Lecture 20: Interfaces and Abstract classes

Building Java Programs: A Back to Basics Approach

by Stuart Reges and Marty Stepp

Interface

Interface

- Generally, an interface is a shared boundary across which two or more separate components of a computer system exchange information(Wiki)
  - exchange can be between software, peripheral devices, hardware, humans and combinations of these.
  - an interface exposes the functionality of an application or service

- Examples:
  - an interface between a person and a car’s engine/transmission driving system is the steering wheel, the brake and shift gears.
  - a waiter is an interface between a person and the kitchen of a restaurant.
  - the Facebook phone app is an interface to Facebook’s services.

Interface in Java

- **interface**: A list of methods that a class can implement. An interface defines a contract that implementing classes must fulfill.

- An interface definition consists of signatures of methods without any implementing code.
  - Inheritance gives you an is-a relationship and code-sharing.
    - A `Lawyer` object can be treated as an `Employee`, and `Lawyer` inherits `Employee`'s code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a `Shape`.
  - Analogous to the idea of roles or certifications:
    - "I'm certified as a CPA accountant. That means I know how to compute taxes, perform audits, and do consulting.”
    - "I'm certified as a Shape. That means I know how to compute my area and perimeter."

Declaring an interface

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
}
```

Example:

```
public interface Vehicle {
    public double speed();
    public void setDirection(int direction);
}
```

- **abstract method**: A header without an implementation.
  - The actual body is not specified, to allow/force different classes to implement the behavior in its own way.

Interface

- An interface can have only abstract methods.
- An interface can’t have static methods.
- An interface supports multiple inheritance. That is, a class can implement multiple interfaces.

```
public class Panel implements MouseListener,
  ActionListener, KeyboardListener
{
…
}
```

- An interface has only static and final variables.
  - An interface cannot have instance variables. Thus, an interface does not have constructors.

Relatedness of types

Write a set of `Circle`, `Rectangle`, and `Triangle` classes.

- Certain operations that are common to all shapes.

  perimeter- distance around the outside of the shape

  area- amount of 2D space occupied by the shape

- Every shape has them but computes them differently.

Shape area, perimeter

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

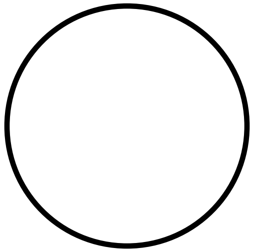
  area $= w\,h$

  perimeter $= 2w + 2h$

- Circle (as defined by radius r):

  area $= \pi\, r^2$

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

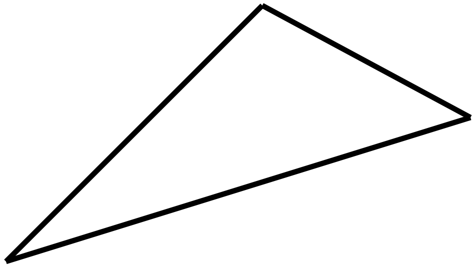


- Triangle (as defined by side lengths a, b, and c)

  area $= \sqrt{(s\,(s - a)\,(s - b)\,(s - c))}$

  where $s = ½\,(a + b + c)$

  perimeter $= a + b + c$



Common behavior

- Write shape classes with methods `perimeter` and `area`.

- We'd like to be able to write client code that treats different kinds of shape objects in the same way, 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.
  - Write a method that could return a rectangle, a circle, a triangle, or any other shape we've written.

Shape interface

```
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.)
- **These methods are implicitly abstract.**

Implementing an interface

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

- Example:

```
public class Bicycle implements Vehicle {
    ...
}
```

- A class can declare that it *implements* an interface.
  - This means the class must contain each of the abstract methods in that interface. (Otherwise, it will not compile.)

    (What must be true about the `Bicycle` class for it to compile?)

Interface requirements

- If a class claims to be a `Shape` but doesn't implement the `area` and `perimeter` methods, it will not compile.

  - Example:

    ```
    public class Banana implements Shape {
        ...
    }
    ```

  - 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 {
              ^
    ```

Complete Circle class

```
// Represents circles.
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 this circle.
    public double area() {
        return Math.PI * radius * radius;
    }

    // Returns the perimeter of this circle.
    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```

Complete Rectangle class

```
// Represents rectangles.
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);
    }
}
```

Complete Triangle class

```java
// Represents triangles.
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 this 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 this triangle.
    public double perimeter() {
        return a + b + c;
    }
}
```

Interfaces + polymorphism

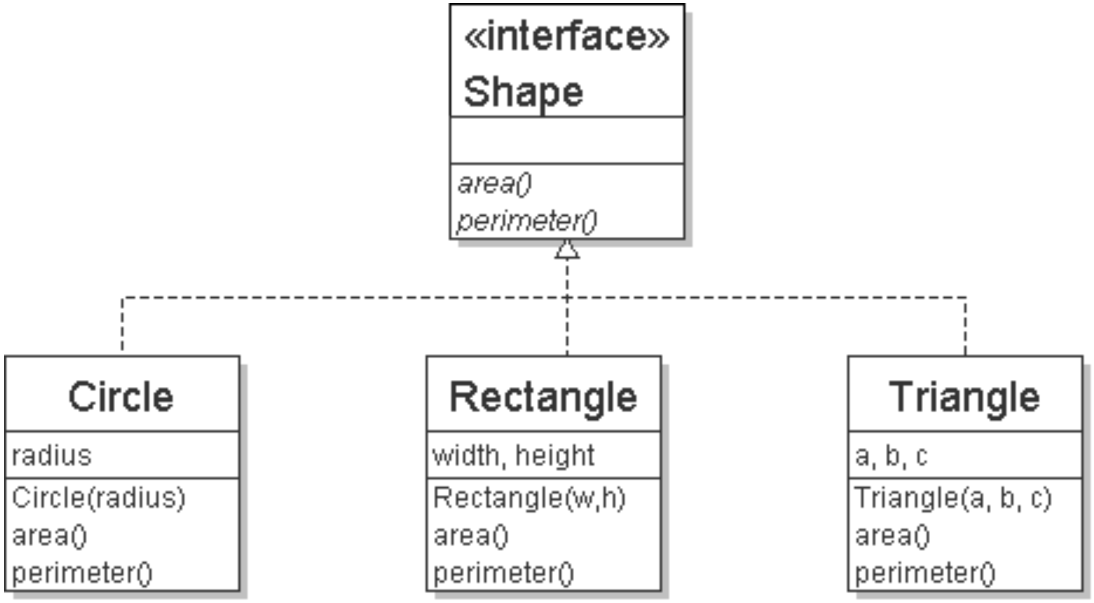
- Interfaces don't benefit the class so much as the *client.*
  - Interface's is-a relationship lets the client use polymorphism.

```
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.

```
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);
Shape s1=new Circle(5.0);
printInfo(circ);
printInfo(tri);
printInfo(rect);
printInfo(s1);
Shape[] shapes = {tri, s1, circ, rect};
```

Interface diagram



- Arrow goes up from class to interface(s) it implements.
  - There is a supertype-subtype relationship here;
    e.g., all Circles are Shapes, but not all Shapes are Circles.
  - This kind of picture is also called a *UML class diagram*.

Examples

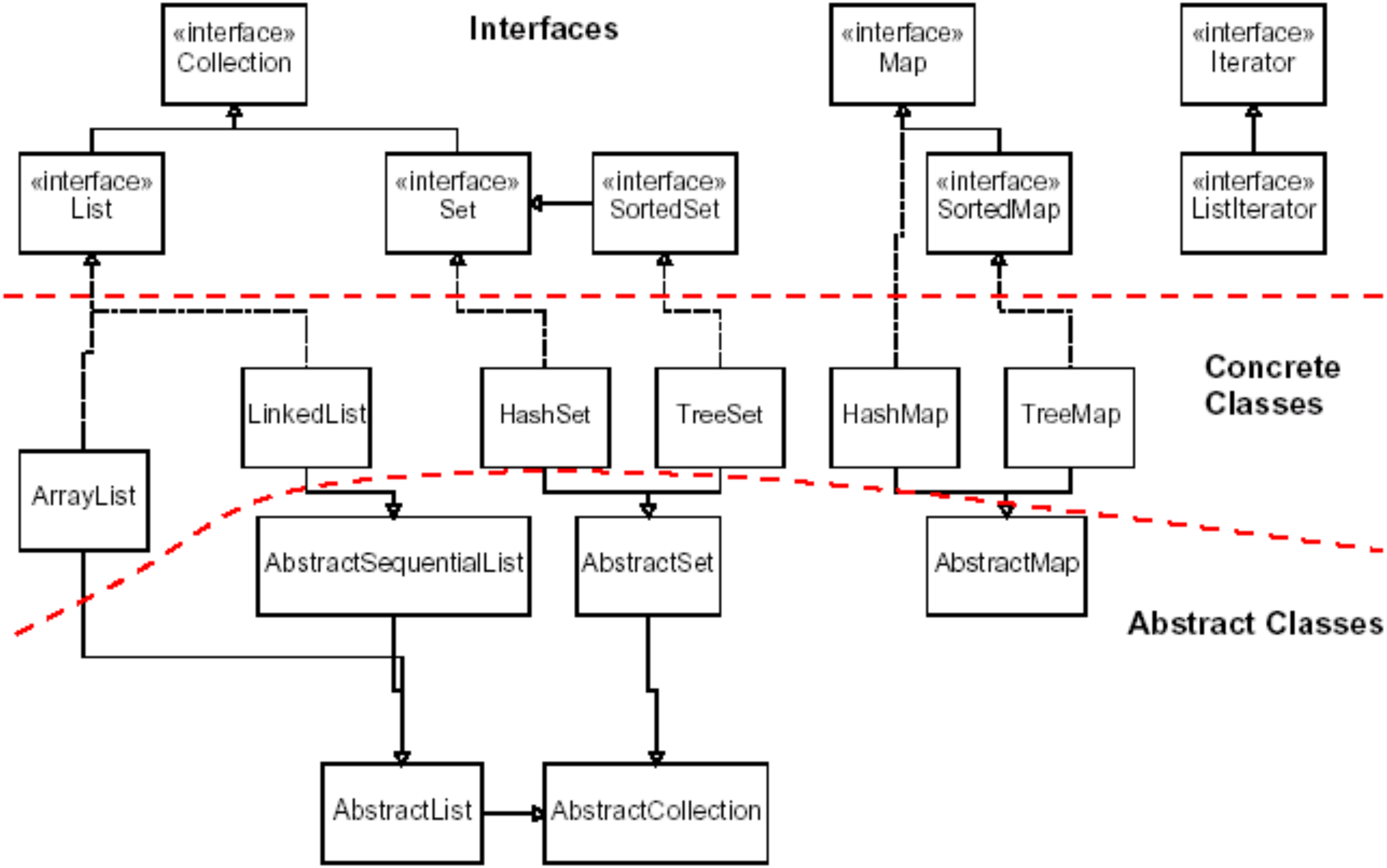
```
public static void main(){

  Shape a=new Shape();//error, can’t create Shape
                          //interface.
  Shape b=new Circle(); //ok
  Shape c=new Rectangle(); //ok
  Shape d=new Triangle(); //ok

  Shape[] shapeArray=new Shape[3]; //all null
  shapeArray[0]=b;
  shapeArray[1]=c;
  shapeArray[2]=d;

  ArrayList<Shape> list=new ArrayList<Shape>();
  list.add(a); list.add(b); list.add(c,1);
  list.remove(2);
  }
```

Java collections framework



Shape interface

Shape interface

The ArrayList class that we have been using implements the List interface from the Java collections framework.

```
public interface List<E> {
    void add(int index, E element);
    void add(E element);
    E get(int index);
    E remove(int index);
}

public class ArrayList<E> implements List<E> {
    // code to implement all of the abstract
    //methods from List<E>.
}
```

Shape interface

On the AP Exam, you’ll see the following construction of an arraylist.

```
List<String> myList = new ArrayList<String>();
List<Employee> empList= new ArrayList<Employee>();

myList.add(“Java”);
empList.add(new Employee());
```

Interface

- Some examples:
  1. For example, how does a driver of a car interact or interface with the engine/transmission to drive it?
     - –the interface of a car is the steering wheel, the brake and gas pedals and the shift. This interface is simply a list of methods that allow you to interact with your engine/transmission, etc.
  2. How does one access the applications on a smart TV?
     - the remote control is the interface that allows a person to interact with their TV.
     - the buttons on the remote are the methods of the interface.

Application Programming Interface

- an application programming interface(API) is a set of method definitions, protocols, and tools for building application software.
  - In the remote control example, the remote is an “application human interface”.

- Youtube, Google, and Facebook has API that allows programmers to interact with their software.
  - If you study Facebook’s API, you can write a program that allows someone to interact with facebook while using your program.
  - Or you can use Youtube’s public API to write a program that download, for example, the top 10 most viewed videos of the week.

Abstract Class

Abstract Class

- An abstract class is a class that is only partially implemented by the programmer. It must be declared **abstract**.
- An abstract class may or may not have abstract methods.
  - An abstract method is that is declared without an implementation.
  - -A class is usually declared abstract if one or more of its methods cannot be implemented and needs to be implemented in a child class.
- Abstract classes **cannot** be instantiated(created with new operator).
- An abstract class can both have instance variables and concrete(nonabstract) methods.
- An abstract class may or may not have constructors.

Abstract Class

- Abstract class can be subclassed.
- When an abstract class is subclassed, the subclass usually provides the implementations for all of the abstract methods.
- When a subclass doesn’t provide implementations, it must also be declared abstract.
- Polymorphism works with abstract classes as it does with concrete classes and interfaces.

Example

```
public abstract class Character
{
  private String name;

  public Character(String name)
  {
    this.name=name;
  }
  //concrete method
  public String getName()
  {
    return name;
  }
  //abstract method
  public abstract void drawCharacter();
}
```

A Subclass

```
public class Mario extends Character
{
  private int numLives;
  //constructors and other methods not shown.
  public void drawCharacter()
  {
    //must provide implementation to draw Mario
  }}
public class Bowser extends Character
{
  //constructors and other methods not shown.
  public void drawCharacter()
  {
    //must provide implementation to draw Bowser
  }}
```

Examples

```
public static void main(){

  //error, can’t create abstract object Character.
  Character a=new Character();

  Character b=new Mario();//ok
  Bowser c=new Bowser();//ok



  Character[] CharacterArray=new Shape[2];
  CharacterArray[0]=b;
  CharacterArray[1]=c;

  ArrayList<Character> list=new ArrayList<Character>();
  list.add(b); list.add(c);
}
```

Interface Vs Abstract Class

- Use an abstract class for an object that is application-specific but incomplete without its subclasses.
- Use an interface when its methods are suitable for your program but could be equally applicable in a variety of programs.

The classes that implement a given interface may represent objects that are vastly different. They, however, all have in common a capability or feature expressed in the methods of the interface. For example, an interface `FlyingObject` may have methods `fly` and `isFlying`. Some classes that implement `FlyingObject` could be `Bird, Airplane, Missile, Butterfly, Witch.`

Classes that inherit from an abstract(or concrete) class are similar. For example, all subclasses of Employee are “employees”.

Interface Vs Abstract Class

- An interface cannot provide implementations for any of its methods, whereas an abstract class can.
- An interface cannot contain instance variables, whereas an abstract class can.
- An interface can declare final static constants.
- It is not possible to create an interface object.
- It is not possible to create an abstract class object.

Interface Vs Abstract Class

- A class can inherit one one class but can implement multiple interfaces.
- A class can both inherit and implement interfaces at the same time. In this case, the extends clause must precede the implements clause.

```
public class Fraction extends Number implements
                        Comparable, Computable{
      …
}
```

Lab(CS50 IDE)

Write an interface called Movable which has four abstract methods:

```
public void moveUp();
public void moveDown();
public void moveLeft();
public void moveRight();
```

Lab 1

Write a class called MovablePoint which implements Movable. MovablePoint should have private member variables int x and y, one constructor to initialize x and y and an overridden toString method as well as the necessary implementations of the abstract methods from Movable.

Lab 1

Write another class called MovableCircle which also implements Movable. MovableCircle should have private members MovablePoint center and an integer radius, one constructor to initialize the center(with a x1 and y1) and the radius, an overridden toString and any necessary implementations of the abstract methods from Movable.

Lab 1

Write the driver class. Create two Movable objects. One should be a MovablePoint and the other a MovableCircle. Both of these objects must have Movable references. Move the objects and print out their coordinates.

Create an array of 2 Movable objects and add the above objects to it.

Create an arraylist of 2 Movable objects and add the above objects to it.

Lab 2

Modify Lab 1 at the end of the Polymorphism Lecture # 18 by

making the Shape class abstract.