CS 228: Introduction to Data Structures
Lecture 4
Monday, August 29, 2016

Recap

Last week we introduced three key concepts:

- ***Polymorphism:*** a variable of a given type T can hold a reference to an object of any of T's subtypes.

- ***Static type:*** The type a variable is declared to be at compile time.

- ***Dynamic type:*** They actual type of the object a variable refers to at run time.

We also saw that Java uses

- ***static type checking*** at compile time, and

- ***dynamic binding*** at run time.

Finally, we studied inheritance by interface implementation and class extension. We saw that subclasses can

- ***implement*** methods in an interface,
- ***inherit*** methods from a superclass,
- ***add*** new methods, and
- ***override*** existing methods.

Consider the class hierarchy we saw on Friday. Suppose we type the statement below.

```
1. Insect i = new Grasshopper(2, “green”);
```

Statement 1 does three things:

(i) It invokes a constructor to instantiate an object of type `Grasshopper`. The constructor returns a reference to the object.

(ii) It declares a variable `i` of type `Insect`. In fact, it declares that `i` will reference an `Insect` object.

(iii) It makes `i` reference the new `Grasshopper` object.

Step (iii) is correct, by the principle of polymorphism (an object can hold a reference to any of its subtypes):

every `Grasshopper` object is an `Insect` object.

However, the reverse is not true.

After Line 1 is executed,

- the ***static type*** of `i` is `Insect`
- the ***dynamic type*** of `i` is `Grasshopper`

Now suppose we carry out the following statement.

```
2.  i = new Bee(1,"gold","lake");
```

This is OK, because Bee is a subtype of `Insect`. The static type of `i` remains `Insect` (it will ***always*** be that), but its dynamic type has gone from `Grasshopper` to Bee.

```
3.  Grasshopper g = new Katydid(2,"brown");
```

This is OK, because `Katydid` is a subtype of `Grasshopper`.

```
4. Katydid k = (Katydid) g;
```

This is also OK, because, although the ***static type*** of `g` is `Grasshopper`, its ***dynamic type*** is `Katydid`. Therefore, `g` can be assigned to `k`. Nevertheless, because Java uses ***static type checking***, an explicit downcast is required.

Abstract Classes

The `Insect` class *factors out* the similarities among different insects. In this way, we avoid duplicating the code for `getSize()` and `getColor()` among `Bee`, `Grasshopper` and `Katydid`.

Suppose we want to add an `attack()` method to Insect. The precise method of attack depends on the species, though, so it seems wasteful to define the method for the class as a whole. Further, there is no real need to instantiate general `Insect` objects, since we are more

interested in the specific types of `Insect`. Thus, it makes sense to declare `Insect` to be `abstract`.

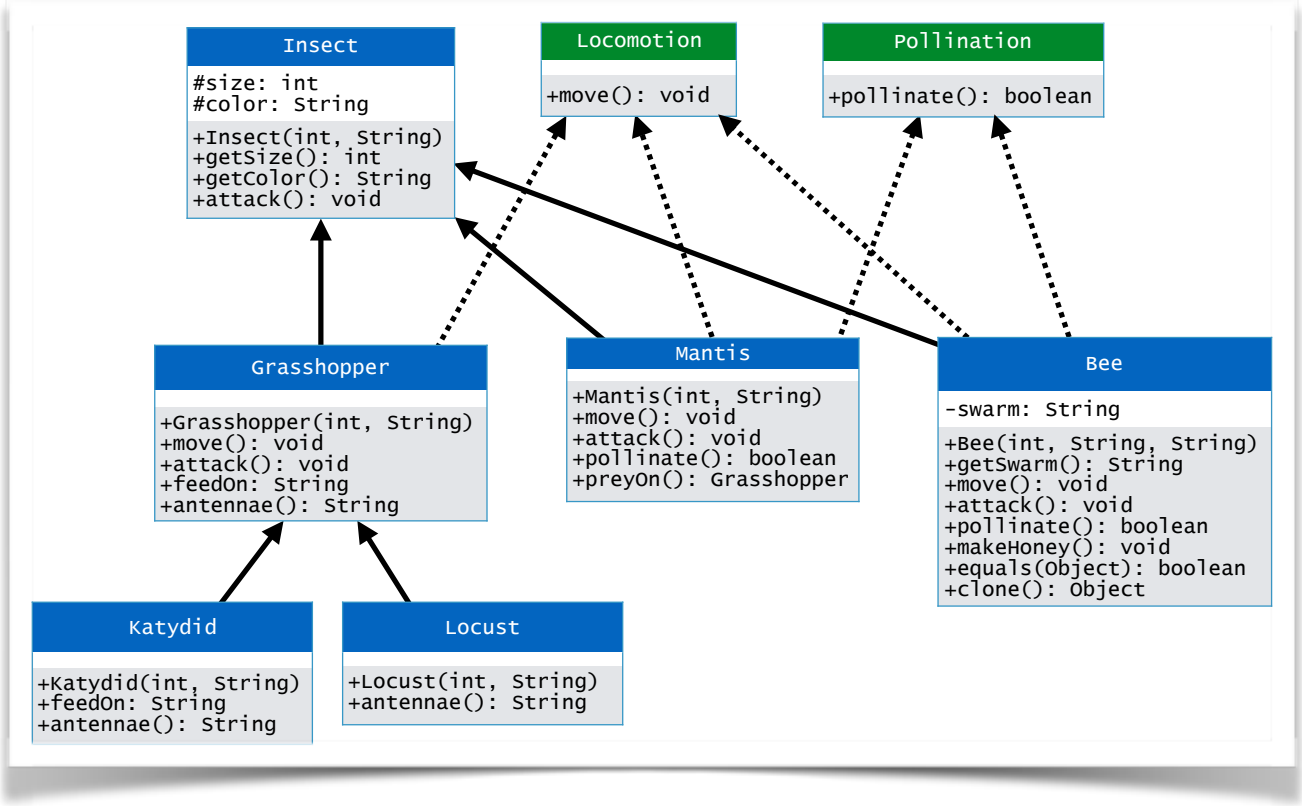
Abstract classes allow you to provide partial implementations, while leaving some details for subclasses to implement. The rules are:

- 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`.

Here is a modified version of `Insect` (see also Blackboard), with an abstract `attack()` method, which has to be implemented by subclasses.

```
public abstract 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;
    }

    public abstract void attack();
}
```

Blackboard also has an abstract `Grasshopper` class, with two concrete subclasses, `Katydid` and `Locust`. Additionally, there `Insect` has a `Mantis` class. The complete class hierarchy looks like this:



Remarks

A Java interface is in many ways like an abstract class. One difference is that, while a Java class can inherit from only one class — even if the superclass is `abstract` —, a class can implement (inherit from) as many Java interfaces as you like.

Up to Java 7, a Java interface could not implement any methods, nor could it include any fields except "`final static`" constants. It could only contain method prototypes and constants. Starting with Java 8, "`default`" and `static` methods may have be implemented in an

interface definition. We will not use these new features in this course, but it is important to be aware that they exist.

The guiding principle behind the distinction between abstract classes and Java interfaces is to avoid the problems associated with "multiple inheritance" (MI), which is what happens when a subclass can inherit from several superclasses. This is allowed in languages like C++. MI is responsible for some of the scariest tricks and traps of C ++. Problems arise, for example when two superclasses define differing methods or fields with the same name, but this is not the only problem that can occur. You will study this and other issues in programming language design in CS 342. Java avoids these problems by not having MI, except in the limited form of Java interfaces.

We should also note that an abstract class does not have to implement all the methods in an interface it implements. For instance, suppose abstract class A implements interface `I`, which has a `void` method `m()`. Then, the definition of A may omit any mention of `m()`. Omitting `m()` from the definition of A is equivalent to having the following line in the definition of A:

```
public abstract void m();
```

An Extended Example (Refer to Blackboard for Code)

There are 12 parts to this example. We assume that each part is executed separately.

1. Declaring static type.

```
Locomotion l;      // okay
Grasshopper g;     // okay
```

After these statements, the static types of `l` and g are `Locomotion` and `Grasshopper`, respectively.

2. Cannot instantiate interfaces or abstract classes.

An attempt to do so leads to a compile error.

```
~~Pollination p = new Pollination();~~
  // compile error: Cannot instantiate the
  // type Pollination
~~Insect i = new Insect(3, "Green");~~ // ERROR
  // compile error: Cannot instantiate the
  // type Insect
~~Insect i = new Grasshopper(2, “Brown");~~
  // SAME ERROR
```

3. We can assign subtype to supertype, but need a cast to go the other direction.

```
Katydid k = new Katydid(3, "Green");

Grasshopper g = k;  // OK by polymorphism:
 // Katydid is a subtype of Grasshopper

Locomotion l = new Katydid(3, "Brown");

g = l;   // Compile error: Cannot convert
   // from Locomotion to Grasshopper
```

Since the dynamic type of `l` is `Katydid`, we can fix the error with a downcast.

```
g = (Katydid) l;  // OK
```

4. Unrelated types cannot be converted to each other. That is, you cannot cast between types that are not in subtype/supertype relationship to each other.

```
Grasshopper g = new Locust(3, "Red");

Katydid k = (Katydid) g; // ERROR:
 // ClassCastException thrown at execution

g = new Mantis(4, "Green");
    // Compile error: Type mismatch cannot
    // convert from Mantis to Grasshopper
```

Note that the first error is ***not*** caught at compile time. The reason is that the static type of g — `Grasshopper` — is a super type of `Katydid` and, in principle, one can downcast a `Grasshopper` into a `Katydid`. At run time, however, Java will see that g is actually a Locust, which is not a subtype of `Katydid`.