

# Design Patterns (part 1)

CSE 403

University of Washington

Michael Ernst

# What is a design pattern?

- A standard solution to a common programming problem

# Example 1: Encapsulation (data hiding)

- Problem: **Exposed fields** can be directly manipulated
  - Violations of the representation invariant
  - Dependences prevent changing the implementation
- Solution: Hide some components
  - Constrain ways to access to the object
- Disadvantages:
  - Interface may not (efficiently) provide all desired operations
  - Indirection may reduce performance

## Example 2: Subclassing (inheritance)

- Problem: **Repetition** in implementations
  - Similar abstractions have similar members (fields, methods)
- Solution: Inherit default members from a superclass
  - Select an implementation via run-time dispatching
- Disadvantages:
  - Code for a class is spread out, and thus less understandable
  - Run-time dispatching introduces overhead

# Example 3: Iteration

- Problem: To access all members of a collection, must perform a **specialized traversal** for each data structure
  - Introduces undesirable dependences
  - Does not generalize to other collections
- Solution:
  - The implementation performs traversals, does bookkeeping
    - The implementation has knowledge about the representation
  - Results are communicated to clients via a standard interface (e.g., `hasNext()`, `next()`)
- Disadvantages:
  - Iteration order is fixed by the implementation and not under the control of the client

# Example 4: Exceptions

- Problem:
  - **Errors** in one part of the code should be **handled** elsewhere.
  - Code should not be cluttered with error-handling code.
  - Return values should not be preempted by error codes.
- Solution: Language structures for throwing and catching exceptions
- Disadvantages:
  - Code may still be cluttered.
  - It may be hard to know where an exception will be handled.
  - Use of exceptions for normal control flow may be confusing and inefficient.

# Example 5: Generics

- Problem:
  - Well-designed data structures hold **one type of object**
  - Wish to avoid code duplication
- Solution:
  - Programming language checks for errors in contents
  - `List<Date>` instead of just `List`
- Disadvantages:
  - More verbose types

# Other examples

- Reuse implementation without subtyping
- Reuse implementation, but change interface
- Permit a class to be instantiated only once
- Constructor that might return an existing object
- Constructor that might return a subclass object
- Combine behaviors without compile-time  
`extends` clauses



# Why should you care about design patterns?

- You could come up with these solutions on your own
  - You shouldn't have to!
- A design pattern is a known solution to a known problem

# What is a design pattern?

- A standard solution to a common programming problem
  - a design or implementation structure that achieves a particular purpose
  - a high-level programming idiom
- A technique for making code more flexible
  - reduce coupling among program components
- Shorthand for describing program design
  - connections among program components
  - the shape of a heap snapshot or object model
- Vocabulary for communication & documentation

# Why do we need design patterns?

- The programming language does not build in solutions to every problem (why not?)
  - Best solution depends on context
  - Every language has shortcomings
    - So does every paradigm: OO, functional, declarative, ...
  - Language features start out as design patterns

# When (not) to use design patterns

- Rule 1: delay
  - Get something basic and concrete working first
  - Improve or generalize it once you understand it
- Design patterns can increase or decrease understandability
  - Usually adds indirection, increases code size
  - Improves modularity and flexibility, separates concerns, eases description
- If your design or implementation has a problem, consider design patterns that address that problem
- Canonical reference: the "Gang of Four" book
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides
- Another good reference for Java
  - *Effective Java: Programming Language Guide*, by Bloch

# Outline

- Introduction to design patterns
- **Creational** patterns
  - constructing objects
- **Structural** patterns
  - combining objects, controlling heap layout
- **Behavioral** patterns
  - communicating among objects, affecting object semantics

# Creational patterns

```
new MyClass(x, y, z)
```

- Constructors in Java are inflexible
  1. Can't return a subtype of the class they belong to
  2. Always return a fresh new object, never re-use one
- Factories – ADT creators that are not Java constructors
  - Factory method
  - Factory object
  - Prototype
  - Dependency injection
- Sharing – reuse objects (save space)
  - Singleton – only one object ever exists
  - Interning – only one object with a given abstract value exists
  - Flyweight – share part of an object's representation

# Factories

- Problem: client desires control over object creation
- **Factory method** = creator method
  - Hides decisions about object creation
  - Implementation: put code in methods in client
- **Factory object** = has a creator op, can be *passed around*
  - Bundles factory methods for a family of types
  - Implementation: put code in a separate object
- **Prototype** = knows how to clone itself
  - Every object is a factory, can create more objects like itself
  - Implementation: put code in `clone` methods
- **Dependency injection** = external reference to a creator op
  - Client controls construction, without changing code
  - Implementation: read method name from a file, call reflectively

# Motivation for factories:

## Changing implementations

- A supertype may have multiple implementations

```
interface Matrix { ... }
```

```
class DenseMatrix implements Matrix { ... }
```

```
class SparseMatrix implements Matrix { ... }
```

- Clients declare variables using the supertype (**Matrix**)
  - Clients must use a `SparseMatrix` Or `DenseMatrix` **constructor**
    - Code: `new SparseMatrix(...)` Or `new DenseMatrix(...)`
  - Switching implementations requires **code changes** 😞



# Use of factories

## Factory

```
class MatrixFactory {  
    public static Matrix createMatrix(...) {  
        if (...) {  
            return new SparseMatrix(...);  
        }  
        else {  
            return new DenseMatrix(...);  
        }  
    }  
}
```

Clients call **createMatrix**, not a particular constructor

## Advantages

- To switch the implementation, only change **one** place
- Factory method can decide at run time what to create

# Factory method in the Java JDK

```
class Calendar {  
    static Calendar getInstance(Locale) ;  
}
```

might return a BuddhistCalendar,  
JapaneseImperialCalendar,  
GregorianCalendar, ...

# DateFormat factory methods

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)); // "juedi 4 juillet 1776"
```

# Bicycle race without factories

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

# Specializations of bicycle race

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

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

Reimplemented the constructor just to use a different subclass of **Bicycle**

# Bicycle race without factories

```
class Race {  
  
    Race() {  
        Bicycle bike1 = new Bicycle();  
        Bicycle bike2 = new Bicycle();  
        ...  
    }  
}
```

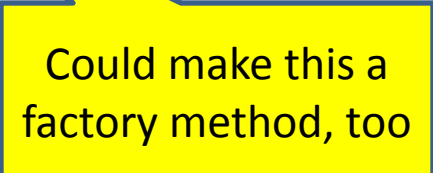
# Defining a factory method

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

Defining and using  
a factory method  
requires foresight.

# Overriding a factory method

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



Could make this a factory method, too

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

No need to override constructor!

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



# Factory **objects**/classes

## encapsulate factory methods

```
class BicycleFactory {  
    Bicycle createBicycle() { ... }  
    Frame createFrame() { ... }  
    Wheel createWheel() { ... }  
    ...  
}
```

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

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

# Using a factory object

```
class Race {  
    public Race(BicycleFactory bfactory) {  
        Bicycle bike1 = bfactory.createBicycle();  
        Bicycle bike2 = bfactory.createBicycle();  
        ...  
    }  
}
```

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

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

# Separate control over bicycles and races

```
class Race {  
    public Race(BicycleFactory bfactory) {  
        Bicycle bike1 = bfactory.createBicycle();  
        Bicycle bike2 = bfactory.createBicycle();  
        ...  
    }  
}
```

No need for a constructor for **TourDeFrance** or **Cyclocross**

Delegate bicycle creation to a factory object  $\Rightarrow$  flexibility

- Specify the race and the bicycle separately

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

- Change the factory at run time by setting the field.

Might want a default constructor: **new TourDeFrance()**

# Prototype pattern

- Every object is itself a factory
- Each class contains a **clone** method that creates a copy of the receiver object

```
class Bicycle {  
    Bicycle clone() { ... }  
}
```

- You might see **Object** as the return type of **clone**
  - **clone** is declared in **Object**
  - Design flaw in Java 1.4 and earlier: the return type may not change covariantly in an overridden method

# Using prototypes

```
class Race {  
  
    public Race(Bicycle bproto) {  
        Bicycle bike1 = (Bicycle) bproto.clone();  
        Bicycle bike2 = (Bicycle) bproto.clone();  
        ...  
    }  
}
```

Compare to using a factory object:

```
public Race(BicycleFactory bfactory) {  
    Bicycle bike1 = bfactory.createBicycle();  
}
```

Again, we can specify the race and the bicycle separately:

```
new TourDeFrance(new Tricycle())
```

# Dependency injection

Change the factory without changing the code

- With a regular factory object:

```
BicycleFactory f = new TricycleFactory();  
Race r = new TourDeFrance(f)
```

- With external dependency injection:

```
BicycleFactory f = (BicycleFactory)  
    DependencyManager.get("BicycleFactory");  
Race r = new TourDeFrance(f);
```

*Plus* an external file:

```
<service-point id="BicycleFactory">  
  <invoke-factory>  
    <construct class="Bicycle">  
      <service>Tricycle</service>  
    </construct>  
  </invoke-factory>  
</service-point>
```

- + Change the factory without recompiling
- Program requires external file to run
- Mistakes in the file are caught at run time

# Aside: Reflection (= meta-programming)

- Call a method named by a string

Ordinary code: `Date y = MyClass.foo();`

Reflective code:

```
String className = "MyClass";
```

```
String methodName = "foo";
```

```
Class[] paramTypes = {};
```

```
Object[] args = {};
```

```
Class clazz = Class.forName(className);
```

```
// above line has same effect as: c
```

```
Method method = clazz.getMethod(methodName, paramTypes);
```

```
Date y = (Date) method.invoke(null, args);
```

- Access fields
- List methods and fields
- Change visibility (private to public)

- + Powerful
- Error-prone, confusing, no compile-time checking

Why?

- Dependency injection
- Access private fields & methods
- Different library versions (w/ or w/o a method)
- Debuggers & programming tools
- (De)serialization
- Obfuscation (malicious code)

# Sharing

Recall the second weakness of Java constructors

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 run time
  - 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



# Singleton

- Only one object of the given type exists
- Shared resource
- Examples:
  - Cache
  - `FileSystem`, `ThreadPool`, `Runtime`
  - I/O: `KeyboardReader`, `PrinterController`, `Desktop`
  - `Logger` for diagnostic messages
  - Configuration file
- An object has fields like “static fields” but a constructor decides their values
  - Logically group the values (don’t pollute namespace)
  - Example: Internationalization: messages in a particular language

# Singleton

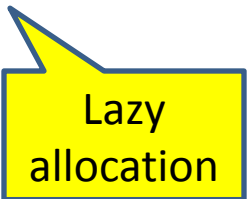
Only one object of the given type exists

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

# Singleton


Only one object of the given type exists

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

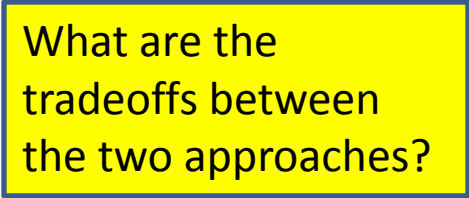


Lazy allocation

```
class Bank {  
    private static Bank theBank  
        = new Bank();  
  
    // private constructor  
    private Bank() { ... }  
  
    // factory method  
    public static Bank getBank() {  
  
        return theBank;  
    }  
    ...  
}
```



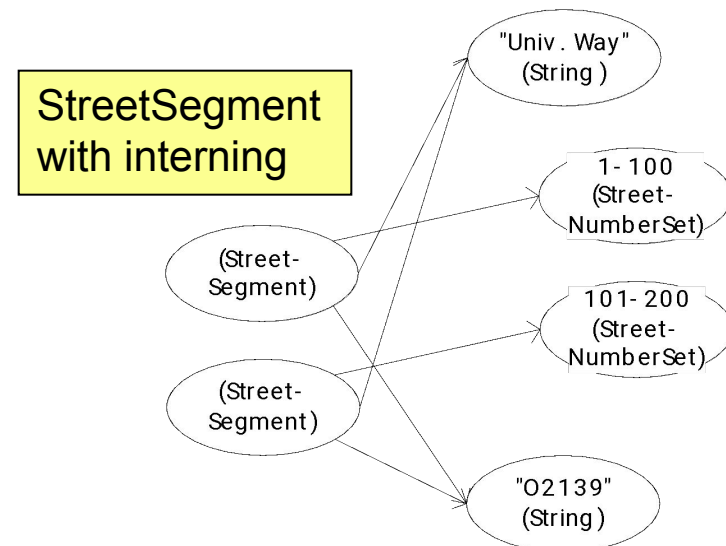
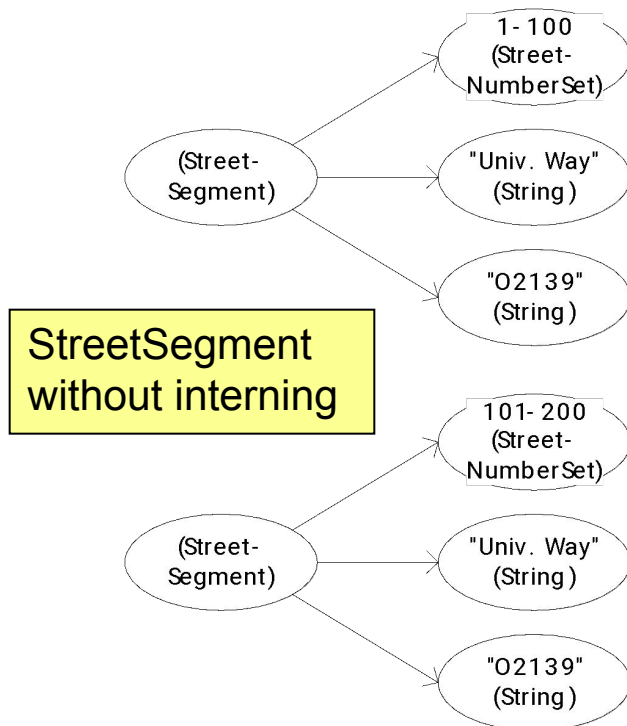
Eager allocation



What are the tradeoffs between the two approaches?

# Interning pattern

- Reuse existing objects instead of creating new ones
  - Less space
  - May compare with `==` instead of `equals()`
- Sensible only for **immutable** objects
- Java builds this in for strings: `String.intern()`



# How to implement interning

This ought to use *weak references*.

- Maintain a collection of all objects
- If an object already appears, return that instead

```
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>`?

Why not  
`return n`?

It is not the  
canonical  
value

Set supports  
`contains` (which  
USES `equals`) but  
not `get`; no good  
way to get the  
canonical value.

- Two approaches:
  - Create the object, but perhaps discard it and return another
  - Check against the arguments before creating the new object

# java.lang.Boolean constructor does not use interning

```
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 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.**

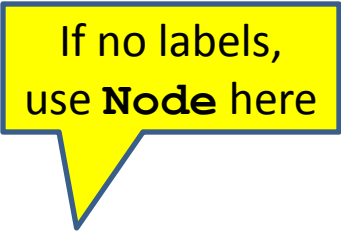
There's really no great advantage to allow multiple trues or multiple falses, and I've seen programs that produce **millions of trues and millions of falses**, creating needless work for the garbage collector.

So, in the case of immutables, I think factory methods are great.

# Save space by not storing data twice (one aspect of “Flyweight” pattern)

```
class Edge {  
    Node start;  
    Node end;  
    String label;  
}  
  
class Graph {  
    Map<Node, Set<Edge>> edges;  
}  
  
// client code  
Edge e = g.getFirstEdge(n);  
... e.start ... e.end ... e.label ...
```

```
class OutgoingEdge {  
    Node start;  
    Node end;  
    String label;  
}  
  
class Graph {  
    Map<Node, Set<OutgoingEdge>> edges;  
}  
  
// client code  
Edge e = g.getFirstEdge(n);  
... n ... e.end ... e.label ...
```



If no labels,  
use Node here

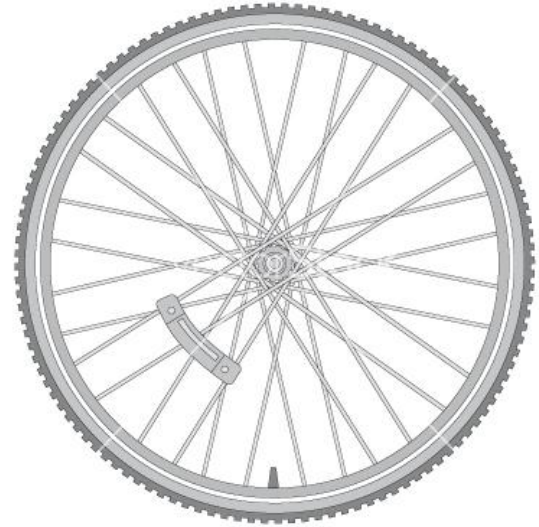


# Flyweight pattern

- Good when many objects are mostly the same
  - Interning works only if objects are **entirely** the same and **do not change** (e.g., immutable)
- **Intrinsic state**: same across all objects
  - Technique: intern it (interning requires immutability)
- **Extrinsic state**: different for different objects
  - Represent it explicitly
  - Advanced technique: make it implicit (don't even represent it!)
    - Clients store or compute it
    - Implicit data must not change (or can be recomputed)

# Example without flyweight: bicycle spoke

```
class Wheel {  
    FullSpoke[] spokes;  
    ...  
}  
class FullSpoke {  
    int length;  
    int diameter;  
    bool tapered;  
    Metal material;  
    float weight;  
    float threading;  
    bool crimped;  
    int location;    // position on the rim and hub  
}
```



Typically 32 or 36 spokes per wheel

but only 3 varieties per bicycle.

In a bike race, hundreds of spoke varieties, millions of instances

# Alternatives to FullSpoke

```
// Represents a spoke but not its location
class IntrinsicSpoke {
    int length;
    int diameter;
    boolean tapered;
    Metal material;
    float weight;
    float threading;
    boolean crimped;
    // no location field (FullSpoke has a location field)
}
```

This doesn't save space: it's the **same** as FullSpoke:

```
class InstalledSpokeFull extends IntrinsicSpoke {
    int location;
}
```

This saves space:

```
class InstalledSpokeWrapper {
    IntrinsicSpoke s;    // refer to interned object
    int location;
}
```

... but the flyweight version will use even less space

# Original code to true (align) a wheel

```
// This class is interned
class FullSpoke {
    // Tension the spoke by turning the nipple the
    // specified number of turns.
    // modifies: the wheel but not the spoke
    void tighten(int turns) {
        ... location ...    // location is a field
    }
}

class Wheel {
    FullSpoke[] spokes;

    void align() {
        while (wheel is misaligned) {
            // tension the  $i^{\text{th}}$  spoke, which affects the wheel
            ... spokes[i].tighten(numturns) ...
        }
    }
}
```

What is the value of the  
location field in `spokes[i]`?



# Flyweight code to true (align) a wheel

```
// This class is interned
```

```
class IntrinsicSpoke {
```

```
    // Tension the spoke by turning the nipple the
```

```
    // specified number of turns.
```

```
    // modifies: the wheel but not the spoke
```

```
    void tighten(int turns, int location) {
```

```
        ... location ...    // location is a parameter
```

```
    }
```

```
}
```

```
class Wheel {
```

```
    IntrinsicSpoke[] spokes;
```

```
    void align() {
```

```
        while (wheel is misaligned) {
```

```
            // tension the  $i^{\text{th}}$  spoke, which affects the wheel
```

```
            ... spokes[i].tighten(numturns, i) ...
```

```
        }
```

```
    }
```

```
}
```

Represent only the intrinsic state.

Use interning to save space.

Logically, each spoke has intrinsic state and a location.

Is this a reasonable abstraction?

Clients store or compute extrinsic state (location).



# Flyweight discussion

**Wheel** methods pass this to the methods that use the **wheel** field.

- What if **FullSpoke** contains a **wheel** field pointing at the **Wheel** containing it?
- What if **FullSpoke** contains a **boolean** broken field?

Add an array of **booleans** in **Wheel**, parallel to the array of **Spokes**.

Flyweight is rarely used

- Flyweight is manageable only if there are very few mutable (extrinsic) fields.
- Flyweight complicates the code.
- Use flyweight only when profiling has determined that space is a *serious* problem.