Abstract Data Types

- Modularity
  - Keeps the complexity of a large program manageable by systematically controlling the interaction of its components
  - Isolates errors
  - Eliminates redundancies
    - DRY (Don't Repeat Yourself)
  - A modular program is
    - Easier to write
    - Easier to read
    - Easier to modify

© 2006 Pearson Addison-Wesley. All rights reserved

Abstract Data Types

- Procedural abstraction
  - Separates the purpose and use of a module from its implementation
  - A module's specification should
    - Detail how the module behaves
    - Identify details that can be hidden within the module
- Information hiding
  - Hides certain implementation details within a module
  - Makes these details inaccessible from outside the module

© 2006 Pearson Addison-Wesley. All rights reserved

Abstract Data Types

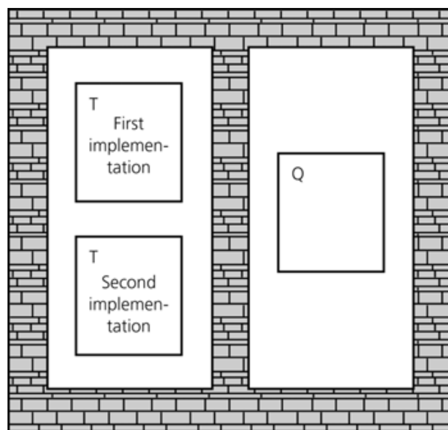

Figure 4-1

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

© 2006 Pearson Addison-Wesley. All rights reserved

Abstract Data Types

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

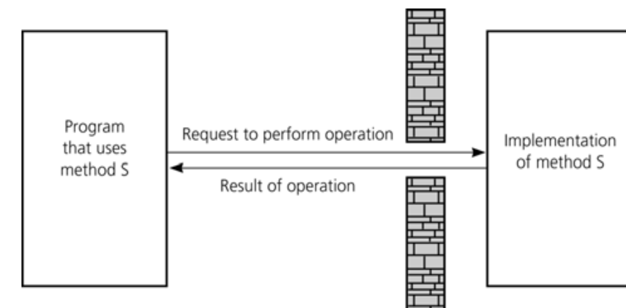

Figure 4-2

A slit in the wall

© 2006 Pearson Addison-Wesley. All rights reserved

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

© 2006 Pearson Addison-Wesley. All rights reserved

Abstract Data Types

- Data structure
  - A construct that is defined within a programming language to store a collection of data
  - Example: arrays
- ADTs and data structures are not the same
- Data abstraction
  - Results in a wall of ADT operations between data structures and the program that accesses the data within these data structures

© 2006 Pearson Addison-Wesley. All rights reserved

Abstract Data Types

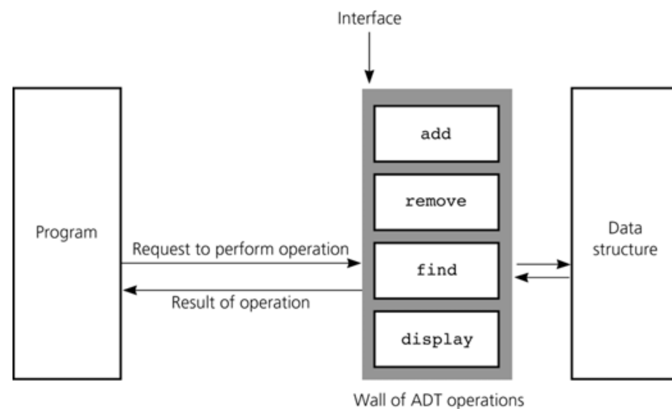

Figure 4-4

A wall of ADT operations isolates a data structure from the program that uses it

© 2006 Pearson Addison-Wesley. All rights reserved

Specifying ADTs

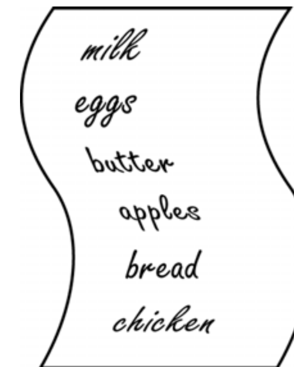

Figure 4-5

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

© 2006 Pearson Addison-Wesley. All rights reserved

The ADT List

- ADT List operations
  - Create an empty list
  - Determine whether a list is empty
  - Determine the number of items in a list
  - Add an item at a given position in the list
  - Remove the item at a given position in the list
  - Remove all the items from the list
  - Retrieve (get) the item at a given position in the list
- Items are referenced by their position within the list

© 2006 Pearson Addison-Wesley. All rights reserved

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
- ADT operations can be used in an application without the knowledge of how the operations will be implemented

© 2006 Pearson Addison-Wesley. All rights reserved

The ADT List

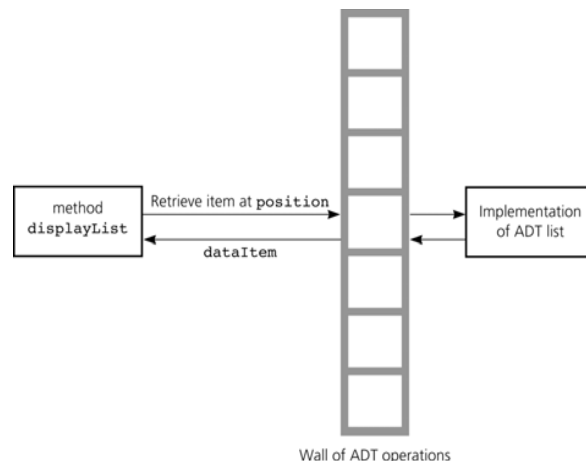

Figure 4-7

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

© 2006 Pearson Addison-Wesley. All rights reserved

Designing an ADT

- The design of an ADT should evolve naturally during the problem-solving process
- Questions to ask when designing an ADT
  - What data does the problem require?
  - What operations does the problem require?

© 2006 Pearson Addison-Wesley. All rights reserved

Axioms

- For complex abstract data types, the behavior of the operations must be specified using axioms
  - Axiom: A mathematical rule
- In future courses (e.g. 220 and 230 you can use Test classes to ensure the axioms are implemented correctly)

© 2006 Pearson Addison-Wesley. All rights reserved

Axioms

- Axioms for the ADT List
  - (aList.createList()).size() = 0
  - (aList.add(i, x)).size() = aList.size() + 1
  - (aList.remove(i)).size() = aList.size() – 1
  - (aList.createList()).isEmpty() = ?
  - (aList.add(i, item)).isEmpty() = ?
  - (aList.createList()).remove(i) = ?
  - (aList.add(i, x)).remove(i) = ?
  - (aList.createList()).get(i) = ?
  - (aList.add(i, x)).get(i) = ?
  - aList.get(i) = (aList.add(i, x)).get(i+1)
  - aList.get(i+1) = (aList.remove(i)).get(i)

© 2006 Pearson Addison-Wesley. All rights reserved

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

© 2006 Pearson Addison-Wesley. All rights reserved

Implementing ADTs

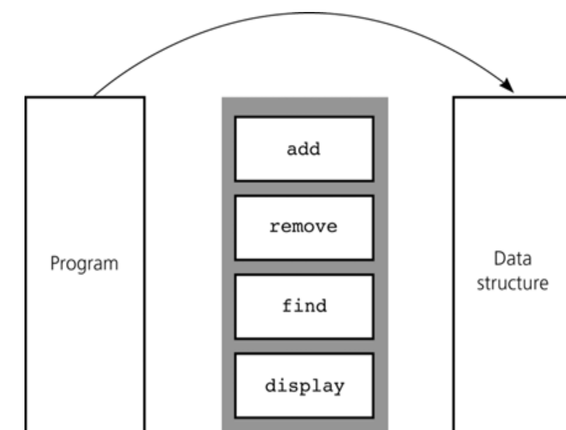

Figure 4-9

Violating the wall of ADT operations

© 2006 Pearson Addison-Wesley. All rights reserved

Java Classes Revisited

- Object-oriented programming (OOP) views a program as a collection of objects
- Encapsulation
  - A principle of OOP
  - Can be used to enforce the walls of an ADT
  - Combines an ADT's data with its method to form an object
  - Hides the implementation details of the ADT from the programmer who uses it

© 2006 Pearson Addison-Wesley. All rights reserved

Java Classes Revisited

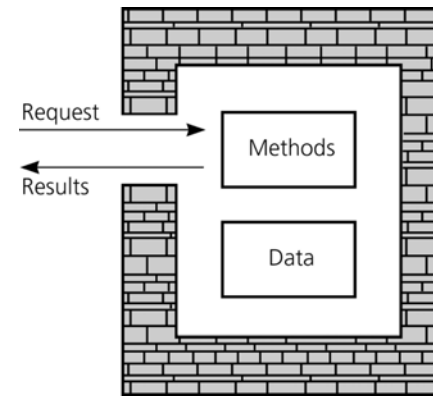

Figure 4-10
An object's data and methods are encapsulated

© 2006 Pearson Addison-Wesley. All rights reserved

Java Classes Revisited

- A Java class
  - Class instances
    - Data fields
      - Should almost always be private
    - Methods
      - Should only be public if part of the external contract (specification)

© 2006 Pearson Addison-Wesley. All rights reserved

Java Interfaces

- An interface
  - Specifies methods and constants, but supplies no implementation details
  - Can be used to specify some desired common behavior that may be useful over many different types of objects
  - This means an interface is perfect to provide the Wall between an implementation and its users

© 2006 Pearson Addison-Wesley. All rights reserved

Java Interfaces

- A class that implements an interface must
  - Include an `implements` clause
  - Provide implementations of the methods of the interface
- To define an interface
  - Use the keyword `interface` instead of `class` in the header
  - Provide only method specifications and constants in the interface definition

© 2006 Pearson Addison-Wesley. All rights reserved

Example Interface

```
public interface OrderedData {

    public void add(int item);

    public void displayData();

}
```

- Any class which implements this must define these two methods.

© 2006 Pearson Addison-Wesley. All rights reserved

Example Class implementing the interface

```
public class OrderedOnDisplay implements OrderedData {

    private int[] data = new int[10];
    private int size = 0;

    public void add(int item) {
        if (size == data.length) // grow the array
            data = Arrays.copyOf(data, data.length * 2);
        data[size] = item;
        ++size;
    }

    public void displayData() {
        Arrays.sort(data, 0, size);
        System.out.print("[");
        for (int i = 0; i < size; ++i) {
            System.out.print(data[i]);
            if (i < size - 1)
                System.out.print(", ");
        }
        System.out.println("]");
    }
}
```

© 2006 Pearson Addison-Wesley. All rights reserved

Another class implementing the interface

```
public class AlwaysOrdered implements OrderedData {

    private int[] data = new int[10];
    private int size = 0;

    public void add(int item) {
        int position = 0;
        while (position < size && item > data[position]) {
            ++position;
        }
        ++size;
        if (size > data.length) // grow the array
            data = Arrays.copyOf(data, data.length * 2);
        if (position < size - 1) // shuffle data up to make room
            for (int i = size - 1; i > position; --i)
                data[i] = data[i - 1];
        data[position] = item;
    }

    public void displayData() {
    // the same as the previous slide without the call to sort
    }
}
```

© 2006 Pearson Addison-Wesley. All rights reserved

Class using the interface

```
public class UseOrderedData {

   public static void main(String[] args) {
      OrderedData data = new OrderedOnDisplay();
        // new AlwaysOrdered();
      int number = (int)(Math.random() * 20 + 1);
      for (int i = 0; i < number; ++i)
         data.add((int)(Math.random() * 100));
      data.displayData();
   }

}
```

© 2006 Pearson Addison-Wesley. All rights reserved

The Java Collections Framework `List` interface

- http://www.cs.auckland.ac.nz/references/java/java1.5/api/java/util/List.html
- Determine whether a list is empty
  `isEmpty()`
- Determine the number of items in a list
  `size()`
- Add an item at a given position in the list
  `set(index, element)`
- Remove the item at a given position in the list
  `remove(index)`
- Remove all the items from the list
  `clear()`
- Retrieve (get) the item at a given position in the list
  `get(index)`

© 2006 Pearson Addison-Wesley. All rights reserved

The ArrayList implementation

- The `ArrayList` class provides a growable array. It implements the `List` interface.
- Very useful when you don't know the size of an array.

```
import java.util.*;
List testScores = new ArrayList();
testScores.add(score);
```

See ArrayListExample.java.

© 2006 Pearson Addison-Wesley. All rights reserved

But Java is fussy about types

- The previous code generates warnings.
- Java wants you to specify the type of object stored in the ArrayList.
- This is done by specifying the type like a parameter. Such classes are called generic classes (they work with different types).

© 2006 Pearson Addison-Wesley. All rights reserved

List example with Generic type

```
public static void main(String[] args) {
   List<Integer> testScores = new
      ArrayList<Integer>();
   for (int random = 0; random < 10; random =
     (int)(Math.random() * 10) + 1) {
      int score = (int) (Math.random() * 100);
      testScores.add(score);
   }
   System.out.println(testScores);
}
```

- Java no longer complains.
- All elements of testScores are Integers.
- Can anyone see that some magic is happening here?

© 2006 Pearson Addison-Wesley. All rights reserved

List iterators

- Iterators are objects that produce successive elements in Java Collections
- The `hasNext()` method returns true if the collection has any more items
- The `next()` method returns the next item

```
for (Iterator<Integer> iterator =
  testScores.iterator();
  iterator.hasNext();) {
     int element = iterator.next();
     System.out.println(element);
}
```

© 2006 Pearson Addison-Wesley. All rights reserved