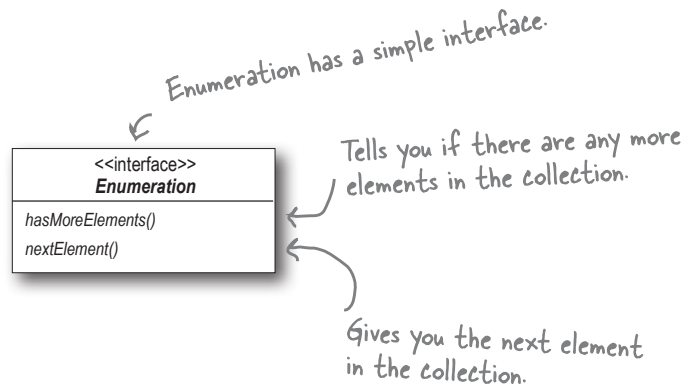


Real-world adapters

Let's take a look at the use of a simple Adapter in the real world (something more serious than Ducks at least)...

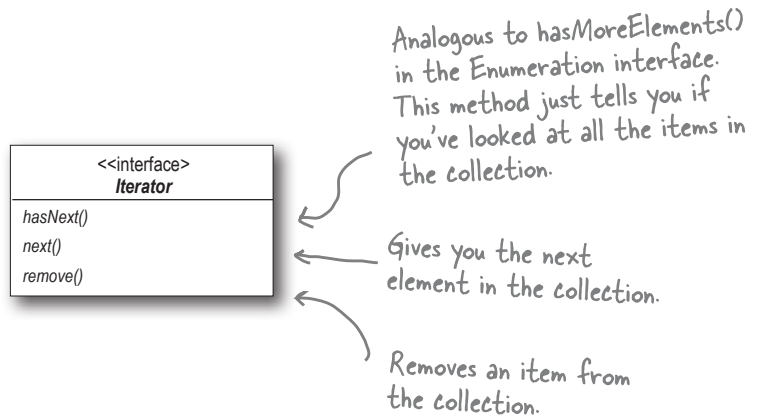
Enumerators

If you've been around Java for a while, you probably remember that the early collection types (Vector, Stack, Hashtable, and a few others) implement a method, `elements()`, which returns an Enumeration. The Enumeration interface allows you to step through the elements of a collection without knowing the specifics of how they are managed in the collection.



Iterators

The more recent Collection classes use an Iterator interface that, like the Enumeration interface, allows you to iterate through a set of items in a collection, and adds the ability to remove items.

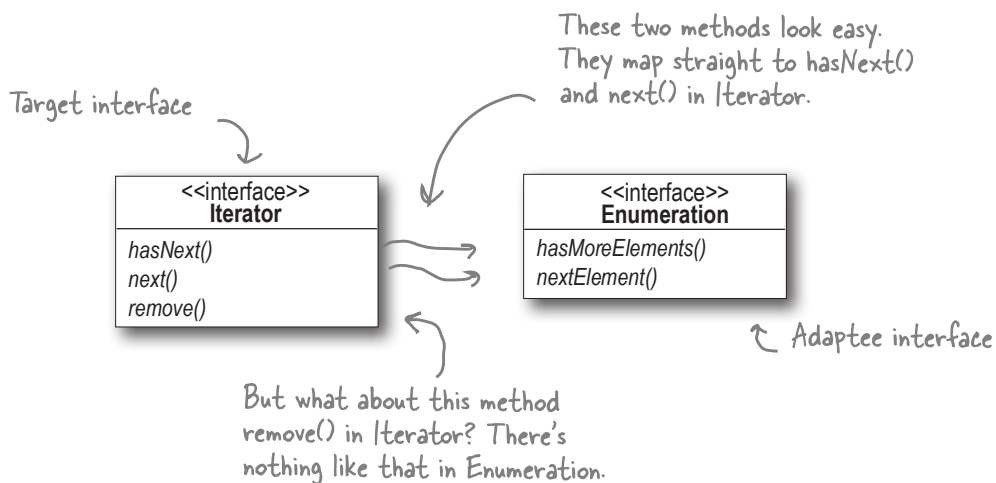


Using Enumerators with code that expects Iterators

We are sometimes faced with legacy code that exposes the Enumeration interface, yet we'd like for our new code to use only Iterators. It looks like we need to build an adapter.

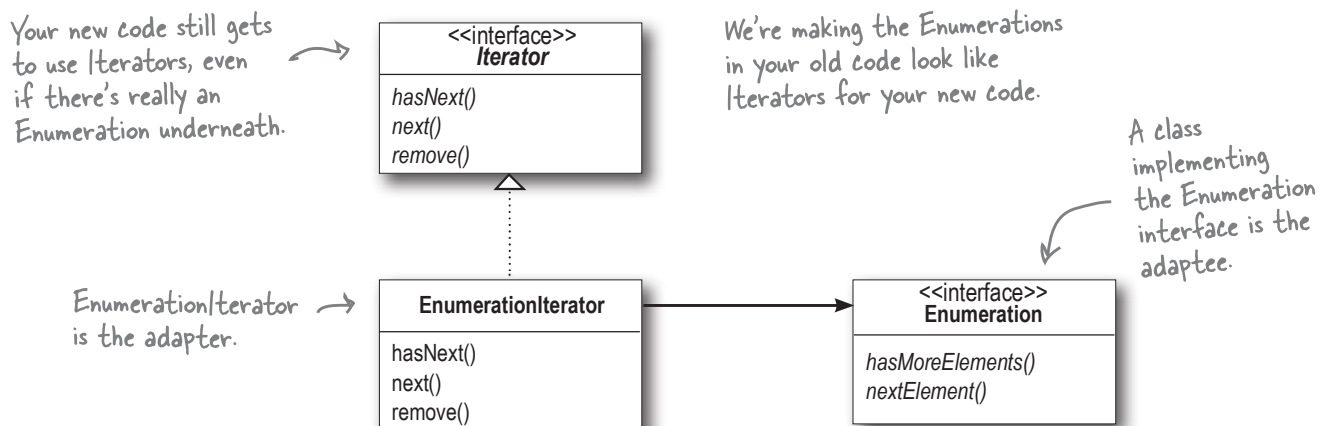
Adapting an Enumeration to an Iterator

First we'll look at the two interfaces to figure out how the methods map from one to the other. In other words, we'll figure out what to call on the adaptee when the client invokes a method on the target.



Designing the Adapter

Here's what the classes should look like: we need an adapter that implements the Target interface and is composed with an adaptee. The `hasNext()` and `next()` methods are going to be straightforward to map from target to adaptee: we just pass them right through. But what do you do about `remove()`? Think about it for a moment (and we'll deal with it on the next page). For now, here's the class diagram:



Dealing with the remove() method

Well, we know Enumeration doesn't support remove(). It's a "read only" interface. There's no way to implement a fully functioning remove() method on the adapter. The best we can do is throw a runtime exception. Luckily, the designers of the Iterator interface foresaw this need and defined the remove() method so that it supports an UnsupportedOperationException.

This is a case where the adapter isn't perfect; clients will have to watch out for potential exceptions, but as long as the client is careful and the adapter is well documented, this is a perfectly reasonable solution.

Writing the EnumerationIterator adapter

Here's simple but effective code for all those legacy classes still producing Enumerations:

```
public class EnumerationIterator implements Iterator<Object> {
    Enumeration<?> enumeration;

    public EnumerationIterator(Enumeration<?> enumeration) {
        this.enumeration = enumeration;
    }

    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    public Object next() {
        return enumeration.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Since we're adapting Enumeration to Iterator, our Adapter implements the Iterator interface...it has to look like an Iterator.

The Enumeration we're adapting. We're using composition, so we stash it in an instance variable.

The Iterator's hasNext() method is delegated to the Enumeration's hasMoreElements() method...

...and the Iterator's next() method is delegated to the Enumeration's nextElement() method.

Unfortunately, we can't support Iterator's remove() method, so we have to punt (in other words, we give up!). Here we just throw an exception.