Chapter 4: Writing Classes

We've been using predefined classes. Now we will learn to write our own classes to define objects.

Chapter 4 focuses on:
- class definitions
- encapsulation and Java modifiers
- method declaration, invocation, and parameter passing
- method overloading
- method decomposition
- graphics-based objects

Objects
- An object has:
  - state - descriptive characteristics
  - behaviors - what it can do (or what can be done to it)
- For example, consider a coin that can be flipped so that it's face shows either "heads" or "tails"
  - The state of the coin is its current face (heads or tails)
  - The behavior of the coin is that it can be flipped
  - Note that the behavior of the coin might change its state

Classes
- A class is a blueprint of an object
- It is the model or pattern from which objects are created
- For example, the String class is used to define String objects
  - Each String object contains specific characters (its state)
  - Each String object can perform services (behaviors) such as toUpperCase
© 2006 Pearson Education

Classes

- The String class was provided for us by the Java standard class library
- But we can also write our own classes that define specific objects that we need
- For example, suppose we want to write a program that simulates the flipping of a coin
- We can write a Coin class to represent a coin object

© 2006 Pearson Education

The Coin Class

- In our Coin class we could define the following data:
  - face, an integer that represents the current face
  - HEADS and TAILS, integer constants that represent the two possible states
- We might also define the following methods:
  - a Coin constructor, to initialize the object
  - a flip method, to flip the coin
  - a isHeads method, to determine if the current face is heads
  - a toString method, to return a string description for printing

© 2006 Pearson Education

Classes

- A class contains data declarations and method declarations

© 2006 Pearson Education

The Coin Class

- See CountFlips.java (page 199)
- See Coin.java (page 200)
- Note that the CountFlips program did not use the toString method
- A program will not necessarily use every service provided by an object
- Once the Coin class has been defined, we can use it again in other programs as needed

© 2006 Pearson Education
Data Scope

- The scope of data is the area in a program in which that data can be used (referenced).
- Data declared at the class level can be used 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.

Instance Data

- The face variable in the Coin class is called instance data because each instance (object) of the Coin class has its own.
- A class declares the type of the data, but it does not reserve any memory space for it.
- Every time a Coin object is created, a new face variable is created as well.
- The objects of a class share the method definitions, but each 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 variables the object holds and the methods that make the object useful
  - External - the services that an object provides and how the object interacts
- From the external view, an object is an encapsulated entity, providing a set of specific services.
- These services define the interface to the object.
- Recall from Chapter 2 that an object is an abstraction, hiding details from the rest of the system.
Encapsulation

- An object should be self-governing
- Any changes to the object's state (its variables) should be made only by that object's methods
- We should make it difficult, if not impossible, to access an object's variables other than via its methods
- The user, or client, of an object can request its services, but it should not have to be aware of how those services are accomplished

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 value
- We've used the modifier final to define a constant
- We will study two visibility modifiers: public and private

Encapsulation

- An encapsulated object can be thought of as a black box
- Its inner workings are hidden to the client, which invokes only the interface methods

Visibility Modifiers

- Members of a class that are declared with public visibility can be accessed from anywhere
- Public variables violate encapsulation
- Members of a class that are declared with private visibility can only be accessed from inside the class
- Members declared without a visibility modifier have default visibility and can be accessed by any class in the same package
### Visibility Modifiers

- Methods that provide the object's services are usually 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.

### Driver Programs

- A driver program drives the use of other, more interesting parts of a program.
- Driver programs are often used to test other parts of the software.
- The Banking class contains a main method that drives the use of the Account class, exercising its services.
- See Banking.java (page 209)
- See Account.java (page 211)

### Method Declarations

- A method declaration specifies the code that will be executed when the method is invoked (or called).
- When a method is invoked, the flow of control jumps to the method and executes its code.
- When complete, the flow returns to the place where the method was called and continues.
- The invocation may or may not return a value, depending on how the method is defined.
Method Control Flow

- The called method can be within the same class, in which case only the method name is needed.

Method Header

- A method declaration begins with a method header:

  ```
  char calc (int num1, int num2, String message)
  ```

  - **method name**: The name of the method.
  - **parameter list**: The parameter list specifies the type and name of each parameter.
  - **return type**: The name of a parameter in the method declaration is called a formal argument.

Method Body

- The method header is followed by the method body:

  ```
  char calc (int num1, int num2, String message)
  {
      int sum = num1 + num2;
      char result = message.charAt (sum);
  return result;
  }
  ```

  - **sum and result are local data**: They are created each time the method is called, and are destroyed when it finishes executing.
  - **The return expression must be consistent with the return type**: The return expression must be consistent with the return type specified in the method header.
The return Statement

- The return type of a method indicates the type of value that the method sends back to the calling location.
- A method that does not return a value has a void return type.
- A return statement specifies the value that will be returned:
  \[ \text{return expression;} \]
- Its expression must conform to the return type.

Parameters

- Each time a method is called, the actual parameters in the invocation are copied into the formal parameters:
  \[ \text{ch = obj.calc(25, count, "Hello");} \]
  \[ \text{char calc(int num1, int num2, String message)} \]
  \[ \{ \]
  \[ \text{int sum = num1 + num2;} \]
  \[ \text{char result = message.charAt(sum);} \]
  \[ \text{return result;} \]

Preconditions and Postconditions

- A precondition is a condition that should be true when a method is called.
- A postcondition is a condition that should be true when a method finishes executing.
- These conditions are expressed in comments above the method header.
- Both preconditions and postconditions are a kind of assertion, a logical statement that can be true or false which represents a programmer’s assumptions about a program.

Constructors Revisited

- Recall that a constructor is a special method that is used to initialize a newly created object.
- When writing a constructor, remember that:
  - it has the same name as the class.
  - it does not return a value.
  - it has no return type, not even void.
  - it typically sets the initial values of instance variables.
- The programmer does not have to define a constructor for a class.
Local Data

- Local variables can be declared inside a method
- The formal parameters of a method create *automatic local variables* when the method is invoked
- When the method finishes, all local variables are destroyed (including the formal parameters)
- Keep in mind that instance variables, declared at the class level, exists as long as the object exists
- Any method in the class can refer to instance data

Accessors and Mutators

- Since instance data usually has private visibility, it can only be accessed through methods
- An *accessor method* provides read-only access to a particular value
- A *mutator method* changes a particular value
- For a data value $X$, accessor and mutator methods are usually named $\text{get}X$ and $\text{set}X$

Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler determines which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature

Overloading Methods

```
Version 1
double tryMe (int x)
{
    return x + .375;
}

Version 2
double tryMe (int x, double y)
{
    return x*y;
}
```

Invocation

```
result = tryMe (25, 4.32)
```
Overloaded Methods

- The `println` method is overloaded:
  ```java
  println (String s)
  println (int i)
  println (double d)
  ```
  and so on...

- The following lines invoke different versions of the `println` method:
  ```java
  System.out.println ("The total is:");
  System.out.println (total);
  ```

Method Decomposition

- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A service method of an object may call one or more support methods to accomplish its goal
- Support methods could call other support methods if appropriate

Overloading Methods

- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

- See `SnakeEyes.java` (page 221)
- See `Die.java` (page 222)

Pig Latin

- The process of translating an English sentence into Pig Latin can be decomposed into the process of translating each word
- The process of translating a word can be decomposed into the process of translating words that
  - begin with vowels
  - begin with consonant blends (sh, cr, tw, etc.)
  - begins with single consonants

- See `PigLatin.java` (page 224)
- See `PigLatinTranslator.java` (page 225)
Object Relationships

- Objects can have various types of relationships to each other
- A general association is sometimes referred to as a use relationship
- A general association indicates that one object (or class) uses or refers to another object (or class) in some way

Aggregation

- An aggregate object is an object that contains references to other objects
- For example, an account object contains a reference to a String object (the owner's name)
- An aggregate object represents a has-a relationship
- A bank account has a name
- Likewise, a student may have one or more addresses

Applet Methods

- In previous examples we've used the paint method of the Applet class to draw on an applet
- The Applet class has several methods that are invoked automatically at certain points in an applet's life
- The init method, for instance, is executed only once when the applet is initially loaded
- The start and stop methods are called when the applet becomes active or inactive
- The Applet class also contains other methods that generally assist in applet processing
Graphical Objects

- Any object we define by writing a class can have graphical elements.
- The object must simply obtain a graphics context (a Graphics object) in which to draw.
- An applet can pass its graphics context to another object just as it can any other parameter.
- See LineUp.java (page 240).
- See StickFigure.java (page 242).

Summary

- Chapter 4 has focused on:
  - class definitions
  - encapsulation and Java modifiers
  - method declaration, invocation, and parameter passing
  - method overloading
  - method decomposition
  - graphics-based objects