Outline

- Review
  - Data Structures
  - Abstract Data Types
  - 2 Principles of OO Programming
    - encapsulation
    - reusability

- Java interfaces
  - can be used to specify ADT in Java
  - helps with encapsulation
Data Structures

- Data Structures
  - Is a systematic way of organizing and accessing data

- Examples
  - Stacks, queues, lists
Abstract Data Types

- ADT is a mathematical model of data structure that specifies
  - type of data stored
  - operations and their parameters supported on the data
  - ADT specifies what each operation does, but not how it does it
ADT Example: List

- List stores elements sequentially
- Each item has position $p$
  - $p$ is an abstract *place where we can be* in the list
- Operations for traversal
  - $IsEmpty()$: return true if list is empty, false otherwise
  - $first()$: return position $p$ of first elements in the list, ERROR if list is empty
  - $after(p)$: return position of element of the list following the one at position $p$; ERROR if $p$ is the last position
  - $isLast(p)$: true if $p$ is the last item in the list, false otherwise
ADT Example: List

- Operations for insertion and removal
  - `replaceElement(p,e)` replace element at position `p` with `e`, returning the element formerly at position `p`
  - `remove(p)` remove and return element at position `p`
  - `insertFirst(e)` insert new element `e` as the first element, return position of newly inserted element `e`
  - `insertAfter(p,e)` insert new element `e` after position `p`
To implement an ADT, we use the following principles of Object Oriented Programming:
- encapsulation (information hiding)
- reusability
Encapsulation

- User of data structure should not need to know details of its implementation.
- Creator of data structure should be able to change implementation without affecting applications that use it.
- Therefore implementation information should be hidden.
Reusability

- Encapsulation promotes reusability
- If data structure is useful for one application, it is probably useful for another application
- Therefore we should design it to be as reusable as possible
Non-Reusable List Implementation

Say we need a list of integers

A non reusable way to implement a list directly in Java code

- class Node{
    int n;    //integer in this node
    Node next;   // next node in the list
}

- In application,
  - to create an empty new linked list, use
    Node head = null;
  - to add a new element in front, use
    Node newNode = new Node();
    newNode.n = i;
    newNode.next = head; //head=list’s first element
    head = newNode;
Non-reusable List Disadvantages

- We must know and use `next` in applications
- If we want to re-implement `List` using an array, must change all the code
- To get a list of `String`, must copy and edit code
Reusable List implementation

- Define a class List
  
  ```java
  class List{
      boolean isLast(Position p){...};
      Position insertFirst(Object element){...};
      Object remove(Position p){...}
  }
  ```

- In application, declare a list
  
  ```java
  List data = new List();
  ```

- Use only the defined methods to access data
  
  ```java
  data.insertFirst(newElement)
  ```
Reusable List Implementation Advantages

- User of class does not have to know about next attribute
- We could re-implement List using an array, if needed
- Each node contains an Object, not int, therefore could reuse the code for list of String
We will follow Object Oriented approach of Java to implement ADT

- Start with ADT
- Define a class to implement the ADT
- Make public only the methods specified explicitly by an ADT
- Hide implementation details (encapsulation)
- Also make implementation as reusable as possible
Java Interface

- Interface
  - a way to inform about various methods that an object supports
  - to describe/enforce ADT
  - for encapsulation

- Interface is simply a list of method declarations with no data and no bodies
Public interface List {
    public boolean isEmpty();
    public Position first();
    public Position after(Position p);
    public boolean isLast(Position p);
    public Object remove(Position p);
    public Position insertFirst(Object e);
    public Position insertAfter(Position p, Object e);
}
Java Interface

- We use a class to **implement** an interface
- When a class implements an interface, it must implement all methods declared in the interface
- The implementing class can have other objects and other methods (not from the interface)
- Interface enforces the implementing class to have certain methods with specified *signatures*
  - signature is the list of methods declared in interface
public class NodeList implements List {
    protected int numElts; //number of items
    protected Dnode header, trailer;
    public NodeList() {
        numElts = 0;
        header = new DNode(null, null, null);
        trailer = new DNode(header, null, null);
        header.setNext(trailer);
    }
    public boolean isEmpty() { return (numElts < 1); }
    public boolean isFirst(Position p) {
        Dnode v = checkPosition(p);
        return v.getPrev() == header;
    }
}
Multiple Inheritance in Interfaces

- Multiple inheritance is not allowed for classes
  
  ```
  public class MyClass extends MySuperClass
  ```

- But multiple inheritance is allowed for Interfaces
  
  ```
  public class BoxedItem implements Sellable, Transportable
  ```
Interface as a Type

- Interface defines a new reference data type
  - can be *declared* but not *instantiated*

- Suppose **Position** is an Interface
  - **Legal**: `public Position p;`

- Cannot directly instantiate object of interface type

- Must use a class that implements Interface to instantiate
  - **Illegal**: `public Position p = new Position();`
  - **Legal**: `public Position p = new classImplementingPosition();`
**Interface as a Type**

```java
public interface Person{
    public boolean equalTo(Person other);
}

public class Student implements Interface Person{
    String id;
    public String getID() {return id;}
    public boolean equalTo(Person other) {
        Student otherSt = (Student) other;
        return(id.equals(otherSt.getID()));
    }
}
```

```java
Person p;
Student s = new Student();
```

null pointer

null pointer
Interface as a Type

```java
public interface Person{
    String id;
    public boolean equalTo(Person other);
}

public class Student implements Person{
    String id;
    public String getID() {return id;}
    public boolean equalTo(Person other) {
        Student otherSt = (Student) other;
        return (id.equals(otherSt.getID()));
    }
}
```

```java
Person p;
Student s = new Student();
p = s; // no casting of s into type Person is required
```
Interface as a Type

Person p;
Student s = new Student();
p = s;
Person q;

public class Student implements Interface Person{
    String id;
    public String getID() {return id;}
    public boolean equalTo(Person other) {
        Student otherSt = (Student) other;
        return(id.equals(otherSt.getID()));
    }
}
Person p;
Student s = new Student();
p = s;
Person q;
p.equalTo(q) // will not compile, q is not pointing to any object
p.equalTo(s) // will not compile s is not of type Person

public class Student implements Interface
Person{
  public String id;
  public String getID() {return id;}
  public boolean equalTo(Person other) {
    Student otherSt = (Student) other;
    return(id.equals(otherSt.getID()));
  }
}
**Interface as a Type**

Person p;
Student s = new Student();
p = s;
Person q;
p.equalTo((Person) s)  // this is fine

```java
public interface Person{
    public boolean equalTo(Person other);
}

public class Student implements Interface
Person{
    String id;
    public String getID() {return id;}
    public boolean equalTo(Person other) {
        Student otherSt = (Student) other;
        return(id.equals(otherSt.getID()));
    }
}
```
Interface as a Type

Person p;
Student s = new Student();
p = s;
Person q;
p.getID()  // will not compile, p is of type Person, does not know // about method getID

public interface Person{
    public boolean equalTo(Person other);
}

public class Student implements Interface
    Person{
        String id;
        public String getID() {return id;}
        public boolean equalTo(Person other) {
            Student otherSt = (Student) other;
            return(id.equals(otherSt.getID()));
        }
    }
public interface Person{
  public String id;
  public String getID() {return id;}
  public boolean equalTo(Person other) {
    Student otherSt = (Student) other;
    return(id.equals(otherSt.getID()));
  }
}

public class Student implements Interface
Person{
  String id;
  public String getID() {return id;}
  public boolean equalTo(Person other) {
    Student otherSt = (Student) other;
    return(id.equals(otherSt.getID()));
  }
}
Interface as a Type

Person p;
Student s = new Student();
p = s;
Person q;
Student f = (Student) p // casting is required
f.getID() // this is fine, as opposed to p.getID()

public class Student implements Interface
Person{
    String id;
    public String getID() {return id;}
    public boolean equalTo(Person other) {
        Student otherSt = (Student) other;
        return(id.equals(otherSt.getID()));
    }
}
Interface as a Type

Person p;
Student s = new Student();
p = s; // casting is not required
Student f = (Student) p  // casting is required

- from type A into a wider class B requires no explicit casting
- from type A into a narrower class B requires explicit casting

```java
public interface Person{
    public boolean equalTo(Person other);
}

public class Student implements Interface Person{
    String id;
    public String getID() {return id;}
    public boolean equalTo(Person other) {
        Student otherSt = (Student) other;
        return(id.equals(otherSt.getID()));
    }
}
```
Doubly Linked List

Class Node {
    private Node prev, next;
    private Object element;
    public Node(…)... //constructor
}

Empty linked list

Non-empty linked list
List: Implementing Positions with Interface

- Want to support “positions” in a linked list
  - One way to support positions would be to give the user a direct reference to a node

- User may want
  - to look up an element at a position $p$
  - to get the position $q$ after position $p$
  - etc.

- One way to support positions would be to give the user a direct reference to a node
List: Implementing Positions with Interface

Class Node {
    private Node prev, next;
    private Object element;
    public Node setNext(...) {...};
    public Node setElement() {...};
    ...
}

- If we give user reference to node, the user can
  - change element stored at node with `node.setElement(GoodGrade)`
  - change where `node.next` points to with `node.setNext(newLocation)`
  - erase the whole list

- User should change element stored at node only through list methods, such as
  `list.replaceElement(p, newElement)`

- Never want the user to change `node.next`
How to give node position without letting user do any damage?

Before giving a reference to a node, *wrap it up* (cast) to **Position** interface

**Position** interface allows only to look up stored element

```java
public interface Position{
    Object element();
}
```

What n sees:
- `setNext()`
- `getNext()`
- `element()`

Node n

What p sees:
- `element()`

Position p
public interface Position{
    Object element();
}

- **Node** class implements interface **Position**
  - must have method **element**()

**Class Node implements Position**{
    public Object element(){...} ...
}

- Inside linked list, we see nodes as of type **Node**
- Outside linked list we see nodes as of type **Position**
Implementing Positions in Doubly Linked List

Class Node implements Position{
    private Node prev,next;
    private Object element;
    public Node()... //constructor
    public Object element()
    {
        if (prev == null || next == null)
            "handle error somehow"
        else return element;
    }
}
Implementing Positions in Doubly Linked List

- linked list methods take as input parameters of type `Position` from the ‘outside world’
- Example: `public Position prev(Position p){...}`
- `Position p` actually refers to a specific node in the linked list

For internal use, to find the previous position, convert from `Position p` back to the `Node` class.
Implementing Positions in Doubly Linked List

Class NodeList implements List{

    protected Node checkPosition(Position p){
        // first check if valid position
        if (p == null || p == header || p == trailer )
            "handle the error somehow"
        Node temp = (Node) p;
        return temp;
    }

    ...

}

- checkPosition() converts p from Position back to Node
Implementing Positions in Doubly Linked List

Class NodeList implements List{
    ...

    public Position prev(Position p) {
        Node v = checkPosition(p);
        Node prev = v.getPrev();
        if (prev == header )
            "handle error somehow"
        return prev;  //no explicit cast required
    }
    ...
}

- v.getPrev() is a public method of class Node
- prev is automatically cast into type Position
List: Implementing Positions with Interface

- Book makes extensive use of Position interface
  - all information about position implementation is hidden
  - positions come up with other data structures as well