Object-Oriented Programming with Java

Recitation No. 7: Creational/Sharing Design Patterns and Reference Objects

May 8, 2005

Oranit Dror

Design Patterns

- Known solutions to common problems
- Be aware of tradeoffs
- Patterns that you are familiar with:
  - Factory
  - Iterator
  - Proxy
  - Composite

Creational and Sharing Patterns

- Factory
- Abstract Factory
- Singleton
- Enumeration
- Immutability and Interning
- Flyweight
- Object Pool
- Others…

Factory

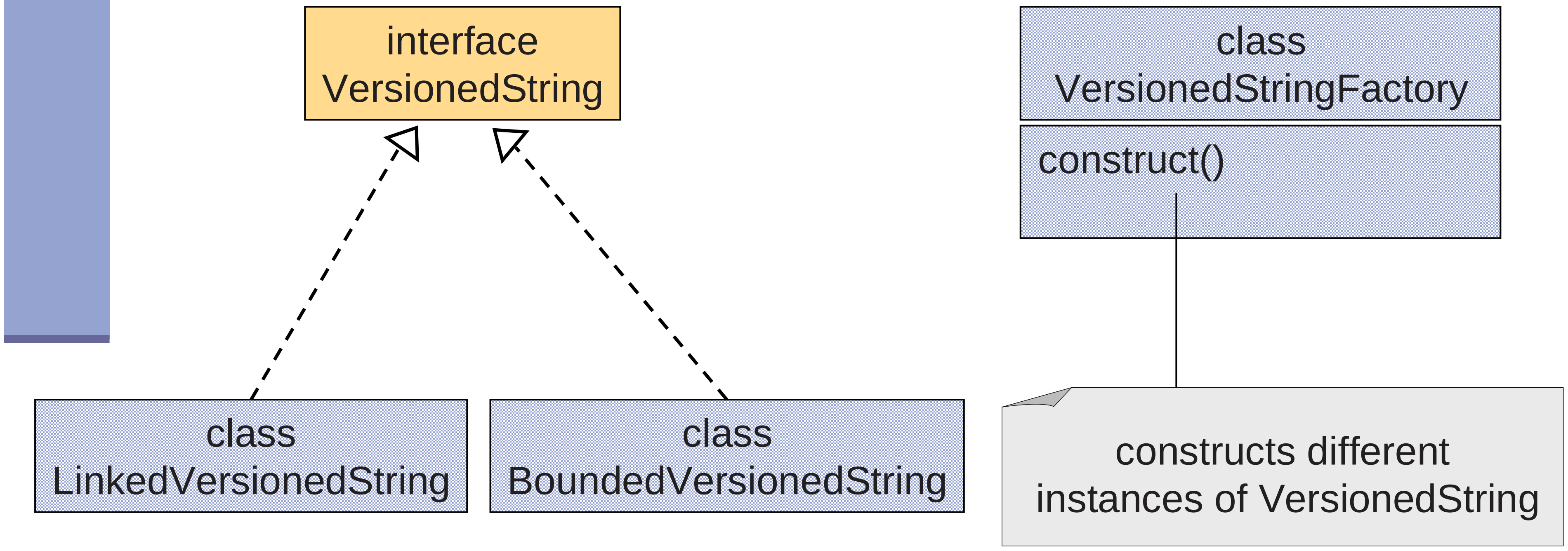
- The new operator gets a class name, (no an interface or abstract class):

  VersionedString vstring = new
  LinkedVersionedString();

- A factory method returns one of several classes with the same interface or super-class

Factory (cont.)

VersionedString vstring =
VersinedStringFactory.construct();



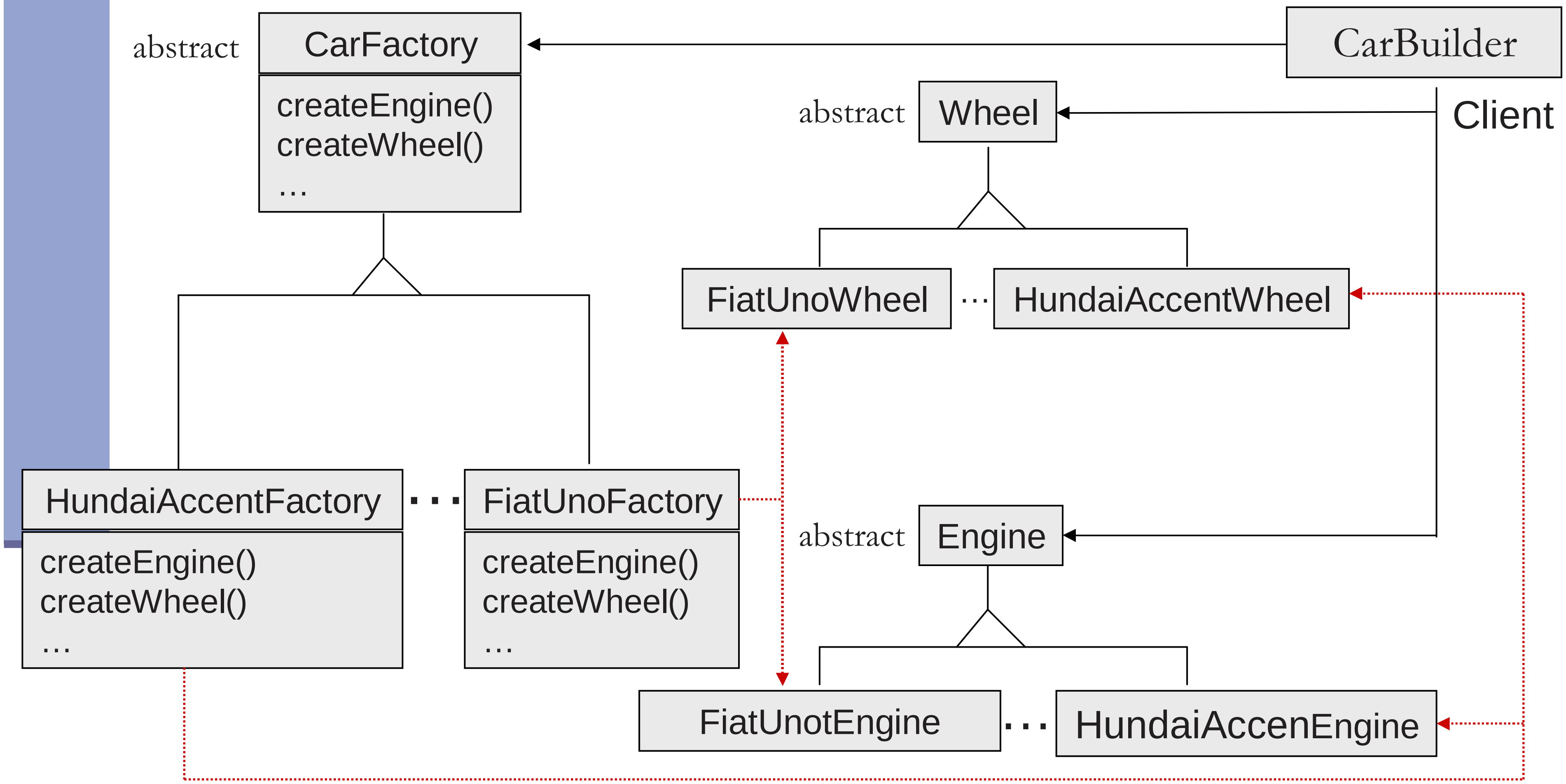
Abstract Factory

- Useful for creating families of related objects without specifying their concrete classes
- Example: An application for building cars
  - builds various types of cars:

    Hundai-Accent, Peuget 205 GTI, Fiat-Uno etc.
  - all cars have the same overall structure, i.e. consist of the same components:

    engine, wheels, brakes etc.
  - The components are different.

Abstract Factory (cont.)



Abstract Factory (cont.)

- Isolates concrete classes
- Exchanging product families is easy
- Promotes consistency among products
- Supporting new kinds of products involves changing the AbstractFactory class and all of its subclasses.
- Typically implemented as a singleton.

Singleton

- Ensures a class has only one instance and provides a global access point to it.

```
public class Logger {
        private static final Logger instance = new Logger();

        private Logger() {…}

        public static Logger getInstance() {
                return instance;
        }
}
```

Singleton (cont.)

```
public class Logger {
    private static Logger instance;

    private Logger() {…}

    public static Logger getInstance() {
        if (instance == null)
            instance = new Logger();

        return instance;
    }
}
```

Lazy evaluation
(not thread-safe)

Enumeration

- Enforces a final set of instances and provides a global access point to them.

```
public final class Boolean … {
        public static final Boolean FALSE  = new Boolean(false);
        public static final Boolean TRUE = new Boolean(true);

        // Constructor
        public Boolean(boolean value) {…}
        // Factory Method
        public static Boolean valueOf(boolean b) {…}
        …
}
```

Enumeration (cont.)

```
public final class Boolean …{
    public static final Boolean FALSE  = new Boolean(false);
    public static final Boolean TRUE = new Boolean(true);

    public Boolean(boolean value) {…}

    static Boolean valueOf(boolean b) {
        return (b ? Boolean.TRUE : Boolean.FALSE);
    }
}
```

private is better

Immutability

- Cannot be changed after creation
- A thread-safe
- Examples: Java Strings, Integers
- All fields are private
- Declared as final
- No methods that change the fields
- A method that changes the attributes should return a new instance:

```
public String String.toUpperCase();
```

Interning

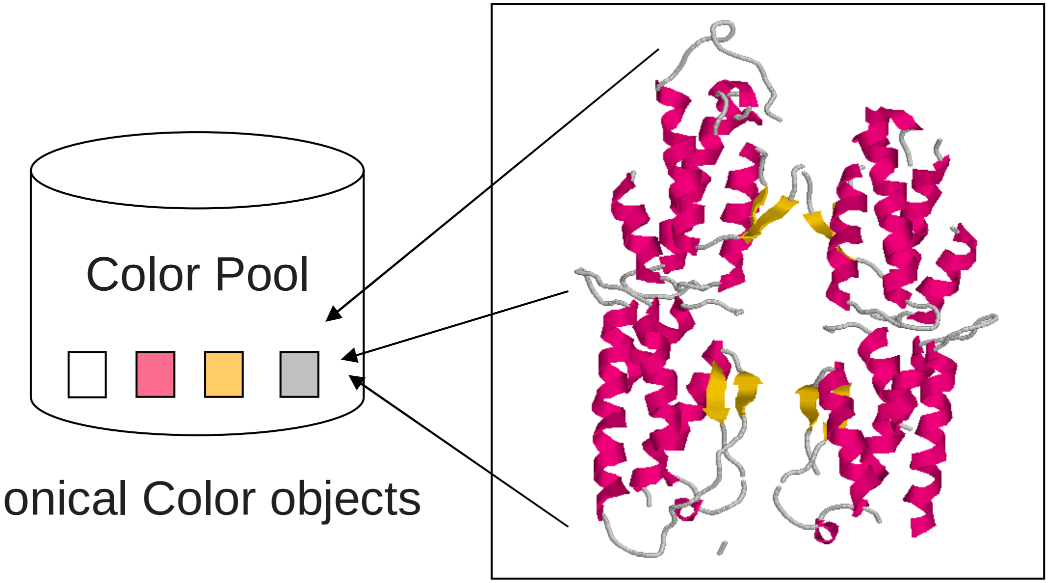
- Reuses existing objects
- Reduces the number of class instances
- Permitted only to immutable objects
- Example:

```
public String String.intern();
```

Interning (cont.)

Example:

Representing an image as an array of pixels, each of which is a color



Canonical Color objects

Many pixels
A few colors

Interning (cont.)

Immutable

```
public final class Color {
    …
    private static Map colors = new HashMap();

    private Color (int rgb) {…}

    public static Color getColor(int rgb) {
        if (colors.containsKey(rgb))
                return (Color) colors.get(rgb);

        Color color = new Color(rgb);
        colors.put(rgb, color);
        return color;
    }
    …
}
```

Factory method

Flyweight

- A generalization of interning
- Reuses existing objects
- Useful when class instances can share most of their fields:
  - Intrinsic fields (can be shared)
  - Extrinsic fields (variable)

Flyweight (cont.)

OO Document Editor Example:

- Use objects to represent documents, pages, lines, tables, images, etc.
- What about representing each character by an object?
  - A flexible representation
  - The naïve design requires huge memory

Flyweight (cont.)

- The naïve design (memory consuming):

```
class Character ... {
    private int x, y;
    private char c;
    private int size;
    private Font font;
    private Color color;
    ...
    draw() {…}
    ...
}
```

extrinsic

intrinsic

Most characters in a document use the same size, font, color etc. Thus, can be shared.

Flyweight (cont.)

- A better design:
  - The class is broken into two classes:
    - a class that holds the intrinsic fields (the flyweight class)
    - The original class holds the extrinsic fields and a reference to the flyweight.
  - The flyweight class is interned

Flyweight (cont.)

The Flyweight class:

```
final class CharacterAttributes {
        private char c;
        private int size;
        private Font font;
        private Color color;
        ...
        draw(int x, int y) {…}
        ...
  }
```

Flyweight (cont.)

The original class:

```
class Character ... {
        private int x, y;
        CharacterAttrbutes attributes;

        Character(int x, int y, char c, int size, Font font, Color color) {
            …
            attributes = CharacterAttributeFactory.construct(c, size, font, color);
        }

        draw() {
            attributes.draw(x,y);
        }
}
```

If possible, it is better to remove these fields

Holds a pool of shared CharacterAttributes objects.

Flyweight (cont.)

- A better approach (if possible):

```
final class Character ... {
            private char c;
            private int size;
            private Font font;
            private Color color;

            private Character();
            ...
            draw(int x, int y) {…}
            ...
    }
```

-Only one class, the original one
-A flyweight class (interned)

Clients should not instantiate the class directly. They must obtain objects from a factory.

The extrinsic fields are supplied by the client

Flyweight (cont.)

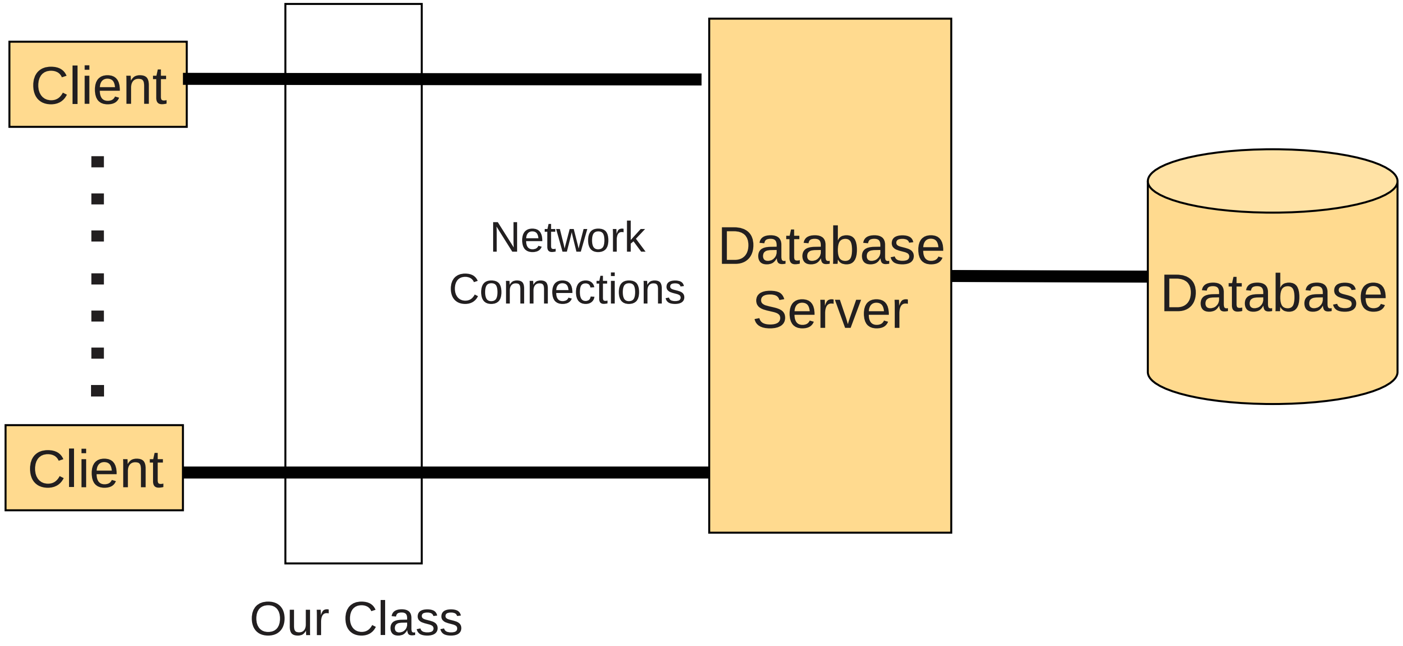
Consequences:

- may introduce run-time costs
- Storage saving is a function of:
  - the reduction in the total number of instances
  - the amount of intrinsic state per object
  - whether extrinsic state is computed or stored

Object Pool

Database Example:

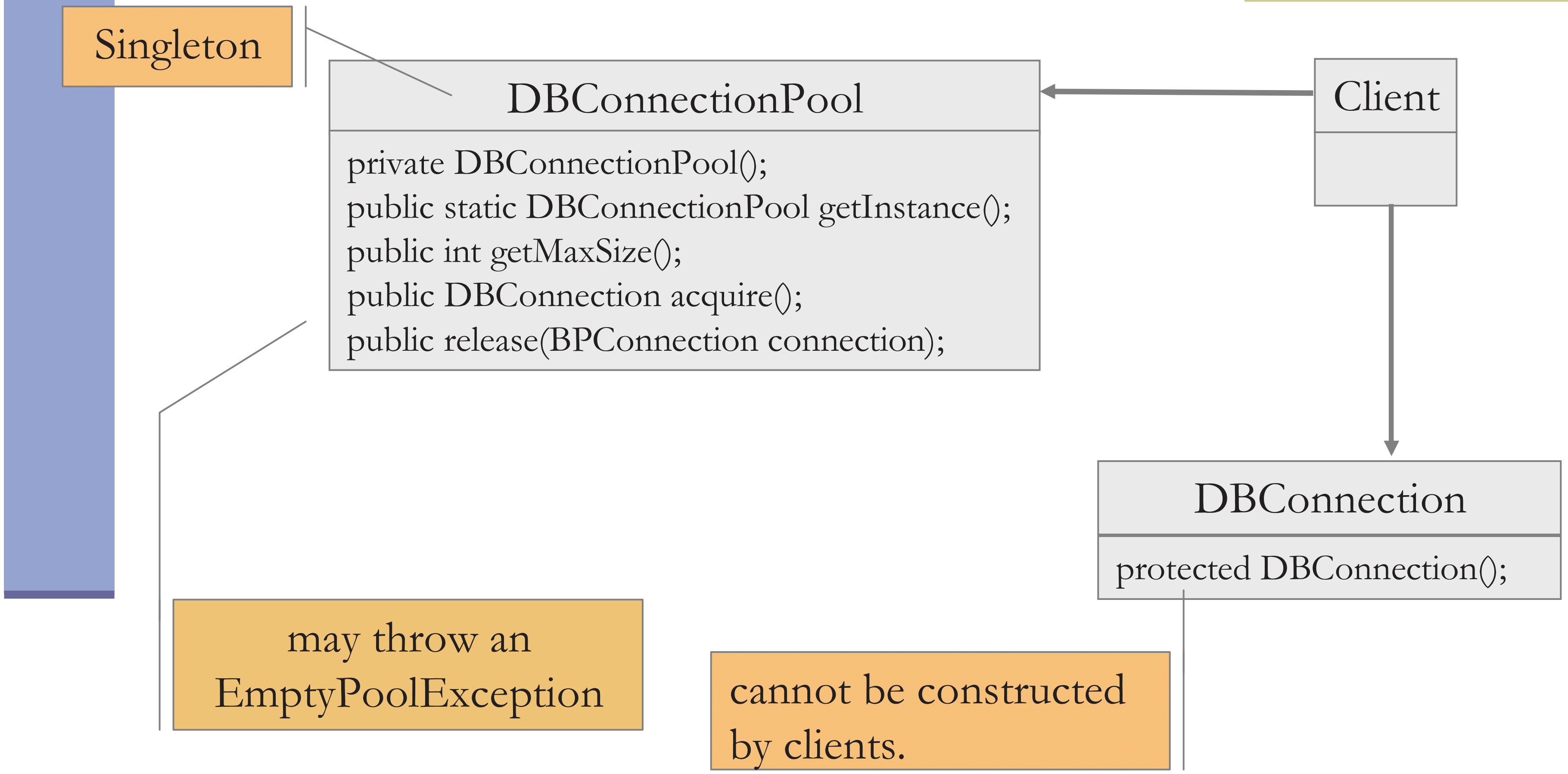
- Task: Design a class for accessing a DB



Object Pool (cont.)

- Constraints:
  - Establishing and cleaning up connections to a database are time-consuming
  - Connecting/Disconnecting time may depend on the number of open connections.
  - The number of open connections may be limited (server capacity, DB license)
- Solution:
  - Maintain a pool of open connections for reuse

Object Pool (cont.)



Singleton

DBConnectionPool

private DBConnectionPool();
public static DBConnectionPool getInstance();
public int getMaxSize();
public DBConnection acquire();
public release(BPConnection connection);

Client

DBConnection

protected DBConnection();

may throw an EmptyPoolException

cannot be constructed by clients.

Reference Objects

- Consider the following case:
  - we have an unlimited pool of DB connections
  - we may end up in an out of memory situation
- To overcome this problem:
  - The pool will use soft references to hold DB connections
  - Unused connections will be cleared by the garbage collector if memory is required.

Reference Objects (cont.)

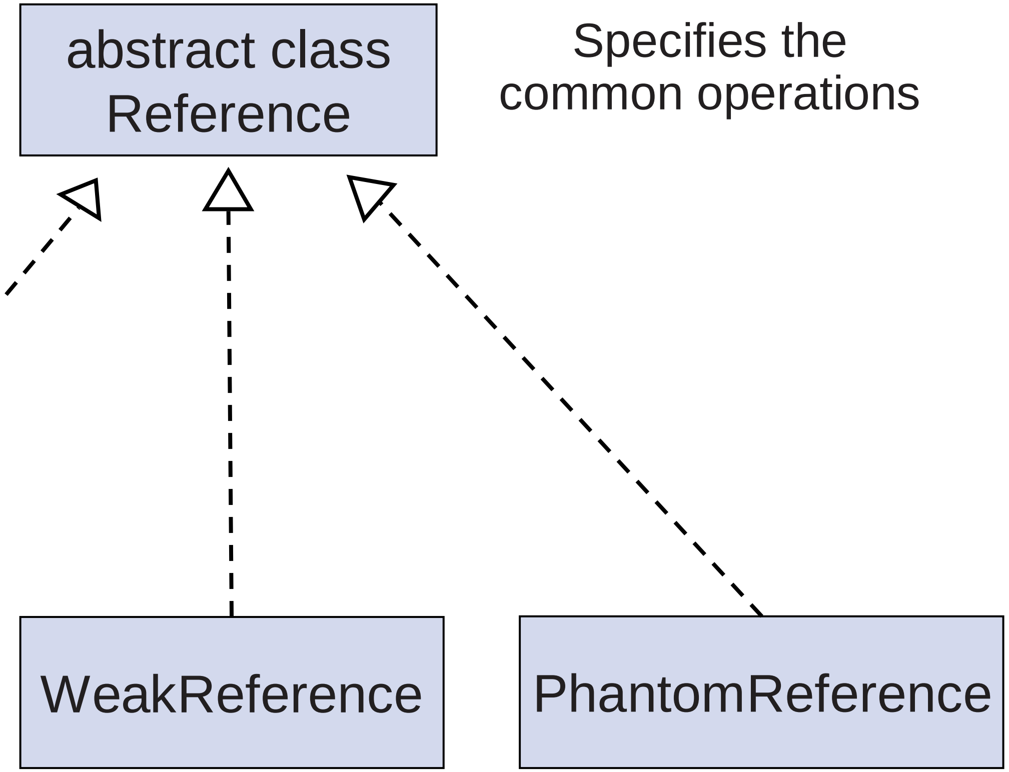
- Specified in the java.lang.ref package
- Provide special references to objects for a limited interaction with the garbage collector.
- Four types of references to objects:
  - Regular references
  - Soft references
  - Weak references
  - Phantom references

Strength Level

Specified by Reference objects

Reference Objects (cont.)

- Class Hierarchy:



abstract class Reference

Specifies the common operations

SoftRefernece

WeakReference

PhantomReference

Reference Objects (cont.)

Object Type	When garbage-collected
Strongly reachable	Never
Softly reachable	If memory is tight
Weakly reachable	Automatically
Phantom reachable	After finalization

Reference Objects (cont.)

Reference Object	Useful for…
SoftReference	memory-safe caches
WeakReference	canonicalizing mappings
PhantomReference	scheduling pre-mortem cleanup

Reference Objects (cont.)

- Usage Example:

```
DBConnection connection = new DBConnection();
SoftReference connectionRef = new SofReference(connection);
```

```
connection = (DBConnection) (connectionRef.get());
if (connection == null) {
  connection = new DBConnection();
  connectionRef = new SoftReference(connection);
}
```

Books

- The Gang of Four (GoF) book:

  Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements Of Reusable Object-Oriented Software. 1995.