

What is an interface?

- A construct that names a type of data with specified methods
 - Must be implemented to associate the name with a particular kind of data
 - Implementor must contain all methods specified in the interface
 - Clients of the interface may use these methods

What is an interface?

```
interface Warning {  
    // Provide any warning message to consumers  
    String warningMessage();  
}
```

What is an interface?

```
class Aspirin implements Warning {  
    // Provide a warning message to consumers  
    String warningMessage() {  
        return "Don't take with blood thinners";  
    }  
}
```

What is an interface?

```
abstract class WarningList {  
    //Concat all warnings in this WarningList  
    abstract String allWarnings();  
}  
class EmptyWL extends WarningList {  
    //Concat all warnings in this EmptyWL  
    String allWarnings() {  
        return "";  
    }  
}  
class LargerWL extends WarningList {  
    Warning w;  
    WarningList rest;  
    LargerWL( ... )  
    //Concat all warnings in this LargerWL  
    String allWarnings() {  
        return this.w.warningMessage().concat(this.rest.allWarnings());  
    }  
}
```

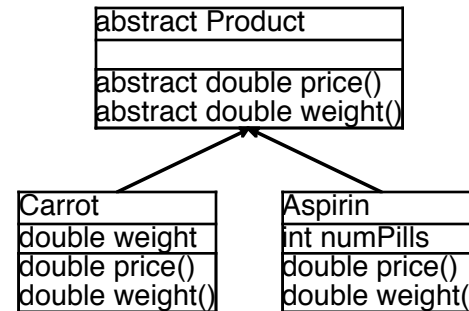
interface vs abstract class

- How are interfaces and abstract classes different?
 - A class can implement multiple interfaces, but can only extend one class
 - An abstract class can contain non-abstract entities

Non-abstract entities in abstract classes

- Permits code reuse
- Permits data reuse

Consider the classes below:



Write a method to calculate the price per ounce of the products

non-abstract methods in abstract class

```
abstract class Product {
    abstract double price();
    abstract double weight();
    double pricePerOunce() {
        return this.price() / this.weight();
    }
}
```

fields in abstract classes

- abstract classes can also have fields, shared between the sub classes
 - Can be initialized in either the abstract class or the in the constructors of the subclasses

```
abstract class Shape {
    Posn p;
}
class Disk extends Shape {
    Disk(Posn p) {
        this.p = p;
    }
}
```

A bit of Java

- abstract classes vs interfaces a bit of a Java centric discussion
- Some general principles
 - Don't repeat code where you don't have to: reuse
 - Programming using the methods, instead of the fields, allows partial implementation change
 - Use opportunities to write more general data structures

With reuse in mind...

Let's reconsider the following code:

```
class Person extends Ancestor {
    //To produce a list of family members of this Person with c eyes
    FamilyList withEye( String c ) {
        // ... this.color ... this.name ...
        // ... this.father.AncestorMethod( ... ) this.mother.AncestorMethod( ) ... c
        if ( this.color.equals(c) )
            return new LargerFL(this.name, this.father.withEye(c).append(this.mother.withEye(c)));
        else
            return this.father.withEye(c).append(this.mother.withEye(c));
    }
}
```

With reuse in mind...

We can name the value represented by the repeated code, instead of repeating it

```
class Person extends Ancestor {
    //To produce a list of family members of this Person with c eyes
    FamilyList withEye( String c ) {
        // ... this.color ... this.name ...
        // ... this.father.AncestorMethod( ... ) this.mother.AncestorMethod( ) ... c
        FamilyList ancestorsWithEye = this.father.withEye(c).append(this.mother.withEye(c));
        if ( this.color.equals(c) )
            return new LargerFL(this.name, ancestorsWithEye);
        else
            return ancestorsWithEye;
    }
}
```

Queues

- Extracting data is often important in programming
 - i.e. Storing data in a separate data structure until it is needed
- Speed of extraction can be important
- Lists, and trees can both be used for this
- A queue is another data structure

Queues

- Standard operations
- pop()
 - Return the top element of a queue
 - Occurs in constant time
 - Each pop on a queue returns the the next element down
- push(Object o)
 - Adds o to the queue, in the proper position
 - The proper position depends on the kind of queue

Queues

```
interface Queue {  
    // To return the top of the queue  
    Object pop();  
    // To add o to this queue  
    Queue push( Object o );  
    // To determine if this queue is empty  
    boolean empty();  
}
```

- Kinds of queues
 - fifo -- first-in first-out
 - filio -- first-in last-out
 - priority queue -- ranked by a weight

Implementing first-in last-out

- A list is first-in last-out
- Implementation (using List with Object)

```
class FiLoQueue implements Queue {  
    List theQueue;  
    FiLoQueue(List theQueue) {  
        this.theQueue = theQueue;  
    }  
  
    // To determine if the queue is empty  
    boolean empty() {  
        return this.theQueue.isEmpty();  
    }  
  
    // To return the top of the queue  
    Object pop() {  
        return this.theQueue.firstElt();  
    }  
  
    //To add o to this queue  
    Queue push( Object o ) {  
        return new FiLoQueue( this.theQueue.add(o) );  
    }  
}
```

List

```
abstract class List {  
    // To return the first element of a List  
    abstract Object firstElt();  
    // To determine if this List is Empty  
    abstract boolean isEmpty();  
    // To add o to this List  
    abstract List add(Object o);  
}
```

```
class Empty extends List {  
    // To return the first element ... more later  
    Object firstElt() {  
        return this;  
    }  
  
    // To determine if this is Empty  
    boolean isEmpty() {  
        return true;  
    }  
  
    // To add o to this Empty  
    List add( Object o ) {  
        return new Larger( o, this );  
    }  
}
```

```
class Larger extends List {  
    Object first;  
    List rest;  
    Larger( Object first, List rest ) {  
        this.first = first;  
        this.rest = rest;  
    }  
  
    // To return this.first  
    Object firstElt() {  
        return this.first;  
    }  
  
    // To determine if this is Empty  
    boolean isEmpty() {  
        return false;  
    }  
  
    // To add o to this Larger  
    List add( Object o ) {  
        return new Larger( o, this );  
    }  
}
```

List -- take 2

```
abstract class List {
  abstract Object firstElt();
  // To add o to this List
  List add(Object o) {
    return new Larger( o, this);
  }
}

class Empty extends List {
  Object firstElt() {
    return this;
  }
  ...
}

class Larger extends List {
  Object first;
  List rest;
  Larger( Object first, List rest ) {
    this.first = first;
    this.rest = rest;
  }
}

Object firstElt() {
  return this.first;
}
...
}
```

Using our Queue

```
class Roster {
  Queue students;
  Roster( Queue students ) {
    this.students = students;
  }

  String allStudents() {
    if ( students.empty() )
      return "";
    else
      return students.pop().toString().
        concat( new Roster(students).allStudents());
  }
}
```

What happens ...

What happens if we try to call allStudents on

```
new Roster( new FiLoQueue( new Empty()).push("Fred").push("Betty"))
```

What should have happened

- With a queue, we expect a different value (and smaller queue) after each pop
- pop() on our FiLoQueue always returns the same value

Mutation

- There are times when we must mutate data instead of copying it
- Some Reasons
 - Performance characteristics
 - Data usage expectations
 - Interactions with others
- All can be reasons not to mutate as well
 - To mutate or not: decide on a case-by-case basis

What does it mean to mutate a value?

- The value associated with a name changes
- Everyone who uses the data after the change uses the new value

How to mutate a value

```
class Canvas {  
  Pixel p1;  
  Pixel p2;  
  ...  
  void setPixelColor( Color c ) {  
    this.p1 = new Pixel( c );  
    this.p2 = new Pixel( c );  
  }  
}
```

Sample mutations

To DrScheme

Testing mutation

- Testing a side-effect can be difficult
 - Effective testing an open research question
- One technique looks at the value before and after the mutation
- Our current testing framework does not support tests of mutation

A mutating FiLoQueue

```
class FiLoQueue implements Queue {
  List theQueue;
  FiLoQueue(List theQueue) {
    this.theQueue = theQueue;
  }

  Object pop() {
    Object top = this.theQueue.firstElt();
    this.theQueue = this.theQueue.next();
    return top;
  }

  void push( Object o ) {
    this.theQueue = this.theQueue.add(o);
  }
}
```

List must also have a next() method, much like firstElt()