CS 494

Adv. SW Design and Development

A Tasting….

- **Course 1: Design patterns: Intro, example**
- **Course 2: Inheritance, Interfaces, OO Design**

Reading Assignment

- **Read for understanding (code if it helps)**
  - Basic Java program structure
    - Classes and files; importing packages
    - main() method
  - Types in Java
    - Primitive vs. class; class hierarchy; class Object
    - Boxing
    - References
    - Defining classes
  - In *Just Java 2*: Chapters 1-5

Tastings: Course 1

- **Idioms, Patterns, Frameworks**

Idioms, Patterns, Frameworks

- **Idiom: a small language-specific pattern or technique**
  - A more primitive building block
- **Design pattern: a description of a problem that reoccurs and an outline of an approach to solving that problem**
  - Generally domain, language independent
  - Also, analysis patterns
- **Framework:**
  - A partially completed design that can be extended to solve a problem in a domain
    - Horizontal vs. vertical
  - Example: Microsoft's MFC for Windows apps using C++

Examples of C++ Idioms

- **Use of an Init() function in constructors**
  - If there are many constructors, make each one call a private function Init()
    - Init() guarantees all possible attributes are initialized
    - Initialization code in one place despite multiple constructors
- **Don't do real work in a constructor**
  - Define an Open() member function
    - Constructors just do initialization
    - Open() called immediately after construction
  - Constructors can't return errors
    - They can throw exceptions

Design Patterns: Essential Elements

- **Pattern name**
  - A vocabulary of patterns is beneficial
- **Problem**
  - When to apply the pattern, what context.
  - How to represent, organize components
  - Conditions to be met before using
- **Solution**
  - Design elements: relationships, responsibilities, collaborations
  - A template for a solution that you implement
- **Consequences**
  - Results and trade-offs that result from using the pattern
  - Needed to evaluate design alternatives

Patterns Are (and Aren't)

- **Name and description of a proven solution to a problem**
- **Documentation of a design decision**
- **They're not:**
  - Reusable code, class libraries, etc. (At a higher level)
  - Do not require complex implementations
  - Always the best solution to a given situation
  - Simply "a good thing to do"

Example 1: Singleton Pattern

- **Global variables are bad!**
  - Why?
- **We are tempted to use global variables?**
  - Why?

Example 1: Singleton Pattern

- **Context: Only one instance of a class is created. Everything in the system that needs this class interacts with that one object.**
- **Controlling access: Make this instance accessible to all clients**
- **Solution:**
  - The class has a static variable called *theInstance* (etc)
  - The constructor is made private (or protected)
  - Clients call a public operation *getInstance()* that returns the one instance
    - This may construct the instance the very first time or be given an initializer

Singleton: Java implementation

```
public class MySingleton {
  private static theInstance =
       new MySingleton();
  private MySingleton() { // constructor
    …
  }
  public static MySingleton getInstance() {
       return theInstance;
  }
}
```


Static Factory Methods

- **Singleton pattern uses a *static factory method***
  - Factory: something that creates an instance
- **Advantages over a public constructor**
  - They have names. Example:
    BigInteger(int, int, random) vs.
    BigInteger.probablePrime()
  - Might need more than one constructor with same/similar signatures
  - Can return objects of a subtype (if needed)
- **Wrapper class example:**
  Double d1 = Double .valueOf("3.14");
  Double d2 = new Double ("3.14");
- **More info: Bloch's *Effective Java***

Tastings: Course 2

- **Interfaces and Collections in Java**

Java Interfaces

- **Note that the word "interface"**
  - Is a specific term for a language construct
  - Is not the general word for "communication boundary"
  - Is also a term used in UML (but not in C++)

Why Use Inheritance?

- **Why inherit? Create a class that…**
  1. Makes sense in problem domain
  2. Locates common implementation in superclass
  3. Defines a shared API (methods) so we can…
  4. Use polymorphism
     - Define a reference or parameter in terms of the superclass
- **If just last two, then use Java interface**
  - No shared implementation
  - You commit that part of what defines a class is that it meets a particular API
  - We can write methods etc. that operate on objects of any class that meets or supports that interface

Two Types of Inheritance

- **How can inheritance support reuse?**
- ***Implementation Inheritance***
  - A subclass reuses some implementation from an ancestor
  - In Java, keyword *extends*
- ***Interface Inheritance***
  - A "subclass" shares the interface with an "ancestor"
  - In Java, keyword *implements*
  - I.e. this class will support this set of methods

Interfaces and Abstract Classes

- **Abstract classes:**
  - Cannot create any instances
- **Prefer Java interfaces over abstract classes!**
  - Existing classes can add an interface
  - Better support for *mix-in classes*
    - E.g. *Comparable* interface -- supports compare()
  - Do not need a hierarchical framework
  - Composition preferred over inheritance
    - E.g. *wrapper classes*
- **But, abstract classes have some implementation**
  - Skeletal implementation classes, e.g. AbstractCollection
- **Disadvantage: once released, a public interface shouldn't be updated**

Interfaces in Other Languages

- **A modeling method in UML**
- **Interfaces in C++**
  - All methods are pure virtual
  - No data members
  - Use multiple inheritance

Collections in Java

- **ADT: more than one implementation meets same interface, models same data**
- **In Java, separate interface from implementation**
- **We'll illustrate with "fake" Java example:**
  - Queue interface
  - Two implementations

Defining an Interface

- **Java code:**

```
interface Queue {
    void add (Object obj);
    Object remove();
    int size();
}
```

- **Nothing about implementation here!**
  - methods and no fields

Using Objects by Interface

- **Say we had two implementations:**

```
Queue q1 = new CircularArrayQueue(100);
  or
Queue q1 = new LinkedListQueue();

q1.add( new Widget() );
Queue q3 = new …
Queue q2 = mergeQueue(q2, q3);
```


Implementing an Interface

- **Example:**

```
class CircularArrayQueue implements Queue
{   CircularArrayQueue(int capacity) {…}
    public void add(Object o) {…}
    public Object remove() {…}
    public int size() {…}

    private Object[] elements;
    private int head;
    private int tail;
}
```


Implementing an Interface (2)

- **Implementation for LinkedListQueue similar**
- **Question: How to handle errors?**
  - Array version is bounded. add() when full?
  - Throw an exception, perhaps
  - Not an issue for linked list version, though

Real Collection Interfaces in Java

- **All collections meet Collection interface:**

```
boolean add(Object obj);
Iterator iterator();
int size();
boolean isEmpty();
boolean contains(Object obj);
boolean containsAll (Collection other);
…
```

- **See Java API documentation for all methods**

Iterator Interface

- **Three fundamental methods:**

```
Object next();
boolean hasNext();
void remove();
```

- **We use an iterator object returned by Collection.iterator() to visit or process items in the collection**
  - Don't really know or care how its implemented

Example Iterator Code

- **Traverse a collection of Widgets and get each object**

```
Iterator iter = c.iterator();
while ( iter.hasNext() ) {
    Object obj = iter.next();
    // or:  Widget w = (Widget) iter.next();
    // do something with obj or w
}
```

- **Note the cast!**

New in Java 1.5 – generics and foreach

- **Java 1.5 has generics, and…**
- **A foreach statement simplifies the previous idiom**

```
Collection<Double> c = new HashSet<Double>;

for (Double d : c )
   System.out.println("c has " + d );
```


Methods Defined by Other IF Methods

- **Some collection methods can be defined "abstractly"**

```
public boolean addAll (Collection from) {
    Iterator iterFrom = from.iterator();
    boolean modified = false;
    while ( iterFrom.hasNext() )
       if ( add(iterFrom.next()) ) modified = true;
    return modified;
}
```


Collections and Abstract Classes

- **To define a new Collection, one must implement all methods -- a pain!**
- **Better: define a *skeletal implementation class***
  - Leaves primitives undefined: add(), iterator()
  - Defines other methods in terms of those
- **Concrete collection class inherits from skeletal class**
  - Defines "primitives"
  - Overrides any methods it chooses too
- **Java library: AbstractCollection**
  - Implements Collection IF
  - You inherit from it to roll your own Collection

Java's Concrete Collection Classes

- **Vector is like array but grows dynamically**
  - Insertion or deletion in the middle expensive
- **LinkedList class**
  - Doubly-linked
  - Ordered collection
    - add() inserts at end of list
    - How do we add in middle?

ListIterator Interface

- **ListIterator (sub)interface extends Iterator**

```
// add element before iterator position
void add(Object o); // on ListIterator object
Object previous();
boolean hasPrevious();
void set(Object o);
int nextIndex();   and   int previousIndex();
```

- **Also a factory that takes an initial position. E.g.**
  ListIterator backIter = c.listIterator(c.size());
- **Concurrent modification by two iterators?**
  - ListIterator checks for this

ArrayList Collection

- **Like a Vector but implements the List IF**
  - Stores an array internally
  - Access to element by index is constant, O(1)
  - Element insertion/removal is W(n) = O(n)
  - Expansion automatic (but with time costs)
- **Supports get(index) and set(index)**
  - So does LinkedList but inefficient
  - Note: in Vector, elementAt() and setElementAt()
- **Supports synchronization**
  - Vector does not.

List Interface

- **All methods from Collection interface, plus…**

- **int indexOf(Object elem) -- not found? -1**
- **int lastIndexOf(Object elem)**
- **Object remove(int index)**
- **Object set(int index, Object elem)**
- **Object clone() -- makes a shallow copy**
- **List subList(int fromIndex, int toIndex)**

Other ArrayList Methods

- **Constructors:**
  - default; given initial capacity; given Collection
- **Capacity management:**
  - `void ensureCapacity();`
    `void trimToSize();`
- **Collection to array:**
  - `Object[] toArray();`

Map Interface and Map Classes

- **Map interface defines generic map collection methods**
- **Two implementations**
  - HashMap: classic hash-table, not sorted
  - TreeMap: sorted, uses red-black trees
- **Defines three *collection views*, which allow a map's contents to be viewed as one of:**
  - set of keys; collection of values; or set of key-value mappings.
- **Map's order: how the iterators return their elements**

HashMap methods

- **Constructors:**
  - initial capacity, optionally a *load factor*
- **Object put(Object key, Object value)**
- **Object get(Object key)**
- **boolean containsKey(Object key)**
- **boolean containsValue(Object value)**
- **Object remove(Object key)**

- **Notes: key pass separately from Object**
- **Also: key must have good hashCode() defined**