Encapsulation

September 13, 2006

Administrative Stuff

• HW3 is due on Friday
• No new HW will be out this week
• Next Tuesday we will have Midterm 1:
  • Sep 19 @ 6:30 – 7:45pm.
  • Location: Hoover Hall Auditorium (room 2055)
• On Monday we will have a review session
• No class on Friday (Sep 29, 2006)

HW3

• Printout Due This Friday ‘BEFORE’ class
  • The source code for the three programming projects is also due by Friday on WebCT.

WebCT Submission Demo

• I need a volunteer

Writing Classes

• The programs we’ve written in previous examples have used classes defined in the Java standard class library
• Now we will begin to design programs that rely on classes that we write ourselves
• The class that contains the main method is just the starting point of a program
• True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality

Quick review of last lecture
Classes and Objects
• Recall from our overview of objects in Chapter 1 that an object has state and behavior
• Consider a six-sided die (singular of dice)
  • Its state can be defined as which face is showing
  • Its primary behavior is that it can be rolled
• We can represent a die in software by designing a class called Die that models this state and behavior
  • The class serves as the blueprint for a die object
• We can then instantiate as many die objects as we need for any particular program

Classes
• A class can contain data declarations and method declarations

```
int size, weight;
char category;
```

Data declarations

Method declarations

Classes
• The values of the data define the state of an object created from the class
• The functionality of the methods define the behaviors of the object
• For our Die class, we might declare an integer that represents the current value showing on the face
• One of the methods would “roll” the die by setting that value to a random number between one and six

Classes
• We’ll want to design the Die class with other data and methods to make it a versatile and reusable resource
• Any given program will not necessarily use all aspects of a given class
• See RollingDice.java (page 157)
• See Die.java (page 158)

The Die Class
• The Die class contains two data values
  • a constant MAX that represents the maximum face value
  • an integer faceValue that represents the current face value
• The roll method uses the random method of the Math class to determine a new face value
• There are also methods to explicitly set and retrieve the current face value at any time

The toString Method
• All classes that represent objects should define a toString method
• The toString method returns a character string that represents the object in some way
• It is called automatically when an object is concatenated to a string or when it is passed to the println method

```
System.out.println("Die One: "+ die1 + ", Die Two: " + die2);
```
**Constructors**

- As mentioned previously, a constructor is a special method that is used to set up an object when it is initially created.
- A constructor has the same name as the class.
- The `Die` constructor is used to set the initial face value of each new die object to one.
- We examine constructors in more detail later in this chapter.

**Data Scope**

- The scope of data is the area in a program in which that data can be referenced (used).
- Data declared at the class level can be referenced by all methods in that class.
- Data declared within a method can be used only in that method.
- Data declared within a method is called local data.
- In the `Die` class, the variable `result` is declared inside the `toString` method -- it is local to that method and cannot be referenced anywhere else.

**Instance Data**

- The `faceValue` variable in the `Die` class is called instance data because each instance (object) that is created has its own version of it.
- A class declares the type of the data, but it does not reserve any memory space for it.
- Every time a `Die` object is created, a new `faceValue` variable is created as well.
- The objects of a class share the method definitions, but each object has its own data space.
- That’s the only way two objects can have different states.

**Encapsulation**

- We can take one of two views of an object:
  - internal - the details of the variables and methods of the class that defines it.
  - external - the services that an object provides and how the object interacts with the rest of the system.
- From the external view, an object is an encapsulated entity, providing a set of specific services.
- These services define the interface to the object.

**Diagram**

Each object maintains its own `faceValue` variable, and thus its own state.

- `die1` with `faceValue` 5.
- `die2` with `faceValue` 2.
Encapsulation

- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished
- Any changes to the object’s state (its variables) should be made by that object’s methods
- We should make it difficult, if not impossible, for a client to access an object’s variables directly
- That is, an object should be *self-governing*

Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data

Method Control Flow

- If the called method is in the same class, only the method name is needed

```
myMethod();
```

- The called method is often part of another class or object

```
obj.doIt();
```

Why we don’t have to use ‘new’ with the NumberFormat class?

- The ‘new’ is performed for you inside that class

```
main

NumberFormat format = NumberFormat.getInstance();
return format;
```

UML Diagrams

- UML stands for the *Unified Modeling Language*
- *UML diagrams* show relationships among classes and objects
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)
- Lines between classes represent *associations*
- A dotted arrow shows that one class *uses* the other (calls its methods)
UML Class Diagrams

- A UML class diagram for the `RollingDice` program:

```
<table>
<thead>
<tr>
<th>RollingDice</th>
<th>Die</th>
</tr>
</thead>
<tbody>
<tr>
<td>main( args : String[] ) : void</td>
<td></td>
</tr>
</tbody>
</table>

Die

- faceValue : int
- roll() : int
- setFaceValue( int value ) : void
- getFaceValue() : int
- toString() : String
```

Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of visibility modifiers
- A modifier is a Java reserved word that specifies particular characteristics of a method or data
- We’ve used the `final` modifier to define constants
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss later

Visibility Modifiers

- Public variables violate encapsulation because they allow the client to "reach in" and modify the values directly
- Therefore instance variables should not be declared with public visibility
- It is acceptable to give a constant public visibility, which allows it to be used outside of the class
- Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed
Visibility Modifiers

<table>
<thead>
<tr>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violate encapsulation</td>
<td>Enforce encapsulation</td>
</tr>
</tbody>
</table>

Variables

Methods

Provide services to clients

Support other methods in the class

Accessors and Mutators

• Because instance data is private, a class usually provides services to access and modify data values
• An accessor method returns the current value of a variable
• A mutator method changes the value of a variable
• The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `x` is the name of the value
• They are sometimes called “getters” and “setters”

Mutator Restrictions

• The use of mutators gives the class designer the ability to restrict a client’s options to modify an object’s state
• A mutator is often designed so that the values of variables can be set only within particular limits
• For example, the `setFaceValue` mutator of the `Die` class should have restricted the value to the valid range (1 to `MAX`)
• We’ll see in Chapter 5 how such restrictions can be implemented

Run examples from the book

THE END