1 声明数组列表

`ArrayList`类是一个有类型参数的泛型类，它类似于数组，但在添加或删除元素时可以自动调整数组容量。`ArrayList`类在`java.util.ArrayList`包中。

声明数组列表时，要指定数组列表保存的元素对象的类型，用一对尖括号将类名括起来追加到`ArrayList`后面。例如：

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

在Java 10中，最好使用关键字`var`，以避免重复写类名。例如：

```
var staff = new ArrayList<Employee>();
```

如果没有使用`var`关键字，可以省去右边的类型参数，例如：

```
ArrayList<Employee> staff = new ArrayList<>();
```

这称为“菱形语法”。可以结合`new`操作符使用菱形语法。如果将新值赋值给一个变量，或传递给某个方法，或者从某个方法返回，编译器会检查这个变量、参数或方法的泛型类型，然后将这个类型放在`<>`中。在这个例子中，`new ArrayList<>()`将赋值给一个类型为`ArrayList<Employee>`的变量，所以泛型类型为`Employee`。

如果使用`var`声明数组列表，就不要使用菱形语法。在这种情况下，泛型类型默认设置为`Object`。

使用`add`方法可以将元素添加到数组列表末尾。例如：

```
staff.add(new Employee("Harry", 50000, 1989, 10, 1));
```

数组列表管理着一个内部的对象引用数组，如果调用`add`方法时内部数组已经满了，数组列表就会自动创建一个更大的数组，并将所有对象从较小的数组中拷贝到较大的数组中。

如果已经知道或能够估计出数组可能存储的元素数量，就可以在填充数组之前调用`ensureCapacity`方法。`ensureCapacity`方法的功能是分配一个指定大小的内部数组，它的签名为：

```
void ensureCapacity(int capacity)
```

也可以把初始容量传递给`ArrayList`构造器。例如：

```
ArrayList<Employee> staff = new ArrayList<>(100);
```

`size`方法将返回数组列表中包含的实际元素个数，它的签名为：

```
int size()
```

一旦能够确认数组列表的大小不再发生变化，就可以调用`trimToSize`方法。`trimToSize`方法将数组列表的存储容量缩减到当前大小，它的签名为：

```
void trimToSize()
```

一旦缩减了数组列表的大小，添加新元素就需要花时间再次移动存储块，所以应该在确认不会再向数组列表添加任何元素时再调用`trimToSize`方法。

2 访问数组列表元素

`set`方法用于设置指定位置的内容，它的签名为：

```
E set(int index, E obj) // 将值 obj 放在数组列表的指定索引位置
```

> 注意：只有当数组列表的大小大于i时，才能调用`list.set(i, x)`。例如，下面这段代码是错误的：

```
var list = new ArrayList<Employee>(100);
list.set(0, x);
```

> `list`创建之后还没有内容，其`size`为0，此时还没有索引值为0的元素，访问失败。
>
> `set`方法只能用于替换数组列表中已有的元素，而不能用于加入新元素。

`get`方法用于得到指定索引位置的值，它的签名为：

```
E get(int index)
```

有时需要在数组列表的中间插入元素，为此可以使用`add`方法并提供一个索引参数。它的签名为：

```
void add(int index, E obj) // 将 index 位置及以后的元素后移，将 obj 放在 index 位置
```

同样地，可以从数组列表中删除元素。`remove`方法用于删除指定索引位置的元素，并将后面的元素前移，返回值为删除的元素。它的签名为：

```
E remove(int index)
```

利用下面的技巧，既可以灵活地扩展数组，又可以方便地访问数组元素：

```
// 首先，创建一个数组列表，并添加所有的元素
ArrayList<X> list = new ArrayList<>();
while(...)
{
    x = ...;
    list.add(x);
}
// 使用 toArray 方法将数组元素拷贝到一个数组中
X[] a = new X[list.size()];
list.toArray(a);
// 之后，就可以使用方括号来访问数组元素了
```

可以使用`for each`循环遍历数组列表：

```
for (X x : list)
{
    // 对 x 做操作
}
```

与其等价的for循环形式为：

```
for (int i = 0; i < list.size(); i++)
{
    X x = list.get(i);
    // 对 x 做操作
}
```

3 类型化与原始数组列表的兼容性

假设有这样一个遗留下来的类：

```
public class EmployeeDB
{
    public void update(ArrayList list) {...}
    public ArrayList find(String query) {...}
}
```

可以将一个类型化的数组列表传递给原始`ArrayList`，而不需要强制类型转换：

```
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

相反，将一个原始`ArrayList`赋给一个类型化`ArrayList`会得到一个警告：

```
ArrayList<Employee> result = employeeDB.find(query);
// Type safety: The expression of type ArrayList needs unchecked conversion to
conform to ArrayList<Employee>
```

使用强制类型转换并不能避免出现警告：

```
ArrayList<Employee> result = (ArrayList<Employee>)employeeDB.find(query);
// Type safety: The expression of type ArrayList needs unchecked conversion to
conform to ArrayList<Employee>
// Type safety: Unchecked cast from ArrayList to ArrayList<Employee>
```

处于兼容性的考虑，编译器检查到没有发现违反规则的现象之后，就将所有的类型化数组列表转换成原始`ArrayList`对象。在程序运行时，所有的数组列表都是一样的，虚拟机中没有类型参数。

在这种情况下，程序员并不能做什么。在与遗留的代码交互时，要研究编译器的警告，确保这些警告不太严重就行了。一旦确保问题不太严重，就可以用`@SuppressWarings("unchecked")`注解来标记接受强制类型转换的变量，例如：

```
@SuppressWarings("unchecked") ArrayList<Employee> result
    = (ArrayList<Employee>)employeeDB.find(query);
```