

Encapsulation

COMP 401, Fall 2014

Lecture 06

9/4/2014

Picking up from last time...

- Abstraction
 - Defining a class to represent an abstraction
 - Instance fields hold object state
 - Instance methods define functions / procedures associated with the abstraction

Motivating Encapsulation

- Consider lec6.v1
- What's the danger?

Principle of Encapsulation

- Do not expose the internal state of an object directly.
 - Protects object fields from being put into an inconsistent or erroneous state.
 - Avoids situation in which external code is dependent on this specific implementation.
 - Or said another way: allows for the implementation of an abstraction to be improved/changed without breaking other code.
- Separate “exposed” behavior from “internal” behavior
 - Exposed behavior
 - Procedures / functions other objects / code expected to interact with.
 - Internal behavior
 - Procedures / functions defined only for use by methods that are part of the class.

Encapsulation In Practice

Part 1: Do Not Expose Internal State

- Make all fields *private*
 - Amend field declaration with “private” access modifier.
- Provide *public* methods that retrieve and/or alter properties
 - Methods that retrieves a property is called a “getter”.
 - Methods that set a property is called a “setter”
- Benefits
 - Can support “read-only” fields by NOT providing a setter
 - Setter can validate new value to prevent misuse or illegal values.
 - Can define derived or complex properties that are actually related to multiple field values.

JavaBeans Conventions

- JavaBeans
 - Software engineering framework
 - Associated tools
 - Relies on code following certain conventions
 - In particular, getters and setters for object properties.
- Given type T and property P:
 - Signature of a getter:
`public T getP()`
 - Signature of a setter:
`public void setP(T value)`

lec6.v2

- Provides getters for x and y values of a Point, but not setters.
 - Ensures Point is immutable
- Provides getters and setters for point of a Triangle
- Notice effect on original code in main method in Lec6Ex1.java

Setter Validation

- Setters should validate their values if possible.
 - One of the advantages of providing access to properties only through methods.
- Illegal / improper values should cause a runtime exception like this:

```
throw new RuntimeException("Explanation string");
```


lec6.v3

- Adds *equals* method to Point for comparison.
- setA(), setB(), and setC() in Triangle validate by...
 - making sure that points are distinct
 - checking for co-linearity
- Added area() method
- Added check_colinearity() method
 - Notice that I've chosen a specific precision for the check based on area.

Derived Properties

- Property that is a combination or transformation of object state fields.
 - Can you recognize two of these already in Triangle?
- Same principle for getters and setters applies here.
 - If using JavaBeans conventions, name methods with proper form and signature.
 - Read-only properties should not have a setter.
 - Setters should validate if necessary.

lec6.v4

- Changed `area()` and `perimeter()` to `getArea()` and `getPerimeter()` to follow JavaBeans conventions.
 - What about individual side lengths?
 - Could have done the same, but didn't to make another point later on.
- Created `getPoints()` and `setPoints()` as derived properties for dealing with all three points at once as an array.

Using Fields Internally

- Marking a field as “private” prevents access from code outside of the class.
 - But notice that there is no restriction to access private fields between different instances.
 - Look at *distanceTo()* and *equals()* methods in Point
- Does this violate principle of encapsulation?
 - Gray area
 - Could argue no since code is within the class.
 - Could argue yes since access to other point’s state is outside the context of the *this* reference.
 - My advice
 - Always safe to use exposed getter / setter, so do so.
 - There are sometimes good reasons not to, but generally these are related to issues of performance and optimization.

lec6.v5

- Re-wrote distanceTo() and equals() using getters for x and y values

Encapsulation In Practice

Part 2: Separate Exposed Behavior

- Define an “interface” for all exposed behavior
 - In Java, an interface is like a contract.
 - Indicates that a certain set of public methods are available.
 - One or more classes can indicate that they implement the interface.
 - Name of interface can be used as a type name
 - Just like class names are used as type names.
 - Value of an interface type variable can be set to any object that is an instance of a class that implements the interface.

Interfaces in Java

- Like classes, should go in their own .java file
 - Should have same name as file
 - Body of interface is a just list of method signatures.
 - Implementing classes MUST declare these methods as *public*

- Form:

```
interface InterfaceName {  
    type method1(parameters);  
    type method2(parameters);  
    // etc...  
}
```

- Classes specify which interfaces they implement with “implements” modifier as in:

```
class ClassName implements InterfaceA, InterfaceB {
```

Interface Naming Conventions

- Interface name must be different from class names that implement the interface.
- Convention A
 - Start all interface names with “I” for interface.
 - For example: ITriangle, Ipoint
 - Class names can be anything that is not in this form.
- Convention B
 - Use generic abstraction name for interface.
 - Make class names descriptive of implementation
 - If no natural way to do this, simply append “Impl” to generic abstraction name to differentiate.
- Personally, I generally go with convention B.

lec6. v6

- Separates Point into an interface and an implementing class.
 - Notice that distanceTo() and equals() are part of behavior I want the abstraction to expose.
- Notice that main method uses variables with type Point (the interface name), but that the actual object is created as an instance of a specific class that implements the interface Point.
- Notice that Triangle only interacts with the methods specified in the Point interface.

Advantage of Encapsulation

- Can provide different implementations of the same behavior
 - lec6.v7
 - Create a new implementation of Point based on polar coordinates.

Exposed vs Internal Behavior

- Exposed behavior should be reflected in the interface(s) that a class implements
 - Recall that any method declared in an interface must be defined by an implementing class as a *public* method.
- Internal behavior should be hidden
 - Use *private* modifier on these methods to ensure that access only occurs within the class

lec6.v8

- Continued application of encapsulation principle to Triangle by...
 - ... defining Triangle as an interface
 - ... rewriting what used to be the class Triangle as the class PointTriangle that implements the interface
 - ... hiding internal behaviors as private methods