Abstract Data Types

Abstract Data Types

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection
- Data abstraction
  - Asks you to think *what* you can do to a collection of data independently of *how* you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of procedural abstraction

Abstract Data Types

- Abstract data type (ADT)
  - An ADT is composed of
    - A collection of data
    - A set of operations on that data
  - Specifications of an ADT indicate
    - What the ADT operations do, not how to implement them
  - Implementation of an ADT
    - Includes choosing a particular data structure
    - A data structure is a construct that can be defined in a programming language to store a collection of data

Abstract Data Types-Walls

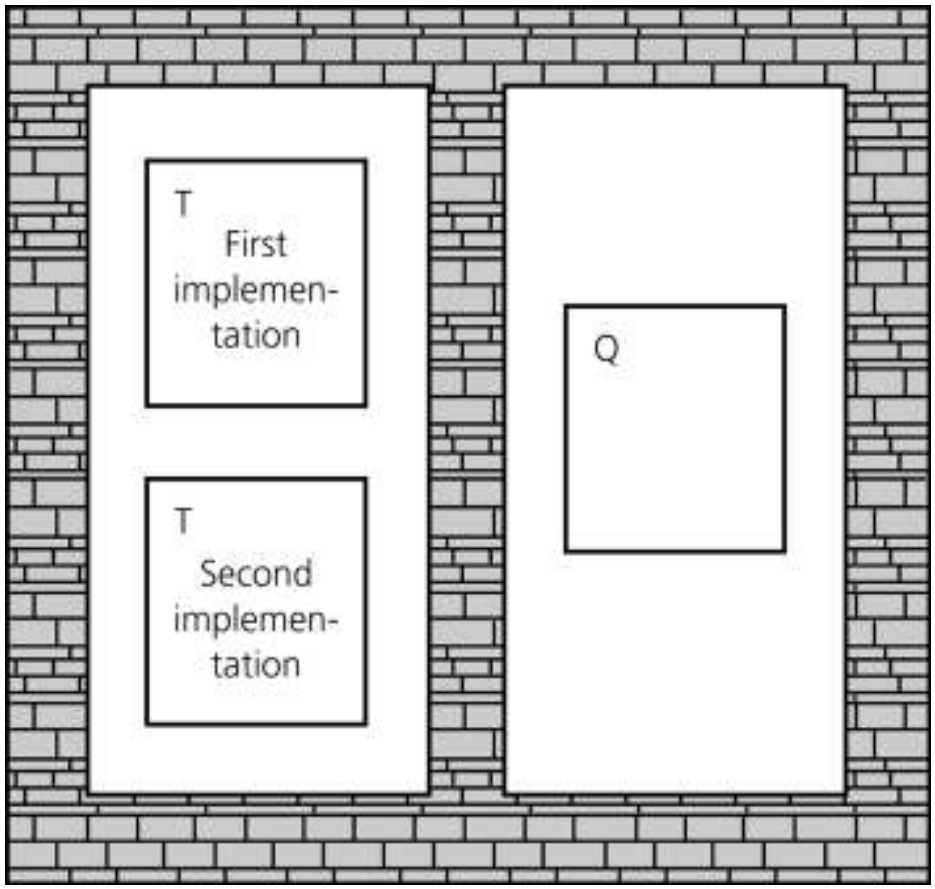

Figure 4-1

Isolated tasks: the implementation of task *T* does not affect task *Q*

Abstract Data Types

- The isolation of modules is not total
  - Methods' specifications, or contracts, govern how they interact with each other

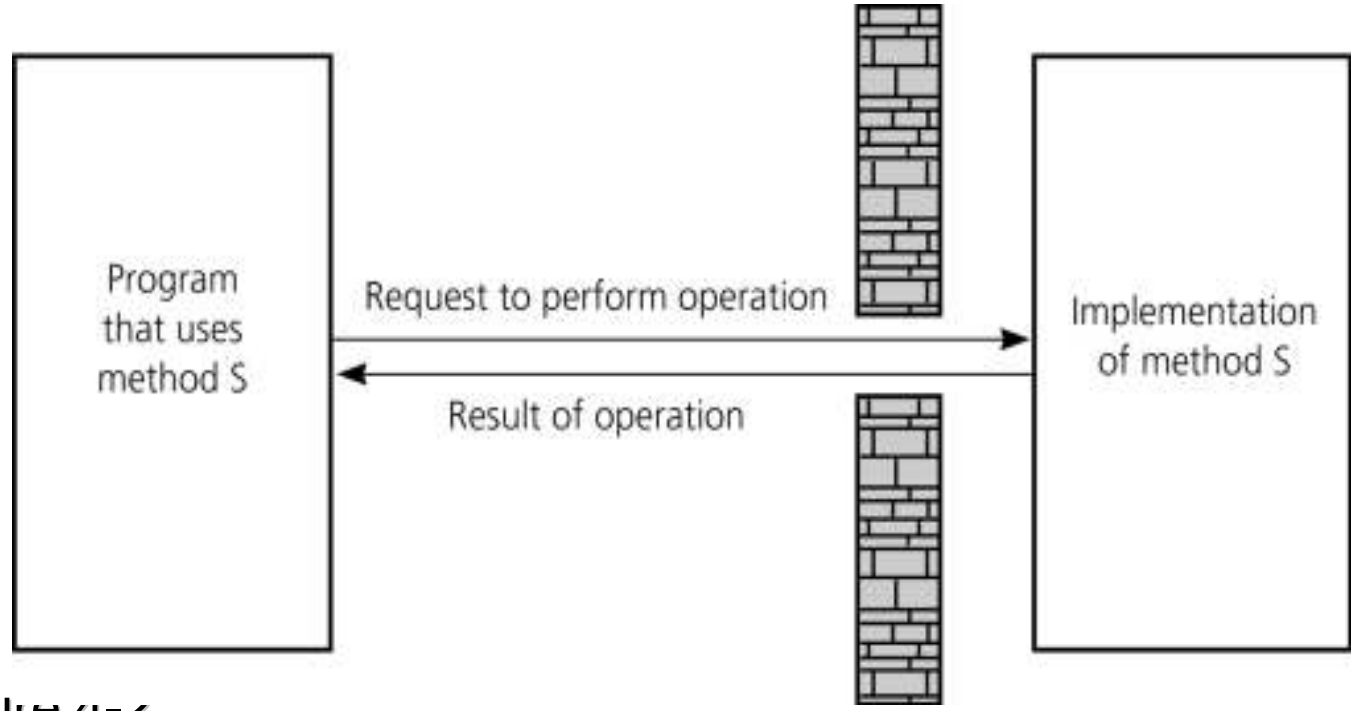

Figure 4-2

A slit in the wall

Specifying ADTs

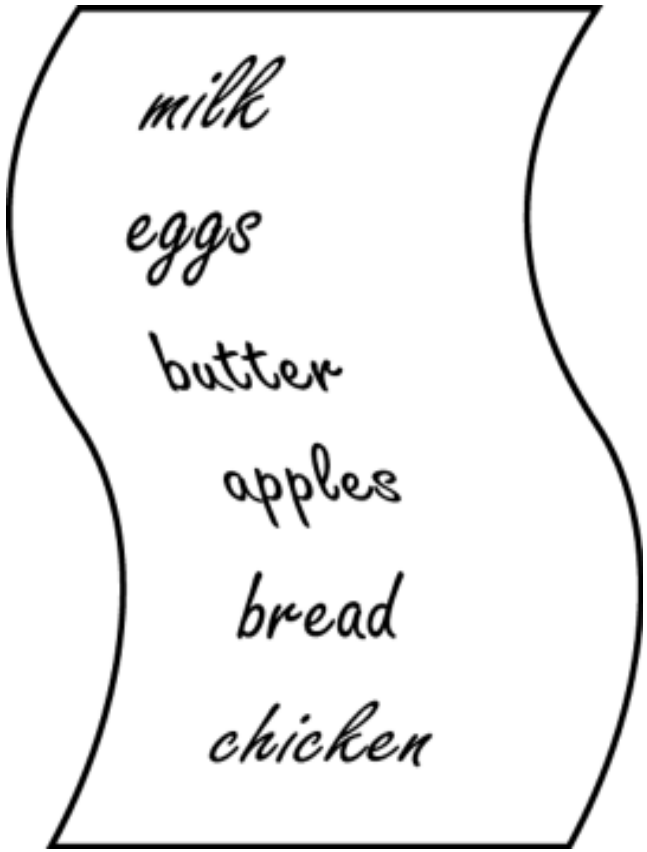

Figure 4-5

list A grocery

- In a list
  - Except for the first and last items, each item has
    - A unique predecessor
    - A unique successor
  - Head or front
    - Does not have a predecessor
  - Tail or end
    - Does not have a successor

The ADT List

- ADT List operations
  - Create an empty list
    - `aList.createList();`
  - Determine whether a list is empty
    - `aList.isEmpty();`
  - Determine the number of items in a list
    - `aList.size();`
  - Add an item at a given position in the list
    - `aList.add(index, item);`
  - Remove the item at a given position in the list
    - `aList.remove(index);`
  - Remove all the items from the list
    - `aList.removeAll();`
  - Retrieve (get) the item at a given position in the list
    - `aList.get(index);`
- Items are referenced by their position within the list

Pseudocode for the ADT List Operations

```
+createList()
// Creates an empty list.

+isEmpty():boolean {query}
// Determines whether a list is empty.

+size():integer {query}
// Returns the number of items that are in a list.

+add(in index:integer, in item:ListItemType)
// Inserts item at position index of a list, if
// 1 <= index <= size()+1.
// If index <= size(), items are renumbered as
// follows: The item at index becomes the item at
// index+1, the item at index+1 becomes the
// item at index+2, and so on.
// Throws an exception when index is out of range or if
// the item cannot be placed on the list (list full).

+remove(in index:integer)
// Removes the item at position index of a list, if
// 1 <= index <= size(). If index < size(), items are
// renumbered as follows: The item at index+1 becomes
// the item at index, the item at index+2 becomes the
// item at index+1, and so on.
// Throws an exception when index is out of range or if
// the list is empty.

+removeAll()
// Removes all the items in the list.

+get(index):ListItemType {query}
// Returns the item at position index of a list
// if 1 <= index <= size(). The list is
// left unchanged by this operation.
// Throws an exception if index is out of range.
```

The ADT List

- Specifications of the ADT operations
  - Define the contract for the ADT list
  - Do not specify how to store the list or how to perform the operations
    - If you request these operations be performed, this is what will happen.
- ADT operations can be used in an application without the knowledge of how the operations will be implemented

Using an ADT.

The following methods manipulate the data in the list without knowing the implementation details of the List.

```
displayList(in aList:List){
        for(index=1 through aList.size()){
                dataItem=aList.get(index);
                Print dataItem;
        }
}
replace(in aList:List, in i:int, in
        newItem:ListItemType){
        if(i>=1 and i<=aList.size()){
                aList.remove(i);
                aList.insert(i, newItem);
        }
}
```

The ADT List

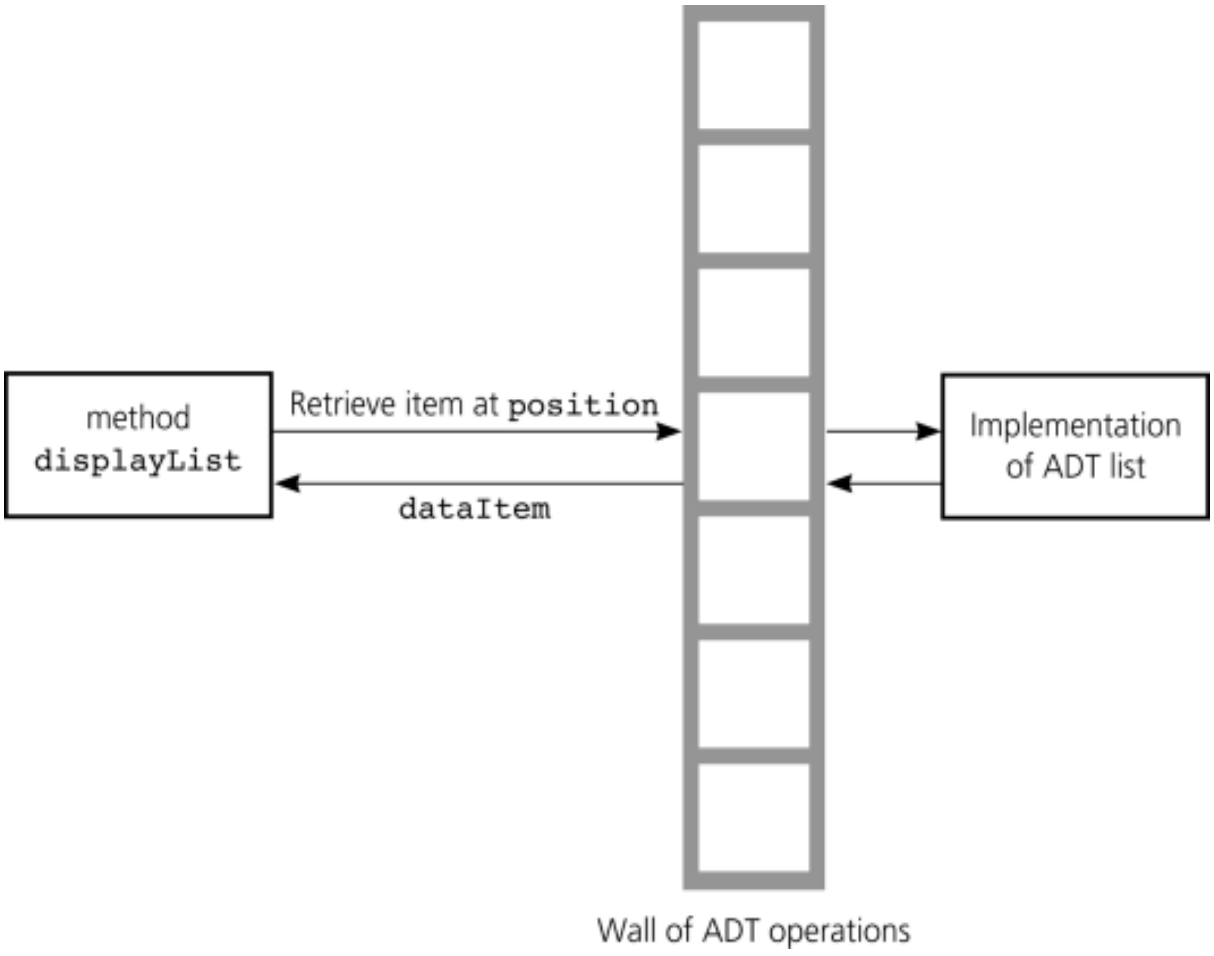

Figure 4-7

The wall between ***displayList*** and the implementation of the ADT list

The ADT Sorted List

- The ADT sorted list
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions

Pseudocode for the ADT Sorted List Operations

```
+createSortedList()
// Creates an empty sorted list.

+sortedIsEmpty():boolean {query}
// Determines whether a sorted list is empty.

+sortedSize():integer {query}
// Returns the number of items that are in a sorted list.

+sortedAdd(in item:ListItemType)
// Inserts item into its proper sorted position in a
// sorted list. Throws an exception if the item
// cannot be placed on the list (list full).

+sortedRemove(in item:ListItemType)
// Deletes item from a sorted list.
// Throws an exception if the item is not found.

+sortedGet(in index:integer)
// Returns the item at position index of a
// sorted list, if 1 <= index <= sortedSize().
// The list is left unchanged by this operation.
// Throws an exception if the index is out of range.

+locateIndex(in item:ListItemType):integer {query}
// Returns the position where item belongs or
// exists in a sorted list; item and the list are
// unchanged.
```

Implementing ADTs

- Choosing the data structure to represent the ADT's data is a part of implementation
  - Choice of a data structure depends on
    - Details of the ADT's operations
    - Context in which the operations will be used
- Implementation details should be hidden behind a wall of ADT operations
  - A program would only be able to access the data structure using the ADT operations

Implementing ADTs

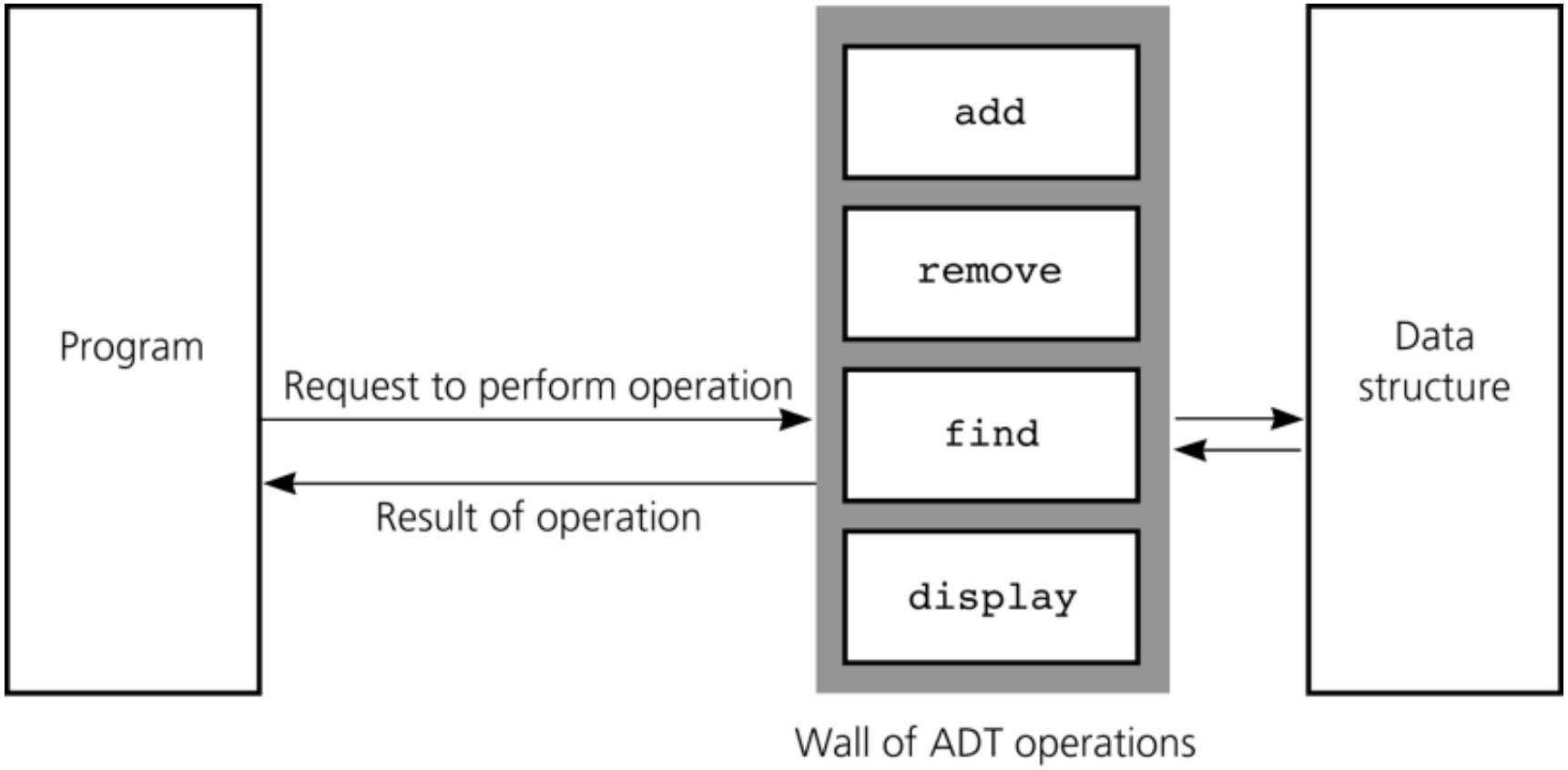

Figure 4-8

ADT operations provide access to a data structure

Implementing ADTs

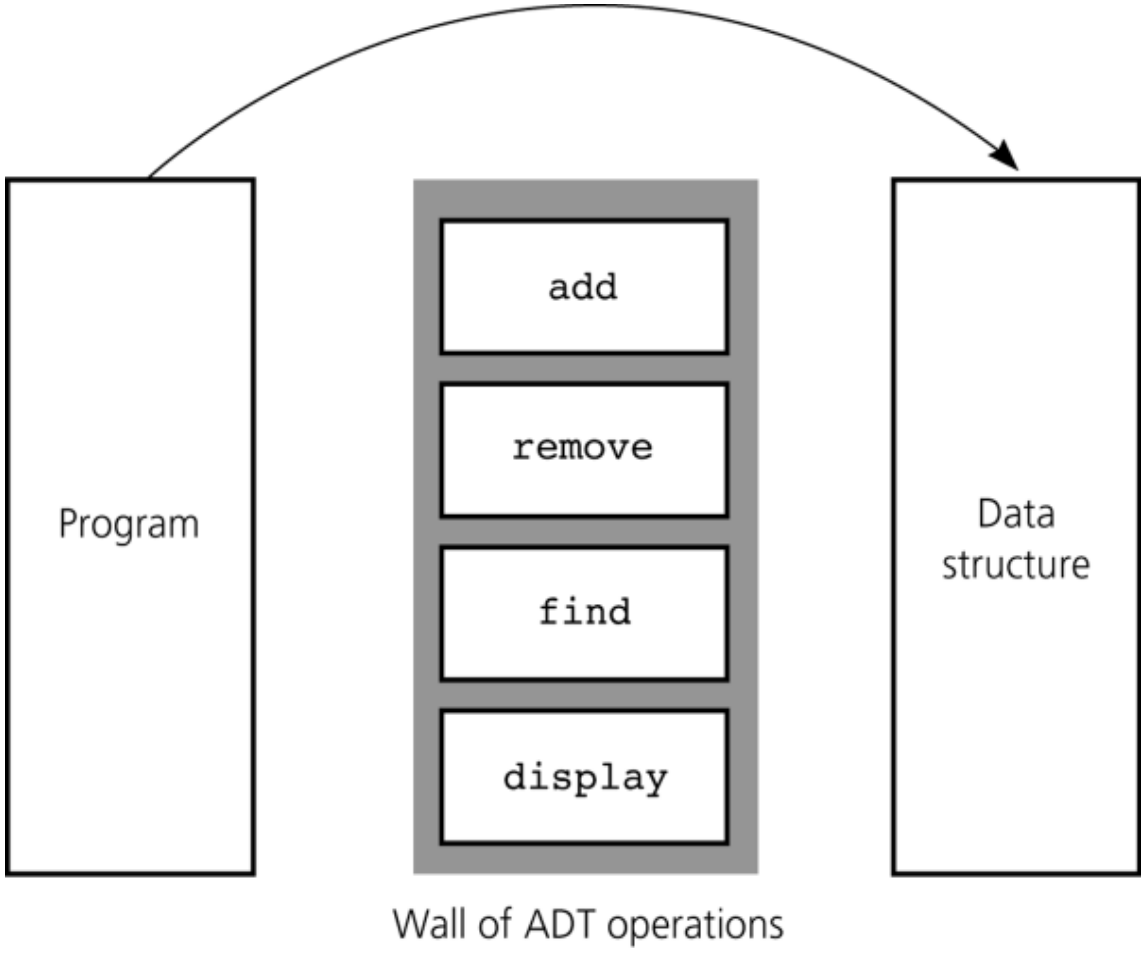

Figure 4-9

Violating the wall of ADT operations

An Array-Based Implementation of the ADT List

- An array-based implementation
  - A list's items are stored in an array `items`
  - A natural choice
    - Both an array and a list identify their items by number
  - A list's k^{th} item will be stored in `items[k-1]`

An Array-Based Implementation of the ADT List

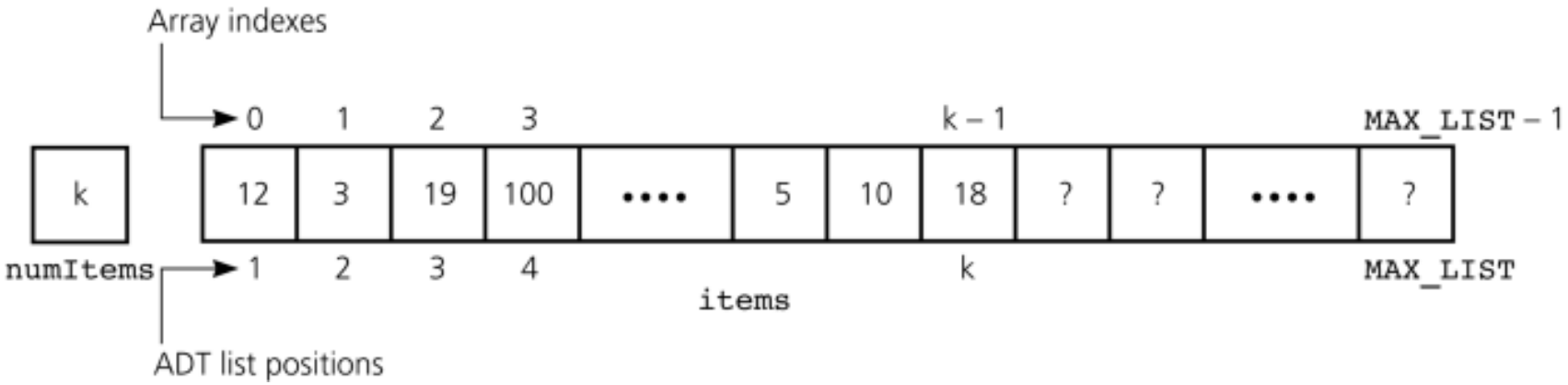

Figure 4-11

An array-based implementation of the ADT list

Create an Interface for the List ADT

```
public interface ListInterface{
        public boolean isEmpty();
        public int size();
        public void add(int index, Object item) throws
        ListException;
        public Object get(int index) throws ListException;
        public void remove(int index) throws ListException;
        public void removeAll();
}
```

Array-Based implementation.

```
public class ListArrayBased implements ListInterface{
        private static final int MAX_LIST = 50;
        private Object items[];
        private int numItems;

        public ListArrayBased(){ //act as the createList()
                items = new Objects[MAX_LIST];
                numItems = 0;
        }
        public boolean isEmpty(){
                return (numItems == 0);
        }
        public int size() {return numItems;}
        public void add(int index, Object item) throws
        ListException {[…]}
        public Object get(int index) throws ListException {[…]}
        public void remove(int index) throws ListException{[…]};
        public void removeAll(){[…]}
}
```

Using the List ADT

```
Static public void main(){
        […]
        ListArrayBased groceryList = new ListArrayBased();
        groceryList.add(1, "Milk");
        […]
        String item=groceryList.get(1);

        //violates the terms of abstraction.
        String item=groceryList.items[0]; //illegal statement.

        […]
}
```

Java Exceptions (review)

- Exception
  - A mechanism for handling an error during execution
  - A method indicates that an error has occurred by throwing an exception
  - When a method ***throws*** an exception the control of the program returns to the point at which the method is called.
  - At this point you must ***catch*** the exception an deal with it.

Java Exceptions –catching exception

- Java provides `try-catch` blocks to handle exceptions
- Syntax:

```
try{
    statements;
}
catch(exceptionClass id){
    statements;
}
```

- `try` block
  - A statement that might throw an exception is placed within a `try` block

Java Exceptions

- Catching exceptions (Continued)
  - `catch` block
    - Used to catch an exception and deal with the error condition
    - Syntax

```
catch (exceptionClass identifier) {
   statement(s);
} // end catch
```

```
try {
        int result = 99/0; //this statement throws an exception
        //other statements may appear here;
}catch (ArithmetichException e){
        System.out.println("ArithmetichException caught");
}
```

- When a statement throws an exception it returns an *object* that must be caught by the ***catch*** block.
- When a statement in a try block ***throws*** an exception the remainder of the try block is abandoned, and the control is passed to the catch block that corresponds to the type of exception thrown.

Some exceptions must be handled.

```
public static void getInput(String filename){
        FileInputStream fis;
        fis = new FileInputStream(filename);
        […]
}
```

- The **new** `FileInputStream` statement throws an exception if file is not found.
- In the above example this exception is not handled and therefore it won't get compiled.

```
public static void getInput(String filename){
        FileInputStream fis;
        try{
                fis = new FileInputStream(filename);
        }catch(FileNotFoundException e){
                System.out.println("The file"+filename+"Not found");
        }
        […]
}
```

Java Exceptions

- Types of exceptions
  - Checked exceptions
    - Instances of classes that are subclasses of the `java.lang.Exception` class
    - Must be handled locally
    - Used in situations where the method has encountered a serious problem

Java Exceptions

- Types of exceptions (Continued)
  - Runtime exceptions
    - Used in situations where the error is not considered as serious
    - Can often be prevented by fail-safe programming
    - Instances of classes that are subclasses of the `RuntimeException` class
    - Are not required to be caught locally

Java Exceptions

- Throwing exceptions
  - A `throw` statement is used to throw an exception

```
throw new exceptionClass (stringArgument);
```

- Defining a new exception class
  - A programmer can define a new exception class

```
class MyException extends Exception {
  public MyException(String s) {
    super(s);
  } // end constructor
}  // end MyException

public void myMethod() throws MyException{
          //some code here.
          if( some condition){
                    throw new MyException("MyException: reason");
          }
}
public void yourMethod(){
          try{
                    myMethod();
          }
          catch (MyException e){
                    //code to handle the exception
          }
}
```

Finally block

- The `finally` block
  - Executed whether or not an exception is thrown
  - Can be used even if no `catch` block is used
  - Syntax

```
finally {
   statement(s);
}
```