OO Review and Composition

BJP: Chapters 1-8

Object Dereferencing Review

• Exercise 01.01
Object-Oriented Programming

- Reasoning about a program as a set of objects rather than as a set of actions
- Object: A programming entity that contains state and behavior
  - State: internal data of an object (fields or instance variables)
  - Behavior: set of actions performed by the object (instance [non-static] methods)
- Class: a blueprint for an Object

Constructors

- Classes can have multiple constructors
  - Different signatures – number and type(s) of parameters
- Default constructor (no parameters) automatically available if you have no defined constructors
- After you define one constructor, you have to write a default constructor
- Programming Paradigm: one constructor contains true initialization code – all other constructors call it
  - Every object goes through a common code path
Object Construction

Book b = new Book(“Harry Potter”, “J. K. Rowling”, 2007, 7);

1. Creates a Book reference, b
2. Creates a Book object
3. Calls one of the possible many Book constructors on new object passing parameters
4. Assigns the newly created object to be stored in reference variable b

Reference Semantics

- **Reference Semantics**: behavior where variables refer to a common value when assigned to each other or when passed as a parameter
  - Object variables don’t store the object directly; instead they store the object's location in memory
- Objects in Java use reference semantics
  - **Efficiency**: copying large objects takes time
  - **Sharing**: can share information when calling methods
Reference Semantics

```java
public class Library {
    public void checkOut(Book b) {
        ... //assume that field of b is changed that
        //shows book is checked out
    }
    public static void main(String[] args) {
        Book b1 = new Book(...);
        Book b2 = new Book(...);
        Book b3 = b2;
        b3.setName("Inferno");
        Library l = new Library();
        l.checkOut(b2);
        if (b3.isCheckedOut()) { ... } else { ... }
    }
}
```

Dereferencing an Object

- **Dereference**: to access data or methods of an object
  - Use the dot notation
  - Specify which object to dereference and what field or method to access

```java
Book b = new Book(...);
String t = b.getTitle().toUpperCase();
b.setTitle("...");
```
Null References

- **null**: A built in value that does not refer to any object
  - Uninitialized object variables are set to `null`
- Uses of null
  - Initialize a local object variable
    ```java
    Book b = null;
    ```
  - Check if an object is null
    ```java
    if (b == null) { ... }
    ```
  - Pass null as a parameter
    - Useful in some cases when use of parameter is unknown
    - Be prepared for that method to throw an exception!
  - Return null from a method
    - Indicate failure

NullPointerException

- **null** cannot be dereferenced!
  - `null` has no methods or data

```java
Book b = new Book(); //default const.
String s = b.title().toUpperCase();
```

![Diagram](image)
Composition and Delegation

• Creating a class from existing classes
  – Classes have members (fields) that are instances of other classes
• Our class works with the object fields by calling methods (and fields) of the object
  – Delegation is a way of extending and reusing a class
  – An instance of the original class provides functionality for the class you’re writing
  – You’re delegating to the field through its methods

Abstraction & Encapsulation

• Abstraction: Distancing between ideas and details
  – Can use objects without knowing how they work
  – Example: iPod
• Encapsulation: Hiding the implementation details of an object from the clients of the object
  – Protects data from unwanted access
  – Clients cannot directly access or modify its internal workings – nor do they need to do so
  – Encapsulation leads to abstraction
Steps for Achieving Encapsulation

• Make fields private
• Ensure class invariants
  – A property that is true of every object of a class
  – An assertion about an object’s state that is true for the lifetime of that object
  – Enforce within public methods and call public methods within class (e.g. from constructor)
• If class contains Object fields, defensively copy the object to ensure immutability

Method Overriding

• Every object you create is an Object
• Object has useful methods that you can override
  – toString()
  – equals()
  – hashCode()
The Object Class

- Object is a (direct or indirect) superclass of all Java classes
- Contains nine methods – we’ll focus on 3

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>String toString()</td>
<td>Returns a String representation of the object.</td>
</tr>
<tr>
<td>boolean equals(Object o)</td>
<td>Indicates whether some other object is “equal to” this one.</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns a hash code value for the object.</td>
</tr>
</tbody>
</table>

The toString() method

- Called automatically when ever a String is concatenated with an Object
- If you don’t override toString(), the default output is something like
  - “ClassName@d24606bf”
  - Name of the class followed by the hash code
- Have to override toString() so that interesting and relevant information is printed
  - Convention: “ClassName[fieldName=fieldValue,…]"
The `equals()` method

- Called when want to compare objects
- Default behavior of `equals()`?
  - The same thing as `==`, which compares object references to see if two objects are the SAME object
- The `equals()` method has a parameter of type `Object`
  - We must cast the parameter to the same type of object as the class to compare the fields

The `hashCode()` method

- Method that uses an algorithm to compute an unsigned integer value that is likely to be unique for different objects
- When two or more objects that we would consider different have the same hash code we have a collision
  - Try to write hash functions that minimize collision
  - Hash functions scramble info about the instance fields
    - Typically prime numbers are used
- `hashCode()` *should be compatible with* `equals()`
  - Objects that are the same should hash to the same value
  - Use the same fields to compare for equality and create the hash code
- *If you override* `equals()`, *you must override* `hashCode()`
equals() and hashCode()

- Let Eclipse write your equals() and hashCode() methods for you
- For exam, know how to write equals() methods
  - Suggest using syntax template from book
- Do not need to know how to write hashCode() method, but need to know what it is

Implicit Parameter

- The object being referenced during an instance method call
- Implied knowledge of what object the method is operated on
- We can access the implicit parameter using the keyword this
- Calling a field, like x, is compiled to this.x
Welcome to the Heckman Brewery

- The Heckman Brewery has hired you and your classmates to create an automated brewery system. Given a beer recipe, the Heckman Brewery system will automatically brew a delicious beverage for future enjoyment!

- What makes a Beer?
Hops Implementation (POJO)

```java
import java.util.ArrayList;
public class Hops {
    private String type;
    private int amount;
    private int addTime;
    private ArrayList<String> flavors;

    public Hops(String type, int amount, int addTime,
                 ArrayList<String> flavors) {
        this.type = type;
        this.amount = amount;
        this.addTime = addTime;
        this.flavors = flavors;
    }

    public String getType() {
        return this.type;
    }
    public int getAmount() {
        return this.amount;
    }
    public int addTime() {
        return this.addTime;
    }
    public ArrayList<String> getFlavors() {
        return this.flavors;
    }
    public boolean equals() { ...
    public int hashCode() { ...
```

Hops Implementation (POJO) (2)

```java
public String getType() {
    return this.type;
}
public int getAmount() {
    return this.amount;
}
public int addTime() {
    return this.addTime;
}
public ArrayList<String> getFlavors() {
    return this.flavors;
}
public boolean equals() { ...
public int hashCode() { ...
```
Beer Implementation

```java
public class Beer {
    private ArrayList<Hops> hops;
    private ArrayList<Maltose> maltose;
    private Water water;
    private Yeast yeast;
    private String brewMethod;
    //default constructor
    public Beer(String brewMethod, ArrayList<Hops> hops,
                 ArrayList<Maltose> maltose, Water water,
                 Yeast yeast) {
        this.brewMethod = brewMethod;
        ...
    }
    public void brew() {
        water.addWaterToBrew(); //Delegation
    }
}
```

Composition Summary

- Using objects within another class allows for the creation of custom objects
- Start with primitive types and build larger objects from those types
- The composition class can define its own methods to call the methods of its instance variables
Class Design Paradigm

- Do not provide any functionality that does not have a clear use
- Omit mutators that have no use
- Limit object creation to the constructor
- “Classes should be immutable unless there’s a very good reason to make them mutable... If a class cannot be made immutable, limit its mutability as much as possible.” Joshua Bloch

Class Design Paradigm

- Classes should have **cohesion**
  - The extent to which the code for a class represents a single abstraction
  - Allows for reusability of the class in other programs
- The Book class represents only things a Book can do and knows.
  - A Book should not perform console input and output
  - A Book should not ensure the immutability of the due date.
Program Design Paradigms

- Classes should not have unnecessary dependences
  - Coupling is the degree to which one part of a program depends on another
- Related data and behavior should be in the same place (same class)

Exercise and Knowledge Survey

- Complete Exercise 01.04: Syllabus Review
  - Due August 26 @ 9:00pm
- Complete CSC216 Knowledge Survey
  - Understand your background knowledge
  - If you find that you’re answering disagree/strongly disagree for most statements, come see me during office hours to discuss
Guided Project 1

• Install Eclipse and plug-ins needed for CSC216
  – Do this before Lab 1 – PTFs will check installation in lab and help with troubleshooting

• Learn how to use Eclipse & GitHub

• Start WolfScheduler program
  – Guided Project 1
  – Review 116 materials & advanced OO
  – Due Friday, September 2, 2016 @ 9:00pm
  – Late Deadline Sunday, September 4, 2016 @ 9:00pm