Lesson 6.02 — Overriding Methods & Accessing Inherited Code

Overview

Objectives — *Students will be able to...*

- **Replace** superclass behavior by writing overriding methods in the subclass.
- **Write** subclass methods that access superclass methods.

Assessments — *Students will...*

- **Add** code to their Class Posters from the previous lesson.

Homework — *Students will...*

- **Read** the rest of BJP 9.2 starting from "The Object Class."

Materials & Prep

- **Projector and computer**
- **Whiteboard and markers**
- **Classroom copies** of WS 6.2
- **Class posters** from 6.1
- **Art supplies** for each group:
  - Markers
  - Poster paper (alternatively printer paper and tape)
  - Tape, magnets, or tacks to hang finished work

Pacing Guide

Section	Total Time
Bell-work and attendance	5min
Introduction & review of the project	10min
Student work	15min
Peer review	10min
Whole-group discussion/critique	10min

Procedure

Hook your students by prominently displaying art materials, and sample work (of your own making, or saved from a previous year). To feature the most engaging student examples, look for work that has a particularly complex or impressive flow-of-control pattern (see WS. 6.2 for explanation). If you have the time, this would be a great opportunity to display some complex code for students to examine on their own time. The more intricate the string pattern on the poster, the more intriguing the sample will appear.

Bell-work and Attendance [5 minutes]

Introduction & Review of the Project [10 minutes]

Emphasize with students...

Big Ideas - Products can be designed for life cycle Software organizations are always updating and altering software source code. This is done in order to fix bugs, add features or improve performance. It is for this reason that it's always important to remember that software can be designed for life cycle.

The original programmer can ensure that their code is well written and well documented. This can facilitate later updates and changes.

The original programmer can also think carefully about how additional features and updates might be implemented. This can impact the original design of objects or classes in order to ensure the ability to further add to and improve the design.

---

1. Inheritance makes it convenient to reuse code between classes. However, sometimes we'll want to specialize code in a subclass, or ignore a method that doesn't apply. Ask students for examples when this might be the case. If they're having trouble thinking of concrete examples, ask them to think of an example from their own class hierarchy that they created in the previous lesson. Some examples to get the class started:
   - Subclass Mammal might have a special case of Animal superclass method *feedYoung* (because they lactate).
   - Subclass *HotDrink* might use a different method chug from the *Drink* superclass (maybe the method involves sipping or burning your tongue).
2. Replacing superclass behavior by writing a new version of the methods in the subclass is called **overriding**. To override a method, write the method you want to replace in the subclass! No special syntax is required!
   - Building on our *Drink* example from the last lesson, we can write our own *chug* method for subclass *SugarFreeDrink*:

     ```
     public void chug(double gulp) {
         System.out.println("Yuck, this tastes terrible!");
     }
     ```

   - Compare this to the Drink superclass method chug, which is reproduced here for convenience:

     ```
     public void chug (double gulp) {    // Superclass Drink method
         if (ounces < gulp) {
             throw new IllegalArgumentException();
         } else {
             System.out.println("Glug, glug, glug!");
             ounces -=gulp;
             System.out.println("You have " + ounces + "oz. of drink left.");
         }
     }
     ```

3. Have students point out the differences between the two methods, predict the new output, and offer additional or alternative changes to the overridden *SugarFreeDrink chug* method.
   - Make sure that they (or you) point out that the number of ounces in the drink will NOT be updated in the overridden method as it stands.
4. It would be a lot of extra work to re-write the rest of the glug method if all you had wanted to do was add an extra println statement that we put in the overridden method. Fortunately, there is a way to access that method to put it back into our new, overridden method (highlighted below):

   ```
   public void chug (double gulp) {
       System.out.println("Yuck, this tastes terrible!");
       super.chug(gulp);
   }
   ```

   - This method now outputs "Yuck, this tastes terrible!", updates the number of ounces to reflect the amount you drank, throws an exception if you don't have any ounces, and outputs the number of ounces left in the drink.
   - Ask students why it's valid to call the overridden method *chug*, and reference the superclass method *chug* by the same name. (The superclass method is accessed using dot notation, which tells Java where to direct the flow of control.)

5. What if we want to access other information directly from the *Drink* class? Remember, our drink class had a fields for *name*, *hasCarbonation*, *gramsOfSugar*, and *ounces*, but they're all private because we were smart and remembered to encapsulate them.

   - If we wanted to write a method in our *SugarFreeDrink* subclass that accesses the data contained in *name*, we would have to add a *get* method first in the Superclass *Drink*:

     ```
     public double getName() {    // Written in superclass Drink.
         return name;
     }
     ```

   - This makes a copy of *name* that is public, and can be accessed outside of the *Drink* class. Now we can go ahead and write our subclass method using the accessor (highlighted below):

     ```
     public void advertising() {
         System.out.println (
             "Avoid the extra calories by drinking delicious " + getName() + "every day!!"
         );
     }
     ```

6. Have students offer additional examples using the *Drink* superclass, or using examples from their own class hierarchy.
7. Complete the introduction by asking students to explain what the difference is between overriding and overloading methods. (Overloading methods is when one class contains multiple methods with the same name, but a different number of parameters—sometimes called the *parameter signature*.)

Student Work [15 minutes]

1. Briefly review WS 6.2 with your students, reading the directions aloud if need be.
2. If you haven't already distributed project materials at this point, do so while your students are rearranging into partner pairs.
3. Encourage students to take 5 – 10 minutes on Step 1. They should review all steps of the project to ensure that their additional methods make sense.

   Announce that you'll offer extra credit to funny or creative code (if that fits in with your teaching style).
4. Offer time checks so students can stay on pace. Before you allow students to begin the peer review tour of others' work, remind them to take notes on their feedback so they will be able to contribute to the group critique/discussion at the end of class.

Peer Review [10 minutes]

Allow students 10 minutes to tour each other's work and offer feedback.

Whole-group Discussion/Critique [10 minutes]

If possible, rearrange student seats into a circle for the critique to encourage informal discussion.

- As a whole group, ask students to volunteer what they really liked about others' projects.
- Solicit questions and critiques, and re-teach if needed.
- Award classroom participation points to all students who contribute to the discussion.

Accommodation and Differentiation

Encourage advanced students to add additional classes, fields, methods, and client code. If students still have time to spare, encourage them to increase code complexity, add additional levels to the class hierarchy, or help their peers.

If you have a few students that are struggling with the assignment, allow them to work in groups of 4, each pair helping the other with their code. If students need additional guidance, have students complete the worksheet as a

series of think-pair-shares, where you return to whole group to share and discuss answers before moving on to the next step. **Teaching the class this way will roughly double the time required to complete the exercise.**

Misconceptions

When first learning polymorphism, students learn method override before method overload. However, in order to successfully override a method, the subclass method must have the same method signature as the superclass, otherwise the method will be overloaded. The code must match the method parameters and return type and the methods public, they cannot be private or static. When helping students debug their code where the overridden method is not behaving as anticipated, asking the student if the method signatures match can help find the error on their own.

Video

- BJP 9-2, *Polymorphism* http://media.pearsoncmg.com/aw/aw_reges_bjp_2/videoPlayer.php?id=c9-2
- CSE 142, *Polymorphism* (35:07–49:57) https://www.youtube.com/watch?v=WPdv8X291hE&start=2107

Forum discussion

Lesson 6.02 Overriding Methods & Accessing Inherited Code (TEALS Discourse account required)