# CSE 331

## Software Design & Implementation

### Topic: Design Patterns II

💬 **Discussion:** What advice would you give to a future CSE 331 student?

# Reminders

- No extensions on HW9 (one late day only)
  - Will not accept *any* work after Aug. 19 (Friday) at 11pm
- Next Friday we will do project demos in class for HW9

# Upcoming Deadlines

- Prep. Quiz: HW9 — due Monday (8/13)
- HW9 — due Thursday (8/17)

# Last Time…

- HW9 Overview
- Anonymous Inner Classes
- JSON
- Spark Java (demo)
- Fetch (demo)

Finished demo in section

# Today's Agenda

- More Design Patterns!
  - Creational
  - Behavioral
  - Structural

# Review: Factories

Goal: want more flexible abstractions for what class to instantiate

Factory method (also Singleton)

- call a method to create the object
- method can do computation, return subtype, reuse objects

# Review: Bicycle race

```
class Race {
    public Race() {
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();
        // assume lots of other code here
    }
}
```

Suppose there are different types of races

Each race needs its own type of bicycle…

# Review: Tour de France

```
class TourDeFrance extends Race {
    public TourDeFrance() {
        Bicycle bike1 = new RoadBicycle();
        Bicycle bike2 = new RoadBicycle();
        …
    }
    …
}
```

The Tour de France needs a road bike…

# Review: Cyclocross

```
class Cyclocross extends Race {
    public Cyclocross() {
        Bicycle bike1 = new MountainBicycle();
        Bicycle bike2 = new MountainBicycle();
        …
    }
    …
}
```

And the cyclocross needs a mountain bike.

**Problem**: must override the constructor in every **`Race`** subclass just to use a different subclass of **`Bicycle`**

# Factory *method* for Bicycle

```
class Race {
    Bicycle bike1, bike2;

    Bicycle createBicycle() { return new Bicycle(); }
    public Race() {
        bike1 = createBicycle();
        bike2 = createBicycle();
        ...
    }
}
```

**Solution**: use a factory method to avoid choosing which type to create

- let the subclass decide by overriding **`createBicycle`**

# Subclasses override factory method

```
class TourDeFrance extends Race {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}

class Cyclocross extends Race {
    Bicycle createBicycle() {
        return new MountainBicycle();
    }
}
```

- Requires foresight to use factory method in superclass constructor
- Subtyping in the overriding methods!
- Supports other types of reuse (e.g. **`addBicycle`** could use it too)

# Factory objects

- Let's move the method into a separate class
  - so that it is part of a *factory object*

- Advantages:
  - no longer risks horrifying bugs
  - can pass factories around at runtime
    - e.g., let **`main`** decide which one to use

- Disadvantages:
  - uses bit of extra memory
  - debugging can be more complex when decision of which object to create is far from where it is used

# Factory ***objects*** encapsulate factory method(s)

```
class BicycleFactory {
    Bicycle createBicycle() {
        return new Bicycle();
    }
}
class RoadBicycleFactory extends BicycleFactory {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}
class MountainBicycleFactory extends BicycleFactory {
    Bicycle createBicycle() {
        return new MountainBicycle();
    }
}
```

Note: Ok to return subtypes of **`Bicycle`**!

# Using a factory object

```
class Race {
    BicycleFactory bfactory;

    public Race(BicycleFactory f) {
        bfactory = f;
        Bicycle bike1 = bfactory.createBicycle();
        Bicycle bike2 = bfactory.createBicycle();
        …
    }

    public Race() { this(new BicycleFactory()); }
    …
}
```

Setting up the flexibility here:

- Factory object stored in a field, set by constructor
- Can take the factory as a constructor-argument
- But an implementation detail (?), so 0-argument constructor too
  - Java detail: call another constructor in same class with **`this`**

# The subclasses

```
class TourDeFrance extends Race {
    public TourDeFrance() {
        super(new RoadBicycleFactory());
    }
}

class Cyclocross extends Race {
    public Cyclocross() {
        super(new MountainBicycleFactory());
    }
}
```

Voila!

- Just call the superclass constructor with a different factory
- **`Race`** class had foresight to delegate "what to do to create a bicycle" to the factory object, making it more reusable

# Separate control over bicycles and races

```
class TourDeFrance extends Race {

    public TourDeFrance() {
        super(new RoadBicycleFactory()); // or this(…)
    }

    public TourDeFrance(BicycleFactory f) {
        super(f);
    }
}
```

By having factory-as-argument option, we can allow arbitrary mixing by client:

```
new TourDeFrance(new TricycleFactory())
```

Less useful in this example: Swapping in different factory object whenever you want

Reminder: Not shown here is also using factories for creating *races*

# Builder

Builder: object with methods to describe object and then create it

- fits well with immutable classes when clients want to add data a bit at a time
  - (mutable Builder creates immutable object)

Example 1: **`StringBuilder`**

```
StringBuilder buf = new StringBuilder();
buf.append("Total distance: ");
buf.append(dist);
buf.append(" meters");
return buf.toString();
```

# Builder

Builder: object with methods to describe object and then create it

- fits well with immutable classes when clients want to add data a bit at a time
  - (mutable Builder creates immutable object)

Example 2: **`Graph.Builder`**

- `addNode`, `addEdge`, and `createGraph` methods
- (static inner class `Builder` can use **private** constructors)
- `containsNode` etc. may not need to be especially fast

# Builder Idioms: return `this`

```
class FooBuilder {
    public FooBuilder setX(int x) {
        this.x = x;
        return this;
    }
    public FooBuilder setY(int y) { … }
    public Foo build() { ... }
}
```

You can use this type of Builder like so:

```
Foo f = new FooBuilder().setX(1).setY(2).build();
```

# Methods with Many Arguments

- Builders useful for cleaning up methods with too many arguments
  - recall the problem that clients can easily mix up argument order

E.g., turn this

```
myMethod(x, y, true, false, true);
```

into this

```
myMethod(x, y, Options.create()
                        .setA(true)
                        .setB(false)
                        .setC(true).build());
```

This simulates named (rather than positional) argument passing.

# Prototype pattern

- Each object is itself a factory:
  - objects contain a **`clone`** method that creates a copy

- Useful for objects that are created via a process
  - Example: java.awt.geom.AffineTransform
    - create by a sequence of calls to translate, scale, etc.
    - easiest to make a similar one by copying and changing
  - Example: android.graphics.Paint
  - Example: JavaScript classes
    - use prototypes so every instance doesn't have all methods stored as fields

# Review: Factories and Prototypes

Goal: want more flexible abstractions for what class to instantiate

Factory method (also Singleton)

- call a method to create the object
- method can do computation, return subtype, reuse objects

Factory object (also Builder)

- Factory has factory methods for some type(s)
- Builder has methods to describe object and then create it

Prototype

- every object is a factory, can create more objects like itself
- call **`clone`** to get a new object of same subtype as receiver

# Sharing

Second weakness of constructors: they always return a *new object*

Singleton: only one object exists at runtime

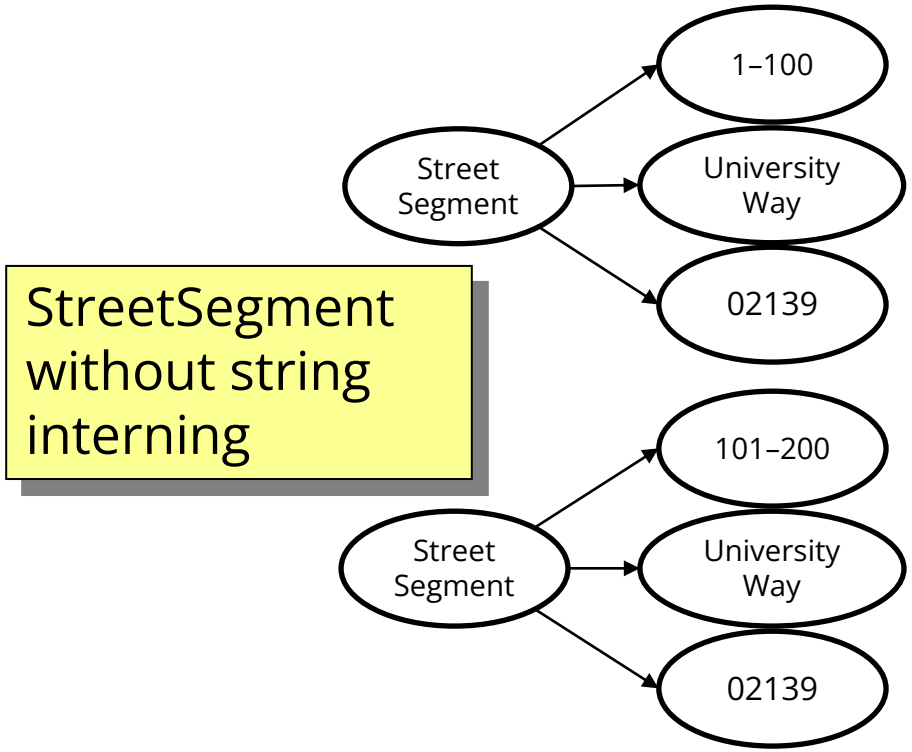
- factory method returns the same object every time
- (we've seen this already)

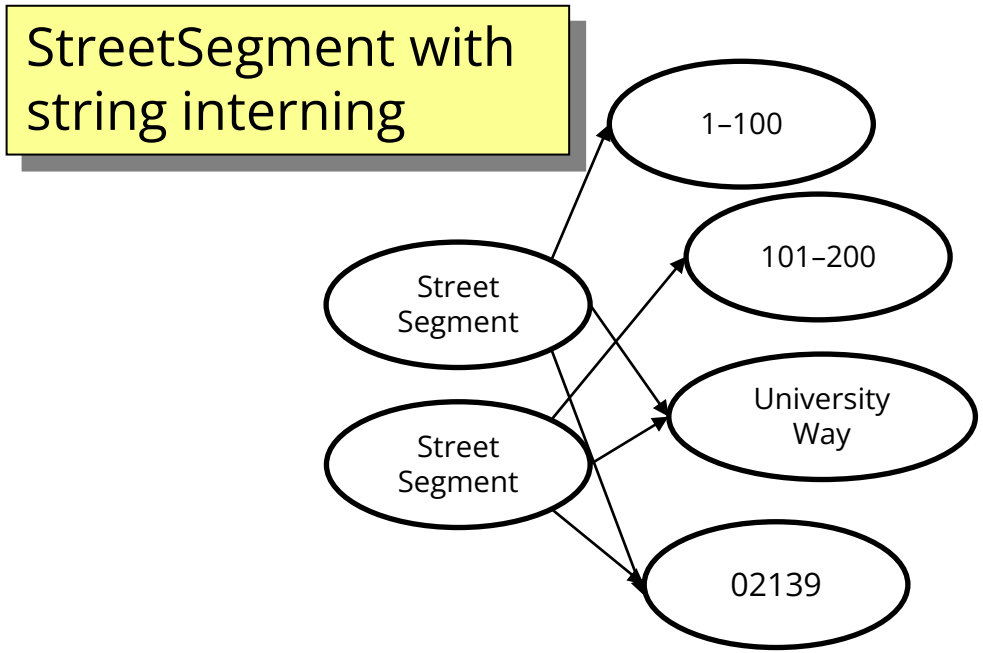
Interning: only one object with a particular (abstract) value exists at runtime

- factory method can return an existing object (not a new one)
- interning can be used without factory methods
  - see `String.intern`

# Interning pattern

Reuse existing objects instead of creating new ones:



# Interning mechanism

- Maintain a collection of all objects in use
- If an object already appears, return that instead
  - (be careful in multi-threaded contexts)

```
HashMap<String, String> segNames;
String canonicalName(String n) {
    if (segNames.containsKey(n)) {
        return segNames.get(n);
    } else {
        segNames.put(n, n);
        return n;
    }
}
```

Why not `Set<String>` ?

`Set` supports `contains` but not `get`

- Java builds this in for strings: `String.intern()`

# Interning pattern

- Benefits of interning:

  1. May compare with `==` instead of `equals()`
     - eliminates a source of common bugs!! Although still good to use .equals

  2. May save space by creating fewer objects
     - (space is less and less likely to be a problem nowadays)
     - also, interning can actually waste space if objects are not cleaned up when *no longer needed*
       - there are additional techniques to fix that ("weak references")

- Sensible only for immutable objects

# `java.lang.Boolean`
## does not use the Interning pattern

```
public class Boolean {
    private final boolean value;

    // construct a new Boolean value
    public Boolean(boolean value) {
        this.value = value;
    }

    public static Boolean FALSE = new Boolean(false);
    public static Boolean TRUE = new Boolean(true);

    // factory method that uses interning
    public static Boolean valueOf(boolean value) {
        if (value) {
            return TRUE;
        } else {
            return FALSE;
        }
    }
}
```

# Recognition of the problem

Javadoc for **`Boolean`** constructor:

> Allocates a **`Boolean`** object representing the value argument.
>
> **Note:** It is rarely appropriate to use this constructor. Unless a new instance is required, the static factory `valueOf(boolean)` is generally a better choice. It is likely to yield significantly better space and time performance.

Josh Bloch (JavaWorld, January 4, 2004):

> The **`Boolean`** type should not have had public constructors. There's really no great advantage to allow multiple **`true`**s or multiple **`false`**s, and I've seen programs that produce millions of **`true`**s and millions of **`false`**s, creating needless work for the garbage collector.
>
> So, in the case of immutables, I think factory methods are great.

# GoF patterns: three categories

*Creational Patterns* are about the object-creation process

Factory Method, Abstract Factory, Singleton, Builder, Prototype, Interning …

*Structural Patterns* are about how objects/classes can be combined

Adapter, Bridge, Composite, Decorator, Façade, Proxy, …

*Behavioral Patterns* are about communication among objects

Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method, …

Green = ones we've seen already

# Structural patterns: Wrappers

Wrappers are a thin veneer over an encapsulated class

- modify the interface
- extend behavior
- restrict access

The encapsulated class does most of the work

| | **Functionality** | **Interface** |
|---|---|---|
| **Adapter** | same | different |
| **Decorator** | different | same |
| **Proxy** | same | same |

Some wrappers have qualities of more than one of adapter, decorator, and proxy

# Adapter

Real life example: adapter to go from US to UK power plugs

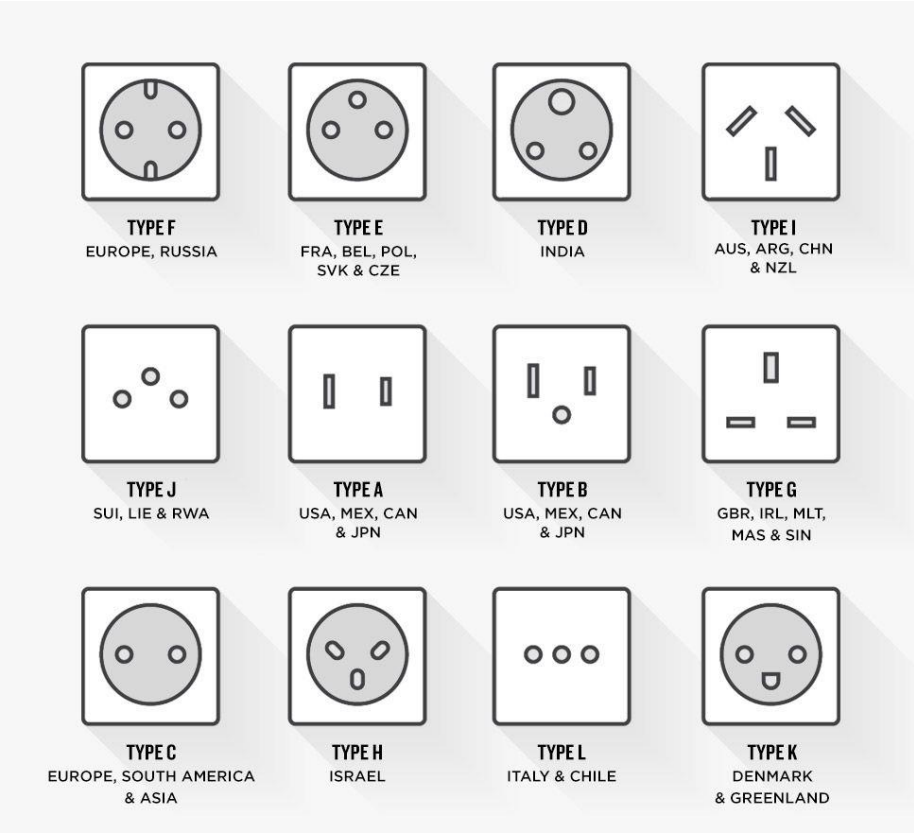
- both do the same thing
- but they have slightly interface expectations

Change an interface without changing functionality

- rename a method
- convert units
- implement a method in terms of another

Example: angles passed in radians vs. degrees

Example: use “old” method names for legacy code



# Adapter example: rectangles

Our code is using this **`Rectangle`** interface:

```
interface Rectangle {
  // grow or shrink this by the given factor
  void scale(float factor);
  // move to the left or right
  void translate(float x, float y);
}
```

But we want to use a library that has this class:

```
class JRectangle {
  void scaleWidth(float factor) { ... }
  void scaleHeight(float factor) { ... }
  void shift(float x, float y) { ... }
}
```

# Adapter example: rectangles

Create an adapter that delegates to **`Rectangle`**:

```
class RectangleAdapter implements Rectangle {
    private JRectangle rect;

    public RectangleAdapter(JRectangle rect) {
        this.rect = rect;
    }

    void scale(float factor) {
        rect.scaleWidth(factor);
        rect.scaleHeight(factor);
    }


    void translate(float x, float y) {
        rect.shift(x, y);
    }
}
```

# Adapters

- This sort of thing happens **a lot**
  - unless two libraries were designed to work together, they won't work together without an adapter
- The example code uses **delegation**
  - special case of composition where the outer object just forwards calls on to one other object
- Adapters can also **remove** methods
- Adapters can (in principle) be written by subclassing
  - but then all the usual warnings about subclassing apply ***if*** you override any methods of the superclass
  - your subclass could easily break when superclass changes

# Decorator

Add functionality without breaking the interface:

1. Add to existing methods to do something extra
   - satisfying a stronger specification
2. Provide extra methods

Subclasses are often decorators

- but not always: Java subtypes are not always true subtypes

# Decorator example: Bordered windows

```
interface Window {
    // rectangle bounding the window
    Rectangle bounds();
    // draw this on the specified screen
    void draw(Screen s);
    ...
}

class WindowImpl implements Window {
    ...
}
```

# Bordered window implementations

```
class BorderedWindow1 extends WindowImpl {
    void draw(Screen s) {
        super.draw(s);
        bounds().draw(s);
    }
}


class BorderedWindow2 implements Window {
    Window innerWindow;
    BorderedWindow2(Window innerWindow) {
        this.innerWindow = innerWindow;
    }
    void draw(Screen s) {
        innerWindow.draw(s);
        innerWindow.bounds().draw(s);
    }
}
```

Delegation permits multiple borders on a window, or a window that is both bordered and shaded

# A decorator can remove functionality

Remove functionality without changing the Java interface

- no longer a true subtype, but *sometimes* that is necessary

Example: **`UnmodifiableList`**

- What does it do about methods like **`add`** and **`put`**?
  - throws an exception
  - moves error checking from the compiler to runtime

  Problem: **`UnmodifiableList`** is not a true subtype of **`List`**

  Decoration via delegation can create a class with no Java subtyping relationship, which is often desirable
  - Java subtypes that are not true subtypes are **confusing**
  - maybe necessary for **`UnmodifiableList`** though

# Proxy

- Same interface *and* functionality as the wrapped class
  - so… uh… wait, what?

- 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

# Structural patterns: Wrappers

Wrappers are a thin veneer over an encapsulated class

- modify the interface
- extend behavior
- restrict access

The encapsulated class does most of the work

| | **Functionality** | **Interface** |
|---|---|---|
| **Adapter** | same | different |
| **Decorator** | different | same |
| **Proxy** | same | same |

Some wrappers have qualities of more than one of adapter, decorator, and proxy

# Composite pattern

- Composite permits a client to manipulate either an *atomic* unit or a *collection* of units in the same way
  - no need to "always know" if an object is a collection of smaller objects or not
- Good for dealing with "part-whole" relationships
- Used by jQuery in JavaScript
- An extended example…

# Composite example: Bicycle

- Bicycle
  - Wheel
    - Skewer
      - Lever
      - Body
      - Cam
      - Rod
    - Hub
    - Spokes
    - Nipples
    - Rim
    - Tape
    - Tube
    - Tire
  - Frame
  - Drivetrain
  - …

# Methods on components

```
interface BicycleComponent {
    int weight();
    public float cost();
}

class Skewer extends BicycleComponent {
    float price;
    public float cost() { return price; }
}

class Wheel extends BicycleComponent {
    float assemblyCost;
    Skewer skewer;
    Hub hub;
    ...
    public float cost() {
        return assemblyCost + skewer.cost() + hub.cost() + ...;
    }
}
```

# Composite example: Libraries

Library
  Section (for a given genre)
    Shelf
      Volume
        Page
          Column
            Word
              Letter

```
interface Text {
    String getText();
}
class Page implements Text {
    String getText() {
        ... return concatenation of column texts ...
    }
}
```

# Composite example: jQuery

- jQuery provides a function **`$`** that returns one or many objects
  - **`$("p")`** would return a collection of all **`<p>`** nodes
  - **`$("#foo")`** would return the object with ID "foo"
    - (or returns an empty collection if none exists)

- Calling a method on a jQuery object calls that method on all objects in the collection:
  - **`$("p").hide()`** would hide *all* the **`<p>`** nodes
  - if **`foo`** is a node with id "foo", then **`foo.hide()`** has the same effect as **`$("#foo").hide()`**

# Before next class…

1. Start on HW9
   - React is new, you will likely have many questions
   - See examples from lecture + section for ideas

2. Wrap-up any regrades for HW1-8
   - Won't accept late work after the last day of class