<< Lecture 13 >>

Preamble

- **QUIZ**—too few quizzes so far: attendance + grade
- **FINAL EXAM SCHEDULED!**
  - **16 December 2015, Wednesday**
  - **From 5:30 to 7:30 pm**
  - **Room 40 SH**, i.e., **HERE!!!**
- http://math.hws.edu/javanotes/ Excellent **free** book
- Job Fair Season http://careers.uiowa.edu/
- **Covered contents:**
  - In class review of recent contents—all through questions
  - The **super** keyword, and constructors
  - Visibility (access rights) in inheritance
  - Covariance and contravariance **(to be continued next lecture)**

Review of polymorphism so far

Let us review polymorphism through questions.

1. What is *polymorphism*?
   - **Ans.** The ability of objects/values of one type to also have or assume other types.
2. What is *subtyping*?
   - **Ans.** It is the situation when a type *covers* another type(s), called subtype(s). An object/value of a subtype always have the supertype as type as well, but the converse is not true.
3. What is *ad-hoc polymorphism*?
   - **Ans.** It is the ability of methods to *behave differently* (function differently, have different code) upon receiving arguments of different types.
4. What is *multiple subtyping*? How does Java allow it?
   - **Ans.** Multiple subtyping occurs when a type subtypes at least two other (super)types in parallel, without them having a subtyping relation. Single subtyping generates *trees* of subtypes while multiple subtyping generates more complex structures. (The exact topology is that of a *polytree* or *directed acyclic graph* (DAG), which you might study or have studied in discrete structures. DAGs do not have cycles. In our context, a subtype cannot be its supertype.)

5. Provide a code example of ad-hoc polymorphism
    - **Ans.** I think I wrote a very similar piece of code in class. It is an oversized example for the question, however; ad-hoc polymorphism is not necessarity related to subtyping. But I wrote something like this to make three points:
        1. Primitive types can be interexchanged with their wrapper classes, for example, you can use **int** for integers, or **Integer** for integers. You can exchange both seamlessly, but the former is a primitive type while the later is a class.
        2. All numbers subtype **Number**. That is, **Integer**, **Short**, **Double**, etc., subtype **Number**.
        3. You can pick the exact behavior (method) by setting the type at hand using casting. This is done, for example, in instruction **prints((Number)5)**.

```
class Question5 {

    public static void prints(Number x) {
        System.out.println(x + " is a number");
    }

    public static void prints(Integer x) {
        System.out.println(x + " is an integer");
    }

    public static void prints(String x) {
        System.out.println(x + " is text");
    }

    public static void main(String[] args) {
        prints("5");
        prints(5);
        prints(5.5);
        prints((Number)5);
    }

}
```

6. Provide a code example of multiple subtyping
    - **Ans.** Easy, you could just do something like:

```
interface IFace1 {}
interface IFace2 {}
class MultipleSubtype implements IFace1, IFace2 {}
```

7. Mention two differences between **abstract class** and **interface**
    - **Ans.** There are various differences between them. For example, while neither can be instantiated, an abstract class can have some implemented methods, while an interface cannot have this. Another difference is that you can inherit multiple interfaces by a single class. Yet another difference is that that interfaces cannot subtype classes while abstract classes can subtype both. Et cetera...
8. (Unfair question) Can an interface implement another interface?
    - **Ans**. No. Interfaces cannot implement code, so interfaces can only *extend* other interfaces. This is very strange, but this is the Java way...

9. What does the following line print?

```
System.out.println( (Alpha)new Beta() );
```

Where:

```
class Alpha {
  public String toString() {return "Alpha";}
}
class Beta extends Alpha {
  public String toString() {return "Beta";}
}
```

- **Ans.** It prints **Beta**. Why? Because **casting** (which is when you do **(type)sth** in the code) only changes the manifest type, **not the code**. Class Beta subtypes Alpha, so objects of type Beta can be used as of type Alpha or Beta. But they still have the same code.

THE SUPER KEYWORD

- The **super** keyword works very similarly to the **this** keyword, except that it calls the version of the methods available in the superclass (extends)
    - *Simple example with methods*

```
class BigGuy {

    public String toString() {
        return "I'm the BigGuy";
    }

}

class BigGuyJr extends BigGuy{

    public String toString() {
        return super.toString() + "'s kiddo";
    }

}
```

Then, the following line does:

```
System.out.println( new BigGuy() );       // prints I'm the BigGuy
System.out.println( new BigGuyJr() );     // prints I'm the BigGuy's kiddo
```

By the way, in the lines above, **new** returns an object of type **BigGuy**/**BigGuyJr**. Then, **System.out.println** calls the **toString** method inside.

For the second println, the execution goes like this: upon calling **toString()** in the object of type **BigGuyJr**, we are returning **super.toString()** + some text. So, we access the method in the superclass, **BigGuy**, and use its **toString()** method.

- Keyword **super** acquires an additional meaning and usage within constructors

CONSTRUCTORS

- We can call alternative constructors in classes by using **this(**arguments**)**
  - *Example of alternative constructors:* **this**

```
class MakeMe {

    String name;

    public MakeMe(String x) {
        name = x;
    }

    public MakeMe() {
        this("I have no name...");   // using this as constructor
    }

    public String toString() {
        return name;
    }
}
```

Now, let us check how the code above works:

```
System.out.println( new MakeMe("Fido") );       // Fido
System.out.println( new MakeMe() );             // I have no name...
```

- You can do **this(...)** only in the first line of the constructor
  - it does not make sense anywhere else, anyways
  - also, it cannot be called outside a constructor, of course
- When extending a class (extends), the concrete subclass' constructor will call, by default, the empty constructor of the superclass

```
class SuperGuy {
    public SuperGuy() {
        System.out.println("The super guy has been created");
    }
}
class SubGuy extends SuperGuy {
    public SubGuy() {
        System.out.println("The sub guy has been created");
    }
}

public class NewMain {
    public static void main(String[] args) {
        new SubGuy();         // prints  The super guy... and The sub guy...
    }
}
```

- This works even when no constructor is explicitly defined

```
class SuperGuy {
    public SuperGuy() {
        System.out.println("The super guy has been created");
    }
}
class SubGuy extends SuperGuy {}
```

```
public class NewMain {
    public static void main(String[] args) {
        new SubGuy();         // prints The super guy has been created
    }
}
```

- The above works that way because **the default constructor just calls the superclass' constructor**
    - When you are not extending a class, well, you are actually extending **Object**, whose default constructor is empty; this is why you can get away without defining a constructor
- If the superclass does not have an empty constructor, implicit calling does not work!

```
// this code does not compile
class SuperGuy {
    public SuperGuy(int x) {
        System.out.println("The super guy has been created with "+s);
    }
}
class SubGuy extends SuperGuy {
    public SubGuy() {   // this constructor is invalid
        System.out.println("The sub guy has been created");
    }
}
```

- In such case, we need to call the superclass' constructor explicitly
- How can we call the superclass' constructor? Much like **this(...)**, we can use **super(...)**
    - It must be called in **first line** of the constructor (as with **this(...)**)

```
// now, this code works
class SuperGuy {
    public SuperGuy(int s) {
        System.out.println("The super guy has been created with "+s);
    }
}
class SubGuy extends SuperGuy {
    public SubGuy() {
        super(0);   // adding this line made the constructor valid
        System.out.println("The sub guy has been created");
    }
}

public class NewMain {
    public static void main(String[] args) {
        new SubGuy();
    }
}
```

- How could we get away without using **super(...)**? In turns out that **super()** is called implictly in the same line, unless **super(...)** is explicitly called

```
// this code has a redundant line...
class SuperGuy {
    public SuperGuy() {
        System.out.println("The super guy has been created with ");
    }
}
```

```
class SubGuy extends SuperGuy {
    public SubGuy() {
        super();        // this line is absolutely optional
        System.out.println("The sub guy has been created");
    }
}

public class NewMain {
    public static void main(String[] args) {
        new SubGuy();
    }
}
```

- *Lesson*: if the superclass' constructor is not empty, you **always** need to call it
- *Lesson*: the superclass' constructors are always called, in total inexorableness...

VISIBILITY AND INHERITANCE

- Let us refresh visibility, but about inheritance
  - **public**: available to the "world"
  - **default**: available to the package
  - **protected**: available to the subclasses
  - **private**: available only to the class, not to subclasses
- *Example: protected versus private*

```
// make field x private to get an error
class SuperGuy {
    protected String x = "SuperGuy's field";
}
class SubGuy extends SuperGuy {
    public SubGuy() { System.out.println(x); }
}
public class NewMain {
    public static void main(String[] args) {
        new SubGuy();
    }
}
```

COVARIANCE AND CONTRAVARIANCE

- Consider superclass SuperClass with method funk:

```
abstract class SuperClass {
   abstract public TypeU funk( TypeV arg );
}
class SubClass extends SuperClass {
   public TypeX funk( TypeY arg ) {…};
}
```

<< We ended today's lecture with the question: what are the relations between **TypeU** and **TypeX**, and between **TypeV** and **TypeY**? **Lecture 14 will resume from here>>**