Abstract Classes and Interfaces

Reading:
• Reges and Stepp: 9.5-9.6

Abstract Classes

• A Java class that cannot be instantiated, but instead serves as a superclass to hold common code and declare abstract behavior
  – Want a class with common code (to reduce redundancy) but the class should never be instantiated (doesn’t make sense in context of program)
  – Abstract methods: declared but not implemented – this is behavior that you promise to implement when you implement the interface
  – If every subclass of a superclass would override a particular method, consider making it abstract
A Deeper Look at Beer

- Beer is a very general name for lots of different types of beverages
- Beers fall into two high level categories
  - Ales: top-fermenting yeast with fermentation range around room temp
  - Lagers: bottom-fermenting yeast with fermentation range between 46-59°F
- Ales and Lagers would add different types of ingredients – particularly yeast

Beer Hierarchy

```
Beer
- name: String
  - Beer(name: String)
  - addIngredient(i: BeerIngredient): void
    - getIngredients(): Collection<BeerIngredient>
    - getName(): String

Ale
- ingredients: ArrayList<BeerIngredient>
  - Ale(name: String)

Lager
- ingredients: LinkedList<BeerIngredient>
  - Lager(name: String)
```
Beer Class

• Make the addIngredient() method abstract
• Why?
  – Ales and Lagers will each require different ingredients
  – Ales and Lagers may also want to store their ingredients in different data structures

Beer Class (Abstract)

```java
public abstract class Beer {  
    private String name;
    
    public Beer(String name) {  
        this.name = name;
    }
    
    public abstract boolean addIngredient(BeerIngredient i);
    
    public String getName() {  
        return name;
    }
}
```
Abstract Classes

• Any class with abstract methods must be declared abstract in class header
• Abstract classes can have constructors to initialize its state
  – Concrete classes should call those constructors to maintain paradigm of “one path of construction”
• Cannot create instances of classes that are abstract
  Beer b = new Beer(“Pale Ale”); //legal?
• Classes without abstract methods can still be abstract
  – Make classes abstract when you don’t want them instantiated
• Concrete methods in abstract classes may call abstract methods, even if there is no implementation of the abstract method in the file.
  – Concrete subclass is required to implement the abstract methods
  – Can only call superclass method if there is an instantiated concrete object – with the overridden methods!
• Document like concrete classes – Javadoc!

Ale Implementation (Concrete)

```java
public class Ale extends Beer {
    private ArrayList<BeerIngredient> ingredients;
    public Ale(String name) {
        super(name);
        ingredients = new ArrayList<BeerIngredient>();
    }
    public boolean addIngredient(BeerIngredient i) {
        if (i instanceof Yeast) {
            Yeast y = (Yeast)i;
            //Check to see if yeast’s fermentation range is that required for an ale, add (if appropriate)
            //and return appropriate value
        } else {
            ingredients.add(i);
            return true;
        }
    }
}
```

Concrete Classes

- Concrete classes **extend** abstract classes and must implement abstract methods
  - Single inheritance still applies for the concrete classes
- Concrete class constructors still call one of the superclass’ constructors
  - Pass appropriate parameters to superclass’ constructor to initialize superclass’ state
  - If no explicit call to a superclass constructor, try to go through the parameterless constructor

```java
public class BeerMain {
    public static void main(String[] args) {
        ArrayList<Beer> beersOnTap = new ArrayList<Beer>();
        beersOnTap.add(new Ale("Pale Ale");
        beersOnTap.add(new Lager("Pilsener");
        Yeast y = new Yeast("British Ale Liquid Yeast", 125, 0,
                            null, 70, 85);
        for (int i = 0; i < beersOnTap.size(); i++) {
            //Which line is printed for each Beer?
            if (beersOnTap.get(i).addIngredient(y)) {
                System.out.println("Yeast added to " +
                                   beersOnTap.get(i).getName());
            } else {
                System.out.println("Yeast NOT added to " +
                                   beersOnTap.get(i).getName());
            }
        }
    }
}
```
Pros and Cons of Abstract Classes

• Pros
  – Contains both concrete and abstract methods
    • Allows for common code
    • Provide common interface for functionality that all child classes should override
    • Guarantees functionality of abstract method(s) will exist when overridden
  – Allow for polymorphism

• Cons
  – Concrete classes can only extend one class (even if the extended class is abstract)

Drawbacks of Inheritance

• Java uses single inheritance
  – A class can only extend ONE class (only have ONE superclass)
  – Single inheritance prevents ambiguity if two parent classes share the same method signature(s)
  – Can’t set up multiple is-a relationships

• What if we want an is-a relationship without sharing code?
  – We want to guarantee shared functionality of objects that are the same thing, but we don’t care about the implementation of the functionality.
Interfaces

- A set of methods that classes can promise to implement, allowing you to treat those classes similarly in your code
  - An interface is a contract that a subclass MUST have the listed behaviors
- If an object implements an interface, the object has certain known behavior
  - Still can take advantage of polymorphism
- Classes can implement multiple interfaces

Interfaces (2)

- Interfaces define what something can do
- A class defines what something is

- Example Interfaces
  - Runnable: threads
  - Comparable: ordering and comparing objects
  - Observable: responding to UI events
  - Serializable: objects are saved to files
  - List, Set, Map, and Iterator: data structures
What are things Beer can do?

- What are the adjectives associated with Beer?
  - Drinkable
  - Brewable
  - Fermentable

Beer Class Diagram Revised
Brewable Interface

/**
 * Interface for objects that can be brewed
 * @author Sarah Heckman
 */

public interface Brewable {
    /**
     * Brew the object
     */
    void brew();
}
Revised Beer Class

```java
public abstract class Beer implements Brewable, Fermentable {
    private String name;
    public abstract boolean addIngredient(BeerIngredient i);
    public String getName() { return name; }
    public String setName(String name) {
        this.name = name;
    }
    public void brew() {
        //order ingredients by add time (descending)
        //and brew a delicious beverage!
        System.out.println("I’m brewing!!");
    }
    public void ferment() {
        System.out.println("I’m fermenting!!");
    }
}
```

Interfaces

- Interfaces can be used with other things not in the Beer hierarchy
  - What else can you brew?
Interfaces and Abstract Classes

- Abstract classes (like Beer) can implement interfaces which means all child classes automatically inherit the interface behaviors
  - Even better, you can choose NOT to implement the abstract method from the interface
  - Requires that the concrete child of the abstract class MUST implement the interface’s behavior

Heckman Brewery Revisited

```java
public class HeckmanBrewery {
    public static void main(String[] args) {
        ArrayList<Beer> beersOnTap = new ArrayList<Beer>();
        beersOnTap.add(new Ale("Pale Ale"));
        beersOnTap.add(new Lager("Pilsener"));

        for (int i = 0; i < beersOnTap.size(); i++) {
            //add ingredients
            beersOnTap.get(i).brew();
        }
    }
}
```
Benefits of Interfaces

- Implementing a common interface provides a type hierarchy similar to inheritance
  - Smart, uniform design
- Use to achieve polymorphism
  - All Beers have a common set of functionality
    - Brewable
    - Fermentable
- Example of additive not invasive
- Bi-pass limitations of single inheritance
- Specifies behaviors you want your class to have!
  - Think “something-able”

The **Object** Methods and Inheritance

- **Object** is a (direct or indirect) superclass of all Java classes

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String toString()</code></td>
<td>Returns a String representation of the object.</td>
</tr>
<tr>
<td><code>boolean equals(Object obj)</code></td>
<td>Indicates whether some other object is “equal to” this one.</td>
</tr>
<tr>
<td><code>int hashCode()</code></td>
<td>Returns a hash code value for the object.</td>
</tr>
<tr>
<td><code>protected Object clone()</code></td>
<td>Creates and returns a copy of this object.</td>
</tr>
<tr>
<td><code>Class&lt;?&gt; getClass()</code></td>
<td>Returns the runtime class of this <code>Object</code>.</td>
</tr>
</tbody>
</table>
Generalizing `toString()` in Superclasses

- To make `toString()` useful by subclasses

```java
public String toString() {
    return getClass().getName() + // info on fields;
}
```

- `getClass()` is a method of `Object` that returns a `Class` object. The `getName()` method returns the name of the class.

- The `toString()` on a child class should also be overridden to include info on the child’s fields – use `super.toString()` to access the parent.

equals() and `hashCode()` with Inheritance

- Always handle the superclass first!

```java
public class FlavoredBeerIngredient {
    public boolean equals(Object o) {
        // Test the superclass for equality first!
        if (!super.equals(o)) return false;
        if (getClass() != o.getClass()) return false;
        FlavoredBeerIngredient f = (FlavoredBeerIngredient)o;
        // test fields
    }
}
```
The `clone()` method

- Use the `clone()` method if you want to make a copy of an object
  - Because copying an object reference gives you two references to the SAME underlying object
- `clone()` returns a new (as in stored in a different location) object that has identical state to the existing object
  - `clone()` creates a shallow copy – object fields are references to the objects not new copies of the objects
  - Must override `clone()` to create a deep copy
- `clone()` returns an object of type `Object`
  - Must cast to the type of object you want
    
    ```java
    Hops h = (Hops) oldHops.clone();
    ```
    - Could potentially lead to `ClassCastException`

Immutable Objects and `clone()`

- Use the `clone()` method to make copies of mutable objects in accessor methods of classes
  - Prevents the breaking of encapsulation

```java
public class FlavoredBeerIngredient {
    private ArrayList<String> flavors;
    public ArrayList<String> getFlavors() {
        return (ArrayList<String>)flavors.clone();
    }
}
```
Overriding `clone()`

- The `Object` class’ implementation of `clone()` has safeguards
  - `clone()` is protected so can’t be called accidentally
    - Implementation of `clone()` must be public
  - Class must be an instance of Cloneable
  - Must catch `CloneNotSupportedException`

- If you want a deep copy, then object references must be cloned as well
- `Object.clone()` does the right thing for primitives

```java
public class FlavoredBeerIngredient implements Cloneable {
    public Object clone() {
        try {
            FlavoredBeerIngredient cloned = (FlavoredBeerIngredient) super.clone();
            cloned.flavors = (ArrayList<String>) flavors.clone();
            return cloned;
        } catch (CloneNotSupportedException e) {
            //Always have to catch, even if implement Cloneable
            return null;
        }
    }
}
```

Clone() Example
References

• Dr. Jo Perry’s lecture notes