CORE JAVA

Volume I: Fundamentals

TWELFTH EDITION

ORACLE

Cay S. Horstmann

Core Java

Volume I: Fundamentals

Twelfth Edition

Once you are reasonably sure that the array list is at its permanent size, you can call the `trimToSize` method. This method adjusts the size of the memory block to use exactly as much storage space as is required to hold the current number of elements. The garbage collector will reclaim any excess memory.

Once you trim the size of an array list, adding new elements will move the block again, which takes time. You should only use `trimToSize` when you are sure you won't add any more elements to the array list.

C++ NOTE: The `ArrayList` class is similar to the C++ `vector` template. Both `ArrayList` and `vector` are generic types. But the C++ `vector` template overloads the `[]` operator for convenient element access. Java does not have operator overloading, so it must use explicit method calls instead. Moreover, C++ vectors are copied by value. If `a` and `b` are two vectors, then the assignment `a = b` makes `a` into a new vector with the same length as `b`, and all elements are copied from `b` to `a`. The same assignment in Java makes both `a` and `b` refer to the same array list.

`java.util.ArrayList<E>` 1.2

- `ArrayList<E>()`

  constructs an empty array list.
- `ArrayList<E>(int initialCapacity)`

  constructs an empty array list with the specified capacity.
- `boolean add(E obj)`

  appends `obj` at the end of the array list. Always returns `true`.
- `int size()`

  returns the number of elements currently stored in the array list. (Of course, this is never larger than the array list's capacity.)
- `void ensureCapacity(int capacity)`

  ensures that the array list has the capacity to store the given number of elements without reallocating its internal storage array.
- `void trimToSize()`

  reduces the storage capacity of the array list to its current size.

5.3.2 Accessing Array List Elements

Unfortunately, nothing comes for free. The automatic growth convenience of array lists requires a more complicated syntax for accessing the elements.

The reason is that the ArrayList class is not a part of the Java programming language; it is just a utility class programmed by someone and supplied in the standard library.

Instead of the pleasant [] syntax to access or change the element of an array, you use the get and set methods.

For example, to set the ith element, use

```
staff.set(i, harry);
```

This is equivalent to

```
a[i] = harry;
```

for an array a. (As with arrays, the index values are zero-based.)

CAUTION: Do not call list.set(i, x) until the *size* of the array list is larger than i. For example, the following code is wrong:

```
var list = new ArrayList<Employee>(100); // capacity 100, size 0
list.set(0, x); // no element 0 yet
```

Use the add method instead of set to fill up an array, and use set only to replace a previously added element.

To get an array list element, use

```
Employee e = staff.get(i);
```

This is equivalent to

```
Employee e = a[i];
```

NOTE: When there were no generic classes, the get method of the raw ArrayList class had no choice but to return an Object. Consequently, callers of get had to cast the returned value to the desired type:

```
Employee e = (Employee) staff.get(i);
```

The raw ArrayList is also a bit dangerous. Its add and set methods accept objects of any type. A call

```
staff.set(i, "Harry Hacker");
```

compiles without so much as a warning, and you run into grief only when you retrieve the object and try to cast it. If you use an ArrayList<Employee> instead, the compiler will detect this error.

You can sometimes get the best of both worlds—flexible growth and convenient element access—with the following trick. First, make an array list and add all the elements:

```
var list = new ArrayList<X>();
while (. . .)
{
   x = . . .;
   list.add(x);
}
```

When you are done, use the `toArray` method to copy the elements into an array:

```
var a = new X[list.size()];
list.toArray(a);
```

Sometimes, you need to add elements in the middle of an array list. Use the `add` method with an index parameter:

```
int n = staff.size() / 2;
staff.add(n, e);
```

The elements at locations `n` and above are shifted up to make room for the new entry. If the new size of the array list after the insertion exceeds the capacity, the array list reallocates its storage array.

Similarly, you can remove an element from the middle of an array list:

```
Employee e = staff.remove(n);
```

The elements located above it are copied down, and the size of the array is reduced by one.

Inserting and removing elements is not terribly efficient. It is probably not worth worrying about for small array lists. But if you store many elements and frequently insert and remove in the middle of a collection, consider using a linked list instead. I explain how to program with linked lists in Chapter 9.

You can use the "for each" loop to traverse the contents of an array list:

```
for (Employee e : staff)
   do something with e
```

This loop has the same effect as

```
for (int i = 0; i < staff.size(); i++)
{
   Employee e = staff.get(i);
   do something with e
}
```

Listing 5.7 is a modification of the `EmployeeTest` program of Chapter 4. The `Employee[]` array is replaced by an `ArrayList<Employee>`. Note the following changes:

- You don't have to specify the array size.
- You use `add` to add as many elements as you like.
- You use `size()` instead of `length` to count the number of elements.
- You use `a.get(i)` instead of `a[i]` to access an element.

Listing 5.7 `arrayList/ArrayListTest.java`

```
1  package arrayList;
2
3  import java.util.*;
4
5  /**
6   * This program demonstrates the ArrayList class.
7   * @version 1.11 2012-01-26
8   * @author Cay Horstmann
9   */
10 public class ArrayListTest
11 {
12    public static void main(String[] args)
13    {
14       // fill the staff array list with three Employee objects
15       var staff = new ArrayList<Employee>();
16
17       staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
18       staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
19       staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));
20
21       // raise everyone's salary by 5%
22       for (Employee e : staff)
23          e.raiseSalary(5);
24
25       // print out information about all Employee objects
26       for (Employee e : staff)
27          System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
28             + e.getHireDay());
29    }
30 }
```

java.util.ArrayList<E> 1.2

- `E set(int index, E obj)`

  puts the value `obj` in the array list at the specified index, returning the previous contents.

- `E get(int index)`

  gets the value stored at a specified index.

- `void add(int index, E obj)`

  shifts up elements to insert `obj` at the specified index.

- `E remove(int index)`

  removes the element at the given index and shifts down all elements above it. The removed element is returned.

5.3.3 Compatibility between Typed and Raw Array Lists

In your own code, you will always want to use type parameters for added safety. In this section, you will see how to interoperate with legacy code that does not use type parameters.

Suppose you have the following legacy class:

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

You can pass a typed array list to the `update` method without any casts.

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

The `staff` object is simply passed to the `update` method.

CAUTION: Even though you get no error or warning from the compiler, this call is not completely safe. The `update` method might add elements into the array list that are not of type `Employee`. When these elements are retrieved, an exception occurs. This sounds scary, but if you think about it, the behavior is simply as it was before generics were added to Java. The integrity of the virtual machine is never jeopardized. In this situation, you do not lose security, but you also do not benefit from the compile-time checks.

Conversely, when you assign a raw ArrayList to a typed one, you get a warning.

```
ArrayList<Employee> result = employeeDB.find(query); // yields warning
```

NOTE: To see the text of the warning, compile with the option `-Xlint:unchecked`.

Using a cast does not make the warning go away.

```
ArrayList<Employee> result = (ArrayList<Employee>) employeeDB.find(query);
   // yields another warning
```

Instead, you get a different warning, telling you that the cast is misleading.

This is the consequence of a somewhat unfortunate limitation of generic types in Java. For compatibility, the compiler translates all typed array lists into raw ArrayList objects after checking that the type rules were not violated. In a running program, all array lists are the same—there are no type parameters in the virtual machine. Thus, the casts (ArrayList) and (ArrayList<Employee>) carry out identical runtime checks.

There isn't much you can do about that situation. When you interact with legacy code, study the compiler warnings and satisfy yourself that the warnings are not serious.

Once you are satisfied, you can tag the variable that receives the cast with the @SuppressWarnings("unchecked") annotation, like this:

```
@SuppressWarnings("unchecked") ArrayList<Employee> result
      = (ArrayList<Employee>) employeeDB.find(query); // yields another warning
```

5.4 Object Wrappers and Autoboxing

Occasionally, you need to convert a primitive type like int to an object. All primitive types have class counterparts. For example, a class Integer corresponds to the primitive type int. These kinds of classes are usually called *wrappers*. The wrapper classes have obvious names: Integer, Long, Float, Double, Short, Byte, Character, and Boolean. (The first six inherit from the common superclass Number.) The wrapper classes are immutable—you cannot change a wrapped value after the wrapper has been constructed. They are also final, so you cannot subclass them.