Inheritance and Polymorphism

Reading:
• Reges and Stepp: 9.1-9.4

Reminders
• Unity ID
  – Is NOT your student ID number (e.g. 000123456)
  – Is something like sesmith5
    • First letter of first name
    • First letter of middle name (if you have one)
    • First 6 letters of last name (may be shorter if your last name is shorter)
    • May end in a digit if there is already an ID with the same characters

• Google Forms: Use your NCSU account
  – I will not accept requests for you to access the form with a gmail account.
  – If you are already logged in with on Firefox, use Chrome or use a incognito/private browsing window

• Moodle Submission
  – Log In! You must be enrolled (and logged in) to submit assignments.
  – Never mark a submission “Send for Marking” unless otherwise specified to in the assignment
  – We need to “Revert your submission to draft” for you to make changes, which takes time and may not happen before a deadline.
  – No submissions via email!!!
John Vlissides (GoF)

“I’ve said it before and I’ll say it again: A hallmark – if not the hallmark – of good object-oriented design is that you can modify and extend a system by adding code rather than by hacking it. In short, change is additive, not invasive.”

Code Reuse

- When writing programs, we want to write common code once and use it many times and in many contexts
  - Find commonalities
  - Customize with any additional information
What are some commonalities in our Beer ingredients (hops, maltose, water, and yeast)?

Inheritance

- Increases ability to reuse code
  - One class is an extension of another
  - Hierarchies of related object types
- Ingredients
  - All our ingredients have a common idea of type, amount, and add time
  - Some ingredients add distinct flavors to beer
Is-a Relationship

- A hierarchical connection between two categories where one type can be treated as a specialized version of the other

- Hops is a specialized version of an Ingredient

- Inheritance Hierarchy: a set of hierarchical relationships between classes of objects.

Revised Beer Class Diagram
Definitions

• Inheritance (Inherit)
  – A programming technique in which a derived class extends the functionality of a base class, inheriting all of its state and behavior

• Superclass: the parent class in an inheritance relationship

• Subclass: the child class in an inheritance relationship

• Subclass extends the superclass

Single Inheritance

• In Java, a class can have ONLY ONE superclass

public class <subclass> extends <superclass> {
  
}

• A superclass may have many subclasses
**Subclasses**

- Are an instance of the super class
  - Superclass is constructed first
- Can have new state and add new behavior
  - Constructed/initialized after the superclass’ state and behavior
- If we don’t like the superclass’ behavior, we can override it
  - To implement a new version of a method to replace code that would otherwise have been inherited from a superclass
  - Write the method you want to replace in the subclass
  - Method name and signature MUST match exactly

**Constructing an Ingredient**

- When constructing an ingredient, like Maltose, you must
  - Construct the parent ingredient (FlavoredBeerIngredient)
  - Which must in turn create its parent ingredient (BeerIngredient)
  - And then construct the new information about Maltose
**Example Ingredient Implementation**

```java
public class Maltose extends FlavoredBeerIngredient {
    private String maltType;
    public Maltose(String type, double amount, double addTime, ArrayList<String> flavors, String maltType) {
        this.type = type; //Set BeerIngredient field
        this.amount = amount; //Set BeerIngredient field
        this.addTime = addTime; //Set BeerIngredient fld.
        this.flavors = flavors; //Set FBI fld.
        this.maltType = maltType; //Set Maltose field
    }
}
```

This implementation requires that the instance fields from `FlavoredBeerIngredient` and `BeerIngredient` are public so that Hops can “see” the data.

Does this preserve the encapsulation of `FlavoredBeerIngredient` and `BeerIngredient`?

---

**Redundancy (BAD!)**

- Our implementation takes advantage of inheritance to describe all parent fields
  - By inheriting `FlavoredBeerIngredient`
- To set superclass’ fields, it requires `BeerIngredient`’s instance fields to be public/protected
  - Violates encapsulation
- Each subclass constructor sets the `BeerIngredient` class’ instance fields
  - Violates code reuse paradigm by having `BeerIngredient` set its own fields
- Each subclass could construct the parent type differently
  - Violates “one path of construction” paradigm
- How could we fix?
Interacting with the Superclass

- Subclasses may not refer to private fields of the superclass
  - How do we access private fields of a class?
- Subclasses can call any of the public and protected methods and constructors of the superclass
  - What method or constructor should be used to create an instance of the superclass?
- Constructors in the subclass need to accept the same parameters as the superclass’ constructor in addition to parameters associated with specific data of the subclass
  - How would we create the superclass object?
- Constructors of superclass are NOT inherited
  - Want to continue principal of ONE path of construction

Interacting with the Superclass (2)

- Subclass’ constructors MUST call a superclass’ constructor
  - Assumes that superclass has parameterless (default) constructor
  - Parameterless constructor is called automatically from subclass’ constructor
  - If you don’t have a parameterless constructor – compilation error!
- super keyword provides a reference to the behavior of the superclass
  - Use to call superclass constructor
  - Use to call overridden methods
Example Ingredient Implementation
public class Maltose extends FlavoredBeerIngredient {

    private String maltType;
    public Maltose(String type, double amount, double
    addTime, ArrayList<String> flavors, String
    maltType) {
        super(type, amount, addTime, flavors);
        this.maltType = maltType;
    }
}

Overridden Methods

• If you override a method, and then want to call
  the superclass’ version of the method, you must
  use super to call the method
    – Do this if you want to use a value from the super
      class in your overridden calculation (reduces
      redundancy)

    //Override BeerIngredient.getType() in
    //FlavoredBeerIngredient to include flavors
    public String getType() {
        return super.getType() + flavors.toString();
    }
Polymorphism

- Legal for a variable of a superclass type to refer to an object of one of its subclasses
  
  `BeerIngredient i = new Hops();`

- Even though the variable is of type `BeerIngredient`, the object is a `Hops` object, and it will behave like a `Hops` object.

Legal Statements?

- `BeerIngredient i = new BeerIngredient();`
- `BeerIngredient i2 = new FlavoredBeerIngredient();`
- `BeerIngredient i3 = new Hops();`
- `FlavoredBeerIngredient fi = new Water();`
- `FlavoredBeerIngredient fi2 = new Hops();`
- `Hops h = new FlavoredBeerIngredient();`
- `Maltose m = new BeerIngredient();`
- `Yeast y = new Hops();`
- `Hops h2 = new Hops();`
Beer

- Therefore, our **Beer** class can really contain an **ArrayList of BeerIngredients**
  - We can treat the ingredients polymorphically – at least for the type, amount, and add time.

---

**Beer Implementation**

```java
public class Beer {
    private ArrayList<BeerIngredient> ingredients;

    public Beer() {
        ingredients = new ArrayList<BeerIngredient>();
    }

    public void addIngredient(BeerIngredient i) {
        ingredients.add(i);
    }

    public ArrayList<BeerIngredient> getIngredients() {
        return ingredients;
    }
}
```
Creating a Beer

```java
public class BeerMain {
    public static void main(String[] args) {
        // flavor ArrayLists created here
        Beer b = new Beer();
        b.addIngredient(new Water("Tap", 4, 60, 6.5));
        b.addIngredient(new Water("Mineral", 3, 0, 7.0));
        b.addIngredient(new Yeast("British Ale Liquid", 125, 0,
                                  yeastFlavors, 70, 85));
        b.addIngredient(new Hops("Centennial", 1, 60,
                                  centennialHopsFlavors));
        b.addIngredient(new Hops("Cascade", 1, 15,
                                  cascadeHopsFlavors));
        b.addIngredient(new Hops("Cascade", 1, 5, cascadeHopsFlavors));
        b.addIngredient(new Maltose("British Pale Ale", 3.3, 60,
                                    maltFlavors, "Dry"));
    }
}
```

Describing the Beer’s Ingredients

```java
ArrayList<BeerIngredient> ingredients = b.getIngredients();
for (int i = 0; i < ingredients.size(); i++) {
    BeerIngredient ing = ingredients.get(i);
    System.out.println(ing.toString());
}
```

- What’s printed?
Using Inheritance

• When should you use inheritance?
  – When you see similarities between classes that can be modeled in a hierarchy

• When should you NOT use inheritance?
  – When a subclass cannot be substituted everywhere a superclass could be used.

Has-a versus Is-a Relationships

• Has-a relationships are preferred over is-a relationships in cases where you class cannot or should not substitute for the other class
  – Most people choose to have and use software rather be a programmer
• Unless an object MUST be an extension of another, use composition!