

Reusing Classes

Hendrik Speleers

Reusing Classes

- **Overview**
 - Composition
 - Inheritance
 - Polymorphism
 - Method overloading vs. overriding
 - Visibility of variables and methods
 - Specification of a contract
 - Abstract classes, interfaces

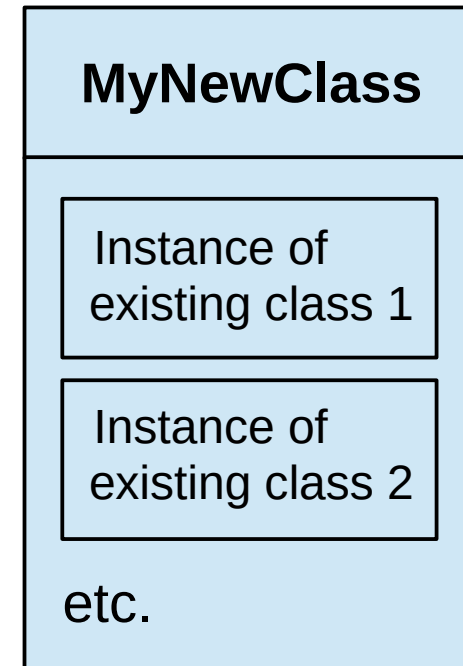
Reusing Classes

- **Software development**
 - One of the holy grails of OOP: reusing classes
 - When you need a class, you can
 - Get the perfect one off the shelf
 - e.g., library, GUI builder environment» one extreme
 - Write it completely from scratch» other extreme
 - Reuse an existing class with **composition**
 - Reuse an existing class or class framework with **inheritance**
 - A good class design is important

Reusing Classes

- **Composition**
 - Simplest way to reuse existing code
 - Instances of existing classes inside a new class
 - Flexibility: can change objects at runtime
 - A “has-a” relationship between classes

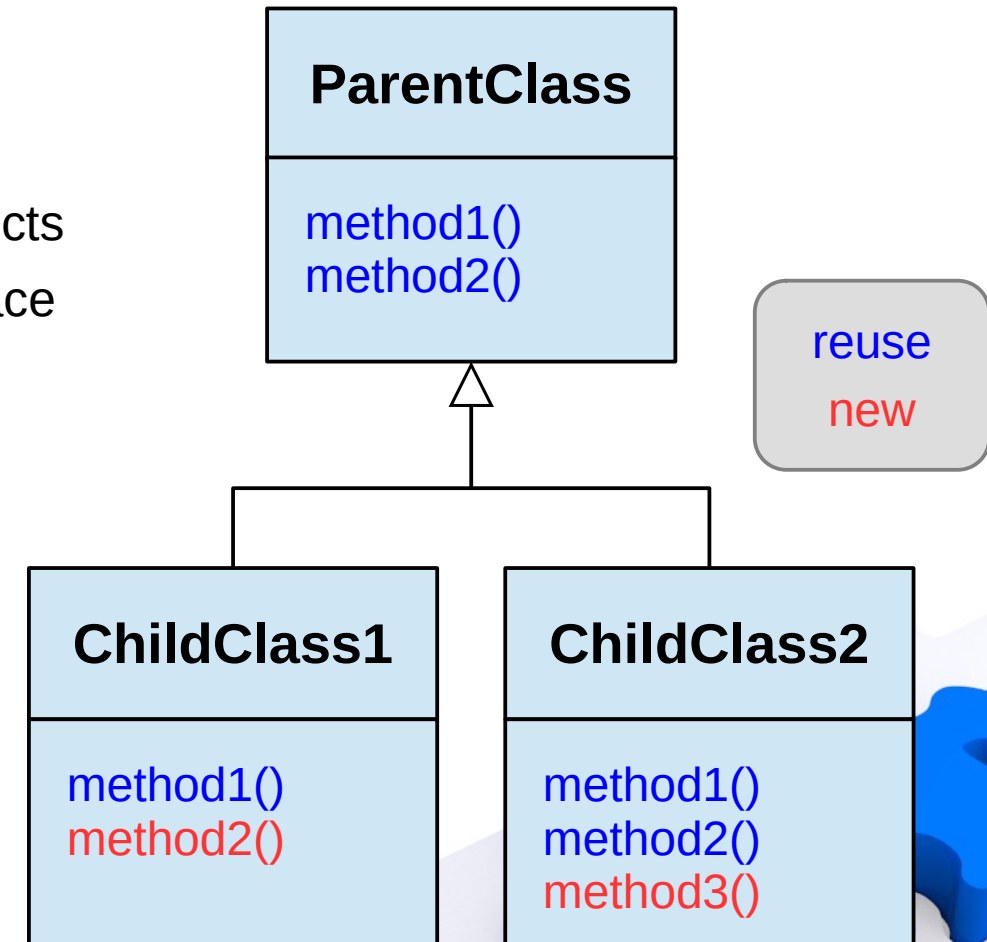
```
class MyNewClass {  
    Foo x = new Foo();  
    Bar x = new Bar();  
    Baz x = new Baz();  
    ...  
}
```



Reusing Classes

- **Inheritance**

- Pure inheritance
 - Interface duplication for interchangeable objects
 - Redefinition of methods with the same interface
 - An “is-a” relationship between classes
- Extension inheritance
 - Inheritance to extend the interface
 - Additional variables and methods
 - An “is-like-a” relationship between classes
- Single inheritance (e.g., Java)
vs. Multiple inheritance (e.g., C++)



Reusing Classes

- **Inheritance**

- Terminology
 - Parent, superclass, base class, ...
 - Child, subclass, derived class, ...
- Class **Object** is the root of the class hierarchy
 - Every class has Object as a superclass
- A class can have at most one parent but of course more ancestors
- Creating a subclass by the **extends** keyword

```
class <class name> extends <class name>
```

Reusing Classes

- **Inheritance**
 - Example: the classes Shape, Circle, Rectangle

Shape	Circle	Rectangle
<code>getCenter()</code> <code>moveCenterTo()</code> <code>moveCenterBy()</code> <code>toString()</code> <code>getArea()</code> <code>getPerimeter()</code>	<code>getCenter()</code> <code>moveCenterTo()</code> <code>moveCenterBy()</code> <code>toString()</code> <code>getArea()</code> <code>getPerimeter()</code> <code>getRadius()</code> <code>setRadius()</code> ...	<code>getCenter()</code> <code>moveCenterTo()</code> <code>moveCenterBy()</code> <code>toString()</code> <code>getArea()</code> <code>getPerimeter()</code> <code>getWidth()</code> <code>getHeight()</code> ...

a Circle is a Shape
a Rectangle is a Shape

Reusing Classes

- Inheritance
 - Example: the classes Shape, Circle, Rectangle

```
public class Shape {  
    protected Point center;  
  
    public Shape() { center = new Point(); }  
    public Shape(int x, int y) { center = new Point(x, y); }  
    public Point getCenter() { return center; }  
    public void moveCenterTo(int x, int y) {  
        center.setLocation(x, y); }  
    public double getArea() { return 0.0; }  
    ...  
}
```

Reusing Classes

- Inheritance
 - Example: the classes Shape, Circle, Rectangle

```
public class Point {  
    protected int coordX, coordY;  
  
    public Point() { setLocation(0, 0); }  
    public Point(int x, int y) { setLocation(x, y); }  
    public void setLocation(int x, int y) {  
        coordX = x; coordY = y; }  
    public int getX() { return coordX; }  
    public int getY() { return coordY; }  
    ...  
}
```

Reusing Classes

- Inheritance

- Example: the classes Shape, Circle, Rectangle

```
public class Circle extends Shape {  
    protected double radius;  
  
    public Circle() {  
        center = new Point(); radius = 1.0; }  
    public Circle(int x, int y, double r) {  
        center = new Point(x, y); radius = r; }  
    public double getRadius() { return radius; }  
    public double getArea() {  
        return Math.PI * radius * radius; }  
    ...  
}
```

inherits from Shape the methods
getCenter, moveCenterTo, ...
but NOT the constructors

initialize class variables
of class and superclass

Reusing Classes

- **Polymorphism**

- Upcasting

- A variable of class X can refer to objects of class X or any of its subclasses

```
Shape shape;  
shape = new Shape();  
shape = new Circle(1, 0, 2.5);
```

- Separation of interface from implementation

- Substitutability
 - Extensibility

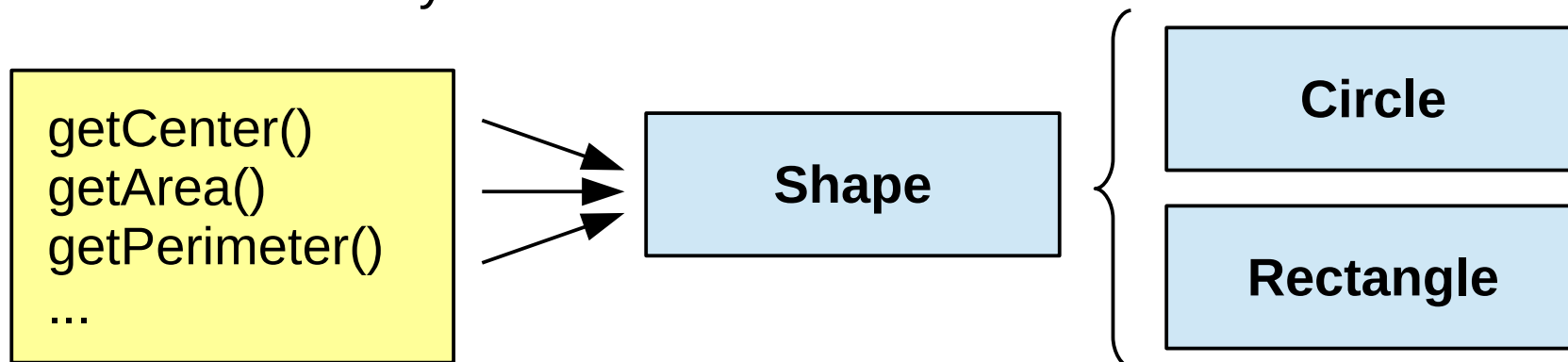
Reusing Classes

- **Polymorphism**

- Upcasting

- A variable of class X can refer to objects of class X or any of its subclasses

- Substitutability



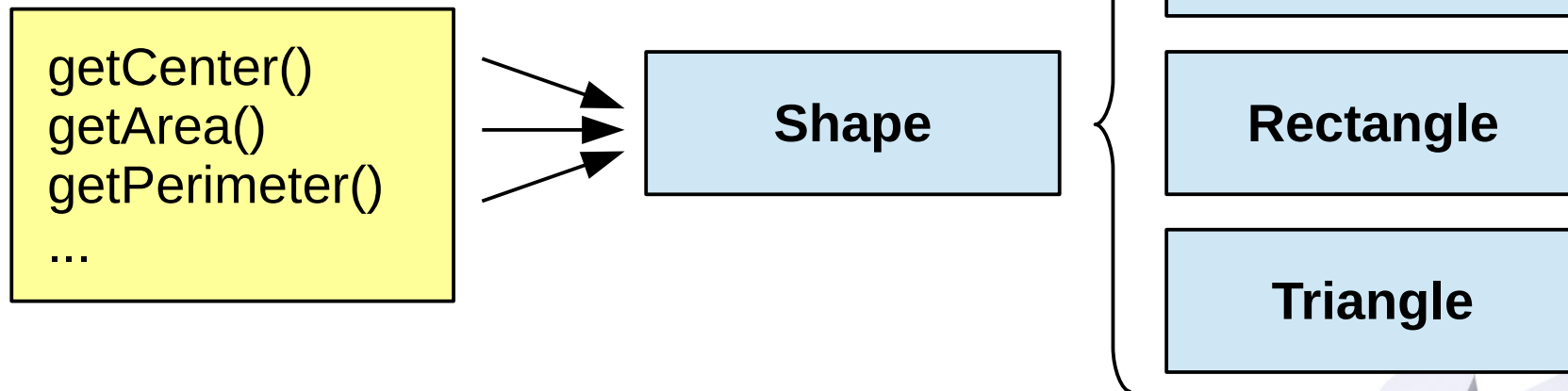
Reusing Classes

- **Polymorphism**

- Upcasting

- A variable of class X can refer to objects of class X or any of its subclasses

- Extensibility



Reusing Classes

- **Method overriding versus overloading**
 - Method binding = connecting a method call to a method body
 - Method overriding
 - Redefinition of a (parent) method with exactly the same interface
 - Same name, same number of parameters, same type of parameters
 - Dynamic binding (at run-time)
 - Method overloading
 - Redefinition of a method with a similar interface
 - Same name, but different set of parameters (number and/or type)
 - Static binding (at compile time)

Reusing Classes

- **Visibility of variables and methods**
 - Recall: **public** – **private** – **protected**
 - Public: visible to the world (everybody outside and inside the class)
 - Private: visible only to the class
 - Protected: visible to the package and all subclasses
 - Default (friendly), no keyword: visible to the package
 - Use **public** or **protected** to be accessible to subclasses
 - Good access strategy
 - Limit direct access to variables
 - Control access via Getters/Setters (methods)

Reusing Classes

- **Visibility of variables and methods**
 - Name conflicts of variables inside methods
 - Priority: local variables > parameters > class variables > parent class variables
 - Name conflicts of methods
 - Priority: methods > parent methods
 - Name conflicts can be avoided by a self-referencing pointer
 - The **this** keyword
 - The **super** keyword

Reusing Classes

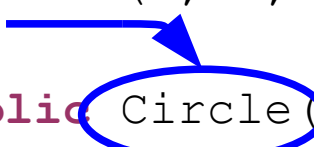
- **Visibility of variables and methods**
 - The **this** keyword is a reference to the current object
 - Referring to class variables
 - Referring to methods inside class (no extra functionality)

```
public class Circle extends Shape {  
    protected double radius;  
  
    public void setRadius(double radius) {  
        this.radius = radius;  
    }  
}
```

Reusing Classes

- **Visibility of variables and methods**
 - Constructor delegation using the **this** keyword
 - This guarantees consistent initialization
 - It has to be the first statement in the constructor

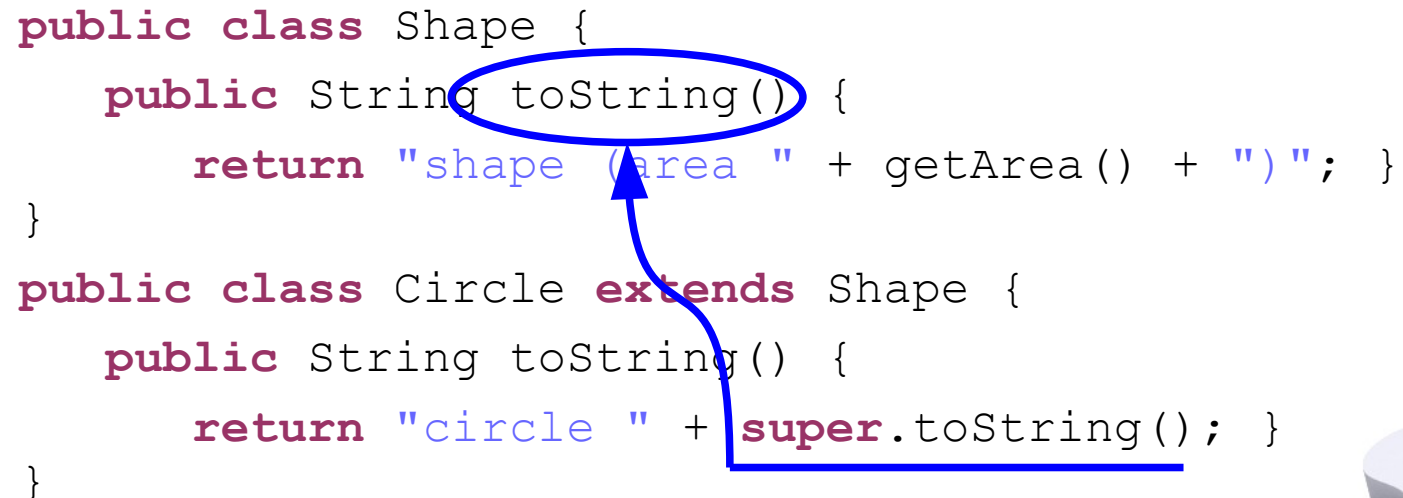
```
public class Circle extends Shape {  
    public Circle() {  
        this(0, 0, 1.0);  
    }  
    public Circle(int x, int y, double r) {  
        center = new Point(x, y); radius = r;  
    }  
}
```



Reusing Classes

- **Visibility of variables and methods**
 - The **super** keyword is a reference to the current parent object
 - Referring to parent class variables
 - Referring to parent methods

```
public class Shape {  
    public String toString() {  
        return "shape (area " + getArea() + ") ";  
    }  
    public class Circle extends Shape {  
        public String toString() {  
            return "circle " + super.toString(); }  
    }  
}
```



Reusing Classes

- **Visibility of variables and methods**
 - Constructor delegation using the **super** keyword
 - Constructors are not inherited; implicit default call super() if available
 - It has to be the first statement in the constructor

```
public class Shape {  
    public Shape(int x, int y) {  
        center = new Point(x, y);  
    }  
    public class Circle extends Shape {  
        public Circle(int x, int y, double r) {  
            super(x, y); radius = r;  
        }  
    }  
}
```

Reusing Classes

- **Specification of a contract**
 - Separation of interface from implementation
 - Abstract classes using the **abstract** keyword
 - an “is-a” or “is-like-a” relationship
 - One or more methods in the class have no implementation
 - Pure interfaces using the **interface** keyword
 - a “can-do” relationship
 - No method has an implementation
 - A class can implement
 - One abstract class via the **extends** keyword
 - Multiple interfaces via the **implements** keyword



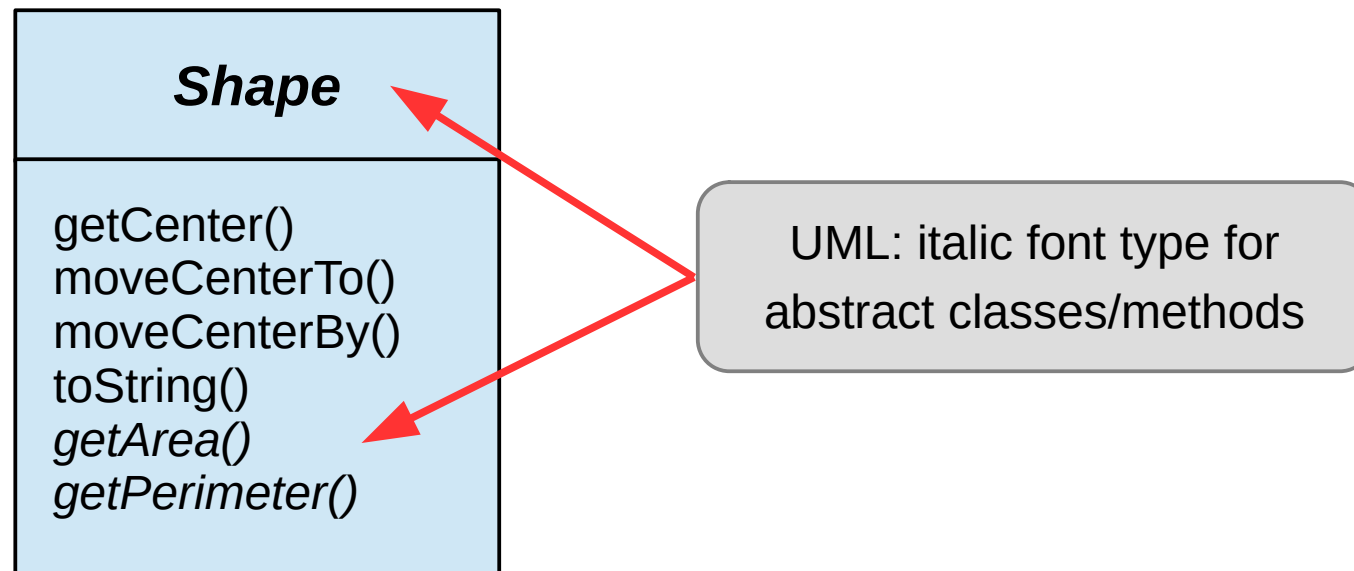
Reusing Classes

- **Specification of a contract**
 - Abstract classes
 - Postponing implementation till where it makes sense
 - Creating objects can only through (not abstract) subclasses

```
public abstract class Shape {  
    ...  
    public abstract double getArea();  
}  
public class Circle extends Shape {  
    ...  
    public double getArea() {  
        return Math.PI * radius * radius;  
    }  
}
```

Reusing Classes

- **Specification of a contract**
 - Abstract classes
 - Postponing implementation till where it makes sense
 - Creating objects can only through (not abstract) subclasses



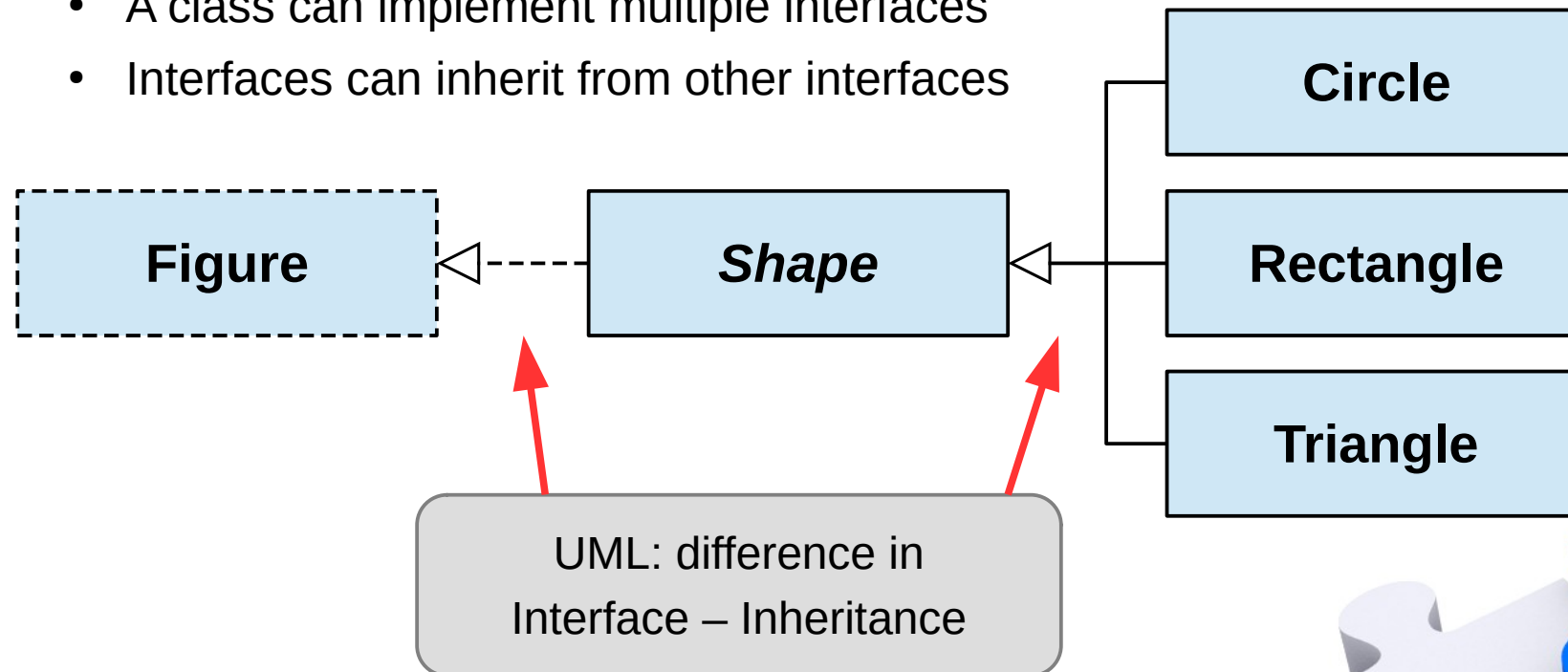
Reusing Classes

- **Specification of a contract**
 - Pure interfaces
 - A class can implement multiple interfaces
 - Interfaces can inherit from other interfaces

```
public interface Figure {  
    public void moveCenterTo(int x, int y);  
}  
  
public class Shape implements Figure {  
    ...  
    public void moveCenterTo(int x, int y) {  
        center.setLocation(x, y);  
    }  
}
```

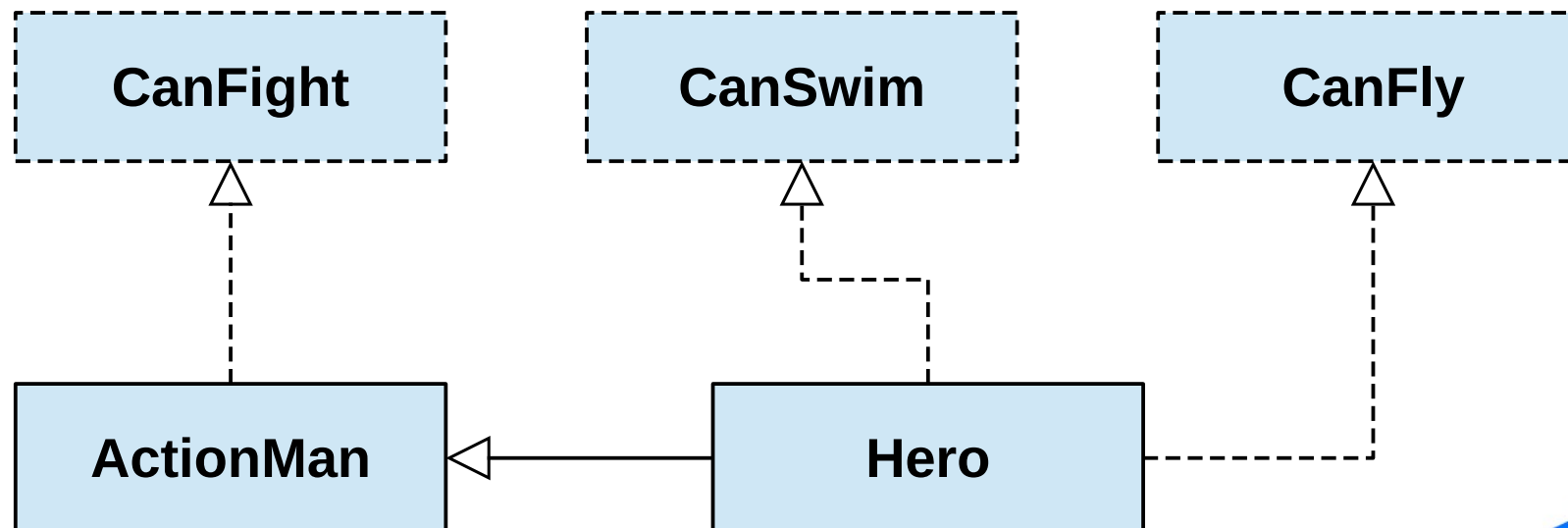
Reusing Classes

- **Specification of a contract**
 - Pure interfaces
 - A class can implement multiple interfaces
 - Interfaces can inherit from other interfaces



Reusing Classes

- **Specification of a contract**
 - Example: Adventure



Reusing Classes

- Specification of a contract
 - Example: Adventure

```
public interface CanFight {  
    public void fight();  
}  
  
public interface CanSwim {  
    public void swim();  
}  
  
public interface CanFly {  
    public void fly();  
}
```

Reusing Classes

- Specification of a contract
 - Example: Adventure

```
public class ActionMan implements CanFight {  
    public void fight() { perform("Fight!"); }  
    protected void perform(String action) {  
        System.out.println(action); }  
}  
  
public class Hero extends ActionMan  
    implements CanSwim, CanFly {  
    public void swim() { perform("Swim!"); }  
    public void fly() { perform("Fly!"); }  
}
```

Reusing Classes

- Specification of a contract
 - Example: Adventure

```
public class Adventure {  
    public static void t(CanFight x) { x.fight(); }  
    public static void u(CanSwim x) { x.swim(); }  
    public static void v(CanFly x) { x.fly(); }  
    public static void main(String[] args) {  
        Hero h = new Hero();  
        t(h); // Treat it as a CanFight  
        u(h); // Treat it as a CanSwim  
        v(h); // Treat it as a CanFly  
    }  
}
```

Reusing Classes

- **Specification of a contract**
 - Pure interfaces
 - Class variables are automatically **static** and **final**
 - An interface is convenient to create groups of constants (like **enum**)

```
public interface Months {  
    int JANUARY = 1, FEBRUARY = 2, MARCH = 3,  
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,  
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,  
        NOVEMBER = 11, DECEMBER = 12;  
}
```

Reusing Classes

- **Example: interfaces**
 - Look at the file InterFaceEx.java
 - Make a class diagram of all the involved classes/interfaces
 - Predict the output of the main method