Anatomy of a Method

September 14, 2007

Midterm 1

• Next Tuesday 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, 2006)

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

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

Client-Server Relation

Client
Methods
Data

Server
Methods
Data

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

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

Visibility Modifiers

- 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

- 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

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 `get{x}` and `set{x}`, 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

Examples

Chapter 4
Sections 4.4 & 4.5

Method Declarations

- Let’s now examine method declarations in more detail
- A method declaration specifies the code that will be executed when the method is invoked (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

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

Method Control Flow

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

Method Header

- A method declaration begins with a method header

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

- 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

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;
}
```

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

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

Parameters

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

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

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

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

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 (page 172)
• See Account.java (page 173)

Bank Account Example

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

Bank Account Example
• 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
Run examples from the book

THE END