Lesson 6.01 — Inheritance Basics

Overview

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

- **Correctly define** inheritance.
- **Use** proper syntax to extend a class.
- **Illustrate** is-a relationships.
- **Properly implement** constructors of derived classes using super.

Assessments — *Students will...*

- **Complete** a Class Hierarchy poster as indicated in WS 6.1

Homework — *Students will...*

- **Read** BJP 9.2 up to "DividendStock Behavior."
- **Collect** images that represent instances of the classes created for in-class poster project.

Materials & Prep

- **Projector and computer** (if you are able to/opt to use Eclipse with your students)
- **Whiteboard and markers**
- **Classroom copies** of WS 6.1 Start class poster, Example 6.1
- **Pictures** of Pokémon (http://tinyurl.com/l6mybmr) or Pokémon Cards
- **Student pair assignments**
- **Art supplies** for each group:
  - Poster paper or cut sheets of butcher paper
  - Lined paper (at least 3 sheets per group)
  - Markers
  - Glue Sticks
  - Old magazines, flyers, newspapers to cut up for collage
  - Scissors
  - Yarn, string, or embroidery floss
  - Tape, magnets, or tacks to hang finished work

Most of the supplies required for this lesson are readily available in high schools. If the school doesn't have poster paper, butcher paper works well. Other supplies may be available to borrow from the Math, Science, or Art teachers. To get an idea what a final student project should look like, check out the picture of sample student work "Example 6.1."

Pacing Guide: Day 1

Section	Total Time
Bell-work and attendance	5min
Introduction	20min
Review of the project	5min
Student work	25min

Pacing Guide: Day 2

Section	Total Time
Student work & teacher check	20min
Peer review	10min
Whole-group discussion and reteach if needed	15min

Section	Total Time
Quiz	5min

Procedure

Hook your students by prominently displaying pictures or cards of Pokémon, 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 many instances of each class, and uses classes/objects that are popular with your class. Invoke an air of mystery and don't offer an explanation for any of it.

Bell-work and Attendance [5 minutes]

Introduction [20 minutes]

Emphasize with students...

Big Ideas - Products can be designed for life cycle This activity will take you through the process of designing superclasses and subclasses. As you do so, consider carefully how your superclass could be reused and repurposed, later, by another programmer to create something different.

Many of the programs you create could be altered, remixed and tweaked to create new and innovative software that is slightly different than the original. This is a great thing about Object Oriented Programming. Much of the code can be reused and repurposed to create things we haven't even thought of yet.

---

1. Have a class discussion about the Pokémon picture/cards. Ask a few probing questions to model the use of proper programming terminology as you have students work through a starter assignment:
2. Ask students what the picture is of. Do they know the names or types of the Pokémon featured? Assign a quick Think-Pair-Share assignment having the students creating the instance of that Pokémon. Here's a sample prompt:

   "Create the object Pikachu. Pikachu should have a type, a level, and 2 methods."
3. Show a few more cards/pictures. Ask students to do another Think-Pair-Share with another prompt to create another instance (your prompt should have whatever Pokémon you've just discussed):

   "Create the object Squirtle. Squirtle should have a type, a level, and 2 methods."
4. Repeat this sequence one more time; showing the cards, and having students create a Pokémon instance object. Your students should be getting annoyed at having to write the same things over and over again.
5. Ask students what all of the Pokémon have in common. Encourage them to list additional traits other than the ones you've required them to include in your class exercises.
6. Ask students if they can think of a way to create new Pokémon objects without having to "reinvent the wheel" each time. (They might be able to sketch out an answer based on the previous night's readings.)
7. Students might suggest subclasses of Pokémon, in which case you should point out that they're creating a class that is included within the larger classification of "all Pokémon."

   The individual Pokémon demonstrate an *instance of* the Pokémon class. Each *subclass* of Pokémon is a specialized version of the parent class (or *superclass*) Pokémon.

   Examples

   - An electric Pokémon is a Pokémon.
   - A computer science student is a student.
   - A math teacher is a teacher.

- Soda is a drink.

Have students describe the hierarchical structure of each relationship above. Electric Pokémon is a subclass, Pokémon is the superclass. Student is the superclass, computer science student is the subclass. Drink is the superclass, soda is the subclass.

8. Confirm understanding by asking for students to generate some examples. In each case, their two categories exhibit a hierarchical connection; one type is a specialized version of the other.
9. Ask students to define an inheritance hierarchy in their own words. Briefly discuss why you would want to use inheritance in programming.
   - An **inheritance hierarchy** is a set of hierarchical relationships between classes of objects.
   - **Inheritance** is a programming technique that allows a derived class to extend the functionality of a base class, inheriting all of its state and behavior.)
   - **Superclass** is the parent class in an inheritance relationship.
   - **Subclass, or child class** is the derived class in an inheritance relationship.
10. Check for understanding by returning to the examples above, and asking students to give an example of some characteristics (fields) the parent class would have, and what characteristics students would add to the specialized subclasses.

Example: Drinks could have a String *name* and boolean *carbonated*, and Soda could add a boolean *caffeinated*.

11. The class header for a subclass that extends the functionality of the parent class looks like this:

```
public class Mammal extends Animal {

public class Motorcycle extends Vehicle {

public class Churro extends Pastry {
```

- Point out that the subclass names are capitalized by convention, and that you always use the *extends* keyword. Give students a moment to think of a few hierarchical relationships in a think-pair-share, and ask several volunteers to come to the front of the room to demonstrate the correct class header.

12. For the following example, we will create subclasses that extend the *Drink* superclass written below. If your students need additional practice building classes of objects, you can have them help you write this code. In more advanced classes, you may just reveal this class as a fully-formed starting point to demonstrate how to write subclasses.

```
public class Drink {
    private String  name;
    private boolean hasCarbonation;
    private double  gramsOfSugar;
    private double  ounces;

    public Drink (String n, Boolean h, double g) {
        name = n;
        hasCarbonation = h;
        gramsOfSugar = g;
        ounces = 8;           //FDA defines a serving as 8 oz.
    }

    public void chug (double gulp) {
        if (ounces < gulp) {
            throw new IllegalArgumentException ("Not enough " + name + " left.");
        } else {
            System.out.println ("Glug, glug, glug!");
            ounces -=gulp;
```

```
            System.out.println("You have " + ounces + "oz. of " + name + " left.");
        }
    }

    public String getState() {
        return "liquid";
    }

    public void printLabel() {
        System.out.println ("Enjoy refreshing " + name + " !");
    }
}
```

13. Because the subclass is still a class, you should add fields and constructors, as you do with any class:

```
public class SugarFreeDrink extends Drink {
    private boolean hasSweetener;
    private double  caffeineContent;
```

The additional fields `hasSweetener` and `caffeineContent` characterize all `SugarFreeDrink` `Drink` objects. You point out to students that `SugarFreeDrink` "is a kind of `Drink`." Spot-check student understanding by asking if objects of the `Drink` superclass will initialize with a value for `hasSweetener` or `caffeineContent`. (*No.*)

14. `SugarFreeDrink` drinks still have a name, a boolean carbonation value, sugar content, and ounces, but we've added a new fields specifying whether or not the sugar free drinks have caffeine and artificial sweeteners. The constructor then looks like this:

```
public SugarFreeDrink(String name, boolean hasCarbonation, boolean h, double c) {
```

- The fields in the subclass' constructor now contain <type> <superclass parameter values> (highlighted), except for the new fields, which still have a formal parameter (in this case h and c).
- To complete the constructor so it can access the fields you already wrote in the superclass, you use the keyword super: `Java // must be first line after constructor header super(name, hasCarbonation, 0.0);`
- Notice that we've initialized all objects in the *SugarFreeDrink* class to have 0.0 grams of sugar.
- Complete the constructor with your subclass' new fields: `Java hasSweetener = h; caffieneConten = c; }`

15. You can also add methods that only apply to your subclass, just the way you normally write object methods:

```
public void printWarningLabel() {
    if (hasSweetener) {
        System.out.println("This drink is not safe for Phenylketonurics.");
    } else {
        System.out.println("This drink contains no artificial sweeteners.");
    }
}
```

Review of the Project [5 minutes]

1. Briefly review the assignment 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.

Student Work [25 minutes]

1. Encourage students to take 5–10 minutes on Step 1. They should review all steps of the project to ensure that their selection of classes lends itself to the project (e.g. they shouldn't pick something they don't know a lot about because they'll have trouble coming up with fields and methods).

2. Offer time checks every 10 minutes so students can stay on pace. By the end of the first day, they should have gotten to step 6 or 7. Visit each group to make sure that they haven't veered off course.
3. On **day two**, check student work and help students display their work around the room.
4. Check that the flow-of-control string (see WS 6.1 for explanation) correctly shows how a method is passed through subclasses to the superclass.
5. Remind students to take notes (Step 11 on WS 6.1) to help them remember talking points for later in the class.
6. As a whole group, ask students to volunteer what they really liked about others' projects. Solicit questions and critiques, re-teaching if needed.
7. Administer quiz 6.1 to assess student understanding.

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 read on method overriding, and invite them to add that code as well. Students can attach this "extra code" using paper and tape/glue or sticky notes.

If you have a few students that are struggling with the class, give them your starter Drink class and SugarFree subclass, and let them build off of your examples. You can print out your starter code and cut it into pieces and shuffle them so students have to place each line in the correct location (as with a Parson problem).

If your students need further instruction on calling a superclass' constructor, go through a few more examples using the Drink superclass, or classes that you created as a whole group. One example has been outlined below:

```
public class SugarDrink extends Drink {
    private boolean isJuice;

    public SugarDrink (String name, boolean hasCarbonation, double gramsOfSugar,
                       double ounces, boolean iJ) {
        super(name, hasCarbonation, gramsOfSugar, ounces);
        isJuice = iJ;
    }
```

Teacher Prior CS Knowledge

- The Object Oriented Programming (OOP) paradigm could be thought of as mimicking the real world where objects consists of data that define them and actions that can be performed on the data. As you will see, the process of learning OOP is infinitely more complex.
- The pillars of Object Oriented Programming (OOP): inheritance, encapsulation, and polymorphism. In a nutshell inheritance allows for code reuse by defining methods once in a superclass, encapsulation provides data security by hiding data implementation from the user and only allowing methods in the class to modify the data, and polymorphism offers flexibility to the designer by way of methods defined in many forms.

Misconceptions

Students' use of "inheritance" prior to computer science are in the context of inheritance from an ancestor and genetic inheritance of traits. However, in computer science, inheritance is used for classification where the class that inherits (extends) from a more general class (super class). Neither of the students' prior knowledge and use of inheritance is an accurate representation of Java's class structure.

Common Mistakes

Object oriented concepts common mistakes: http://interactivepython.org/runestone/static/JavaReview/OOBasics/ooMistakes.html

Video

- BJP 9-1, *Inheritance: Interacting with the Superclass* http://media.pearsoncmg.com/aw/aw_reges_bjp_2/videoPlayer.php?id=c9-1
- CSE 142, *Inheritance* (23:28–35:06) https://www.youtube.com/watch?v=WPdv8X291hE&start=1408
- CS Homework Bytes, *Inheritance, with Zach* https://www.youtube.com/watch?v=Alv2ApK_jdo

Forum discussion

Lesson 6.01 Inheritance Basics (TEALS Discourse account required)