



UW  BUSINESS SCHOOL

Web/Java Catchups

MSIS 531 – Spring 2006

Topics 2

- Architecture patterns
 - Common design patterns/blueprints for enterprise applications
- Additional Java content
 - Follow-on to previous Java discussion, covering topics we'll see in subsequent material

UW  BUSINESS SCHOOL

UW  BUSINESS SCHOOL

J2EE Design Patterns

MSIS 531 – Spring 2006

Overview

4

- Why use patterns?
 - Improve code reusability
 - Increase maintainability
 - Project Lifecycle cost savings
 - Common language for discussion
- Our focus will be on using patterns in the J2EE world, but they apply everywhere
- Usually grouped into presentation, business and data patterns

Client Tier: Intercepting Filter

5

- Issues:
 - Verify authentication
 - Check type of client
 - Is the session valid?
- Solution: implement one or more Intercepting filters
- We'll see this pattern used in MyPetstore to do authentication

Client Tier: Front Controller (MVC)

6

- Problems:
 - Multiple views (esp. in web applications)
 - Navigation logic embedded in views
 - Difficult to maintain view logic as application grows
 - Limited reuse across clients
- Front Controller acts as "traffic cop" for view requests
- View passes request to Controller
 - Controller determines business request, accesses model
 - Controller dispatches to the View.
 - View displays model in appropriate client

Client Tier: Composite View

7

- Problem:
 - Many common, recurring components in a view
 - Fragments may display differently depending on source object (example: employee, manager)
 - Look and feel needs to be easily updateable
- Can be implemented by JavaBeans, jsp actions or custom tag libraries
- Composite View decides which sub-views to include, while another pattern (View Helper) decides how to display a given sub-view
 - In MyPetstore, we'll use JSF + Struts Tiles



Business Tier: Session Façade

8

- Problem:
 - Need very loose coupling between client and ever changing business tier
 - Data tier may change (e.g. Oracle -> DB2)
 - Need flexibility to offer up business rules in different ways to multiple clients
- Solution: Session Façade
 - Can be a session bean or a plain old Java class
 - Takes care of intermediating between business, data tiers
 - Contains or can access all of the applicable business rules
 - When designing, typically the façade contains a series of common use cases



Business Tier: Service Locator

9

- Problem:
 - How to abstract the locations of things?
 - Databases are an easy example; consider messaging services, legacy applications, web services, etc.
- A service locator abstracts business services from their locations
- We saw an example with JNDI lookups to access MySQL from NetBeans
 - Easy (or at least clean) to drop in another database
- We'll see that Spring "wires together" services and their associated "users" (beans)



Business Tier: Dependency Injection

10

- Question: should we go looking for things (e.g. services) or have them come to us?
 - “Pull” vs. “Push”
 - Service Locator is a “pull” pattern
 - Dependency injection (also called “Inversion of Control”, or IOC) is a “push”
- When we talk about Spring “wiring things together”, what it’s really doing is injecting functionality into classes
- Any common functionality (logging, transactions, security, etc.) is fair game for this approach
 - It is harder to grasp but has some significant benefits



Data Tier: Data Access Objects (DAO)

11

- Problem:
 - Abstract changes to database
 - Need flexibility to use JDBC, Hibernate, etc.
 - May need separate data stores or storage types (XML, file-based, http)
- Data Access Objects abstract and encapsulate all access to a data source
 - For MyPetstore , the DAO is a Java interface (e.g. CatalogDao) implemented by a class (CatalogDaoImpl)





Java – Additional Topics

MSIS 531 – Spring 2006

Method Overloading

13

- Multiple methods with same names, different arguments
 - method signatures differ
- Example of polymorphism: useful results when different argument types used
- If more than one possible signature, most specific type is chosen
- Choice of which version to use is made at compile time



Events

14

- An event is an action that has occurred
 - User has clicked right mouse button
 - Floppy disk is full
- We can choose which events we respond to
- For HelloWorld2, we care about mouse events
 - We do so by “listening for them”
 - We register our intent to do so by calling `addMouseListener()` in the class constructor
 - Then call `mouseDragged()` to respond
 - We must call `mouseMoved()` as well
- How does this work?



Interfaces

15

- Interfaces allow us to “act on objects based on capability, rather than type”
- A contract of sorts, which is strictly enforced
 - In our case, to handle mouse movement
- We do the following:
 1. Implement the `MouseListener` interface
 - A class may implement many interfaces, but extend only one other class
 2. Call `addMouseListener()` in our constructor
 - We pass a copy of ourselves to this method (??)
 - Our “mouse ears” are closed without it
 3. Provide `mouseDragged()` and `mouseMoved()` methods
 - `mouseMoved()` is empty, but must be present



The synchronized keyword

16

- Use of synchronized allows us to guarantee that while we're accessing a method, no one else can
 - What if more than one button on a Swing form, and two of them executed a method at exactly the same time?
 - All sorts of bad things might happen (e.g. array bounds problem) and Java doesn't allow that
 - Use of synchronized methods guarantees this won't happen
- This topic is related to the subject of threads, which are important but out of scope for the class

Inheritance and Subclassing

17

- Class inherits variables, methods from superclass
 - Used as though defined in the class itself
- Java classes can only inherit from one superclass
 - Single inheritance
 - Can implement multiple interfaces: more on this shortly
 - Subclass can be used wherever parent can
- All superclass members except those designated as private are inherited

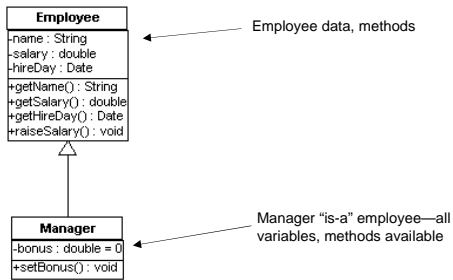
Overriding Methods

18

- Recall: overloaded methods -> same method name, different argument pattern
- Overridden method -> same method name and argument pattern as parent
 - Change in implementation from parent
- JVM decides which method to use at runtime
 - Most derived method (furthest down the inheritance tree) is used

Example: Employee/Manager

19



this And super References

20

- Refer to current object with this, parent object with super
- Use to:
 - Pass a reference to the current object
 - Refer to shadowed superclass members
 - Combine behaviors of subclass with superclass
 - Call superclass constructor
 - No-argument constructor called implicitly
 - Use superclass constructor when you want to call a non-default constructor

Casting Object References

21

- Tell compiler to change the apparent type of an object
 - Casts checked for legality at compile time and runtime
 - No change in actual type
- Explicit cast usually used to narrow (downcast) the type of reference
- Implicit cast occurs when widening the type of a reference
- Use instanceof() to verify class type, avoid ClassCastException

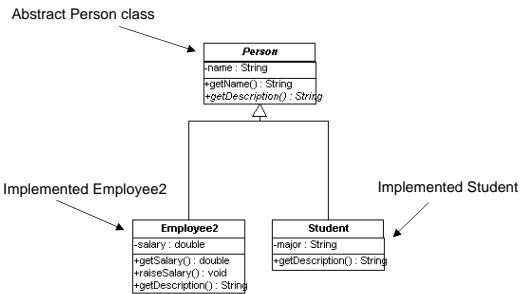
Abstract Methods/Classes

22

- Abstract class -> prototype
 - Outline (framework) for inheriting classes
 - If any abstract methods, class must be abstract
 - Abstract classes may include non-abstract data, methods
- Abstract method -> no body, just semicolon
- If any abstract methods, class must be abstract
 - Subclass extending an abstract class must implement abstract methods, or it must be abstract too.

Example: Employee/Student

23



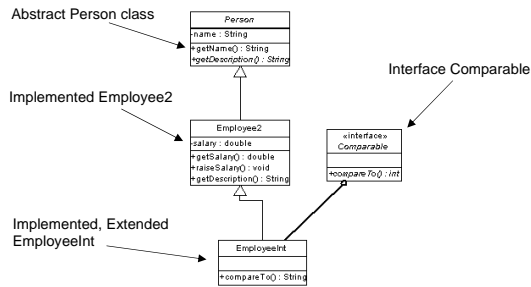
Interfaces

24

- Another approach to prototyping
 - Methods without implementation
 - What a class can do
 - If class implements the interface, it must implement all interface methods
- Defines another data type
- Cannot be instantiated, only implemented
 - Subinterfaces (extended interfaces) allowed
- No variables allowed except static final variables (constants)
- Interfaces cut across the class hierarchy

Example: Employee w/Interface

25



Interfaces vs. Abstract Classes

26

- To extend an abstract class, must be a subclass
 - And remember, no multiple inheritance
 - Interfaces cut across the class hierarchy
- Changes to abstract classes can break implementations
- Abstract classes allow concrete data members, methods
 - Interfaces can provide skeletal implementation wrapper classes
- Both methods are tools in the toolbox

Packages

27

- **Package:** group of related classes, interfaces
 - Organization
 - Additional level of scoping
- Designate a package as the first statement in a compilation unit
 - Unit of programming work; a .java file
- Package names organized hierarchically
 - Used to construct a path to local files
- No such thing as a subpackage—relationships in a package hierarchy are informal

Class Visibility

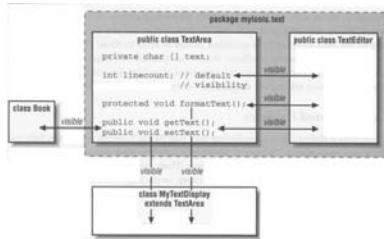
28

- By default, classes visible only within their enclosing package
 - Override with **public** declaration:

```
public class MyClass { . . . }
```
 - One public class per compilation unit (.java file must be named for that class)
 - Other (private) classes are hidden: the public classes provide the **interface** for the package
- Recall the import statement allows reference to external packages using simple names
 - Name conflicts resolved by using the hierarchy names

Visibility Modifiers

29



- **Public:** visible to all classes
- **Protected:** classes in the package and subclasses inside or outside package
- **none (default):** classes in the package
- **private:** none
