get 和 set 使得用户可以访问或改变通过由位置索引 idx 给定的表中指定位置上的项。索引 0 位于表的前端，索引 size() - 1 代表表中的最后一项，而索引 size()则表示新添加的项可以被放置的位置。add 使得在位置 idx 处置入一个新的项(并把其后的项向后推移一个位置)。于是，在位置 0 处 add 是在表的前端进行的添加，而在位置 size()处的 add 是把被添加项作为新的最后项添入表中。除以 AnyType 作为参数的标准的 remove 外，remove 还被重载以删除指定位置上的项。最后，List 接口指定 listIterator 方法，它将产生比通常认为的还要复杂的迭代器。ListIterator 接口将在 3.3.5 节讨论。

List ADT 有两种流行的实现方式。ArrayList 类提供了 List ADT 的一种可增长数组的实现。使用 ArrayList 的优点在于，对 get 和 set 的调用花费常数时间。其缺点是新项的插入和现有项的删除代价昂贵，除非变动是在 ArrayList 的末端进行。LinkedList 类则提供了 List ADT 的双链表实现。使用 LinkedList 的优点在于，新项的插入和现有项的删除均开销很小，这里假设变动项的位置是已知的。这意味着，在表的前端进行添加和删除都是常数时间的操作，由此 LinkedList 更提供了方法 addFirst 和 removeFirst、addLast 和 removeLast、以及 getFirst 和 getLast 等以有效地添加、删除和访问表两端的项。使用 LinkedList 的缺点是它不容易作索引，因此对 get 的调用是昂贵的，除非调用非常接近表的端点(如果对 get 的调用是对接近表后部的项进行，那么搜索的进行可以从表的后部开始)。为了看出差别，我们考察对一个 List 进行操作的某些方法。首先，设我们通过在末端添加一些项来构造一个 List。

```
public static void makeList1( List<Integer> lst, int N )
{
    lst.clear( );
    for( int i = 0; i < N; i++ )
        lst.add( i );
}
```

不管 ArrayList 还是 LinkedList 作为参数被传递，makeList1 的运行时间都是 $O(N)$，因为对 add 的每次调用都是在表的末端进行从而均花费常数时间(可以忽略对 ArrayList 偶尔进行的扩展)。另一方面，如果我们通过在表的前端添加一些项来构造一个 List：

```
public static void makeList2( List<Integer> lst, int N )
{
    lst.clear( );
    for( int i = 0; i < N; i++ )
        lst.add( 0, i );
}
```

那么，对于 LinkedList 它的运行时间是 $O(N)$，但是对于 ArrayList 其运行时间则是 $O(N^2)$，因为在 ArrayList 中，在前端进行添加是一个 $O(N)$操作。

下一个例程是计算 List 中的数的和：

```
public static int sum( List<Integer> lst )
{
    int total = 0;
    for( int i = 0; i < N; i++ )
        total += lst.get( i );
}
```

这里，ArrayList 的运行时间是 $O(N)$，但对于 LinkedList 来说，其运行时间则是 $O(N^2)$，因为在 LinkedList 中，对 get 的调用为 $O(N)$操作。可是，要是使用一个增强的 for 循环，那么它对