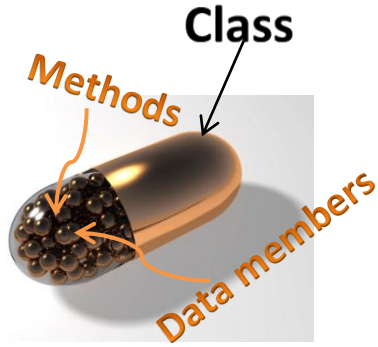
# Secure Coding Lab 2 – Encapsulation

## PREREQUISITE

Prior to completing this lab, students should have a basic knowledge of and have practice with creating new classes in Java. They should be familiar with the concept of encapsulation and related terminology.

## SUMMARY




One of the four major principles of object-oriented programming (OOP) is encapsulation. This lab will focus on the principle of encapsulation and its potential impact on secure code.

Encapsulation allows programmers to control/restrict access to class **data** (also called data members, fields, or instance variables) and to class **methods**. Common recommendations in secure coding guidelines (like those guidelines mentioned below) are: *Unless there is a compelling reason otherwise, programmers should limit the accessibility of class data and methods within a class. Input validation should be performed on all method parameters to ensure data falls within reasonable ranges and expectations.*

## RISK

If classes are not constructed properly and/or do not limit access to data members and methods, attackers can write code to access and maliciously destroy data. Failure to validate method parameters can result in improper calculations, runtime exceptions, and invalid values for instance variables.

## GUIDELINES AND RECOMMENDATIONS FOR ENCAPSULATION

The following guidelines and guidance materials address secure coding relative to encapsulation. Students are encouraged to read through this material.

Secure Coding Guidelines for Java SE

- Guideline 0 Fundamentals: 0-6 Encapsulate
- Guideline 4 Accessibility and Extensibility
  http://www.oracle.com/technetwork/java/seccodeguide-139067.html

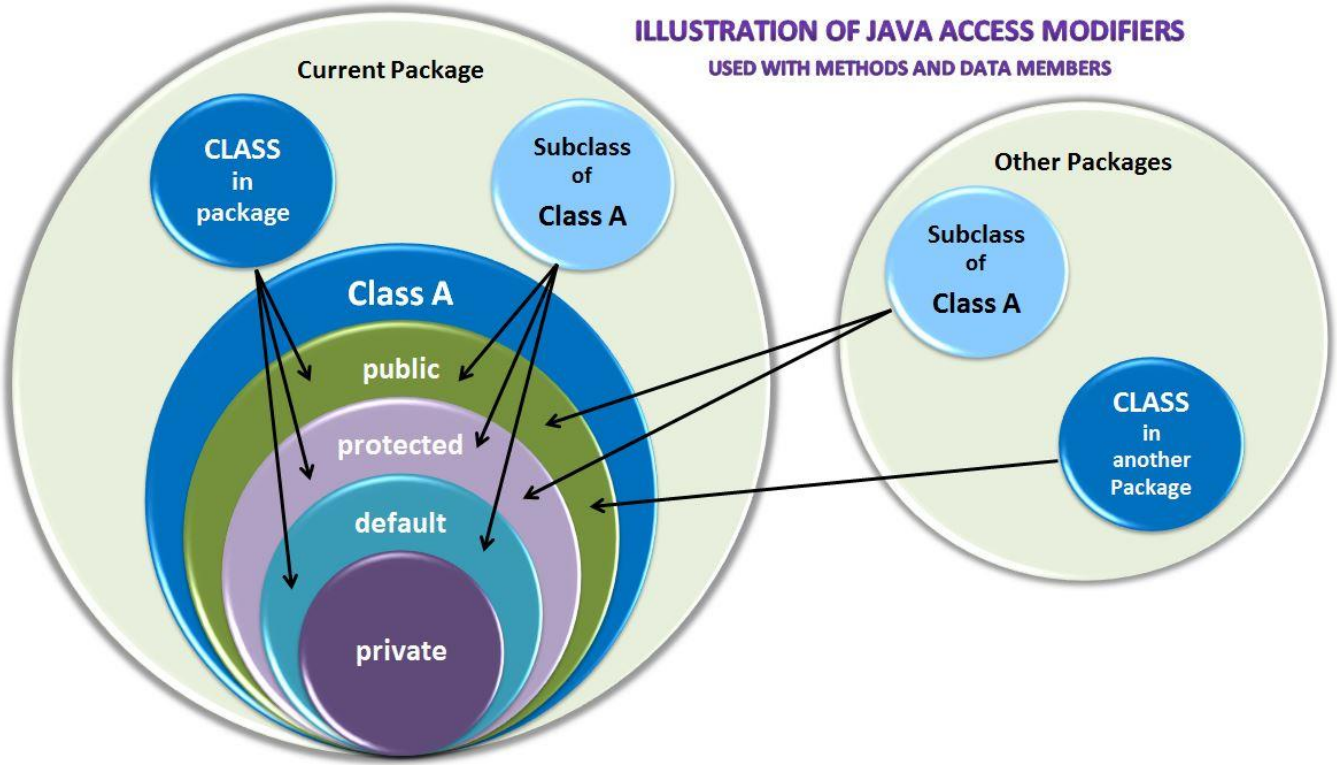
ACM Computer Science Curricula 2013

- Knowledge area: PL – Programming Languages – Core Tier 1 and Core Tier 2
  *(Secure coding topics to include in computer science curriculum)*
  https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf

## OVERVIEW OF TOPIC:

In Java, each of the data members and methods in a class definition can be qualified with an **access modifier**. The following access modifiers can be used:

| Access Modifier | Accessible by the **current class** | Accessible in the **same package** (subclasses and other classes) | Accessible to **subclasses** located in **other packages** | Accessible to **classes** located in **other packages** |
|---|---|---|---|---|
| **public** | Yes | Yes | Yes | Yes |
| **protected** | Yes | Yes | Yes | No |
| **no modifier** (default or also called package-private) | Yes | Yes | No | No |
| **private** | Yes | No | No | No |




The stacked Venn diagram for Class A in the figure above visually illustrates increasing accessibility for the Java access modifiers. For example, the **private** modifier is the most restrictive (only available to the current class in which it is defined) and the **public** modifier is the least restrictive (available to all external classes and subclasses).

When encapsulation is implemented properly it controls/limits access to the fields and methods within a class. For this reason, encapsulation is a recommended secure coding topic in beginning programming courses. One of the purposes of this lab is to learn more about using access modifiers with data members.

In addition to class accessibility regarding security, there are several advantages of proper encapsulation. Hiding the complexity of a class provides a very simple way for other developers to *use* objects from the given class without needing to *understand* the internal details of the class. For large projects that are designed and coded by a team of programmers, encapsulation can save development time and facilitate separation of team tasks. Encapsulated code is more flexible and can provide an efficient way of incorporating new requirements.

## LABORATORY REVIEW:

1. Review the concept of encapsulation. You may want to reference Java textbooks you have used in the past or research a bit on the Internet. There are a host of websites and online videos which explain encapsulation. Pay particular attention to descriptions and examples of access modifiers (public, protected, default (also call package-private), and private).

2. Watch the following videos which walk through an example of class creation and encapsulation in Java. These videos are meant for students who are new to creating classes in Java (which Android also uses). They include a discussion on the anatomy of a Java class, the public access modifier, and the private access modifier. Students who are familiar with creating classes in Java will find this information to be a basic review.

    1. UML Diagram: http://youtu.be/1lsd7A9DGLk
    2. Encapsulation – Data and Constructors: http://youtu.be/paAZ0-X2RsA
    3. Encapsulation – Accessors and Mutators: http://youtu.be/DKI_kvMcdVc
    4. Encapsulation – Additional Methods and Instantiation: http://youtu.be/qFyBUJn-iQI

    *Transcripts of videos available upon request from cindy.tucker@kctcs.edu*

## LABORATORY ASSIGNMENT:

1. Accompanying this assignment is a compressed file named **CardGame.zip** which contains the following files:

    - Card.java
    - CardGame.java

    Save the ZIP file on your computer, unzip it, and open the two Java files it contains using the IDE/editor that you typically use to compile and run Java.

2. Print Card.java. Complete the following actions using your printed copy of Card.java:

## Secure Coding Vulnerabilities: Improper Encapsulation

***Complete the following actions using a printed copy of a Java class.***
***Place a check beside each box as the task is completed.***

*Locate vulnerabilities in data members (instance variables and constants):*

- ☐ Place a ✓ beside each data member with **private** access.
- ☐ Write **V1** beside each data member with **public** access.
- ☐ Write **V2** beside each data member with **no access** modifier (also called the default modifier or package-private modifier).

  *(The protected modifier is related to inheritance and is not covered in this lab.)*

*Locate vulnerabilities in methods:*

- ☐ Place a ✓ beside the method header of each method that does not return a value and does not update any data member's value.
- ☐ Write **V3** beside the method header of each method that returns the value of a data member. These methods are often called accessors or getters.
- ☐ Write **V4** beside the method header of each method that modifies the value of data members. These methods are often called mutators or setters.
- ☐ Write an **P** beside the method header of each method (including constructors) that uses a parameter list. All parameter values should be validated to ensure they fall within the bounds of the method's intended purpose. For each parameter, write **V5** over that parameter's first appearance in code within the method.

*Eliminate vulnerabilities where feasible:*

- ☐ For each data member marked with **V1**: Consider changing the access modifier to **private**. Unless there is a compelling reason, each data members should be declared using the **private** access modifier to protect it from direct access and/or manipulation from outside classes. If the value of the data member is needed by external classes, use the **private** access modifier and create an accessor method for that data member. ***Mark any changes you made on your program listing.***
- ☐ For each data member marked with **V2**: Consider adding the **private** access modifier. Unless there is a compelling reason, each data members should be declared using the **private** access modifier to protect it from direct access and/or manipulation from outside classes. If the value of the data member is needed by external classes, use the **private** access modifier and create an accessor method for that data member. ***Mark any changes you made on your program listing.***
- ☐ For each method marked with **V3**: These are accessor methods which return the value of a data member. Consider the purpose of this method. Does an external class really need this value? Secure

coding guidelines recommend avoiding accessor methods that are not truly needed. ***Strike-through the method header for each accessor method that you recommend deleting from the class you are reviewing.*** If there are data members that you feel need an accessor method, consider the sensitivity of that data item (such as social security numbers, credit card numbers, etc.) ***Write E (for encryption) beside the method header for any accessor method that you recommend using encryption on the returned value.***

- ☐ For each method marked with **V4**: These are mutator methods which modify the value of one or more data members. Review the code for each data members being modified. Is this method needed? Do not include a mutator method for a data member unless it is absolutely needed. ***Strike-through the method header for each mutator method that you recommend deleting from the class you are reviewing.***
- ☐ **For each parameter marked with V5:** Review the method's code for each use of the parameter to ensure the parameter's value is validated before it is used in the method. ***For any parameter that is not validated, write code, on the program listing, that will validate the parameter's value and handle invalid data properly.***

3. Using your marked-up Card.java listing, update the class to improve class encapsulation.
4. Compile your modified version of the Card class. Debug the program to ensure it is working properly with CardGame.java. Modify CardGame.java, if needed (i.e. remove references to methods you may have removed from Card.java).

Scan your marked up class for submitting your work.

## DISCUSSION QUESTIONS:

1. Define encapsulation.
2. What are four different access modifiers in Java? Explain each.
3. Explain why method parameters should be validated before they are used in a method.
4. Why are private data members in a class more secure than public data members?
5. Why are private data members in a class more secure than package-private data members (fields without an access modifier)?
6. From a security viewpoint, why should you not routinely create an accessor for each instance variable?
7. From a security viewpoint, why should you not routinely create a mutator for each instance variable?
8. Describe the changes you made to Card.java and CardGame.java and why you made each change.

Type the answer to these questions in a Word document.

## DELIVERABLES:

- Zip together your updated Card.java and CardGame.java files along with your scanned marked up Card class and your Word Document. (Do not zip folder and additional files that are created from IDEs such as Eclipse, Netbeans, etc. Only zip the two indicated files.) Submit the zipped file as directed by your instructor.