Tema 17: java.util

Antonio J. Sierra

Índice

Introducción

- Los tipos genéricos añaden estabilidad al código proporcionando detección de fallos en compilación.
- Se introduce en la Java SE 5.0.
- Permite a un método operar con objetos de varios tipos mientras se proporciona seguridad en los tipos en tiempo de compilación.
- Los tipos genéricos se usa en *Java Collections Framework* para evitar el uso cast.

Tipos genéricos

Ejemplo

```
public class Box {

    private Object object;

    public void add(Object object) {
      this.object = object;
    }

    public Object get() {
         return object;
    }
}
```

```
public class BoxDemo1 {

 public static void main(String[] args){

 // SOLO objetos Integer en Box!
 Box integerBox = new Box();

 integerBox.add(new Integer(10));
 Integer unInteger = (Integer)unBox.get();
        System.out.println(unInteger);
 }
}
```

```
//Otro ejemplo ...
public class BoxDemo2 {

 public static void main(String[] args) {

  // SOLO objetos Integer en Box!
  Box integerBox = new Box();

  // Si es parte de una aplicación
  // modificada por un programador.
  integerBox.add("10"); // es un String

  // ... Y este es otro, escrito quizas
  // por otro programador.
  Integer unInteger =
    (Integer)integerBox.get();
        System.out.println(unInteger);
    }
}
/*
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to
        java.lang.Integer at BoxDemo2.main(BoxDemo2.java:6)

If the Box class had been designed with generics in mind, this mistake would have been caught by
        the compiler, instead of crashing the application at runtime.
*/
```

Ejemplo

- Para crear un tipo genérico se debe cambiar "`class Box`" por "`class Box<T>`".
- De esta forma se introduce un tipo de variable, llamado `T`, que puede usarse en dentro de la clase (Puede aplicarse a interfaces).
- `T` es una clase especial de variable, cuyo "valor" se le proporciona.
  - Puede ser el tipo de cualquier clase, cualquier interfaz o otro tipo de variable.
  - No puede ser un tipo de dato primitivo.
- En este contexto, se puede decir que `T` es un tipo de parámetro forma de la clase `Box`.
- Reemplaza todas las ocurrencias de `Object` con `T`.

```
/**
 * Generic version of the Box class.
 */
public class Box<T> {

    private T t; // T stands for "Type"

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

Referencias e instanciación

- Para referenciar esta clase genérica desde el propio código, se reemplaza `T` con algún valor concreto, como `Integer`:

```
Box<Integer> integerBox;
```

- Se puede pensar en una invocación de un tipo genérico similar a una invocación de método ordinaria, donde **se proporciona el tipo del argumento** (`Integer` en este caso) a la propia clase `Box`.
- Para instanciar esta clase, hay que usar la nueva palabra, colocando la `<Integer>` palabra entre el nombre de la clase y el paréntesis.

```
integerBox = new Box<Integer>();
```

- O en una sola sentencia:

```
Box<Integer> integerBox = new Box<Integer>();
```

```
public class BoxDemo3 {

 public static void main(String[] args) {
  Box<Integer> integerBox = new Box<Integer>();

  integerBox.add(new Integer(10));
  Integer someInteger = integerBox.get(); // sin
    cast!
       System.out.println(someInteger);
 }
}
```

- Si se intenta añadir un tipo incompatible a `Box`, como por ejemplo un `String`, la compilación falla, alertando de lo que previamente sucedía en tiempo de ejecución:

```
BoxDemo3.java:5: add(java.lang.Integer)
   in Box<java.lang.Integer> cannot be
   applied to (java.lang.String)
   integerBox.add("10"); ^ 1 error
```

Convenios de nombrado

- Los tipos genéricos pueden tener múltiples parámetros, y deberían ser únicos en la declaración de la clase o interfaz.
  - Una declaración de `Box<T,T>`, debería generar un error en la segunda ocurrencia de T, pero debería estar permitida `Box<T,U>`.
- Por convenio, los nombres de los parámetros son letras mayúsculas.
- Los nombres de los tipos de parámetros usados habitualmente son:
  - E - Element (usado en *Java Collections Framework*)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. – tipos 2º, 3º, 4º

Aplicación a métodos

- Se pueden declarar métodos y constructores que contengan *tipos genéricos*.
- Es similar a declarar un tipo genérico con alcance al método o constructor.
- La salida del programa es:

```
T: java.lang.Integer
U: java.lang.String
```

```
/**
 * This version introduces a generic method.
 */
public class Box<T> {

 private T t;

 public void add(T t) {
  this.t = t;
 }
 public T get() {
  return t;
}

 public <U> void inspect(U u){
  System.out.println("T: " + t.getClass().getName());
  System.out.println("U: " + u.getClass().getName());
 }

 public static void main(String[] args) {
  Box<Integer> integerBox = new Box<Integer>();
  integerBox.add(new Integer(10));
  integerBox.inspect("some text");
 }
}
```

Límites a los parámetros de tipo

```
/**
 * This version introduces a bounded type parameter.
 */
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text"); // error: es String!
    }
}
```

- Para especificar una interfaz adicional que podría implementarse, usar el carácter &, por ejemplo:

```
<U extends Number & MyInterface>
```

Bucles for

- Considerando el problema de escribir una rutina que imprima todos los elementos de una colección.
- Con las versiones antiguas del lenguaje (anteriores a la 5.0) se puede realizar de la siguiente forma:

```
void printCollection(Collection c)
   {
 Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++)
   {

   System.out.println(i.next());
    }
}
```

- Un mejor intento de escribirlo es usando un tipo genérico y una nueva sintaxis:

```
void printCollection(Collection<Object>
    c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

- El viejo código podría ser llamado con cualquier clase de colección como un parámetro, el nuevo código solo vale para `Collection<Object>`.

Wildcard type

- El supertipo de todas las clases de objetos se escribe con <?>.
- Se puede escribir

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

- Esto es una colección que puede ser válida con cualquier tipos de elementos (desconocidos).

Wildcard

- Para especificar una “jaula” capaz de contener algún animal:

```
Jaula<? extends Animal> unaJaula = ...;
```

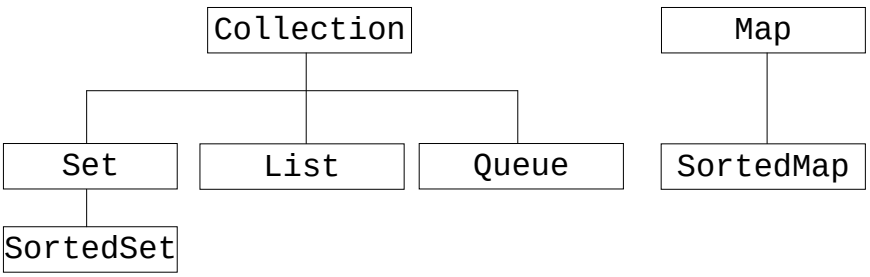
- Es posible especificar un umbral inferior usando la palabra `super` en lugar de `extends`.
  - El código `<? super Animal>`, además, debería se leído como “un tipo desconocido que es un supertipo de `Animal`, posiblemente el mismo `Animal`.
- Se puede especificar un tipo desconocido con sin límite, que es simplemente `<?>`. Un comodín es esencialmente igual que
  - `<? extends Object>`.

Colecciones

Introducción

- Una colección es un objeto que agrupa múltiples elementos en una sola unidad.
- Las colecciones se usan para almacenar, recuperar, manipular y comunicar agregación de datos.
- Normalmente representan ítem de datos que forma un grupo natural, tal y como una baraja de cartas, carpeta de correos o directorio telefónico.
- Las primeras implementaciones del lenguaje de programación Java (antes de la versión 1.2) incluyeron `Vector`, `Hashtable`, y `array`.
- Pertenen al paquete: `java.util`.

Clasificación de las Interfaces



La interfaz `Collection<E>`

- Raíz de la jerarquía de colecciones.
- Una colección representa un grupo de objetos conocido como sus elementos.
- La interfaz `Collection<E>` es el mínimo común denominador de todas las colecciones implementadas y se usa para pasar colecciones y manipularlas cuando se desee tener una mayor generalidad.
  - Unos tipos de colecciones permiten duplicar elementos y otras no.
  - Unas colecciones mantienen un orden y otras no.

Métodos de `Collection<E>`

`boolean add(E e)` Asegura que esta colección contiene el elemento especificado (opcional).

`boolean addAll(Collection<? extends E> c)` Anade todos los elementos especificadas en la colección a la colección especificada (operación opcional).

`void clear()` Elimina todos los elementos de la colección (operación opcional)

`boolean contains(Object o)` Devuelve cierto si la colección conteien el elemento especificado.

`boolean containsAll(Collection<?> c)` Devuelve cierto si la colección contiene todos los elementos especificados en la colección especificada.

`boolean equals(Object o)` Compara el objeto especificado con esta colección.

`int hashCode()` Devuelve el código hash de esta colección.

`boolean isEmpty()` Devuelve cierto si esta colección no contiene elementos

`Iterator<E> iterator()` Devuelve un iterador sobre los elementos de la colección.

`boolean remove(Object o)` Elimina una sola instancia de la colección (opcional).

`boolean removeAll(Collection<?> c)` Elimina todos los elementos de la colección que están contenidos en la colección especificada (opcional).

`boolean retainAll(Collection<?> c)` Mantiene sólo los elementos que están contenidos en la colección (opcional).

`int size()` Devuelve el número de elementos de esta colección.

`Object[] toArray()` Devuelve un array que contiene todos los elementos de la colección.

`<T> T[] toArray(T[] a)`

Devuelve un array que contiene todos los elementos de esta colección; el tipo del array devuelve se especifica en tiempo de ejecución.

`Set`, `List` y `Queue`

- `Set<E>`:
  - es una colección que no puede contener elementos duplicados.
- `SortedSet<E>`:
  - Un conjunto que mantiene sus elementos ordenados de forma ascendente.
  - Se proporcionan operaciones para mantener la ordenación.
  - La ordenación realizada es la natural para estos conjuntos, tales como lista de palabra.
- `List<E>`:
  - es una colección ordenada (en secuencia). Pueden contener elementos duplicados.
  - El usuario de `List` generalmente tiene un control preciso sobre donde está insertado cada elemento en la lista, y puede acceder a los elementos mediante un índice entero que indica la posición.
  - La versión anterior es `Vector`.
- `Queue<E>`:
  - Es una colección usada para determinar la forma en que varios elementos se procesen.
  - `Queue` proporciona operaciones adicionales de inserción, extracción e inspección.
  - Algunas ordenan sus elementos como FIFO (first-in, first-out). Otras son con prioridad.

Implementaciones de `Collection`

- Implementaciones de **`Set<E>`**:
  - `AbstractSet`
  - `ConcurrentSkipListSet`
  - `CopyOnWriteArraySet`
  - `EnumSet`
  - `HashSet`
  - `JobStateReasons`
  - `LinkedHashSet`
  - `TreeSet`
- Implementaciones de **`List<E>`**:
  - `AbstractList`
  - `AbstractSequentialList`
  - `ArrayList`
  - `AttributeList`
  - `CopyOnWriteArrayList`
  - `LinkedList`
  - `RoleList`
  - `RoleUnresolvedList`
  - `Stack`
  - `Vector`
- Implementaciones de **`SortedSet<E>`**:
  - `ConcurrentSkipListSet`
  - `TreeSet`
- Implementaciones de **`Queue<E>`**:
  - `AbstractQueue`
  - `ArrayBlockingQueue`
  - `ArrayDeque`
  - `ConcurrentLinkedQueue`
  - `DelayQueue`
  - `LinkedBlockingDeque`
  - `LinkedBlockingQueue`
  - `LinkedList`
  - `PriorityBlockingQueue`
  - `PriorityQueue`
  - `SynchronousQueue`

Mapas: `Map<K,V>`

- `Mapas`
  - Son objetos que mapean claves a un valor.
  - Un Mapa no puede contener elementos duplicados.
  - Cada clave puede mapear al menos un valor.
  - Se especifica en la interfaz `Map<K,V>`.
  - Las versiones anteriores utilizaban `Hashtable`.
- `Map<K,V>` toma el lugar de la clase **`Dictionary`**, que era una clase completamente abstracta en lugar de una interfaz.
- La interfaz proporciona tres vistas de colecciones,
  - Conjunto de claves,
  - Colección de valores, o
  - Conjunto de mapeos clave-valor.
- El orden de un mapa está definido como el orden en el cual el iterador en la ´vista de colección de mapas devuelve sus elementos.
  - **`TreeMap`**, garantiza el orden,
  - **`HashMap`** no.
- Si el valor de una clave del mapa cambia de forma que afecta a la comparación con **`equals`** entonces la propiedad de un mapa no está especificada.
  - Un caso especial de esta prohibición es que no se permita a un mapa que se contenga a sí mismo.

Método `equals`

- `SortedMap`: es un Mapa que mantiene un orden ascendente en las claves.
  - Es análogo a `SortedSet`.
  - Los Mapas ordenados se usan de una forma natural en colecciones ordenadas de pares claves-valor, tales como diccionarios y directorios de teléfono.
- Muchos métodos en las interfaces Collections Framework están definidas en términos del método `equals`.
- Por ejemplo, la especifiación del método **`containsKey(Object key)`** dice:
  - "returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k))."
- Esta especificación no debería ser construida para invocar **`Map.containsKey`** con una clave no nula provocará que invoque a **`key.equals(k)`** con la clave k.
- Las implementaciones pueden evitar las invocaciones a **`equals`**, por ejemplo, comparando el código hash de las dos claves.
  - (La especificación de **`Object.hashCode()`** garantiza que dos objetos con distinto código hash no puedan ser iguales).

Métodos de `Map<K,V>`

`void clear()` Elimina todas los mapeo de este mapa (opcional).

`boolean containsKey(Object key)` Devuelve cierto si este mapa contiene una mapeo para la clave especificada.

`boolean containsValue(Object value)` Devuelve cierto si este mapa mapea uno o más claves del valor especificado.

`Set<Map.Entry<K,V>> entrySet()` Devuelve una vista en un `Set` del mapeo contenido en este mapa.

`boolean equals(Object o)` Compara el objeto especificado con este mapa.

`V get(Object key)` Devuelve el valor para el cual la clave especificada está mapeada, o `null` si el mapa no contiene ningún mapeo para la clave.

`int hashCode()` Devuelve el valor del código hash de este mapa.

`boolean isEmpty()` Devuelve cierto si este mapa no contiene mapeados clave-valor.

`Set<K> keySet()` Devuelve una vista `Set` de las claves contenidas en este mapa.

`V put(K key, V value)` Asocia el valor especificado con el la clave en este mapa (opcional).

`void putAll(Map<? extends K,? extends V> m)` Copia todos los mapeos dese el mapa especificado (opcional).

`V remove(Object key)`

Elimina el mapeo de una clave del mapa si existe (opcional).

`int size()` Devuelve el número de mapeos clave-valor en este mapa.

`Collection<V> values()` Devuelve una vista en `Collection` de los valores contenidos en este mapa.

Implementaciones de `Map<K,V>`

- Implementaciones de `Map<K,V>`:
  - `AbstractMap`
  - `Attributes`
  - `AuthProvider`
  - `ConcurrentHashMap`
  - `ConcurrentSkipListMap`
  - `EnumMap`
  - `HashMap`
  - `Hashtable`
  - `IdentityHashMap`
  - `LinkedHashMap`
  - `PrinterStateReasons`
  - `Properties`
  - `Provider`
  - `RenderingHints`
  - `SimpleBindings`
  - `TabularDataSupport`
  - `TreeMap`
  - `UIDefaults`
  - `WeakHashMap`
- Implementaciones de `SortedMap<K,V>`:
  - `ConcurrentSkipListMap`
  - `TreeMap`

Recorriendo las colecciones

- Hay dos tipos de colecciones para recorrer:
  - Con la construcción `for-each`.
  - Usando iteradores.

Código para el recorrido

- Construcción `for-each`:
- Permite de forma concisa atravesar una colección o array usando un bucle `for`.
- El siguiente código usa la construcción `for-each` para imprimir cada elemento de una colección en una línea separada.

```
for (Object o : collection)
    System.out.println(o);
```

- Iteradores:
- Un iterador es un objeto que permite atravesar una colección y borrar elementos de la colección de la colección de forma selectiva.
- Se puede obtener un `Iterator` de una colección invocando al método `iterator`.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

- El método`hasNext` devuelve `true` si la iteración tiene más elementos, y
- El método `next` devuelve el siguiente elemento de la iteración.
- El método `remove` elimina el último elemnto que es devuelto mediante `next`.

Comparación de `for-each` e `Iterator`

- Se debe usar un `Iterator` en lugar de `for-each` cuando se necesite:
  - Borrar el actual elemento. Los bloques `for-each` ocultan el iterador, y por tanto no se pueden eliminar.
  - Iteración sobre múltiples colecciones en paralelo.
- El siguiente método muestra como usar un iterador para filtrar una colección arbitraria, es decir que atraviesa la colección borrando elementos específicos:

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

- Este trozo de código es polimórfico, que significa que funciona con cualquier colección sin importar su implementación.

Referencias

- Colecciones
  - http://java.sun.com/docs/books/tutorial/collections/index.html
- Tipos genéricos
  - http://java.sun.com/docs/books/tutorial/java/generics/index.html