

Encapsulation

- **Data hiding**
- **No new functionality, but restrictions that help with software engineering**
 - Abstract data type (ADT)
 - Separate interface from implementation
- **Smalltalk: Implicit via simple rules (the Actors model)**
 - An object's data can only be modified by that object's methods
 - Methods can be called by anyone, but can use introspection to check who's calling
- **Simula, C++, Java: public/protected/private**
 - C++: declare "friend" classes/functions that have full access
 - Java:
 - Top-level classes available within package
 - Or everywhere if declared public
 - Inner classes obey rules for members
 - Members that are not public, protected, or private are available within package
- **Alternative: Common Lisp's package system**
 - No private methods or fields
 - Initial impression: No encapsulation, everything open
 - The symbols (class names, field names, function names, etc.) are organized into package (name spaces)
 - Names are not just strings, but objects (symbols)
 - Define which symbols are exported from a package (and thereby accessible from other packages)
 - A form of encapsulation that is orthogonal to the classes

Design pattern and language features

- **What are design patterns?**
 - Common ways of defining a set of classes (in Java, Smalltalk, or similar languages) for achieving some purpose.
 - The "Gang of Four" book (Gamma et al.) describes a number of them.
 - Design patterns were huge in the late 1990s
 - Idea: If you know the common patterns, you can recognize them in source code that you read. Way of understanding class hierarchies that would otherwise be very complex.
- **Language features**
 - We have seen that some other languages have many features that are not present in Java.
 - Some design patterns correspond to language features.
 - Ways of working around Java not having those features.
 - Become unnecessary in languages that have those features.
- **The "observer-observable" pattern (aka. the "listener" pattern)**
 - Example:

- Your code wants to know whenever someone clicks a button in the UI.
- The UI library defines an interface (e.g., `ButtonListener`). The interface typically has only one method (e.g., `buttonPressed`).
- You define a class that implements the `ButtonListener` interface.
- You pass an instance of that class to the UI library.
- When the button is clicked, the UI library calls the `buttonPressed` method on the instance you gave it.
- Unnecessary in languages that have first-order functions.
 - Just give the UI library a function it can call when someone clicks the button.
 - Anonymous functions (lambda) make this even more convenient.
- **The "visitor" pattern**
 - Very similar to "observer-observable"
 - Example:
 - You have a collection of students and want to print all the names.
 - The collection class defines a `CollectionVisitor` interface with a single method, "visit", that takes a single parameter of type `Student`.
 - You define a class that implements the `CollectionVisitor` interface. The "visit" method on this class prints the name of the student given as parameter.
 - You call the "foreach" method on the collection with an instance of your visitor class as a parameter.
 - The collection will call the "visit" method on your instance once for each student in the collection.
 - Unnecessary in languages that have first-order functions.
 - Just give the collection's foreach method a function. It will then call this function once for each student, with the student as a parameter.
 - Again, anonymous functions make this more convenient.
 - Just like the "map" function in Haskell.
- **The "adapter" pattern**
 - Example:
 - You are working on a new web interface to the Gold system.
 - There is an existing administrative system that you're not allowed to modify. It defines classes such as `Student`.
 - You wish that the `Student` class had a `renderHTML` method that generates a nice HTML table with all the student's information.
 - Subclassing it doesn't help
 - The existing system returns `Student` instances, and there's not way to tell it to use your subclass.
 - Write a new class that implements the `Student` interface
 - Member variable `s` is a reference to a `Student` object
 - Actual code for `renderHTML()`
 - All other methods just call the method on `s` that has the same signature
 - Use introspection to generate most of the methods automatically.

- Unnecessary in languages that have prototype-based object systems
 - Just add the new method to the object
- Unnecessary in languages that have methods outside objects
 - Can just write a renderHTML method that specializes on Student, even though you cannot modify the Student class
 - Very useful. (Have you ever wanted to add methods to String?)
- **The "decorator" pattern**
 - Example:
 - A GUI framework may have a Window class.
 - Want to add extra functionality: Vertical scroll bar
 - Can make a new subclass VerticalScrollBarWindow of Window.
 - Adds drawVerticalScrollBar() to the draw() method.
 - Want to add more functionality: borders
 - Could make a subclass BorderWindow of Window.
 - But what if I want both a vertical scroll bar and a border?
 - Would need another subclass: VerticalScrollBarAndBorderWindow
 - Becomes unmanagable as we add more functionality (horizontal scroll bar, etc.)
 - Using the decorator pattern:
 - interface Window
 - class SimpleWindow implements Window
 - abstract class WindowDecorator implements Window
 - member w: reference to the Window being decorated
 - class VerticalScrolBar extends WindowDecorator
 - draw() calls drawVerticalScrollBar() first, then w.draw()
 - class Border extends WindowDecorator
 - draw() calls drawBorder() first, then w.draw()
 - In main program:
 - Window w = new WindowDecorator(new VerticalScrollBar(new Window ()));
- Unnecessary in languages that have multiple inheritance
 - class Window, class VerticalScrollbar, class Border
 - class BrowserWindow inherits from all three classes
 - The latter two classes are called "mixins"
 - But how to call drawVerticalScrollbar() and drawBorder() from draw()
 - One option: Override draw() in BrowserWindow
 - Tedious and error-prone if the mixing provide much new functionality
 - Better option: "after-methods"
 - The mixins define "draw :after" methods. These are called after Window's draw()
 - Present in Common Lisp