COM6516
Object Oriented Programming and Software Design

The contents of this module has been developed by Adam Funk, Kirill Bogdanov, Mark Stevenson, Richard Clayton and Heidi Christensen

Practical 4

Inheritance, Polymorphism, Abstract classes and interfaces

- Override
- Polymorphism
- Abstract classes and Interfaces

Override

Superclass:

```
public class Person {
    …
    public String toString() {
        return this.name + "  DOB=" + this.dateOfBirth;
    }
    …
}
```

Subclass:

```
public class Student extends Person {
    …
    @Override
    public String toString() {
        return this.getName() + " DOB=" + this.getDateOfBirth() + " Course=“
                + this.course;
    }
    …
}
```

Override

Superclass:

```
public class Person {
    …
    public String toString() {
        return this.name + "  DOB=" + this.dateOfBirth;
    }
    …
}
```

Subclass (alternative):

```
public class Student extends Person {
    …
    @Override
    public String toString() {
        return super.toString() + " Course=" + this.course;

    }
    …
}
```

Override

What is the advantage of using the `@Override` annotation?

- compile-time safeguard against a common programming mistake
- easier understanding of code

Polymorphism

Legal statements:

```
Cow daisy = new Cow();
Animal daisy = new Cow();
```

- a cow **is-an** animal
- `Cow` (subclass) has the functionality of `Animal` (superclass)

Not legal:

```
Cow daisy = new Animal();
```

- an animal **is-not-a** cow
- `Cow` may have additional functionality that is not implemented for `Animal`

Polymorphism

How does the compiler know which methods to call?

- **Method candidates:** compiler finds all candidates to be called based on the class and the method name
- **Overload resolution:** the compiler looks at the supplied parameters and chooses the method that matches
- **Static binding:** for a constructor, or private, static or final methods, the compiler knows precisely which method to call
- **Dynamic binding:** otherwise the exact implementation of a method is determined at the run-time based on both the operation and the object
- **Method table:** this table lists all method signatures and the actual methods to be called

.

Abstract classes

Concrete `Animal` class:

```
public class Animal {
    public void talk() {
        System.out.println("Animals can't talk");
    }
}
```

Abstract `Animal` class:

```
public abstract class Animal {
    public void talk() {
        System.out.println("Animals can't talk");
    }
}
```

or

```
public abstract class Animal {
    public abstract void talk();
}
```

Abstract classes

The abstract `Animal` class cannot be instantiated:

```
public class AnimalTest {
    public static void main(String[] args) {
        Cow daisy = new Cow();
        Pig wilbur = new Pig();
        Sheep shaun = new Sheep();
        Animal animal = new Animal();
        …
    }
}
```

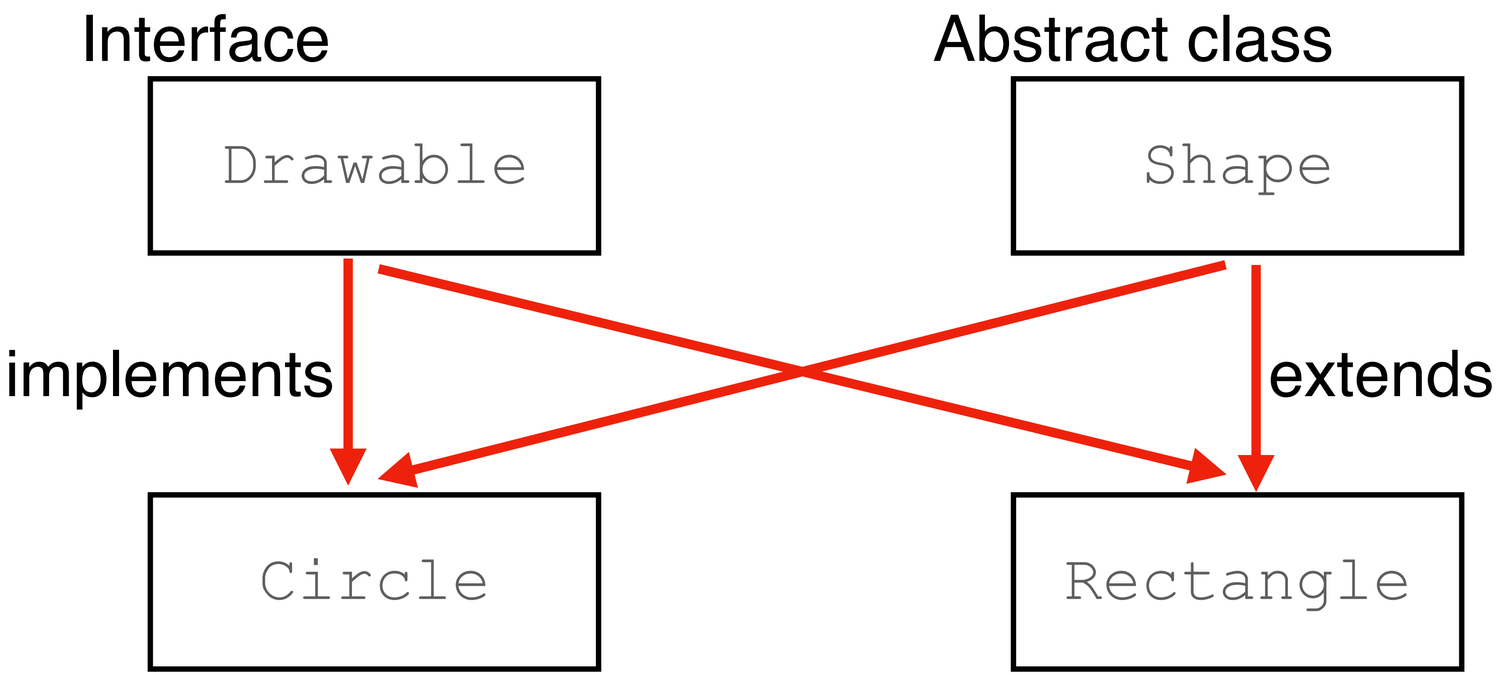
 compile time error

Abstract classes

Is it sensible to define the `Animal` class as abstract?

- It can represent different kinds of animals
- All such animals have common attributes, e.g., weight, #legs, …
- They also have some different attributes, e.g., fly, swim, …
- They have common methods, e.g., reporting weight, finding if they fly or swim, which may be different for each kind of animal

Abstract Classes and Interfaces



Abstract Classes and Interfaces

Interface:

```
import sheffield.*;
public interface Drawable {
    public abstract void draw(EasyGraphics g);
}
```

Abstract superclass:

```
public abstract class Shape {
    …

//  public abstract void draw(EasyGraphics g);

    …
}
```

Abstract Classes and Interfaces

Subclasses:

```
public class Circle extends Shape implements Drawable {
    …
}

public class Rectangle extends Shape implements Drawable {
    …
}
```

Test class:

```
public class ShapeDemo {
    public static void main(String[] args) {
        …
//      Shape[] list = new Shape[5];
        Drawable[] list = new Drawable[5];
        …
    }
}
```

Abstract Classes and Interfaces

Recall that Interfaces are not classes and cannot be instantiated:

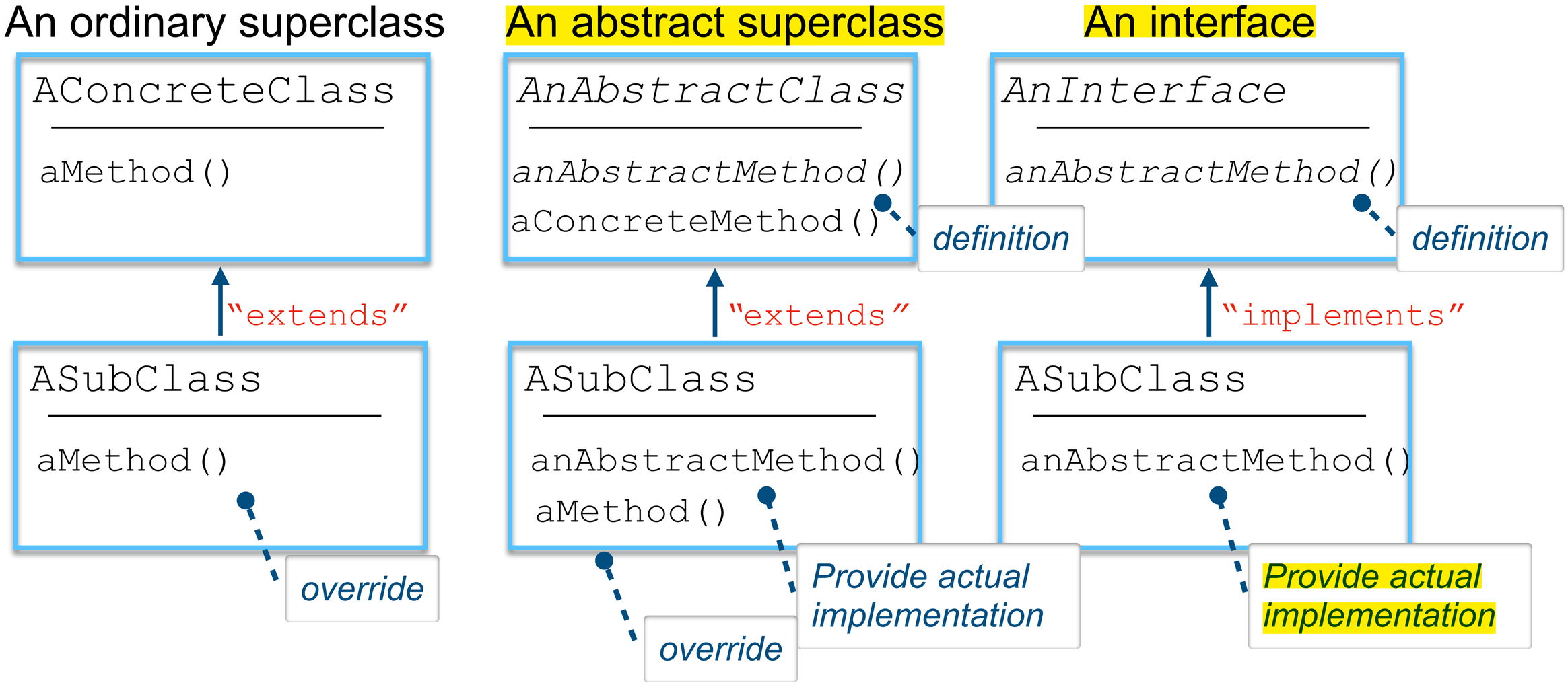
```
Drawable list = new Drawable(…);  // error
```

You can declare a variable within an Interface type if it refers to an object that implements the Interface — the following is legal if `Circle` implements `Drawable`:

```
Drawable list = new Circle(…);
```

- **Interface** is like a ‘face’ for an object — any object that has that **Interface** can be used methods in the **Interface**
- **Abstract Class** provides a partial implementation — a good example is an object that is declared to implement an **Interface** but does not include implementation of all the methods
- `Shape` provides a partial implementation and `Drawable` provides a 'face'

Concrete classes vs Abstract classes vs Interfaces



https://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

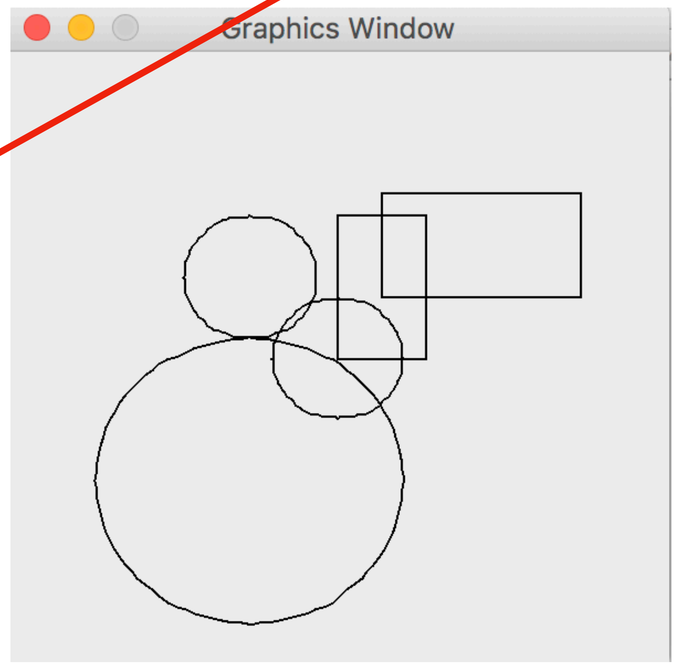
What if ‘normal’ inheritance isn’t appropriate?

Abstract classes:

Allows inheritance, but you can’t make instances of the abstract class itself. When you want to inherit from a *super class* (e.g., so you can exploit polymorphism), but when having an instance is not appropriate

Advantages of abstract `Shape` class: 1) we could still inherit; 2) we didn’t have problems with objects of type `Shape`; 3) and we could still explore polymorphism

```
import sheffield.*;
public class ShapeDemo {
  Public attic void main(String[] args) {
    EasyGraphics g = new EasyGraphics(300,300,15050);
    Shape[] list = new Shape[5];
    // fill the array with shapes
    list[0] = new Rectangle(20,20,40,70);
    list[1] = new Circle (-40,-60,70);
    list[2] = new Triangle (-40,40,30);
    list[3] = new Rectangle (20,50,90,50);
    list[4] = new Circle(0,0,30);
    // now update the display
    for (int i=0; i<5; i++) {
      list[i].draw(g);
    }
}
```



What if you want to inherit from multiple classes?

Interfaces:

In Java, an *interface* is another way of achieving abstraction. It describes what a class does, but not how it does it. An interface is essentially a set of requirements for a class. It's a complete abstract class; think of it as a group of related methods with empty bodies.

Why and when to use **Interfaces**?

1) To achieve security - hide certain details and only show the important details of an object (*interface*).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma.

3) Conveniently convert from UML