Java Chapter 6 Part 2

* OOP: Object Arguments, Overloading, Shadowing, Packages
* CIS 255 • Shelby-Hoover Campus

Objects as Arguments

* With the static methods in chapter 5, it was possible to send a primitive value or an object as an argument to a method
  + A primitive parameter receives a copy of its argument
  + An object parameter receives the address of the object argument; thus, it references the original object
* A static method with an object parameter can contain calls to several of the object’s methods
* Example: the static method rollDie in DieArgument.java (Code Listing 6-16)

Overloading

* Remember that two method definitions may have the same name as long as they have different parameter lists (this applies to object-oriented methods in addition to static methods)
* Constructors are usually overloaded to allow several different ways to set up an object
  + Parameters are assigned to the fields
  + A no-arg constructor uses default values
* Example:
  + In the class BankAccount (Code Listing 6-20), there are three constructors, and the methods deposit, withdraw, and setBalance are overloaded to accept either a double or a String
  + In the program AccountTest.java (Code Listing 6-21), the program receives the input via dialog boxes as String data

Instance Fields

* When an instance of a class is declared, a set of fields for that instance is created
  + The field variables associated with a specific instance of a class are known as **instance fields**
  + These instance fields can be accessed by any method of the class
* Be careful not to declare a local variable in a method with the same name as an instance field (a process known as **shadowing**)
  + The local variable’s scope is within the method only
  + Values assigned to the local variable will not be assigned to the instance field

Shadowing Example

* In the Rectangle class, if the method setLength contains a local variable named length (the same name as the private field), the value of the parameter len will be assigned to the local variable rather than to the field, and the local variable goes out of scope after the method terminates:  
    
  public class Rectangle  
  {  
   private double length;  
   private double width;  
   …  
   public void setLength(double len)  
   {  
   int length; // shadowing  
   length = (int) len;  
   }  
   …  
  }

import Statements

* The classes in the Java API are organized into **packages**, or groups of related classes
* Sometimes a program only needs a specific Java class from a package, so it includes an **explicit import statement** for that class
  + import java.util.Scanner;
  + import javax.swing.JOptionPane;
  + import java.text.DecimalFormat;
* A program may also import every class within a package using a **wildcard import statement**
  + import java.io.\*;
  + import java.awt.\*;

Standard Java Packages

|  |  |
| --- | --- |
| **Package** | **Description** |
| java.applet | Classes necessary to create an applet |
| java.awt | Classes for the Abstract Windowing Toolkit used in drawing and GUIs |
| java.io | Classes for various types of input and output |
| java.lang | General classes (automatically imported) |
| java.net | Classes for network communications |
| java.security | Classes that implement security features |
| java.sql | Classes for accessing databases |
| java.text | Various classes for formatting text |
| java.util | Various utility classes |
| javax.swing | Classes for creating GUIs |

OOP Design Process

* A program may need to utilize several programmer-defined classes
* Prior to writing code in Java (or any other language that supports OOP), a programmer should identify what classes are needed and what responsibilities each class should have
* Example: Service Quotes (section 6.9)
  + Write a description of the problem domain
  + Identify all of the nouns
  + Refine the list of nouns to eliminate redundancy, extraneous information, specific objects, and simple values
  + Once you have identified the classes, create UML diagrams listing the fields and operations for each class

Reminders for Chapter 6

* In an OOP class definition, the class name also serves as the data type used to declare the objects
* Fields are declared at the class level and are usually private
* Instance method headers do **not** include the keyword static
* Accessor methods (get) have no parameter variables and return the value of a field or a calculated value based on some field(s)
* Mutator methods (set) have at least one parameter that matches the type of a field and do not return a value; they assign the parameter(s) to the corresponding field(s)
* Constructor methods (with the same name as the class) have no return type and are used to set up an object’s fields with initial values
* Java does not provide a default constructor for a class that contains its own constructor definition, so a programmer may need to define a no-arg constructor to replace it
* Both constructors and mutators should include validation to ensure that an invalid value is not assigned to a field

More Reminders for Chapter 6

* When declaring an object, use the syntax  
    
  ClassName objectName = new ClassName(args);  
  + The parentheses are required even if there are no arguments
  + If a constructor requires arguments, list them in the parentheses after the class name
* When invoking a method on an object, use the dot operator (.) between the object name and the method name
* Only public members may be accessed using dot notation
* An object parameter in a method references the object argument in the call
* Overloading requires a change in the parameter lists between two or more methods with the same name
* A method that contains a local variable that shadows an instance field loses access to the instance field within that method
* An OOP class definition cannot be executed by itself; a programmer must instantiate the class in a program (usually in another file)