se propagan.

Como se ha mencionado anteriormente, las excepciones del tipo *standard checked* deben ser recogidas o propagadas explícitamente hacia la rutina que ha realizado la llamada, pero en algunas ocasiones, y como último recurso, deben procesarse en `main`. Para hacer esto último, el método que no desea capturar la excepción debe indicar, a través de la cláusula `throws`, qué excepciones se van a propagar. La cláusula `throws` se escribe al final de la cabecera del método. La Figura 2.9 muestra un método que propaga las excepciones del tipo `IOException` que encuentra, éstas deben ser recogidas en `main` (ya que no incluiremos una cláusula `throws` en `main`).

2.6 Entrada y salida

`java.io` debe importarse para realizar cualquier E/S no trivial.

La *entrada y salida* (*E/S*) se realiza en Java empleando el paquete `java.io`. La instrucción

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

```
1  import java.io.*;
2
3  public class DemoThrow
4  {
5         public static void procesaFichero( String Fichero )
6                                                 throws IOException
7         {
8         // Esta implementación omitida propaga cualquier
9         // IOException hacia la rutina invocadora
10        }
11
12        public static void main( String [ ] args )
13        {
14              for( int i = 0; i < args.length; i++ )
15              {
16                    try
17                      { procesaFichero( args[ i ] ; }
18                    catch( IOException e )
19                      { System.out.println( e ); }
20              }
21        }
22 ;
```

Figura 2.9 Muestra de la cláusula `throws`.

debe aparecer en cualquier programa que emplee las rutinas de E/S de un modo no trivial. La librería de Java es bastante compleja y tiene multitud de opciones. Examinaremos sólo las más básicas, concentrándonos por completo en la E/S con formato.

2.6.1 Operaciones básicas de E/S

Los canales predefinidos son `System.in`, `System.out` y `System.err`.

Contamos con tres canales predefinidos para realizar la E/S mediante el terminal. `System.in`, que es la entrada estándar, `System.out`, que es la salida usual, y `System.err`, para la salida de errores.

Como ya hemos mencionado, los métodos `print` y `println` se emplean para la salida con formato. Gracias al método `toString`, un valor de cualquier tipo puede convertirse en un `String` que puede imprimirse; en muchos casos esta transformación es automática. A diferencia de C y C++, que tienen multitud de opciones de formato, la salida de Java se realiza, casi exclusivamente, mediante la concatenación de `String`s sin formato alguno.

`BufferedReader` se emplea para la lectura línea a línea de la entrada.

Un modo sencillo de realizar la lectura de la entrada es leer una sola línea, almacenándola en un objeto de tipo `String`, para lo que se emplea `readLine`. Este método lee hasta que encuentra un final de línea o el final del fichero. Los caracteres leídos, salvo el final de línea (en el caso de que se encuentre), se devuelven como un nuevo `String`. Para emplear `readLine`, se debe construir primero un objeto `BufferedReader` sobre un objeto `InputStreamReader`, que a su vez se crea partiendo de `System.in`. Esto se muestra en las líneas 8 y 9 de la Figura 2.6. Observe que las `IOException` son del tipo *standard checked*, por lo que, en algunos casos, deben ser recogidas. En muchas situaciones, se permite que la `IOException` se propague hasta un bloque `catch` del método `main`; esta técnica se ilustra en la Figura 2.9.

2.6.2 El objeto `StringTokenizer`

Recuerde que para leer un solo carácter de tipo primitivo, como un `int`, se debe emplear `readLine` para leer la línea y aplicar después el método adecuado para generar el tipo primitivo a partir del `String`. Para el tipo `int` podemos emplear `parseInt`.

`StringTokenizer` se emplea para obtener cadenas de caracteres delimitadas a partir de una cadena compleja.

En algunos casos nos encontramos con más de un elemento en la misma línea. Por ejemplo, supongamos que cada línea contiene dos valores de tipo `int`. Java tiene definido el objeto `StringTokenizer` que permite separar en `tokens` un `String`. Para usarlo debemos emplear el mandato `import`

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

El empleo de `StringTokenizer` se ilustra en la Figura 2.10. En primer lugar, en la línea 19 construimos un objeto de este tipo, a partir del `String` que representa la línea de entrada. El método `countTokens`, en la línea 20, devuelve el número de tokens del `String`. En este ejemplo deberían ser dos, de no ser errónea la entrada. El método `nextToken` devuelve el siguiente token como un `String`. Este método lanza una `NoSuchElementException` en el caso de no encontrar ningún token, que representa un error y por tanto no debe ser recogido. En las líneas 22 y 23 usamos `nextToken`, seguido de `parseInt`, para obtener un valor de

```
1  import java.io.*;
2  import java.util.*;
3
4  public class MaxTest
5  {
6     public static void main( String args[ ] )
7     {
8        BufferedReader in = new BufferedReader( new
9                                 InputStreamReader( System.in ) );
10       String unaLinea;
11       StringTokenizer str;
12       int x;
13       int y;
14
15       System.out.println( "Introduzca 2 ints en una misma
            línea: " );
16       try
17       {
18          unaLinea = in.readLine( );
19          str = new StringTokenizer( unaLinea );
20          if( str.countTokens( ) != 2 )
21              throw new NumberFormatException( );
22          x = Integer.parseInt( str.nextToken( ) );
23          y = Integer.parseInt( str.nextToken( ) );
24          System.out.println( "Max: " + Math.max( x, y ) );
25       }
26       catch( Exception e )
27         { System.err.println( "Error: se necesitan dos
             enteros" ); }
28    }
29 }
```

Figura 2.10 Programa que muestra el troceo de cadenas de caracteres.

tipo `int`. Todos los errores, incluyendo el producido cuando no se obtienen exactamente dos tokens, son tratados en el bloque `catch`. Como es habitual, con un mayor esfuerzo se podrían generar mejores mensajes de error.

Por defecto, los tokens se separan mediante espacios en blanco. El `StringTokenizer` puede construirse de modo que reconozca otros caracteres como delimitadores, o para considerar dichos delimitadores como tokens.

2.6.3 Ficheros de acceso secuencial

`FileReader` se emplea para leer los ficheros de entrada.

Una de las reglas básicas de Java es que lo que funciona en la E/S con el terminal también funciona en la E/S con ficheros. Para trabajar con un fichero no construimos un objeto `BufferedReader` a partir de un `InputStreamReader`, sino a partir de un `FileReader`, que es creado a su vez especificando el nombre de un fichero.

En la Figura 2.11 se muestra un ejemplo que ilustra estas ideas básicas. Se trata de un programa que imprime el contenido de los ficheros de texto que se

```
1  import java.io.*;
2
3  public class ContenidoFicheros
4  {
5      public static void main( String [ ] args )
6      {
7          if( args.length == 0 )
8              System.out.println( "No se especifican ficheros" );
9          for( int i = 0; i < args.length; i++ )
10             contenidoFichero( args[ i ] );
11     }
12
13     public static void contenidoFichero( String nombreFichero )
14     {
15         FileReader elFichero;
16         BufferedReader lector = null;
17         String unaLinea;
18
19         System.out.println( "Fichero: " + nombreFichero );
20         try
21         {
22             elFichero = new FileReader( nombreFichero );
23             lector = new BufferedReader( elFichero );
24             while( ( unaLinea = lector.readLine( ) ) != null )
25                 System.out.println( unaLinea );
26         }
27         catch( Exception e )
28           { System.out.println( e ); }
29
30         // Cierre del fichero
31         try
32         {
33             if(lector != null )
34                 lector.close( );
35         }
36         catch( IOException e ) { }
37     }
38 }
```

Figura 2.11 Programa que imprime el contenido de un fichero.

indiquen en la línea de comandos. La rutina `main` simplemente recoge los argumentos de la línea de comandos, pasando cada uno a `contenidoFichero`. En `contenidoFichero` se construye el objeto `FileReader` en la línea 22, que se emplea para crear el objeto `BufferedReader` – `lector` – en la línea 23. A partir de este punto la lectura es igual a la ya explicada.

Después de terminar con un fichero, debemos cerrarlo, ya que de otro modo podríamos quedarnos sin canales. Observe que esto no puede hacerse al final del bloque `try`, pues una excepción causaría la salida del bloque prematuramente. En consecuencia, cerramos el fichero tras la secuencia `try/catch`. Como todos los errores recuperables están incluidos en `Exception`, no es necesaria una cláusula `finally`. Estamos seguros de alcanzar la instrucción de cierre de fichero, por si hay más trabajo que hacer.

La salida con formato es similar a la entrada con formato. Como no la usaremos en este texto, remitimos al lector a cualquier referencia de Java si desea conocer los detalles de la misma.

Resumen

En este capítulo hemos estudiado los tipos referencia. Una referencia es una variable que almacena, o bien una dirección de memoria donde se almacena un objeto, o bien una referencia especial `null`. Sólo pueden referenciarse los objetos, aunque un objeto puede ser referenciado por más de una variable. Cuando dos referencias son comparadas vía `==`, el resultado es `true` si ambas referencias apuntan al mismo objeto. De forma similar, `=` hace que una variable referencia apunte a otro objeto. Las referencias pueden manipularse con muy pocas operaciones. La más importante es la representada por el operador punto, que permite la selección de un método de una clase, o bien el acceso a sus atributos internos.

Debido a que en Java sólo existen ocho tipos primitivos, casi cualquier cosa es un objeto accesible a través de una referencia. Esto incluye las cadenas de caracteres, los vectores, los objetos de la clase `Exception`, los canales de entrada y salida de datos y el `StringTokenizer`.

El `String` es un tipo referencia especial pues sobre el mismo están definidos los operadores de concatenación `+` y `+=`. Por lo demás, un `String` se comporta como cualquier otra referencia; así, se necesita `equals` para comprobar si los estados de dos `Strings` son idénticos. Un vector es una colección de elementos del mismo tipo. El vector se indexa partiendo de 0 y se garantiza el chequeo del rango. Los vectores se pueden expandir dinámicamente, usando `new` para asignar mayor cantidad de memoria y copiando después los elementos uno a uno.

Las excepciones se emplean para detectar eventos excepcionales. Una excepción se lanza mediante la cláusula `throw`, y se propaga hasta que es recogida por algún bloque `catch` asociado a un bloque `try`. Excepto para las excepciones en tiempo de ejecución (*run-time exceptions*) y los `Errores`, cada método debe indicar las excepciones que puede propagar empleando la cláusula `throws`.

El `StringTokenizer` se emplea para partir un `String`, obteniendo nuevos `Strings`. Normalmente se emplea junto a otras rutinas de procesamiento de la entrada. La entrada se manipula a través de los objetos `BufferedReader`, `InputStreamReader` y `FileReader`.

El siguiente capítulo muestra cómo diseñar nuevos tipos definiendo una *clase*.

Elementos del juego

agregado Colección de objetos almacenados en la misma unidad.
argumento de la línea de comandos Empleado como parámetro del método `main`.
atributo `length` Usado para conocer la longitud de un vector.
bloque `catch` Se emplea para el procesamiento de excepciones.
bloque `try` Comprende el código que puede generar una excepción.
`BufferedReader` Se emplea para el procesamiento línea a línea de la entrada.
concatenación de cadenas Realizada a través de los operadores `+` y `+=`.
construcción Para los objetos se efectúa mediante la palabra reservada `new`.
entrada y salida (E/S) Implementada en el paquete `java.io`.
`equals` Se emplea para comprobar si los valores de dos objetos almacenados son iguales.
`Error` Excepción irrecuperable.
excepción del tipo *standard checked* Debe ser recogida o declarada explícitamente en la cláusula `throws` del método correspondiente.
excepción *run-time* No necesita ser declarada. Ejemplos notables son `ArithmeticException` y `NullPointerException`.
excepción Sirve para detectar situaciones excepcionales tales como errores.
expansión dinámica de vectores Permite aumentar el tamaño de los vectores cuando ello sea preciso.
`FileReader` Empleado cuando la entrada se efectúa por fichero.
`finally` Se ejecuta siempre antes de salir de un bloque `try/catch`.
inmutable Se dice de un objeto que no puede cambiar de estado. Por ejemplo, los `Strings` son inmutables.
`java.io` Paquete que debe ser importado para producir cualquier salida no trivial.
`java.util` Paquete que debe ser importado para poder emplear el objeto `StringTokenizer`.
`lizq` y `lder` Variables empleadas para indicar los lados izquierdo y derecho, respectivamente.
método `length` Usado para conocer la longitud de una cadena de caracteres.
`new` Empleado para crear un nuevo objeto.
`null` Valor de una variable referencia que no apunta a ningún objeto.
`NullPointerException` Generada cuando se intenta acceder a una referencia `null`.
objeto Entidad no primitiva.
operador de indexación de vectores `[ ]` Permite el acceso a cada elemento de un vector.
operador punto (`.`) Permite el acceso a cada componente de una estructura.
paso por referencia En muchos lenguajes de programación significa que el parámetro formal es una referencia al parámetro actual. Éste es el efecto producido en Java cuando el paso por valor se utiliza sobre un tipo referencia.
recogida de basura Recuperación automática de la memoria que ya no es referenciada.
`String` Objeto especial empleado para almacenar una secuencia de caracteres.
`StringTokenizer` Empleado para obtener `Strings` delimitados a partir de una sola cadena de caracteres.

`System.in`, `System.out` y `System.err` Canales de E/S predefinidos.
`throw` Empleado para lanzar una excepción.
`throws` Indica que un método puede propagar una excepción.
tipo referencia Cualquier tipo no primitivo.
`toString` Convierte un tipo primitivo, o un objeto, en un `String`.
vector Almacena una colección de elementos del mismo tipo.
vector multidimensional Vector cuya manipulación requiere varios índices.

Errores comunes

1. Para tipos referencia y vectores, = no hace una copia de los objetos sino de las referencias.
2. Para tipos referencia y cadenas de caracteres, para comprobar si dos objetos tienen exactamente el mismo estado debe emplearse `equals` en lugar de `==`.
3. Los errores por salida de rango son habituales en todos los lenguajes de programación.
4. Los tipos referencia se inicializan a `null` por defecto; es decir, no se construye ningún objeto nuevo sin haber llamado previamente a `new`. Una «variable referencia sin inicializar» o la `NullPointerException` indica que el programador ha olvidado crear un objeto.
5. En Java los vectores se indexan desde 0 hasta `N-1`, donde `N` es el tamaño del vector. El acceso a componentes fuera de ese rango se detecta de forma automática en tiempo de ejecución.
6. Los vectores bidimensionales se indexan como `A[i][j]` y no como `A[i,j]`.
7. Las excepciones *standard checked* deben ser recogidas, o declaradas en la cláusula `throws` del método correspondiente para permitir su propagación.
8. Debe emplearse `" "` en lugar de `' '`, para mostrar un espacio en blanco.

En Internet

A continuación mostramos los ficheros disponibles para este capítulo. Todos ellos son autocontenidos y no vuelven a utilizarse en el resto del texto. Se encuentran en el directorio **Chapter02**.

DivideBytwo.java	Código correspondiente a la Figura 2.6. Es la versión inglesa de la clase `DividePorDos`.
Echo.java	Código correspondiente a la Figura 2.5. Es la versión inglesa de la clase `Eco`.
ListFiles.java	Código correspondiente a la Figura 2.11. Es la versión inglesa de la clase `ContenidoFicheros`.
Lottery.java	Código correspondiente a la Figura 2.3. Es la versión inglesa de la clase `Loteria`.
MaxTest.java	Código correspondiente a la Figura 2.10. Es la versión inglesa de la clase `MaxTest`.
ReadInts.java	Código correspondiente a la Figura 2.4. Es la versión inglesa de la clase `LecturaEnteros`.

Ejercicios

Cuestiones breves

2.1. Enumere las principales diferencias entre tipos referencia y tipos primitivos.
2.2. Enumere cinco operaciones que pueden ser aplicadas a los tipos referencia.
2.3. Describa cómo funcionan las excepciones en Java.
2.4. Enumere las principales operaciones que pueden realizarse con cadenas de caracteres.

Problemas teóricos

2.5. Si `x` e `y` tienen los valores 5 y 7, respectivamente, determine la salida de la secuencia siguiente:

```
System.out.println( x + ' ' + y );
System.out.println( x + " " + y );
```

Problemas prácticos

2.6. Un *testsum* es el entero 32-bit calculado sumando todos los caracteres Unicode de un fichero (se permite `DesbordamientoInferior` silencioso, aunque éste es improbable cuando todos los caracteres son ASCII). Dos ficheros iguales tienen el mismo testsum. Escriba un programa que calcule el testsum de un fichero cuyo nombre se introducirá en la línea de comandos.
2.7. Modifique el programa de la Figura 2.11 de modo que no se introduzca ningún parámetro por la línea de comandos, sino que se utilice la entrada estándar.
2.8. Escriba un método que devuelva `true` si `String str1` es un prefijo de `String str2`. No debe emplear ninguna rutina de búsqueda de cadenas de caracteres, excepto `charAt`.

Prácticas de programación

2.9. Escriba un programa que muestre la cantidad de caracteres, palabras y líneas de los ficheros que se indiquen en la línea de comandos.
2.10. Implemente un programa de copia de ficheros. Incluya un test que se asegure de que el fichero de origen y el de destino son diferentes. Necesitará leer alguna referencia sobre Java para aprender a programar la salida de datos en fichero.

Referencias

Se puede encontrar más información en las referencias listadas al final del Capítulo 1.