JC2002 Java Programming

Day 4: Abstract classes and interfaces (AI, CS)

Friday, 3 November

JC2002 Java Programming

Day 4, Session 1: Abstract classes

References and learning objectives

- Today's sessions are mostly based on:
    - Evans, B. and Flanagan, D., 2018. ***Java in a Nutshell: A Desktop Quick Reference***, 7th edition. O'Reilly Media.
    - Deitel, H., 2018. ***Java How to Program, Early Objects***, *Global Edition*, 11th Edition. Pearson.
- After today's session, you should be able to:
    - Use abstract classes and interfaces in your Java programs
    - Design appropriate class hierarchies with abstract classes and interfaces

Abstract classes

- *Abstract classes* are classes you cannot instantiate as objects
  - Used only as superclasses in inheritance hierarchies, so they are sometimes called *abstract superclasses*
  - Cannot be used to instantiate objects—abstract classes are *incomplete*
  - Subclasses must declare the “missing pieces” to become “concrete” classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract
- An abstract class provides a superclass from which other classes can inherit and thus share a common design

Abstract vs. concrete classes

- Classes that can be used to instantiate objects are *concrete classes*
  - Such classes provide implementations of every method they declare (some of the implementations can be inherited)
- Abstract superclasses are too general to create real objects—they specify only what is common among subclasses
- Concrete classes provide the specifics that make it reasonable to instantiate objects
- Not all hierarchies contain abstract classes

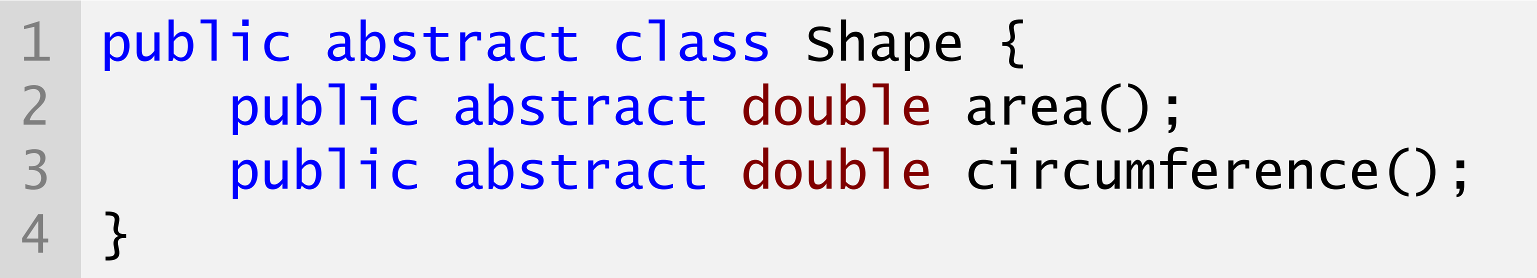
Declaring abstract classes

- You make a class abstract by declaring it with keyword **`abstract`**
- An abstract class normally contains one or more *abstract methods*
  - An abstract method is defined with keyword `abstract`, e.g.:

```
public abstract void draw(); // abstract method
```

  - Abstract methods do not provide implementations
- A class that contains abstract methods must be an abstract class even if that class contains some concrete (nonabstract) methods
- Each concrete subclass of an abstract superclass also must provide concrete implementations of the superclass's abstract methods

Example of defining an abstract class



```java
public abstract class Shape {
    public abstract double area();
    public abstract double circumference();
}
```

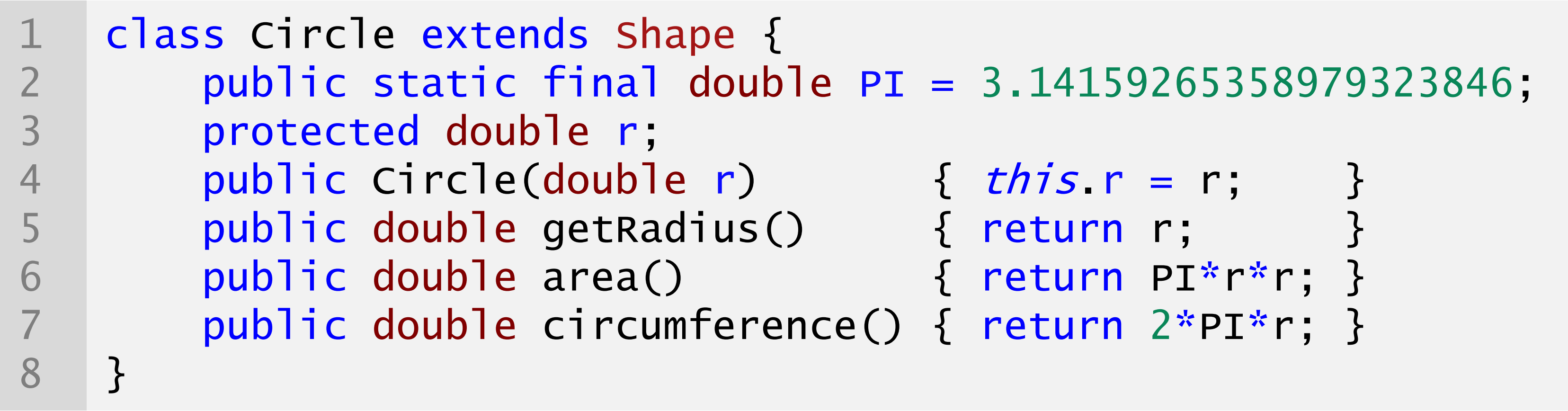
Example of defining an abstract class

```java
public abstract class Shape {
    public abstract double area();
    public abstract double circumference();
}
```

Abstract methods:
note semicolon ; instead of body { }

Note that public class must be in its own java file!

Example of extending an abstract class (1)



```java
class Circle extends Shape {
    public static final double PI = 3.14159265358979323846;
    protected double r;
    public Circle(double r)       { this.r = r;    }
    public double getRadius()     { return r;      }
    public double area()          { return PI*r*r; }
    public double circumference() { return 2*PI*r; }
}
```

Example of extending an abstract class (2)

```
class Rectangle extends Shape {
    protected double w, h;
    public Rectangle(double w, double h) {
        this.w = w;
        this.h = h;
    }
    public double getWidth()      { return w;       }
    public double getHeight()     { return h;       }
    public double area()          { return w*h;     }
    public double circumference() { return 2*(w+h); }
}
```

Example of testing inherited classes

```java
class TestShape {
    public static void main(String[] args) {
        Shape shape;
        shape = new Circle(5);
        System.out.println("Area: " + shape.area());
        shape = new Rectangle(5,10);
        System.out.println("Area: " + shape.area());
    }
}
```

```
Area: 78.53981633974483
Area: 50.0
```

Overriding abstract methods

- A subclass can override public non-static methods from its parent class
  - If the superclass contains abstract methods, a concrete subclass ***must*** override them!
- Use of **`@Override`** annotation is optional
  - However, if you don't use **`@Override`** annotation, the compiler will not check if you are really overriding an existing method

Example of method overriding (1)

```java
abstract class Animal {
  public abstract void
    makeSound();
}

class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow!");
  };
}

```

```java
class Dog extends Animal {
  public void makeSound() {
    System.out.println("Woff woff!");
  };
}
public class AnimalTest {
  public static void main(String[] args){
    Cat cat = new Cat(); cat.makeSound();
    Dog dog = new Dog(); dog.makeSound();
  }
}
```

```
$ java AnimalTest
Meow!
Woff woff!
```

Example of method overriding (2)

```java
abstract class Animal {
  public abstract void
    makeSound();
}

class Cat extends Animal {
  public void makeNoise() {
    System.out.println("Meow!");
  };
}
```

```java
class Dog extends Animal {
  public void makeSound() {
    System.out.println("Woff woff!");
  };
}
public class AnimalTest {
  public static void main(String[] args){
    Cat cat = new Cat(); cat.makeSound();
    Dog dog = new Dog(); dog.makeSound();
  }
}
```

```
$ javac AnimalTest.java
error: Cat is not abstract and does not override abstract method makeSound() in Animal
```

Example of method overriding (3)

```java
abstract class Animal {
  public void makeSound() {
    System.out.println(“Burp!");
  }
}

class Cat extends Animal {
  public void makeNoise() {
    System.out.println("Meow!");
  };
}
```

```java
class Dog extends Animal {
  public void makeSound() {
    System.out.println("Woff woff!");
  };
}
public class AnimalTest {
  public static void main(String[] args){
    Cat cat = new Cat(); cat.makeSound();
    Dog dog = new Dog(); dog.makeSound();
  }
}
```

```
$ java AnimalTest
Burp!
Woff woff!
```

Example of method overriding (4)

```java
abstract class Animal {
  public void makeSound() {
    System.out.println(“Burp!");
  }
}
class Cat extends Animal {
  @Override
  public void makeNoise() {
    System.out.println("Meow!");
  };
}
```

```java
class Dog extends Animal {
  public void makeSound() {
    System.out.println("Woff woff!");
  };
}
public class AnimalTest {
  public static void main(String[] args){
    Cat cat = new Cat(); cat.makeSound();
    Dog dog = new Dog(); dog.makeSound();
  }
}
```

```
$ javac AnimalTest.java
error: method does not override or implement a method from a supertype
```

Dynamic binding

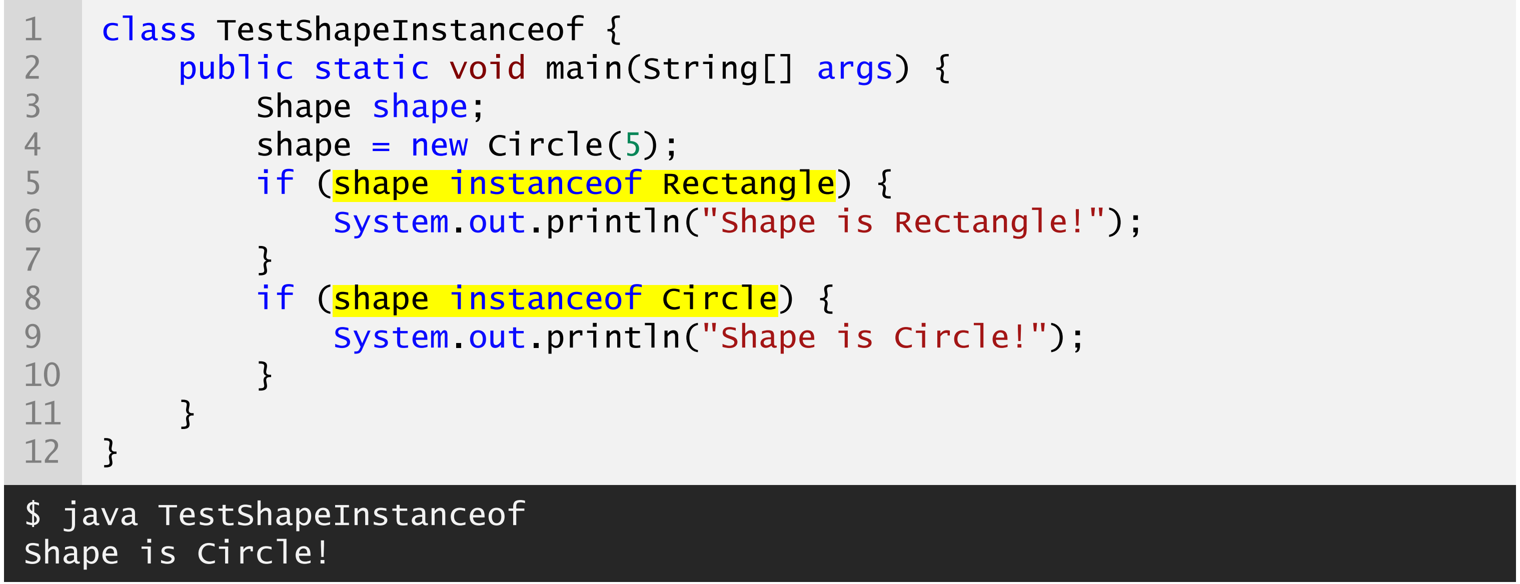
- *Dynamic binding* or (*late binding*): e.g., Java decides which class's method to call at execution time, not at compile time
    - A superclass reference can be used to invoke only methods of the *superclass*—the *subclass* method implementations are invoked *polymorphically*
- Attempting to invoke a subclass-only method directly on a superclass reference is a compilation error
- Operator **`instanceof`** may be used to check if the object can be cast into a particular type

Example of polymorphic processing

```java
class TestShape {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Circle(3.0);
        shapes[1] = new Rectangle(5.0,2.0);
        shapes[2] = new Rectangle(4.0,4.0);
        double totalArea = 0.0;
        for(int i=0; i<shapes.length; i++)
            totalArea += shapes[i].area();
        System.out.println("Total area: " + totalArea);
    }
}
```

```
$ java TestShape
Total area: 54.27433388230814
```

Example using instanceof



```java
class TestShapeInstanceof {
    public static void main(String[] args) {
        Shape shape;
        shape = new Circle(5);
        if (shape instanceof Rectangle) {
            System.out.println("Shape is Rectangle!");
        }
        if (shape instanceof Circle) {
            System.out.println("Shape is Circle!");
        }
    }
}
```

```
$ java TestShapeInstanceof
Shape is Circle!
```

Example of casting to a subclass

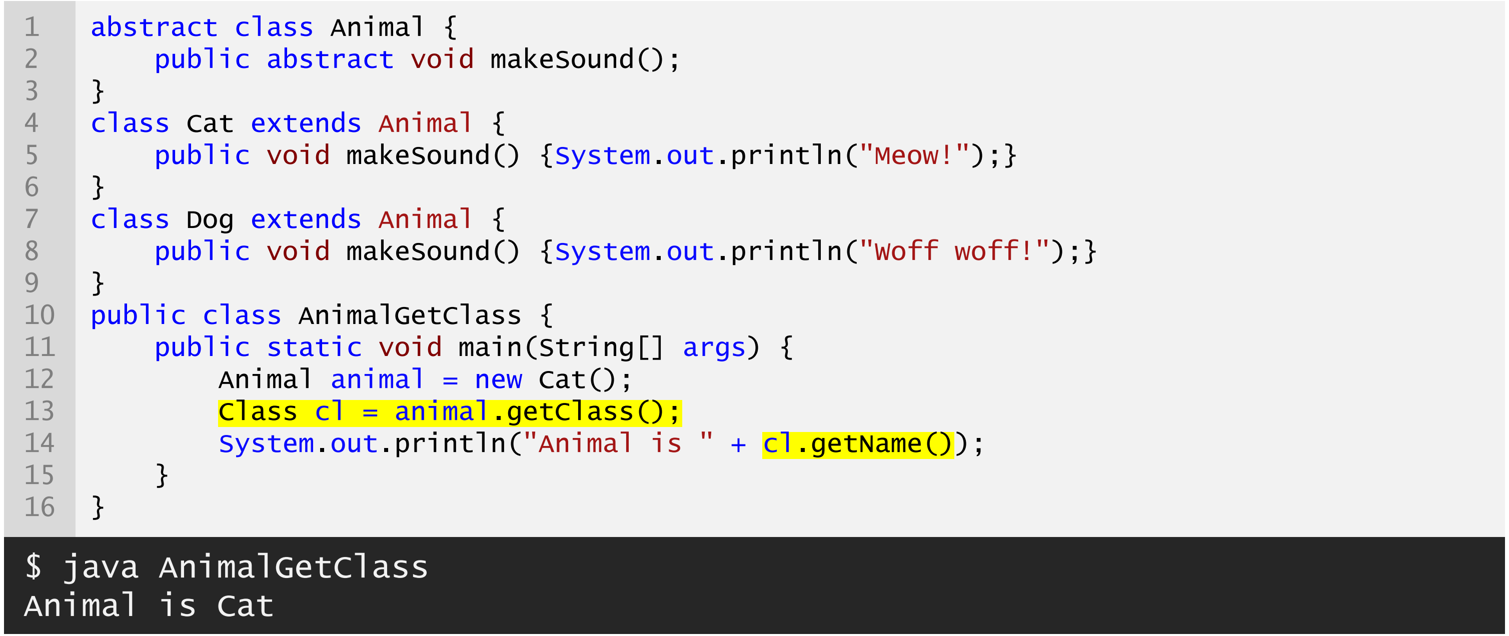
```java
class ShrinkShape2 {
    public static void main(String[] args) {
        Shape shape = new Rectangle(1.0,3.0);
        System.out.println("Original area: " + shape.area());
        if(shape instanceof Rectangle) {
            Rectangle rect = (Rectangle)shape;
            double w = rect.getWidth();
            double h = rect.getHeight();
            shape = new Rectangle(w/2, h/2);
            System.out.println("New area: " + shape.area());
        }
    }
}
```

```
$ java ShrinkShape2
Original area: 3.0
New area 0.75
```

Get information about a class

- Every object *knows its own class* and can access this information through the **`getClass()`** method, which all classes inherit from class `Object`
  - The `getClass` method returns an object of type **`Class`** (from package `java.lang`), which contains information about the object's type, including its class name
    - Note that keyword `class` and class `Class` are different things!
  - The result of the `getClass` call is used to invoke **`getName()`** to get the object's class name

Example of getClass()



```java
abstract class Animal {
    public abstract void makeSound();
}
class Cat extends Animal {
    public void makeSound() {System.out.println("Meow!");}
}
class Dog extends Animal {
    public void makeSound() {System.out.println("Woff woff!");}
}
public class AnimalGetClass {
    public static void main(String[] args) {
        Animal animal = new Cat();
        Class cl = animal.getClass();
        System.out.println("Animal is " + cl.getName());
    }
}
```

```
$ java AnimalGetClass
Animal is Cat
```

Final methods and classes

- A *final method* in a superclass cannot be overridden in a subclass
  - Methods that are declared `private` are implicitly `final`, because it’s not possible to override them in a subclass
  - Methods that are declared `static` are implicitly `final`
  - A `final` method’s declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as *static binding*
- A *final class* cannot be extended to create a subclass
  - All methods in a `final` class are implicitly `final`

Final classes in Java API

- Class `String` is an example of a `final` class
  - If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected
  - Since class `String` cannot be extended, programs using `String`s can rely on the functionality of `String` objects as specified in the Java API.
  - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions
- Note that in the JAVA API, most of the classes are ***not*** declared final

Calling methods from constructors

- ***Do not call overridable methods from constructors***: when creating a *subclass* object, this could lead to an overridden method being called before the *subclass* object is fully initialized
  - Recall that when you construct a *subclass* object, its constructor ***first*** calls one of the direct *superclass's* constructors
  - If the *superclass* constructor calls an overridable method, the *subclass's* version of that method will be called by the *superclass* constructor—before the *subclass* constructor's body has a chance to execute
  - Difficult-to-detect errors can occur if the *subclass* method depends on initialization not yet been performed in the *subclass* constructor
- However, it is acceptable to call a `static` method from a constructor

Example of casting to a subclass

```java
abstract class Animal {
    public Animal() {
        System.out.println("Called constructor Animal");
    }
}
abstract class Mammal extends Animal {
    public Mammal() {
        System.out.println("Called constructor Mammal");
    }
}
class Cat extends Mammal {
    public Cat() {
        System.out.println("Called constructor Cat");
    }
}
public class ConstructorExample1 {
    public static void main(String[] args) {
        Cat cat = new Cat();
    }
}
```

```
$ java ConstructorExample1
Called constructor Animal
Called constructor Mammal
Called constructor Cat
```

Example of casting to a subclass

```java
abstract class Animal {
    public String sound() { return "nothing"; }
    public Animal() {
        System.out.println("Animal says “ + sound());
    }
}
class Cat extends Animal {
    public String sound() { return "meow"; }
    public Cat() {
        System.out.println("Cat says “ + sound());
    }
}
public class ConstructorExample2 {
    public static void main(String[] args) {
        Cat cat = new Cat();
    }
}
```

```
$ java ConstructorExample1
Animal says meow
Cat says meow
```

Questions, comments?

1495 UNIVERSITY OF ABERDEEN

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 4, Session 2: Interfaces

Interface

- With ***interfaces***, unrelated classes can implement a set of common methods: people and systems can interact one with another in a standardized way via the interfaces
- Example: The controls on a radio serve as an interface between the user of radio the internal components of the radio
  - Offers a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM)
  - Different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands)
  - The interface specifies ***what*** operations a radio must permit users to control, but does not specify ***how*** the operations are performed

Interfaces in Java

- A Java interface describes a set of methods that can be called on an object
- An *interface declaration* begins with the keyword **`interface`** and typically contains only constants and `abstract` methods
  - All interface members *must* be `public`
  - Mandatory methods declared in an interface are implicitly `public abstract` methods
  - All fields are implicitly `public`, `static` and `final`
- An interface cannot be instantiated, so it does not define a constructor

Using interface in a class

- To use an interface, a concrete class must specify that it `implements` the interface and must declare each method in the interface with specified signature
- A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.
  - Implementing an interface is like signing a contract with the compiler: *"I will declare all the methods specified by the interface or I will declare my class abstract"*

Example of using interface

```java
abstract class Animal {
    protected boolean hungry = true;
}
interface Feedable {
    public void feed();
}
class Cat extends Animal implements Feedable {
    public void feed() {
        hungry = false;
    }
}
public class InterfaceExample1 {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.feed();
        System.out.print("Is the cat hungry? ");
        System.out.println(cat.hungry ? "Yes" : "No");
    }
}
```

```
$ java InterfaceExample1
Is the cast hungry? No
```

New features of interfaces in Java

- From Java SE 8, interfaces also may contain `public` default methods with concrete default implementations that specify how operations are performed if not overridden
    - If a class implements such an interface, the class also receives the interface’s `default` implementations (if any)
    - To declare a default method, place the keyword `default` before the method’s return type and provide a concrete method implementation
- From JAVA SE 8 interfaces may contain static methods
- From JAVA SE 9 interfaces may also contain private methods, however, defining a protected method causes compilation error

Example of interface with default method

```java
abstract class Animal {
    protected boolean hungry = true;
}
interface Feedable {
    public default void feed() {
        System.out.println("No method for feeding!");
    }
}
class Cat extends Animal implements Feedable {
}
public class InterfaceExample2 {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.feed();
        System.out.print("Is the cat hungry? ");
        System.out.println(cat.hungry ? "Yes" : "No");
    }
}
```

```
$ java InterfaceExample2
No method for feeding!
Is the cat hungry? Yes
```

Using multiple interfaces

- Java does not allow subclasses to inherit from more than one superclass (multiple heritance); however, a class can inherit from one superclass, *and* implement as many interfaces as it needs
- To implement more than one, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```java
public class Subclass extends Superclass implements
    FirstInterface, SecondInterface {
```

- The Java API contains a lot of interfaces, and many of the Java API methods take interface arguments and return interface values

When to use an interface

- An interface is often used when disparate classes (i.e., unrelated classes) need to share common methods and constants
  - Allows objects of unrelated classes to be processed *polymorphically* by responding to the *same* method calls
  - You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality
- An interface should be used in place of an `abstract` class when there is no default implementation to inherit
- Like `public abstract` classes, interfaces are typically `public`
  - A `public` interface must be declared in a file with the same name as the interface and the `.java` filename extension

Same method in multiple interfaces

- If a class implements two interfaces, both defining a default method with the same name, then the class *must* override that method and provide an implementation
- It is possible to call one of the interface default methods using the following syntax:

```
InterfaceName.super.method( );
```

Example of interface with default method

```java
interface Pianist {
    default void play() { System.out.println("Bling blong"); }
}
interface Violinist {
    default void play() { System.out.println("Viih vooh"); }
}
class Musician implements Pianist, Violinist {
    public void play() {
        Pianist.super.play();
    }
}
public class MusicianExample {
    public static void main(String[] args) {
        new Musician().play();
    }
}
```

```
$ java InterfaceExampleMusician
Bling blong
```

Extending interfaces

- Like classes, interfaces can be extended
  - Extended interface inherits all the methods from the superinterface
- An interface can extend more than one superinterfaces
- A class that implements such an interface must implement the abstract methods defined directly by the interface and all the abstract methods inherited from all the superinterfaces

Example of extended interfaces

```java
interface Scalable  {  void scale(double scaler); }
interface Rotatable {  void rotate();             }
interface Transformable extends Scalable, Rotatable {}
class Rectangle implements Transformable {
    public double w, h;
    public Rectangle(double w, double h) { this.w = w; this.h = h; }
    public void scale(double scaler) { this.w *= scaler; this.h *= scaler; }
    public void rotate() {
        double temp = this.w; this.w = this.h; this.h = temp; }
}
public class TransformableExample {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10.0,5.0);
        rect.scale(0.5);
        System.out.printf("New dimensions: %f,%f\n", rect.w, rect.h);
    }
}
```

```
$ java InterfaceExampleTransformable
New dimensions: 5.000000,2.500000
```

Functional interfaces

- As of Java SE 8, any interface containing only one `abstract` method is known as a *functional interface*—also called SAM (Single Abstract Method) interfaces
- Optional annotation **@FunctionalInterface** can be used
- Example functional interfaces defined in Java API:
  - **`Comparator`** (Chapter 16 in Deitel book) — implement this interface to define a method to compare two objects of given type to determine if the first object is less than, equal to or greater than the second
  - **`Runnable`** (Chapter 23 in Deitel book) — implement this interface to define a task that runs in parallel with other parts of your program

Example of functional interface

```java
@FunctionalInterface
interface Talkable {
    void talk(String msg);
}
public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        Talkable person = new Talkable() {
            public void talk(String msg) {
                System.out.println(msg);
            }
        };
        person.talk("Hello world!");
    }
}
```

```
$ java FunctionalInterfaceExample
Hello world!
```

Lambda expressions

- *Lambda expression* is a new feature introduced in Java SE 8, allowing to represent the single method of a functional interface
- Format of lambda expression: **`(argument list) -> { body }`**
  - Argument list can be empty **`()`** or contain one or more arguments
  - Body contains the implementation of the method
- Lambda expressions are used in *functional programming*
  - We will revisit lambda expressions later in this course in more detail

Example of lambda expression (1)

```java
@FunctionalInterface
interface Talkable {
    void talk(String msg);
}
public class LambdaExample1 {
    public static void main(String[] args) {
        Talkable person = (msg) -> {System.out.println(msg);};
        person.talk("Hello world!");
    }
}
```

```
$ java LambdaExample1
Hello world!
```

Example of lambda expression (2)

```java
@FunctionalInterface
interface Talkable {
    void talk(String msg);
}
public class LambdaExample2 {
    public static void main(String[] args) {
        Talkable person = (msg) -> {System.out.println(msg);};
        person.talk("Hello world!");
        Talkable quietPerson =
            (msg) -> {System.out.println("Shh!");};
        quietPerson.talk("Hello world!");
    }
}
```

```
$ java LambdaExample2
Hello world!
Shh!
```

Summary

- Abstract classes are classes including methods without concrete implementation
  - Abstract methods used as a “placeholder” for concrete implementations in subclasses of an abstract class
  - Helps to keep definition and implementation of functionality separate
- Interfaces define a set of common functionalities, like abstract classes
  - Interface is a kind of “agreement” on what your class can do
  - Java does not support multiple inheritance, but similar effect can be achieved by implementing multiple interfaces
  - Functional interface is a type of interface that contains exactly one abstract class

Questions, comments?