

# Design Patterns

CSE 331 – Section 9

11/28/12

Slides by Kellen Donohue  
with material from Hal Perkins, Mike Ernst

# Course Logistics

- hw4, hw6 returned
- hw7 being graded
- hw8 due with 2 late days tonight
- hw9 due Wednesday
  
- Final Exam 8:30 AM on Dec 10
  - Review in section next week & Sunday before

# Anonymous Comparators

- Recall anonymous inner classes:

```
button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        model.doSomething()
    }
});
```

- Recall comparators:

```
class PointYCoordComparator implements Comparator<Point> {
    public int compare(Point p1, Point p2) {
        return p1.Y - p2.Y;
    }
}
```

# Anonymous Comparators

- Combine them to get anonymous comparators:

```
Collections.sort(list, new Comparator<Point>() {  
    @Override  
    public int compare(Point p1, Point p2) {  
        return p1.Y - p2.Y;  
    }  
});
```

- The code for sorting is inline, so you don't have to look anywhere else

# What is a design pattern?

- A standard solution to a common programming problem
- A technique for making code more flexible
- Shorthand for describing program design
  - a description of connections among program components

# Outline

- Creational patterns (constructing objects)
- Structural patterns (controlling heap layout)
- Behavioral patterns (affecting object semantics)
  - covered in lecture tomorrow

# Creational patterns

- Constructors in Java are inflexible
  - Always return a fresh new object, never re-use one
  - Can't return a subtype of the class they belong to
- Sharing
  - Singleton
  - Interning
  - Flyweight
- Factories
  - Factory method
  - Factory object
  - Prototype
  - Dependency injection
- Builder

# Sharing

- Java constructors always return a new object, never a pre-existing object
- **Singleton**: only one object exists at runtime
  - Factory method returns the same object every time
- **Interning**: only one object with a particular (abstract) value exists at runtime
  - Factory method returns an existing object, not a new one
- Flyweight: separate intrinsic and extrinsic state, represent them separately, and intern the intrinsic state
  - Implicit representation uses no space
  - Not as common / important – not covered today



# Singleton

- For a class only one object of that class can ever exist
- Variety of possible implementations – this one creates instance lazily

```
class Bank {  
    private static Bank INSTANCE;  
  
    // private constructor  
    private Bank() { ... }  
  
    // factory method  
    public static Bank getInstance() {  
        if (INSTANCE == null) {  
            INSTANCE = new Bank();  
        }  
        return INSTANCE;  
    }  
    ...  
}
```

```
Bank b = new Bank();
```

```
Bank b = Bank.getInstance();
```

# Singleton Example -- HttpRequest

- HttpRequest class handles authentication, don't want to have to redo this for each HTTP request, so create a singleton that everyone uses.
- <http://code.google.com/p/lab-specimen-transport-system/source/browse/android/src/edu/washington/cs/labspecimentransport/data/HttpRequest.java> (line 65)

```
private static class HttpRequestHolder {
    public static final HttpRequest INSTANCE = new HttpRequest();
}

/* Singleton - Don't instantiate */
private HttpRequest() { }

public static HttpRequest getInstance() {
    return HttpRequestHolder.INSTANCE;
}
```

# Singleton Example -- Comparator

- Comparators have no state

```
public class LengthComparator implements Comparator<String> {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

```
}
```

# Singleton Example -- Comparator

- Comparators have no state

```
public class LengthComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }

    private LengthComparator() {}

    private static LengthComparator comp = null;
    public static LengthComparator getInstance() {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }
}
```

# Singleton Variant -- Random

- You'll often want to use a random number generator in many places, but you only want one instance of `java.util.Random`
- Use a variation of singleton to hold the `Random` instance

Example from HW3:

```
public String getGreeting() {  
    Random randomGenerator = new Random();  
    String[] greetings = new String[5];  
    ...
```

# Singleton Variant -- Random

- You'll often want to use a random number generator in many places, but you only want one instance of `java.util.Random`
- Use a variation of singleton to hold the `Random` instance

Example from HW3:

```
public String getGreeting() {
```

```
    Random randomGenerator = new Random();
```

```
    String[] greetings = new String[5];
```

```
    ...
```

```
public class RandomSingleton() {
```

```
    private static final Random INSTANCE = new Random();
```

```
    private RandomSingleton() {}
```

```
    public Random getInstance() { return INSTANCE; }
```

```
}
```

# Singleton Variant -- Random

- You'll often want to use a random number generator in many places, but you only want one instance of `java.util.Random`
- Use a variation of singleton to hold the `Random` instance

Example from w/ Singleton HW3:

```
public String getGreeting() {  
    Random randomGenerator = RandomSingleton.getInstance();  
    String[] greetings = new String[5];  
    ...  
}
```

```
public class RandomSingleton() {  
    private static final Random INSTANCE = new Random();  
    private RandomSingleton() {}  
    public Random getInstance() { return INSTANCE; }  
}
```

# Singleton Variant -- Random

- You'll often want to use a random number generator in many places, but you only want one instance of `java.util.Random`
- Use a variation of singleton to hold the `Random` instance

Example from w/ Singleton HW3:

```
public String getGreeting() {  
    Random randomGenerator = RandomSingleton.getInstance();  
    String[] greetings = new String[5];  
    ...  
}
```

```
public class RandomSingleton() {  
    private static final Random INSTANCE = new Random();  
    private RandomSingleton() {}  
    public Random getInstance() { return INSTANCE; }  
}
```

Different than Singleton. Can't force use of `RandomSingleton` like we could if we wrote `Random` ourselves, so don't depend on that.



# Interning

- Similar to Singleton, except instead of there just being one object per class, there's one object per abstract value of the class
- Saves memory by compacting multiple copies
- Requires the class being interned is immutable  
Why?

# Interning – Point Example

```
public class Point {  
    private int x, y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() { return x; }  
    public int getY() { return y; }  
  
    @Override  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
}
```

# Interning – Point Example

```
public class Point {
    private static Map<String, Point> instances =
        new HashMap<String, Point>();

    public static Point getInstance(int x, int y,
        String key = x + ", " + y;
        if (!instances.containsKey(key)) {
            instances.put(key, new Point(x, y));
        }
        return instances.get(key);
    }

    private final int x, y; // immutable

    private Point(int x, int y) {
        ...
    }
}
```

key has to be a unique representation of abstract value.

If our point was represented with  $r$  and  $\theta$ , we'd need to constrain them for use in the key. Else we'd have "5,  $\pi$ " and "5,  $3\pi$ " as different entries in our map even though they are the same abstract value

# Interning – Strings

- Java uses interning to implement String
- So when you have
  - String a = “neat”
  - String b = “neat”
  - These two strings refer to the same object
- You can concatenate two string literals to match another
  - String c = “n” + “eat”

# Interning – String Caveats

- Not all Strings can be interned
  - Ones you get from the user
  - Ones built by combining string variables
  - When you get a string from `StringBuilder`
- This is why you have to call `equals()` with strings but sometimes `==` will work

```
String a = "neat";  
Scanner console = new Scanner(System.in);  
String b = console.next(); // user types "neat"  
if (a == b) { ...           // false
```

# Interning – String Caveats

- Not all Strings can be interned
  - Ones you get from the user
  - Ones built by combining string variables
  - When you get a string from `StringBuilder`
- This is why you have to call `equals()` with strings but sometimes `==` will work

```
String a = "neat";  
Scanner console = new Scanner(System.in);  
String b = "neat" // for debugging  
if (a == b) { ... // true
```

# Factories

- Factories solve the problem that Java constructors Can't return a subtype of the class they belong to
- Two factory patterns
  - Factory method (helper creates and returns objects)
  - Abstract factory
    - Abstract superclass defines what *can* be customized
    - Concrete subclass dose customization, returns appropriate subclass

# Factory Method

DateFormat class encapsulates knowledge about how to format dates and times as text

- Options: just date? just time? date+time? where in the world?
- Instead of passing all options to constructor, use factories.
- The subtype created doesn't need to be specified.

```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL,
    Locale.FRANCE);
Date today = new Date();
System.out.println(df1.format(today)); // "Jul 4, 1776"
System.out.println(df2.format(today)); // "10:15:00 AM"
System.out.println(df3.format(today)); // "jeudi 4 juillet 1776"
```



# Factory Method – Image Example

- Suppose we are making implementing the Swing method for loading an image onto the screen
- We need to handle different image formats, which all have very different logic to decode
- Common output image interface though

# Factory Method – Image Example Code

```
abstract class ImageReader {
    abstract Image displayImage();

    public ImageReader getImageReader(String str) {
        if (str.endsWith("gif")) {
            return new GifReader(str);
        } else if (str.endsWith("jpg")) {
            return new JpgReader(str);
        } else {
            ...
        }
    }
}

class GifReader extends ImageReader { GifReader(str) { this.img = new GifImage(); ... }}
class JpgReader extends ImageReader { JpgReader(str) { this.img = new JpgImage(); ... }}
```

## Factory Method – Image Example Alternative

- Instead of using a factory here you could imagine having different helper methods within ImageReader such as displayJpeg(), displayGif(), etc.
- The Factory method is advantageous because these methods could be very large and complex – makes more sense to have in their own class

# Abstract Factory

- You have a superclass that includes one or more abstract methods
- The superclass factory can be extended to provide different sub-factories
- Client gets one of the sub-factories, but doesn't care which

# Abstract Factory -- GUI Example

```
interface Button { void paint(); }
```

```
class WinButton implements  
    Button {  
    public void paint() {  
        System.out.println("I'm  
            a WinButton");  
    }  
}
```

```
class OSXButton implements  
    Button {  
    public void paint() {  
        System.out.println("I'm  
            an OSXButton");  
    }  
}
```

```
interface GUIFactory {  
    Button createButton();  
}
```

```
class WinFactory implements  
    GUIFactory {  
    public Button createButton()  
    { return new WinButton(); }  
}
```

```
class OSXFactory implements  
    GUIFactory {  
    public Button createButton()  
    { return new OSXButton(); }  
}
```

# Abstract Factory -- GUI Example

```
public class Application {  
    public static void main(String[] args) {  
        GUIFactory factory = createOSSpecificFactory();  
        Button button = factory.createButton();  
        button.paint();  
    }  
  
    public static GUIFactory createOsSpecificFactory() {  
        int sys = readFromConfigFile("OS_TYPE");  
        if (sys == 0) return new WinFactory();  
        else return new OSXFactory();  
    }  
}
```

# Abstract Factory -- GUI Example Explanation

- GUIFactory is the abstract factory
  - It's subclassed to be either a WinFactory or OSXFactory
  - Client doesn't care whether a WinFactory or OSXFactory is returned – they only want to create a button from the GUI Factory and print it

# Abstract Factory -- ImageFactory

- Now suppose we are loading many images, e.g. for a web server
- If we have spare memory let's cache images, otherwise just load them as normal
- Let's do this with an abstract factory



# Abstract Factory – ImageFactory Code

```
abstract class ImageFactory {
    public abstract Image loadImage(String fileName);
    public static ImageFactory getImageFactory(String filename) {
        if (/* we can cache */) return new CachingImageFactory();
        else return new StandardImageFactory();
    }
}

class StandardImageFactory extends ImageFactory {
    public Image loadImage(String filename) { ... }
}

class CachingImageFactory extends ImageFactory {
    public Image loadImage(String filename) { ... }
}

class ImageLoader {
    public static Image getImage(String filename) {
        ImageFactory imageFactory =
            ImageFactory.getImageFactory(filename);
        return imageFactory.loadImage();
    }
}
```

# Builder Pattern

- Builder is another variation on object construction
- The class has an inner class Builder and is created using the Builder instead of the constructor
- The Builder takes optional parameters via setter methods (setX(), setY(), etc.)
- When the client is done supplying parameters he calls build() on the Builder, finalizing the builder and returning an instance of the object desired

# Builder Example – Before

```
// Telescoping constructor pattern - does not scale well!
public class NutritionFacts {
    private final int servingSize; // (mL)      required
    private final int servings;    //           required
    private final int calories;    //           optional
    private final int fat;         // (g)       optional
    private final int sodium;     // (mg)     optional
    private final int carbohydrate; // (g)    optional

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings,
        int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,
        int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings,
        int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
}

public NutritionFacts(int servingSize, int servings,
    int calories, int fat, int sodium) {
    this(servingSize, servings, calories, fat, sodium, 0);
}

public NutritionFacts(
    int servingSize, int servings, int calories,
    int fat, int sodium, int carbohydrate) {
    this.servingSize = servingSize;
    this.servings    = servings;
    this.calories    = calories;
    this.fat         = fat;
    this.sodium      = sodium;
    this.carbohydrate = carbohydrate;
}
}
```

From Joshua Bloch's *Effective Java #2*

# Builder Example – Before

```
// Telescoping constructor pattern - does not scale well!  
public class NutritionFacts {  
    private final int servingSize; // (mL)      required  
    private final int servings;    //           required  
    private final int calories;    //           optional  
    private final int fat;         // (g)       optional  
    private final int sodium;     // (mg)     optional  
    private final int carbohydrate; // (g)     optional  
  
    public NutritionFacts(int servingSize, int servings) {  
        this(servingSize, servings, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings,  
        int calories) {  
        this(servingSize, servings, calories, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings,  
        int calories, int fat) {  
        this(servingSize, servings, calories, fat, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings,  
        int calories, int fat, int sodium) {  
        this(servingSize, servings, calories, fat, sodium, 0);  
    }  
}  
  
public NutritionFacts(int servingSize, int servings,  
    int calories, int fat, int sodium) {  
    this(servingSize, servings, calories, fat, sodium, 0);  
}  
  
public NutritionFacts(  
    int servingSize, int servings, int calories,  
    int fat, int sodium, int carbohydrate) {  
    this.servingSize = servingSize;  
    this.servings    = servings;  
    this.calories    = calories;  
    this.fat         = fat;  
    this.sodium      = sodium;  
    this.carbohydrate = carbohydrate;  
    }  
}
```

```
NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
```

# Builder Example – After

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;

        // Optional parameters
        // initialized to default values
        private int calories    = 0;
        private int fat         = 0;
        private int carbohydrate = 0;
        private int sodium      = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings    = servings;
        }

        public Builder calories(int val)
            { calories = val;      return this; }
        public Builder fat(int val)
            { fat = val;          return this; }
        public Builder carbohydrate(int val)
            { carbohydrate = val; return this; }
        public Builder sodium(int val)
            { sodium = val;       return this; }

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    } // end builder

    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

# Builder Example – After

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;

        // Optional parameters
        // initialized to default values
        private int calories    = 0;
        private int fat         = 0;
        private int carbohydrate = 0;
        private int sodium      = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings    = servings;
        }

        public Builder calories(int val)
            { calories = val;      return this; }
        public Builder fat(int val)
            { fat = val;          return this; }
        public Builder carbohydrate(int val)
            { carbohydrate = val; return this; }
        public Builder sodium(int val)
            { sodium = val;       return this; }

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    } // end builder

    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).  
calories(100).sodium(35).carbohydrate(27).build();

# Builder Advantages

- Useful when you have many constructor parameters
  - It's hard to remember which order they all go in
  - Makes it explicit which parameters mean what
- Easily allows “optional” parameters
  - If you have  $n$  parameters, all of which are optional you'd need  $2^n$  constructors, but only 1 builder

# Builder Example

- Google Protocol Buffers  
(<https://developers.google.com/protocol-buffers/docs/javatutorial>)
- Google uses Protocol Buffers to pass messages between different machines, programs, etc.
- You define PB in a .proto file
  - For example a Person with a name, age, optional phone number, etc.
  - This creates a Person.class with string name, int age, etc.
- To instantiate the Person PB you use Person.Builder



# Structural patterns: Wrappers

- A wrapper translates between incompatible interfaces
- Wrappers are a thin veneer over an encapsulated class
  - modify the interface
  - extend behavior
  - restrict access
- The encapsulated class does most of the work

<b>Pattern</b>	<b>Functionality</b>	<b>Interface</b>
<b>Adapter</b>	<b>same</b>	<b>different</b>
<b>Decorator</b>	<b>different</b>	<b>same</b>
<b>Proxy</b>	<b>same</b>	<b>same</b>

# Adapter

- Change an interface without changing functionality
  - rename a method
  - convert units
  - implement a method in terms of another
- Examples
  - angles passed in radians vs. degrees
  - bytes vs. strings
  - hex vs. decimal numbers

# Decorator

- Add functionality without changing the interface
  - e.g. add caching or benevolent side effects
- Add to existing methods to do something additional (while still preserving the previous specification)
  - e.g add Logging
- Not all subclassing is decoration

# A decorator can remove functionality

- Remove functionality without changing the interface
- Example: UnmodifiableList
  - What does it do about methods like add and put?

# Proxy

- Same interface and functionality as the wrapped class
- Control access to other objects
  - communication: manage network details when using a remote object
  - locking: serialize access by multiple clients
  - security: permit access only if proper credentials
  - creation: object might not yet exist (creation is expensive)
    - hide latency when creating object
    - avoid work if object is never used

# Exercise 1

- Your World of CSEcraft game is slow, because you frequently have to load bulky character classes from the harddisk. The character classes are unique and immutable, and each consumes a lot of memory. Any two clients that want to access the same character class could theoretically share the same object. What pattern would be useful here?

## Exercise 2

- In your code, you often find yourself constructing Shape3D objects, then setting several properties of the object, and so on. To make this less tedious, you write a ShapeMaker class with static methods that you pass appropriate arguments to, and it returns a fully built Shape3D with all properties set appropriately.

# Exercise 3

- Your application uses various kinds of Java input streams to read data. You decide that you'd prefer to read all data in lowercase. So you provide a `LowercaseInputStream` that accepts another `InputStream` as a parameter to its constructor and wraps the original stream with one that reads all of its data as lowercase characters. The lowercase input stream implements the same interfaces as the original stream, so it can be used in place of the original.



# Exercise 4

- Your project needs to connect to a MySQL database. The existing Java JDBC API for connecting to the database demands that you pass it various objects that implement the DatabaseConnector interface. Such an object must implement several complex required methods, but you only want to use one or two of them. So you write a SimpleConnector class that your actual connectors can extend, which implements default versions of most of the methods and leaves the rest empty for subclasses to override with their specific behavior.