Abstraction

- Modularity in software design and programming
  - Smaller pieces are easier to understand
  - Helps manage complexity of large software
  - Promotes reuse and reflects algorithm decomposition
- Abstract Data Types
  - Cornerstone of software design
  - Information hiding by layers of abstraction
  - Abstractions are the walls of design
  - Procedural abstractions hide details of algorithms
  - Data abstractions hide details of data design

Abstract Data Types

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

An Abstract Data Type (ADT) is
- a collection of data
- a set of operations to perform on the data

Describes what can be done with information
- But not how it is done
- An implementation specifies details of operations
- Many possible implementations for one ADT
ADTs versus Data Structures

- Concrete data structures are defined by a programming language, e.g.,
  - Arrays of integers
  - A class with an array and an integer data members
  - Nested class definitions
- ADTs are not data structures
  - Independent of choice of data used for implementation as well as programming logic
  - ADTs correspond to interfaces or abstract classes in Java, not concrete classes
  - The ADT provides a wall that hides the details of the actual data structure and method implementations

Abstract Data Types

Figure 4.4
A wall of ADT operations isolates a data structure from the program that uses it
Specifying an ADT

- An ADT specification describes the abstraction
- For example, a list might be described as
  - A collection of items with operations
  - The operations are:
    - Determine if the list is empty
    - Determine how many items are in the list
    - Determine if a particular item is in the list
    - Remove a particular item from the list
    - Add a new item to the list

Specifying an ADT in Java

- Operations on the ADT are methods
- A Java interface lists the methods
  - But does not specify bodies of methods
  - A concrete class implements the interface
- Alternatively, an abstract Java class can specify method headers without bodies
  - Can be used as partial implementation
  - Methods without bodies are abstract methods
  - A concrete class extends the abstract class
- May use one or both techniques
List ADT in Java

- List as interface
  ```java
  public interface ListI {
      public int length();
      public boolean isEmpty();
      public boolean isMember(int n);
      public void remove(int n);
      public void add(int n);
  }
  ```

- List as abstract class
  ```java
  public abstract class List implements ListI {
      abstract public int length();
      abstract public boolean isEmpty();
      abstract public boolean isMember(int n);
      abstract public void remove(int n);
      abstract public void add(int n);
  }
  ```

Implementation of a List in Java

```java
public class IntList extends List {
    private int[] data;
    public IntList() { data = null; }
    public int length() { return data==null ? 0 : data.length; }
    public boolean isEmpty() { return length() == 0; }
    public boolean isMember(int n) {
        if (data == null) return false;
        for (int i = 0; i < data.length; ++i)
            if (n == data[i]) return true;
        return false;
    }
    public void remove(int n) { . . . }
    public void add(int n) { . . . }
}
```
List ADT

- All details of implementation are hidden
  - Interface ListI has no data or method bodies
  - Abstract class List has no data or method bodies
  - But class object can be used when it is only known as the interface or abstract class

```java
public void foo(ListI list) { . . . }
public void bar(List list) { . . . }

ListI list1 = new IntList();
foo(list1);
List list2 = new IntList();
bar(list2);
```

Changing Implementations

- Change to use linked list instead of array
  - Interface and abstract class are not affected
  - Code using ListI and List is not affected
  - Only implementation in concrete class is changed

```java
class IntLinkedList extends List {
    private class IntItem {
        int value; IntItem next;
        IntItem(int v) { value = v; next = null; }
    }
    private IntItem first, last;
    private int length;
    public IntLinkedList() { first = last = null; length = 0; }
    public int length() { return length; }
    public boolean isEmpty() { return length() == 0; }
}
```
Linked List Implementation

public boolean isMember(int v) {
    for (IntItem n = first; n != null; n = n.next)
        if (v == n.value) return true;
    return false;
}

public void remove(int v) {
    IntItem prev = null, cur;
    for (cur = first; cur != null; prev=cur, cur=cur.next)
        if (v == cur.value) break;
    if (cur != null) {
        if (prev == null) first = cur.next;
        else prev.next = cur.next;
        if (cur == last) last = prev;
        --length;
    }
}

public void add(int v) {
    IntItem newv = new IntItem(v);
    if (first == null)
        first = last = newv;
    else {
        last.next = newv;
        last = newv;
    }
    ++length;
}

public int get(int index) throws NotFoundException {
    int n = 0;
    for (IntItem i = first; i != null; i = i.next, ++n)
        if (index == n) return i.value;
    throw new NotFoundException();
}
More Abstractions

- Example was a list of integers, but what if we want a list of something else?
  - Replace "int" by "Object" in appropriate places
  - Replace "==" by ".equals()" for object comparisons
  - May need messy type casting
    ```java
    public void add(Object o);
    public Object get(int index);
    if (o.equals(i.value))
    ```

- Strict type checking is lost

Java Generics

- New feature in Java 1.5
  - Class, interface definitions may be parameterized by a type
  - E.g., List of <something>

```java
interface ListI<T> {
    public int length();
    public boolean isEmpty();
    public boolean isMember(T o);
    public void remove(T o);
    public void add(T o);
    public T get(int index) throws NotFoundException;
}
```
Java Generics

- Placeholder is used for a type, e.g., "T"
  - Compiler checks consistent use
  - Explicit type used for instantiation

```java
List<Integer> list1 = new LinkedList<Integer>();
foo(list1);
List<String> list2 = new LinkedList<String>();
bar(list2);
```

```java
public static void foo(List<Integer> list) { . . .
public static void bar(List<String> list) { . . .
```

Java Generics

- Type placeholder use in implementation:

```java
class LinkedList<T> extends List<T> {
  private class ListItem<T> {
    T value;
    ListItem<T> next;
    ListItem(T o) { value = o; }
  }
  private ListItem<T> first, last;
  private int length;
  public T get(int index) throws NotFoundException {
    int n = 0;
    for (ListItem<T> i = first; i != null; i = i.next, ++n)
      if (index == n) return i.value;
    throw new NotFoundException();
  }
}
```

Singly Linked List Complexity

- Length maintained as data
  - length(), isEmpty() are O(1)
- Front and back of list maintained
  - add() is O(1)
  - Array implementation is O(n) since re-allocation and copying is required
- isMember(), get() are O(n) in worst case
- Linked items allow easy removal of items
  - remove() is O(n) in worst case

Other Operations

- Insert an item at the beginning of the list
  - Complexity O(1)

```java
public void prepend(T o) {
    ListItem<T> newv = new ListItem<T>(o);
    if (first == null)
        first = last = newv;
    else {
        newv.next = first;
        first = newv;
    }
    ++length;
}
```
Other Operations

- Insert an item at a specific location
  - Complexity $O(n)$

```java
public void add(int index, T o) {
    if (index < 0 || index > length) throw new RangeException();
    if (index == length) add(o);
    else if (index == 0) prepend(o);
    else {
        ListItem<T> newv = new ListItem<T>(o);
        int n = 0;
        ListItem<T> cur;
        for (cur = first; n < index-1; cur=cur.next, ++n);
        newv.next = cur.next;
        cur.next = newv;
        ++length;
    }
}
```

Other Operations

- Convert to String (internal, as member)

```java
public String toString() {
    String result = "[ ";
    for (ListItem<T> i = first; i != null; i = i.next)
        result += i.value + " ";
    result += "]";
    return result;
}
```

- Convert to String (external, from outside of class)

```java
public static String listToString(List<String> list) {
    String result = "[ ";
    for (int i = 0; i < list.length(); ++i)
        result += list.get(i) + " ";
    result += "]";
    return result;
}
```
Iterators

- A common pattern is to use an integer to control the iteration over an array of items
- Abstract this concept to the general idea of an iterator
  - An iterator is used to step through the elements in a list in some order
- An iterator needs the ability to
  - Determine if there is another item: hasNext()
  - Provide the next item: next()
    - And advance iterator
  - Delete the retrieved item from the list: remove()
- Java defines such an interface Iterator in java.util

Iterator for Linked Lists

- Hide the details of how iteration is done
- Use an inner class of the list to allow for efficient implementation
- Interface for list provides access to the iterator

```java
interface ListI<T> {
  . . .
  public Iterator<T> listIterator();
}
```

- Example of using the iterator:

```java
List<String> list = new LinkedList<String>();
 . . .
Iterator<String> iter = list.listIterator();
while (iter.hasNext())
    System.out.print(iter.next() + ", ");
```
A possible implementation for our LinkedList

class LinkedList<T> extends List<T> {
  
  // Inner iterator class
  private class LinkedListIterator implements Iterator<T> {
    private ListItem<T> prevItem;
    private ListItem<T> curItem;
    private ListItem<T> nextItem;
    private LinkedList<T> theList;

    LinkedListIterator(LinkedList<T> list) {
      theList = list;
      prevItem = curItem = null;
      nextItem = theList.first;
    }

    public boolean hasNext() { return nextItem != null; }
    public T next() {
      prevItem = curItem;
      curItem = nextItem;
      if (nextItem != null) nextItem = nextItem.next;
      return curItem == null ? null : curItem.value;
    }
    public void remove() {
      if (curItem == null) return;
      if (prevItem == null) theList.first = curItem.next;
      else prevItem.next = curItem.next;
      if (curItem == theList.last) theList.last = prevItem;
      --theList.length;
    }
  }
}
Doubly Linked List

- Each item knows its predecessor as well as successor
- Allows for traversal in either direction
- Allows for efficient item removal and insertion
  - O(1) complexity

ADT Summary

- Abstractions provide walls that hide details of implementation
- Focus on external behavior
- Allow generalization of algorithms