Relatedness of types

- Consider the task of writing classes to represent 2D shapes such as Circle, Rectangle, and Triangle.
- There are certain attributes or operations that are common to all shapes: perimeter, area
- By being a Shape, you promise that you can compute those attributes, but each shape computes them differently.

The area and perimeter of shapes

- Rectangle (as defined by width w and height h):
  - area \( = w \cdot h \)
  - perimeter \( = 2w + 2h \)
- Circle (as defined by radius r):
  - area \( = \pi r^2 \)
  - perimeter \( = 2\pi r \)
- Triangle (as defined by side lengths a, b, and c)
  - area \( = \frac{1}{2} (a \cdot b \cdot c) \) if known
  - perimeter \( = a + b + c \)
  
  where \( s = \frac{1}{2} (a + b + c) \)

Interfaces with abstract classes

```java
public abstract class Shape {
    public abstract double area();
    public abstract double perimeter();
}
```
Java Interfaces

- An interface for shapes:
  ```java
  public interface Shape {
    public double area();
    public double perimeter();
  }
  ```

  This interface describes the functionality common to all shapes.
  (Every shape knows how to compute its area and perimeter.)

- Interface declaration syntax:
  ```java
  public interface <name> {
    public <type> <name>(<type> <name>, ...);
    public <type> <name>(<type> <name>, ...);
    ...
  }
  ```

  All methods are public!

Implementing an interface

- A class can declare that it implements an interface:
  ```java
  public class Circle implements Shape {
    private double radius;
    // Constructs a new circle with the given radius.
    public Circle(double radius) {
      this.radius = radius;
    }
    // Returns the area of the circle.
    public double area() {
      return Math.PI * radius * radius;
    }
    // Returns the perimeter of the circle.
    public double perimeter() {
      return 2.0 * Math.PI * radius;
    }
  }
  ```

Requirements

- If we write a class that claims act like a `Shape` but doesn't implement the `area` and `perimeter` methods, it will not compile.

  Example:
  ```java
  public class Banana implements Shape {
    // without implementing area or perimeter
  }
  ```

  The compiler error message:
  ```java
  Banana.java:1: Banana is not abstract and does not override abstract method area() in Shape
  public class Banana implements Shape {
  ^
  ```

Diagramming an interface

- We draw arrows upward from the classes to the interface(s) they implement.

Rectangle

```java
public class Rectangle implements Shape {
  private double width, height;
  // Constructs a new rectangle with the given dimensions.
  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }
  // Returns the area of this rectangle.
  public double area() {
    return width * height;
  }
  // Returns the perimeter of this rectangle.
  public double perimeter() {
    return 2.0 * (width + height);
  }
}
```
Triangle

```java
public class Triangle implements Shape {
    private double a;
    private double b;
    private double c;

    // Constructs a new Triangle given side lengths.
    public Triangle(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // Returns a triangle's area using Heron's formula.
    public double area() {
        double s = (a + b + c) / 2.0;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }

    // Returns the perimeter of the triangle.
    public double perimeter() {
        return a + b + c;
    }
}
```

Interfaces and polymorphism

- Polymorphism is possible with interfaces.
- Example:
  ```java
  public static void printInfo(Shape s) {
      System.out.println("The shape: ");
      System.out.println("area: ");
      System.out.println("perimeter: ");
  }
  
  Any object that implements the interface may be passed as the parameter to the above method.
  Circle circ = new Circle(12.0);
  Triangle tri = new Triangle(5, 12, 13);
  printInfo(circ);
  printInfo(tri);
  ```

Comments about Interfaces

- The term interface also refers to the set of public methods through which we can interact with objects of a class.
- Methods of an interface are abstract.
- Think of an interface as an abstract base class with all methods abstract.
- Interfaces are used to define a contract for how you interact with an object, independent of the underlying implementation.
- Separate behavior (interface) from the implementation.

Interfaces and inheritance

- Interfaces allow us to get around the Java limitation of no multiple inheritance – a class can implement several interfaces.
  ```java
class ImplementsSevera1 implements Interface1, Interface2 {
    // implementation
}
```
- A class can implement an interface AND extend another class.
- Inheritance can be applied to interfaces – an interface can be derived from another interface.

When to use interfaces or abstract classes

- An abstract class: mix of abstract and non-abstract methods, so some default implementations.
- An abstract class can also have static methods, private and protected methods, etc.
Commonly used Java interfaces

- The Java class library contains several interfaces:
  - `Comparable` - allows us to order the elements of an arbitrary class
  - `Serializable` (in `java.io`) - for saving objects to a file.
  - `List`, `Set`, `Map`, `Iterator` (in `java.util`) - describe data structures for storing collections of objects

The Java `Comparable` interface

A class can implement the `Comparable` interface to define an ordering for its objects.

```java
public interface Comparable<E> {
    public int compareTo(E other);
}
```

```java
public class Employee implements Comparable<Employee> {
    // ...
}
```

A call of `a.compareTo(b)` should return:
- a value < 0 if `a` comes "before" `b` in the ordering,
- a value > 0 if `a` comes "after" `b` in the ordering,
- or 0 if `a` and `b` are considered "equal" in the ordering.

Comparable and sorting

If you implement `Comparable`, you can sort arbitrary objects using the method `Arrays.sort`.

```java
StaffMember[] staff = new StaffMember[3];
staff[0] = new Executive();
staff[1] = new Employee();
staff[2] = new Volunteer();
Arrays.sort(staff);
```

Note that you will need to provide an implementation of `compareTo`

compareTo tricks

- Delegation trick - If your object's attributes are comparable (such as strings), you can use their `compareTo`:

  ```java
  // sort by employee name
  public int compareTo(StaffMember other) {
      return name.compareTo(other.getName());
  }
  ```

Another example

```java
public class Contact implements Comparable<Contact> {
    private String firstName, lastName, phone;
    public boolean equals(Object other) {
        if (!(other instanceof Contact)) return false;
        return lastName.equals(((Contact)other).getLastName()) &&
                 firstName.equals(((Contact)other).getFirstName());
    }
    // Use both last and first names to determine ordering.
    public int compareTo(Contact other) {
        String otherFirst = other.getFirstName();
        String otherLast = other.getLastName();
        if (lastName.equals(otherLast))
            return firstName.compareTo(otherFirst);
        else
            return lastName.compareTo(otherLast);
    }
}
```

Note the difference in the parameters of `compareTo()` and `equals()`.

In version 1.4 of Java `compareTo()` needed parameter of type `Object`.

```java
import java.util.Arrays;

Contact[] friends = new Contact[6];
friends[0] = new Contact("John", "Smith", "610-555-7384");
friends[2] = new Contact("Mark", "Riley", "733-555-2969");
Arrays.sort(friends);
for (int i=0; i<friends.length; i++)
    System.out.println(friends[i]);
```
ArrayList

- The ArrayList declaration:
  public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

- The List interface includes:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E get(int index)</td>
<td>Returns the element at the specified position</td>
</tr>
<tr>
<td>int indexOf(Object o)</td>
<td>Returns the index of the first occurrence of the specified element</td>
</tr>
<tr>
<td>E remove(int index)</td>
<td>Removes the element at the specified position</td>
</tr>
<tr>
<td>E set(int index, E element)</td>
<td>Replaces the element at the specified position</td>
</tr>
</tbody>
</table>

Lists and collections

- The declaration of the List interface:
  public interface List<E> extends Collection<E>

- Has methods that any collection of elements should have: add, clear(), contains, isEmpty(), remove, size()

Interface for a sorted list

- Let’s design the interface for a list of items that is supposed to be maintained in sorted order.

The Predator interface

```java
public interface Predator {
    void stalk(Prey p);
    boolean chase(Prey p);
    void eat(Prey p);
}
```

The Predator interface

```java
public interface VenomousPredator extends Predator, Venomous {
    //interface body
}
```

Note: an interface can extend multiple interfaces. Why is this not a problem?