Encapsulation

September 12, 2007

Administrative Stuff

- HW3 is due on Friday
- No new HW will be out this week
- Next Tuesday we will have Midterm 1:
  - Sep 18 @ 6:30 – 7:45pm.
  - Location: Curtiss Hall 127 (classroom)
- On Monday we will have a review session
- No class on Friday (Sep 21, 2007)

Quick review of last lecture

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

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
  - It’s 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;
```
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

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.

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

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.

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.
- It should be difficult, if not impossible, for a client to access an object’s variables directly.
- That is, an object should be self-governing.
Method Control Flow

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

```java
myMethod();
```

Method Control Flow

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

```java
main
```

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

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

```java
NumberFormat.format = new NumberFormat(...); 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)

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

| Variables | | Methods |
|-----------|-------------------|
| **public** | **private** | | **Violate encapsulation** | **Enforce encapsulation** |
| **Provide services to clients** | **Support other methods in the class** |

• Members of a class that are declared with **public visibility** can be referenced anywhere
• Members of a class that are declared with **private visibility** can be referenced only within that class
• Members declared without a visibility modifier have **default visibility** and can be referenced by any class in the same package
• An overview of all Java modifiers is presented in Appendix E

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

• Methods that provide the object’s services are declared with public visibility so that they can be invoked by clients
• Public methods are also called **service methods**
• A method created simply to assist a service method is called a **support method**
• Since a support method is not intended to be called by a client, it should not be declared with public visibility

Visibility Modifiers

| Variables | | Methods |
|-----------|-------------------|
| **public** | **private** | | **Violate encapsulation** | **Enforce encapsulation** |
| **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

THE END