The Java™ Tutorials

Trail: Collections
Lesson: Interoperability

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Compatibility

The Java Collections Framework was designed to ensure complete interoperability between the core collection interfaces and the types that were used to represent collections in the early versions of the Java platform: `Vector`, `Hashtable`, array, and `Enumeration`. In this section, you'll learn how to transform old collections to the Java Collections Framework collections and vice versa.

Upward Compatibility

Suppose that you're using an API that returns legacy collections in tandem with another API that requires objects implementing the collection interfaces. To make the two APIs interoperate smoothly, you'll have to transform the legacy collections into modern collections. Luckily, the Java Collections Framework makes this easy.

Suppose the old API returns an array of objects and the new API requires a `Collection`. The Collections Framework has a convenience implementation that allows an array of objects to be viewed as a `List`. You use `Arrays.asList` to pass an array to any method requiring a `Collection` or a `List`.

```
Foo[] result = oldMethod(arg);
newMethod(Arrays.asList(result));
```

If the old API returns a `Vector` or a `Hashtable`, you have no work to do at all because `Vector` was retrofitted to implement the `List` interface, and `Hashtable` was retrofitted to implement `Map`. Therefore, a `Vector` may be passed directly to any method calling for a `Collection` or a `List`.

```
Vector result = oldMethod(arg);
newMethod(result);
```

Similarly, a `Hashtable` may be passed directly to any method calling for a `Map`.

```
Hashtable result = oldMethod(arg);
newMethod(result);
```

Less frequently, an API may return an `Enumeration` that represents a collection of objects. The `Collections.list` method translates an `Enumeration` into a `Collection`.

```
Enumeration e = oldMethod(arg);
newMethod(Collections.list(e));
```

Backward Compatibility

Suppose you're using an API that returns modern collections in tandem with another API that requires you to pass in legacy collections. To make the two APIs interoperate smoothly, you have to transform modern collections into old collections. Again, the Java Collections Framework makes this easy.

Suppose the new API returns a `Collection`, and the old API requires an array of `Object`. As you're probably aware, the `Collection` interface contains a `toArray` method designed expressly for this situation.

```
Collection c = newMethod();
oldMethod(c.toArray());
```

What if the old API requires an array of `String` (or another type) instead of an array of `Object`? You just use the other form of `toArray` — the one that takes an array on input.

```
Collection c = newMethod();
oldMethod((String[]) c.toArray(new String[0]));
```

If the old API requires a `Vector`, the standard collection constructor comes in handy.

```
Collection c = newMethod();
oldMethod(new Vector(c));
```

The case where the old API requires a `Hashtable` is handled analogously.

```
Map m = newMethod();
oldMethod(new Hashtable(m));
```

Finally, what do you do if the old API requires an `Enumeration`? This case isn't common, but it does happen from time to time, and the `Collections.enumeration` method was provided to handle it. This is a static factory method that takes a `Collection` and returns an `Enumeration` over the elements of the `Collection`.

```
Collection c = newMethod();
oldMethod(Collections.enumeration(c));
```

Previous page: Interoperability
Next page: API Design