Java SE Documentation

The For-Each Loop

Iterating over a collection is uglier than it needs to be. Consider the following method, which takes a collection of timer tasks and cancels them:

```
void cancelAll(Collection<TimerTask> c) {
    for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
        i.next().cancel();
}
```

The iterator is just clutter. Furthermore, it is an opportunity for error. The iterator variable occurs three times in each loop: that is two chances to get it wrong. The for-each construct gets rid of the clutter and the opportunity for error. Here is how the example looks with the for-each construct:

```
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask t : c)
        t.cancel();
}
```

When you see the colon (`:`) read it as "in." The loop above reads as "for each `TimerTask t` in `c`." As you can see, the for-each construct combines beautifully with generics. It preserves all of the type safety, while removing the remaining clutter. Because you don't have to declare the iterator, you don't have to provide a generic declaration for it. (The compiler does this for you behind your back, but you need not concern yourself with it.)

Here is a common mistake people make when they are trying to do nested iteration over two collections:

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();

// BROKEN - throws NoSuchElementException!
for (Iterator i = suits.iterator(); i.hasNext(); )
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(i.next(), j.next()));
```

Can you spot the bug? Don't feel bad if you can't. Many expert programmers have made this mistake at one time or another. The problem is that the `next` method is being called too many times on the "outer" collection (`suits`). It is being called in the inner loop for both the outer and inner collections, which is wrong. In order to fix it, you have to add a variable in the scope of the outer loop to hold the suit:

```
// Fixed, though a bit ugly
for (Iterator i = suits.iterator(); i.hasNext(); ) {
    Suit suit = (Suit) i.next();
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(suit, j.next()));
}
```

So what does all this have to do with the for-each construct? It is tailor-made for nested iteration! Feast your eyes:

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
```

The for-each construct is also applicable to arrays, where it hides the index variable rather than the iterator. The following method returns the sum of the values in an `int` array:

```
>// Returns the sum of the elements of a>
int sum(int[] a) {
    int result = 0;
    for (int i : a)
        result += i;
    return result;
}
```

So when should you use the for-each loop? Any time you can. It really beautifies your code. Unfortunately, you cannot use it everywhere. Consider, for example, the `expurgate` method. The program needs access to the iterator in order to remove the current element. The for-each loop hides the iterator, so you cannot call `remove`. Therefore, the for-each loop is not usable for filtering. Similarly it is not usable for loops where you need to replace elements in a list or array as you traverse it. Finally, it is not usable for loops that must iterate over multiple collections in parallel. These shortcomings were known by the designers, who made a conscious decision to go with a clean, simple construct that would cover the great majority of cases.
