Chapter 8

Arrays

8.1 Arrays

8.1.1 What is an Array?

To motivate why we might be interested in using arrays, let us implement an app that creates a collection of doubles. We will keep track of the number of milliseconds it takes to create the collection. In the snippet below we use a `LinkedList` to represent the collection.

```
1 output.print("Provide the size of the collection: ");
2 final int SIZE = input.nextInt();
3 long start = System.currentTimeMillis();
4 List<Double> collection = new LinkedList<Double>();
5 for (int i = 0; i < SIZE; i++)
6 {
7    collection.add(new Double(1.0));
8 }
9 output.println(System.currentTimeMillis() - start);
```

If we run the above snippet for different sizes of the collection, we get the following times (in milliseconds).

SIZE	time
100000	28
1000000	267
10000000	13348

Next, we do the same but this time we exploit arrays. We only have to changes line 4 and 7 of the above snippet. We replace them with the following.

```
4 double[] collection = new collection[SIZE];
```

```
7 collection[i] = 1.0;
```

We will discuss the above two lines in detail later in this section. For now, we are merely interested in the output that the modified app produces.

SIZE	time
100000	20
1000000	128
10000000	5035

Comparing the two tables, we can conclude that the app which uses the array takes less time than the one which uses the `LinkedList`.

Arrays are objects, but of a special type. Like other objects, we declare arrays and we create them (but in a special way). An array has a single attribute, which we will discuss later. In principle, we can invoke methods on arrays but we hardly ever do. Arrays also have some special features that ordinary objects do not possess.

As we have seen in the above example, we can exploit an array to implement a collection. We can declare an array of `double`s named `collection` as follows.

```
double[] collection;
```

The type of the variable `collection` is `double[]`. The `[]` denotes that we are dealing with an array. The `double` denotes that the elements stored in the array are of type `double`. The type `double` is known as the base type of the array.

To create an object of type `double[]`, we use the keyword `new`. At the time of creation, we have to fix the length of the array. For example,

```
collection = new double[SIZE];
```

creates an array of `double`s of length `SIZE` (and assigns it to the variable `collection`).

The attribute `length` contains the length of the array. This attribute is final which reflects that the length of an array is fixed. The snippet

```
output.println(collection.length);
```

prints the length of the array `collection`.

The elements of the array can be found in the so-called cells of the array. Each cell has an index. The index of the first cell is 0. The second one has index 1, etcetera. The index of the last cell of the array `collection` is $\texttt{SIZE} - 1$. Initially, each array cell contains the default value of the base type of the array. Since the base type of the array `collection` is `double` and the default value for `double` is 0.0, each cell of `collection` contains 0.0 when the array is created.

The elements of an array can be accessed via their indices. For example, to print the first element and last element of the array `collection` we can use

```
output.println(collection[0]);
output.println(collection[SIZE - 1]);
```

If we provide an invalid index, that is, one that is smaller than 0 or one that is greater than or equal to the length of the array, then an `ArrayIndexOutOfBoundsException` is thrown. For example,

```
output.println(collection[-1]);
```

and

```
output.println(collection[SIZE]);
```

both give rise to such an exception.

To change the content of an array cell, we can assign it a new value. For example,

```
collection[0] = 1.0;
```

assigns 1.0 to the first cell of the array `collection`. Also in this case an `ArrayIndexOutOfBoundsException` is thrown if the used index is invalid.

In the above example, we have used an array the base type of which is a primitive type, namely `double`. However, the base type of an array can also be non-primitive. For example, we can declare and create an array of `Double`s as follows.

```
Double[] collection = new Double[SIZE];
```

Initially, each cell of this array contains the default value of the type `Double` which is null.

8.1.2 Memory Diagrams

Recall that arrays are objects. As a consequence, in our memory diagrams each array will have its own block. In that block we find its single attribute `length` and its value. Furthermore, the block also contains the cells of the array and their values.

For example, if the execution reaches the end of line 4 of the array version of the code snippet of the previous section and the user entered four, then memory can be depicted as follows.

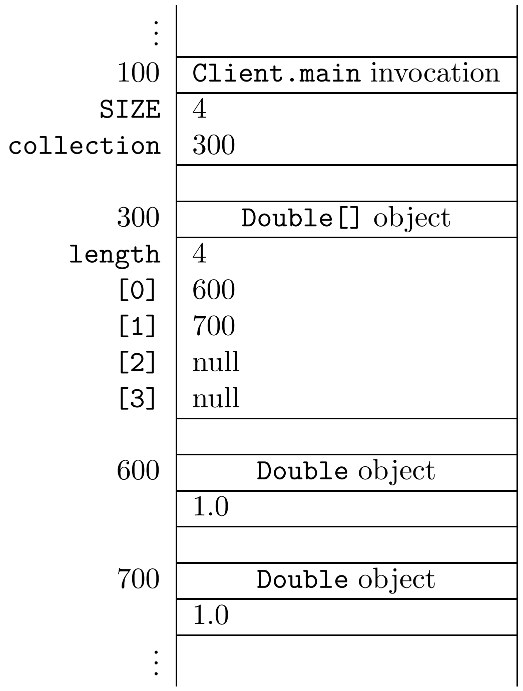
⋮	
100	`Client.main` invocation
`SIZE`	4
`start`	1225548517068
`collection`	300
300	`double[]` object
`length`	4
`[0]`	0.0
`[1]`	0.0
`[2]`	0.0
`[3]`	0.0
⋮	

Let us consider another example.

```
output.print("Provide the size of the collection: ");
final int SIZE = input.nextInt();
Double[] collection = new Double[SIZE];
for (int i = 0; i < SIZE; i++)
{
```

```
6     collection[i] = new Double(1.0);
7  }
```

Once the execution reaches the end of line 6 for the second time, memory can be depicted as follows.



8.2 Implementing a List by means of an Array

8.2.1 Introduction

In this section we implement the `List` interface by means of an array. The API of the `List` interface can be found at this link.

The `List` interface is generic. This is reflected by the type parameter `T` in `List<T>`. The type parameter captures the type of the elements of the list. To use the `List` interface, the client has to provide a non primitive type as argument. For example, `List<Double>` represents a list containing `Double` objects.

Note that the `List` interface extends the `Iterable` interface. As a consequence, when we implement the `List` interface we not only have to provide an implementation for each method specified in the `List` interface, but also for the single method `iterator` specified in the `Iterable` interface.

Note also that our `List` interface is different from the interface `java.util.List`. To simplify matters, our `List` interface contains fewer methods and some of the methods are specified slightly differently.

We provide two implementations of the `List` interface: `BoundedArrayList` and `UnboundedArrayList`. In the former implementation, the maximum size of the list is bound, whereas in the latter implementation the list can grow unboundedly. Both classes are generic. The APIs of the classes can be found at this link and this link.

Like we used a `LinkedList` in Section 8.1.1, we can use a `BoundedArrayList` as follows.

```java
output.print("Provide the size of the collection: ");
final int SIZE = input.nextInt();
long start = System.currentTimeMillis();
List<Double> collection = new BoundedArrayList<Double>(SIZE);
for (int i = 0; i < SIZE; i++)
{
   collection.add(new Double(1.0));
}
output.println(System.currentTimeMillis() - start);
```

If we run the above snippet for different sizes of the collection, we get the following times (in milliseconds).

`SIZE`	`time`
100000	23
1000000	131
10000000	5092

Note that the `BoundedArrayList` is more efficient than the `LinkedList`.

8.2.2 The Class Header

The class header is identical to the one found in the API.

```java
public class BoundedArrayList<T> implements List<T>
```

The type parameter `T` captures the type of the elements of the list. The type parameter `T` can be used within the body of the class as a non primitive type. For example, we can declare a variable named `element` of type `T` as follows.

```java
T element;
```

8.2.3 The Attributes Section

The API of the `BoundedArrayList` class contains a single public attribute named `CAPACITY`. As we can see in the API, this attribute is static and it is a constant of type `int`. In the API, we can also find its value, which is 100. Hence, we introduce the following declaration and initialization.

```java
public static final int CAPACITY = 100;
```

To store the elements of the list, we introduce an array. The first element of the list is stored in the first cell of the array, the second element of the list is stored in the second cell of the array, etcetera. The elements of the list are of type `T`. Hence, it seems natural to introduce an array the base type of which is `T`. However, Java does not allow for arrays the base type of which is a type parameter.[1] Since each class extends the `Object` class, each object is-an `Object`. Hence, an array of `Objects` can store elements of any type. Therefore, we introduce the following array to store the elements of the list.

```java
private Object[] elements;
```

If the list contains n elements, then these elements are stored in the cells with indices 0, ..., $n-1$. The cells with an index greater than or equal to n contain no element, that is, their values are null.

We also keep track of the size of the list by introducing the following attribute.

```java
private int size;
```

To ensure that all the elements stored in the array are of type `T`, we will maintain the following class invariant.

```java
/*
 * class invariant:
 * this.size >= 0 &&
 * this.size <= this.elements.length &&
 * for all 0 <= i < this.size : this.elements[i] != null && this.elements[i]
    instanceof T &&
 * for all this.size <= i < this.elements.length : this.elements[i] == null
 */
```

8.2.4 The Constructors Section

The API of the `BoundedArrayList` class contains three constructors. Both the default constructor and the constructor that takes a capacity as its argument give rise to an empty list, that is, the `size` is zero and all the cells of the array `elements` contain null. As we will see below, we can implement the former by delegating to the latter. In the latter constructor, we set the `size` to zero and we create an array of `Objects` of length `capacity` as follows.

```java
public BoundedArrayList(int capacity)
{
   this.size = 0;
   this.elements = new Object[capacity];
}
```

Note that the cells of the array all contain null, since upon creation all cells of an array contain the default value of the base type of the array and null is the default value of `Object`.

The default constructor can simply delegate to the above implemented constructor as follows.

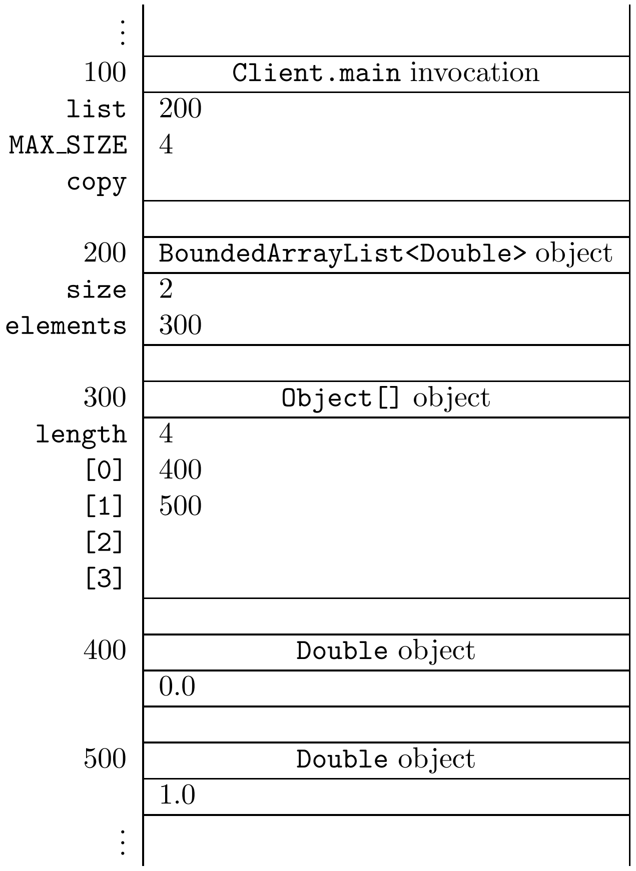
```java
public BoundedArrayList()
{
   this(BoundedArrayList.CAPACITY);
}
```

[1]The reasons why Java disallows such arrays is beyond the scope of these notes.

Before implementing the copy constructor, let us consider the following snippet of client code.

```
final int MAX_SIZE = 4;
List<Double> list = new BoundedArrayList<Double>(MAX_SIZE);
list.add(new Double(0.0));
list.add(new Double(1.0));
List<Double> copy = new BoundedArrayList<Double>(list);
```

When the execution reaches the end of line 4, memory can be depicted as follows.



After the execution reaches the end of line 5, our memory diagram looks as follows.

⋮	
100	`Client.main` invocation
`list`	200
`MAX_SIZE`	4
`copy`	600
200	`BoundedArrayList<Double>` object
`size`	2
`elements`	300
300	`Object[]` object
`length`	4
`[0]`	400
`[1]`	500
`[2]`	
`[3]`	
400	`Double` object
	0.0
500	`Double` object
	1.0
600	`BoundedArrayList<Double>` object
`size`	2
`elements`	700
700	`Object[]` object
`length`	4
`[0]`	400
`[1]`	500
`[2]`	
`[3]`	
⋮	

Note that the arrays at the addresses 300 and 700 contain the same `Double` objects, namely the ones at the addresses 400 and 500. This implies that we have to create a new array in the copy constructor and copy the content of the original array to the new one. Hence, we can implement the copy constructor as follows.

```
public BoundedArrayList(BoundedArrayList list)
{
   this.size = list.size;
   this.elements = new Object[list.elements.length];
   for (int i = 0; i < list.size; i++)
   {
      this.elements[i] = list.elements[i];
   }
}
```

From the class invariant we can conclude that the above code fragment does not throw an `ArrayIndexOutOfBoundEx`

Rather than creating a new array and copying the content from the old one to the new one ourselves, we can also delegate to the `Arrays` class, which is part of the `java.util` package. The static method `copyOf(original, length)` returns a new array and copies the values of the `original` array to the new one, truncating or padding with default values (if necessary) so the returned copy has the specified `length`. Exploiting this method, we can implement the copy constructor as follows.

```
public BoundedArrayList(BoundedArrayList list)
{
   this.size = list.size;
   this.elements = Arrays.copyOf(list.elements, list.elements.length);
}
```

8.2.5 The Methods Section

The accessor `getSize` can be implemented as follows.

```
public int getSize()
{
   return this.size;
}
```

The method `get(index)` returns the element of the list with the given `index`. This corresponds to the element of the array `elements` with the given `index`. Note that the return type of the method `get` is `T` whereas the base type of the array `elements` is `Object`. Hence, we have to cast. If the given index is invalid, that is, it is smaller than zero, or it is greater than or equal to the size of the list, then the method `get` throws an `IndexOutOfBoundsException`.

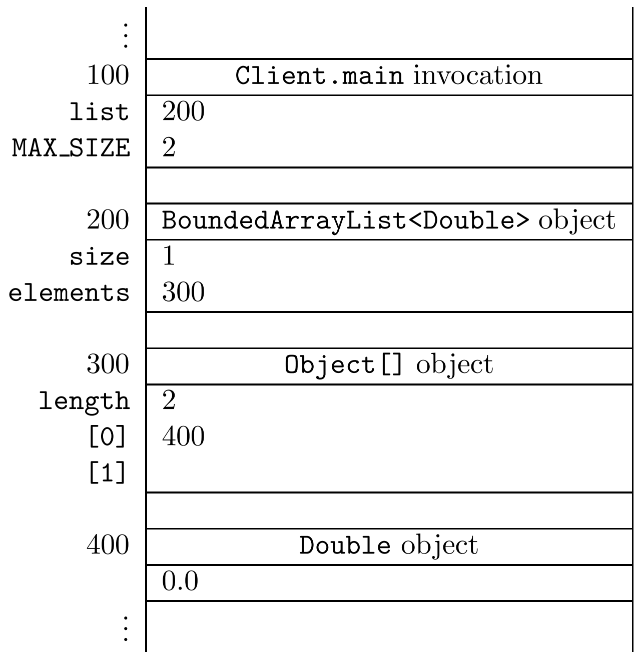
```
public T get(int index) throws IndexOutOfBoundsException
{
   if (index >= 0 && index < this.size)
   {
      return (T) this.elements[i];
   }
   else
   {
      throw new IndexOutOfBoundsException("Index is invalid");
   }
11 }
```

Note that we can conclude from the class invariant that the cast in line 5 will never fail.

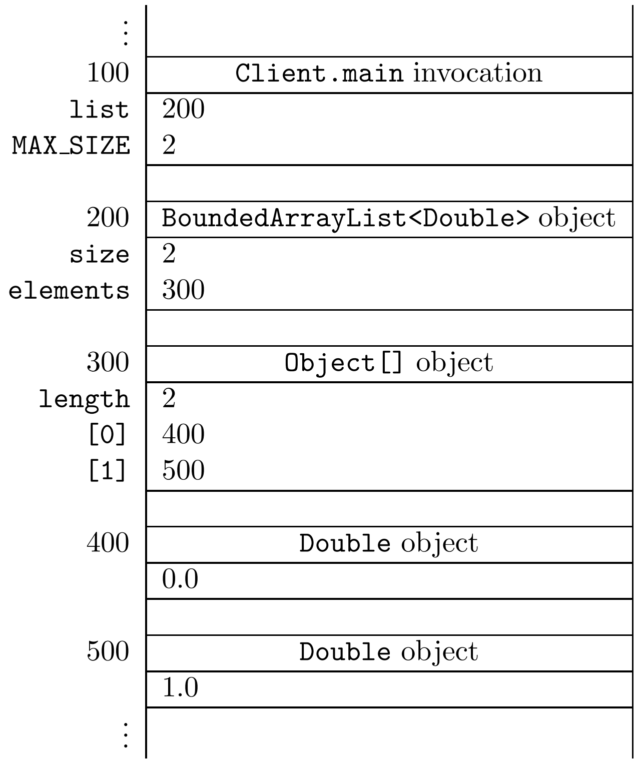
Before implementing the `add` method, let us consider the following client snippet.

```
1 final int MAX_SIZE = 2;
2 List<Double> list = new BoundedArrayList<Double>(MAX_SIZE);
3 list.add(new Double(0.0));
4 list.add(new Double(1.0));
5 list.add(new Double(2.0));
```

Once we reach the end of line 3, memory can be depicted as follows.



In line 4 we add another `Double` object to the collection. We add a reference to a `Double` object to the first available free cell, that is, the first cell that contains null. The index of this cell is the value of the attribute `size`. Furthermore, the attribute `size` has to be incremented by one to reflect that an element has been added to the collection. Since the element has been added successfully, the methods returns true. Once we reach the end of line 4, memory can be depicted as follows.



In line 5 we attempt to add yet another `Double` object to the collection. However, since there is no free cell available, the method simply returns false.

We can implement the `add` method as follows.

```
public boolean add(T element)
{
   boolean added;
   if (this.size == this.elements.length)
   {
      added = false;
   }
   else
   {
      this.elements[this.size] = element;
      this.size++;
      added = true;
   }
   return added;
}
```

The `contains(element)` method checks whether a given `element` is part of the list. This method can be implemented as follows.

```
public boolean contains(T element)
```

```
{
   boolean found = false;
   for (int i = 0; i < this.size; i++)
   {
      found = found || this.elements[i].equals(element);
   }
   return found;
}
```

The variable `found` keeps track whether we have found `element`. This is captured by the following loop invariant.

$$\texttt{found} == \exists 0 \leq j < \texttt{i} : \texttt{this.elements}[j].\texttt{equals}(\texttt{element})$$

From the class invariant we can conclude that the above method does not throw an `ArrayIndexOutOfBoundsExcept`

Once we have found `element`, we can exit the loop. This can be accomplished by changing the condition of the loop as follows.

```
public boolean contains(T element)
{
   boolean found = false;
   for (int i = 0; i < this.size && !found; i++)
   {
      found = found || this.elements[i].equals(element);
   }
   return found;
}
```

Now that we have modified the condition of the loop, we can further simplify the body of the loop as follows.

```
public boolean contains(T element)
{
   boolean found = false;
   for (int i = 0; i < this.size && !found; i++)
   {
      found = this.elements[i].equals(element);
   }
   return found;
}
```

The first part of the `remove(element)` method is (almost) the same as the `contains(element)` method: we try to locate the given `element`. If the element is not found then the method simply returns false. Otherwise, the element has to be removed from the list and true has to be returned. Assume that we have found the element at index i_f in the array. Then we have to move the elements at the indices $i_f + 1$, $i_f + 2$, ... one cell back. This can be accomplished by copying the element from cell $i_f + 1$ to cell i_f, copying the element from cell $i_f + 2$ to $i_f + 1$, ..., and removing the element from the last cell (by setting its value to null).

```
public boolean remove(T element)
{
   boolean found = false;
   int i;
   for (i = 0; i < this.size && !found; i++)
   {
      found = this.elements[i].equals(element);
   }
   if (found)
   {
      for (; i < this.size; i++)
      {
         this.elements[i - 1] = this.elements[i];
      }
      this.elements[this.size - 1] = null;
      this.size--;
   }
   return found;
}
```

Note that we declare the variable `i` outside the scope of the first loop so that we can employ it in the second loop as well. Assume that we have found the element at index i_f. Then `i` is equal to $i_f + 1$ when exiting the first loop.

The following is a loop invariant for the second loop.

$$\forall i_f < j < \texttt{i} : \texttt{this.elements}[j] \text{ has been copied to } \texttt{this.elements}[j-1]$$

Two lists are the same if they contain the same elements in the same order. Hence, the `equals` method can be implemented as follows.

```
public boolean equals(Object object)
{
   boolean equal;
   if (object != null && this.getClass() == object.getClass())
   {
      BoundedArrayList other = (BoundedArrayList) object;
      equal = this.size() == other.size();
      for (int i = 0; i < this.size() && equal; i++)
      {
        equal = equal && this.elements[i].equals(other.elements[i]);
      }
   }
   else
   {
      equal = false;
   }
   return equal;
}
```

For the above loop, we have the following loop invariant.

$$\begin{aligned}\texttt{equal} \quad == \quad & \texttt{this.size}() == \texttt{other.size}() \;\; \&\& \\ & \forall 0 \leq j < \texttt{i} : \texttt{this.elements}[j].\texttt{equals}(\texttt{other.elements}[j])\end{aligned}$$

The hash code of a list `list` is defined as

$$\sum_{0 \leq j < \texttt{list.getSize}()} 31^{\texttt{list.getSize}()-1-j} \times \texttt{list.get}(j).\texttt{hashCode}()$$

We can implement the `hashCode` method as follows.

```
public int hashCode()
{
   final int BASE = 31;
   int hashCode = 0;
   for (int i = 0; i < this.size; i++)
   {
      hashCode = BASE * hashCode + this.elements[i].hashCode();
   }
   return hashCode;
}
```

A loop invariant for the above loop is

$$\texttt{hashCode} == \sum_{\texttt{i} < j < \texttt{this.size}} 31^{j} \times \texttt{this.elements}[j].\texttt{hashCode}()$$

Consider the following snippet of client code.

```
final int MAX_SIZE = 4;
List<Double> list = new BoundedArrayList<Double>(MAX_SIZE);
list.add(new Double(0.0));
list.add(new Double(1.0));
list.add(new Double(2.0));
output.println(list);
```

In line 6 the `toString` method is invoked implicitly. The above code fragment gives rise to the following output.

```
[0.0, 1.0, 2.0]
```

We can implement the `toString` method as follows.

```java
public String toString()
{
   StringBuffer representation = new StringBuffer("[");
   if (this.size != 0)
   {
      for (int i = 0; i < this.size - 1; i++)
      {
         representation.append(this.elements[i]);
         representation.append(", ");
      }
      representation.append(this.elements[this.size - 1]);
   }
   representation.append("]");
   return representation.toString();
}
```

8.2.6 The `iterator` Method

According to the API of the `Iterable` interface, the method `iterator` returns an object of type `Iterator`. Since `Iterator` is an interface we cannot create instances of it. Hence, we have to return an instance of a class that implements the `Iterator` interface. Therefore, we develop a class that implements the `Iterator` interface. We call the class `BoundedArrayListIterator`. The `iterator` method of the `BoundedArrayList` class will be implemented in such a way that it returns an instance of the `BoundedArrayListIterator` class.

The `BoundedArrayListIterator` Class

To specify that our class implements `Iterator<T>`, we introduce the following class header.

```java
public class BoundedArrayListIterator<T> implements Iterator<T>
```

For our class to implement `Iterator<T>`, we have to implement the following methods

```
boolean hasNext()
T next()
void remove()
```

according to the API of the `Iterator` interface. Recall that, by default, all methods specified in an interface are public.

The Attributes Section

Like any other class, we start with choosing the attributes. Let us first have a look at a small fragment of client code that utilizes the `BoundedArrayListIterator` class. The snippet

```
List<Double> list = new BoundedArrayList<Double>();
list.add(new Double(0.0));
list.add(new Double(1.0));
Iterator<Double> iterator = list.iterator();
while (iterator.hasNext())
{
   output.println(iterator.next());
}
```

produces the output

```
0.0
1.0
```

Recall that the `iterator` method will be implemented in such a way that it returns an instance of the `BoundedArrayListIterator` class. Hence, the variable `iterator` refers to a `BoundedArrayListIterator` object. This object needs to have access to the elements of the array `elements` of the `BoundedArrayList` object `list`. Hence, we introduce an attribute `elements` of type `Object[]`. Furthermore, we have to keep track which element to return next. For that purpose, we introduce an attribute `next` of type `int` that contains the index of the cell of the array `elements` which is returned when the method `next` is invoked. Hence, we declare the following two attributes.

```
private Object[] elements;
private int next;
```

Since `next` is used as an index of the array `elements`, we maintain the following class invariant.

```
/*
 * class invariant: this.next >= 0 &&
 * for all 0 <= i < this.elements.length : this.elements[i] instanceof T
 */
```

The Constructor Section

As we already mentioned above, the `elements` attribute of the `BoundedArrayListIterator` object should have access to the elements of the list, which are captured by the `elements` attribute of the `BoundedArrayList` object. The `BoundedArrayList` object can pass this information to the `BoundedArrayListIterator` object as an argument of the constructor. Hence, the header of the constructor of the `BoundedArrayListIterator` class is

```
public BoundedArrayListIterator(Object[] elements)
```

Before implementing this constructor, let us implement the `iterator` method of the `BoundedArrayList` class first.

```
public Iterator<T> iterator()
{
   return new BoundedArrayListIterator<T>(this.elements);
}
```

Since the `BoundedArrayListIterator<T>` class implements the `Iterator<T>` interface, the return statement of the `iterator` method is compatible with the return type of the method.

Let us get back to the implementation of the constructor. We implement it as follows.

```
public BoundedArrayListIterator(Object[] elements)
{
   this.elements = elements;
   this.next = 0;
}
```

The Methods Section

We have left to implement the following methods.

```
boolean hasNext()
T next()
void remove()
```

To test if the iterator has a next element, we have to check if the cell of the array `elements` with index `next` is filled. To ensure that we do not get an `ArrayIndexOurOfBoundsException`, we have to verify that the index `next` is valid, that is, is smaller than `elements.length` (from the class invariant we already know that it is greater than or equal to zero). Hence, we can implement the `hasNext` method as follows.

```
public boolean hasNext()
{
   return this.next < this.elements.length && this.elements[this.next] != null
       ;
}
```

Recall that if the left hand side of the conjunction, `this.next < this.elements.length`, evaluates to false, then the right hand side of the conjunction, `this.elements[this.next] != null`, is not evaluated in Java.

The `next` method returns the next element. If no such element exists, then a `NoSuchElementException` is thrown. This method can be implemented as follows.

```
public T next() throws NoSuchElementException
{
   if (this.hasNext())
   {
      T element = (T) this.elements[this.next];
      this.next++;
      return element;
   }
   else
   {
      throw new NoSuchElementException("Iterator has no more elements");
   }
}
```

According to the API of the `Iterator` interface, the `remove` method is optional. The API also specifies that an `UnsupportedOperationException` has to be thrown if the method is not supported. To keep the class as simple as possible, we decided not to support the method. Hence, the `remove` method can be implemented as follows.

```
public void remove() throws UnsupportedOperationException
{
   throw new UnsupportedOperationException("Iterator does not support remove")
       ;
}
```

8.2.7 Unbounded Size

Whereas the maximum size of a `BoundedArrayList` is fixed, an `UnboundedArrayList` can grow without any limit. In the `UnboundedArrayList` class, the only method that needs to be implemented differently from the `BoundedArrayList` class is the `add` method. In case there is no free cell available for the element to be added to the list, we simply double the length of the array. That is, we create a new array of twice the length and copy all elements from the original array to the new one.[2]

```
public boolean add(T element)
{
   if (this.size == this.elements.length)
   {
      this.elements = Array.copyOf(this.elements, this.elements.length * 2);
   }
   this.elements[this.size] = element;
   this.size++;
   return true;
}
```

[2]Doubling the length of the array is much more efficient than adding a single cell to the array. The details are beyond the scope of these notes.

8.3 Command Line Arguments

The main method of an app has the following header.

```
1 public static void main(String[] args)
```

The method has a single parameter of type `String[]`, an array of `Strings`. This array contains the so called command line arguments.

The next app uses the command line arguments.

```
 1 public class StudentClient
 2 {
 3    public static void main(String[] args)
 4    {
 5       PrintStream output = System.out;
 6
 7       if (args.length == 0)
 8       {
 9          output.println("Provide a student ID as the command line argument");
10       }
11       else
12       {
13          final String VALID = "\\d{9}";
14          if (args[0].matches(VALID))
15          {
16             Student student = new Student(args[0]);
17             ⋮
18
19          }
20          else
21          {
22             output.println("Provide a valid student ID as the command line
                   argument");
23          }
24       }
25    }
26 }
```

In line 7 we check if any command line arguments are provided. In line 13 and 15, the first command line argument is used.

When running the app, the ID 123456789 can be provided as a command line argument as follows.

```
java StudentClient "123456789"
```

8.4 Beyond the Basics

8.4.1 The `getSize` Method

The attribute `size` is redundant since it can be computed from the attribute `elements` as follows.

```
1 public int getSize()
2 {
3    int size = 0;
4    while (size < this.elements.length && this.elements[size] != null)
5    {
6       size++;
7    }
8    return size;
9 }
```

This implies that we do not have to introduce the attribute `size`. However, the attribute `size` somewhat simplifies our code and makes it more efficient as we will see below.

Let us determine the running time[3] of the above `getSize` method. First, let us estimate the number of elementary operations that are performed at each line of the method.

line	number
3	3
4	11
6	3
8	2

Next, we determine how often each line is executed. Obviously, line 3 and 8 are only executed once. How often line 4 and 6 are executed depends on the number elements that are stored in the array `elements`. Assume that this array contains n elements. Then line 4 is executed $n+1$ times and line 6 is executed n times. Hence, we estimate the total number of elementary operations to be

$$3 + (n+1) \times 11 + n \times 3 + 2 = 14n + 16.$$

Therefore, (an estimate of) the number of elementary operations performed by the above `getSize` method can be represented by the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$f(n) = 14n + 16,$$

where n is the number of elements stored in the array `elements`.

Proposition 1 *$f \in O(n)$.*

Proof According to Definition 3, we have to show that

$$\exists M \in \mathbb{N} : \exists F \in \mathbb{N} : \forall n \leq M : 14n + 16 \leq F \times n.$$

[3]We refer the reader to Appendix A for an introduction to running time analysis.

First, we have to pick a particular "minimal size" M and a particular "factor" F. We pick $M = 16$ and $F = 15$. Then it remains to show that

$$\forall n \geq 16 : 14n + 16 \leq 15n.$$

Let n be an arbitrary number with $n \geq 16$. Then

$$14n + 16 \leq 14n + n = 15n.$$

$\square$

Hence, we say that the running time of the above `getSize` method is $O(n)$, where n is the number elements of the list.

Let us also determine the running time of the `getSize` method of Section 8.2.5. We estimate the number of elementary operations executed in line 3 to be 3. This line is only executed once. Hence, (an estimate of) the number of elementary operations performed by the `getSize` method can be represented by the function $g : \mathbb{N} \to \mathbb{N}$ defined by

$$g(n) = 3,$$

where n is the number of elements stored in the array `elements`.

Proposition 2 *$g \in O(1)$.*

Proof According to Definition 3, we have to show that

$$\exists M \in \mathbb{N} : \exists F \in \mathbb{N} : \forall n \leq M : 3 \leq F \times 1.$$

First, we have to pick a particular "minimal size" M and a particular "factor" F. We pick $M = 1$ and $F = 3$. Then it remains to show that

$$\forall n \geq 1 : 3 \leq 3 \times 1$$

which is obviously the case. $\square$

Hence, we say that the running time of the `getSize` method is $O(1)$.

The implementation presented in Section 8.2.5 only performs a constant number of elementary operations, no matter how many elements the list contains. However, if we were not to introduce the (redundant) attribute `size`, then we would have to implement the `getSize` method as described above. In that case, the number of elementary operations would depend on the number elements of the list. This would be less efficient, especially if the number of elements of the list is large.

8.4.2 The `contains` and `remove` Methods

As we already mentioned in Section 8.2.5, the first part of the `remove` method is almost identical to the `contains` method. To avoid code duplication, we can introduce an auxiliary method that consists of the common part.

The method `find(element)` tries to find an index at which the given `element` can be found in the list. If no such index exists, then the method returns a special value different from any valid index, namely -1.

```java
private int find(T element)
{
   int index = -1;
   for (int i = 0; i < this.size && index == -1; i++)
   {
      if (this.elements[i].equals(element))
      {
         index = i;
      }
   }
   return index;
}
```

We have that

$$\begin{array}{ll} & (\texttt{index} == -1\ \ \&\&\ \ \forall 0 \leq j < \texttt{i} : !\texttt{this.element}[j].\texttt{equals}(\texttt{element})) \\ || & (\texttt{index}! = -1\ \ \&\&\ \ \texttt{this.element}[\texttt{index}].\texttt{equals}(\texttt{element})) \end{array}$$

is a loop invariant for the above loop.

Using the `find` method, we can implement the `contains` method as follows.

```java
public boolean contains(T element)
{
   return this.find(element) != -1;
}
```

We can also use the `find` method to implement the `remove` method.

```java
public boolean remove(T element)
{
   int index = this.find(element);
   if (index != -1)
   {
      for (int i = index + 1; i < this.size; i++)
      {
         this.elements[i - 1] = this.elements[i];
      }
      this.elements[this.size - 1] = null;
      this.size--;
   }
   return index != -1;
}
```

8.4.3 The `BoundedArrayListIterator` as an Inner Class

The `BoundedArrayListIterator` class is only used in the `BoundedArrayList` class. To ensure that only the `BoundedArrayList` class can make use of the `BoundedArrayListIterator` class, we can define the latter class as a private inner class of the former class as follows.

```
public class BoundedArrayList<T> implements List<T>
{
   private int size;
   private Object[] elements;

   ⋮

   private class BoundedArrayListIterator<T> implements Iterator<T>
   {
      ⋮
   }
}
```

Note that the class `BoundedArrayListIterator` is private and, hence, will not show up in the API. Also note that the class `BoundedArrayListIterator` is defined within the scope of the `BoundedArrayList` class and, hence, is not visible outside the latter class.

Appendix A

Running Time Analysis

We use $\mathbb{N}$ to denote the set $\{0, 1, 2, \ldots\}$ of natural numbers. Let $f : \mathbb{N} \to \mathbb{N}$ be a function from natural numbers to natural numbers. We will associate such a function with a piece of Java code. This function will capture the answer to the question "How many elementary operations are performed at most when executing the Java code for an input of a given size?"

Consider the following method.

```
1  /**
2   * Returns the factorial of the given number.
3   *
4   * @param n a number.
5   * @pre. n > 0
6   * @return the factorial of the given number.
7   */
8  public static long factorial(int n)
9  {
10    long factorial = 1;
11    int i = 1;
12    while (i <= n)
13    {
14       factorial *= i;
15       i++;
16    }
17    return factorial;
18 }
```

Let us first estimate how many elementary operations are performed at each line of the above method.

line	number
10	2
11	2
12	4
14	4
15	3
17	2

For example, in line 14 we

- look up the value of `i`,
- look up the value of `factorial`,
- multiply these two values, and
- assign the result of the multiplication to `factorial`.

We will come back to the fact that these are just estimates. Next, we determine how often each line is executed. Obviously, line 10, 11 and 17 are executed once. How often line 12, 14 and 15 are executed depends on the value of `n`. Assume the value of the variable `n` is the integer n. Then, line 12, 14 and 15 are executed n times. Therefore, we estimate the total number of elementary operations to be

$$2 + 2 + n \times 4 + n \times 4 + n \times 3 + 2 = 11n + 6.$$

Hence, (an estimate of) the total number of elementary operations can be captured by a function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = 11n + 6,$$

where n represents the value of `n`.

The actual number of elementary operations that are performed depends on many factors. For example, some computers can assign a value to a variable of type `long` in a single operation whereas other computers may need two. Hence, estimates like

$$f_1(n) = 14n + 12$$

and

$$f_2(n) = 7n + 3$$

are equally reasonable. The big O notation provides us with an abstraction that can capture all these estimates. The functions f, f_1 and f_2 are all elements of $O(g)$, where the function $g : \mathbb{N} \to \mathbb{N}$ is defined by

$$g(n) = n.$$

Why are f, f_1 and f_2 elements of $O(g)$? For that we need the formal definition of the big O notation.

Definition 3 *Let $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ be functions. Then $f \in O(g)$ if*

$$\exists M \in \mathbb{N} : \exists F \in \mathbb{N} : \forall n \geq M : f(n) \leq F \times g(n),$$

That is, we can find a particular "minimal size" M and a particular "factor" F such that for all inputs of size n that are greater than or equal to the "minimal size," we have that $f(n) \leq F \times g(n)$.

The above introduced function g, which assigns to n the value n, is often simply denoted by n and, hence, we often write $f \in O(n)$ instead of $f \in O(g)$.

Proposition 4 $f \in O(n)$.

Proof To prove that $f \in O(n)$ we need to pick a particular M and a particular F. We pick $M = 6$ and $F = 12$. Then it remains to show that

$$\forall n \geq 6 : 11n + 6 \leq 12n,$$

that is, $11n + 6 \leq 12n$ for all $n \geq 6$. Let n be an arbitrary natural number with $n \geq 6$. Then

$$11n + 6 \leq 11n + n = 12n.$$

Hence, the above is true. □

Similarly, we can prove that $f_1 \in O(n)$. This time we pick $M = 12$ and $F = 15$. It remains to prove that

$$\forall n \geq 12 : 14n + 12 \leq 15n,$$

which is trivially true. To prove that $f_2 \in O(n)$, we pick $M = 8$ and $F = 3$. In this case, it remains to prove that

$$\forall n \geq 8 : 7n + 3 \leq 8n,$$

which is obviously true.

Let us consider another Java snippet.

```java
/**
 * Sorts the given array.
 *
 * @param a an array of integers.
 * @pre. a != null && a.length > 0
 */
public static void sort(int[] a)
{
   for (int i = 0; i < a.length; i++)
   {
      int min = i;
      for (int j = i + 1; j < a.length; j++)
13          {
14             if (a[j] < a[min])
15             {
16                min = j;
17             }
18          }
19          int temp = a[i];
20          a[i] = a[min];
21          a[min] = temp;
22       }
23    }
```

Again, let us first estimate how many elementary operations are performed at each line of the above method.

line	number
9	10
11	2
12	10
14	8
16	2
19	4
20	6
21	4

Next, we determine how often each line is executed. Line 9, 11, 19, 20 and 21 are executed n times, where n is the length of the array. Line 12 and 14 are executed

$$(n-1)+(n-2)+\cdots+1=\sum_{i-1}^{n-1} i=\frac{n(n-1)}{2}$$

times.[1]

[1]We can prove that for all $n \geq 1$, $\sum_{i-1}^{n-1} i = \frac{n(n-1)}{2}$ by induction on n. In the base case, when $n = 1$, we have that $0 = \frac{1\times 0}{2}$. In the inductive step, let $n > 1$ and assume that $\sum_{i-1}^{n-2} i = \frac{(n-1)(n-2)}{2}$ (the induction hypothesis). Then

$$\begin{array}{rcll}
\sum_{i-1}^{n-1} i & = & (\sum_{i-1}^{n-2} i) + (n-1) & \\
& = & \frac{(n-1)(n-2)}{2} + (n-1) & \text{[induction hypothesis]} \\
& = & \frac{n^2-3n+2+2n-2}{2} & \\
& = & \frac{n^2-n}{2} & \\
& = & \frac{n(n-1)}{2}. &
\end{array}$$

Note that we do not know how often line 16 is executed. However, we do know that line 16 is executed at most $\frac{n(n-1)}{2}$ times. An upperbound of the total number of elementary operations that is performed is

$$\begin{aligned} & n \times 10 + n \times 2 + \frac{n(n-1)}{2} \times 10 + \frac{n(n-1)}{2} \times 8 + \frac{n(n-1)}{2} \times 2 + n \times 4 + n \times 6 + n \times 4 \\ = \quad & 10n^2 + 16n. \end{aligned}$$

This can be captured by a function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = 10n^2 + 16n.$$

To abstract from the estimates, we again exploit the big O notation and show that $f \in O(n^2)$, where we use n^2 to denote the function that maps n to n^2.

Proposition 5 *$f \in O(n^2)$.*

Proof To show that $f \in O(n^2)$, we pick $M = 16$ and $F = 11$. It remains to prove that

$$\forall n \geq 16 : 10n^2 + 16n \leq 11n^2.$$

Let $n \geq 16$. Then

$$10n^2 + 16n \leq 10n^2 + n^2 = 11n^2.$$

That concludes the proof. □