CSC115 Lecture 4

LillAnne Jackson

Intro to Abstract Data Types

Chapter 4: Data Abstraction: The Walls

Recall Program Modularity

- Modularity
  - Keeps the complexity of a large program manageable by systematically controlling the interaction of its components
  - Isolates errors
  - Eliminates redundancies
  - A modular program is
    - Easier to write
    - Easier to read
    - Easier to modify

Procedural Abstraction

- Procedural abstraction
  - Separates the purpose and use of a module from its implementation
  - A module's specifications 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

Abstract Data Types

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

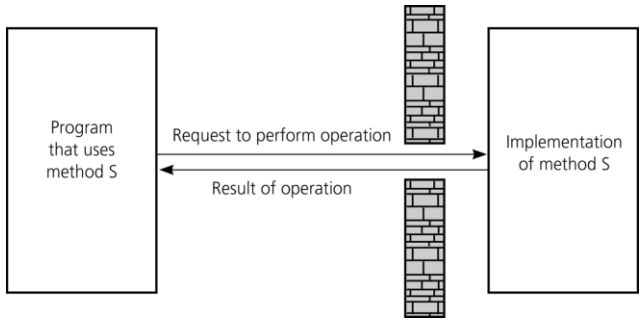


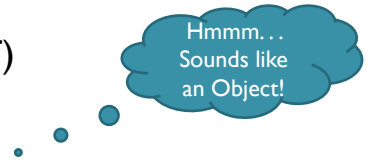
Figure 4-2

A slit in the wall

Data Abstraction

- 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

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

Abstract Data Types

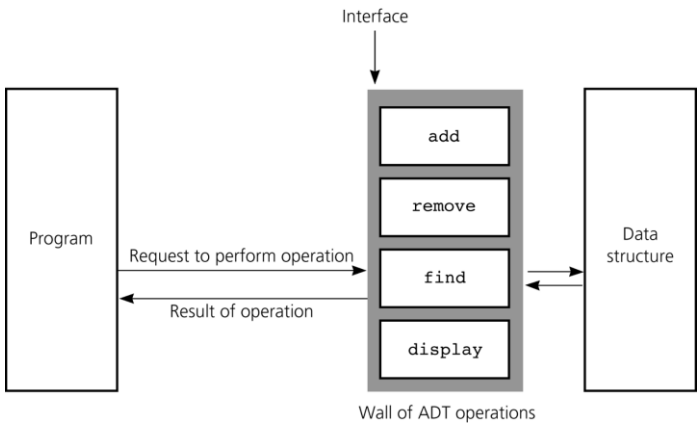


Figure 4-4

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

An ADT Example: A List

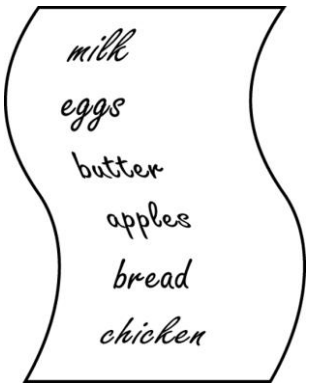


Figure 4-5

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
  - 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

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

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]`

- **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**

An Array-Based Implementation of the ADT List

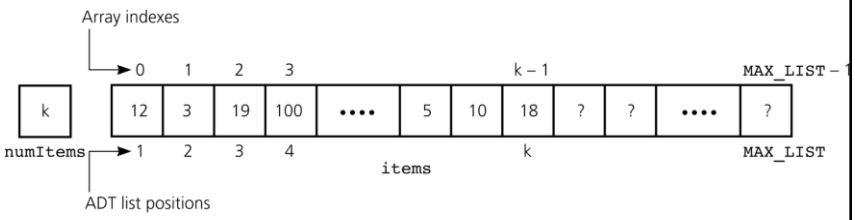


Figure 4-11

An array-based implementation of the ADT list

A Simpler Example: The ADT IntegerList

- The ADT simplified: IntegerList
  - adds items to the beginning or end of the list
  - determines the size of the list, and
  - determines the location of a specified item in the list
- The Java structure for an ADT: an *interface*

```
public interface IntegerList
{
  public void addFront (int x);
  public void addBack (int x);
  public int size();
  public int  get (int pos);
  public String toString();
}
```

We will implement this ADT in class

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 a problem require?
  - What operations does a problem require?

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

Summary

- A client should only be able to access the data structure by using the ADT operations
- An object encapsulates both data and operations on that data
  - In Java, objects are instances of a class, which is a programmer-defined data type
- A Java class contains at least one constructor, which is an initialization method
- Typically, you should make the data fields of a class private and provide public methods to access some or all of the data fields

