Object-Oriented Design	Lecture 9
CSU 370 Fall 2007 (Pucella)	Friday, Oct 5, 2007

Inheritance Subtleties

Last time, we saw inheritance as a way to re-use code in a subclass.

Ther are some general rules for what is accessible from an inheriting subclass, and what is not.

Given an object A of a class T inheriting from S. The basic idea is that object A has all the fields and methods of T, as well as all the public and protected fields and methods of S.

- The constructor of T, when constructing A, will invoke the constructor of S, meaning that the latter has to be protected or public. This invocation can be explicit by using the syntax `super (arg1,...,argk);` as the first line in the constructor body in T. If not such call is made, then the constructor of S is invoked automatically by the compiler, with no arguments. (Meaning that S must implement a protected or public constructor taking no arguments for this to compile.)
- Every time you invoke a method on A, the method code is looked up in the definition of T. If there is no such definition, then the method is looked for in S, and it is found only if it is protected or public. It is important that the *first* definition found is executed. This lets you overwrite a definition of a method in a subclass. (This is what happens for the canonical methods; the defaults are defined in class `Object`, but you are welcome to overwrite.) The overwriting defintion can invoke the superclass's method by invoking `super.`method (...).
- Fields are more complicated. An object of class T can refer to a field defined in S, as long as that field is protected or public. Field shadowing—defining a field in a subclass that is also defined in the superclass—is the field-equivalent of method overwriting, except that the rules are much more painful to remember. Don't shadow fields unless you know what you are doing.[1]

Method overwriting, in combination with dynamic method lookup, is very powerful. It lets you, for instance, have an array of objects and iterate over the array invoking a particular method, and possibly get different behaviors based on the object actually stored in the array.

Recall the Person/Student/Faculty example of several lectures back. Suppose that add a field and method to the Faculty class, to keep track of salary:

[1]See `http://articles.techrepublic.com.com/5100-22_11-5031837.html`, for instance.

```
public class Faculty extends Person {
  ...

  private int salary = 20000;

  public int getSalary () {
    return this.salary;
  }

  ...
}
```

and suppose that we create a subclass `SeniorFaculty` that inherits from `Faculty`:

```
public class SeniorFaculty extends Faculty {
  ...

  public int getSalary() {
    return super.getSalary() + 10000;
  }
}
```

If we have an array `Faculty[] department;` initialized with some `Faculty` objects and some `SeniorFaculty` objects, we can write code to sum up all the salaries:

```
int total = 0;
for (int i=0; i<department.length;i++)
  total = total + department[i].getSalary();
```

and, most importantly, the *right* `getSalary` method will be invoked, depending on the actual class of the object being looked at.

There are some subtleties with how inheritance works in general, and in Java in particular. We already saw the issues with method and field access, requiring the need for a protected qualifier, and the difficulty with field shadowing.

For another subtlety, let's build up another example.

Recall the stack signature we had in lecture 7. Here is an implementation, a rather dirty one at that. (We'll see a better one in a bit.)

```
public class Stack {
  private Integer topVal;
  private Stack rest;
```

```
    protected Stack (Stack s, Integer i) {
      topVal = i;
      rest = s;
    }

    public static emptyStack () {
      return new Stack (null, null);
    }

    public static push (Stack s, int i) {
      return new Stack (s,i);
    }

    public boolean isEmpty () {
      return (this.topVal==null);
    }

    public int top () {
      if (this.isEmpty ())
        throw new RuntimeException ("top of empty stack");
      return this.topVal;
    }

    public Stack pop () {
      if (this.isEmpty ())
        throw new RuntimeException ("pop of empty stack");
      return this.rest;
    }
  }
```

Let's extend the Stack ADT. Suppose we really cared about keeping track of length of stacks in some application. A measurable stack has the following interface, an extension of the Stack ADT.

```
public static MStack emptyStack ();
public static MStack push (MStack, int);
public boolean isEmpty ();
public int top ();
public MStack pop ();
public int length ();

MStack.push(s,i).top() == i
MStack.push(s,i).pop() == s
MStack.push(s,i).isEmpty() == false
MStack.emptyStack().isEmpty() == true
MStack.emptyStack().length() = 0
MStack.push(s,i).length() = 1 + s.length()
```

The naive implementation of a length method by simply counting how many elements are in the stack can be inefficient if the stack is large. There is no way to make the "count how many elements are in the stack" algorithm more efficient, but there is a way to implement measurable stacks to make the `length` method more efficient: keep a count of the current stack size alongside the stack content, and simply increment the count upon a `push`. The `length` method now simply returns the current stack size, a constant-time field lookup operation. This is an example of an *augmented data structure*, a data structure augmented with information that make some operations more efficient.

It makes sense to want to implement measurable stacks, which clearly should be a subclass of stacks, by inheriting from stacks:

```
public class MStack extends Stack {
  private int count;

  private MStack (MStack s, Integer i, int c) {
    super (i,s);
    count = c;
  }

  public static MStack emptyStack () {
    return new MStack (null,null,0);
  }

  public static MStack push (MStack s, int i) {
    return new MStack (s,i,1+s.length());
  }

  public int length () {
    return this.count;
  }

  public MStack pop () {
    return (MStack) super.pop ();
  }
}
```

It all goes as expected, except for the `pop` method. That method does nothing special in the `MStack` class—all the action is in the `Stack` class. But the types are not right. The `pop` method should return an `MStack`. But we know by construction that what gets stored in the `rest` field when constructing an `MStack` is a measurable stack (although the field it is stored in has type `Stack`), so we need to re-implement the `pop` method in `MStack` to simply "correct" the type of the returned value.

This is a limitation of Java, and a subtlety to be aware of, that it requires you to jump through such hoops when inheriting from classes that have a method returning an object of the class itself.

Subclassing for ADTs Implementation

Subclassing is useful for expressing related ADTs, in particular, ADTs that extend others. Inheritance is a useful implementation technique for subclassing.

There is another useful use of subclassing and inheritance, and that is to implement some forms of ADTs more cleanly, especially ADTs that have different representations for their values.

Take the stack as an example. The representation we have used is ugly, and I mean that as a technical term. Part of the problem is that there is an implicit invariant: whenever the `topVal` field is not null, then the `rest` field better not be null either. (Can you see where the problem lies if this is not the case?) We must make sure that the implementation always preserves that invariant. There are ways around that, making the invariant more robust, but they're a bit unsatisfying.

Another problem with the implementation is that there are all kinds of checks in the code that test whether the stack is empty or not. It would be much better to instead have two kinds of stacks, an empty stack and a "push stack", and have them implement their respective methods knowing full well what their representation is. This is what we're going to explore now. We will use subclasses to keep track of the kind of stack we have, and the subclasses will implement their methods in full confidence that the stack they are manipulating is of the right kind, without needing to check the representation.

But that means that the `Stack` class, by itself, does not represent anything. The representation is in the subclasses. It does not make sense to create an object of type `Stack` anymore. We will only create an object of class `EmptyStack` or `PushStack`, as they will be named. To enforce this, we are going to make the `Stack` class `abstract`. An abstract class is a class that cannot be instantiated.[2] Therefore, it does not have a Java constructor. The only use of an abstract class is to serve as a superclass for other classes. An abstract class also need not implement all its methods. It can have abstract methods that simply promise that subclasses will implement those methods. (Java will enforce this, by making that all concrete subclasses of the abstract class do provide an implementation for the methods.)

[2]In contrast, a class that can be instantiated is sometimes called a *concrete* class.

Here is a recipe for implementing an immutable ADT that is specified by an algebraic specification using subclasses.

Let T be the name of the ADT.

Assumptions on the structure of the ADT interface:

- Assume that the signature is given in OO-style: except for the basic creators, each operation of the ADT takes an implicit argument which is an object of type T.
- Except for the basic creators, each operation is specified by one or more equations. If an operation is specified by more than one equation, then the left hand sides of the equations differ according to which basic creator was used to create an argument of type T.
- The equations have certain other technical properties that allow them to be used as rewriting rules.
- We are to implement the ADT in Java.
- The creators of the ADT are to be implemented as static methods of a class named T.
- Other operations of the ADT are to be implemented as methods of a class named T.

Steps of the recipe:

- Determine which operations of the ADT are basic creators and which are other operations.
- Define an abstract class named T.
- For each basic creator c of the ADT, define a concrete subclass of T whose instance variables correspond to the arguments that are passed to c. For each such subclass, define a Java constructor that takes the same arguments as c and stores them into the instance variables.

  (So far we have defined the representation of T. Now we have to define the operations.)
- For each creator of the ADT, define a static method within the abstract class that creates and returns a new instance of the subclass that corresponds to c.
- For each operation f of the ADT that is not a basic creator, define an abstract method f.
- For each operation f of the ADT that is not a basic creator, and for each concrete subclass C of T, define f as a dynamic method within C that takes the arguments that were declared for the abstract method f and returns the value specified by the

algebraic specification for the case in which Java's special variable `this` will be an instance of C. If the algebraic specification does not specify this case, then the code for f should throw a `RuntimeException` such as an `IllegalArgumentException`.

Following this recipe for the Stack ADT above gives the following. Can you match the steps with what gets produced?

```
public abstract class Stack {

  public static Stack emptyStack () {
    return new EmptyStack ();
  }

  public static Stack push (Stack s, int i) {
    return new PushStack (s,i);
  }

  public abstract boolean isEmpty ();
  public abstract int top ();
  public abstract Stack pop ();

}


class EmptyStack extends Stack {

  public EmptyStack () { }

  public boolean isEmpty () {
    return true;
  }

  public int top () {
    throw new IllegalArgumentException ("Invoking top() on empty stack");
  }

  public Stack pop () {
    throw new IllegalArgumentException ("Invoking pop() on empty stack");
  }
}


class PushStack extends Stack {
    private int topVal;
    private Stack rest;

    public PushStack (Stack s, int v) {
      topVal = v;
      rest = s;
    }

    public boolean isEmpty () {
      return false;
    }

    public int top () {
      return this.topVal;
    }

    public Stack pop () {
      return this.rest;
    }
  }
```

The problem with this? How do you subclass to get MStacks?