Object-Oriented Programming (OOP) Advanced Topics

CSCI 161 – Introduction to Programming I

Professor Thomas Rogers

Overview

- Chapter 8 in the textbook *“Building Java Programs”*, by Reges & Stepp.
- Object Constructors – Advanced
- Using Objects in a Method
- Object Encapsulation
- Class Invariant
- Inheritance

Object Constructors - Advanced

- **Constructor** – A method having the same name as the Object class.
- **Default Constructor** – Always include the default constructor (no params) even if just to give state variables their initial values.
- **Multiple Constructors** – You can include multiple constructors, each with different parameter sets, each with a different purpose.

Object Constructors – Advanced (continued)

- **Multiple Constructors** – A class with multiple constructors:

```
public class Point {

  /** Fields (aka State variables) **/
  int x;
  int y;

  /** Constructors **/
  // constructs a point with a location of 0, 0
  public Point() {
    x = 0;
    y = 0;
  }

  // constructs a point at a given location
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Using Objects in a Method

- **Using Objects in a Method -** For example, the main method:

```
public static void main(String[] args) {

   // Declare two points
   Point p1 = new Point();       // Uses default
   constructor
   Point p2 = new Point(3, 2);   // Uses alt constructor

   System.out.printf(“p1.x=%d p1.y=%d\n”, p1.x, p1.y);
   System.out.printf(“p2.x=%d p2.y=%d\n”, p2.x, p2.y);

   p2 = p1;  // p2 object now set to p1.
   p1.x = 5;
   System.out.printf(“p2.x=%d p2.y=%d\n”, p2.x, p2.y);

}
```

Question: What will be printed?

Object Encapsulation, etc.

- **Encapsulation** - Hiding the implementation details of an object from the clients (callers) of the object:
  - Make the state fields of the class private
  - Provide accessor and mutator methods for accessing and changing state fields
- **Private Fields** – Aka ***State*** (field) variables that are marked as private are not directly accessible by client code (code within a program that declares the class as an object and then uses the object).
- **Abstraction** - Focusing on essential properties, methods rather than inner details.

Encapsulation example

- Encapsulating the x and y fields of the ***Point*** class:

```
public class Point {

  /** Fields (aka State variables) **/
  private int x;
  private int y;

  // Will now require accessor methods so client
  // may get values of x and y
  public int getX() {
       return x;
  }

  public int getY() {
       return y;
  }

  // Also requires a mutator method for setting x and y
  public void setLocation(int x, int y) {
       this.x = x;
       this.y = y;
  }
}
```

Class Invariant

- **Class Invariant** - An ***assertion*** (or fact) about an object's state that is true for the lifetime of the object:
  - Java has no formal mechanism in the language for maintaining assertions.
  - It is up to the programmer and the logic of the class to enforce assertions.
- Examples from a ***"Time"*** class that has state fields that must have restricted value ranges:
  - **hours** - must be between 0 and 23, inclusive
  - **Minutes -** must be between 0 and 59, inclusive
  - **dayOfTheWeek** - must be from the list of Strings in the set: {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"}

Class Invariant (continued)

- Another example is the Point class in which you only want to deal with the upper-right (positive x, positive y) quadrant:

```
// Requires a mutator method for setting x and y
// that enforces the invariant
public void setLocation(int x, int y) {
    if (x < 0 || y < 0) {
        throw new IllegalArgumentException();
    }
    this.x = x;
    this.y = y;
}
```

Inheritance

- **Inheritance** - A mechanism in which one object acquires all the properties and behaviors of a parent object. Inheritance allows there to be a base class, aka the superclass, that is extended to make a derived class, aka the subclass.
- **Superclass** - The parent class in an inheritance relationship.
- **Subclass** - The child, or derived, class in an inheritance relationship.
- **Syntax Notation** for a class that inherits from a superclass:

```
public class <name> extends <superclass> {
    ...
}
```

Inheritance (continued)

- Consider a program that kept track of animals in a zoo. That program could use a class for each type of animal, including **Lion**, **Tiger**, **Snake**, and **Turtle** classes.
  - Further suppose that each of these classes had accessor methods to determine if the animal of that class is *warm-blooded*, whether or not it *lays eggs*, and whether or not it *has a tail*.
  - However, many types of animals have things in common. For example, all reptiles are cold-blooded and lay eggs, and all mammals are *warm-blooded* and do not lay eggs (is the platypus still a thing?).
  - It is with this "sameness", these "commonalities", where inheritance can help by introducing **Reptile** and **Mammal** superclasses.
  - The **Turtle** and **Snake** classes would be based on a superclass called **Reptile**.
  - Lastly, the **Mammal** and **Reptile** classes would also be derived from a superclass called **Animal.**

Inheritance (continued)

- **@Override** - Use of the @Override directive before any methods in the subclass class that will override those from the superclass. This directive must be included for the override (aka "specialization") to occur.

- **Sample:** Take a look in...

    `/home/grader/rogers161/Public/Zoo`

    ... and in...

    /home/grader/rogers161/Public/Zoo2

    ...for examples that show no inheritance and inheritance through the Reptile, Mammal and Animal classes, respectively.