Ch 4: Writing Classes

Java Software Solutions
Foundations of Program Design
Sixth Edition

by
Lewis & Loftus
Classes and Objects

- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality.

- A single die:
  - State: Face that is showing
  - Behavior: roll

- 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.

Coming up: Classes
A class can contain data declarations and method declarations

int size, weight;
char category;

Data declarations
Holds the state!

Method declarations
Defines the behaviors!
• Let's create the Die class
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(myObject);`  
  // Automatically calls the toString method provided by myObject

Coming up: Constructors
Constructors

- 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.

- A constructor has NO RETURN TYPE.
  - public Book(int numPage, String author) { … }
Data Scope

- The **scope** of data is the area in a program in which that data can be referenced (used)

- **Class scope**: Data declared at the class level can be referenced by all methods in that class. *What did we call data declared at the class level?*

- **Local scope**: Data declared within a method can be used only in that method

- *What is local in Die.java? What is class-level?*

Coming up: Instance Data
We can depict the two Die objects from the RollingDice program as follows:

```
1. die1
   +---+
   |   |
   v

2. faceValue 5
```

```
2. die2
   +---+
   |   |
   v

3. faceValue 2
```

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

Intuition: A client doesn’t care HOW it works, only that it satisfies the interface!
Encapsulati}

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

Good class design means USE ENCAPSULATION!
Visibility Modifiers enable encapsulation

- `public` int anInt; // Anyone can access me
  public void aMethod() { … } // Anyone can invoke me

- // Only this class and subclasses of this class
  `protected` int anotherInt;
  `protected` void aMethod2() { .. }

- // Only methods defined in this class
  `private` int anotherInt;
  `private` void aMethod2() { .. }

- // Default visibility: only this class and other classes in the same package
  int yaInt;
  int void yaMethod() { .. }

- 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.

Coming up: Visibility Modifiers
## Visibility Modifiers

<table>
<thead>
<tr>
<th>Public</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variables</strong></td>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td>Violate encapsulation</td>
<td>Provide services to clients</td>
</tr>
<tr>
<td>Enforce encapsulation</td>
<td>Support other methods in the class</td>
</tr>
</tbody>
</table>

Coming up: Accessors and Mutators
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”

- **Lets update the die class!**

**Why do we use these methods?**

**What’s the point?**

Coming up: Method Control Flow
Method Control Flow

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

- Method invocation
- Optionally returns a result
A method declaration begins with a *method header*

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

- **Optional modifier**
  - **method name**
  - **return type**
  - **parameter list**

  The parameter list specifies the type and name of each parameter.

  The name of a parameter in the method declaration is called a *formal parameter*.

Coming up: Method Body
The method header is followed by the *method body*

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

    return result;
}
```

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

Coming up: The return Statement
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

```
return expression;
```

- Its expression must conform to the return type

Coming up: Parameters
Parameters

- When a method is called, the actual parameters in the invocation are copied into the formal parameters in the method header

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

ch = obj.calc (25, count, "Hello");
Local Data

- As we’ve seen, 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.

Coming up: Bank Account Example
Bank Account Example

- Let’s look at another example that demonstrates the implementation details of classes and methods.
- We’ll represent a bank account by a class named `Account`.
- It’s state can include the account number, the current balance, and the name of the owner.
- An account’s behaviors (or services) include deposits and withdrawals, and adding interest.

Coming up: Driver Programs
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 *Transactions* class contains a *main method* that drives the use of the *Account* class, exercising its services

- See [Transactions.java](#)
- See [Account.java](#)

Coming up: Bank Account Example
Bank Account Example

acct1

acctNumber: 72354
balance: 102.56
name: "Ted Murphy"

acct2

acctNumber: 69713
balance: 40.00
name: "Jane Smith"
There are some improvements that can be made to the `Account` class.

- Formal getters and setters could have been defined for all data.
- The design of some methods could also be more robust, such as verifying that the `amount` parameter to the `withdraw` method is positive.

Coming up: Constructors Revisited
Constructors Revisited

- Note that a constructor has no return type specified in the method header, not even `void`.

- A common error is to put a return type on a constructor, which makes it a “regular” method that happens to have the same name as the class.

- The programmer does not have to define a constructor for a class.

- Each class has a `default constructor` that accepts no parameters.