CS 228: Introduction to Data Structures
Lecture 3

Inheritance

As mentioned last time, Java offers two ways to implement inheritance:

- interface implementation, and
- class extension.

We will illustrate these concepts using the `Insect` class[1], which is posted on Blackboard.

Interface Implementation

As we have seen, a Java ***interface*** — specified via the keyword `interface` — is a set of public methods without bodies. That is, an interface specifies the method names,

[1] A version of the `Insect` class was used in a ComS 228 exam in a previous semester,

return types, and parameter types, and nothing else[2]. An interface is a ***contract*** between module writers, specifying exactly how they will communicate.

Interface implementation — whereby one provides code for all the methods that an interface declares — is one of the ways inheritance is implemented in Java. As an illustration, consider the following simple interface specifying just one method.

```
public interface Locomotion
{
   void move();
}
```

Note. All methods in a Java interface are public by default, so the `public` keyword is redundant.

An implementation of this interface could be:

[2] This is not strictly true for the latest release of Java, Java 8, which allows the definition of static and default methods in interfaces. For the time being, we will ignore this issue.

```
public class Grasshopper
    implements Locomotion
{
    @Override
    public void move()
    {
        System.out.println("hop");
    }

    public String feedOn()
    {
        return "grass";
    }

}
```

The `implements` keyword means:

"I promise to provide an implementation (a method body containing actual code) for each method specified by the `Locomotion` interface."

We say that `Grasshopper` is a ***subtype*** of `Locomotion`, and `Locomotion` is a ***supertype*** of `Grasshopper`.

Note. "`@Override`" is an ***annotation***. It is not required for the code to compile and run, but it is very useful: it tells the compiler about your *intention* 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.

Observe that the `Grasshopper` class:

- ***overrides*** the `move()` method from `Locomotion` and
- ***adds*** a method, `feedsOn()`, that is not present in `Locomotion`.

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

```
public class Bee implements Locomotion
{
    @Override
    public void move()
    {
        System.out.println("fly");
    }

    public void makeHoney()
    {
        System.out.println
            ("Orange Blossom");
    }
}
```

Now `Bee` is a subtype of `Locomotion`, and `Locomotion` is a supertype of `Bee`. Note that `Bee` has a method, `makeHoney()`, that is neither in `Locomotion` nor in `Grasshopper`.

Note that

- it is ***legal*** to declare an object whose type is an interface, but

- it is ***illegal*** to try to instantiate an interface variable

Thus, the following is ***legal***:

```
Locomotion b;   // OK
```

After the above statement, b can reference an object of type `Bee` or `Grasshopper`. E.g., you could do:

```
b = new Grasshopper();
```

In contrast, the following is ***illegal***:

```
Locomotion b = new Locomotion();  // NO!!
```

This makes sense, since an interface does not define any constructors.

Dynamic and Static Types, Polymorphism, and Dynamic Binding

Java variables are ***polymorphic***. That is, suppose variable x has been declared to be of a certain type T. Then, throughout the execution of the program, x may reference an object of any of T's subtypes. The object x references may change during the execution, though.

Let us illustrate polymorphism using `Locomotion`, `Grasshopper` and Bee. Suppose we have the following declaration.

```
1.  Locomotion b;
```

We say that the ***static type*** (or ***compile-time type***) of b is Locomotion. The static type of b never changes — that's why it's called static.

Now, b can reference any object that is of type `Locomotion` or a subtype of `Locomotion`. For example, it is OK to do do the following.

```
2.   b = new Grasshopper();   // OK
```

This is OK, because `Grasshopper` is a subtype of `Locomotion`. Now, though, the actual type of b is `Grasshopper`, so when we invoke the `move()` method, we are referring to the `move()` method for the `Grasshopper` class.

```
3.   b.move(); // prints “hop”
```

This “actual type” of b is formally known as the ***dynamic type*** (or ***run-time type***). Unlike the static type, the dynamic type can change during program execution — that’s why it’s called “dynamic”. For example, after the preceding statements, it is OK to do:

```
4.   b = new Bee();   // OK
```

This works because Bee is also a subtype of `Locomotion`. Although b's static type remains `Locomotion`, b's dynamic type is now Bee.

Method invocations are always done using an object's dynamic type. Thus, after line 4, if we invoke the `move()` method on b, we will use the Bee version.

```
5.  b.move(); // prints “fly”
```

It is important to keep the following fact in mind:

Java does type checking using static types.

To get an idea of what this implies, let us see why the Java compiler would flag the next statement as an error.

```
6.  b.makeHoney(); // ERROR!!!
```

After statement 5, the dynamic type of b is Bee, so we expect b to have a `makeHoney()` method. However, the Java compiler only knows that the static type of b is `Locomotion`, and `Locomotion` objects to not have a `makeHoney()` method.

To summarize

- At compile time, Java uses ***static type checking***: It checks for type errors using the ***static types*** of the variables, not their run-time types.

- At run time, Java uses ***dynamic binding***: The code that is executed when method `m()` is invoked on a variable `x`, is determined by the ***dynamic*** type of `x`, regardless of `x`'s static type. Java implements dynamic binding via ***dynamic method lookup***.

Now, let's return to Line 6. The static type of b is indeed `Locomotion`, but b in fact references a `Bee` object (that's its dynamic type), so we should be able to invoke the `makeHoney()` method. In fact, we *can*, but for that we need a *cast*:

```
7.  ((Bee) b).makeHoney(); // OK
```

Casts change the static type of variables. In this particular case, the cast reassures the compiler that b is indeed a `Bee` object. The cast is legal, because `Bee` is a subtype of `Locomotion`.

Inheritance by Class Extension

Class extension is another form of inheritance. It allows a subclass to inherit all attributes and operations of its superclass. This helps with code reuse. Class extension allows us to

- ***add*** new attributes or behavior (new instance variables and/or methods) to a class and

- ***modify*** behavior by ***overriding*** existing methods.

An example. Since grasshoppers and bees are insects, it is natural to have the corresponding classes be subclasses of an Insect type — that is, as *extensions* of an `Insect` class.

```
public class Insect
{
    protected int size;
    protected String color;

    public Insect(int size, String color)
    {
        this.size = size;
        this.color = color;
    }
    public int getSize()
    {
        return size;
    }

    public String getColor()
    {
        return color;
    }

}
```

Note the use of the `protected` keyword. A `protected` variable is visible to subclasses.

We can now define `Grasshopper` as a subclass of `Insect` as follows.

```
public class Grasshopper extends Insect
    implements Locomotion
{

    public Grasshopper
        (int size, String color)
    {
        super(size, color);
    }
    @Override
    public void move()
    {
        System.out.println("hop");
    }

    public String feedOn()
    {
        return "grass";
    }

}
```

The `extends` keyword signals that the `Grasshopper` class extends the Insect class. Thus, `Grasshopper` is a *subclass* or *subtype* of `Insect`, from which it ***inherits*** the `getSize()` and `getColor()` methods. It also ***adds*** some behavior: `move()` and the `feedOn()`. Note the use of `super` in `Grasshopper`, in order to call the constructor for the superclass (`Insect`). (By the way, `Grasshopper` is also a subtype of `Locomotion`.)

A katydid is also called a long-horned grasshopper. We can define a `Katydid` class as a subclass of `Grasshopper` as follows[3].

[3] Note that a katydid is not actually a grasshopper, but a cricket!

```
public class Katydid extends Grasshopper
{
    public Katydid(int size, String color)
    {
        super(size, color);
    }

    @Override
    public String feedOn()
    {
        return "variety";
    }
}
```

`Katydid` inherits all the methods of `Grasshopper` (and, therefore, also of `Insect`, but it also ***modifies*** the existing behavior of `feedOn()`; that is, it ***overrides*** `feedOn()`.

`Katydid` is a *subclass* of `Grasshopper` (it is also a subtype of `Insect` and of `Locomotion`) and `Grasshopper` is a *superclass* or *supertype* of `Katydid`.

A Java class can ***extend*** only ***one*** other class. On the other hand, it can ***implement*** more than one interface. For example, suppose that in addition to the Locomotion interface, we have the following.

```java
public interface Pollination
{
   boolean pollinate();
}
```

Let us modify Bee so that it implements both `Locomotion` and `Pollination`. We also add a private variable, `swarm`.

```java
public class Bee extends Insect implements
Locomotion, Pollination
{
   private String swarm;

   public Bee
   (int size, String color, String swarm)
   {
      super(size, color);
      this.swarm = swarm;
   }

   public String getSwarm()
   {
      return swarm;
   }
```

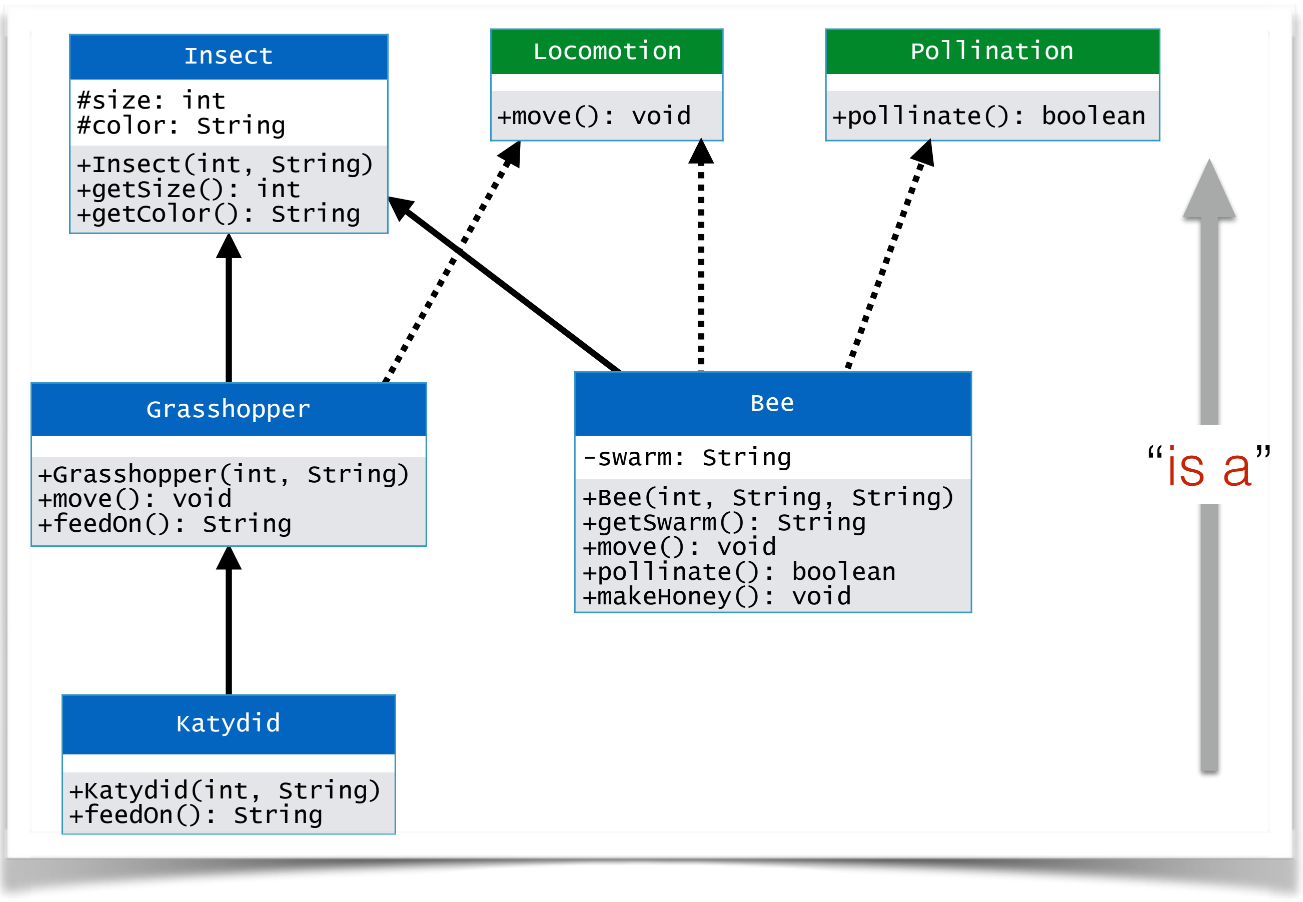
```
    @Override
    public void move()
    {
        System.out.println("fly");
    }

    @Override
    public boolean pollinate()
    {
        return true;
    }

    public void makeHoney()
    {
        System.out.println
        ("Orange Blossom");
    }
}
```

Class Hierarchies

A collection of class/subclass relationships defines a ***class hierarchy***, which can be represented by a ***class diagram*** (also called a ***UML diagram***, where UML stands for “Unified Modeling Language”). For the example we have just seen, the diagram looks like this:



The superclass/subclass (supertype/subtype) relationships are indicated by arrows. A dotted line means “***implements*** (an interface)”; a solid line means “***extends*** (a class)".   Drawing all the arrows pointing upwards allows us to see the subtype-supertype relations easily. The subtype relation is also called the "***is-a***" relation: A `Katydid` ***is a*** type of `Grasshopper`, a `Grasshopper` ***is a*** type of `Insect`.