使用泛型的全部原因就在于产生编译器错误而不是类型不匹配的运行时异常，所以，泛型集合不是协变的。因此，我们不能把 Collection<Square>作为参数传递到图 1-13 中的方法里去。

```
1   public static double totalArea( Collection<Shape> arr )
2   {
3       double total = 0;
4
5       for( Shape s : arr )
6           if( s != null )
7               total += s.area( );
8
9       return total;
10  }
```

图 1-13 totalArea 方法，如果传递一个 Collection<Square>，则该方法不能运行

现在的问题是，泛型(以及泛型集合)不是协变的(但有意义)，而数组是协变的。若无附加的语法，则用户就会避免使用集合(collection)，因为失去协变性使得代码缺少灵活性。

Java 5 用**通配符**(wildcard)来弥补这个不足。通配符用来表示参数类型的子类(或超类)。图 1-14 描述带有限制的通配符的使用，图中编写一个将 Collection<T>作为参数的方法 totalArea，其中 T *IS-A* Shape。因此，Collection<Shape>和 Collection<Square>都是可以接受的参数。通配符还可以不带限制使用(此时假设为 extends Object)，或不用 extends 而用 super(来表示超类而不是子类)；此外还存在一些其他的语法，我们就不在这里讨论了。

```
1   public static double totalArea( Collection<? extends Shape> arr )
2   {
3       double total = 0;
4
5       for( Shape s : arr )
6           if( s != null )
7               total += s.area( );
8
9       return total;
10  }
```

图 1-14 用通配符修正后的 totalArea 方法，如果传递一个 Collection<Square>，则方法能够正常运行

1.5.4 泛型 static 方法

从某种意义上说，图 1-14 中的 totalArea 方法是泛型方法，因为它能够接受不同类型的参数。但是，这里没有特定类型的参数表，正如在 GenericMemoryCell 类的声明中所做的那样。有时候特定类型很重要，这或许因为下列的原因：

1. 该特定类型用做返回类型；
2. 该类型用在多于一个的参数类型中；
3. 该类型用于声明一个局部变量。

如果是这样，那么，必须要声明一种带有若干类型参数的显式泛型方法。

例如，图 1-15 显示一种泛型 static 方法，该方法对值 x 在数组 arr 中进行一系列查找。通过使用一种泛型方法，代替使用 Object 作为参数的非泛型方法，当在 Shape 对象的数组中查找 Apple 对象时我们能够得到编译时错误。