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.

API Design

In this short but important section, you'll learn a few simple guidelines that will allow your API to interoperate seamlessly with all other APIs that follow these guidelines. In essence, these rules define what it takes to be a good "citizen" in the world of collections.

Parameters

If your API contains a method that requires a collection on input, it is of paramount importance that you declare the relevant parameter type to be one of the collection interface types. **Never** use an implementation type because this defeats the purpose of an interface-based Collections Framework, which is to allow collections to be manipulated without regard to implementation details.

Further, you should always use the least-specific type that makes sense. For example, don't require a `List` or a `Set` if a `Collection` would do. It's not that you should never require a `List` or a `Set` on input; it is correct to do so if a method depends on a property of one of these interfaces. For example, many of the algorithms provided by the Java platform require a `List` on input because they depend on the fact that lists are ordered. As a general rule, however, the best types to use on input are the most general: `Collection` and `Map`.

Caution: Never define your own ad hoc `collection` class and require objects of this class on input. By doing this, you'd lose all the benefits provided by the Java Collections Framework.

Return Values

You can afford to be much more flexible with return values than with input parameters. It's fine to return an object of any type that implements or extends one of the collection interfaces. This can be one of the interfaces or a special-purpose type that extends or implements one of these interfaces.

For example, one could imagine an image-processing package, called `ImageList`, that returned objects of a new class that implements `List`. In addition to the `List` operations, `ImageList` could support any application-specific operations that seemed desirable. For example, it might provide an `indexImage` operation that returned an image containing thumbnail images of each graphic in the `ImageList`. It's critical to note that even if the API furnishes `ImageList` instances on output, it should accept arbitrary `Collection` (or perhaps `List`) instances on input.

In one sense, return values should have the opposite behavior of input parameters: It's best to return the most specific applicable collection interface rather than the most general. For example, if you're sure that you'll always return a `SortedMap`, you should give the relevant method the return type of `SortedMap` rather than `Map`. `SortedMap` instances are more time-consuming to build than ordinary `Map` instances and are also more powerful. Given that your module has already invested the time to build a `SortedMap`, it makes good sense to give the user access to its increased power. Furthermore, the user will be able to pass the returned object to methods that demand a `SortedMap`, as well as those that accept any `Map`.

Legacy APIs

There are currently plenty of APIs out there that define their own ad hoc collection types. While this is unfortunate, it's a fact of life, given that there was no Collections Framework in the first two major releases of the Java platform. Suppose you own one of these APIs; here's what you can do about it.

If possible, retrofit your legacy collection type to implement one of the standard collection interfaces. Then all the collections you return will interoperate smoothly with other collection-based APIs. If this is impossible (for example, because one or more of the preexisting type signatures conflict with the standard collection interfaces), define an *adapter class* that wraps one of your legacy collections objects, allowing it to function as a standard collection. (The `Adapter` class is an example of a *custom implementation*.)

Retrofit your API with new calls that follow the input guidelines to accept objects of a standard collection interface, if possible. Such calls can coexist with the calls that take the legacy collection type. If this is impossible, provide a constructor or static factory for your legacy type that takes an object of one of the standard interfaces and returns a legacy collection containing the same elements (or mappings). Either of these approaches will allow users to pass arbitrary collections into your API.

Previous page: Compatibility
Next page: End of Trail