CS 145

Chapter 9 – Polymorphism

- Interfaces

Interfaces

AKA : Abstract classes

Relatedness of types

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

- Certain operations that are common to all shapes.

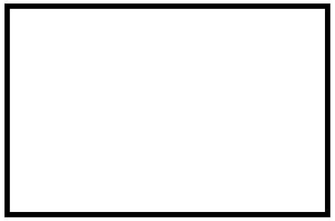
perimeter	- distance around the outside of the shape
area	- amount of 2D space occupied by the shape

- Every shape has them but computes them differently.

Shape area, perimeter

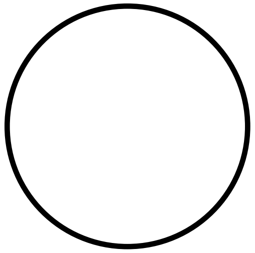
- Rectangle (as defined by width w and height h):

area	$= w\,h$
perimeter	$= 2w + 2h$



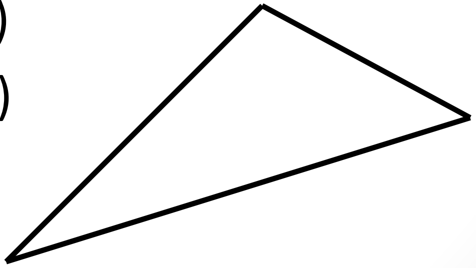
- Circle (as defined by radius r):

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

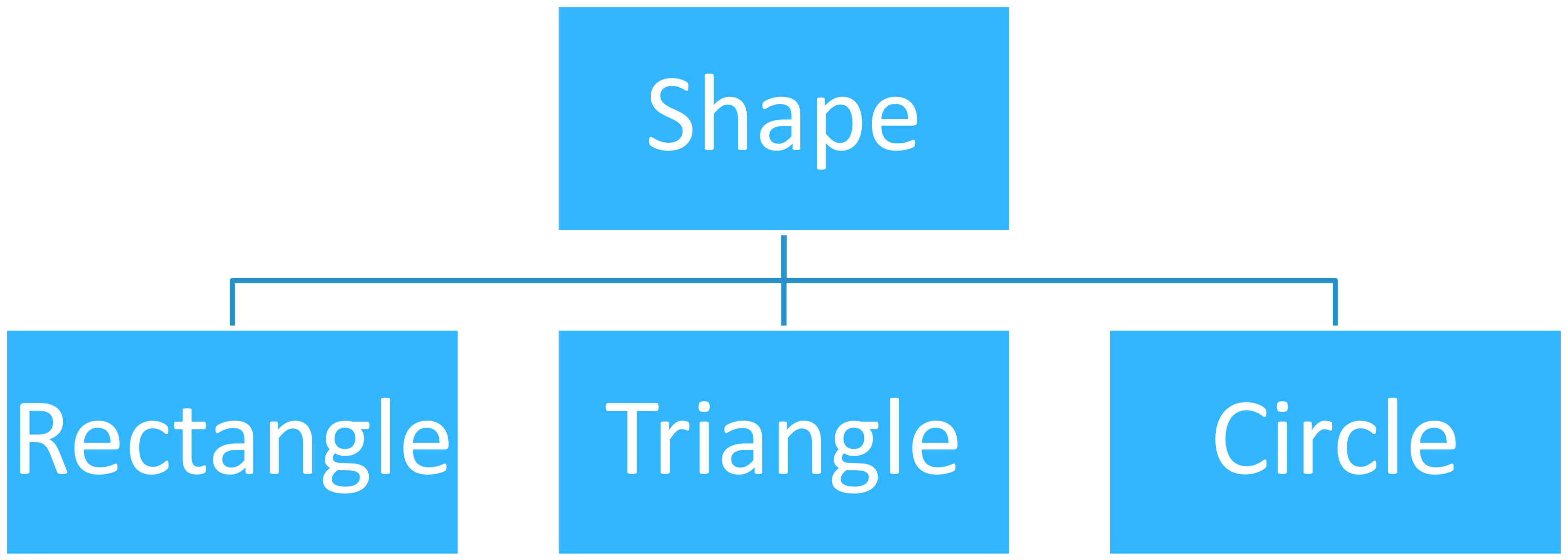


- Triangle (as defined by side lengths a, b, and c)

area	$= \surd(s\,(s - a)\,(s - b)\,(s - c))$
	where $s = \frac{1}{2}\,(a + b + c)$
perimeter	$= a + b + c$



Basic Idea



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.

But a question

- Would you ever actually make a "shape" object.

- You might have an array of shapes,
- A function that uses a shape

- But would you ever actually `new` a shape?

Interfaces

- **interface**: A list of methods that a class can implement.
  - 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`.
  - It is a type of class template.

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.

Implementing an interface

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

- 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 `Vehicle` but doesn't implement the `speed` and `setDirection` methods, it will not compile.

  - Example:

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

  - The compiler error message:

```
Banana.java:1: Banana is not abstract and
does not override abstract method speed() in
setDirection
public class Banana implements Vehicle {
           ^
```

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.)
- Note that there isn't actually the ability to instantiate a shape as the area() and perimeter() methods don't actually have any code.

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

```
// 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

- Interface's is-a relationship lets the client use polymorphism.

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

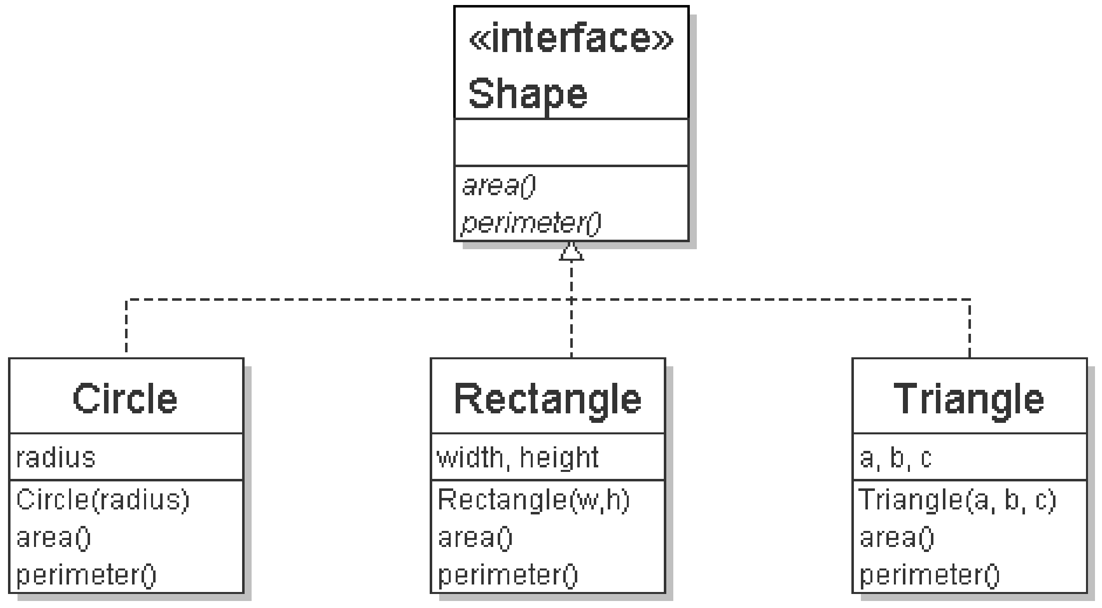
- Any object that implements the interface may be passed.

```
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);

printInfo(circ);
printInfo(tri);
printInfo(rect);

Shape[] shapes = {tri, 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*.
    - Universal Modeling Language

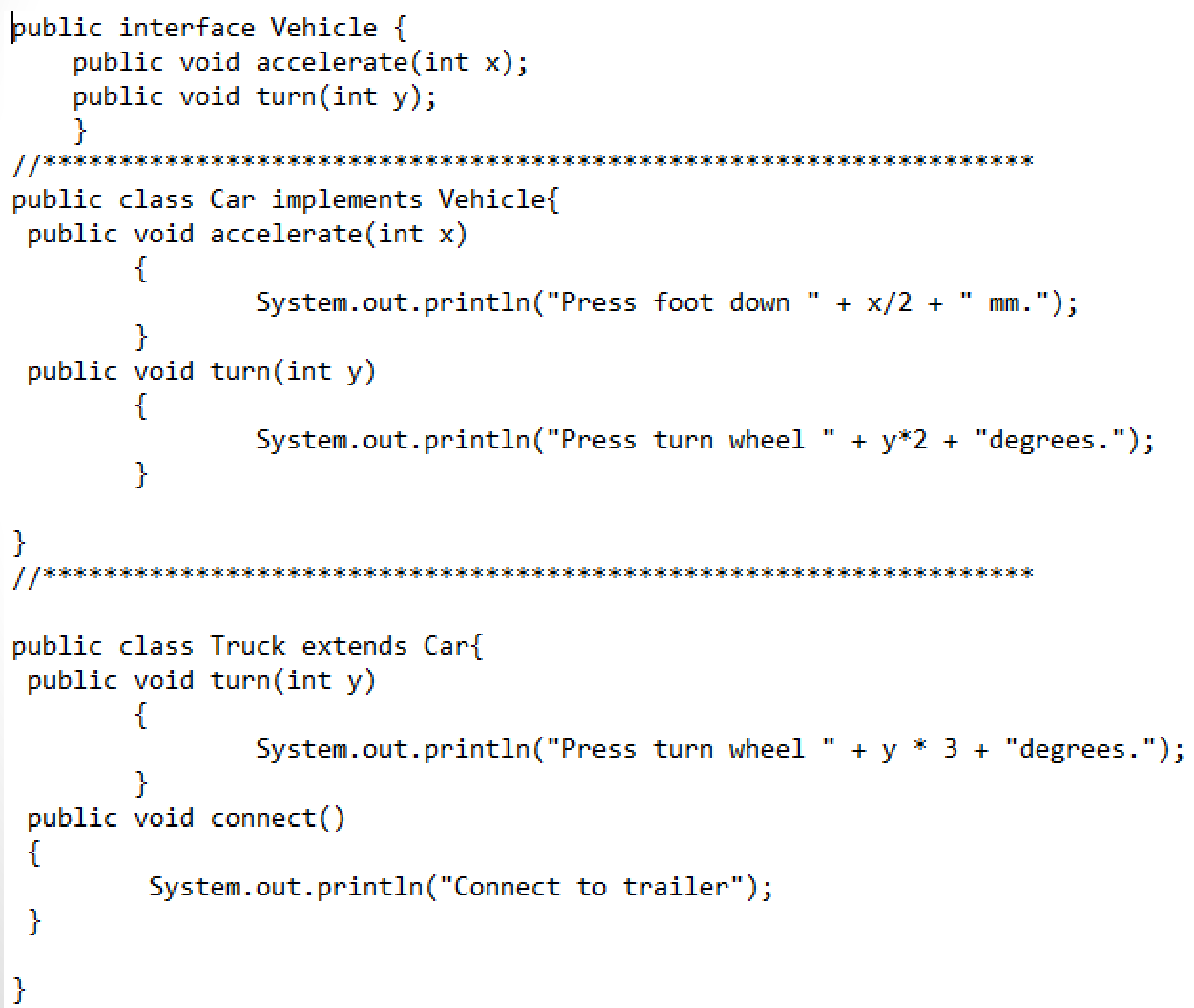
What is the difference

- A class
  - Defines who you are, what your actions will be
  - States how you plan on performing your actions.
  - Can only come from one superclass

- An interface
  - Defines what roles you can perform.
  - A set of promises
    - Things that you promise that you can take care of.
  - You can fulfill multiple roles, i.e. be part of many interfaces.
    - Class can implement more than one interface

Lets Do some practice

Using the Vehicle interface and the provided documentation



```
public interface Vehicle {
    public void accelerate(int x);
    public void turn(int y);
    }
//*****************************************************************
public class Car implements Vehicle{
 public void accelerate(int x)
        {
                System.out.println("Press foot down " + x/2 + " mm.");
        }
 public void turn(int y)
        {
                System.out.println("Press turn wheel " + y*2 + "degrees.");
        }

}
//*****************************************************************

public class Truck extends Car{
 public void turn(int y)
        {
                System.out.println("Press turn wheel " + y * 3 + "degrees.");
        }
 public void connect()
 {
         System.out.println("Connect to trailer");
 }

}
```

Lets look at an example

- Suppose the following variables are defined:

    - `Vehicle[] list = new Vehicle[4];`
        - `list[0] = new Car();`
        - `list[1] = new Truck();`
        - `list[2] = new Hybrid();`
        - `list[3] = new Railroad();`

What is the output?

- Suppose the following variables are defined:

  - `Vehicle[] list = new Vehicle[4];`
    - `list[0] = new Car();`
    - `list[1] = new Truck();`
    - `list[2] = new Hybrid();`
    - `list[3] = new Railroad();`

A. `list[0].accelerate(10);`

Press foot down 5 mm.

B. `list[1].accelerate(10);`

Press foot down 5 mm.

C. `list[2].accelerate(10);`

Press foot down 2 mm.

D. `list[3].accelerate(10);`

Put in 10 coal

E. `list[0].turn(30);`

Press turn wheel 60 degrees.

F. `list[1].turn(30);`

Press turn wheel 90 degrees.

G. `list[2].turn(30);`

Press turn wheel 60 degrees.

H. `list[3].turn(30);`

Please don’t

What is the output?

- `Vehicle[] list = new Vehicle[4];`
  - `list[0] = new Car();`
  - `list[1] = new Truck();`
  - `list[2] = new Hybrid();`
  - `list[3] = new Railroad();`

A. `((Car)list[0]).accelerate(20);` Press foot down 10 mm.

B. `((Car)list[1]).accelerate(20);` Press foot down 10 mm.

C. `((Car)list[2]).accelerate(20);` Press foot down 4 mm.

D. `((Car)list[3]).accelerate(20);`

Error: Railroad can not be cast to Car

What is the output?

- `Vehicle[] list = new Vehicle[4];`
  - `list[0] = new Car();`
  - `list[1] = new Truck();`
  - `list[2] = new Hybrid();`
  - `list[3] = new Railroad();`

A. `((Truck)list[0]).turn(10);`

Error: Car can not be cast to Truck

B. `((Truck)list[1]).turn(10);`

Press turn wheel 30 degrees.

C. `((Truck)list[2]).turn(10);`

Error: Hybrid can not be cast to Truck

D. `((Truck)list[3]).turn(10);`

Error: Railroad can not be cast to Truck

What is the output?

- `Vehicle[] list = new Vehicle[4];`
  - `list[0] = new Car();`
  - `list[1] = new Truck();`
  - `list[2] = new Hybrid();`
  - `list[3] = new Railroad();`

A. `((Vehicle)list[1]).connect();`

Error: connect() is not defined in Vehicle

B. `((Car)list[1]).connect();`

Error: connect() is not defined in Car

C. `((Truck)list[2]).connect();`

Error: Hybrid can not be cast to Truck

D. `((Truck)list[3]).connect();`

Error: Railroad can not be cast to Truck

JGRASP HINTS

jGrasp Projects

- Project Files

jGrasp Debugger and Canvas

- Example