Waukesha County Technical Institute
152-135 Advanced Java Class

Encapsulation Checklist

When writing code use this checklist to make sure you are following the best practices presented. This may require some critical thinking and analysis so be prepared to use your brain and ***think*** about what you are doing and what could be done better. A glossary of terms is provided at the end of this doc.

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 calulation.

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.

- ❑ 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.

  Examples:

  A Query method to calculate and return a tip amount without changing state (calculation performed on the fly):

  ```
  public double getTip(double billAmount, double tipPercent)
  ```

  A combination of Command and Query methods to change state and return a value:

  ```
  public void storeTip(double billAmount, double tipPercent)
  ```

```
public double getTip()
```

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.

Glossary

Encapsulation

Hiding data (properties), functionality (methods) and entire objects (classes). Once hidden these items are exposed in a controlled way. But encapsulation is more than just hiding things, it's also about combining functionality in a single place. So for example, a method that sets a value can also do validation of the value being set for greater reliability. This combination of these hidden tasks is also encapsulation. (Think of a drug capsule that encapsulates many tiny specs of that drug substance. Those specs are combined and hidden within the capsule.)

Class Property

A property declared in a class as “static”. Static properties live in the class space in memory and are global in the sense that one static property is shared by all instances of that class.

Instance Property

A non-static property declared in a class. Every instance created from a class will have its own version of an instance property and it will be stored in memory with the instance, not in the class. These are not global and are not shared.

Getter Method (Synonym: Accessor)

A method that starts with “get” and ends with the property name as camel case. For example a property named “hairColor” will have a getter method named “getHairColor”. Getter methods must always return something.

Setter Method (Synonym: Mutator)

A method that starts with “set” and ends with the property name as camel case. For example a property named “hairColor” will have a setter method named “setHairColor”. Setter methods do not return values, but do require at least one method parameter – the thing you are setting.

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. So 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 hae a method that calculates a tip, it should not output that tip to the console.