Collection Helpers

Introduction

Sometimes you need to write your own collection extensions. Perhaps you want to add special behavior when elements are added to a list, or you want to write an `Iterable` that's actually backed by a database query. Guava provides a number of utilities to make these tasks easier for you, and for us. (We *are*, after all, in the business of extending the collections framework ourselves.)

Forwarding Decorators

For all the various collection interfaces, Guava provides `Forwarding` abstract classes to simplify using the decorator pattern.

The `Forwarding` classes define one abstract method, `delegate()`, which you should override to return the decorated object. Each of the other methods delegate directly to the delegate: so, for example, `ForwardingList.get(int)` is simply implemented as `delegate().get(int)`.

By subclassing `ForwardingXXX` and implementing the `delegate()` method, you can override only selected methods in the targeted class, adding decorated functionality without having to delegate every method yourself.

Additionally, many methods have a `standardMethod` implementation which you can use to recover expected behavior, providing some of the same benefits as e.g. extending `AbstractList` or the other skeleton classes in the JDK.

Let's do an example. Suppose you wanted to decorate a `List` so that it logged all elements added to it. Of course, we want to log elements no matter which method is used to add them -- `add(int, E)`, `add(E)`, or `addAll(Collection)` -- so we have to override all of these methods.

```java
class AddLoggingList<E> extends ForwardingList<E> {
  final List<E> delegate; // backing list
  @Override protected List<E> delegate() {
    return delegate;
  }
  @Override public void add(int index, E elem) {
    log(index, elem);
    super.add(index, elem);
  }
  @Override public boolean add(E elem) {
    return standardAdd(elem); // implements in terms of add(int, E)
  }
  @Override public boolean addAll(Collection<? extends E> c) {
    return standardAddAll(c); // implements in terms of add
  }
}
```

Remember, by default, all methods forward directly to the delegate, so overriding `ForwardingMap.put` will not change the behavior of `ForwardingMap.putAll`. Be careful to override every method whose behavior must be changed, and make sure that your decorated collection satisfies its contract.

Generally, most methods provided by the abstract collection skeletons like `AbstractList` are also provided as `standard` implementations in the `Forwarding` decorators.

Interfaces that provide special views sometimes provide `Standard` implementations of those views. For example, `ForwardingMap` provides `StandardKeySet` , `StandardValues` , and `StandardEntrySet` classes, each of which delegate their methods to the decorated map whenever possible, or otherwise, they leave methods that can't be delegated as abstract.

Interface	Forwarding Decorator
`Collection`	[`ForwardingCollection`]
`List`	[`ForwardingList`]
`Set`	[`ForwardingSet`]
`SortedSet`	[`ForwardingSortedSet`]
`Map`	[`ForwardingMap`]
`SortedMap`	[`ForwardingSortedMap`]
`ConcurrentMap`	[`ForwardingConcurrentMap`]
`Map.Entry`	[`ForwardingMapEntry`]
`Queue`	[`ForwardingQueue`]
`Iterator`	[`ForwardingIterator`]
`ListIterator`	[`ForwardingListIterator`]
`Multiset`	[`ForwardingMultiset`]
`Multimap`	[`ForwardingMultimap`]
`ListMultimap`	[`ForwardingListMultimap`]
`SetMultimap`	[`ForwardingSetMultimap`]

PeekingIterator

Sometimes, the normal `Iterator` interface isn't enough.

`Iterators` supports the method `Iterators.peekingIterator(Iterator)` , which wraps an `Iterator` and returns a `PeekingIterator` , a subtype of `Iterator` that lets you `peek()` at the element that will be returned by the next call to `next()` .

Note: the `PeekingIterator` returned by `Iterators.peekingIterator` does not support `remove()` calls after a `peek()` .

Let's do an example: copying a `List` while eliminating consecutive duplicate elements.

```
List<E> result = Lists.newArrayList();
PeekingIterator<E> iter = Iterators.peekingIterator(source.iterator());
while (iter.hasNext()) {
  E current = iter.next();
  while (iter.hasNext() && iter.peek().equals(current)) {
    // skip this duplicate element
    iter.next();
  }
  result.add(current);
}
```

The traditional way to do this involves keeping track of the previous element, and falling back under certain conditions, but that's a tricky and bug-prone business. `PeekingIterator` is comparatively straightforward to understand and use.

AbstractIterator

Implementing your own `Iterator`? [`AbstractIterator`] can make your life easier.

It's easiest to explain with an example. Let's say we wanted to wrap an iterator so as to skip null values.

```
public static Iterator<String> skipNulls(final Iterator<String> in) {
  return new AbstractIterator<String>() {
    protected String computeNext() {
      while (in.hasNext()) {
        String s = in.next();
        if (s != null) {
          return s;
        }
      }
      return endOfData();
    }
  };
}
```

You implement one method, `computeNext()`, that just computes the next value. When the sequence is done, just return `endOfData()` to mark the end of the iteration.

Note: `AbstractIterator` extends `UnmodifiableIterator`, which forbids the implementation of `remove()`. If you need an iterator that supports `remove()`, you should not extend `AbstractIterator`.

AbstractSequentialIterator

Some iterators are more easily expressed in other ways. `AbstractSequentialIterator` provides another way of expressing an iteration.

```
Iterator<Integer> powersOfTwo = new AbstractSequentialIterator<Integer>(1) { // note
the initial value!
  protected Integer computeNext(Integer previous) {
    return (previous == 1 << 30) ? null : previous * 2;
  }
};
```

Here, we implement the method `computeNext(T)`, which accepts the previous value as an argument.

Note that you must additionally pass an initial value, or `null` if the iterator should end immediately. Note that `computeNext` assumes that a `null` value implies the end of iteration -- `AbstractSequentialIterator` cannot be used to implement an iterator which may return `null`.