Abstract Data Types (ADT) / Interfaces
Barbara Liskov

- First woman to get a Ph.D. in Computer Science in the USA (Stanford 1968)
- Turing Award, 2008
- Inventor of Abstract Data Types
Abstract Data Types (ADTs)

As users of an object, we don't need to know how the object was written & implemented.

- Only the ways to use an abstraction are relevant!
- Make a connection: you can use the In class without knowing how it's implemented, just knowing the list of methods it had available.

An Abstract Data Type defines a class of abstract objects which is completely characterized by the operations available on those objects.

- one higher level of abstraction!
ADT in Java: Interfaces

An interface defines an ADT in Java

- An interface is a class-like construct that contains only constants and abstract methods
  - An abstract method is a method that is not implemented. Only the method signature is listed
  - A constant is a variable which value does not change during the execution of the program. They are declared static and final
- Gives a type for an object based on what it does, not on how it was implemented
- Describes a contract that objects must satisfy
Purposes of Interfaces

Abstract Data Types in Object Oriented Design have several purposes, including:

- defining contracts for objects
- enabling polymorphism (?)
- enabling multiple inheritance (???)

The latter two points are very powerful and interesting techniques in programming, but are a bit beyond what we can cover at this point in the class. We will focus on the contract aspect of ADTs in CIS 1100.
Contracts in Specifying Program Requirements

To specify program behavior, we've used a few different techniques in this class:

- long writeups on the course website
- writing a bunch of TODO comments in a starter file
- stubbed out functions with big specifier comments

These have only very weak enforcement mechanisms—it's up to you to check your compliance
**Interfaces as Contracts**

- Any class may be explicitly marked as **implementing** an interface
  - "signing the contract"

- If a class implements an interface, it must provide implementations for all of the abstract methods in the interface or else the program will not compile.
  - "enforcing the contract"

Useful on programming teams & in course environments to *formally specify* and *automatically enforce* what a class is supposed to do before you write it.
**Defining an Interface**

```java
public interface InterfaceName {
    constant declarations;
    abstract method signatures;
}
```
### Constant Declarations

Constants are `public static final` variables that are declared and initialized on the same line.

- **public**: accessible outside the interface file, which is important for keeping these values handy
- **static**: belongs to the ADT itself and not any particular instance
- **final**: putting the "constant" in constant—the compiler enforces that the variable's value cannot change.

```java
public static final double PI = 3.14159265358;
public static final int MAX_SIZE = 1000;
```
Abstract Methods

Unimplemented methods that consist only of a signature (name, return type, input parameter list)

- Completely abstract: defines only what the method should do, not how it works
- Classes that implement the interface will have methods that implement these signatures

```java
public double area();
public Point generateMidpoint(Point other);
public BankAccount openSharedAccount(BankAccount other, double split);
```
The **Shape Interface**

```java
public interface Shape {
    public static final double PI = 3.14159;
    public double area();
    public double perimeter();
    public void draw();
}
```

This interface says:

*Any class that calls itself a Shape must implement the methods double area(), double perimeter(), and void draw(). The class will also have access to the variable PI in scope everywhere throughout its definition.*
More Examples

```java
public interface Displaceable {
    public static final double DELTA = 0.00001;
    public boolean equals(Displaceable other);
    public void moveBy(double x, double y);
}
```

Any class that calls itself a `Displaceable` must implement the methods `boolean equals(Displaceable other)` and `void moveBy()`. The class will also have access to the variable `DELTA` in scope everywhere throughout its definition.
public interface Playable {
    public void play();
    public void stop();
    public void fastForward(double seconds);
    public void rewind(double seconds);
}

Any class that calls itself a Playable must implement the methods void play(), void stop(), void fastForward(double seconds), and void rewind(double seconds). The interface does not have any constants defined.
Implementing an Interface

To write a class `MyClass` that implements an interface `MyInterface`, you write:

```java
public class MyClass implements MyInterface {...}
```

For example, if `Circle.java` implements the `Shape` interface:

```java
public class Circle implements Shape {...}
```

- The class implementing the interface must implement all the methods defined in the interface
- The class is a **subtype** of the interface; the interface is a **supertype** of the class
  - Classes can implement multiple interfaces; interfaces can be implemented by multiple classes
```java
public class Circle implements Shape {
    private double radius, x, y;

    public Circle(double radius, double x, double y) {
        this.radius = radius;
        this.x = x;
        this.y = y;
    }

    @Override // <-- this annotation is explained on the next slide
    public double area() {
        return radius * radius * PI;
    }
    @Override
    public double perimeter() {
        return 2 * PI * radius;
    }
    @Override
    public void draw() {
        PennDraw.circle(x, y, radius);
    }
}
```
Implementing an interface: @Override

The @Override keyword can be used to indicated that a method implements (overrides) a method defined in the interface.

<table>
<thead>
<tr>
<th></th>
<th>Has @Override</th>
<th>Does Not Have @Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method in Interface</td>
<td>Compiles without Error</td>
<td>Compiles without Error</td>
</tr>
<tr>
<td>Method Not in Interface</td>
<td>Compilation Error</td>
<td>Compiles without Error</td>
</tr>
</tbody>
</table>

Optional, but very useful:

- If the interface changes, methods annotated with @Override keyword will raise a compiler error. To fix the problem, make your code to adhere to the new interface.
Using Objects from an Interface

Declare a variable of type the interface and initialize it using the subtype constructor.

- Invoke the methods defined in the ADT on the object

Example:

```java
Shape c = new Circle(0.5, 0.1, 0.2);
c.area();
c.perimeter();
c.draw();
```
Using Objects from an Interface

Also OK to just use the concrete subtype for the variable if you want to ignore the ADT

- (this makes the ADT pointless if you do it everywhere)

Example:

```java
Circle c = new Circle(0.5, 0.1, 0.2);
c.area();
c.perimeter();
c.draw();
```
Using Objects from an Interface

NOT OK to try to instantiate an object from the interface

- Interfaces don't ever have constructors!

NONFUNCTIONAL EXAMPLE:

```java
Shape c = new Shape(0.5, 0.1, 0.2); // DOES NOT WORK!!
c.area();
c.perimeter();
c.draw();
```
Grouping Objects by Behavior

It's possible to collect multiple objects of different classes in the same array or list as long as they all have the same ADT that the structure is declared to store.

```java
Shape[] shapes = new Shape[3];
Shape smallRectangle = new Rectangle(0.5, 0.5, 0.1, 0.2);
Shape bigRectangle = new Rectangle(0.3, 0.6, 0.2, 0.1);
Shape myCircle = new Circle(0.1, 0.8, 0.2);
shapes[0] = smallRectangle;
shapes[1] = bigRectangle;
shapes[2] = myCircle;
```
Grouping Objects by Behavior

Given the Shape interface:

```java
public interface Shape {
    public static final double PI = 3.14159;
    public double area();
    public double perimeter();
    public void draw();
}
```

We know that we can write:

```java
Shape[] shapes = new Shape[3];
// refer to the way we filled the array on the last slide
for (int i = 0; i < shapes.length; i++) {
    Shape current = shapes[i];
    current.draw();
}
```
Abstract Data Type Relationships

- An abstract object (an ADT is the object’s type) may be operated upon by the operations which define its abstract type
- An abstract object may be listed as a parameter to a procedure (function/method)
- An object may be assigned to a variable with an abstract data type, but only if the object's type is a subtype of that ADT.
List ADT

List.java is an interface that defines the List ADT.

- The complete List ADT is huge.
  - Bad: doesn't fit on a slide
  - Good: if you're using a List implementation (ArrayList/LinkedList), you know it has a ton of stuff it can do!
- There are multiple classes built in to Java that implement the List ADT
  - ArrayList, as we studied already. Built on an array that is managed for you.
  - LinkedList, built on a series of linked Nodes accessible from some start point.
**List ADT: Why?**

Different List implementations have different performance (memory requirements, speed efficiency) tradeoffs.

- Most important thing about all of them is that they are Lists! So you know that you can add, get, set, etc.
- You can swap out different implementations to fit your performance requirements without changing the logic of the implementation.
List Profiling Demo