Classes and Inheritance

CS151

Chris Pollett

Oct. 3, 2005.

Outline

- Overloading Methods and Constructors
- Extending Classes

Overloading Methods and Constructors

- Overloading is the ability to allow different methods or constructors of a class to share the same name. The name is said to be overloaded.
- Two methods or constructors in the same class can be overloaded, if: (1) they share the same name, but have either a different number of parameters, or (2) the name number of parameters but with different types. For example: Point might have two methods multiply():

```
public void multiply(Point p){/*code */}
and
public void multiply(double scalar){/* code */}
```

More on Overloading

- Some languages support operator overloading. For instance, +=, -=, etc in C++ can be overriden.
- One advantage of this is that it can make the code more succinct; one disadvantage is that operators can be overloaded in misleading ways.
- In general, one should only overload if one of the following applies:
  - there is a general similarity in functionality which is being provided by all the overloaded functions
  - some of the methods will supply default arguments for a common more general method.

Extending Classes

- Inheritance defines a relationship among classes.
- When C2 *inherits* or *extends* from C1, class C2 is called a **subclass** of C1 and C1 is called a **superclass** of C2.
- All public and protected methods of C1 will be accessible in C2.
- Interface extension and implementation can also be viewed as a weak kind of inheritance.
- The extension relation among classes forms a hierarchy with the class Object as its root. Every class other than Object has a unique superclass.

Constructors of Extended Classes

- Initialization of an extended class consists of two phases: (1) Initialization of the parent class. (2) Initialization of the fields of the current class.
- One of the constructors of the parent class must be called to initialize the fields of the parent:

```
class MySubclass extends MyClass
{
     public int mySubInt;
     public MySubclass(int xParent, xSub)
     {
        super(xParent);
        mySubInt = xSub;
        // do other stuff.
     }
     public MySubclass()
     { mySubInt =0; // no-arg constructor of parent implicitly invoked
     }
}
```

Order of Initialization

- The fields of the superclass are initialized, using explicit initializaers or default values
- One of the constructors of the superclass is executed
- The fields of the extended class are initialized using field initializers or default values
- One of the constructors of the child class is executed.

Subtypes and polymorphism

- One important characteristic of OO-programming is the dynamic binding of methods.
- The idea is we have several subclasses of some parent and each implementing some method differently.
- When we use instances, the code that actually gets called can be determined at run-time rather than compile time.
- To use this idea we need to say when one type (primitive like an int or defined like a class) can be substituted for another.

Subtypes

- Type T1 is a *subtype* of T2 if very legitimate value of T1 is a legitimate value of T2. T2 is called a *supertype* of T1.
- Subtypes have the property that wherever a value of a supertype is expected a value of a subtype can be used.
- The conversion of a subtype to its supertype is called *widening*. The reverse is called *narrowing*.

Polymorphic Assignment

- Consider:

  ```
  class Student {}
  class Graduate extends Student{}
  class Undergrad extends Student{}
  ```

- Then the following is legal

  ```
  Student s = new Graduate();//different subtypes of Student
  Student s = new Undergrad ();//are legal on RHS (polymorphism)
  ```

  But the following is not:

  ```
  Graduate g =new Student(); /*you could try an explicit cast
     (Graduate)(new Student()) but this might break elsewhere */
  ```

- The basic rule is that the right hand of assignment must be a subtype of the left hand side.

Subtyping and Arrays

- The following is legal:

```
Student as[];
//code
Student g = new Graduate();
as[1] = g;
```

Now suppose did :

```
Graduate g2 =  as[1]; /* would get a compilation error*/
```

Need to explicitly cast event though we know as[1] was originally a Graduate.

```
Graduate g2 =  (Graduate)as[1];
```