

Clase 33

Marco para las colecciones de Java

Historia

En la versión original del kit de desarrollo de Java, JDK 1.0, los desarrolladores contaban con muy pocas estructuras de datos. Éstas eran:

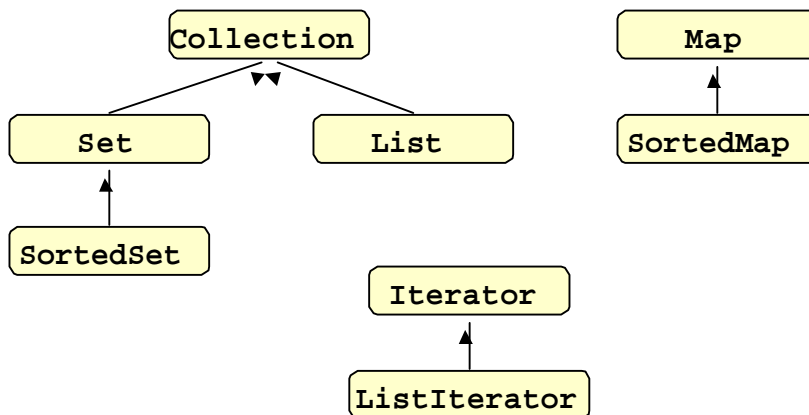
- Vector
- Stack: con Vector ampliado
- Hashtable: muy parecido a nuestra implementación de HashMap
- Dictionary: una clase abstracta que definía la interfaz y cierta funcionalidad para las clases que asignaban claves a valores. Dictionary servía de clase base para Hashtable.
- Enumeration: era una versión simple de nuestro Iterator que permitía iterar en instancias de Hashtable y Vector.

Marco de colecciones de Java

- Los diseñadores de Java se dieron cuenta de que estas estructuras de datos no eran las adecuadas.
- Realizaron el prototipo de un nuevo conjunto de estructuras de datos como un conjunto de herramientas independiente al término de JDK 1.1 y lo convirtieron en una parte formal del JDK en su versión 1.2.
- Este conjunto de clases posterior, mejor y más completo recibe el nombre de marco de colecciones de Java.
- Puede encontrar un buen manual de aprendizaje en <http://java.sun.com/docs/books/tutorial/collections/index.html>.

3

Interfaces de las colecciones, 1



4

Interfaces de las colecciones, 2

- **Collection:** es la interfaz más básica; tiene la funcionalidad de una lista no ordenada, también conocida como *multiconjunto*, un conjunto que no necesita buscar duplicados.
- **Set:** agrega semántica de conjuntos, es decir, no permitirá miembros duplicados.
- **List:** agrega semántica de listas, es decir, un sentido de orden y posición
- **SortedSet:** agrega orden a la semántica de conjuntos; nuestro árbol binario de búsqueda que incluí sólo claves sin valores podría implementarse como un `SortedSet`

5

Interfaces de las colecciones, 3

- **Map:** interfaz básica para estructuras de datos que asignan claves a valores; utiliza una clase interna llamada `Entry`
- **SortedMap:** mapa de claves ordenadas con valores; Nuestro árbol binario de búsqueda es un buen ejemplo
- **Iterator:** igual que nuestro iterador, salvo que se trata de un *fail fast*: arroja una excepción `ConcurrentModificationException` si se utiliza una instancia después de modificar la interfaz `Collection` subyacente.
- **ListIterator:** iterador bidireccional.

6

Implementaciones de las colecciones

- Los diseñadores de Java desarrollaron la arquitectura de las interfaces de cualquier estructura de datos. En una segunda etapa, crearon un número específico de implementaciones de dichas interfaces basándose en versiones más sofisticadas de las estructuras de datos que hemos estudiado:
 - *Array reajutable*: similar a la técnica utilizada para la implementación de nuestras pilas.
 - *Lista enlazada*: utiliza una lista doble, no sencilla.
 - *Tabla hash*: muy parecida a nuestra implementación, salvo en que el número de *slots* crece cuando el factor pasa un valor que puede definirse en el constructor.
 - *Árbol equilibrado*: similar a la implementación de nuestro árbol binario de búsqueda, pero basada en el árbol *Red-Black*, un método más sofisticado que reequilibra el árbol después de algunas operaciones.

7

Implementaciones de las colecciones, 2

		<i>Implementaciones</i>			
		<i>Tabla Hash</i>	<i>Array reajutable</i>	<i>Árbol equilibrado</i>	<i>Lista enlazada</i>
<i>Inter-faces</i>	<i>List</i>		<i>Array-List</i>		<i>Linked-List</i>
	<i>Set</i>	<i>HashSet</i>			
	<i>Ordered Set</i>			<i>TreeSet</i>	
	<i>Map</i>	<i>HashMap</i>			
	<i>Ordered Map</i>			<i>TreeMap</i>	

8

Implementaciones de las colecciones, 3

- Observe los huecos de la tabla. No hay ninguna lista basada en una tabla *hash*. ¿Por qué? ¿Qué ocurre con la implementación de Set basada en una lista enlazada?
- El objetivo de los diseñadores de Java era crear un pequeño conjunto de implementaciones para que los usuarios se iniciasen, no crear un conjunto exhaustivo.
- Esperaban que los desarrolladores ampliaran las clases de las colecciones y que agregasen más clases de flujos.
- La parte más importante del diseño de colecciones era la arquitectura de las interfaces. Por eso reciben el nombre de *marco*.

9

Opciones de implementación

- Tenga cuidado con las opciones de implementación. No todas las implementaciones son eficaces en todas sus operaciones.
- Por ejemplo, un `ArrayList` le permitirá eliminar el primer elemento llamando a `remove(0)`. Ya sabemos que esto resulta especialmente ineficaz.
- También puede iterar en `LinkedList`, `l`, con el siguiente código, pero una vez más, no resulta demasiado eficaz. ¿Por qué?

```
for ( int i = 0; i < l.size(); i++ ) {  
    Object o = l.get( i );  
    // haga lo que necesite con o  
}
```

10

Interfaz Collection

```
public interface Collection {  
    // Operaciones de consulta  
    int size();  
    boolean isEmpty();  
    boolean contains(Object o); boolean  
containsAll(Collection c); Iterator  
iterator();  
Object[] toArray();  
Object[] toArray(Object a[]);  
boolean equals(Object o);  
int hashCode();  
}
```

11

Interfaz Collection, 2

```
// Operaciones de modificación  
boolean add(Object o); boolean  
remove(Object o); boolean  
addAll(Collection c);  
boolean removeAll(Collection c);  
boolean retainAll(Collection c);  
void clear();  
}
```

12

Operaciones globales

- **Observe las operaciones *globales*. Todas las interfaces que provienen de `Collection` las aceptan:**
 - `boolean containsAll(Collection c)`
 - `boolean addAll(Collection c)`
 - `boolean removeAll(Collection c)`
 - `boolean retainAll(Collection c)`
- **`containsAll()` devuelve *true* si el objetivo del método contiene todos los elementos en la colección del argumento. Los otros tres métodos devuelven *true* si la operación cambia la colección del objetivo.**

13

Constructores globales

Todas las implementaciones `Collection (Map)` admiten, al menos, dos constructores.

- uno predeterminado que crea un `Collection (Map)` vacío del tipo adecuado y
- uno que toma un argumento de `Collection (Map)` que crea un `Collection (Map)` del tipo adecuado que contiene referencias a todos los objetos de `Collection (Map)` proporcionados como argumento.

14

Operaciones de *arrays*

- Todas las implementaciones `Collection` también poseen un método con la firma:

```
Object [] toArray()
```

que devuelve el contenido de la colección en un *array* del tamaño adecuado.

- Una variante:

```
Object[] toArray(Object a[]);
```

devuelve todos los elementos de la colección que coinciden con el tipo de *array* `a[]` en `a[]` i un *array* del tamaño adecuado y del mismo tipo.

15

Ejemplos de *arrays*

- ```
Collection c = new HashSet();
```

```
// poner las cosas en c
```

```
String[] s =
```

```
 (String[]) c.toArray(new String[0]);
```

devuelve todos los elementos `String` de `c` en un *array* de `String`.
- ```
Collection c = new HashSet();
```



```
// si c contiene sólo strings
```

```
String[] s = (String[]) c.toArray( );
```

16

Interfaz List

```
public interface List extends Collection {
    // agrega lo siguiente a la colección
    boolean addAll(int index, Collection c);
    Object get(int index);
    Object set(int index, Object element);
    void add(int index, Object element);
    Object remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator listIterator();
    ListIterator listIterator(int index);
    List subList(int fromIndex, int toIndex);
}
```

17

Vistas

- Muchas de las interfaces también aceptan operaciones de *vistas*. En una operación de este tipo, la colección devuelve otra colección que proporciona una vista especializada y normalmente parcial de su contenido. Un buen ejemplo sería

```
List subList( int from, int to )
```
- Este método devuelve una segunda lista que comienza con el elemento `from` de la lista padre y termina justo antes del `enésimo` elemento. Pero la sublista devuelta no es una copia: sigue siendo parte de la lista original.
- Como ejemplo, una forma elegante de eliminar los elementos del 4 al 10 de una lista es la siguiente:

```
myList.subList( 3, 10 ).clear();
```

18

Interfaz Set

```
public interface Set extends Collection {  
    // tiene los mismos métodos que la colección  
    // pero con semántica más estricta  
}
```

19

Interfaz SortedSet

```
public interface SortedSet extends Set {  
    // agrega lo siguiente a Set  
    Comparator comparator();  
    SortedSet subSet(Object fromElement,  
                     Object toElement);  
    SortedSet headSet(Object toElement);  
    SortedSet tailSet(Object fromElement);  
    Object first();  
    Object last();  
}
```

20

Interfaz Iterator

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

21

Interfaz ListIterator

```
public interface ListIterator extends Iterator {  
    // agrega lo siguiente a Iterator  
    boolean hasPrevious();  
    Object previous();  
    int nextIndex();  
    int previousIndex();  
    void set(Object o);  
    void add(Object o);  
}
```

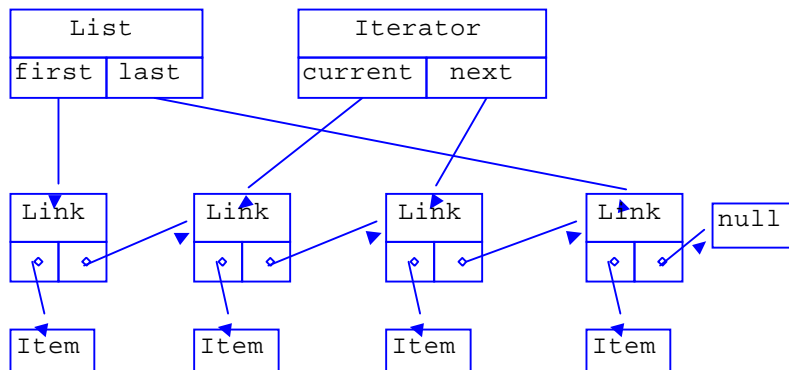
22

Modificación concurrente

- Los iteradores deben conservar referencias a la estructura de datos subyacente para mantener su sentido de posición actual.
- Experimentalmente:
 - cree una `LinkedList`, agregue elementos, obtenga un iterador, aváncelo
 - ahora, con el método de la lista (no con el iterador) elimine el siguiente elemento del iterador
 - ahora llame a `next()` en el iterador
 - ¿Qué va mal?
- Siempre que cree un iterador, modifique la colección subyacente y después lo utilice, obtendrá un excepción `ConcurrentModificationException`.
- ¿Cómo cree que podría implementarse la comprobación?

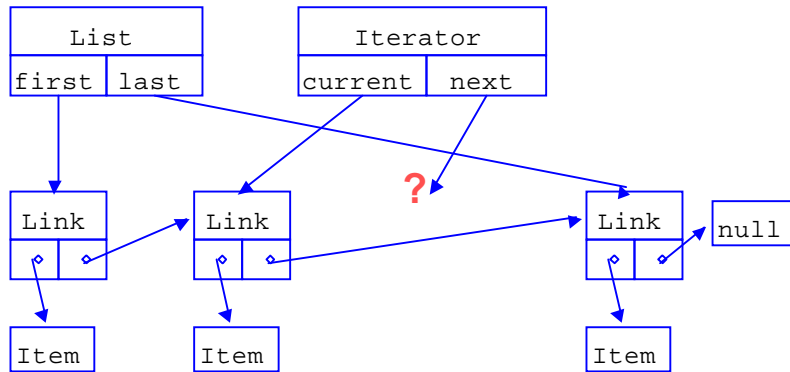
23

Modificación concurrente, antes



24

Modificación concurrente, después



25

Interfaz Map

```
public interface Map {  
    // Operaciones de consulta  
    int size();  
    boolean isEmpty();  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    Object get(Object key);  
}
```

26

Interfaz Map, 2

```
// Operaciones de modificación
Object put(Object key, Object value);
Object remove(Object key);
void putAll(Map t);
void clear();
// Vistas
public Set keySet();
public Collection values();
public Set entrySet();
boolean equals(Object o);
int hashCode();
```

27

Interfaz anidada Map.Entry

```
public interface Entry {
    Object getKey();
    Object getValue();
    Object setValue(Object value);
    boolean equals(Object o);
    int hashCode();
}
}
```

28

Vistas Map

- `Map` también ofrece vistas. De hecho, son cruciales, ya que un `Map` no proporciona un método `iterator()`.
- Hay dos idiomas independientes para iterar en un `Map`, según se desee iterar en las claves o en los valores:

```
Map m = ...;
// itera en las claves
for (Iterator i=m.keySet().iterator(); i.hasNext();)
{...}
// itera en los valores
for (Iterator i=m.values().iterator(); i.hasNext();)
{...}
```

- En el segundo ejemplo, `values()` devuelve un `Collection`, no un `Set`, ya que el mismo valor puede producirse varias veces en un mapa, mientras que la clave debe ser única.

29

Interfaz SortedMap

```
public interface SortedMap extends Map {
    // agrega lo siguiente a Map
    Comparator comparator();
    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);
    Object firstKey();
    Object lastKey();
}
```

30

Operaciones opcionales

- No todas las implementaciones implementan todas las operaciones en una interfaz. De la documentación se puede deducir claramente si una implementación específica implementa un método dado. Si no lo hace, arrojará una excepción `UnsupportedOperationException` (sin comprobar).
- En un principio, esto puede parecer hasta perverso, pero el objetivo es aportar más flexibilidad. Imagine un programa en el que desea mantener un índice maestro para que un usuario normal pueda buscar entradas, pero únicamente los usuarios con privilegios puedan agregar entradas al índice. El índice se podría implementar como un `SortedMap`, pero, ¿cómo podría evitar que un usuario arbitrario pudiese agregar y eliminar entradas? La respuesta está en la subclase `SortedMap` y en ignorar cada método que se quería prohibir con un simple método que arroje una excepción `UnsupportedOperationException`.

31

Algoritmos

- La clase `Collections` contiene implementaciones de varios algoritmos útiles implementados como métodos `static` que toman una lista `o`, en ocasiones, una `Collection` más general como objetivo del algoritmo.
- Los algoritmos incluyen `sort()`, `reverse()`, `min()`, `max()`, `fill(Object o)`, etc.
`static void sort(List l, Comparator c)`
`static void fill(List l, Object o)`
- Existe una clase muy parecida, `Arrays`, que implementa la mayoría de los mismos algoritmos para *arrays*.

32

Colecciones de sólo lectura

Las colecciones también pueden generar vistas no modificables (de sólo lectura) de todos los tipos de colecciones que arrojarán una excepción `UnsupportedOperationException` en cualquier operación de escritura:

```
public static Collection  
    unmodifiableCollection(Collection c);  
public static Set unmodifiableSet(Set s);  
...
```

