Chapter 12 Interfaces

CS1: Java Programming
Colorado State University

Original slides by Daniel Liang
Modified slides by Kris Brown

Motivations

- You have learned how to write simple programs to create and display GUI components. Can you write the code to respond to user actions, such as clicking a button to perform an action?
- In order to write such code, you have to know about interfaces. An *interface* is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.

Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface? Why is an interface useful?

An interface is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects. For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {
  constant declarations;
  abstract method signatures;
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Interface is a Special Class

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

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.

Interface as a contract

- Analogous to the idea of roles or certifications in real life:
  - "I'm certified as a CPA accountant. The certification assures you that I know how to do taxes, perform audits."

  Compare to:
  - "I'm certified as a Shape. That means you can be sure that I know how to compute my area and perimeter."

The area and perimeter of shapes

- Rectangle (as defined by width w and height h):

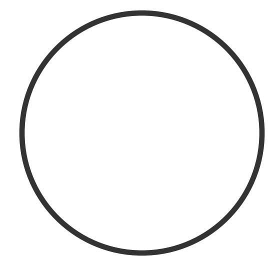
  area $= w\,h$

  perimeter $= 2w + 2h$

- Circle (as defined by radius r):

  area $= \pi\, r^2$

  perimeter $= 2\,\pi\, r$

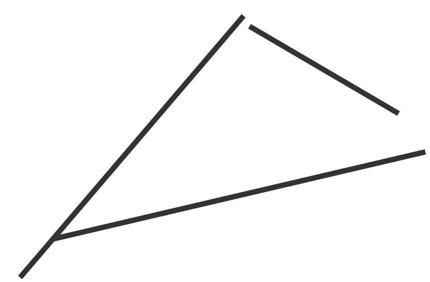


- Triangle (as defined by side lengths a, b, and c)

  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 encodes an is-a relationship and provides code-sharing.
    - An Executive object can be treated as a StaffMember, and Executive inherits StaffMember’s code.
  - An interface specifies what an object is capable of; no code sharing.
    - Only method **stubs** in the interface
    - Object **can-act-as** any interface it **implements**
    - A Rectangle does what you expect from a Shape as long as it implements the interface.

Java Interfaces

- An interface for shapes:

```
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:

```
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

```
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;
    }
}
```

Implementing an interface

- A class can declare that it *implements* an interface.
  - This means the class needs to contain an implementation for each of the methods in that interface.

    (Otherwise, the class will fail to compile.)

- Syntax for implementing an interface

```
public class <name> implements
  <interface name> {
      ...
}
```

Requirements

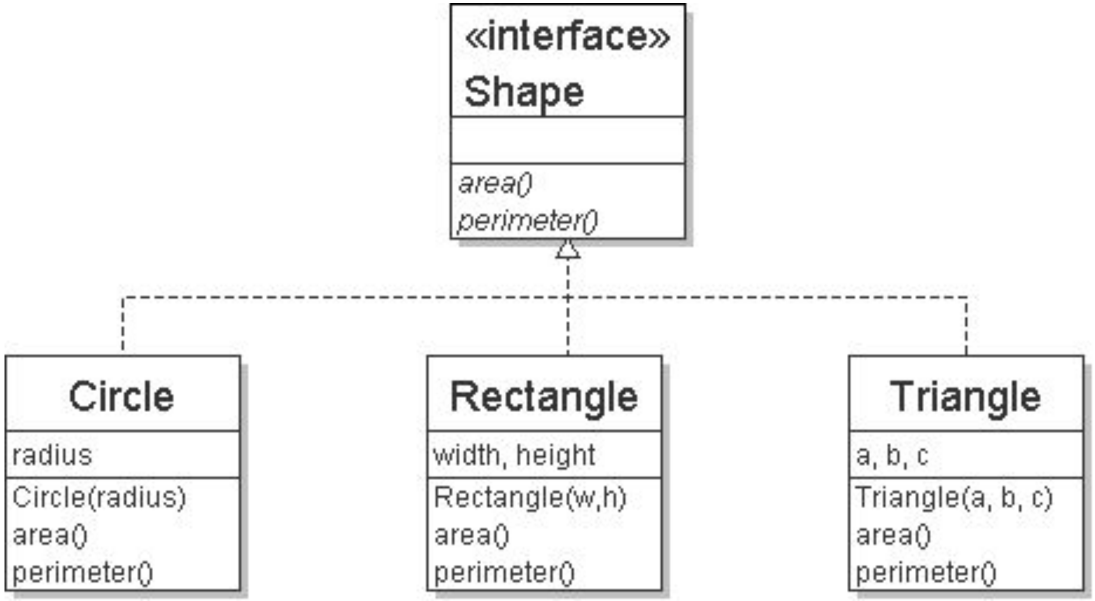
- If we write a class that claims act like a `Shape` but doesn't implement the `area` and `perimeter` methods, it will not compile.
  - Example:

```
public class Banana implements Shape {
    //without implementing area or perimiter
}
```

  - The compiler error message:

```
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 from the classes to the interface(s) they implement.
- Like inheritance, an interface represents an is-a relationship (a Circle is a Shape).

Rectangle

```
public class Rectangle implements Shape {
    private double width;
    private double 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

```
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:

```
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " +
s.perimeter());
    System.out.println();
}
```

Interface is a type!

- 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);
```

Interfaces and polymorphism

- We can create an array of an interface type, and store any object implementing that interface as an element.

```
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);
Shape[] shapes = {circ, tri, rect};
for (int i = 0; i < shapes.length; i++) {
    printInfo(shapes[i]);
}
```

  - Each element of the array executes the appropriate behavior for its object when it is passed to the `printInfo` method, or when `area` or `perimeter` is called on it.

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

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.

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

```
StaffMember [] staff = new StaffMember[3];
staff[0] = new Executive(…);
staff[1] = new Employee(…)
staff[2] = new Hourly(…);
staff[3] = 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`:

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