Advanced Java

LESSON 7: ENCAPSULATION

Creative Commons

Attribution 4.0 International (CC BY 4.0)

This workforce product was funded by a grant awarded by the U.S. Department of Labor's Employment and Training Administration. The product was created by the grantee and does not necessarily reflect the official position of the U.S. Department of Labor. The U.S. Department of Labor makes no guarantees, warranties, or assurances of any kind, express or implied, with respect to such information, including any information on linked sites and including, but not limited to, accuracy of the information or its completeness, timeliness, usefulness, adequacy, continued availability, or ownership. This is an equal opportunity program. Assistive technologies are available upon request and include Voice/TTY (771 or 800-947-6644).

Agenda

- Encapsulation
- Check List
- Activities

Encapsulation

Objectives of Encapsulation

- **Hide Complexity**
(increases quality – easier to use code)
- **Expose only a small set of public methods**
(the public interface to your object)
- **Classes and Methods Should Have a Single Responsibility**
(rely on collaboration with other classes and methods)
- **Programmer controls access**
(object works as designed – meets spec)

Why make your code easy to use?

- **Always assume your code will be used by other developers**
  (code reuse)
- **Will they be able to use it easily and correctly?**
  (object works as designed – meets spec)
- **If it's hard to understand will you or others be able to make changes quickly and accurately?**
  (worker efficiency, quality)

Real world example

Buttons encapsulate technical details of electric circuits

Too many buttons (public methods) make device hard to use

Encapsulation: key concepts

Hide Properties

```
public class Dog {
        private int age;
        private String name;
}
```

Encapsulation: key concepts

Only Expose Properties via Public Getters and Setters (also called Accessors and Mutators)

```
public class Dog {
      private int age;
      public int getAge() {
          return age;
      }
      // You are in control!!!
      public void setAge(int value) {
          if(value < 0) { System.exit(1);}
          age = value;
      }
}
```

Hidden Complexity: Always validate!

Encapsulation: key concepts

Avoid making all method public
(private methods hide details)

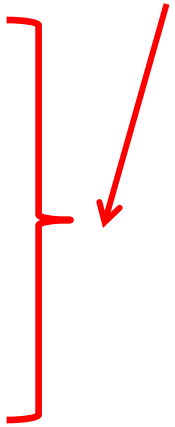
```
// Code to chase a Postman
public class Dog {

      public void startRunning() {}

      public void navigateToPostman() {}

      public void closeGapToPostman();
}
```

Too much detail exposed!



Encapsulation: key concepts

Avoid making all method public
(private methods hide details)

```
// Code to chase a Postman
public class Dog {
      public void chasePostman() {
         startRunning();
            navigateToPostman();
            closeGapToPostman();
      }
      private void startRunning() {}
      private void navigateToPostman() {}
      private void closeGapToPostman();
}
```

Details are Hidden!

Limit dependencies

When you start a car in real life, you turn a key and the car starts the engine

So why would a Person need to know about (depend on) the engine object?

Declare the Engine object inside the Car object (composition) … that way the Person only needs to collaborate with the Car.

Principles

- **Don’t Repeat Yourself (DRY principle)**
  (modularize your code)
- **Single Responsibility Principle**
  (Don’t Repeat Yourself: DRY)
- **Limit Dependencies on other objects**
  (increase flexibility)
- **Command-Query Separation Principle**
- **No Magic Numbers (literal values)**
  (use constants for easy editing)

Related principles

- **Composition**
  (object containing objects and delegating work to them)
- **Private Helper methods**
  (don't expose all methods as public – hide complexity)
- **Private instance properties**
  (don't give direct access, use getters/setters to control)
- **Be modular**
  (break your code into many small classes – follow SRP)

Related principles

Single Responsibility Principle

A class or method should have a most one responsibility. Note that a responsibility is not a method, so we are not saying that a class should have just one method. Nor are we saying that a method should have just one line of code. A responsibility is work that may encompass many functions, but all functions are geared to the same goal.

For example, a class “Person” whose main responsibility is to be Person-like may have many methods … setName, getName, getAge, speak(), etc. But doing a System.out.println() is NOT a job for a Person object. That is output and output should be delegated to an object whose job it is to do that – say a ConsoleOutput object. Same is true for a method. If you have a method that calculates a tip, it should not output that tip to the console.

Check List

Encapsulation Check List

Property Encapsulation

- Declare all class and instance properties private. No exceptions. This prevents storage of values that do not meet requirements. Constants may be public because they cannot be changed.
- For each private class or instance property consider providing a public getter and setter method, unless you want read only properties, in which case you can eliminate the setter. A public setter method gives the programmer control over the values passed into the set method. Using if logic, values can be compared against valid values or ranges of values.
- Don’t store a calculated value in a property. If you do, the value will always be stale if running the calculation again with different parameters changes the result. Just call a getter method that produces the result of the calculation.

Encapsulation Check List

Method Encapsulation

- In any class consider whether you really need to make all methods public. If a method is just a helper method used by another method in the class, and shouldn't be accessed by independently by other objects, then make it private.
- Remember the **Single Responsibility Principle** applies to classes and methods. If you have a method doing too much work, split that method up into smaller pieces (helper methods) and call them from a single parent method. Creating smaller methods also makes your code more modular, more readable and more flexible. When used with a parent method you can also control execution order.
- **Don't Repeat Yourself (DRY).** If you have lines of code duplicated in multiple methods, create a helper method for the duplicate code and call that instead. Now you have only one place to do edits should that code need to change.

Encapsulation Check List

Method Encapsulation (Continued)

- When the order of method execution matters, make those methods private and call them from a single pubic parent method.
- All public method parameters should be validated, unless the argument value does not matter.
- Command-Query Separation Principle: It states that every method should either be a command that performs an action (set, create, output, etc.), or a query that returns data to the caller (get, find, retrieve, etc.), but not both. In other words, asking a question should not change the answer. And neither type should perform a task like the other, or produce side effects. So, for example, a query method should just retrieve data and not cause internal or external state to change (a side-effect). A command method is usually designed to perform a task that does change internal (instance property) or external state (a database or file, e.g.). But it shouldn't return any data – that's not its job.

Encapsulation Check List

Class Encapsulation

- The Principle of Least Knowledge states that a class should know as little as possible about other classes in the system. A good analogy is a Car has an Engine and needs to know about it to start the Engine. But a Person who has a Car does not need to know about the Engine to start it. Simply hide the details about starting the Engine inside of a method in the Car class, e.g., start(). Now the Person object calls the start method of the Car object. Hiding such details and complexity makes software easier to use and therefore prevents errors. It also minimizes dependencies, which makes our code more flexible. You will find a sample of this example in the “Encapsulation” project.
- Look for ways to delegate work to other objects, effectively hiding the details of how that work is performed.
- The key to properly encapsulating classes is to be clear about the responsibilities of each class. Before you decide to perform some work in a class consider whether you should be doing this work at all. Delegate that work to some other object. For example, are writing a method that performs output? Is output the Single Responsibility of that class and/or method? No? Then delegate that work to a class or method whose job is to do output.

Naming Best Practices

- All class names must be nouns, singular and start with a capital letter and be camel case there after. Names should be nouns. Example: BalanceSheet
- All class and instance property names should begin with a lower-case letter and be camel case thereafter (no spaces or weird punctuation; no cryptic names). Example: yAxisCoordinate
- All method names should contain a verb (remember these are action names) and begin with a lower-case letter and be camel case thereafter (no spaces or weird punctuation; no cryptic names).
  Example: setLastName(String value)
- No cryptic names! In general it's better to have names that are unusually long vs. unusually short if it helps readability.

In-Class Activity

1. The “Encapsulation” sample project on Blackboard will be used for this lab. (This project is NOT Git-enabled)
2. Enable it for Git and then create an empty project on GitHub and push your local copy to GitHub before proceeding. Remember, add and commit often and push your changes to GitHub when done working for the day.
3. In the Encapsulation project you will find several packages. Those NOT labeled“Lab1, Lab2, etc.” are for discussion purposes only. These will be discussed now as examples of good and poor encapsulation.

In-Class Activity (Continued)

4. There are four packages: lab1, lab2, lab3 and lab4. You will work on each of these in order, practicing Property, Method and Class Encapsulation. Each lab gets progressively harder. Perform them in order. Instructions for each lab are at the top of the “Employee” class in each lab. Read these instructions carefully. Forlab4 a solution is also provided. Please do not look at the solution until you have made your best effort to solve lab4 on your own. Only peek if you have to. Please note that much of this lab work involves Critical Thinking and lots of trial and error, where you consider what you have learned and try to find ways to apply those skills to this new problem. This is the best way to learn but it may feel like you have been set adrift, alone in the sea, without any help at all. Not true! You can get help – from your peers and from your instructor. But don’t ask for help until you have tried your best. And don’t let anyone write the code for you.