<< Lecture 12 >>

Preamble

- Packages and *namespaces* (parenthesis for homework 2)
- **Polymorphism:** continuing on the animal kingdom (ecosystem simulation)

Packages and namespaces

- Suppose we have the following package
  - package *packech*
    - class AKlass
    - class OtherKlass
- From another package, how do we initialize Aklass?

```
packech.AKlass ak = new packech.AKlass(...);
```

- How do we access a static method in OtherKlass?

```
packech.OtherKlass.someStaticFunk(...)
```

- Just like members in classes and objects, we access classes and objects inside packages by using the period

```
pack1.pack2.pack3.class.method(...)
```

- How to save some time:

```
import packech.AKlass;
```

- How to save even more time:

```
import packech.*;
```

- Can do with nested packages:

```
import java.awt.image.renderable.*;
```

- Keyword **import** hints of **namespace**
  - The *namespace* is the *space* of available names for you to work with
  - Packages java.lang and java.util are always present in the namespace
  - Class String is actually java.lang.String, for instance
  - The following lines are, thus, redundant:

```
import java.lang.*;
import java.util.*;

public class...
```

Short review of what we have seen

- Recall the **Animal** *ontology* (in the ecosystem simulator)
- We have three levels: Animal, then *kingdom*, then species
    - type **Dog** is also of type **Mammal** and of type **Animal**
- We **Dog** extend from **Mammal**, **Mammal** extend from **Animal**, and we made **Animal** a concrete class
    - **Extend: code inheritance**
- All animals have species, which implies that **Animal** and its kingdoms should be **abstract** (again, you can't have an animal which is not a dog, a whale, a bird, a spider, etc.)
- **Interfaces**: true multiple inheritance is not allowed in Java, so multiple subtyping is done through interface/implements
    - Interfaces have signatures of methods

Let us first define a supertype, class Animal:

```
class Animal {
    boolean isalive = true;
    int age;
    //public Animal() {}  //default constructor
    public boolean isAlive() {return isalive;}
    public void killIt() {isalive=false;}
    public boolean canFeedOn(Animal x) {return false;}
}
```

Now, let us inherit from it (extension):

```
class Mammal extends Animal {
    public boolean canFeedOnMilk() {return true;}
}
```

Mammal is an *extension* of Animal because it was defined using **extends Animal**. Class Mammal inherited all the code from class Animal, and also has Animal as type. Let us study that with the method main:

```
public static void main(String[] arrrg) {
   Mammal m = new Mammal();
   System.out.println( m.isAlive() );
   Animal a = new Animal(), b = new Mammal();
   System.out.println( a );
   System.out.println( b );
}
```

However, observe that class Mammal has method canFeedOnMilk, which is not present in class Animal. Henceforth, the compiler will not allow us to compile the following code:

```
Mammal a = new Mammal();
Animal b = a;     // a and b point to the same object, and 'a' points to an
                  // object of type Mammal, which is also of type Animal

System.out.println( a.canFeedOnMilk() );
System.out.println( b.canFeedOnMilk() );     // the compiler complains here
```

Java is statically typed. It means that, even though classes can have multiple types, **the only available methods are indicated by the type in use**.

To properly justify the above, look at following example:

```
public void dairyLover( Animal m ) {
   if ( m.canFeedOnMilk() )
      System.out.println( “Animal “+m+” loves milk!” );
   else
      System.out.println( “Animal “+m+” is lactose intolerant” );
}
```

Since Java is object oriented, there is no way to prevent the above method to be accessed from another method outside the package, even the application we are developing. There is no way to guarantee that argument **m** is an Animal of type Mammal.

There are two compatible workarounds the above problem: **instanceof** and **casting**. We cover them at the end of these notes.

ABSTRACT CLASSES

Let us go back to subtyping. Every time we subtype from a type, we make the subtype an inhertitor of all the types of the supertype. Let us see this by implementing class Dog:

```
class Dog extends Mammal {
    public void fetchBranch() {}
}
```

Since Dog extends Mammal, Dog inherits the types of Mammal, which includes Mammal itself and its supertype Animal. So, Dog can assume types: Animal, Mammal, and Dog. We can modify the main to show this:

```
Animal x = new Dog();
Mammal y = new Dog();
Dog z = new Dog;
```

Again, the method fetchBranch is only valid in type Dog:

```
x.fetchBranch();    // compile-time error, x is of type Animal
y.fetchBranch();    // compile-time error, y is of type Mammal
z.fetchBranch();    // THIS IS OK!
```

The presence of class Dog should make us aware that, conceptually, all objects of Animal should have a species; it does not make sense to have **new Animal** together with **new Dog** in the same code. It just does not make sense.

How do we solve this problem? By making Animal and Mammal **abstract classes**. An abstract class cannot be instantiated.

```
abstract class Animal {
    boolean isalive = true;
    int age;

    public boolean isAlive() {return isalive;}
    public void killIt() {isalive=false;}
    public boolean canFeedOn(Animal x) {return false;}
}
```

```
abstract class Mammal extends Animal {
    public boolean canFeedOnMilk() {return true;}
}
```

Now, the following lines in main cannot compile:

```
Animal a = new Animal(), b = new Mammal();
```

Abstract classes cannot be instantiated, even though they can still contain code. But, they can also provide *bare definitions* of methods; these methods have the signature but no code associated. These methods must have the **abstract** keyword in them.

To illustrate the above point, let us enforce the *canFeedOnMilk* method on all animals.

```
abstract class Animal {
    boolean isalive = true;
    int age;

    public boolean isAlive() {return isalive;}
    public void killIt() {isalive=false;}
    public boolean canFeedOn(Animal x) {return false;}
    abstract public boolean canFeedOnMilk();
}
```

Note that the abstract method **ends in semicolon**. It does not have associated code at all.

Abstract methods must be implemented in *concrete* subclasses. They can be implemented in abstract subclasses, though.

```
abstract class Mammal extends Animal {
    public boolean canFeedOnMilk() {return true;}
}
```

Since the abstract method was implemented in Mammal, it is also implemented in Dog. The **extends** keyword ensures that the code is passed down from the superclass to the class being defined.

INTERFACES AND MULTIPLE INHERITANCE

Java allows inheriting code from at most one class. This is a very strict scheme for subtyping. To circumvent this limitation, Java permits multiple *pseudo-inheritance* through *interfaces*. An interface defines the fields and methods that an *implementing* class must have. Interfaces are useful for defining protocols or setting requirements on classes.

Let us study interfaces with an example. Let us add the following code to our working example:

```
interface Aerial {
    public int height();
}

class Bat extends Mammal implements Aerial {
    public int height() {return 100;}
}
```

Now, Bat has types Mammal (and hence type Animal) and Aerial. Interface Aerial demands implementation of method **height()**.

Let us add the following lines to the main method, to verify the types Bat:

```
Animal a = new Bat();
Mammal b = new Bat();
Bat c = new Bat();
Aerial d = new Bat();
```

Henceforth, Bat can assume four types as opposed to Dog, which can only assume three.

Doing inheritance also inherits compliance with interfaces. Let us define class FruitBat to illustrate this:

```
class FruitBat extends Bat {
    public int height() {return 50;}
}
```

In the main, the following line is admitted:

```
Aerial e = new FruitBat();
```

FruitBat has just inherited type Aerial without explicitly having *implements Aerial* in its definition. As already mentioned, subtyping implies inheriting the types of the supertypes.

Now, let us add the following lines of code, to define even more classes:

```
class Penguin extends Bird {}

class Duck extends Bird implements Aerial {
    public int height() {return 10000;}
}
```

Multiple inheritance permits the creation of rather intricate subtyping structures:

- We have three concrete classes of type Aerial: Bat, FruitBat, Duck
- We have three concrete classes of type Mammal: Dog, Bat, FruitBat
- We have two concrete classes of type Bird: Penguin, Duck

Casting

- Let us look at the following code:

```
Animal x = new Dog();
x.fetchBranch();
```

- Error—fetchBranch belongs in **Dog**, not in **Animal**
- Correct:

```
Dog x = new Dog();
x.fetchBranch();
```

- Correct:

```
Animal x = new Dog();
( (Dog)x ).fetchBranch();      // CAUTION, CAUTION
```

- The following line is dangerous

```
Animal y = new Bat();
( (Dog)y ).fetchBranch();     // compiles, but raises error during execution!
```

- This is a dynamic error, meaning that the compiler was not able to spot the error during compile-time
- Can we check if an object can assume a type? **YES!** Using **instanceof**:

```
Animal x = new Dog(), y = new Bat();
System.out.println( x instanceof Dog );    // true
System.out.println( y instanceof Dog );    // false
```

Now, **instanceof** checks whether an object can assume certain types, so we also have that:

```
System.out.println( x instanceof Dog );       // true
System.out.println( x instanceof Mammal );    // true
System.out.println( x instanceof Animal );    // true
```

<< Lecture 13 will resume from here >>

(Working code so far)

```
abstract class Animal {
    boolean isalive = true;
    int age;

    public boolean isAlive() {return isalive;}
    public void killIt() {isalive=false;}
    public boolean canFeedOn(Animal x) {return false;}
    abstract public boolean canFeedOnMilk();
}

class Mammal extends Animal {
    public boolean canFeedOnMilk() {return true;}
}

abstract class Bird extends Animal {
    public boolean canFeedOnMilk() {return false;}
}

class Dog extends Mammal {
    public void fetchBranch() {}
}

interface Aerial {
    public int height();
}

class Bat extends Mammal implements Aerial {
    public int height() {return 100;}
}

class FruitBat extends Bat {
    public int height() {return 50;}
}

class Penguin extends Bird {}

class Duck extends Bird implements Aerial {
    public int height() {return 10000;}
}
```