STUDENT LESSON

A5 – Designing and Using Classes

STUDENT LESSON

A5 – Designing and Using Classes

INTRODUCTION: This lesson discusses how to design your own classes. This can be the most challenging part of programming. A truly good design can be the difference between hundreds of hours working with complex code and two hours working in an elegant system. A well thought out design can make the programming portion far easier. In fact, for many professional projects, more time is spent designing programs than actually typing in code. Imagine a million lines of code in a project with a design flaw. Redesigning that much code could be horrendous!

The key topics for this lesson are:

A. Designing a Class
B. Determining Object Behavior
C. Instance Variables
D. Implementing Methods
E. Constructors
F. Using Classes

VOCABULARY:

ACCESS SPECIFIER
ATTRIBUTES
BEHAVIORS
CONSTRUCTOR
ENCAPSULATION
INSTANCE VARIABLE
OVERLOADING
PSEUDOCODE
REFERENCE
TOP DOWN DESIGN
VARIABLE

DISCUSSION:

A. Designing a Class

1. One of the advantages of object-oriented design is that it allows a programmer to create a new data type that is reusable in other situations.

2. When designing a new class, three components must be identified – attributes, behaviors, and constructors. To determine attributes of a class, look at the nouns associated with that object. To determine behaviors, look at the verbs.

3. Let's consider a checking account at a bank. The account would need to record such things as the account number, the current balance, the type of checking account it is, etc (these are nouns). These would be the attributes of the checking account. It would also need to be able to do certain actions, such as withdrawing or depositing money (these are verbs). These would be the behaviors of the checking account. Finally, the checking account object needs to be created in order to be used, so the class must define how the creation process works. This is accomplished in the constructors.

B. Determining Object Behavior

1. In this section, you will learn how to create a simple class that describes the behavior of a bank account. Before you start programming, you need to understand how the objects of your class behave. Operations that can be carried out with a checking account could be:

   - Accepting a deposit
   - Withdrawing from the account
   - Getting the current balance

2. In Java, these operations are expressed as *method calls*. For example, assume we have an object `checking` of type `CheckingAccount`. Here are the methods that invoke the required behaviors:

```
checking.deposit(1000);
checking.withdraw(250);
System.out.println("Balance: " + checking.getBalance());
```

   These methods form the behaviors of the `CheckingAccount` class. The behaviors are the list of methods that you can apply to objects of a given class. To the client, an object of type `CheckingAccount` can be viewed as a "black box" that can carry out its methods. The programming concept of not needing to know how things are done internally is called abstraction.

3. Once we understand what objects of the `CheckingAccount` class need to do, it is possible to design a Java class that implements these behaviors. To describe object behavior, you first need to implement a class and then implement methods within that class.

```
public class CheckingAccount{
  // CheckingAccount data

  // CheckingAccount constructors

  // CheckingAccount methods
}
```

   Next we implement the three methods that have already been identified:

   - `deposit`
   - `withdraw`
   - `getBalance`

```
public class CheckingAccount{
  // CheckingAccount data

  // CheckingAccount constructors

  public void deposit( double amount ){
    // method implementation
  }
  public void withdraw( double amount ){
    // method implementation
  }

  public double getBalance(){
```

```
    // method implementation
  }
}
```

4. What we have been doing here is not real code and wouldn't actually do anything. However, it is useful to lay out what your class will look like. When we use a mixture of English and Java to show what we are doing, it is called *pseudocode*. In this example the implementation of the methods is left out because we do not have all the information that we need yet. However, we can still write out what the methods will do with pseudocode so that it becomes easier to see how everything will fit together. This process of starting with a very broad concept or outline and working down to smaller and smaller details is called *top-down design*.

```
public class CheckingAccount{
  // CheckingAccount data

  // CheckingAccount constructors

  public void deposit( double amount ){
    // receive the amount of the deposit
    // and add it to the current balance
  }
  public void withdraw( double amount ){
    // remove the amount of the withdrawal
    // from the current balance
  }

  public double getBalance(){
    // return the current balance in a double value
  }
}
```

5. A method header consists of the following parts:

   access_specifier return_type *method_name* (*parameters*)

   a. An ***access_specifier*** (such as **public**). The access specifier controls where this method can be accessed from. Methods should be declared as public if the method needs to be accessed by something other than the object containing the method. If it should only be accessed within the object, you should declare the method as private.

   b. The ***return_type*** of the method such as **double, void,** or **DrawingTool**. The return type is the data type that the method sends back to the call of the method. This can be any primitive type or any object that your class knows about. For example, in the `CheckingAccount` class, the `getBalance` method returns the current account balance, which is a floating-point number, so its return type is **double**. The `deposit` and `withdraw` methods don't return any value. To indicate that a method does not return a value, you use the keyword **void**.

   c. The *method_name* (such as `deposit`). The name needs to follow the rules of identifiers and should indicate the method's purpose.

d. A list of the *parameters* of the method. The parameters are the input to the method. The `deposit` and `withdraw` methods each have one parameter, the amount of money to deposit or withdraw. The type of parameter, such as **`double`**, and name for each parameter, such as `amount`, must be specified. If a method has no parameters, like `getBalance`, it is still necessary to supply a pair of parentheses `()` behind the method name.

6. Once the method header has been specified, the implementation of the method must be supplied in a block that is delimited by braces `{...}`. The `CheckingAccount` methods will be implemented later in Section D.

C. Instance Variables

1. Before any code can be written for the behaviors, the object must know how to store its current *state*. The state is the set of attributes that describes the object and that influences how an object reacts to method calls. In the case of our checking account objects, the state includes the current balance and an account identifier.

2. Each object stores its state in one or more *instance variables*.

```
public class CheckingAccount{
  private double myBalance;
  private int myAccountNumber;

  // CheckingAccount constructors

  // CheckingAccount methods
}
```

3. An instance variable declaration consists of the following parts:

   `access_specifier type` *`variable_name`*

   a. The ***`access_specifier`*** (such as **`private`**) tells who can access that data member. Instance variables are generally declared with the access specifier **`private`**. That means they can be accessed only by methods of the *same class*. In particular, the balance variable can be accessed only by the `deposit`, `withdraw`, and `getBalance` methods.
   b. The ***`type`*** of the variable (such as **`double`**).
   c. The *`variable_name`* (such as `myBalance`).

4. If instance variables are declared private, then all external data access must occur through the non-private methods. This means that the instance variables of an object are hidden. The process of hiding data is called *encapsulation*. Although it is possible in Java to define instance variables as **`public`** (leave them unencapsulated), it is very uncommon in practice. In this curriculum, instance variables will always be made **`private`**.

5. For example, because the `myBalance` instance variable is **`private`**, it cannot be accessed from outside of the class:

```
double balance = checking.myBalance;  // compiler ERROR!
```

However, the public `getBalance` method to inquire about the balance can be called:

```
double balance = checking.getBalance();  // OK
```

D. Implementing Methods

1. Now that we know how the object stores its state, we can provide the implementations for the methods of the class. The implementation for three methods of the `CheckingAccount` class is given below.

```
public class CheckingAccount{
  private double myBalance;
  private int myAccountNumber;

  public double getBalance(){
    return myBalance;
  }

  public void deposit(double amount){
    myBalance += amount;
  }

  public void withdraw(double amount){
    myBalance -= amount;
  }
}
```

2. The implementation of the methods is straightforward. When some amount of money is deposited or withdrawn, the balance increases or decreases by that amount.

3. The `getBalance` method simply *returns* the current balance. A **`return`** statement obtains the value of a variable and exits the method immediately. The return value becomes the value of the method call expression. The syntax of a **`return`** statement is:

```
return expression;
```

or

```
return;  // Exits the method without sending back a value
```

E. Constructors

1. The final requirement to implement the `CheckingAccount` class is to define a *constructor*, whose purpose is to initialize the values of instance variables of an object. To construct objects of the `CheckingAccount` class, it is necessary to declare an object variable first.

```
CheckingAccount checking;
```

Object variables such as `checking` are *references* to objects. Instead of holding an object itself, a reference variable holds the information necessary to find the object in memory. This is the *address* of the object.

2. The object identifier `checking` does not refer to any object yet. An attempt to invoke a method on this variable would cause a runtime null pointer exception error. To initialize the variable, it is necessary to create a new `CheckingAccount` object using the **`new`** operator

   ```
   checking = new CheckingAccount();
   ```

   Constructors are always invoked using the **`new`** operator. The **`new`** operator allocates memory for the objects, and the constructor initializes it. The "new" operator returns the reference to the newly constructed object.

   In most cases, you will declare and store a reference to an object in an object identifier on one line as follows:

   ```
   CheckingAccount checking = new CheckingAccount();.
   ```

   Occasionally, it would be repetitive and unnecessary to create an object identifier. If the purpose of creating the object is only to pass it in as an argument, you can simply create the object within the method call. For example, when creating `DrawingTool` objects, and you are providing a `SketchPad` object, you do not need to create an identifier for that `SketchPad` object:

   ```
   DrawingTool pen = new DrawingTool(new SketchPad(500,500));
   ```

   Notice that we never create an object identifier for the SketchPad object.

3. Constructors always have the same name as their class. Similar to methods, constructors are generally declared as public to enable any code in a program to construct new objects of the class. Unlike methods, constructors do not have return types.

4. Instance variables are automatically initialized with a default value (0 for number, false for boolean, null for objects). Even though initialization is handled automatically for instance variables, it is a matter of good style to initialize all instance variables explicitly. Generally, all of your instance variables should be initialized in your constructor.

5. Many classes define more than one constructor through *overloading*. For example, you can supply a second constructor for the `CheckingAccount` class that sets the `myBalance` and `myAccountNumber` instance variables to initial values, which are the parameters of the constructor:

   ```
   public class CheckingAccount{
   // CheckingAccount data
      private double myBalance;
      private int myAccountNumber;

   // constructor initializes values to default settings
   ```

```
  public CheckingAccount(){
    myBalance = 0.0;
    myAccountNumber = 0;
  }

  public CheckingAccount(double initialBalance, int acctNum){
    myBalance = initialBalance;
    myAccountNumber = acctNum;
  }
  // CheckingAccount methods
}
```

The second constructor is used if you supply a starting balance and an account number as construction parameters.

```
CheckingAccount checking = new CheckingAccount(5000.0, 12345);
```

The number of constructors is based on the needs of the client.

6. The implementation of the `CheckingAccount` class is complete and given below:

```
public class CheckingAccount{
  private double myBalance;
  private int myAccountNumber;

  public CheckingAccount(){
    myBalance = 0.0;
    myAccountNumber = 0;
  }

  public CheckingAccount(double initialBalance, int acctNum){
    myBalance = initialBalance;
    myAccountNumber = acctNum;
   }

  public double getBalance(){
    return myBalance;
  }

  public void deposit(double amount){
    myBalance += amount;
  }

  public void withdraw( double amount ){
    myBalance -= amount;
  }
}
```

F. Using Classes

1. Using the `CheckingAccount` class is best demonstrated by writing a program that solves a specific problem. We want to study the following scenario:

An interest bearing checking account is created with a balance of $1,000. For two years in a row, add 2.5% interest. How much money is in the account after two years?

2. Two classes are required: the `CheckingAccount` class that was developed in the preceding sections, and a second class called `CheckingTester`. The main method of the `CheckingTester` class constructs a `CheckingAccount` object, adds the interest twice, then prints out the balance.

```
class CheckingTester{
  public static void main(String[] args){
    CheckingAccount checking =
        new CheckingAccount(1000.0, 123);

    double INTEREST_RATE = 2.5;
    double interest;

    interest = checking.getBalance() * INTEREST_RATE / 100;
    checking.deposit(interest);

    System.out.println("Balance after year 1 is $"
        + checking.getBalance());

    interest = checking.getBalance() * INTEREST_RATE / 100;
    checking.deposit(interest);

    System.out.println("Balance after year 2 is $"
        + checking.getBalance());
  }
}
```

SUMMARY/ REVIEW: The topics in this lesson are critical to your study of computer science. The concepts of abstraction and OOP will continue to be developed in future lessons. Designing your classes is the most important part of programming. Without good design in the beginning, a complex program can quickly grow out of control.

ASSIGNMENT:
Lab Exercise A5.1, *PiggyBank*
Lab Exercise A5.2, *Müller*
Worksheet A5.1, *Introduction to Classes*
Worksheet A5.2, *Methods*