9.F. Iteradores.

Sitio: VIRGEN DE LA PAZ
Curso: Programación
Libro: 9.F. Iteradores.

Imprimido por: Cristian Esteban Gómez
Día: miércoles, 31 de mayo de 2023, 11:21

Tabla de contenidos

1. Iteradores (I).

¿Qué son los iteradores realmente? Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma sencilla, de forma secuencial, y de forma segura. Los mapas, como no derivan de la interfaz Collection realmente, no tienen iteradores, pero como veremos, existe un truco interesante.

Los iteradores permiten recorrer las colecciones de dos formas: **bucles for-each** (existentes en Java a partir de la versión 1.5) **y a través de un bucle normal creando un iterador**. Como los bucles for-each ya los hemos visto antes (y ha quedado patente su simplicidad), nos vamos a centrar en el otro método, especialmente útil en versiones antiguas de Java. Ahora la pregunta es, ¿cómo se crea un iterador? Pues invocando el método "`iterator()`" de cualquier colección. Veamos un ejemplo (en el ejemplo `t` es una colección cualquiera):

```
Iterator<Integer> it=t.iterator();
```

Fijate que se ha especificado un parámetro para el tipo de dato genérico en el iterador (poniendo "`<Integer>`" después de `Iterator`). Esto es porque los iteradores son también clases genéricas, y es necesario especificar el tipo base que contendrá el iterador. Sino se especifica el tipo base del iterador, igualmente nos permitiría recorrer la colección, pero retornará objetos tipo `Object` (clase de la que derivan todas las clases), con lo que nos veremos obligados a forzar la conversión de tipo.

Para recorrer y gestionar la colección, el iterador ofrece tres métodos básicos:

- `boolean hasNext()`. Retornará true si le quedan más elementos a la colección por visitar. False en caso contrario.

- `E next()`. Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (`NoSuchElementException` para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.

- `remove()`. Elimina de la colección el último elemento retornado en la última invocación de next (no es necesario pasarselo por parámetro). Cuidado, si next no ha sido invocado todavía, saltará una incomoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy sencilla, un simple bucle mientras (`while`) con la condición `hasNext()` nos permite hacerlo:

```
while (it.hasNext()) // Mientras que haya un siguiente elemento, seguiremos en el bucle.
{
        Integer t=it.next(); // Escogemos el siguiente elemento.
        if (t%2==0) it.remove();  //Si es necesario, podemos eliminar el elemento extraído de la lista.
}
```

¿Qué elementos contendría la lista después de ejecutar el bucle? Efectivamente, solo los números impares.

Reflexiona

Las listas permiten acceso posicional a través de los métodos `get` y `set`, y acceso secuencial a través de iteradores, ¿cuál es para tí la forma más cómoda de recorrer todos los elementos? ¿Un acceso posicional a través un bucle "`for (i=0;i<lista.size();i++)`" o un acceso secuencial usando un bucle "`while (iterador.hasNext())`"?

2. Iteradores (II).

¿Qué inconvenientes tiene usar los iteradores sin especificar el tipo de objeto? En el siguiente ejemplo, se genera una lista con los números del 0 al 10. De la lista, se eliminan aquellos que son pares y solo se dejan los impares. En el ejemplo de la izquierda se especifica el tipo de objeto del iterador, en el ejemplo de la derecha no, observa el uso de la conversión de tipos en la línea 6.

Comparación de usos de los iteradores, con o sin conversión de tipos.	
Ejemplo indicando el tipo de objeto de iterador	Ejemplo no indicando el tipo de objeto del iterador
`ArrayList <Integer> lista=new ArrayList<Integer>();`	`ArrayList <Integer> lista=new ArrayList<Integer>();`
`for (int i=0;i<10;i++) lista.add(i);`	`for (int i=0;i<10;i++) lista.add(i);`
`Iterator<Integer> it=lista.iterator();`	`Iterator it=lista.iterator();`
`while (it.hasNext()) {`	`while (it.hasNext()) {`
`Integer t=it.next();`	`Integer t=(Integer)it.next();`
`if (t%2==0) it.remove();`	`if (t%2==0) it.remove();`
`}`	`}`

Un iterador es seguro porque está pensado para no sobrepasar los límites de la colección, ocultando operaciones más complicadas que pueden repercutir en errores de software. Pero realmente se convierte en inseguro cuando es necesario hacer la operación de conversión de tipos. Si la colección no contiene los objetos esperados, al intentar hacer la conversión, saltará una incómoda excepción. Usar genéricos aporta grandes ventajas, pero usándolos adecuadamente.

Para recorrer los mapas con iteradores, hay que hacer un pequeño truco. Usamos el método entrySet que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien, el método keySet para generar un conjunto con las llaves existentes en el mapa. Veamos cómo sería para el segundo caso, el más sencillo:

```
HashMap<Integer,Integer> mapa=new HashMap<Integer,Integer>test();

for (int i=0;i<10;i++) mapa.put(i, i); // Insertamos datos de prueba en el mapa.

for (Integer llave:mapa.keySet()) // Recorremos el conjunto generado por keySet, contendrá las llaves.

{

        Integer valor=mapa.get(llave); //Para cada llave, accedemos a su valor si es necesario.

}
```

Lo único que tienes que tener en cuenta es que el conjunto generado por **keySet** no tendrá obviamente el método `add` para añadir elementos al mismo, dado que eso tendrás que hacerlo a través del mapa.

Recomendación

Si usas iteradores, y piensas eliminar elementos de la colección (e incluso de un mapa), debes usar el método `remove` del iterador y no el de la colección. Si eliminas los elementos utilizando el método `remove` de la colección, mientras estás dentro de un bucle de iteración, o dentro de un bucle for-each, los fallos que pueden producirse en tu programa son impredecibles. ¿Logras adivinar porqué se pueden producir dichos problemas?

Los problemas son debidos a que el `método` remove del iterador elimina el elemento de dos sitios: de la colección y del iterador en sí (que mantiene interiormente información del orden de los elementos). Si usas el método `remove` de la colección, la información solo se elimina de un lugar, de la colección.

Autoevaluación

¿Cuándo debemos invocar el método `remove()` de los iteradores?

- ○ En cualquier momento.
- ○ Después de invocar el método next().
- ○ Después de invocar el método hasNext().
- ○ No es conveniente usar este método para eliminar elementos, es mejor usar el de la colección.