Section 9: Design Patterns

Slides by Alex Mariakakis

with material from David Mailhot, Hal Perkins, Mike Ernst

Agenda

• What are design patterns?
• Creational patterns
• Structural patterns

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 and how program components are connected

Creational Patterns

• Problem: Constructors in Java are not flexible
  o Always return a fresh new object, never reuse one
  o Can’t return a subtype of the class they belong to

• Solution: Creational patterns!
  o Sharing
    • Singleton
    • Interning
    • Flyweight
  o Factories
    • Factory method
    • Factory object
  o Builder

Creational Patterns: Sharing

• The old way: Java constructors always return a new 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, represents them separately, and interns the intrinsic state
    • Implicit representation uses no space
    • Not as common/important
Creational Patterns: Singleton

- For a class where only one object of that class can ever exist
- Two possible implementations
  - Eager instantiation: creates the instance when the class is loaded to guarantee availability
  - Lazy instantiation: only creates the instance once it’s needed to avoid unnecessary creation

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

Creational Patterns: Singleton

- Lazy instantiation

public 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();
}

Creational Patterns: Singleton

Would you prefer eager or lazy instantiation for an HttpRequest class?
- handles authentication
- definitely needed for any HTTP transaction

Would you prefer eager or lazy instantiation for a Comparator class?
- compares objects
- may or may not be used at runtime

public class HttpRequest {
  private static class HttpRequestHolder {
    public static final HttpRequest INSTANCE = new HttpRequest();
  } /* Singleton - Don’t instantiate */
  private HttpRequest() { … }
  public static HttpRequest getInstance() {
    return HttpRequestHolder.INSTANCE;
  }
}

Creational Patterns: Singleton

public class LengthComparator implements Comparator<String> {
  private int compare(String s1, String s2) {
    return s1.length()-s2.length();
  }
  /* Singleton - Don’t instantiate */
  private LengthComparator() { … }
  private static LengthComparator comp = null;
  public static LengthComparator getInstance() {
    if (comp == null) {
      comp = new LengthComparator();
    }
    return comp;
  }
}

Creational Patterns: Singleton
Creational Patterns: Interning

- Similar to Singleton, except instead of just having 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 to be immutable. Why?

```java
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 + ")";
    }
}
```

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

Creational Patterns: Factories

- Factories solve the problem that Java constructors cannot return a subtype of the class they belong to
- Two options:
  - Factory method
    - Helper method creates and returns objects
    - Method defines the interface for creating an object, but defers instantiation to subclasses
  - Factory object
    - Abstract superclass defines what can be customized
    - Concrete subclass does the customization, returns appropriate subclass
    - Object provides the interface for creating families of related/dependent objects without specifying their concrete classes

```java
public class City {
    public Stereotype getStereotypicalPerson() {…}
}
```

```java
public class Seattle extends City {
    @Override
    public Stereotype getStereotypicalPerson() {
        return new SeattleStereotype();
    }
}
```

```java
City seattle = new Seattle();
seattle.getSterotypicalPerson();
```
Creational Patterns: Factory Object

```java
public class City {
    public City(StereotypeFactory f) {
        this.stereotypicalPerson = f.getStereotype();
    }
    public Stereotype getStereotypicalPerson() {
        return f.getStereotype();
    }
}
```

```java
City seattle = new City(new SeattleStereotypeFactory());
seattle.getStereotypicalPerson();
```

Creational Patterns: Factory Object

```java
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();
    }
}
```

Creational Patterns: Builder

```java
public class NutritionFacts {
    private final int servingSize, servings, calories, fat, sodium;
    public static class Builder {
        private final int servingSize, servings;
        private final int calories = 0;
        private final int fat = 0;
        private final 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 sodium(int val) { sodium = val; return this; }
        public NutritionFacts build() { return new NutritionFacts(this); }
    }
    public NutritionFacts(Builder builder) {
        this.servingSize  = builder.servingSize;
        this.servings     = builder.servings;
        this.calories     = builder.calories;
        this.fat          = builder.fat;
        this.sodium       = builder.sodium;
    }
}
```
Creational Patterns: Builder

- Useful when you have many constructor parameters
  - It is hard to remember which order they should all go in
- Easily allows for optional parameters
  - If you have n optional parameters, you need 2^n constructors, but only one builder

Structural Patterns

- Problem: Sometimes difficult to realize relationships between entities
  - Important for code readability
- Solution: Structural patterns!
  - We’re just going to talk about wrappers, which translate between incompatible interfaces

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Functionality</th>
<th>Interface</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>same</td>
<td>different</td>
<td>modify the interface</td>
</tr>
<tr>
<td>Decorator</td>
<td>different</td>
<td>same</td>
<td>extend behavior</td>
</tr>
<tr>
<td>Proxy</td>
<td>same</td>
<td>same</td>
<td>restrict access</td>
</tr>
</tbody>
</table>

Structural Patterns: Adapter

- Changes an interface without changing functionality
  - Rename a method
  - Convert units
- Examples:
  - Angles passed in using radians vs. degrees
  - Bytes vs. strings
  - Hex vs. decimal numbers

Structural Patterns: Decorator

- Adds functionality without changing the interface
  - Add caching
- Adds to existing methods to do something additional while still preserving the previous spec
  - Add logging
- Decorators can remove functionality without changing the interface
  - UnmodifiableList with remove() and put()

Structural Patterns: Proxy

- Wraps the class while maintaining the same interface and functionality
  - Integer vs. int, Boolean vs. boolean
- Controls access to other objects
  - Communication: manage network details when using a remote object
  - Security: permit access only if proper credentials
  - Creation: object might not yet exist because creation is expensive