Chapter 4 - ADT

Classes, Objects, and Methods

- Information Hiding and Encapsulation
- Objects and Reference
- Aspects of Object-Oriented Programming (OOP)
  - » Encapsulation
  - » Polymorphism
  - » Inheritance

Summary of Class Definition Syntax

```
/*****************************************
 * Class description
 *****************************************/
public class Class_Name
{
   //-----------------
    declarations of "instance variables"
       (also called "member variables" or "global class variables")

   //-----------------
    definitions of class methods

} // end of class Class_Name
```

- instance variables and methods of a class can be defined with the following *scope modifiers,*
  - » **public** (member available inside and outside the class) or
  - » **private** (member available only within the class)
  - » **protected** (*this is a special modifier used with inheritance)*

"Information Hiding"

Information Hiding:

"hiding data and operations of an ADT"

– *giving others only use of the class/object*

- » protecting data inside an object
- » preventing direct access from outside object
- » use **private** modifier for instance variables
- » use **public** modifier for methods that access data,
  - – methods that return private variable values
    - accessor methods ("accessors" or "get methods")
  - – methods that set private variables values
    - mutator methods ("mutators" or "put methods")

OOP: Encapsulation

- **Encapsulation**
  - » binding (or grouping) together related data and the methods that act on that data
  - » binding is "hidden" and is represented by an "abstraction" of the relationships of the data and methods,
  - » "encapsulation" is as much as *concept,* as it is a *technique*
  - » example,
    - – the encapsulation of a "student" contains everything about a single student, but nothing about "faculty wage info" (why should it?)
  - » (most important aspect of OOP)

OOP: Polymorphism

- **Polymorphism**
  - » an expression of a definition that can 'adjust' within a variety of conditions or states
    - – "poly"-many, "morph"-face or shape
  - » ability of a class definition (or method group) to change its representation based on *different conditions* or *data values*
  - » example,
    - – a **print()** method that can display any type
    - – or a *class*.**print()** method that displays to either CLI or GUI depending on which the *class instance* (object) has been pre-defined for

OOP: Inheritance

- **Inheritance**
  - » the creation of a new definition based on the structure of a previous definition, but most importantly, *without requiring complete knowledge of the previous definition*
  - » technique of a class being *created from* another class, with the child automatically "inheriting" all attributes & methods of the parent class
  - » example,
    - – a new JApplet extends JApplet *(the new applet class inherits all data and methods of the standard JApplet)*
    - – creating a new **StringUpp** class based on the standard String class, except it maintains all data in uppercase
  - » terms:
    - – child="sub-class", "parent"="super-class"

Formalised Abstraction: Abstract Data Types (ADTs) – OOP Encapsulation

- technique used in storing "real-world" descriptions
- an ADT implies a *class* implementation in Java
  - » a container for *data items,* and *methods* to act on that data
- encourages *information hiding* and *encapsulation*
- the class's *user interface* allows programmers to use the ADT (class),
  - » descriptions, parameters, and names of its methods
- Implementation:
  - » private instance variables
  - » method definitions (implementations) are hidden
  - » another programmer cannot see or change the *class*
  - » only sees the "interface" is seen or ever required

Variables Review: Class (Reference) Type vs. Primitive Type

- a primitive type variable directly holds the value of the variable (identifier name is directly-associated with the memory address storing the data value)
- A reference variable type holds a *memory address*, that indirectly points to the an *object in memory*,
  - » objects in memory have multiple member variables, and member objects, of different types, along with "pointers" to the class's methods
    - – note: all objects of the same class share the actual same methods, but use different data from the object

Definition of an Entity (Creating an ADT—Class Definition)

- Definition of a "Shape" class (see **Shape.java**)
  - » this class holds the statistics of a general shape, such as:
    - – **shapename** – square, rectangle, triangleright , elllipse, and circle
    - – **shapecode** – s, r, t, e, c
    - – **width, height** – dimensions for s, r, t, and e
    - – **radius** – dimension for c
    - – **area** - area of shape, based on code and dimensions
  - » the "shape" class encapsulates all the attributes that are needed to describe a simple geometric shape

Definition of an Entity (Creating an ADT—Class Definition)

- » this class also holds the public methods (available for any program using the Shape class):
  - – **Shape()** – constructor, to create object at declaration; either create an "empty" shape or fully defined shape
  - – **setShape()** – sets the attributes of an "empty" shape
  - – **changeDim()** – change dim. of s, r, t, e, c
  - – **getShapeCode()** – returns shape code
  - – **getShapeName()** – returns the full shape name
  - – **getArea()** – returns area of shape
  - – **print()** – outputs a formatted display for the shape
  - – **printString()** – returns a String, same as **print()**
  - – **equals()** – return bool, compare against another shape

Using the Shape Class

- Creating a new shape object:

```
Shape shape1 = new Shape();            // empty shape
Shape table = new Shape(r, 10.0, 5.5); // rect. 10.0 x 5.5
Shape pill = new Shape(e, 10.0, 5.5);  // ellipse 10.0 x 5.5
Shape box = new Shape(s,10.1);         // square 10.0 x 10.0
Shape circx = new Shape(c, 3.2);       // circle radius 3.2
```

- Using various methods:

```
shape1.setShape(c, 5.0);     // set shape1 as circle
table.changeDim(5.5, 11.0);  // to 5.5 x 11.0
str = box.getShapeName();    // return shape name
ar = pill.getArea();         // get area

if !(circx.equals(box))      // if shapes not same
   System.out.println ("Circle not equal to box.");

if (box.equals(box))         // if the shape is itself
   System.out.println ("Box is the same as itself!");
```


Examining the Shape Class for OOP

- **Encapsulation**
  - » by being a *class definition*, the Shape class is exhibiting encapsulation: all related attributes and methods related to "shapeness" are gathered together
  - » this class has *accessor* and *mutator* methods
    - – accessors: **getShapeCode(), getShapeName(), getArea()**
    - – mutators: **setShape(), changeDim()**
    - – *why are* **print()** *and* **printString()** *not acessors?*

Examining the Shape Class for OOP

- **Polymorphism**
  - » the Shape class represents shapes of different dims.,
    - – 1 dimension: squares and circles
    - – 2 dimensions: rectangles, triangles (right), ellipses
  - » to allow for this, some methods are overloaded
    - – **Shape()** – constructors
    - – **setShape(),** and **changeDim()** – *depends on shape!*
- **Inheritance**
  - » the Shape class does not explicitly show *inheritance*