• Introduction to design patterns
• Creational patterns (constructing objects)
• Structural patterns (controlling heap layout)
• Behavioral patterns (affecting object semantics)
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
  – a description of connections among program components
  – the shape of a heap snapshot or object model
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
  – Permit only stylized 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
• 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

• Solution:
  – Programming language checks for errors in contents
    – List<Date> instead of just List

• Disadvantages:
  – More verbose types
When (not) to use design patterns

• Rule 1: delay
  – Get something basic working first
  – Improve it once you understand it
• Design patterns can increase or decrease understandability
  – Add indirection, increase code size
  – Improve modularity, separate concerns, ease 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 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.
• Another good reference for Java
Why should you care?

• 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
Creational patterns

• 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
  – Factory method
  – Factory object
  – Prototype
  – Dependency injection

• Sharing
  – Singleton
  – Interning
  – Flyweight
Factories

• Problem: client desires control over object creation
• Factory method
  – Hides decisions about object creation
  – Implementation: put code in methods in client
• Factory object
  – Bundles factory methods for a family of types
  – Implementation: put code in a separate object
• Prototype
  – Every object is a factory, can create more objects like itself
  – Implementation: put code in clone methods
Motivation for factories: Changing implementations

• Supertypes support multiple implementations
  
  ```java
  interface Matrix { ... }
  class SparseMatrix implements Matrix { ... }
  class DenseMatrix implements Matrix { ... }
  ```

• Clients use the supertype (Matrix)
  
  – Still need to use a `SparseMatrix` or `DenseMatrix` constructor
  – Switching implementations requires code changes
Use of factories

• Factory
  class MatrixFactory {
    public static Matrix createMatrix() {
      return new SparseMatrix();
    }
  }

• Clients call createMatrix, not a particular constructor

• Advantages
  – To switch the implementation, only change one place
  – Can decide what type of matrix to create
Example: bicycle race

class Race {

    // factory method
    Race createRace() {
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();
        ...
    }

}
Example: Tour de France

class TourDeFrance extends Race {

    // factory method
    Race createRace() {
        Bicycle bike1 = new RoadBicycle();
        Bicycle bike2 = new RoadBicycle();
        ...
    }

}
Example: Cyclocross

class Cyclocross extends Race {

    // factory method
    Race createRace() {
        Bicycle bike1 = new MountainBicycle();
        Bicycle bike2 = new MountainBicycle();
        ...
    }
}
Factory **method** for Bicycle

class Race {
    Bicycle createBicycle() { ... }
    Race createRace() {
        Bicycle bike1 = createBicycle();
        Bicycle bike2 = createBicycle();
        ...
    }
}
Code using factory methods

class Race {
    Bicycle createBicycle() { ... }
    Race createRace() {
        Bicycle bike1 = createBicycle();
        Bicycle bike2 = createBicycle();
        ...
    }
}

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

class Cyclocross extends Race {
    Bicycle createBicycle(Frame) {
        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 {
    BicycleFactory bfactory;
    // constructor
    Race() { bfactory = new BicycleFactory(); }
    Race createRace() {
        Bicycle bike1 = bfactory.createBicycle();
        Bicycle bike2 = bfactory.createBicycle();
        ...
    }
}

class TourDeFrance extends Race {
    // constructor
    TourDeFrance() { bfactory = new RoadBicycleFactory(); }
}

class Cyclocross extends Race {
    // constructor
    Cyclocross() { bfactory = new MountainBicycleFactory(); }
}
Separate control over bicycles and races

class Race {
    BicycleFactory bfactory;
    // constructor
    Race(BicycleFactory bfactory) { this.bfactory = bfactory; }
    Race createRace() {
        Bicycle bike1 = bfactory.completeBicycle();
        Bicycle bike2 = bfactory.completeBicycle();
        ...
    }
}
// No special constructor for TourDeFrance or for Cyclocross

Now we can specify the race and the bicycle separately:

    new TourDeFrance(new TricycleFactory())
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)); // “jueudi 4 juillet 1776"
Prototype pattern

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

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

- Often, `Object` is 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 {
    Bicycle bproto;
    // constructor
    Race(Bicycle bproto) { this.bproto = bproto; }
    Race createRace() {
        Bicycle bike1 = (Bicycle) bproto.clone();
        Bicycle bike2 = (Bicycle) bproto.clone();
        ...
    }
}

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 in-code factory:
  ```java
  BicycleFactory f = new TricycleFactory();
  Race r = new TourDeFrance(f);
  ```
- With external dependency injection:
  ```java
  BicycleFactory f = ((BicycleFactory)
  DependencyManager.get("BicycleFactory");
  Race r = new TourDeFrance(f);
  ```
- plus an external file:
  ```xml
  <service-point id="BicycleFactory">
    <invoke-factory>
      <construct class="Bicycle">
        <service>Tricycle</service>
      </construct>
    </invoke-factory>
  </service-point>
  ```

+ Change the factory without recompiling
- Harder to understand
- Easier to make mistakes
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 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
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;
    }

    ...
}
Interning pattern

- Reuse existing objects instead of creating new ones
  - Less space
  - May compare with `==` instead of `equals()`
- Permitted only for immutable objects
**Interning mechanism**

- Maintain a collection of all objects
- If an object already appears, return that instead
  ```java
  HashMap<String, String> segnames; // why not Set<String>?
  String canonicalName(String n) {
    if (segnames.containsKey(n)) {
      return segnames.get(n);
    } else {
      segnames.put(n, n);
      return n;
    }
  }
  ```
- Java builds this in for strings: `String.intern()`
- Two approaches:
  - create the object, but perhaps discard it and return another
  - check against the arguments before creating the new object
java.lang.Boolean does not use the Interning pattern

```java
class Boolean {
    private final boolean value;
    // construct a new Boolean value
    public Boolean(boolean value) {
        this.value = value;
    }

    static Boolean FALSE = new Boolean(false);
    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 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.
Flyweight pattern

• Good when many objects are mostly the same
  – Interning works only if objects are *entirely* the same (and *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!)
    • Making it implicit requires immutability (or other properties)
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;  // rim and hub holes this is installed in
}

Typically 32 or 36 spokes per wheel
but only 3 varieties per bicycle.
In a bike race, hundreds of spoke varieties, millions of instances
class IntrinsicSpoke {
    int length;
    int diameter;
    boolean tapered;
    Metal material;
    float weight;
    float threading;
    boolean crimped;
}

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 flyweight version uses even less space
class FullSpoke {
    // Tension the spoke by turning the nipple the
    // specified number of turns.
    void tighten(int turns) {
        ... location ... // location is a field
    }
}

class Wheel {
    FullSpoke[] spokes;
    void align() {
        while (wheel is misaligned) {
            // tension the i^{th} spoke
            ... spokes[i].tighten(numturns) ...
        }
    }
}

What is the value of the location field in spokes[i]?
class IntrinsicSpoke {
    void tighten(int turns, int location) {
        // location is a parameter
    }]
}

class Wheel {
    IntrinsicSpoke[] spokes;

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

Flyweight code to true (align) a wheel
Flyweight discussion

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

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.