# Lecture 10: Arrays

*11:00 AM, Feb 14, 2020*

## Contents

### Motivating Question

How can we quickly access elements by position in a list?

### Objectives

By the end of this lecture, you will know:

- What arrays are and how they differ from lists

## 1 Accessing Items by Position in a List

Imagine that you have a list of items with ranking, such as Rolling Stone's 500 greatest rock-and-roll songs of all time. You want to access the song titles based on the ranking (such as asking for the 174th song, which happens to be "Dancing Queen" by ABBA). For our `LinkList` class to support this operation, we need to add a method like the following:

```
// return the item at the given index (position), numbered from 0
public T get(int index) {
    int currIndex = 0;
    Node current = this.start;
    while (currIndex < index) {
        index = index + 1;
        current = current.next;
    }
    return current.item;
}
```

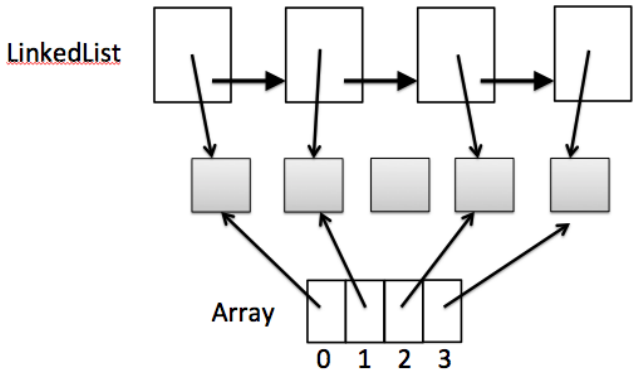
What is the running time of `get`? It's proportional to the index itself, which means it would be linear-time in the worst case. Is it possible to do better?

As we saw with the `contains` method, our `get` code effectively follows references from object to object in memory to traverse a list. So in order to speed this up, we'd need a different organization of the data in the heap that avoids the need to follow the references. How else might we organize the data in memory?

## 2 Arrays

Think about how a LinkedList is laid out in memory: we have a bunch of Node objects linked together through their `next` fields. The nodes can be scattered in memory, and the next fields need

space in addition to the data. What if instead we put the items in consecutive memory locations, as in the following picture:




The consecutive-location data structure is called an *array*. An array has a set number of slots for elements (which can be changed if needed later), in consecutive memory locations. This saves space, while also making it much faster to get to items based on their positions. For example, if you ask for the fifth element in the array, the underlying language implementation can compute the exact memory location rather than traverse `next` references through a linked list. Roughly, the location is

```
arrayLocation + (posWanted * perElementSpace)
```

where `perElementSpace` is the amount of space Java needs to store the reference to each item (the item objects can be anywhere in memory).

Arrays exist in nearly all programming languages. They are provided by default in Java, so you don't have to import anything to use them. For the rest of this lecture, we are going to use the built-in arrays. We are NOT trying to implement them ourselves.

## 2.1 An Example of Arrays

The following code creates an array of 5 strings, representing names:

```
String[] names = new String[5]
```

Square brackets are used for arrays. The notation `type[]` means "an array whose items are of the given type". You can put any type that you want into an array, but an individual array can only hold one type of value (the type can be an interface, which would allow some variation in the concrete classes supported).

We also use square brackets to store items or to access items by position. Positions are numbered starting from zero. An array of five items thus has positions 0 through 4.

```
names[0] = ''Kathi'';  \\ stores a name in the first position of the array
names[3]  \\ accesses the fourth name in the array
```

### 2.2 Traversing Arrays

Iterators are not defined on arrays. Thus, you can't use for loops of the style `for (Integer n : anArray)`, which iterates over the list *contents*. Instead, you use a loop version that iterates over the *indices*, using the [] notation to access the array item in each position.

```
for (int i = 0; i < 5; i++) {
  if (names[i].equals(''Kathi''))
    System.out.println(''found Kathi'')
}
```

### 2.3 Summary: Arrays vs Lists

Here's a table summarizing the differences between arrays and (linked) lists.

| **Feature** | **Array** | **LinkedList** |
|---|---|---|
| memory layout | consecutive elements | elements scattered in memory |
| time to access an element | constant | linear |
| time to add an element | constant if position exists (else must extend array) | constant if inserting at an end |
| space used | proportional to number of elements declared up front | proportional to number of elements in list; also needs space for next references |
| time to remove element | depends on position, since must shift remaining elements over to keep items consecutive | linear |
| Iteration | position-based for loop | iterator-based for or while |

Sometimes, you want both the convenience of iterators and constant-time access to elements. In that case, you can use a Java class called `ArrayList`. They take a bit more space than arrays, and they can only be used with objects (as opposed to plain int or bools). They are often a reasonable choice if you use each of insert, delete, and access with high frequency.

## 3 Summary: Different Kinds of Loops

We have seen three different kinds of loops across the last two lectures:

- `while` loops have you continue a computation until some condition has occured. Here, we saw `while` loops for traversing a list, but we could have also been repeating other tasks (such as asking a user to type in numbers, stopping when they enter the word "done").
- Iterator-style `for` loops visit each element in a list (or other collection of data) exactly once, letting you perform some aggregating computation over each element.
- Position-based `for` loops let you repeat a computation on each item in some formula-defined sequence, such as all the numbers from 0 to the size of an array, or all even numbers.

Usually, which type of loop you use is determined by a combination of the data structure you are using (which style of for-loop) and the computation to perform (for-vs-while). Unless you need the indices as part of your computation into the list, you should use the iterator-style

---

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.