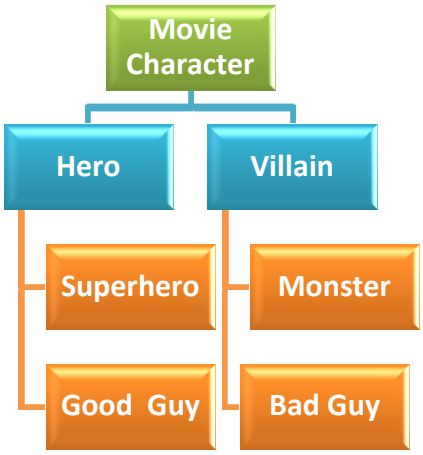
Secure Coding – Extensibility through Inheritance in Java

PREREQUISITE

Prior to completing this lab, students should have a basic knowledge of Java inheritance and related terminology. It is assumed that students will have previously created a simple class hierarchy which included a superclass with one or more subclasses.



SUMMARY

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

Inheritance provides a way to develop class hierarchies whereby new objects can take on the properties of existing objects. There are several benefits of inheritances. For example,

- ***Reusability*** – subclasses can use methods defined in a superclass eliminating duplication of code
- ***Extensibility*** – subclasses can extend a superclass to expand capabilities of the parent class
- ***Data hiding*** – a superclass can control access/modification to its data from other classes and subclasses

RISK

If a class inheritance structure is not designed well, the objects, data, and methods it supports can be vulnerable to malicious attacks. For example,

- a subclass can be created or extended with unauthorized access to data and methods in a superclass
- a subclass can override a method in a superclass class to provide unauthorized access to data or methods in the superclass

GUIDELINES AND RECOMMENDATIONS FOR INHERITANCE AND EXTENSIBILITY

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

Secure Coding Guidelines for Java SE

- 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)
  http://www.acm.org/education/CS2013-final-report.pdf

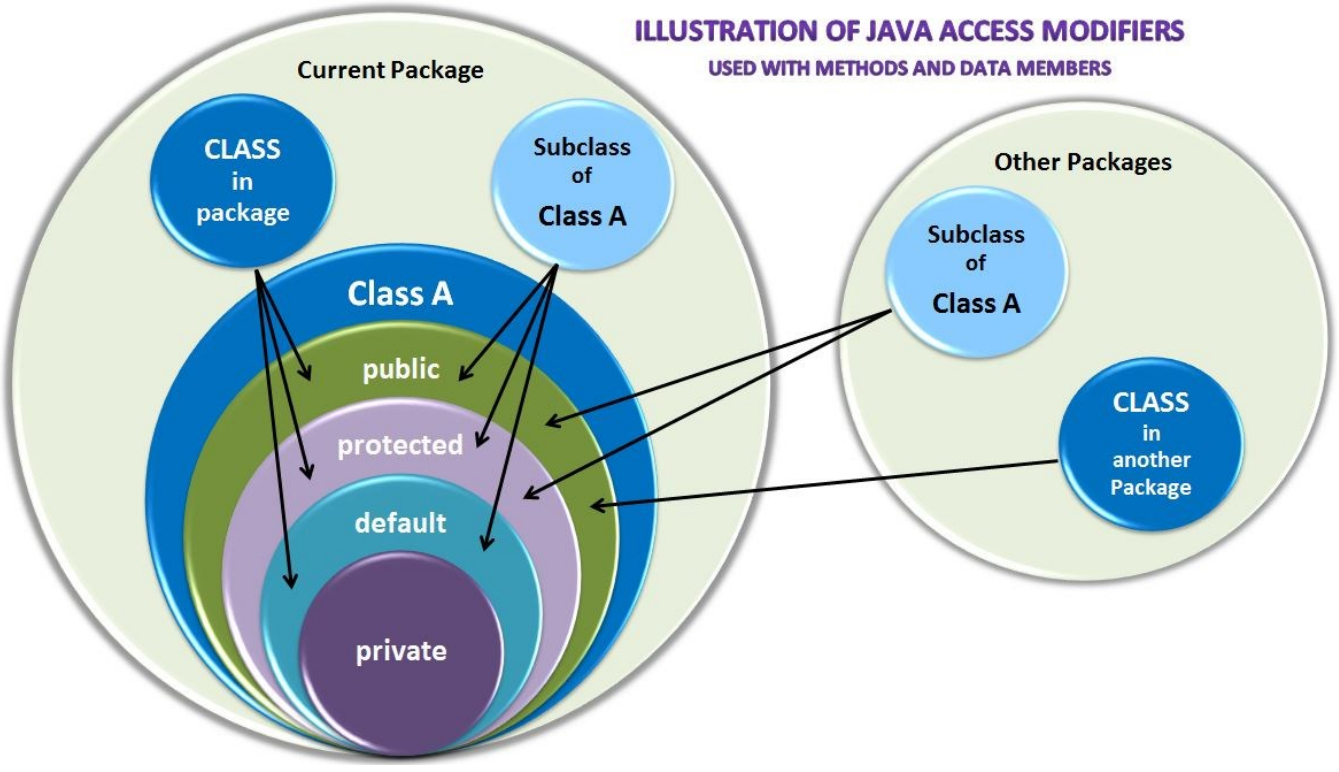
Secure Coding – Extensibility through Inheritance in Java

OVERVIEW OF TOPIC

Inheritance is the process of deriving a new class (referred to as a subclass or child class) from an existing class (referred to as a superclass or parent class). The subclass "inherits" properties (data) and activities (methods) from its superclass.

In Java, each of the data members and methods in a class definition, including classes using inheritance, can be qualified with an **access modifier**. The following access modifiers are available:

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 previous figure 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 inheritance is designed and implemented properly it controls/limits access to the data and methods within its classes. The first objective of this lab is to learn more about using Java access modifiers with class members and the impact they have on inheritance and secure coding. This exercise will explore the **protected** access modifier. *(A different secure code lab focuses on the public, private, and package-private modifiers.)*

Java also has a non-access modifier, **final**, that controls/limits the ability of a class to be extended, a method to be overridden, and an instance variable to be modified. For example,

- ***On a class header***, the class cannot be extended
  - An example: public final class SomeFinalClass { … }
- ***On a method header***, the class cannot be overridden
  - An example: public final void someFinalMethod( ) { … }
- ***On a declaration of an instance variable***, the value for the variable cannot be modified
  - An example: public final double SOME_CONSTANT_NUMBER = 145.8231495;

A second objective of this lab is to learn more about using the Java **final** modifier to limit class extensibility and to restrict select methods from being overridden.

LABORATORY REVIEW

1. Research and study the concept of inheritance. You may find it useful to review Java textbooks you have used in the past. There are a host of webpages and online videos which illustrate this concept. Pay attention to access modifiers (*public, protected, package-private,* and *private*). Review the non-access *final* modifier.

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

   - Box.java
   - StorageBox.java
   - DepositBox.java
   - BoxDriver.java
   - NewBox.java

   Save the ZIP file on your computer, unzip, and open the Java files in the IDE/editor that you use to compile and run Java.

3. Inheritance and the protected access modifier with DATA MEMBERS :

a. Open and review the **Box** class in your preferred IDE for Java. The **protected** access modifier used with the instance variables in the class (lines 16-18 for *height*, *length*, and *width*) provides direct access to these variables by any subclass of **Box**. This means a subclass does not need to use a method to read or modify these data items. This can create security vulnerabilities. Compile this class.

b. Open the **DepositBox** class. Notice this is a child class of **Box**. Review the code so that you understand the class definition. Compile the class.

c. Open the **StorageBox** class. Notice this is a child class of **Box**. Review the code so that you understand the class definition. Compile the class.

d. Open the **BoxDriver** class. This class has several instantiations from the **Box**, **StorageBox**, and **DepositBox** classes. All are a part of the **Box** class hierarchy. Take your time to review the **BoxDriver** code to understand it. Notice the following objects:

- simpleBox is a **Box** object.
- pizzaBox is a **StorageBox** object.
- jewelryBox is a **StorageBox** object.
- companyDepositBox is a **DepositBox** object instantiated with a specified size, box number, and number of objects placed in the box.
- personalDepositBox is a **DepositBox** object instantiated without a size. It will default to a size of 4x24x10. It does not have an assigned box number and the number of items in the box defaults to zero.

Compile and run **BoxDriver**. Pay attention to the output and compare it to the code from the classes so that you fully understand the Box class structure.

e. Let's look at security vulnerabilities in this structure. Open the **NewBox** class. This is an empty child class of **Box**. Add the following constructor starting on line 18 and compile the class. Notice this code is directly accessing *height*, *length*, and *width* which are data members defined in **Box**:

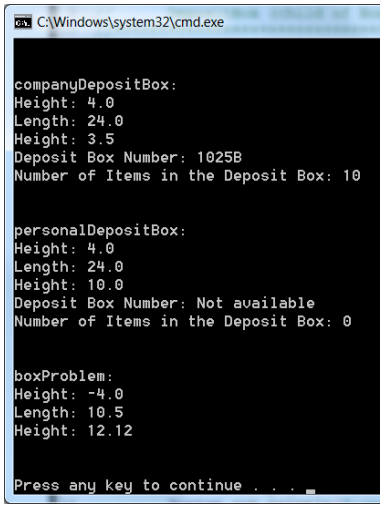
```
public NewBox()
{     height = -4.0f;
      length = 10.5f;
      width = 12.12f;    }
```

f. Open the **BoxDriver** class. Add the following **NewBox** instantiation on line 28:

```
NewBox boxProblem = new NewBox();
```

Add the following statement on line 38 to display the data from the **boxProblem** object:

```
System.out.println("boxProblem: \n" + boxProblem + "\n");
```



Compile and run **BoxDriver**. Notice the output for the **boxProblem** object (the last set of data on the right):

The child class, **NewBox**, was coded to directly update data in the parent class, **Box**. This code was even able to assign an invalid negative value (-4.0) to *height*.

In this example, through inheritance a new child object was created and was able to read, modify, and inject invalid data into an object from the parent class. The **protected** modifier on the data members in the parent class permitted this.

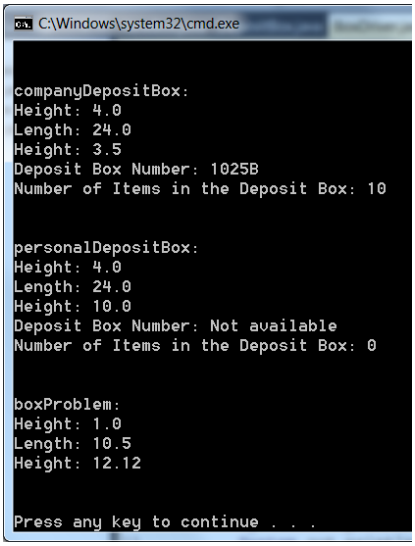
There may be an occasional situation when it is desirable for a child class to have direct access to a parent's data. If so, data validation should be performed on new values before they are used to update inherited data. However, there are many circumstances where a child class should not be allowed to directly update a parent's data and by doing so introduces security vulnerabilities, such as the one just illustrated.

g. In the **Box** class, change the access modifier on the three instance variables (*height*, *length*, and *width*) from ***protected*** to ***private***. Compile **Box**. Run **BoxDriver** again.

You will receive an *IllegalAccessError* exception due to the attempt to directly access a private field. Access was denied to the ***private*** data members (*height*, *length*, and *width*) in **Box** from **newBox**. This is one way of securing direct access to an object's data from a child class.

Limit the use of the protected access modifier on data members to situations when there is a compelling reason to do so. As a general rule, consider using the private access modifier for data members. Provide access to the data, if needed by a subclass, through the parent's protected accessors and/or mutators.

4. Inheritance and the protected access modifier with METHODS :



a. Change the **NewBox** constructor to the following so that **NewBox** is now using mutators to update data members in a **Box** object:

```
public NewBox()
{     setHeight(-4.0f);
      setLength(10.5f);
      setWidth(12.12f);    }
```

Compile **NewBox**. Run **BoxDriver** again. You will no longer receive an exception because you are using **public** mutators from **NewBox**. Sample output is shown to the right. Notice the value for *height* under **boxProblem** (shown on the right). The mutator for *height* performs input validation on its argument

and resets *height* to 1.0 if an invalid value is passed to the mutator method. The developer could just as easily have chosen to not change height at all if the input is invalid. The point is the mutator did not allow the data member to be updated with invalid data. **NewBox, in its present state,** must use the mutators in **Box** to modify the parent class data.

When designing classes, make sure the methods you create perform input validation on method parameters.

b. Attackers exploit vulnerabilities found in inheritance class structures. For example, review the methods in the **Box** class. Every method in **Box** has ***public*** access meaning any class can use these methods. Any class can use the mutators to modify an object's data. Any class can use the accessors to read the object's data.

Consider the mutators in **Box**. Once a box is instantiated, does it make sense to allow the dimensions of the box to be modified? If not, it would be more secure to delete the mutators in **Box** to prevent unwanted modifications to its data members.

c. For the purposes of this example and to later demonstrate issues with overriding methods, we will allow child classes to modify the dimensions of a box.

Change the access modifier for the mutators in **Box** from ***public*** to ***protected*** as shown below:

```
protected void setHeight(float height)
{     if (height > 0) this.height = height; }

protected void setLength(float length)
{     if (length > 0) this.length = length; }

protected void setWidth(float width)
{     if (width > 0) this.width = width; }
```

Compile **Box** and run **BoxDriver** to insure you have not introduced an error in the classes.

5. Impact of the final non-access modifier on inheritance:

a. Add the following method to the NewBox class toward the end of the class in the "other methods" section on line 3:

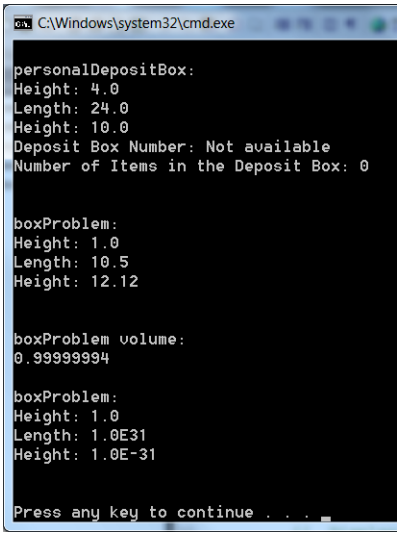
```
public float getVolume()
{     setHeight(-3);
      setLength(99999999999999999999999999999.0f);
      setWidth(getHeight()/getLength());
      return (getHeight() * getLength() * getWidth());   }
```

This code overrides the **getVolume( )** methods in the **Box** class. While it still returns the volume of the box, notice how it also uses ***protected*** mutator methods to modify the data members. Compile the **NewBox** class.

Add the following code to the **BoxDriver** class on line 39:

```
System.out.println("boxProblem volume: \n" +
boxProblem.getVolume() + "\n");
System.out.println("boxProblem: \n" + boxProblem + "\n");
```

Compile and run **BoxDriver**. Your output will be similar to what is illustrated to the right.



Look at the **boxProblem** *height*, *length*, *width*, and *volume*. The *height*, *length*, and *width* values were modified using the mutators and the *volume* was calculated using the new values.

The mutator methods' access modifiers in **Box** are ***protected*** and the **getVolume**() method is ***public***.

Using similar techniques, attackers gain access to an object's data by creating a new child class which overrides a parent method. Initially, you may contemplate changing the access modifier on the **getVolume**() method to ***protected*** in **Box**. Try it. Compile the **Box** class and rerun the **BoxDriver** class.

The same results are produced! Nothing was gained by changing the **getVolume**() access modifier to ***protected***. A ***protected*** method allows subclasses to use the method while it restricts non-subclasses from using it. For this example, external classes may need to know the volume of a box in which case this would not be a desirable approach.

In this specific example as presently defined, any child class of **Box** can override its methods and thus presents security vulnerabilities. What can be done?

b. Java has access modifiers and non-access modifiers. The ***final*** non-access modifier can help secure data and methods from accidental or intentional modification.

- Using ***final*** on a variable prevents the variable's value from being changed and thus creates a constant.
- Using ***final*** on a method header prevents a method from being overridden by a subclass.
- Using ***final*** on a class header prevents a class from being subclassed (restricts inheritance).

Update the **Box** class by modifying the **getVolume**() method header to the following:

```
public final float getVolume()
```

NOTE: ***protected*** was changed to **public** and **final** was added.

Compile **Box**. Compile **NewBox**. You will now receive a *VerifyError* preventing **NewBox** from overriding the **getVolume**() method.

Using the ***final*** modifier will prevent subclasses from overriding methods in the superclass. This is one effective technique to prevent attackers from extending a class, overriding methods, and accessing and/or modifying class data.

When designing an inheritance class structure, restrict overriding methods unless there is a compelling reason to do so. For subclasses at the lowest level in the hierarchy, restrict further subclasses unless there is a compelling reason for doing so (make the class final).

c. Consider the other methods in the **Box** class. Are there any methods which are not needed? For example, do you really need the mutators? Once a box is created, would you need to modify its dimensions? Or would you just instantiate another box with the new dimensions?

For this example, delete the three mutator methods named **setHeight**(), **setLength**(), and **setWidth**() in the **Box** class.

d. The accessors are defined as ***public*** so that any class using the **Box** class (or its child classes) can retrieve the dimensions of the box. However, you would not want a child class to override these methods and add malicious code to the method. Add the **final** modifier to the three accessor method headers **getHeight**(), **getLength**(),and **getWidth**() in the **Box** class.

Compile **Box**.
Delete the two methods in the **NewBox** class and compile.
Compile **BoxDriver** and run the application.
Everything should run fine now and the **Box** class should be more secure.

To make the entire class structure more secure, you should consider similar changes to the other classes in the structure.

LABORATORY ASSIGNMENT:

1. Accompanying this assignment is a compressed file named **InheritanceExample** which contains the following files:
   - Employee.java – Parent class
   - EmployeePayInfo – Child class of Employee which contains sensitive payroll information
   - HourlyEmployee – Child class of EmployeePayInfo
   - SalariedEmployee – Child class of EmployeePayInfo
   - EmployeePayroll – A driver class which runs and displays payroll information for all employees

   Save the ZIP file on your computer, unzip, and open in the files in the IDE/editor that you use to compile and run Java.

2. Compile all of the files. Run EmployeePayroll. Study the classes to make sure you understand this inheritance structure. Consider the security vulnerabilities in these Java classes. Think about new child classes that could be created from this structure. Think about input validation. Think about sensitive data and encryption needs.

3. Print each of the Java files. Complete the following checklist and actions for **Employee**, **EmployeePayInfo**, **HourlyEmployee**, and **SalariedEmployee** (using your printouts):

Secure Coding Vulnerabilities: Improper Inheritance Structure

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, protected, no access** modifier.

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 **V2** beside the method header of each method that returns the value of a data member. These methods are often called accessors or getters.
- ☐ Write **V3** beside the method header of each method that modifies the value of data members. These methods are often called mutators or setters.
- ☐ Write **V4** beside the method header of each method not using the **final** modifier. These methods can be overridden in child classes.
- ☐ Mark **V5** beside the class header if this is class at the "bottom" of the class hierarchy (currently has no child classes).
- ☐ Write **P** beside the method header of each method (including constructors) that uses a parameter list. All parameters should be validated to ensure they fall within the bounds of the method's intended purpose. For each parameter, write **V5** over the argument first appearance in code.

Eliminate vulnerabilities where feasible:

- ☐ For data members 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. **Mark any changes you made on your program listing.**
- ☐ For methods marked with **V2**: Review the instance variable being accessed. Seriously consider whether or not an accessor (method which returns a data member's value) is needed for this data item. Secure coding guidelines recommend avoiding accessors when possible. **Strike-through the method header for each accessor method that you recommend deleting from the class you are reviewing.** If there are instance variables which you believe should have an accessor, further consider the sensitivity of that data item. **Write E (for encryption) beside the method header for any accessor that you recommend using encrypting before returning a data member's value to another class.**
- ☐ For methods marked with **V3**: Review each instance variable and its associated mutator(s) (methods which modify the value of an instance variable). Seriously consider whether or not each mutator is needed. Do not include a mutator for each instance variable unless there is a compelling reason to do so. **Strike-through the method header for each mutator method that you recommend deleting from the class you are reviewing.**
- ☐ For each method (including constructors) marked with a **P**: Review the method's code for each use of the parameter to ensure the parameters value is validated before it is used in further statements. 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.

Secure Coding – Extensibility through Inheritance in Java

DISCUSSION QUESTIONS:

1. In general terms, describe the security vulnerabilities you found in the following classes:
    - Employee
    - EmployeePayInfo
    - HourlyEmployee
    - SalariedEmployee
2. Describe any changes you recommend to the modifiers (access modifiers and the final modifier) for **data members** for:
    - Employee
    - EmployeePayInfo
    - HourlyEmployee
    - SalariedEmployee
3. Describe any changes you recommend to the modifiers (access modifiers and the final modifier) for **methods** for:
    - Employee
    - EmployeePayInfo
    - HourlyEmployee
    - SalariedEmployee
4. Describe any changes you recommend to the modifiers (access modifiers and the final modifier) for **the class headers for:**
    - Employee
    - EmployeePayInfo
    - HourlyEmployee
    - SalariedEmployee
5. Describe **input validation** you suggest for methods in (be specific):
    - Employee
    - EmployeePayInfo
    - HourlyEmployee
    - SalariedEmployee
6. Describe any encryption suggestions you have for the following classes (be specific):
    - Employee
    - EmployeePayInfo
    - HourlyEmployee
    - SalariedEmployee
7. Describe any methods or data you recommend removing (with an explanation) in the following classes:
    - Employee
    - EmployeePayInfo
    - HourlyEmployee
    - SalariedEmployee

DELIVERABLES:

- Submit the answers to the Discussion Questions as directed by your instructor.
- Modify the classes per your suggestions, zip the 5 classes together and submit the compressed file as directed by your instructor.