Programación Avanzada
UD7

Autor:
P. Pablo Garrido Abenza pgarrido@umh.es

Universidad Miguel Hernández

01 de Octubre de 2012

ÍNDICE GENERAL

CAPÍTULO 7

COLECCIONES

Una **colección** es una lista de objetos, a los que se les denomina **elementos**. Tienen un uso similar a los **arrays**, aunque con varias diferencias, la principal es que no ocupan un tamaño fijo especificado durante su creación, sino que su tamaño es dinámico. En las colecciones se pueden insertar elementos, extraer, eliminar, consultar, etc.

Java incluye un *Collections Framework*, que proporciona una arquitectura unificada para manipular colecciones, independientemente del tipo concreto de colección que se utilice. Esto tiene la ventaja de facilitar el aprendizaje y la implementación de nuevas colecciones. Está formado por varias interfaces y clases definidas dentro del paquete `java.util`, siendo `Collection` la interface raíz, de la que heredan varias subinterfaces (`Set`, `Queue`, `Deque`, `List`, ...). Junto con estas interfaces, se tienen algunas clases que las implementan como `ArrayList`, `ArrayDeque`, `LinkedList`, ... y algunas otras heredadas de versiones anteriores a Java 1.2 como la clase `Vector`, `Stack` o `HashTable`. La figura 7.1 presenta la jerarquía de *interfaces* (en cursiva) de la *Collections Framework*, así como algunas de las **clases** (en negrita) que las implementan.

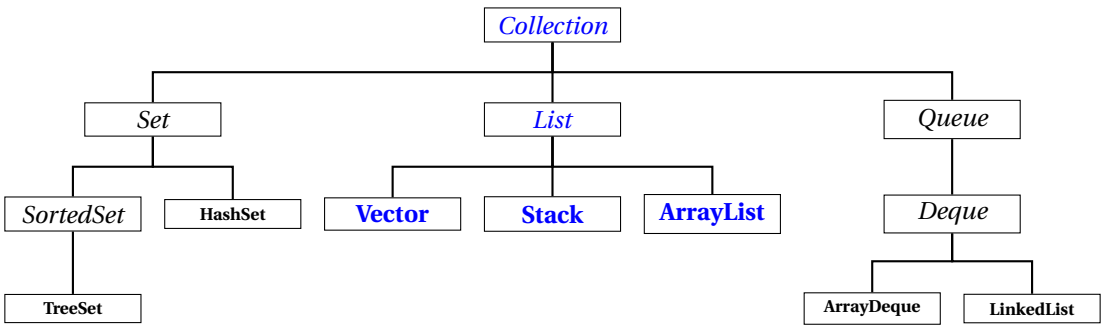

Figura 7.1: *Collection Framework*: jerarquía de *interfaces* y **clases**

Interface `Collection`

La interface `Collection` es la interface raíz del resto de interfaces de la *Collection Framework*, y es implementada por una serie de clases. Por tanto, conociendo esta interface y alguna subinterface como `List` o `Queue`, ya conoceremos el comportamiento de todas las clases de la *Collection Framework*.

Muy en resumen, algunos métodos disponibles en la interface `Collection` son: `add()`, `clear()`, `contains()`, `equals()`, `isEmpty()`, `iterator()`, `remove()`, `removeAll()`, `toArray()`.

Interface `List`

Esta interface hereda de la interface `Collection`, por lo que incluye todos los métodos que acabamos de mencionar, así como alguno nuevo: `get()`, `indexOf()`, `set()`.

La interface List es implementada por varias clases, entre las que destacamos las clases `Vector`, `Stack`, y `ArrayList`.

Otras interfaces y clases

Existen otras interfaces que también heredan de la interface `Collection` (más de las que se muestran en la figura 7.1), y cada una de ellas es implementada por una serie de clases.

Con la versión Java 6 se han incluido nuevas interfaces a la *Collections Framework*, como la `Deque`, `BlockingDeque`, `NavigableSet`, `NavigableMap`, y `ConcurrentNavigableMap`. La diferencia entre estas colecciones es que algunas de ellas permiten elementos duplicados y otras no, o llevan un orden y otras no, permiten su acceso de forma concurrente o no, etc., pero todas ellas tienen una arquitectura unificada para la gestión de colecciones, independientemente de su implementación interna. Por tanto, se remite al alumno a la documentación de la librería de clases de Java (API Specification) para ampliar información.

Objetivos del capítulo

Debido a la gran cantidad de interfaces y clases de la `Collection Framework`, en este capítulo nos vamos a centrar en la interface `List` y algunas de las clases que la implementan, las cuales están señaladas en color azul en la figura 7.1):

- `Vector`: clase heredada que implementa un array de tamaño dinámico (*deprecated*).
- `Stack`: clase heredada que implementa una pila *last-in-first-out* (LIFO) de objetos (deprecated).
- `ArrayList`: lista secuencial de elementos muy similar a la clase `Vector`.

Por tanto, comenzaremos explicando la clase `Vector`, los métodos que implementa, y las distintas formas de recorrer un objeto de esa clase. Después veremos la clase `Stack`, que hereda de `Vector`. Finalmente, se explicará la clase `ArrayList`, que como también implementa la interface `List`, la forma de uso de esa clase será idéntica a la clase `Vector`, cambiando únicamente la implementación interna.

Veremos también la forma de almacenar datos de tipo primitivo mediante las llamadas **clases envoltura**, así como el uso de una nueva característica que nos facilita el trabajo llamada *autoboxing* / *unboxing*.

Para finalizar, se presentará también el uso de **clases genéricas** o parametrizadas (*generics*), que también nos facilita el trabajo con colecciones.

7.1. Clase `Vector`

Un objeto de la clase `Vector` es una lista de objetos que se utiliza de forma similar a un *array*, accediendo a sus elementos por medio de un índice. Sin embargo, su tamaño es dinámico en vez de fijo, es decir, que podemos agregarle nuevos elementos mientras tengamos memoria libre y eliminar otros, ampliándose o encogiéndose como sea necesario.

La clase `Vector` es obsoleta (*deprecated*), aunque se puede seguir utilizando. De hecho, cuando apareció la *Collections Framework* se actualizó para integrarla en ella, haciendo que implementase la interface `List`. Por mantener compatibilidad hacia atrás, se mantuvieron los métodos anteriores, por lo que en muchos casos hay métodos duplicados que realizan la misma función; lógicamente conviene utilizar los métodos nuevos de la interface `List`. La tabla 7.1 muestra una equivalencia de algunos métodos antiguos frente a los nuevos de la interface `List`.

Método antiguo	Método de la interface `List`
`addElement()`	`add()`
`insertElementAt()`	`add()`
`elementAt()`	`get()`
`setElementAt()`	`set()`
`removeElementAt()`	`remove()`
`removeAllElements()`	`clear()` o `removeAll()`

Cuadro 7.1: Clase `Vector` - métodos antiguos

La ventaja de utilizar los métodos de la interface `List` es que son más fáciles de recordar, y sobre todo, que cualquier otra clase que también implemente dicha interface se utilizará de la misma forma, aunque internamente sean distintas. Por ejemplo, la clase `ArrayList`, la cual se explicará en la sección 7.4, también implementa la interface `List`, por lo que conociendo la clase `Vector` también sabremos utilizar dicha clase.

Los **elementos** que puede almacenar un objeto `Vector` son siempre de la clase `Object` o subclases de ella; puesto que la clase `Object` es la clase raíz de la jerarquía de clases de Java, lo que esto quiere decir es que podremos almacenar objetos de cualquier clase. Esto no incluye, sin embargo, a los datos de tipo primitivo (`int`, `float`, `double`, ...), pero aunque no se pueden almacenar directamente, sí que se puede haciendo uso de las *clases envoltura* (`Integer`, `Float`, `Double`, ...). Las *clases envoltura*, así como las nuevas características de *autoboxing* / *unboxing* que automatizan la tarea se explican en la sección 7.5.

NOTA: la clase `Vector` ha sido marcada como obsoleta (*deprecated*), recomendándose el uso de la clase `ArrayList`. Desde el punto de vista del programador su uso es el mismo, es decir, ambas tienen las mismas operaciones; implementan la interface `List`. No obstante, la clase `Vector` es más segura a la hora de sincronizar operaciones en aplicaciones concurrentes (multihilo), sus métodos son sincronizados (`synchronized`). Por otro lado, la clase `ArrayList` es más eficiente, ya que no realiza bloqueos al no ser una clase sincronizada (`synchronized`), teniendo más rendimiento.

Elementos de la clase

La clase `Vector` ofrece los constructores y métodos mostrados en las tablas 7.2 y 7.3, respectivamente. Comentar que hay algunos métodos con nombre diferente para realizar la misma tarea; en la tabla se muestra sólo uno de ellos, pero indicando el equivalente en su descripción.

Constructor	Descripción
`Vector()`	Construye un `Vector` con una capacidad inicial de 10, duplicando su tamaño cada vez que se alcance su capacidad.
`Vector(int initialCapacity)`	Construye un `Vector` con la capacidad inicial especificada, duplicando su tamaño cada vez que se alcance su capacidad.
`Vector(int initialCapacity, int capacityIncrement)`	Construye un `Vector` con la capacidad inicial especificada, y ampliando su capacidad por la cantidad de elementos especificada cada vez que se llene.

Cuadro 7.2: Clase `Vector` - constructores

Método	Descripción
`add(Object o)`	Añade un nuevo elemento al final del `Vector`.
`add(int index, Object o)`	Añade un nuevo elemento en el `Vector`, en la posición indicada.
`capacity()`	Retorna la capacidad el `Vector`.
`clear()`	Elimina todos los elementos del `Vector`; equivalente a `removeAll()`.
`contains(Object o)`	Retorna `true` si el elemento `o` ya está en el `Vector`; `false` en caso contrario. Es útil para averiguar si un objeto ya está insertado en un `Vector` sin necesidad de escribir un bucle para recorrerlo de forma manual.
`indexOf(Object o)`	Retorna el índice de la posición del objeto `o` en el Vector.
`get(int index)`	Obtener el elemento del `Vector` de la posición especificada. Si el índice está fuera de rango (`index<0` ó `index>=size()`) se generará la excepción `ArrayIndexOutOfBoundsException`. Suele ser obligatorio realizar *casting* para transformar el `Object` extraído.
`set(int index, Object o)`	Asigna el objeto `o` en la posición `index`, sustituyendo al ya existente.
`elements()`	Retorna un objeto `Enumeration` que permite recorrer todos los elementos.
`iterator()`	Retorna un objeto `Iterator` que permite recorrer todos los elementos.
`isEmpty()`	Retorna `true` si el `Vector` está vacío (no tiene ningún elemento). Alternativamente podría comprobarse también si el método `size()` retorna 0.
`remove(int index)`	Elimina el elemento de la posición indicada.
`remove(Object o)`	Se buscará el objeto especificado y lo elimina si lo encuentra.
`removeAll()`	Elimina todos los elementos del `Vector`; equivalente a `clear()`.
`size()`	Retorna el tamaño del `Vector`, es decir, el número de elementos (no confundir con su *capacidad*).
`toArray()`	Crear un *array* con los elementos del `Vector`, siendo los elementos de la clase `Object`.
`toString()`	Retorna una representación en forma de cadena del `Vector`. Es útil para imprimir el contenido de un `Vector` con una sóla línea.
`trimToSize()`	Ajustar la capacidad según el número de elementos actual (tamaño).

Cuadro 7.3: Clase `Vector` - métodos

Lo más importante a tener en cuenta al extraer un objeto de un `Vector` mediante `get()` o `elementAt()` es que suele ser necesario hacer *casting* a la clase concreta del objeto que se insertó. A la hora de insertar un objeto con `add()` no es necesario realizar *casting* a `Object`, ya que todo objeto es un `Object` (relación de herencia *es-un*), es decir, cualquier clase es subclase de `Object`; sin embargo, a la inversa no es cierto, por lo que explícitamente es obligatorio realizar la conversión.

Concepto de capacidad

Un objeto `Vector` siempre intenta optimizar el uso de memoria, pero también el rendimiento. Puesto que no sería muy eficiente ir reservando la memoria justa para cada nuevo elemento insertado (lo cual es un proceso lento), lo que se hace es reservar memoria para más elementos de lo necesario, y cuando se llene, reservar de nuevo otro bloque de memoria. Al número máximo de elementos que admitiría un `Vector` sin requerir la reserva de más memoria adicional se le conoce como *capacidad*; en otras palabras, un vector contiene siempre un número de elementos (tamaño) inferior o igual a su capacidad. Cuando hay capacidad suficiente para añadir un nuevo elemento el proceso es muy rápido, mientras que cuando se alcanza la capacidad y es necesario la reserva de memoria adicional es un proceso más lento, pero al hacerlo de este modo, esta situación no ocurre muy frecuentemente.

Relacionados con la capacidad existe el concepto de *incremento de capacidad* (`capacityIncrement`), que indica cuánto debe ampliarse la memoria reservada para el `Vector` cuando se alcanza su capacidad. También tenemos la posibilidad de establecer la *capacidad inicial* (`initialCapacity`), que indicaría el número máximo de elementos que admitiría tras la creación del objeto `Vector` antes de requerir una ampliación de capacidad. Estos dos parámetros, `initialCapacity` y `capacityIncrement` son los que pueden especificarse, de forma opcional, a la hora de crear el `Vector`, pues la clase dispone de los constructores mostrados en la tabla 7.2. Cuando no se especifica `initialCapacity`, la capacidad inicial por defecto es de 10; cuando no se especifica `capacityIncrement` (o es menor o igual que 0), la política para expandir el `Vector` consiste en duplicar su tamaño cada vez que se llene su capacidad.

Una vez creado el `Vector`, podemos actuar sobre la capacidad mediante varios métodos. Por ejemplo, si hemos eliminado muchos elementos del `Vector`, o con la política de crecimiento por defecto (doblando la capacidad) sospechamos que se está ocupando mucha más memoria de la necesaria, podemos "recortar" el espacio de más ocupado por el `Vector` para que ocupe justo lo necesario con el método `trimToSize()`; esto hace que la capacidad (elementos para los que ha reservado memoria) sea la misma que su tamaño (elementos que realmente están en ese momento en el `Vector`). También podemos modificar la capacidad a otro valor distinto con el método `ensureCapacity()`, que nos permite asegurar que se tiene capacidad al menos para el número de elementos especificado. También podemos consultar el número de elementos (tamaño) con `size()`, o la capacidad con `capacity()`.

Sin embargo, el concepto de *capacidad* o el uso de estos dos métodos no es muy relevante, puesto que el uso de memoria que haga un objeto `Vector` es transparente para el programador, es decir, que a menos que el consumo de memoria sea crítico, podemos simplemente utilizar el primer constructor, e ignorar los métodos explicados en este apartado.

Ejemplo de uso

En este apartado vamos un programa que ilustre el uso de la clase `Vector`. Primero se muestra como se crea un `Vector`, y mediante un primer bucle se le añaden unas cadenas de caracteres (`String`) del tipo ‘‘`Valor 1`’’, ‘‘`Valor 2`’’, ... Después, en el segundo bucle se vuelve a recorrer el `Vector` para extraer esas cadenas y mostrarlas por pantalla.

```
1  import java.util.Vector;
2
3  public class VectorDemo1 {
4
5     public static void main (String args[]) {
6        Vector v;
7        int    n;
8        String cadena;
9
10       // Creamos el vector
11       v = new Vector();
12       // Generamos al azar el número de objetos String a insertar (1..10)
13       n = (int) (Math.random() * 10) + 1;
14       // Añadimos varias cadenas al Vector
15       for (int i=0; i<n; i++) {
16          cadena = "Valor " + (i+1);
17          v.add (cadena);
18       }
19       // Extraemos las cadenas y las mostramos por pantalla.
20       for (int i=0; i<v.size(); i++) {
21          // Extraemos el objeto de la posición i (String)
22          cadena = (String) v.get(i);  // Casting necesario
23          // Mostramos la cadena
24          System.out.println (cadena);
25       }
26    }
27 }
```

Cada vez que ejecutemos el programa obtendremos resultados diferentes, ya que el número de cadenas a generar e insertar en el `Vector` es aleatorio. En este caso, los posibles valores que podremos obtener en la variable `n` será entre 1 y 10 (línea 13). Por ejemplo, si obtuviéramos un valor `n=3`, las cadenas generadas serían: ‘‘`Valor 1`’’, ‘‘`Valor 2`’’ y ‘‘`Valor 3`’’.

```
Valor 1
Valor 2
Valor 3
Valor 4
Valor 5
```

```
Valor 1
Valor 2
Valor 3
```


Notar el uso de los paréntesis para sumar 1 al valor de `i` en el primer bucle, ya que parte de 0 (línea 16); de no poner los paréntesis el operador + actuaría como concatenación de cadenas en vez de suma de enteros, resultando en una cadena construida con 3 partes: el texto ‘‘`Valor `’’, el valor de `i` (convertido a cadena automáticamente), y el valor `1` (convertido a cadena automáticamente), resultando en las cadenas: ‘‘`Valor 01`’’, ‘‘`Valor 11`’’, . . . , lo cual no es lo deseado.

En el segundo bucle se recorre todas las cadenas insertadas previamente, y utilizamos `v.size()` como límite del bucle (línea 20). Aunque en este caso particular también podríamos haber utilizado `n`, ya que esta variable indica el número de elementos insertados en el bucle, normalmente no tendremos ninguna variable que nos diga el tamaño del `Vector`, para eso tenemos el método `size()`.

Lo más destable de este segundo bucle es el *casting* obligatorio que debemos hacer cuando extraemos elementos de un `Vector` (línea 22), puesto que lo que se extrae es un `Object`, y no podemos asignar un `Object` a un `String` sin hacer la conversión explícita. El motivo es que cualquier objeto de cualquier clase (por ejemplo `String`) siempre es un `Object` (será subclase de `Object`), pero a la inversa no siempre es cierto: algunos `Object` serán `String`, pero no todos. Sin embargo, existe una nueva característica (desde Java 5) llamada *generics* o clases parametrizadas que evita justamente el uso de este *casting*, simplificando el código y evitando posibles errores en tiempo de ejecución (ver la sección 7.6).

7.2. Recorrer una `Collection`

Existen varias alternativas a la hora de recorrer los elementos de una colección, las cuales se describen a continuación:

1. Mediante un bucle tradicional y los métodos `size()` y `get()`.
2. Mediante un bucle `for-each`.
3. Mediante una `Enumeration`.
4. Mediante un `Iterator`.

Puesto que hasta ahora la única colección explicada ha sido la clase `Vector`, los ejemplos utilizan dicha clase, pero los métodos que se explican en este apartado se pueden utilizar en cualquier colección en general.

Bucle tradicional y métodos `size()` y `get()`

Este método es el que ya se ha mostrado en el ejemplo anterior, en el que se utiliza el método `size()` para conocer el número de elementos. Después se accede a cada elemento mediante `get()` con un índice, siendo necesario realizar `casting` en la mayoría de casos. Simplificando, se trataría de hacerlo como se muestra en el siguiente programa:

```
1  import java.util.Vector;
2
3  public class VectorDemo2 {
4
5     public static void main (String args[]) {
6        // Creamos un objeto Vector y le añadimos varios elementos
7        Vector v = new Vector();
8        v.add ("Uno");
9        v.add ("Dos");
10       // Recorremos todos los elementos del Vector y los mostramos
11       for (int i=0; i<v.size(); i++) {
12          String s = (String) v.get(i);
13          System.out.println (s);
14       }
15    }
16 }
```


Bucle `for-each`

Desde la versión Java 5, existe una nueva estructura de control de flujo especialmente indicada para recorrer *arrays* y objetos `Collection`: la estructura `for-each`. Este nuevo bucle `for-each`, el cual se explicó en un capítulo anterior, es válido para *arrays*, para la clase `Vector`, y para cualquier otro tipo de colección (`ArrayList`, ...).

```
1  import java.util.Vector;
2
3  public class VectorDemo4 {
4
5     public static void main (String args[]) {
6        // Creamos un objeto Vector y le añadimos varios elementos
7        Vector v = new Vector();
8        v.add ("Uno");
9        v.add ("Dos");
10       // Recorremos todos los elementos del Vector y los mostramos
11       for (Object o : v) {
12          String s = (String) o;
13          System.out.println (s);
14       }
15    }
16 }
```

Con esta alternativa se obtiene un código más compacto; si además utilizásemos clases parametrizadas o genéricas (*generics*), aun nos ahorraríamos tener que hacer los *casting* al extraer los objetos del `Vector` (ver sección 7.6).

Mediante `Enumeration`

Existe una forma alternativa de recorrer los elementos de un `Vector` que consiste en utilizar un objeto de la clase `Enumeration`, el cual se obtiene utilizando el método `elements()` de un `Vector`. Una vez obtenido dicho objeto, podemos recorrer el `Vector` desde el primer elemento hasta el último, en el orden en que fueron insertados. Para ello sólo es necesario hacer un bucle cuya condición de finalización sea mientras queden elementos por recorrer, cosa que se averigua mediante el método `hasMoreElements()`. En cada iteración se obtendrá el siguiente elemento mediante el método `nextElement()`, que al igual que el método `get()` de la clase `Vector`, requiere hacer *casting* a la clase que se almacenó. La tabla 7.4 describe estos dos métodos.

Constructor	Descripción
`hasMoreElements()`	Averiguar si aun quedan elementos por recorrer en la enumeración. Retorna `true` si aun queda al menos un elemento y `false` si ya hemos llegado al final.
`nextElement()`	Retorna el siguiente elemento del `Vector` y avanza el puntero para la siguiente vez que se invoque este método. Siempre debemos comprobar antes con el método `hasMoreElements()` si quedan elementos por recorrer, de lo contrario podremos obtener una excepción `NoSuchElementException` si ejecutamos `nextElement()` y ya habíamos llegado al final.

Cuadro 7.4: Clase `Enumeration` - métodos

El programa anterior modificado para utilizar una enumeración quedaría así:

```
1  import java.util.*;     // Vector y Enumeration
2
3  public class VectorDemo3 {
4
5     public static void main (String args[]) {
6        // Creamos un objeto Vector y le añadimos varios elementos
7        Vector v = new Vector();
8        v.add ("Uno");
9        v.add ("Dos");
10       // Recorremos todos los elementos del Vector y los mostramos
11       Enumeration e = v.elements();
12       while ( e.hasMoreElements() ) {
13          String s = (String) e.nextElement();
14          System.out.println (s);
15       }
16    }
17 }
```

Notar que al extraer los elementos del `Vector` (línea 13) ahora no se utiliza el método `get()` de la clase `Vector` sino el `nextElement()` de la clase `Enumeration`. En ambos casos es obligatorio el uso de *casting* por las razones expuestas anteriormente.

Mediante `Iterator`

La interface `Iterator` sustituye a la clase de `Enumeration` dentro del *Collections Framework*. Con un objeto `Iterator` podemos recorrer un `Vector` (o cualquier `Collection` en general) de una forma muy similar a un objeto `Enumeration`, aunque los nombres de los métodos han cambiado, como se puede apreciar en la tabla 7.5. Vemos que además incorpora el método `remove()` para eliminar el último elemento extraído.

Constructor	Descripción
`hasNext()`	Averiguar si aun quedan elementos por recorrer. Retorna `true` si aun queda al menos un elemento y `false` si ya hemos llegado al final.
`next()`	Retorna el siguiente elemento de la colección y avanza el puntero para la siguiente vez que se invoque este método. Siempre debemos comprobar antes con el método `hasNext()` si quedan elementos por recorrer, de lo contrario podremos obtener una excepción `NoSuchElementException` si ejecutamos `next()` y ya habíamos llegado al final.
`remove()`	Elimina de la colección el último elemento retornado con el método `next()`.

Cuadro 7.5: Clase `Iterator` - métodos

El siguiente ejemplo utiliza un `Iterator` para recorrer el `Vector`:

```
1  import java.util.*;     // Vector e Iterator
2
3  public class VectorDemo6 {
4
5     public static void main (String args[]) {
6        // Creamos un objeto Vector y le añadimos varios elementos
7        Vector v = new Vector();
8        v.add ("Uno");
9        v.add ("Dos");
10       // Recorremos todos los elementos del Vector y los mostramos
11       Iterator e = v.iterator();
12       while ( e.hasNext() ) {
13          String s = (String) e.next();
14          System.out.println (s);
15       }
16    }
17 }
```

Cuando se explicó cómo dividir una cadena de caracteres `String`, una de las opciones que teníamos era utilizar la clase `Scanner`. El ejemplo que se mostró utilizaba los métodos explicados aquí (`hasNext()` y `next()`), ya que esta clase también implementa la interface `Iterator`. Por tanto, se aconseja utilizar `Iterator` en lugar de `Enumeration` para nuevos desarrollos.

Por otro lado, la ventaja de utilizar una `Enumeration` o un `Iterator` frente al uso de bucles (tanto el `for` tradicional como el `for-each`), es que se pueden recorrer varias colecciones en paralelo.

7.3. Clase `Stack`

Se trata de una clase que hereda de la clase `Vector`, extendiendo dicha clase con 5 operaciones típicas para una estructura pila *last-in-first-out* (LIFO). Tan sólo tiene un constructor por defecto, es decir, sin parámetros.

Método	Descripción
`empty()`	Averiguar si la pila está vacía. Retorna `true` si no tiene ningún elemento; `false` en caso contrario.
`peek()`	Consultar el elemento de la cima de la pila, pero sin eliminarlo de ella.
`pop()`	Obtener el elemento de la cima de la pila, eliminándolo de ella.
`push(Object e)`	Apilar un elemento en la cima de la pila.
`search(Object e)`	Retorna la posición (profundidad) de un objeto en la pila, estando el elemento de la cima en la posición 1.

Cuadro 7.6: Clase `Stack` - métodos

El siguiente **ejemplo** apila una serie de palabras, y luego las desapila en orden inverso en el que se apilaron (FIFO), tal como puede observarse en la salida del programa.

```
1  import java.util.Stack;
2
3  public class StackDemo1 {
4
5     public static void main (String args[]) {
6        String[] palabras = { "En", "un", "lugar",
7                              "de", "La", "Mancha" };
8
9        Stack pila = new Stack();
10       // Apilamos todos los elementos del array
11       System.out.print ("Apilamos:    ");
12       for (String s : palabras) {
13          System.out.print (s + " ");
14          pila.push(s);
15       }
16       System.out.print ("\r\nDesapilamos: ");
17       while (!pila.empty()) {
18          String s = (String) pila.pop();
19          System.out.print (s + " ");
20       }
21       System.out.println ("");
22    }
23 }
```

```
Apilamos:    En un lugar de La Mancha
Desapilamos: Mancha La de lugar un En
```


¿Podría recorrerse una pila `Stack` mediante los otros mecanismos expuestos en el apartado anterior? Sí, sería posible con un bucle `for-each`, una Enumeration, o un `Iterator`, recorriéndose los elementos en el mismo orden que se apilaron. Por ejemplo, con un `Iterator` sería:

```
1  import java.util.Stack;
2  import java.util.Iterator;
3
4  public class StackDemo2 {
5
6     public static void main (String args[]) {
7        String[] palabras = { "En", "un", "lugar",
8                              "de", "La", "Mancha" };
9
10       Stack pila = new Stack();
11       // Apilamos todos los elementos del array
12       System.out.print ("Apilamos:   ");
13       for (String s : palabras) {
14          System.out.print (s + " ");
15          pila.push(s);
16       }
17       System.out.print ("\r\nRecorremos: ");
18       Iterator e = pila.iterator();
19       while ( e.hasNext() ) {
20          String s = (String) e.next();
21          System.out.print (s + " ");
22       }
23       System.out.println ("");
24    }
25 }
```

```
Apilamos:   En un lugar de La Mancha
Recorremos: En un lugar de La Mancha
```

Al igual que la clase `Vector`, esta clase es obsoleta (*deprecated*), recomendándose para nuevos desarrollos el uso de la interface `Deque` y clases que la implementan como `ArrayDeque` introducidas en Java 6. Dicha clase es más robusta y completa que la clase `Stack`. Por ejemplo: `ArrayDeque pila = new ArrayDeque();`

7.4. Clase `ArrayList`

Se trata de un array redimensionable que implementa la interface `List`, al igual que la clase `Vector` y `Stack` (por heredar de `Vector`). Por tanto, como hemos comentado en el apartado anterior, el uso de la clase `ArrayList` será similar, ya que tiene las mismas operaciones.

Al igual que la clase `Vector`, se puede acceder a los elementos por medio de un índice. La diferencia entre `ArrayList` y `Vector` es que la clase `ArrayList` no es sincronizada, es decir, si en un programa multihilo se accede a un objeto `ArrayList` desde varios hilos simultáneamente, los resultados pueden no ser correctos. Sin embargo, para la mayoría de los programas que no son multihilo, la ejecución de un `ArrayList` será más eficiente que su correspondiente `Vector`.

A continuación se muestran algunos **ejemplos** de uso de la clase `ArrayList`, que como podrá comprobarse, no hay diferencia respecto la clase `Vector`, incluyendo todas las posibles formas de recorrer la colección. En el siguiente **ejemplo** recorremos un objeto `ArrayList` mediante un bucle `for-each` (izquierda) y mediante un `Iterator` (derecha).

```java
import java.util.ArrayList;

public class ArrayListDemo1 {

    public static void main (String
        args[]) {
        ArrayList v;

        v = new ArrayList();
        v.add ("Uno");
        v.add ("Dos");
        // Mediante for-each
        for (Object o : v) {
            String s = (String) o;
            System.out.println (s);
        }
    }
}
```

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

public class ArrayListDemo2 {

    public static void main (String
        args[]) {
        ArrayList v;

        v = new ArrayList();
        v.add ("Uno");
        v.add ("Dos");
        // Mediante un Iterator
        Iterator e = v.iterator();
        while (e.hasNext()) {
            Object o = e.next();
            String s = (String) o;
            System.out.println (s);
        }
    }
}
```

```
Uno
Dos
```

En ambos casos estamos recorriendo la colección del primero al último. Este es el comportamiento por defecto de la interface `Iterator`. Si quisiéramos recorrer la colección en orden inverso tendríamos que utilizar la subinterface `ListIterator`, ya que además de los métodos `hasNext()` y `next()` incorpora los métodos `hasPrevious()` y `previous()`, pudiendo recorrer una colección en cualquier orden.

El método `listIterator()` permite especificar de forma opcional la posición inicial desde la que recorrer el vector: si no se especifica nada será al principio (0), que será el que obtengamos al llamar a `next()`. Sin embargo, para que empiece por el último debemos especificar `size()`, de tal manera que al llamar a `previous()` nos retorne el elemento de la posición `size()-1`, que es el último.

```
1  import java.util.ArrayList;
2  import java.util.ListIterator;
3
4  public class ArrayListDemo3 {
5
6     public static void main (String args[]) {
7        ArrayList v;
8
9        v = new ArrayList();
10       v.add ("Uno");
11       v.add ("Dos");
12       // Mediante un ListIterator
13       ListIterator e = v.listIterator (v.size());
14       while (e.hasPrevious()) {
15          Object o = (String) e.previous();
16          String s = (String) o;
17          System.out.println (s);
18       }
19    }
20 }
```

Ahora vemos que se recorre desde el final:

```
Dos
Uno
```


7.5. Almacenamiento de datos de tipos primitivos

Como se ha mencionado anteriormente, las colecciones permiten almacenar objetos, por tanto, en principio no podríamos almacenar datos de tipo primitivo (`int`, `float`, `double`, `char`, ...). Java nos ofrece la posibilidad de convertir un dato o variable de tipo primitivo a un objeto mediante las denominadas **clases envoltura**, existiendo una para cada tipo primitivo (ver tabla 7.7). Todas estas clases pertenecen al paquete `java.lang`, el cual se importa automáticamente.

Tipo primitivo	Clase envoltura
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Char`
`boolean`	`Boolean`

Cuadro 7.7: Clases envoltura

Todas estas clases consisten en una clase que envuelve un dato de tipo primitivo, junto con varias constantes, constructores necesarios, y métodos útiles para convertir de ese tipo primitivo a `String` y viceversa. Por ejemplo, la clase `Integer` (que utilizaremos en algunos ejemplos de este apartado) encapsula una variable del tipo primitivo `int`, y sus elementos más utilizados se muestran en la tabla 7.8. El resto de clases son muy similares; consultar la documentación para más detalles.

Elemento	Descripción
`MAX_VALUE`	Constante que almacena el valor más grande posible para un `int`: $2^{31}-1$.
`MIN_VALUE`	Constante que almacena el valor más pequeño posible para un `int`: -2^{31}.
`SIZE`	Número de bits necesarios para almacenar un `int` (en cualquier plataforma).
`Integer(int v)`	Constructor partiendo de un valor `int`.
`Integer(String s)`	Constructor partiendo de un `String` que tendrá que ser convertido a `int`.
`intValue()`	Método que retorna el valor `int` que encapsula (con el que fue creado).
`parseInt(String s)`	Método estático que permite convertir una cadena de caracteres a entero `int`. Puede generar la excepción `NumberFormatException` en caso de que la cadena no represente un valor entero válido.
`parseInt(String s, int base)`	Método estático que permite convertir una cadena de caracteres a entero `int`. Es similar al anterior pero recibe un segundo parámetro con la base de numeración en forma de valor numérico (10-decimal, 8-octal, 2-binario, 16-hexadecimal, ...); caso de no especificar la base de numeración (caso anterior) se supone que es el sistema decimal.
`toString()`	Método que retorna el valor que encapsula en forma de cadena. Normalmente no es necesario invocar explícitamente este método, ya que al concatenar cualquier `String` a un objeto `Integer`, éste se convierte automáticamente a `String`.

Cuadro 7.8: Clase Integer: constantes, constructores y métodos

A continuación mostramos un **ejemplo** de uso de los métodos de la clase envoltura `Integer` para almacenar valores `int` en un `ArrayList`, y su posterior recuperación, realizando la suma de todos ellos y mostrando al final el valor calculado.

```
1  import java.util.ArrayList;
2
3  public class EnvolturaDemo1 {
4
5     public static void main (String args[]) {
6        Integer v;
7        int suma, n;
8
9        // Creamos la lista
10       ArrayList lista = new ArrayList();
11       // Añadimos varios valores a la lista (int)
12       // Primero, especificando un valor int a la hora de crear el objeto v
13       v = new Integer(1);    lista.add (v);
14       // Segundo, especificando un valor en forma de cadena
15       // (el objeto v creado antes no se pierde, está almacenado en la lista).
16       v = new Integer("2");  lista.add (v);
17       // Tercero, especificando un valor int pero sin utilizar el objeto v.
18       // No es necesario la declaración y uso de ese objeto si lo hacemos así.
19       lista.add (new Integer(3));
20
21       // Extraemos los valores enteros y calculamos su suma
22       suma = 0;
23       for (int i=0; i<lista.size(); i++) {
24          // Extraemos un valor (objeto Integer)
25          v = (Integer) lista.get(i);  // Casting necesario
26          // Obtenemos el valor int que encapsula el objeto Integer
27          n = v.intValue();
28          // Mostramos el valor
29          System.out.println ("Valor = " + n);
30          // Acumulamos en la suma
31          suma = suma + n;
32       }
33       // Mostramos la suma de los valores de la lista
34       System.out.println ("Suma  = " + suma);
35    }
36 }
```

```
Valor = 1
Valor = 2
Valor = 3
Suma  = 6
```


El ejemplo anterior muestra que para **insertar valores de tipo primitivo** en una colección es necesario realizar dos pasos: (1) crear el objeto de la clase envoltura `Integer`) con el valor a insertar, y (2) insertarlo como cualquier otro objeto mediante el método `add()`. Para crear el objeto `Integer` podemos hacerlo de 2 formas diferentes: desde un valor entero (líneas 13 y 19), o desde un valor especificado en forma de cadena, que debe contener un entero (línea 16). En cuanto al segundo paso, a la hora de insertar los valores en la lista, en el primer caso (línea 13) se crea un objeto `v` de la clase `Integer`, el cual se añade después con `lista.add(v)`. En el segundo caso (línea 16) se vuelve a utilizar ese mismo objeto `v`, y el anterior no se perderá, ya que se había almacenado en la lista. De aquí se puede deducir que no es necesario declarar ni utilizar dicho objeto `v`, puesto que se puede hacer todo en un único paso, tal como se muestra al crear el tercer valor (línea 19).

Del mismo modo, a la hora de **extraer un valor de tipo primitivo** de una colección tenemos que realizar dos pasos: (1) extraer el elemento mediante el método `get()`, haciendo casting a `Integer`, por ser la clase del objeto que se insertó (línea 25), y (2) obtener el valor `int` que encapsula el objeto `Integer`, haciendo uso del método `intValue()` (línea 27).

Se muestran distintas formas de hacer la misma tarea, aunque no hay una mejor que otra, todo dependerá del estilo de programación del programador, o si se prefiere claridad frente a un código más o menos compacto.

Autoboxing / Unboxing

Desde la versión Java 5 podemos hacer uso de una nueva característica llamada *autoboxing* y *unboxing*, que elimina la necesidad de realizar la conversión manual de tipos primitivos a sus respectivas clases envoltura y viceversa, respectivamente. Esto simplifica bastante el código, tal y como se puede ver en el siguiente listado:

```
1  import java.util.ArrayList;  // List
2
3  public class EnvolturaDemo2 {
4
5     public static void main (String args[]) {
6        int suma, n;
7
8        // Creamos la lista
9        ArrayList lista = new ArrayList();
10       // Añadimos varios valores a la lista (int): autoboxing
11       lista.add (1);
12       lista.add (new Integer("2"));
13       lista.add (3);
14
15       // Extraemos los valores enteros y calculamos su suma
16       suma = 0;
17       for (int i=0; i<lista.size(); i++) {
18          // Extraemos un valor (directamente a un valor int): unboxing
19          n = (Integer) lista.get(i);  // Casting necesario
20          // Mostramos el valor
21          System.out.println ("Valor = " + n);
22          // Acumulamos en la suma
23          suma = suma + n;
24       }
25       // Mostramos la suma de los valores de la lista
26       System.out.println ("Suma  = " + suma);
27    }
28 }
```

Como vemos, ya no hemos necesitado utilizar un objeto de la clase `Integer`, ni para insertar los valores en la lista, ni para extraerlo de ella. Sin embargo, el casting de la línea 19 sigue siendo necesario, pero podría eliminarse si hacemos uso de otra propiedad nueva de Java 5 llamada *generics* o clases parametrizadas, la cual se explica a continuación.

7.6. Clases genéricas o parametrizadas (*generics*)

Hasta ahora, las colecciones que hemos visto permiten almacenar objetos de la clase `Object` o subclases, es decir, objetos de cualquier clase. Sin embargo, a la hora de extraer los objetos de la colección estábamos obligados a utilizar *casting* a la clase concreta que se almacenó. Aunque una colección puede contener objetos de diferentes clases, lo más habitual es que todos los objetos almacenados sean del mismo tipo, al igual que ocurre con los *arrays*.

Desde la versión Java 5 podemos hacer uso de una nueva característica para estos casos llamada *generics* que nos ayuda en el uso de colecciones, y es aplicable a todas las colecciones explicadas anteriormente: `Vector`, `ArrayList`, ... Simplemente se trata de especificar en el momento de declarar el objeto colección la clase de los objetos que vamos a almacenar, como un parámetro entre los símbolos '<' y '>', motivo por el que estas clases se dice que son clases parametrizadas o genéricas (*generics*). De esta forma se simplifica el código posterior, ya que no será necesario hacer *casting* en el momento de extraer objetos de la lista.

El siguiente **ejemplo** muestra un pequeño programa que almacena una serie de cadenas de caracteres (`String`) en una colección de tipo `Vector`, primero tal cual lo hemos hecho hasta ahora (izquierda), y luego utilizando *generics* (derecha); puede verse en la línea 6 la declaración del objeto `Vector` especificando que almacenará objetos de tipo cadena (`String`).

En el primer caso vemos que es obligatorio realizar el *casting* a la hora de extraer elementos de la lista (línea 12). Puesto que la lista admite cualquier tipo de objeto, en el caso de realizar *casting* a una clase incorrecta o incompatible se generaría un error en tiempo de ejecución (`ClassCastException`). Por tanto, tendríamos que estar muy seguros de los objetos que insertamos, de lo contrario será nesario averiguar a qué clase pertenece mediante el operador `instanceof` antes de realizar el *casting*. Todo ello hace que sea necesario escribir más código del necesario, y que sea propenso a errores.

```
1  import java.util.Vector;
2
3  public class VectorDemo4 {
4
5      public static void main (String
           args[]) {
6          Vector v;   // Admite Object
7
8          v = new Vector();
9          v.add ("Uno");
10         v.add ("Dos");
11         for (Object o : v) {
12             String s = (String) o;
13             System.out.println (s);
14         }
15     }
16 }
```

```
1  import java.util.Vector;
2
3  public class VectorDemo5 {
4
5      public static void main (String
           args[]) {
6          Vector<String> v;  // Admite
               String
7
8          v = new Vector();
9          //v = new Vector<String>();
10         v.add ("Uno");
11         v.add ("Dos");
12         for (String s : v) {
13             System.out.println (s);
14         }
15     }
16 }
```


Sin embargo, cuando se hace uso de *generics* (a la derecha) no es necesario realizar el *casting* a `String` en el momento de extraer elementos de la colección, pues los únicos objetos que admite la lista serán `String`.

Quizás se vea más claro en el siguiente **ejemplo**, que utiliza un bucle tradicional `for` y los métodos `size()` y `get()` para recorrer un `ArrayList`.

```
1  import java.util.List;
2  import java.util.ArrayList;
3
4  public class ArrayListDemo4 {
5
6      public static void main (String
           args[]) {
7          List lista;  // Admite Object
8
9          lista = new ArrayList();
10         lista.add ("Uno");
11         lista.add ("Dos");
12         for (int i=0; i<lista.size();
               i++) {
13             Object o = lista.get (i);
14             String s = (String) o;
15             System.out.println (s);
16         }
17     }
18 }
```

```
1  import java.util.List;
2  import java.util.ArrayList;
3
4  public class ArrayListDemo5 {
5
6      public static void main (String
           args[]) {
7          List<String> lista;
8
9          lista = new ArrayList();
10         // lista = new ArrayList<
               String>();
11         lista.add ("Uno");
12         lista.add ("Dos");
13         for (int i=0; i<lista.size();
               i++) {
14             String s = lista.get (i);
15             System.out.println (s);
16         }
17     }
18 }
```

Todas las interfaces y clases de la *Collection Framework* son genéricas, es decir, podemos hacer uso de esta característica. La ventaja de esto es que se escribe menos código, y será el propio compilador el que comprobará la clase de los objetos introducidos, lanzando un error en el caso de que no coincidan; asimismo, se comprobará que los elementos extraídos se asignan a objetos de la misma clase, generando también un error en caso contrario. Resaltar que estos errores lanzados al utilizar *generics* son lanzados en tiempo de compilación, y no en tiempo de ejecución como anteriormente, facilitando la depuración del programa final.

En el caso de que tengamos una jerarquía de clases, y queramos almacenar objetos de una serie de clases relacionadas mediante herencia, también podemos utilizar *generics*, especificando como clase la superclase o clase padre de nuestra jerarquía, siendo válido insertar objetos de dicha clase o cualquier subclase de ella. Por ejemplo, en capítulos anteriores se construyó la jerarquía de clases `Figuras`, `Fig2D`, `Cuadrado`, ... Podríamos especificar que la colección es de la clase `Figuras` y podríamos almacenar objetos de cualquier subclase, y podríamos hacer *casting* directamente desde `Object` a `Figuras`.

Un caso diferente sería si intentásemos insertar objetos de clases incompatibles o no relacionadas por herencia. Si por ejemplo, en el programa anterior (utilizando *generics*) quisiéramos insertar un valor entero, convertido a objeto de la clase `Integer` de forma explícita o haciendo uso de *autoboxing* con `lista.add(19)`, el compilador detectaría que estamos intentando insertar un objeto de una clase distinta y no compatible con la especificada (`String`), generando un error de compilación. El siguiente ejemplo ilustra este caso.


```
1  import java.util.ArrayList;
2
3  public class ArrayListDemo4 {
4
5     public static void main (String args[]) {
6        String nombre;
7
8        // Creamos la lista
9        ArrayList<String> lista = new ArrayList();
10
11       // Añadimos varios objetos a la lista (cadenas de texto y un entero)
12       lista.add ("Pepe");    // String
13       lista.add ("Maria");   // String
14       lista.add (19);        // Integer (error de compilación)
15
16       // Extraemos los objetos de la lista y los mostramos
17       for (int i=0; i<lista.size(); i++) {
18          nombre = lista.get(i);  // Casting NO necesario
19          System.out.println (i + ". " + nombre);
20       }
21    }
22 }
```

Vemos que se añaden 3 objetos: 2 `String` y 1 entero (convertido automáticamente a objeto `Integer`). Puesto que hacemos uso de *generics* para indicar que la lista almacenaráa `String`, el programa no compilará, tendremos un error de compilación al intentar insertar el objeto `Integer` (línea 14). Por otro lado, si nuestra intención es realmente almacenar objetos de diferentes clases, entonces no podremos utilizar *generics* y tendremos que hacer *casting* al extraer los objetos. Pero, como los objetos pueden ser de objetos diferentes, antes de hacer *casting* tendremos que comprobar mediante el operador `instanceof` la clase a la que pertenece cada objeto concreto. Si no hiciéramos esto e hiciésemos *casting* a String, el programa compilaría, pero en tiempo de ejecución obtendríamos un error `ClassCastException` en el momento en que intentemos convertir el el tercer objeto que es un `Integer` a `String`, ya que son tipos incompatibles.

En el siguiente **ejemplo**, almacenaremos `String` e `Integer` mezclados, y comprobaremos mediante `instanceof` de qué tipo es cada objeto extraído para hacer *casting*, y además, hacer cosas distintas: mostraremos por pantalla los objetos `String`, y sumaremos los `Integer`.

```
1  import java.util.ArrayList;
2
3  public class ArrayListDemo5 {
4
5     public static void main (String args[]) {
6        String nombre;
7        int    suma, n;
```



```
8
9         // Creamos la lista
10        ArrayList lista = new ArrayList();
11
12        // Añadimos varios objetos a la lista (cadenas de texto y un entero)
13        lista.add ("Pepe");     // String
14        lista.add (19);         // Integer
15        lista.add ("Paco");     // String
16        lista.add (21);         // Integer
17
18         // Extraemos los objetos de la lista y los mostramos
19         suma = 0;
20         for (int i=0; i<lista.size(); i++) {
21            Object o = lista.get(i);
22            // Casting necesario una vez comprobada la clase
23            if (o instanceof String) {
24               // Mostramos los String
25               nombre = (String) o;
26               System.out.println (i + ". " + nombre);
27            } else if (o instanceof Integer) {
28               // Sumamos los Integer
29               n = (Integer) o;   // Unboxing automático
30               suma = suma + n;
31            }
32         }
33         // Mostramos la suma final
34         System.out.println ("Suma = " + suma);
35     }
36 }
```

Como **conclusión**, se nos pueden presentar los siguientes casos:

- Si todos los objetos a almacenar son de la **misma clase** (o clases relacionadas con herencia), entonces utilizaremos *generics*, simplificándose el codigo y ocupándose el compilador de las comprobaciones necesarias.
- Si los objetos a almacenar son de **clases diferentes** (incompatibles entre sí), entonces no podemos utilizar *generics*, y a la hora de extraer los objetos de la lista tendremos que comprobar obligatoriamente la clase de la que es instancia cada objeto mediante el operador `instanceof`; una vez comprobado ya se podrá hacer la conversión mediante *casting*.

GLOSARIO DE ACRÓNIMOS

Siglas	Significado
ANSI	American National Standards Institute
AOT	Ahead-Of-Time (técnica de compilación)
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
CISC	Complex Instruction Set Computing
DBMS	DataBase Management System
DLL	Dynamic Linked Library
ETSI	European Telecommunications Standards Institute
GNU	GNU Not Unix
GPL	GNU General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineer
ISO	International Standards Organization
JAR	Java ARchive
J2SE	Java 2 Platform, Standard Edition (~ Java SE)
J2EE	Java 2 Platform, Enterprise Edition (~ Java EE)
J2ME	Java 2 Platform, Micro Edition (~ Java ME)
JCP	Java Community Process
JDBC	Java DataBase Conectivity
JDK	Java Development Kit (~ Java 2 SDK y J2SDK)
JFC	Java Foundation Classes
JIT	Just-In-Time (técnica de compilación)
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KNI	K Native Interface (~ JNI)
ODBC	Open Database Connectivity
OSI	International Organization of Standarization
PC	Personal Computer

PDA	Personal Digital/Data Assistant
RDBMS	Relational DataBase Management System
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SDK	Standard Development Kit
SQL	Structured Query Language
SRAM	Static Random Access Memory
URL	Uniform Resource Locator
UTF-16	Unicode Transformation Format (16-bit)
WORA	Write Once, Run Anywhere