The Java™ Tutorials

Trail: Collections

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.

Lesson: Custom Collection Implementations

Many programmers will never need to implement their own `Collection`s classes. You can go pretty far using the implementations described in the preceding sections of this chapter. However, someday you might want to write your own implementation. It is fairly easy to do this with the aid of the abstract implementations provided by the Java platform. Before we discuss *how* to write an implementation, let's discuss why you might want to write one.

Reasons to Write an Implementation

The following list illustrates the sort of custom `Collection`s you might want to implement. It is not intended to be exhaustive:

- **Persistent**: All of the built-in `Collection` implementations reside in main memory and vanish when the program exits. If you want a collection that will still be present the next time the program starts, you can implement it by building a veneer over an external database. Such a collection might be concurrently accessible by multiple programs.
- **Application-specific**: This is a very broad category. One example is an unmodifiable `Map` containing real-time telemetry data. The keys could represent locations, and the values could be read from sensors at these locations in response to the `get` operation.
- **High-performance, special-purpose**: Many data structures take advantage of restricted usage to offer better performance than is possible with general-purpose implementations. For instance, consider a `List` containing long runs of identical element values. Such lists, which occur frequently in text processing, can be *run-length encoded* — runs can be represented as a single object containing the repeated element and the number of consecutive repetitions. This example is interesting because it trades off two aspects of performance: It requires less space but more time than an `ArrayList`.
- **High-performance, general-purpose**: The Java Collections Framework's designers tried to provide the best general-purpose implementations for each interface, but many, many data structures could have been used, and new ones are invented every day. Maybe you can come up with something faster!
- **Enhanced functionality**: Suppose you need an efficient bag implementation (also known as a *multiset*): a `Collection` that offers constant-time containment checks while allowing duplicate elements. It's reasonably straightforward to implement such a collection atop a `HashMap`.
- **Convenience**: You may want additional implementations that offer conveniences beyond those offered by the Java platform. For instance, you may frequently need `List` instances representing a contiguous range of `Integer`s.
- **Adapter**: Suppose you are using a legacy API that has its own ad hoc collections' API. You can write an adapter implementation that permits these collections to operate in the Java Collections Framework. An *adapter implementation* is a thin veneer that wraps objects of one type and makes them behave like objects of another type by translating operations on the latter type into operations on the former.

How to Write a Custom Implementation

Writing a custom implementation is surprisingly easy. The Java Collections Framework provides abstract implementations designed expressly to facilitate custom implementations. We'll start with the following example of an implementation of `Arrays.asList`.

```
public static <T> List<T> asList(T[] a) {
    return new MyArrayList<T>(a);
}

private static class MyArrayList<T> extends AbstractList<T> {

    private final T[] a;

    MyArrayList(T[] array) {
        a = array;
    }

    public T get(int index) {
        return a[index];
    }

    public T set(int index, T element) {
        T oldValue = a[index];
        a[index] = element;
```

```
            return oldValue;
        }

        public int size() {
            return a.length;
        }
    }
```

Believe it or not, this is very close to the implementation that is contained in `java.util.Arrays`. It's that simple! You provide a constructor and the `get`, `set`, and `size` methods, and `AbstractList` does all the rest. You get the `ListIterator`, bulk operations, search operations, hash code computation, comparison, and string representation for free.

Suppose you want to make the implementation a bit faster. The API documentation for abstract implementations describes precisely how each method is implemented, so you'll know which methods to override to get the performance you want. The preceding implementation's performance is fine, but it can be improved a bit. In particular, the `toArray` method iterates over the `List`, copying one element at a time. Given the internal representation, it's a lot faster and more sensible just to clone the array.

```
public Object[] toArray() {
    return (Object[]) a.clone();
}
```

With the addition of this override and a few more like it, this implementation is exactly the one found in `java.util.Arrays`. In the interest of full disclosure, it's a bit tougher to use the other abstract implementations because you will have to write your own iterator, but it's still not that difficult.

The following list summarizes the abstract implementations:

- `AbstractCollection` — a `Collection` that is neither a `Set` nor a `List`. At a minimum, you must provide the `iterator` and the `size` methods.
- `AbstractSet` — a `Set`; use is identical to `AbstractCollection`.
- `AbstractList` — a `List` backed up by a random-access data store, such as an array. At a minimum, you must provide the `positional access` methods (`get` and, optionally, `set`, `remove`, and `add`) and the `size` method. The abstract class takes care of `listIterator` (and `iterator`).
- `AbstractSequentialList` — a `List` backed up by a sequential-access data store, such as a linked list. At a minimum, you must provide the `listIterator` and `size` methods. The abstract class takes care of the positional access methods. (This is the opposite of `AbstractList`.)
- `AbstractQueue` — at a minimum, you must provide the `offer`, `peek`, `poll`, and `size` methods and an `iterator` supporting `remove`.
- `AbstractMap` — a `Map`. At a minimum you must provide the `entrySet` view. This is typically implemented with the `AbstractSet` class. If the `Map` is modifiable, you must also provide the `put` method.

The process of writing a custom implementation follows:

1. Choose the appropriate abstract implementation class from the preceding list.
2. Provide implementations for all the abstract methods of the class. If your custom collection is to be modifiable, you will have to override one or more of the concrete methods as well. The API documentation for the abstract implementation class will tell you which methods to override.
3. Test and, if necessary, debug the implementation. You now have a working custom collection implementation.
4. If you are concerned about performance, read the API documentation of the abstract implementation class for all the methods whose implementations you're inheriting. If any seem too slow, override them. If you override any methods, be sure to measure the performance of the method before and after the override. How much effort you put into tweaking performance should be a function of how much use the implementation will get and how critical to performance its use is. (Often this step is best omitted.)

About Oracle | Contact Us | Legal Notices | Terms of Use | Your Privacy Rights

Copyright © 1995, 2017 Oracle and/or its affiliates. All rights reserved.

Previous page: Previous Lesson
Next page: Interoperability