用于 Iterable 的对象的增强的 for 循环的时候，它用对 iterator 方法的那些调用代替增强的 for 循环以得到一个 Iterator 对象，然后调用 next 和 hasNext。因此，前面看到的 print 例程由编译器重写，见图 3-8 所示。

```
1       public static <AnyType> void print( Collection<AnyType> coll )
2       {
3           Iterator<AnyType> itr = coll.iterator( );
4           while( itr.hasNext( ) )
5           {
6               AnyType item = itr.next( );
7               System.out.println( item );
8           }
9       }
```

图 3-8 通过编译器使用一个迭代器改写的 Iterable 类型上的增强的 for 循环

由于 Iterator 接口中的现有方法有限，因此，很难使用 Iterator 做简单遍历 Collection 以外的任何工作。Iterator 接口还包含一个方法，叫做 remove。该方法可以删除由 next 最新返回的项(此后，我们不能再调用 remove，直到对 next 再一次调用以后)。虽然 Collection 接口也包含一个 remove 方法，但是，使用 Iterator 的 remove 方法可能有更多的优点。

Iterator 的 remove 方法的主要优点在于，Collection 的 remove 方法必须首先找出要被删除的项。如果知道所要删除的项的准确位置，那么删除它的开销很可能要小得多。下一节我们将要看到一个例子，是在集合中每隔一项删除一项。这个程序用迭代器(iterator)很容易编写，而且比用 Collection 的 remove 方法潜藏着更高的效率。

当直接使用 Iterator(而不是通过一个增强的 for 循环间接使用)时，重要的是要记住一个基本法则：如果对正在被迭代的集合进行结构上的改变(即对该集合使用 add、remove 或 clear 方法)，那么迭代器就不再合法(并且在其后使用该迭代器时将会有 ConcurrentModificationException 异常被抛出)。为避免迭代器准备给出某一项作为下一项(next item)而该项此后或者被删除，或者也许一个新的项正好插入该项的前面这样一些讨厌的情形，有必要记住上述法则。这意味着，只有在需要立即使用一个迭代器的时候，我们才应该获取迭代器。然而，如果迭代器调用了它自己的 remove 方法，那么这个迭代器就仍然是合法的。这是有时候我们更愿意使用迭代器的 remove 方法的第二个原因。

3.3.3 List 接口、ArrayList 类和 LinkedList 类

本节跟我们关系最大的集合就是表(list)，它由 java.util 包中的 List 接口指定。List 接口继承了 Collection 接口，因此它包含 Collection 接口的所有方法，外加其他一些方法。图 3-9 解释其中最重要的一些方法。

```
1   public interface List<AnyType> extends Collection<AnyType>
2   {
3       AnyType get( int idx );
4       AnyType set( int idx, AnyType newVal );
5       void add( int idx, AnyType x );
6       void remove( int idx );
7
8       ListIterator<AnyType> listIterator( int pos );
9   }
```

图 3-9 包 java.util 中 List 接口的子集