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
- Every shape has these attributes, but each computes them differently.

Common behavior

- Let's write shape classes with methods named perimeter and area.
- We'd like to be able to write client code that treats different shape objects in the same way, insofar as they share common behavior, such as:
  - Write a method that prints any shape's area and perimeter.
  - Create an array of shapes that could hold a mixture of the various shape objects.

Interfaces with abstract classes

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

The area and perimeter of shapes

- Rectangle (as defined by width w and height h):
  area \( = w \times h \)
  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 \( = \sqrt{s(s-a)(s-b)(s-c)} \)
  where \( s = \frac{1}{2}(a+b+c) \)
  perimeter \( = a + b + c \)

Interfaces

- **interface**: A list of methods that a class promises to implement.
  - Inheritance gives you an is-a relationship and code-sharing.
    - An Executive object can be treated as a StaffMember, and Executive inherits StaffMember's code.
  - Interfaces give you an is-a relationship without code sharing.
    - A Rectangle object can be treated as a Shape.
  - Analogous to non-programming idea of roles or certifications
    - "I'm certified as a CPA accountant. The certification assures you that I know how to do taxes, perform audits, and do management consulting."
    - "I'm certified as a Shape. That means you can be sure that I know how to compute my area and perimeter."
Interfaces with Java interfaces

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

  - This interface describes the features common to all shapes. (Every shape has an area and perimeter.)

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

  - All methods are public!

Implementing an interface

- A class can declare that it implements an interface.
  - This means the class contains an implementation for each of the abstract methods in that interface. (Otherwise, the class will fail to compile.)

- Syntax for implementing an interface
  ```java
  public class <name> implements <interface name> {
      ...
  }
  ```

Requirements

- If we write a class that claims to be a Shape but doesn’t implement the area and perimeter methods, it will not compile.
  ```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.
  - There is a supertype-subtype relationship here; e.g., all Circles are Shapes, but not all Shapes are Circles.

Rectangle

- A class can declare that it implements an interface.
  - This means the class contains an implementation for each of the abstract methods in that interface. (Otherwise, the class will fail to compile.)

- Syntax for implementing an interface
  ```java
  public class <name> implements <interface name> {
      ...
  }
  ```

Requirements

- If we write a class that claims to be a Shape but doesn’t implement the area and perimeter methods, it will not compile.
  ```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 {
    ^
    ```
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

- The is-a relationship provided by the interface means that the client can take advantage of polymorphism.
- Example:
  ```java
  public static void printInfo(Shape s) {
      System.out.println("The shape: " + s);
      System.out.println("area: " + s.area());
      System.out.println("perim: " + s.perimeter());
  }
  ```
- Any object that implements the interface may be passed as the parameter to the above method.
  ```java
  Circle circ = new Circle(12.0);
  Triangle tri = new Triangle(5, 12, 13);
  printInfo(circ);
  printInfo(tri);
  ```

Interfaces and inheritance

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

Separation of interface and implementation

- 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.
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 classes and interfaces
- Comparable – allows us to order the elements of an arbitrary class
- Serializable (in java.io) – for classes whose objects are able to be saved to files.
- List, Set, Map, Iterator (in java.util) – describe data structures for storing collections of objects

Comparable

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

A class can implement the Comparable interface to define a natural ordering for its objects.

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.

compareTo tricks

- delegation trick - If your object's fields are comparable (such as strings), use their compareTo results:
  ```java
  // sort by employee name
  public int compareTo(StaffMember other) {
      return name.compareTo(other.getName());
  }
  ```

Show StaffMember example

Comparable and sorting

- The Arrays class in java.util has a static method `sort` that sorts the elements of an array
  ```java
  StaffMember[] staff = new StaffMember[3];
  staff[0] = new Executive();
  staff[1] = new Employee();
  staff[2] = new Hourly();
  Arrays.sort(staff);
  ```

- The Collections class in java.util has a static method `sort` that sorts the elements of a list
  ```java
  List<StaffMember> staff = new ArrayList<StaffMember>();
  staff.add(new Employee());
  Collections.sort(staff);
  ```
Another example

```java
public class Contact implements Comparable<Contact>
{
    private String firstName, lastName, phone;

    public boolean equals(Object other) {
        if (null == other)
            return false;
        if (null == other ||
                !(other instanceof Contact))
            return false;

        Contact c = (Contact) other;
        if (!firstName.equals(c.firstName)
                || !lastName.equals(c.lastName))
            return false;

        return true;
    }

    public int compareTo(Contact other) {
        if (lastName.equals(other.lastName))
            return firstName.compareTo(other.firstName);
        else
            return lastName.compareTo(other.lastName);
    }

    // Uses both last and first names to determine ordering.
}
```

Note the difference in the parameters of `compareTo()` and `equals()`. In version 1.4 of Java `compareTo()` needed parameter of type `Object`.

### ArrayList

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

- The `List` interface includes:
  ```java
  Method                        | Description
  ------------------------------|-----------------|
  E get(int index)              | Returns the element at the specified position
  int indexOf(Object o)         | Returns the index of the first occurrence of the specified element
  E remove(int index)           | Removes the element at the specified position
  E set(int index, E element)   | Replaces the element at the specified position
  ```

### Lists and collections

- The declaration of the `List` interface:
  ```java
  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. Each item is recognized by a key.