Java Chapter 6 Part 1

* OOP: Defining Classes, Creating Objects, Constructors
* CIS 255 • Shelby-Hoover Campus

Procedural Programming

* The programs in the first five chapters of the textbook are structured as procedural programs
  + Programs are composed of procedures that perform tasks
  + The data is independent of the procedures; it is passed from one procedure to another
* If all of a program’s methods are static, the program is procedural
* There are potential problems with procedural programming
  + If the data structure changes, there will be many changes to your code to accommodate the new structure
  + This leads to an increased opportunity for bugs

Object-Oriented Programming

* In object-oriented programming (OOP), the data and procedures are collected together in **objects**
  + The data variables in each object are known as **fields** or **attributes**
  + The procedures used to access or modify the fields are known as **methods** (non-static)
* **Encapsulation** describes the combination of data and operations into a single entity
* **Data hiding** prevents direct access to the fields; outside code must interact with the object through methods
* Object types are often reusable across programs

Classes

* A **class** is the template / blueprint / pattern / mold / “cookie cutter” from which objects are created
* Each object created from that class definition is known as an **instance** of that class; the process of creating an object is known as **instantiation**
* Every variable of a class type will have the same fields (with different values) and the same operations available to them
* Variables of a class type are called **reference variables**; they hold the memory addresses of the objects rather than the objects themselves
  + Primitive variables occupy fixed amounts of memory
  + Class variables may occupy differing amounts of memory
  + One metaphor to describe this relationship is that of a kite and string; the string (the variable that references the object) allows someone to control the kite (the object itself)

Class Definitions

* Fields and methods are defined within the class definition, not inside another method
* Each field requires an access specifier, a type, and a name
  + Access can be public (available to outside code) or private (limited to class methods)
  + Most often these are declared as private to protect the data
* A method designed to work with an instance (object) of a class is called an **instance method**
  + The header of each instance method requires an access specifier, a type, a name, and a parameter list (which can be empty)
  + Most methods are public
  + Leave off the static keyword for OOP methods

Common Methods

* When fields are private, a class commonly provides two types of methods to allow the user to obtain and change their values
* An **accessor** method returns some value from within the object
  + Names usually begin with *get*
  + Parameters: typically none
  + Returns a value that reveals some information about an object (the value of a single field, or a calculated value)
* A **mutator** method allows outside code to modify the value of one or more fields
  + Names usually begin with *set*
  + Parameters: matching the types of values to be changed, with names that differ from the field names
  + Return type is usually void
  + May contain syntax to validate the value before assignment
* Not all fields will require accessors and / or mutators

Class Definition Example

* The textbook demonstrates several phases of developing a class named Rectangle (here examined in Code Listing 6-7)
  + The class name will also be used as the data type for variables to reference objects
  + The fields, length and width, are made private to protect them from harmful changes
  + The mutator methods, setLength and setWidth, allow a programmer to assign new values to the fields (notice that the parameter names are not the same as the field names)
  + Two of the accessor methods, getLength and getWidth, allow a programmer to obtain the values of the fields
  + Instead of storing the area in a field (which would have to be updated every time one of the other fields changes), the class provides a third accessor, getArea, to calculate the value on the fly; this prevents the storage of **stale data** (an out-of-date value of a field dependent on other data)

Mutators and Validation

* A common technique to protect fields from invalid values is to use a decision structure to validate the value of a mutator method’s parameter before it is assigned to a field:  
    
  public void setLength(double len)  
  {  
   if (len > 0.0)  
   length = len;  
   else  
   System.out.println("Error! Length must be positive.");  
  }

UML Class Diagrams

* The Unified Modeling Language (UML) provides a diagram to represent the various aspects of a class definition
* The top segment of the diagram contains the class name
* The middle segment contains the fields, with each field followed by a colon and the data type
* The lower segment contains the methods, with any parameters in parentheses after each method name, and a colon and the return type after the parameter list
* public members are preceded by a + sign, and private members are preceded by a - sign

UML Class Diagram Example

|  |
| --- |
| Rectangle |
| - length : double  - width : double |
| + setLength(len : double) : void  + setWidth(w : double) : void  + getLength() : double  + getWidth() : double  + getArea() : double |

Instantiating a Class

* When you create an object of a class type, you must use the keyword new:  
    
  TypeName objectName = new TypeName();
* Objects are usually created in a separate program (not within the same class definition)
* Methods are invoked on an object using the dot notation: objectName.methodName(args)
* Results of invoking a method will depend on the values in the fields of the calling object
* Example: RectangleDemo.java (Code Listing 6-8)
* Just as a program can contain multiple variables of a primitive type, it can contain multiple objects of a class type, and each object contains a separate set of fields (RoomAreas.java, Code Listing 6-9)

Constructors

* When a program creates an object, it can provide initial values for each field without using mutators
* A **constructor** method indicates what actions should be taken when a class is instantiated
* The header for a constructor method should follow the following form:  
    
  public ClassName(parameterList)  
  + There is no return type
  + The class name is the method name
  + Use parameters for the values to be assigned to the fields
* The body should consist of assignment statements for each field (even if there isn’t a corresponding parameter for a field)

Constructor Example

* The constructor in the next phase of the Rectangle class (Code Listing 6-10) assigns the values of two parameters to their respective fields
* As with mutator methods, it may be wise to validate the values of the parameters; unlike a mutator, a constructor should always assign some value to a field, even if the parameter value is invalid:  
    
  public Rectangle(double len, double w)  
  {  
   if (len > 0.0)  
   length = len;  
   else  
   {  
   System.out.println("Error! Length must be positive.");  
   System.out.println("Length will be set to one.");  
   length = 1.0;  
   }  
    
   if (w > 0.0)  
   width = w;  
   else  
   {  
   System.out.println("Error! Width must be positive.");  
   System.out.println("Width will be set to one.");  
   width = 1.0;  
   }  
  }

Using a Constructor

* If a constructor requires arguments, list them in the parentheses after the class name:  
    
  ClassName objName = new ClassName(argList);
* Example: Setting up a Rectangle object with an initial length of 5.0 and an initial width of 15.0 (ConstructorDemo.java, Code Listing 6-11)  
    
  Rectangle r = new Rectangle(5.0, 15.0);
* Having a constructor does not eliminate the need for mutator methods, as a constructor can only be used when the object is created, not when you want to change the fields of an object later in the program

The Default Constructor

* If a class does not contain a constructor definition, Java automatically provides a **default constructor** with no parameters
* If a class contains a constructor definition with parameters, Java won’t provide the default constructor, and a program cannot create an object without arguments unless the class contains additional code
* A programmer can overload a constructor, providing multiple constructor definitions with different parameter lists
* A class may contain a **no-arg constructor** to indicate what should happen if the call to the constructor has no arguments (effectively replacing the default constructor)
* The header of the no-arg constructor contains no parameters
* The body assigns “default” values to the fields

No-arg Constructor Example

* The “default” dimensions for a Rectangle object:  
    
  public Rectangle()  
  {  
   length = 1.0;  
   width = 1.0;  
  }
* Both constructors can be used in the same program:  
    
  Rectangle box1 = new Rectangle();  
  Rectangle box2 = new Rectangle(3.0, 4.0);
* Additional constructors can be added as necessary

String Constructor

* String is a class type, so there is a constructor that can be used to set up a String object’s initial value:  
    
  String name = new String("Henry");
* Because the String class is “special”, constructor syntax is not required:  
    
  String name = "Henry";

Class Design

* Not every class follows the concept of having an accessor and a mutator for every field
* The underlying functionality in a class may be more involved than simply assigning or returning values
* Example: The class Die (Code Listing 6-14) has no “mutators” for the fields sides and value; the user has the opportunity to set the number of sides for the die using a constructor, and the value of the die changes via random number generation in the method roll()
* The program DiceDemo (Code Listing 6-15) instantiates the class Die twice