# Interfaces and Inheritance

Based on The Java™ Tutorial
(http://docs.oracle.com/javase/tutorial/java/IandI/index.html)
Based on the notes from David Fernandez-Baca and Steve Kautz

Bryn Mawr College
CS206 Intro to Data Structures

## Abstraction in Software Design

**Abstraction** is a way to reduce coupling:

View components in terms of their essential features, ignoring details that aren't relevant to our particular concerns. Each component provides a well-defined **interface** specifying exactly how we can interact with it.

## Interfaces in Java

- An interface is a **contract** between module writers, specifying exactly how they will communicate.
- An **interface** in Java is a bunch of public methods without bodies. That is, an interface specifies the method names, return types, and parameter types, and nothing else.

Here's a simple example:

```
public interface ISpeaking {
    void speak();
}
```

An implementation of this interface:

```
public class Bird implements ISpeaking{
    @Override
    public void speak() {
        System.out.println("tweet");
    }
}
```

## Interface

- The implements keyword means:
  - "I promise to provide an implementation (a method body containing actual code) for each method specified by the ISpeaking interface."
- The "@Override" is an **annotation**. It is **not** required for the code to compile and run, but it is very useful: it tells the compiler that your **intention** is to implement a method defined in an interface (or, we will see later, a superclass). That way, the compiler can check that you have the method signature correct.
- Implementing an interface is the simplest form of inheritance. We say that Bird is a **subtype** of ISpeaking, and ISpeaking is a **supertype** of Bird.

## Implementation of an Interface

There can be other implementations of the same interface, e.g.,

```
public class Person implements ISpeaking {
    @Override
    public void speak() {
        System.out.println("Hi!");
    }
}
```

- Using a Java interface formalizes and enforces the "separation of interface from implementation" that is one of the benefits of encapsulation.
- The purpose of programming with interfaces is to reduce coupling.

## Implementation of an Interface

```
public interface ISpeaking { void speak(); }
public interface ILicensable { License getLicense(); }
```

```
public class Dog implements ISpeaking, ILicensable {
    private String name;
    private License license;
    public Dog(String name, License license) {
        this.name = name; this.license = license;
    }
    @Override
    public void speak() {
        System.out.println("woof");
    }
    @Override
    public License getLicense() { return license; }
    public String getName() { return name; }
}
```

Consider with the interfaces Ispeaking and ILicensable. Let us ignore the License class itself, which is not needed for this discussion.

## Notes on Interface

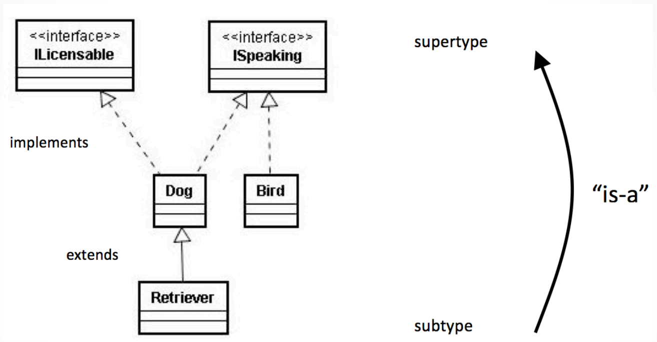
- You **can** declare an object whose type is an interface, so the following is **legal**:
  - ISpeaking b; // OK
- You **cannot** instantiate an interface variable, so the following is **illegal**:
  - ISpeaking b = new ISpeaking(); // NO!!
- If an interface variable refers to an object, that object must belong to a class that implements that interface. For example:
  - ISpeaking b = new Bird(); // OK
- All methods in a Java interface are public, so the public keyword is redundant.
- A class that implements an interface **must** implement all the methods declared in the interface.

## Inheritance (by Class Extension)

```
public class Retriever extends Dog {
    public Retriever(String name, License license) {
        super(name, license); //call superclass (Dog) constructor
    }
    @Override public void speak() {
        System.out.println("raooou");
    }
    public Bird retrieve() { return new Bird();}
}
```

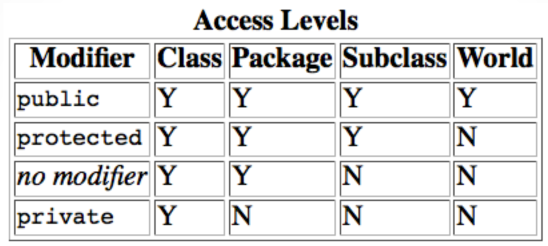
- We say Retriever is a **subclass** or **subtype** of Dog (it is also a subtype of ISpeaking and of ILicensable) and Dog is a **superclass** or **supertype** of Retriever.
- In Java, a class can **implement** more than one interface, but can **extend** only **one** other class.

## Class Hierarchies



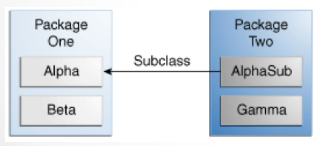
- The superclass/subclass (supertype/subtype) relationships are indicated by the arrows with the big triangles.
- A dotted line means "**implements** (an interface)" and a solid line means "**extends** (a class)". Drawing all the arrows pointing upwards allows us to see the subtype-supertype relations easily.
- We also call the subtype relation the "**is-a**" relation: A Retriever **is a** type of Dog, a Dog **is a** type of ISpeaking.

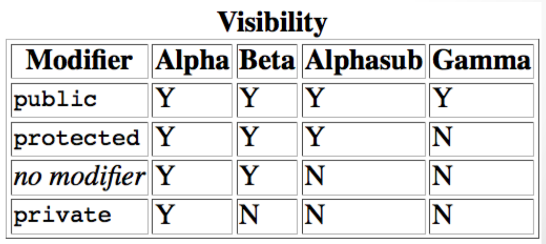
## Recall:Access Levels and Visibility



**Access Levels**

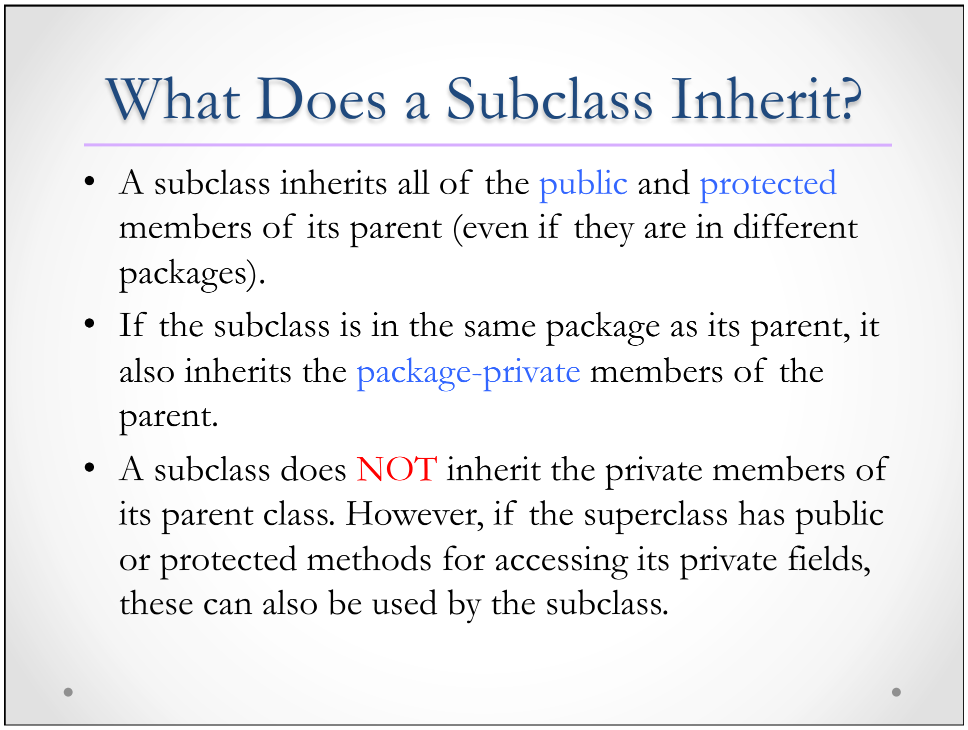
| Modifier | Class | Package | Subclass | World |
|---|---|---|---|---|
| `public` | Y | Y | Y | Y |
| `protected` | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| `private` | Y | N | N | N |





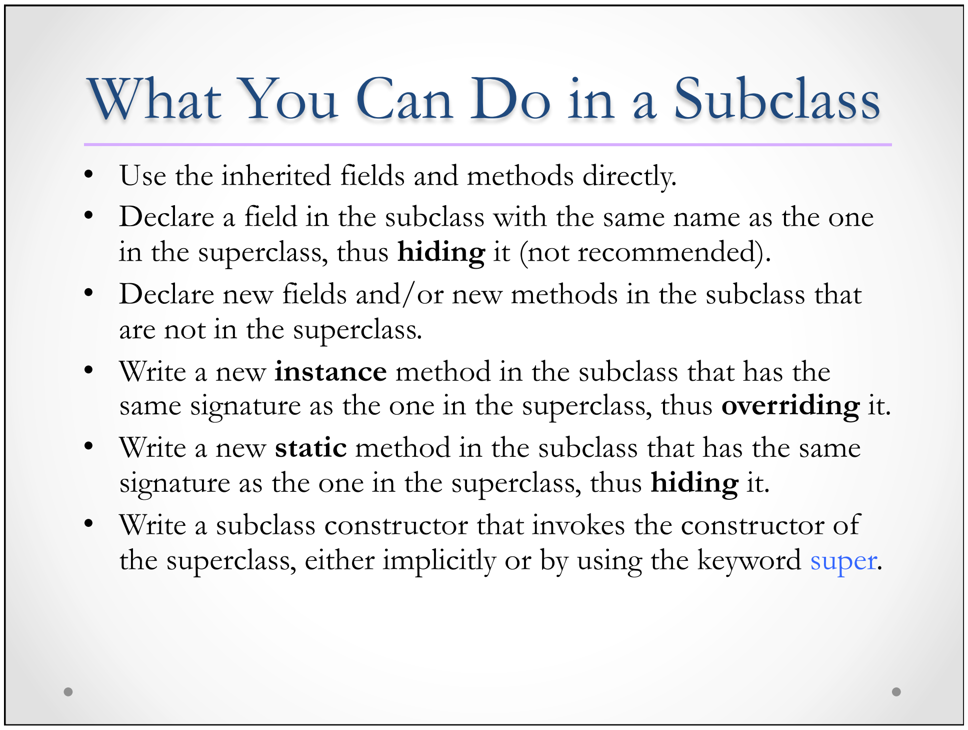
**Visibility**

| Modifier | Alpha | Beta | Alphasub | Gamma |
|---|---|---|---|---|
| `public` | Y | Y | Y | Y |
| `protected` | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| `private` | Y | N | N | N |



## What Does a Subclass Inherit?

- A subclass inherits all of the public and protected members of its parent (even if they are in different packages).
- If the subclass is in the same package as its parent, it also inherits the package-private members of the parent.
- A subclass does NOT inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.



## What You Can Do in a Subclass

- Use the inherited fields and methods directly.
- Declare a field in the subclass with the same name as the one in the superclass, thus **hiding** it (not recommended).
- Declare new fields and/or new methods in the subclass that are not in the superclass.
- Write a new **instance** method in the subclass that has the same signature as the one in the superclass, thus **overriding** it.
- Write a new **static** method in the subclass that has the same signature as the one in the superclass, thus **hiding** it.
- Write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

## Casting Objects

- **Casting** shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations.
- Animal obj = new Cat(); //obj is both Animal and Cat (**implicit casting**).
- Cat aCat = obj; //Compile-time error!
- Cat aCat = (Cat) obj; //**explicit casting**

**Note:** You can make a logical test as to the type of a particular object using the instanceof operator. This can save you from a runtime error owing to an improper cast. For example:

if (obj instanceof Cat) { Cat aCat= (Cat)obj; }

## Overriding and Hiding Methods

- An instance method in a subclass with the same signature and return type as an instance method in the superclass **overrides** the superclass's method.
- Recall that signature of a method consists of the method's name and the parameter types.
- If a subclass defines a class method with the same signature as a class method in the superclass, the method in the subclass **hides** the one in the superclass.

## Distinction between Hiding and Overriding

- The version of the overridden method that gets invoked is the one in the subclass.
- The version of the hidden method that gets invoked depends on where it is invoked (from the superclass or the subclass).

## Overriding and Hiding Methods

```
public class Animal {
   public static void testClassMethod() {
      System.out.println("The class" +
        " method in Animal.");
   }
   public void testInstanceMethod() {
      System.out.println("The instance "
      + " method in Animal.");
   }
}
```

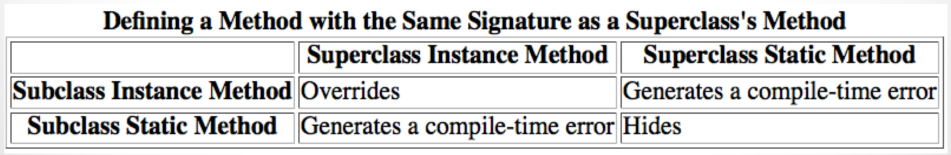
```
public class Cat extends Animal {
   public static void testClassMethod() {
      System.out.println("The class method" +
       " in Cat.");
   }
   public void testInstanceMethod() {
      System.out.println("The instance method" +
       " in Cat.");
   }

   public static void main(String[] args) {
      Cat myCat = new Cat();
      Animal myAnimal = myCat;
      Animal.testClassMethod();
      myAnimal.testInstanceMethod();
   }
}
```

Hiding

Overriding

## Access Level for Overriding Methods

- The access specifier for an overriding method can allow more, but not less, access than the overridden method.
  - E.g., a protected instance method in the superclass can be made public, but not private, in the subclass.



**Defining a Method with the Same Signature as a Superclass's Method**

| | **Superclass Instance Method** | **Superclass Static Method** |
|---|---|---|
| **Subclass Instance Method** | Overrides | Generates a compile-time error |
| **Subclass Static Method** | Generates a compile-time error | Hides |

## Hiding Fields(not recommended!)

- Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.
- Within the subclass, the field in the superclass cannot be referenced by its simple name.
- Instead, the field must be accessed through super.
- If a variable is non-private, you should never *shadow* it (re-declare a variable with the same name) in the subclass, even though this is allowed by the compiler.

## Recap

Two kinds of inheritance:

- **Implementing an interface**: it means you agree to provide code for all the methods that the interface declares. The purpose of using Java interfaces is to decouple components.
- **Inheritance (class extension)**: it allows a subclass to inherit all attributes and operations of its superclass. Additionally, class extension allow us to
  - **add** new attributes or behavior (new instance variables and/or methods) to a class and
  - modify behavior by **overriding** existing methods.

  Class extension aids us in writing classes that share and reuse code.

## Polymorphism

**Polymorphism** means that a variable of a given type T can hold a reference to an object of any of T's subtypes. Consider the statement below:

```
1.   ISpeaking s = new Dog("Ralph", null);
```

Statement 1 does a few things:

- It invokes a constructor to instantiate an object of type Dog. The constructor returns a reference to the object.
- It declares a variable s of type ISpeaking. In fact, it declares that s will reference an ISpeaking object.
- It makes s point to the new Dog object.

A Dog object can masquerade as an ISpeaking object, because every Dog object is an ISpeaking object.

## Polymorphism

Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

```
public class Animal {
   public void printDescription(){
     System.out.println("This is an
        Animal.");
   }
}
```

```
public class Cat extends Animal {
   @Override
   public void printDescription(){
     super.printDescription();
     System.out.println("And this is a Cat.");
   }
}
```

```
public class TestAnimal {
   public static void main(String[] args) {
       Animal myAnimal = new Animal();
       Animal myCat= new Cat();
       myAnimal.printDescription();
       myCat.printDescription();
   }
}
```

output?

## Polymorphism and Dynamic Binding

- Definition:
  - **Static type**: The type (class) of a variable. Also known as **compile-time type**.
  - **Dynamic type**: The class of the object the variable references. Also known as **run-time type**.

```
1.   ISpeaking s = new Dog("Ralph", null);
```

Static Type

Dynamic Type

```
2.   s = new Bird();  //OK
```

This is OK, because Bird is a subtype of Ispeaking.

The static type of s remains Ispeaking (it will **always** be that), but its dynamic type has gone from Dog to Bird.

## Polymorphism and Dynamic Binding

```
3. Dog d = new Retriever("Clover", null);
```

This is **OK**, because Retriever is a subtype of Dog.

```
4. Retriever r = (Retriever) d;
```

This is also **OK**, because

Even though the **static type** of d is Dog,
its **dynamic type** is Retriever.
**However**, an explicit downcast is still required.

The compiler does not see the run- time type of a variable, only its static type. A cast changes the static type of a variable.

## Polymorphism and Dynamic Binding

```
5. d = new Bird();  //compiler error
```

Bird is *not* a subtype of Dog.

```
6. d = (Dog) s;
```

This compiles, but fails at runtime with a `ClassCastException`.

```
7. d = new Dog("Ralph", null);
8. d.speak();              // "woof"
9. d = new Retriever("Clover", null);
10.d.speak();              // "raooou"
```

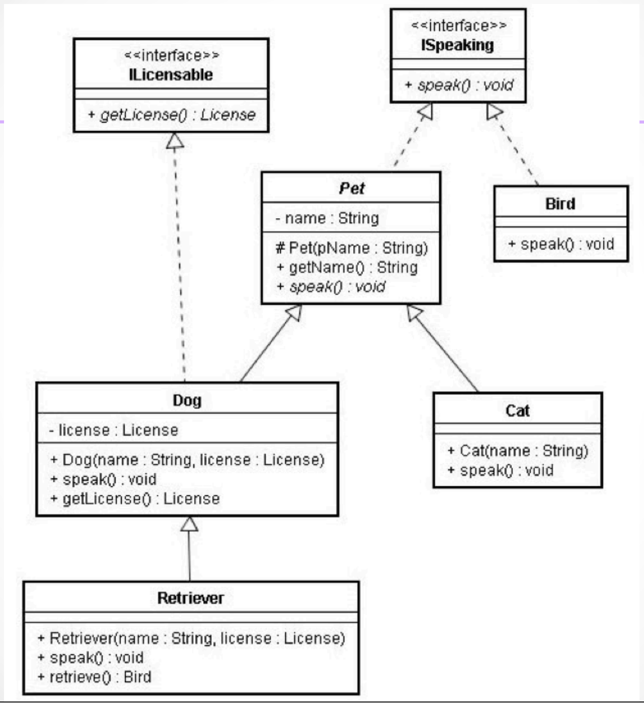
## Abstract Class

- Suppose that we want to add a Cat class

```
class Cat implements ISpeaking {
   private String name;
   public Cat(String name) {
      this.name = name;
   }
   @Override
   public void speak(){
      System.out.println("miao");
   }
   public String getName() { return name; }
}
```

## Recall Dog

```
public class Dog implements ISpeaking, ILicensable {
   private String name;
   private License license;
   public Dog(String name, License license) {
      this.name = name;   this.license = license;
   }
   @Override
   public void speak() {
      System.out.println("woof");
   }
   @Override
   public License getLicense() { return license; }
   public String getName() { return name; }
}
```



## Abstract Class

- To allow the class to have a method that is declared but not implemented, we can make the class abstract.

```
public abstract class Pet implements ISpeaking{
   private String name;
   protected Pet(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
   public abstract void speak();
}
```

## Abstract Class - Rules

- An abstract method is declared with the abstract keyword, and ends with a semicolon instead of a pair of braces with a method body.
- All methods of an interface are automatically abstract.
- If a class contains an abstract method, the class must also be declared abstract.
- You cannot create an instance of an abstract class with new.

## Dog and Cat classes

```
class Cat extends Pet{
   private String name;
   public Cat(String name) {
      super(name);
   }
   @Override
   public void speak(){
      System.out.println("miao");
   }
   ~~public String getName() { return name; }~~
}
```

## Interfaces vs. Abstract Classes

- A Java class can inherit from only one class, even if the superclass is an abstract class. However, a class can "implement" (inherit from) as many Java interfaces as you like.
- A Java interface cannot implement any methods, nor can it include any fields except "final static" constants. It only contains method prototypes and constants.

## Dog and Cat classes

```
public class Dog extends Pet implement ILicensable {
   private String name;
   private License license;
   public Dog(String name, License license) {
      super(name);   this.license = license;
   }
   @Override
   public void speak() {
      System.out.println("woof");
   }
   @Override
   public License getLicense() { return license; }
   ~~public String getName() { return name; }~~
}
```

## Root of the Java Class Hierarchy

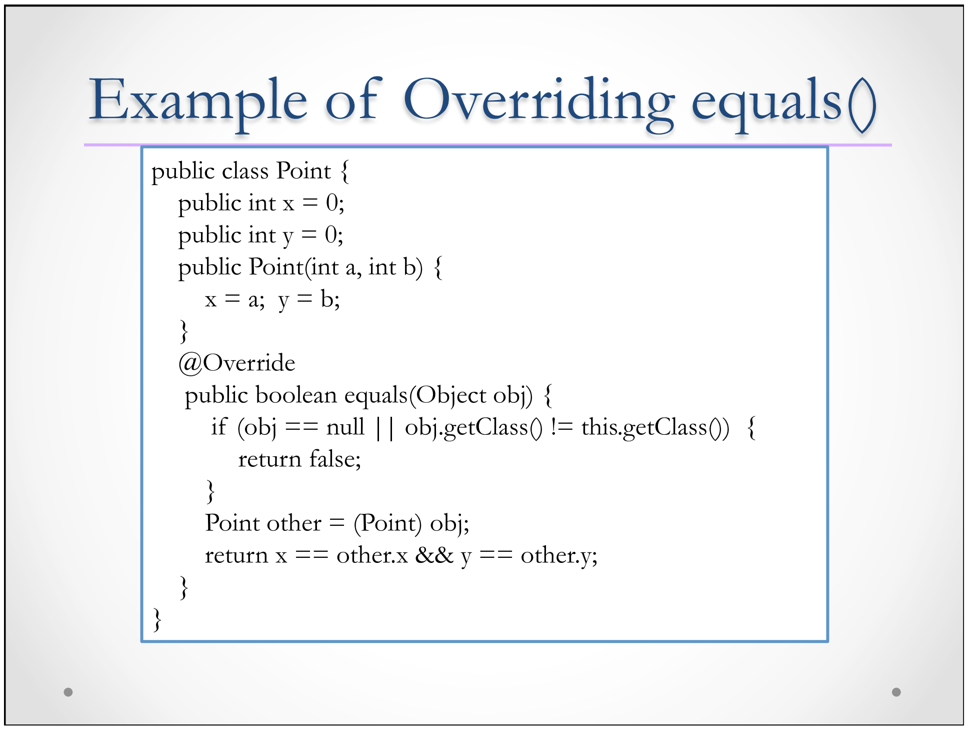
- Every class in Java is a subclass of `java.lang.Object`.
- Several predefined methods:
  - public String toString(): returns a string representation of the object (read the source code for the default implementation). We commonly override toString() to provide a more useful description.
  - public final Class<?> getClass(): returns the runtime class of this Object.
  - public boolean equals(Object obj): indicates whether some other object is "equal to" this one.

## Object References and Equality

- The operation == determines whether or not two **references** are the same. It does not determine whether the objects are “the same”.

```
String s = "hurley";
String t = "HURLEY".toLowerCase();
System.out.println(s==t);    //false
System.out.println(s.equals(t));   //true
```

The String class *overrides* equals() to check whether the characters are the same. (check the source code!)



## Example of Overriding equals()

```
public class Point {
    public int x = 0;
    public int y = 0;
    public Point(int a, int b) {
        x = a;  y = b;
    }
    @Override
     public boolean equals(Object obj) {
         if (obj == null || obj.getClass() != this.getClass())  {
             return false;
        }
        Point other = (Point) obj;
        return x == other.x && y == other.y;
    }
}
```