Outline

- One Problem, Many Solutions
- Polymorphism
- Other Uses For Interfaces
- Generic Classes
- Interface Enhancements in Java SE 8
One Problem, Many Solutions

- We want to implement a set of classes to represent shapes
  - Such as circles, rectangles, etc…
  - We want to calculate area and perimeter of each shape
    - Regardless of its type and characteristics
- One approach:
  - Implement each class independently
  - Limitations:
    - No formal relationship between different shapes
    - Someone could add another shape with out area or perimeter
One Problem, Many Solutions

- Another approach to avoid these limitations
- **Java Interfaces**
  - A mechanism for formally define the relationship between objects
  - Benefits:
    - Enforces standardization between objects.
    - In our shapes it will enforce the need for perimeter and area
    - Will also ensure the methods have the same signature no matter the shape
    - Very important… we will see later
One Problem, Many Solutions

- Simple Interface:
  - Notice it is *not designated* as a class!
  - The method bodies are absent!
    - Responsibility of any class that *implements* it.
  - Depend on what the class wants it to do.

```java
public interface Shape {
    /**
     * Calculate the area of this shape
     *
     * @return the calculated area
     */
    public double area();

    /**
     * Calculate the perimeter of this shape
     *
     * @return the calculated perimeter
     */
    public double perimeter();
}
```
One Problem, Many Solutions

- Simple Interface:
  - For example a *Circle* class

```java
public interface Shape {
    /**
     * Calculate the area of this shape
     * @return the calculated area
     */
    public double area();

    /**
     * Calculate the perimeter of this shape
     * @return the calculated perimeter
     */
    public double perimeter();
}

public class Circle implements Shape {
    private int radius;     // radius of this circle object

    /**
     * Constructor for Circle objects.
     * @param radius the radius of this circle.
     */
    public Circle(int radius) {
        this.radius = radius;
    }

    /**
     * Calculate the area of this circle
     * @return the calculated area
     */
    public double area() {
        // PI is a double.
        return Math.PI * radius * radius;

        // or this: return Math.PI * Math.pow(radius, 2);
    }

    /**
     * Calculate the perimeter of this circle
     * @return the calculated perimeter
     */
    public double perimeter() {
        double diameter = 2 * radius;
        return Math.PI * diameter;
    }
}
One Problem, Many Solutions

- Simple Interface:
  - For example a *Circle* class
  - Notice the *implements* keyword
  - Means it will provide code missing in the *Shape* class
  - We can then do:
    
    ```java
    Shape myCircle = new Circle(10);
    ```

```java
public class Circle implements Shape {
    private int radius; // radius of this circle object
    /**
     * Constructor for Circle objects.
     * @param radius the radius of this circle.
     */
    public Circle(int radius) {
        this.radius = radius;
    }
    /**
     * Calculate the area of this circle
     * @return the calculated area
     */
    public double area() {
        // PI is a double.
        return Math.PI * radius * radius;
        // or this: return Math.PI * Math.pow(radius, 2);
    }
    /**
     * Calculate the perimeter of this circle
     * @return the calculated perimeter
     */
    public double perimeter() {
        double diameter = 2 * radius;
        return Math.PI * diameter;
    }
}
```
One Problem, Many Solutions

- Why do all this?
  - Why not just make a Circle, Rectangle, etc.. as stand alone?
- What if we want an ArrayList of different shapes?
  - We cannot do this with stand alones...
- But we can with a common Interface

```java
ArrayList <Shape> myArrayList = new ArrayList <Shape>();
myArrayList.add(new Circle (10));
myArrayList.add(new Rectangle (10,4));
myArrayList.add(new Square (3));
```

- Notice the ArrayList has been set to take in Shapes
  - Circle, Rectangle, and Shape all implement Shape
  - This makes them also a Shape!
Polymorphism

- Consider our **Shape** example:
- Suppose we create this array:
  
  ```java
  Shape[] items = new Shape[4];
  ```
- Since **Circles**, **Rectangles**, and **Squares** share an is_a relationship with shape
  We can do:

  ```java
  items[0] = new Circle(10);
  items[1] = new Circle(12);
  items[2] = new Rectangle(3, 12);
  items[3] = new Square(7);

  for (int loop = 0; loop < 4; loop++) {
      System.out.println("Area is " + items[loop].area());
      System.out.println("Perimeter is " + items[loop].perimeter());
  }
  ```
Polymorphism

- Lets take a closer look at the loop.
  - The area and perimeter are calculated differently for each of the classes
    - By using an interface, each can do this…
    - At runtime Java will resolve the type of the object
      - Then call the correct method
  - This is referred to as **polymorphism**
  - Greek for “many forms”
  - Calling a single method name, but the object determines the code run.

```java
items [0] = new Circle(10);
items [1] = new Circle(12);
items [2] = new Rectangle(3,12);
items [3] = new Square(7);

for (int loop = 0; loop < 4; loop++) {
    System.out.println("Area is " + items[loop].area());
    System.out.println("Perimeter is " + items[loop].perimeter());
}
```
Other Uses For Interfaces

- Another reason to use interfaces: Sorting objects of a given class
  - Suppose we have Vehicle objects
    - Each has:
      - Instance variable representing the odometer reading
  - We cannot compare Vehicle objects using < and >
  - How could we order them?
  - We can write a specialized version of a standard sorting algorithm
    - Such as merge sort.
  - But would be better to use the same sorting algorithm for every new class type.
    - We can use interfaces to solve this…
    - Then then sorting code will work for any class we create… no specialization
Other Uses For Interfaces

- In Java, a sorting routine is already provided
  - Works for any class that implements the `Comparable` interface
  - We can have our `Vehicle` implement it

```java
public class Vehicle implements Comparable<Vehicle> {
    public int compareTo(Vehicle other) {
        // code here
    }
}
```

- Since we will be comparing `Vehicles` only to other `Vehicles`
  - We implement `Comparable<Vehicle>`
  - We then add code in our `compareTo` function to compare `Vehicles`
  - Note: It will compare the object calling the function to the Object pass in as a parameter.
Other Uses For Interfaces

• In Java, a class can implement many interfaces.

```java
public class LinkedList<E> implements List<E>, Deque<E>, Cloneable, Serializable {
    ...
}
```

• In this case:
  • List: allows the **LinkedList** to be iterable with foreach loops.
  • **Iterable** is actually another interface
    • Implemented by **List**
  • **Cloneable**: can make clones, or copies
  • **Serializable**: can serialize it...
Generic Classes

- Our classes can accommodate any type
- Let's start with the simple interface for a **Queue**
  - It will define the basic operations for the **Queue**
  - This interface contains some methods that we might not need
    - This depends on how we implement the **Queue**
      - Example: if using a linked data structure, `isFull()` might not apply
      - Example: if using an array this method might apply
      - **This is okay!**
Generic Classes

- Making a generic interface
  ```java
  public interface Queue<E> { ... }
  ```
- Notice the `E`
  - It is not the name of any particular type
  - It is just a placeholder
  - Once mentioned in the header, it can be used anywhere
    - Just like a regular type
- We can then use the interface
  ```java
  public class MyGenericQueue<E> implements Queue<E> { ... }
  ```
- We can create a `MyGenericQueue` that contains any object for type `E`
- Example:
  ```java
  private MyGenericQueue<Circle> circleQueue = new MyGenericQueue<Circle>();
  ```
- The compiler will replace the placeholder with `Circle`
Generic Classes

- Both support modularity and code reuse…
- To very important concepts in OO programming!
Interface Enhancements in Java SE 8

- Prior to Java SE 8
  - Only abstract methods were allowed in an interface
  - An abstract method contains only a declaration
    - No implementation
- Java SE 8 added to interfaces:
  - Default methods:
    - Allows the interface to evolve over time
    - But remain backwards compatible
    - Suppose an interface is used for some time…
      - We decide to add additional methods to it
        - If written with abstract methods it will break things.
          - We will have to update all classes using this interface.
  - Solution is to add a default method.
    - They include a default implementation
      - If a class using this interface does not provide code for the method
        - It will use the interfaces default
Interface Enhancements in Java SE 8

- Prior to Java SE 8
  - Only abstract methods were allowed in an interface
  - An abstract method contains only a declaration
    - No implementation
- Java SE 8 added to interfaces (con’t):
  - Static methods
    - Recall: static methods are associated with the class
    - Static methods in a interface allows for improved organization
    - Instead of making a helper class…
    - You just add them to the interface