# CHAPTER 3
# Lists

Lists are one of the most common organizing tools people use in their day-to-day lives. We have to-do lists, grocery lists, top-ten lists, bottom-ten lists, and many other types. Our computer programs can also use lists, particularly if we only have a few items to store in list form. Lists are especially useful if we don't have to perform searches on the items in the list or put them into some type of sorted order. When we need to perform long searches or complex sorts, lists become less useful, especially with more complex data structures.

This chapter presents the creation of a simple list class. We start with the definition of a list abstract data type (ADT) and then demonstrate how to implement the ADT. We wrap up the chapter with some problems that are best solved with lists.

## A List ADT

In order to design an ADT for a list, we have to provide a definition of the list, including its properties, as well as the operations performed on it and by it.

A list is an ordered sequence of data. Each data item stored in a list is called an *element*. In JavaScript, the elements of a list can be of any data type. There is no predetermined number of elements that can be stored in a list, though the practical limit will be the amount of memory available to the program using the list.

A list with no elements is an *empty* list. The number of elements stored in a list is called the *length* of the list. Internally, the number of elements in a list is kept in a `listSize` variable. You can *append* an element to the end of a list, or you can *insert* an element into a list after an existing element or at the beginning of a list. Elements are deleted from a list using a *remove* operation. You can also *clear* a list so that all of its current elements are removed.

The elements of a list are displayed using either a `toString()` operation, which displays all the elements, or with a `getElement()` operation, which displays the value of the *current* element.

Lists have properties to describe location. There is the *front* of a list and the *end* of a list. You can move from one element of a list to the next element using the `next()` operation, and you can move backward through a list using the `prev()` operation. You can also move to a numbered position in a list using the `moveTo(n)` operation, where *n* specifies the position to move to. The `currPos` property indicates the current position in a list.

The List ADT does not specify a storage function for a list, but for our implementation will use an array named `dataStore`.

Table 3-1 shows the complete List ADT.

*Table 3-1. ADT List*

| | |
|---|---|
| `listSize` (property) | Number of elements in list |
| `pos` (property) | Current position in list |
| `length` (property) | Returns the number of elements in list |
| `clear` (function) | Clears all elements from list |
| `toString` (function) | Returns string representation of list |
| `getElement` (function) | Returns element at current position |
| `insert` (function) | Inserts new element after existing element |
| `append` (function) | Adds new element to end of list |
| `remove` (function) | Removes element from list |
| `front` (function) | Sets current position to first element of list |
| `end` (function) | Sets current position to last element of list |
| `prev` (function) | Moves current position back one element |
| `next` (function) | Moves current position forward one element |
| `currPos` (function) | Returns the current position in list |
| `moveTo` (function) | Moves the current position to specified position |

## A List Class Implementation

A `List` class implementation can be taken straight from the List ADT we just defined. We'll start with a definition of a constructor function, though it is not part of the ADT:

```
function List() {
   this.listSize = 0;
   this.pos = 0;
   this.dataStore = []; // initializes an empty array to store list elements
   this.clear = clear;
   this.find = find;
```

```
      this.toString = toString;
      this.insert = insert;
      this.append = append;
      this.remove = remove;
      this.front = front;
      this.end = end;
      this.prev = prev;
      this.next = next;
      this.length = length;
      this.currPos = currPos;
      this.moveTo = moveTo;
      this.getElement = getElement;
      this.length = length;
      this.contains = contains;
   }
```

## Append: Adding an Element to a List

The first function we'll implement is the `append()` function. This function appends a new element onto the list at the next available position, which will be equal to the value of the `listSize` variable:

```
   function append(element) {
      this.dataStore[this.listSize++] = element;
   }
```

After the element is appended, `listSize` is incremented by 1.

## Remove: Removing an Element from a List

Next, let's see how to remove an element from a list. `remove()` is one of the harder functions to implement in the `List` class. First, we have to find the element in the list, and then we have to remove it and adjust the space in the underlying array to fill the hole left by removing an element. However, we can simplify the process by using the `splice()` mutator function. To start, let's define a helper function, `find()`, for finding the element to remove:

```
   function find(element) {
      for (var i = 0; i < this.dataStore.length; ++i) {
         if (this.dataStore[i] == element) {
            return i;
         }
      }
      return -1;
   }
```

## Find: Finding an Element in a List

The find function simply iterates through dataStore looking for the specified element. If the element is found, the function returns the position where the element was found. If the element wasn't found, the function returns -1, which is a standard value to return when an element can't be found in an array. We can use this value for error checking in the remove() function.

The remove() function uses the position returned by find() to splice the dataStore array at that place. After the array is modified, listSize is decremented by 1 to reflect the new size of the list. The function returns true if an element is removed, and false otherwise. Here is the code:

```
function remove(element) {
   var foundAt = this.find(element);
   if (foundAt > -1) {
      this.dataStore.splice(foundAt,1);
      --this.listSize;
      return true;
   }
   return false;
}
```

## Length: Determining the Number of Elements in a List

The length() function returns the number of elements in a list:

```
function length() {
   return this.listSize;
}
```

## toString: Retrieving a List's Elements

Now is a good time to create a function that allows us to view the elements of a list. Here is the code for a simple toString() function:

```
function toString() {
    return this.dataStore;
}
```

Strictly speaking, this function returns an array object and not a string, but its utility is in providing a view of the current state of an object, and just returning the array works adequately for this purpose.

Let's take a break from implementing our List class to see how well it works so far. Here is a short test program that exercises the functions we've created so far:

```
var names = new List();
names.append("Cynthia");
names.append("Raymond");
names.append("Barbara");
print(names.toString());
names.remove("Raymond");
print(names.toString());
```

The output from this program is:

```
Cynthia,Raymond,Barbara
Cynthia,Barbara
```

## Insert: Inserting an Element into a List

The next function to discuss is `insert()`. What if, after removing Raymond from the preceding list, we decide we need to put him back where he was to begin with? An insertion function needs to know where to insert an element, so for now we will say that insertion occurs after a specified element already in the list. With this in mind, here is the definition of the `insert()` function:

```
function insert(element, after) {
   var insertPos = this.find(after);
   if (insertPos > -1) {
      this.dataStore.splice(insertPos+1, 0, element);
      ++this.listSize;
      return true;
   }
   return false;
}
```

`insert()` uses the helper function `find()` to determine the correct insertion position for the new element by finding the element specified in the `after` argument. Once this position is found, we use `shift()` to insert the new element into the list. Then we increment `listSize` by 1 and return `true` to indicate the insertion was successful.

## Clear: Removing All Elements from a List

Next, we need a function to clear out the elements of a list and allow new elements to be entered:

```
function clear() {
   delete this.dataStore;
   this.dataStore = [];
   this.listSize = this.pos = 0;
}
```

The `clear()` function uses the `delete` operator to delete the `dataStore` array, and the next line re-creates the empty array. The last line sets the values of `listSize` and `pos` to 0 to indicate the start of a new list.

## Contains: Determining if a Given Value Is in a List

The `contains()` function is useful when you want to check a list to see if a particular value is part of the list. Here is the definition:

```
function contains(element) {
   for (var i = 0; i < this.dataStore.length; ++i) {
      if (this.dataStore[i] == element) {
         return true;
      }
   }
   return false;
}
```

## Traversing a List

This final set of functions allows movement through a list, and the last function, `getElement()`, displays the current element in a list:

```
function front() {
   this.pos = 0;
}

function end() {
   this.pos = this.listSize-1;
}

function prev() {
   if (this.pos > 0) {
      --this.pos;
   }
}

function next() {
   if (this.pos < this.listSize-1) {
      ++this.pos;
   }
}

function currPos() {
   return this.pos;
}

function moveTo(position) {
   this.pos = position;
}

function getElement() {
   return this.dataStore[this.pos];
}
```

Let's create a new list of names to demonstrate how these functions work:

```
var names = new List();
names.append("Clayton");
names.append("Raymond");
names.append("Cynthia");
names.append("Jennifer");
names.append("Bryan");
names.append("Danny");
```

Now let's move to the first element of the list and display it:

```
names.front();
print(names.getElement()); // displays Clayton
```

Next, we move forward one element and display the element's value:

```
names.next();
print(names.getElement());  // displays Raymond
```

Now we'll move forward twice and backward once, displaying the current element to demonstrate how the `prev()` function works:

```
names.next();
names.next();
names.prev();
print(names.getElement()); // displays Cynthia
```

The behavior we've demonstrated in these past few code fragments is captured in the concept of an *iterator*. We explore iterators in the next section.

## Iterating Through a List

An iterator allows us to traverse a list without referencing the internal storage mechanism of the `List` class. The functions `front()`, `end()`, `prev()`, `next()`, and `currPos` provide an implementation of an iterator for our `List` class. Some advantages to using iterators over using array indexing include:

- Not having to worry about the underlying data storage structure when accessing list elements
- Being able to update the list and not having to update the iterator, where an index becomes invalid when a new element is added to the list
- Providing a uniform means of accessing elements for different types of data stores used in the implemenation of a `List` class

With these advantages in mind, here is how to use an iterator to traverse through a list:

```
for(names.front(); names.currPos() < names.length(); names.next()) {
   print(names.getElement());
}
```

The for loop starts by setting the current position to the front of the list. The loop continues while the value of currPos is less than the length of the list. Each time through the loop, the current position is moved one element forward through the use of the next() function.

We can also traverse a list backward using an iterator. Here is the code:

```
for(names.end(); names.currPos() >= 0; names.prev()) {
   print(names.getElement());
}
```

The loop starts at the last element of the list and moves backward using the prev() function while the current position is greater than or equal to 0.

Iterators are used only to move through a list and should not be combined with any functions for adding or removing items from a list.

## A List-Based Application

To demonstrate how to use lists, we are going to build a system that can be used in the simulation of a video-rental kiosk system such as Redbox.

### Reading Text Files

In order to get the list of videos available in the kiosk into our program, we need to be able to read the data from a file. We first have to create a text file that contains the list of videos available using a text editor. We name the file films.txt. Here are the contents of the files (these movies are the top 20 movies as voted on by IMDB users as of October 5, 2013):

1. *The Shawshank Redemption*
2. *The Godfather*
3. *The Godfather: Part II*
4. *Pulp Fiction*
5. *The Good, the Bad and the Ugly*
6. *12 Angry Men*
7. *Schindler's List*
8. *The Dark Knight*
9. *The Lord of the Rings: The Return of the King*
10. *Fight Club*
11. *Star Wars: Episode V - The Empire Strikes Back*
12. *One Flew Over the Cuckoo's Nest*

13. *The Lord of the Rings: The Fellowship of the Ring*
14. *Inception*
15. *Goodfellas*
16. *Star Wars*
17. *Seven Samurai*
18. *The Matrix*
19. *Forrest Gump*
20. *City of God*

Now we need a code fragment to read the contents of the file into our program:

```
var movies = read(films.txt).split("\n");
```

This line performs two tasks. First, it reads the contents of our movies text file into the program, `read(films.txt)`; and second, it splits the file into individual lines by using the newline character as a delimiter. This output is then stored as an array in the `movies` variable.

This line of code works up to a point, but it's not perfect. When the elements of the text file are split into the array, the newline character is replaced with a space. While a single space seems innocuous enough, having an extra space in a string can cause havoc when you are doing string comparisons. So we need to add a loop that strips the space from each array element using the `trim()` function. This code will work better in a function, so let's create a function to read data from a file and store it in an array:

```
function createArr(file) {
   var arr = read(file).split("\n");
   for (var i = 0; i < arr.length; ++i) {
      arr[i] = arr[i].trim();
   }
   return arr;
}
```

## Using Lists to Manage a Kiosk

The next step is to take the movies array and store its contents in a list. Here is how we do it:

```
var movieList = new List();
for (var i = 0; i < movies.length; ++i) {
   movieList.append(movies[i]);
}
```

Now we can write a function to display the movie list available at the kiosk:

```
function displayList(list) {
   for (list.front(); list.currPos() < list.length(); list.next()) {
      print(list.getElement());
   }
```

The `displayList()` function works fine with native types, such as lists made up of strings, but it won't work for `Customer` objects, which are defined below. Let's modify the function so that if it discovers that the list is made up of `Customer` objects, it will display those objects accordingly. Here's the new definition of `displayList()`:

```
function displayList(list) {
   for (list.front(); list.currPos() < list.length(); list.next()) {
      if (list.getElement() instanceof Customer) {
         print(list.getElement()["name"] + ", " +
               list.getElement()["movie"]);
      }
      else {
         print(list.getElement());
      }
   }
}
```

For each object in the list, we use the `instanceof` operator to test whether the object is a `Customer` object. If so, we retrieve the name and the movie the customer has checked out using each of the two properties as an index for retrieving the associated value. If the object is not a `Customer`, the code simply returns the element.

Now that we have our movie list taken care of, we need to create a list to store the customers who check out movies at the kiosk:

```
var customers = new List();
```

This will contain `Customer` objects, which are made up of the customer's name and the movie checked out. Here is the constructor function for the `Customer` object:

```
function Customer(name, movie) {
   this.name = name;
   this.movie = movie;
}
```

Next, we need a function that allows a customer to check out a movie. This function takes two arguments: the customer's name and the movie he wants to check out. If the movie is available, the function removes the movie from the kiosk's list of movies and adds it to the customer's list. We'll use the `List` class function `contains()` for this task.

Here is the definition for a function to check out a movie:

```
function checkOut(name, movie, filmList, customerList) {
   if (movieList.contains(movie)) {
      var c = new Customer(name, movie);
      customerList.append(c);
      filmList.remove(movie);
   }
   else {
      print(movie + " is not available.");
   }
}
```

The function first checks to see if the movie requested is available. If the movie is available, a `Customer` object is created with the movie's title and the customer's name. The `Customer` object is appended to the customer list, and the movie is removed from the movie list. If the movie is not available, a simple message is displayed indicating such.

We can test the `checkOut()` function with a short program:

```
var movies = createArr("films.txt");
var movieList = new List();
var customers = new List();
for (var i = 0; i < movies.length; ++i) {
   movieList.append(movies[i]);
}
print("Available movies: \n");
displayList(movieList);
checkOut("Jane Doe", "The Godfather", movieList, customers);
print("\nCustomer Rentals: \n");
displayList(customers);
```

The output of the program displays the movie list with `"The Godfather"` removed, followed by the list of customers with movies checked out.

Let's add some titles to our program's output to make it easier to read, along with some interactive input:

```
var movies = createArr("films.txt");
var movieList = new List();
var customers = new List();
for (var i = 0; i < movies.length; ++i) {
   movieList.append(movies[i]);
}
print("Available movies: \n");
displayList(movieList);
putstr("\nEnter your name: ");
var name = readline();
putstr("What movie would you like? ");
var movie = readline();
checkOut(name, movie, movieList, customers);
print("\nCustomer Rentals: \n");
displayList(customers);
print("\nMovies Now Available\n");
displayList(movieList);
```

Here is the result of running this program:

```
Available movies:
```

```
The Shawshank Redemption
The Godfather
The Godfather: Part II
Pulp Fiction
The Good, the Bad and the Ugly
12 Angry Men
Schindler's List
The Dark Knight
The Lord of the Rings: The Return of the King
Fight Club
Star Wars: Episode V - The Empire Strikes Back
One Flew Over the Cuckoo's Nest
The Lord of the Rings: The Fellowship of the Ring
Inception
Goodfellas
Star Wars
Seven Samurai
The Matrix
Forrest Gump
City of God

Enter your name: Jane Doe
What movie would you like? The Godfather

Customer Rentals:

Jane Doe, The Godfather

Movies Now Available

The Shawshank Redemption
The Godfather: Part II
Pulp Fiction
The Good, the Bad and the Ugly
12 Angry Men
Schindler's List
The Dark Knight
The Lord of the Rings: The Return of the King
Fight Club
Star Wars: Episode V - The Empire Strikes Back
One Flew Over the Cuckoo's Nest
The Lord of the Rings: The Fellowship of the Ring
Inception
Goodfellas
Star Wars
Seven Samurai
The Matrix
Forrest Gump
City of God
```

We can add other functionality to make our video-rental kiosk system more robust. You will get to explore some of this added functionality in the exercises that follow.

## Exercises

1. Write a function that inserts an element into a list only if the element to be inserted is larger than any of the elements currently in the list. Larger can mean either greater than when working with numeric values, or further down in the alphabet, when working with textual values.
2. Write a function that inserts an element into a list only if the element to be inserted is smaller than any of the elements currently in the list.
3. Create a `Person` class that stores a person's name and their gender. Create a list of at least 10 `Person` objects. Write a function that displays all the people in the list of the same gender.
4. Modify the video-rental kiosk program so that when a movie is checked out it is added to a list of rented movies. Display this list whenever a customer checks out a movie.
5. Create a check-in function for the video-rental kiosk program so that a returned movies is deleted from the rented movies list and is added back to the available movies list.