Revised: 12/17/2019

Object Oriented Programming

Inheritance and Polymorphism

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Inheritance
  - Subclasses and Superclasses
    - Override Methods
    - Final Methods and Classes

- Polymorphism

- Abstract Classes

- Interfaces

Colors/Fonts

• Local Variable Names	–	Brown
• Primitive data types	–	Fuchsia
• Literals	–	Blue
• Keywords	–	Orange
• Object names	–	Green
• Operators/Punctuation	–	Black
• Field Names	–	Lt Blue
• Method Names	–	Purple
• Parameter Names	–	Gold
• Comments	–	Gray
• Package Names	–	Pink

Source Code	– **Consolas**
Output	– Courier New

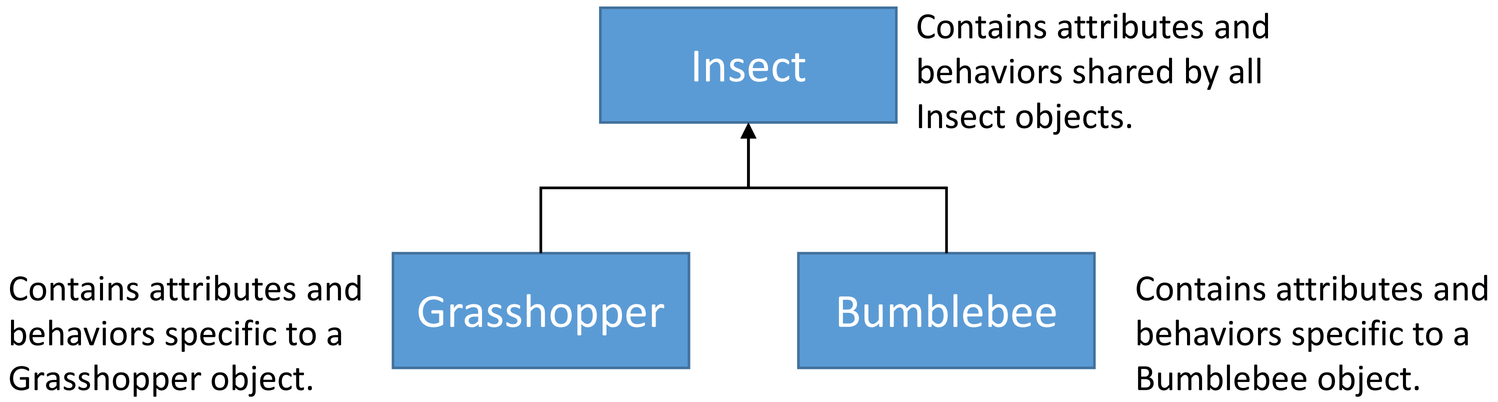
Inheritance in Object Oriented Design

- Real-life objects are often a specialized version of a more general object.
    - For example, a hammer and screwdriver are both tools.
        - They are both instruments used to build something.
        - But, they each have their own special use.

    - As another example, grasshoppers and bumblebees are both insects.
        - They share the general characteristics of an insect.
        - But, they each special characteristics of their own.
            - Grasshoppers have a jumping ability.
            - Bumblebees have a stinger.

Inheritance in Object Oriented Design

- An object oriented system can be designed in a similar way.
- We have the more specific objects *inheriting* attributes and behaviors from a more general object.


Insect
Contains attributes and behaviors shared by all Insect objects.
Grasshopper
Contains attributes and behaviors specific to a Grasshopper object.
Bumblebee
Contains attributes and behaviors specific to a Bumblebee object.

Inheritance: The “is a” Relationship

- In object oriented design, inheritance is used to create an “is a” relationship among classes.
  - A grasshopper “is an” insect.
  - A poodle “is a” dog.
  - A car “is a” vehicle.

- The specialized objects have:
  - All of the characteristics (attributes and behaviors) of the general object.
  - Additional characteristics that make it special.

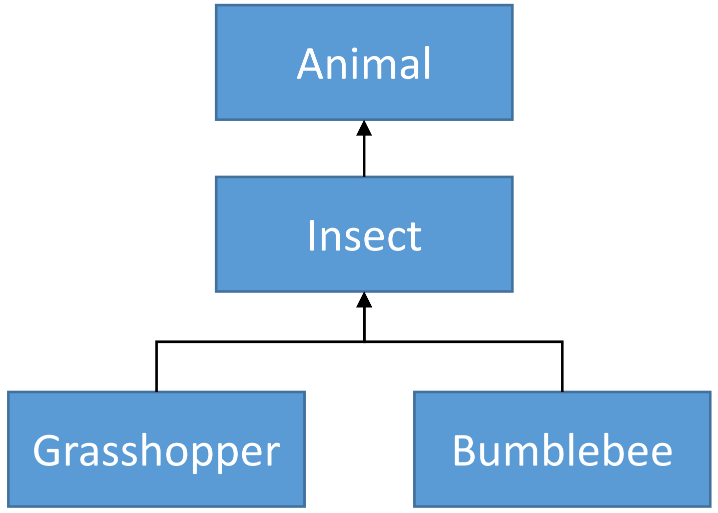
Superclasses and Subclasses

- A **superclass** (sometimes called a *base class* or *parent class*) is a class whose attributes and behaviors are inherited by other classes.
    - In the previous model, the Insect class is a superclass.

- A **subclass** (sometimes called a *derived class* or *child class*) is a class that inherits the attributes and behaviors of another class.
    - In the previous model, the Grasshopper and Bumblebee classes are subclasses.

Superclasses and Subclasses

- A class can be both a superclass and a subclass.
  - In the model below, the Insect class is the superclass of the Grasshopper and Bumblebee classes, but is itself a subclass of an Animal class.



Superclasses and Subclasses

- In Java, a subclass can only have **one** superclass.
  - The subclass will inherit its superclass's <u>public</u> fields and methods.
    - Any private fields or methods of the superclass will not be inherited.
  - A subclass *can* access private fields of its superclass, provided the superclass has an appropriate public accessor method.
    - A subclass can reference its superclass using the **`super`** keyword.

- However, a superclass can have unlimited subclasses.

Superclasses and Subclasses

- Let’s consider a simple class named Employee…

```
public class Employee {

    public String name;
    public double wage;

    public double getWage() {
        return wage;
    }

}
```

Superclasses and Subclasses

- This HourlyEmployee class could be a subclass of the Employee class.
  - It will inherit the name and wage fields and the getWage method.
- The **extends** keyword is used in the class header to specify this class's superclass.

Inherited from superclass

```
public class HourlyEmployee extends Employee {

    public HourlyEmployee(String nameIn, double wageIn) {
        name = nameIn;
        wage = wageIn;
    }

}
```

Subclasses and Superclasses

```
public class EmployeeTest {

    public static void main(String[] args) {
        HourlyEmployee emp1 = new HourlyEmployee("Bill", 10.50);

        System.out.println("This employee's name is " + emp1.name);
        System.out.println("Their wage is $" + emp1.getWage());
    }

}
```

```
This employee's name is Bill
Their wage is $10.50
```

Subclasses and Superclasses

- As we’ve seen previously, it’s not always a good idea to access a field directly.
- But… if the Employee class’s name field was private, its subclasses won’t inherit it.
  - Private fields and methods are not inherited.

```
System.out.println("This employee's name is " + emp1.name);
```

Subclasses and Superclasses

```
public class Employee {

    private String name;
    private double wage;

    public double getWage() {
        return wage;
    }

}
```

```
public class HourlyEmployee extends Employee {

    public HourlyEmployee(String nameIn,
                          double wageIn) {
        name = nameIn;
        wage = wageIn;
    }

}
```

Will not compile

Subclasses and Superclasses

- Adds a constructor to the Employee class.

```
public class Employee {

    private String name;
    private double wage;

    public Employee(String nameIn, double wageIn) {
        name = nameIn;
        wage = wageIn;
    }

    public double getWage() {
        return wage;
    }

}
```

Superclass Constructors

- Constructors are not inherited by subclasses.

- When the superclass has at least one constructor defined, any subclass constructors must call a superclass constructor.

- The `super` keyword is used by a class to refer to its superclass.
    - `super` – how a class refers to its superclass.
    - `this` – how a class refers to itself.

Superclass Constructors

- The call to the superclass constructor **must** be the first statement in the body of a subclass constructor.

```
public class HourlyEmployee extends Employee {

    public HourlyEmployee(String nameIn,
                          double wageIn) {
        super(nameIn, wageIn);
    }

}
```

Calls the superclass constructor that accepts a String and a double for arguments.

Superclass Constructors

```java
public class Employee {

    private String name;
    private double wage;

    public Employee(String nameIn,
                    double wageIn) {
        name = nameIn;
        wage = wageIn;
    }

    public double getWage() {
        return wage;
    }

}
```

```java
public class HourlyEmployee extends Employee {

    public HourlyEmployee(String nameIn,
                          double wageIn) {
        super(nameIn, wageIn);
    }

}
```

Superclass Constructors

- A public accessor would now be required to access the name field.
- This will be inherited by any subclasses (since is it public).

```
public class Employee {

    private String name;
    private double wage;

    public Employee(String nameIn,
                    double wageIn) {
        name = nameIn;
        wage = wageIn;
    }

    public double getWage() {
        return wage;
    }

    public String getName() {
        return name;
    }

}
```

Superclass Constructors

```
public class EmployeeTest {

    public static void main(String[] args) {
        HourlyEmployee emp1 = new HourlyEmployee("Bill", 10.50);

        System.out.println("This employee's name is " + emp1.getName());
        System.out.println("Their wage is $" + emp1.getWage());
    }

}
```

Inherited from the Employee class

```
This employee's name is Bill
Their wage is $10.50
```

Superclasses and Subclasses

- This Supervisor class is a subclass of the Employee class.
  - It will inherit the fields and methods of Employee.
  - Not much of a difference between HourlyEmployee and Supervisor, for now.

```
public class Supervisor extends Employee {

    public Supervisor(String nameIn, double wageIn) {
        super(nameIn, wageIn);
    }

}
```

Override Methods

- An ***override method*** is a method in a subclass that replaces a method inherited from the superclass.
  - The signatures must be the same.

```
public class Supervisor extends Employee {

    public Supervisor(String nameIn, double wageIn) {
         super(nameIn, wageIn);
    }

    public double getWage() {
        return super.getWage() * 1.3;
    }

}
```

Overrides the getWage method inherited from the Employee class

Needs to get the value of the wage field from the superclass.

Override Methods

```
public class EmployeeTest {

    public static void main(String[] args) {
        Supervisor emp2 = new Supervisor("Shirley", 10.50);

        System.out.println("This supervisor's name is " + emp2.getName());
        System.out.println("Their wage is $" + emp2.getWage());

    }

}
```

Inherited from the Employee class

Uses its own getWage method.

```
This supervisor's name is Shirley
Their wage is $13.65
```

Final Methods

- A ***final method*** is a method in a superclass that cannot be overridden by a subclass.
  - Include the **final** keyword in the method header, after the access modifier (if present).
- The method will still be inherited by subclasses.

```
public class Employee {

    private String name;
    private double wage;

    public Employee(String nameIn,
                    double wageIn) {
        name = nameIn;
        wage = wageIn;
    }

    public double getWage() {
        return wage;
    }

    public final String getName() {
        return name;
    }

}
```

This method cannot be overridden by a subclass.

Final Classes

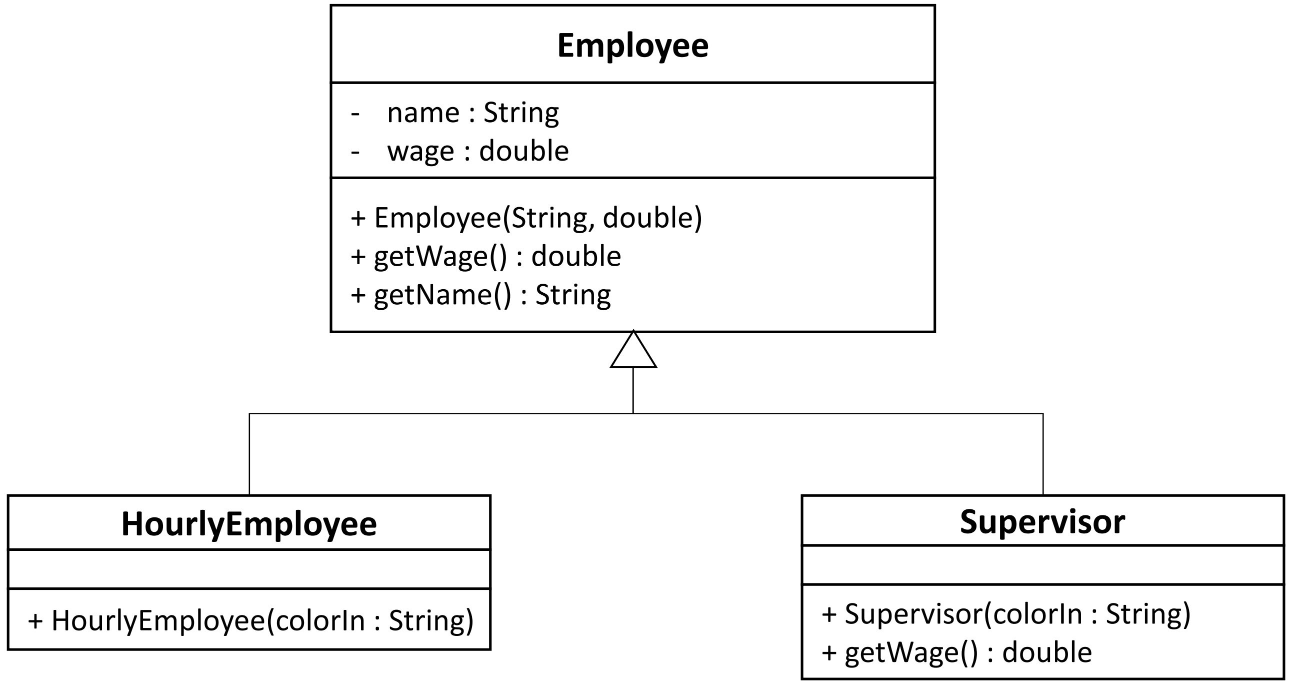
- A ***final class*** is a class that does not allow any subclasses.
    - Include the **`final`** keyword in the class header, after the access modifier (if present).

Prohibits subclasses of the HourlyEmployee object.

```
public final class HourlyEmployee extends Employee {

    public HourlyEmployee(String nameIn,
                          double wageIn) {
        super(nameIn, wageIn);
    }

}
```

Inheritance in Class Diagrams

- The arrow always points from the subclass to the superclass.



Polymorphism

- ***Polymorphism*** is the ability for an object to be more than one data type.

- It's an extension of the "is-a" relationship.

- Like its namesake, polymorphism will appear in a variety of ways.
  - We'll first establish the rules of polymorphism, then see how it is useful.

Polymorphic Variables

- Recall that our HourlyEmployee class is a subclass of the Employee class.
  - The HourlyEmployee class, like any subclass, is inherently polymorphic.
  - Objects of this type are both an HourlyEmployee object **and** an Employee object.

```
public final class HourlyEmployee extends Employee {

    public HourlyEmployee(String nameIn,
                          double wageIn) {
        super(nameIn, wageIn);
    }

}
```

Polymorphic Variables

- We could declare a variable of type Employee and instantiate it with a new HourlyEmployee object.
  - An HourlyEmployee object *is a* Employee object.
  - It is guaranteed to have all the functions of the Employee object (inherited or overridden.)

```
Employee emp1 = new HourlyEmployee("Bill", 10.50);
```

- A limitation is we can only call the methods that exist in Employee.
  - Any other methods in the HourlyEmployee class could not be called.

Polymorphic Variables

- The opposite is not true.
  - An Employee object is *not* an HourlyEmployee object.

Will not compile

```
HourlyEmployee emp1 = new Employee("Bill", 10.50);
```

- There is no guarantee that an Employee object has all of the methods of an HourlyEmployee object, since the Employee class is not a subclass of the HourlyEmployess class.

Polymorphic Variables

- A benefit here is that any subclass of an Employee object can be assigned to this emp1 variable.

```
Employee emp1 = new HourlyEmployee("Bill", 10.50);
emp1 = new Supervisor("Shirley", 10.50);
```

- Since HourlyEmployee and Supervisor objects are Employee subclasses, the lines will compile without issue.
  - However, they are all still limited to using methods inherited (or overridden) from the Employee class.

Polymorphic Variables – Why do this?

- Most local variables won’t be polymorphic.
    - The previous examples were to show how it works.

- It’s more likely you’ll see an object’s field (instance variable) be polymorphic.

```
public class Staff {

    private Employee employee1;
    private Employee employee2;
    private Employee employee3;

}
```

Polymorphic Variables – Why do this?

- In the Staff class below, the three fields can be any type of Employee.

```
public class Staff {

    private Employee employee1;
    private Employee employee2;
    private Employee employee3;

}
```

- Despite the fact that it will be limited to using Employee-specific methods, this Staff class will work with *any* new Employee subclasses.
  - No modifications in the Staff class will be needed.

Polymorphic Parameters/Arguments

- This example Staff class now has separate methods for setting different Employee fields.

```
public class Staff {

    private Employee employee1;
    private Employee employee2;
    private Employee employee3;

    public void setEmployee1(Employee e) {
        employee1 = e;
    }

    public void setEmployee2(Employee e) {
        employee2 = e;
    }

    public void setEmployee3(Employee e) {
        employee3 = e;
    }

}
```

Polymorphic Parameters/Arguments

```
public class StaffTest {

    public static void main(String[] args) {
        Staff myStaff = new Staff();

        HourlyEmployee emp1 = new HourlyEmployee("Bill", 10.50);
        Supervisor emp2 = new Supervisor("Shirley", 10.50);
        HourlyEmployee emp3 = new HourlyEmployee("Joe", 11.00);

        myStaff.setEmployee1(emp1);
        myStaff.setEmployee2(emp2);
        myStaff.setEmployee3(emp3);

    }

}
```

Abstract Classes

- An ***abstract class*** is a class that serves as a base class from which other classes are derived from.

- Abstract classes cannot be instantiated.
  - Abstract classes serve solely as a superclass for other classes.

Abstract Classes

- A class becomes abstract when you place the abstract keyword in the class header.

```
public abstract class Bird
```

Abstract Classes

- We can have variables of this type, but no longer have instances of a Bird object.

```
public abstract class Bird {

    private String color;

    public Bird(String colorIn) {
        color = colorIn;
    }

    public final String getColor() {
        return color;
    }

}
```

OK

```
Bird bird1;

bird1 = new Bird("White");
```

Will not compile

Abstract Classes

- The body of an abstract class can be very similar to that of non-abstract classes.
  - Abstract classes can have instance/class variables.
  - Abstract classes can have constructors.
    - They would have to be called by a subclass constructor, though.
  - Abstract classes can have methods.

- Abstract classes can contain abstract methods.
  - We have been using non-abstract methods until this point.

Abstract Methods

- An ***abstract method*** is a method that must be defined in a subclass.
  - Abstract methods can only be declared in abstract classes.
  - Declarations of abstract methods do not appear in non-abstract classes.

- An abstract method only has a header; it has no body.
  - Abstract methods must be defined in a subclass.

```
public abstract void birdCall();
```

Semicolon!

Abstract Methods

- This ensures the method will be present and defined in each subclass.
  - The abstract class only declares the abstract methods.
  - Each subclass is **required** to define their own functionality for the method.

- The subclass will not compile if you do not define all of the superclass's abstract methods.
  - Unless the subclass is itself abstract, in which case the method(s) will eventually need to be defined by a class somewhere down the class hierarchy.

Abstract Methods

- Subclasses will still inherit fields and methods.
  - Non-abstract subclasses will be forced to define the birdCall method.

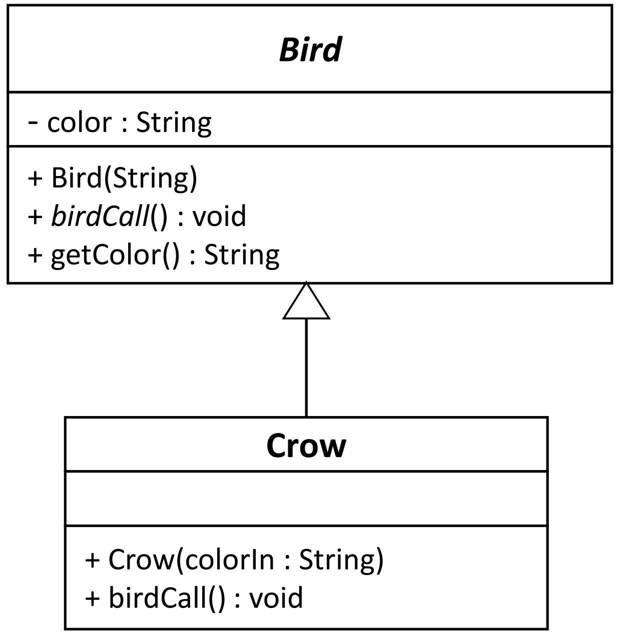
```java
public abstract class Bird {

      private String color;

      public Bird(String colorIn) {
            color = colorIn;
      }

      public abstract void birdCall();

      public final String getColor() {
            return color;
      }

}
```

```java
public class Owl extends Bird {

      public Owl(String colorIn) {
            super(colorIn);
      }

      public void birdCall() {
            System.out.println("Hoot!");
      }

}
```

This class would not compile without this method

Abstract Classes/Methods in Class Diagrams

- Uses arrows.
- Abstract class names/methods are italicized.



Interfaces

- An ***interface*** specifies only behavior for classes.

- Interfaces contain only abstract methods.
  - No constructors or defined methods.
    - Interfaces can contain fields, but they will be public, static, and constant.
  - Like abstract classes, interfaces cannon be instantiated.

- While a class can only extend one superclass, a class can implement multiple interfaces.

Interfaces

- An interface is declared using the `interface` keyword instead of the `class` keyword.

```
public interface TalkingBird {


}
```

Interfaces

- All methods in an interface are abstract methods.

```
public interface TalkingBird {
    public abstract void sayHello();
}
```

Interfaces

- A class implements an interface using an implements clause in the class header.
  - This Parrot class will inherit from the abstract Bird class.
  - This Parrot class must define any abstract methods from the abstract Bird class **and** the TalkingBird interface.

```
public class Parrot extends Bird implements TalkingBird {


}
```

Interfaces

```
public class Parrot extends Bird implements TalkingBird {

    public Parrot(String colorIn) {
        super(colorIn);
    }

    public void birdCall() {
        System.out.println("Squawk!");
    }

    public void sayHello() {
        System.out.println("Hello!");
    }

}
```

Bird Constructor

Required by Bird

Required by TalkingBird

Also inherits the getColor method from the Bird class

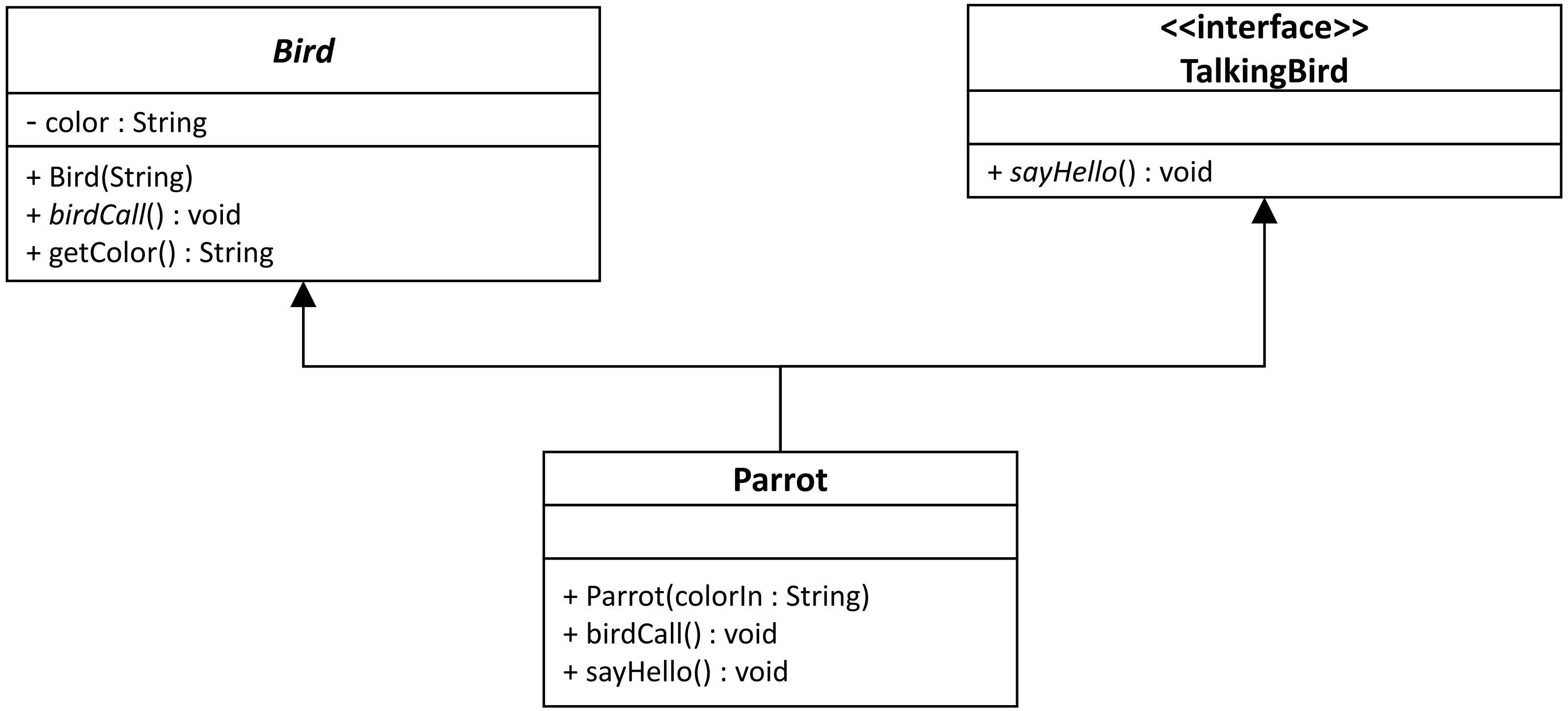
Interfaces

- Interfaces contribute to polymorphism.
  - The Parrot object is a Parrot, a Bird, and a TalkingBird object.

```
Parrot bird1 = new Parrot("Green");
Bird bird2 = new Parrot("Red");
TalkingBird bird3 = new Parrot("Yellow");
```

- Each is limited to using the methods specified by each type.
  - For example, the sayHello method can be called on bird3 (since the variable is of the TalkingBird type) and on bird1 (since Parrot implements the TalkingBird interface.)
  - Since bird2 is of the Bird type, we could not call sayHello without explicitly typecasting bird2 to the Parrot type.

Interfaces in Class Diagrams



Interfaces

- Classes can implement multiple interfaces.
  - (Code not shown for the FlyingBird interface/Illustration only)

```
public class Parrot extends Bird implements TalkingBird, FlyingBird {


}
```

- This Parrot class will need to define methods required by both interfaces and its abstract superclass.
  - A Parrot object would now be a Parrot, Bird, TalkingBird, and FlyingBird object.

Common Questions

- Can abstract classes be a subclass of another abstract class?
  - Yes.
- Can abstract classes be a subclass of a non-abstract class?
  - Yes.
- Can abstract classes implement an interface?
  - Yes.
- Can an interface be a subclass of another interface?
  - No. Remember, an interface is ***not*** a superclass.
- Can an interface implement another interface?
  - Yes. But remember, any class that implements the lower interface will need to implement methods from **both** interfaces.

Comparisons

	Can Be Instantiated	Allows Inheritance	Allows Polymorphism	Can Contain Abstract Methods
Non-Abstract Superclass	✓	✓	✓	✗
Abstract Class	✗	✓	✓	✓
Interface	✗	✗*	✓	✓

*Interfaces can have fields, but they will be public, static, and constant.