CREATIONAL PATTERNS

CSCI 4448/5448: OBJECT-ORIENTED ANALYSIS & DESIGN

LECTURE 25 — 11/14/2011

1

Goals of the Lecture

- Cover material from Chapters 20-22 of the Textbook
  - Lessons from Design Patterns: Factories
  - Singleton Pattern
  - Object Pool Pattern

Pattern Classification

- The Gang of Four classified patterns in three ways

  - The behavioral patterns are used to manage variation in behaviors (think Strategy pattern)

  - The structural patterns are useful to integrate existing code into new object-oriented designs (think Bridge)

  - The creational patterns are used to create objects

    - **Abstract Factory**, **Builder**, **Factory Method**, Prototype & **Singleton**

Factories & Their Role in OO Design

- It is important to manage the creation of objects

  - Code that mixes **object creation** with **the use of objects** can become quickly non-cohesive

  - A system may have to deal with a variety of different contexts with each context requiring a different set of objects

    - In design patterns, the context determines which concrete implementations need to be present

Factories & Their Role in OO Design

- The code to determine the current context, and thus which objects to instantiate, can become complex

  - with many different conditional statements

- If you mix this type of code with the use of the instantiated objects, your code becomes cluttered

  - **often the use scenarios can happen in a few lines of code**; if combined with creational code, the operational code gets buried behind the creational code

Factories provide Cohesion

- The use of factories can address these issues
  - The conditional code can be hidden within them
    - pass in the parameters associated with the current context
    - and get back the objects you need for the situation
  - Then use those objects to get your work done
- Factories concern themselves just with creation, letting your code focus on other things

The Object Creation/Management Rule

- This discussion brings us to this general design rule

  - An object should either create/manage other objects

- OR

  - it should use other objects
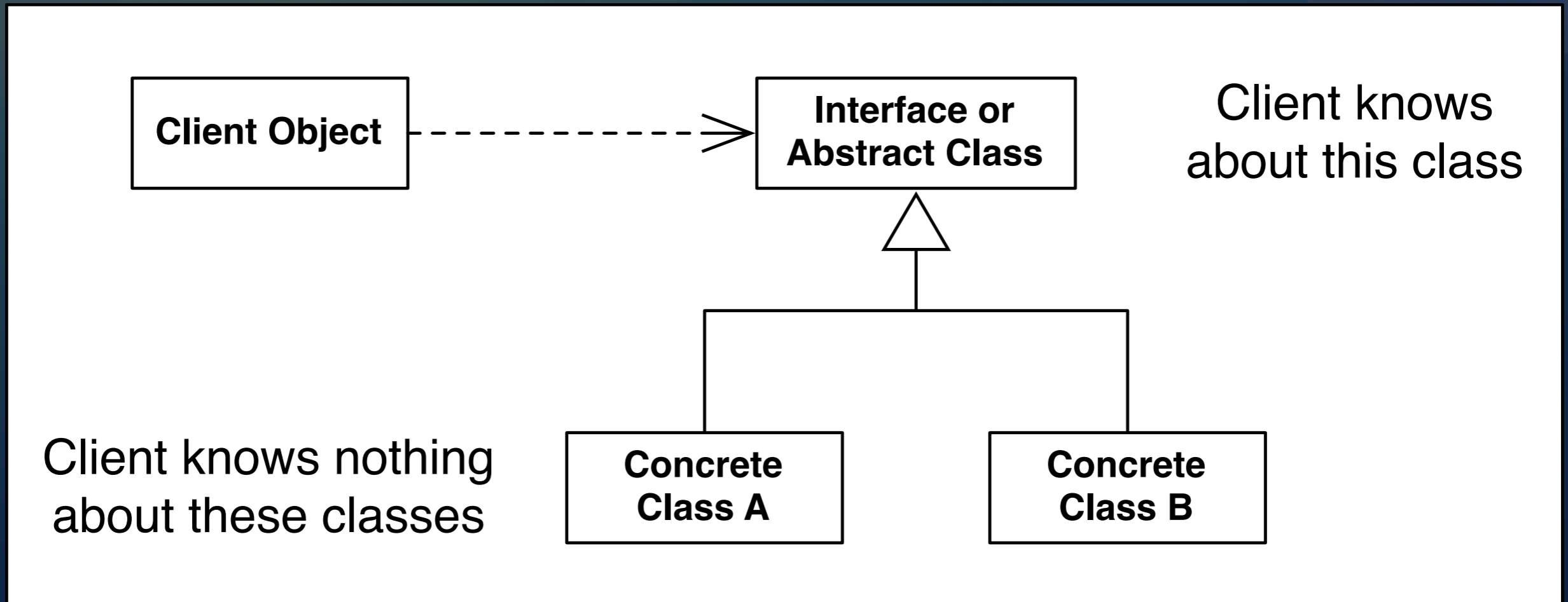
- BUT

  - it should never do both

Discussion (I)

- This rule is a guideline not an absolute

  - The latter is too difficult; think of iOS view controllers

    - They exist to create a view and then respond to requests related to that view (which may involve making queries on the view)

      - This violates the rule, strictly speaking, as it both creates AND uses its associated view

      - But it is primarily a creational pattern within iOS
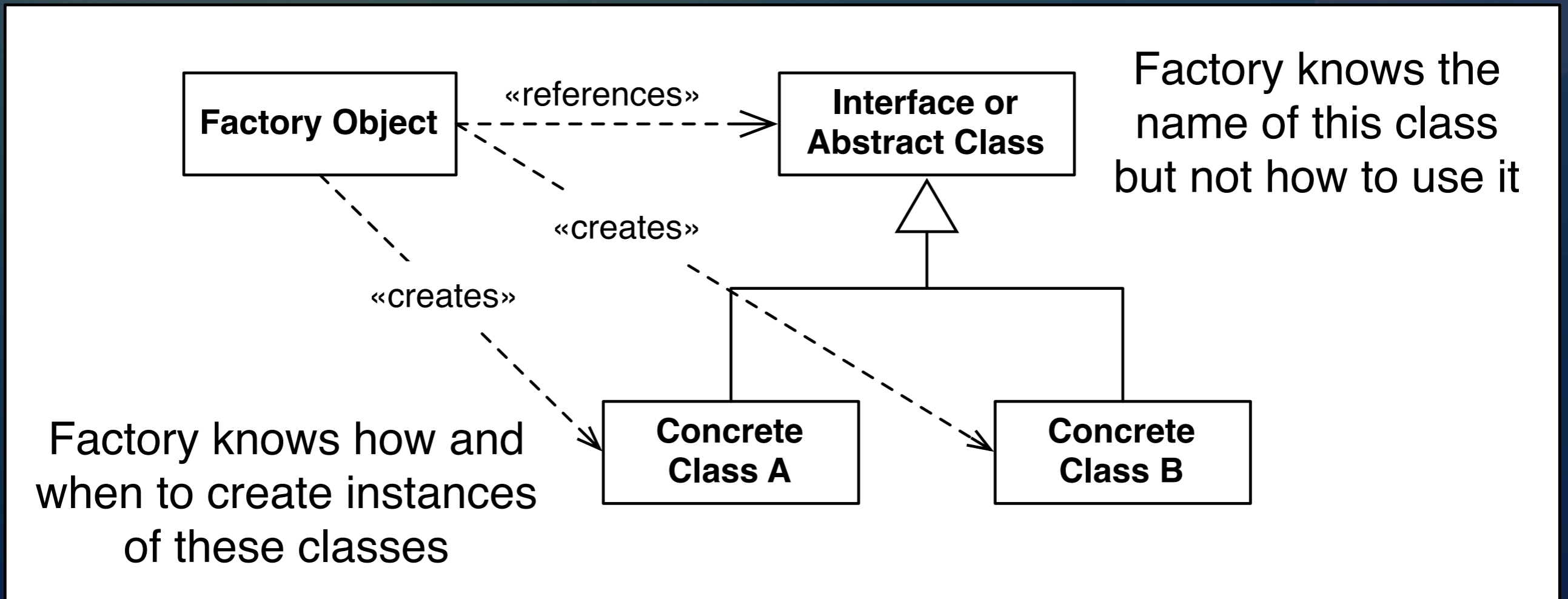
Discussion (II)

- But as a guideline, the rule is useful

  - Look for ways to separate out the creation of objects from the code that make use of those objects

    - encapsulate the creation process and you can change it as needed without impacting the code that then uses those objects

  - The book demonstrates the advantages of the rule with the following two diagrams

The perspective of a client object

Client Object - - - - - > Interface or Abstract Class

Client knows about this class

Client knows nothing about these classes

Concrete Class A

Concrete Class B

The client is completely shielded from the concrete classes and only changes if the abstract interface changes

The perspective of a factory object

Factory Object ---«references»--> Interface or Abstract Class

Factory knows the name of this class but not how to use it

«creates»

«creates»

Factory knows how and when to create instances of these classes

Concrete Class A

Concrete Class B

The factory knows nothing about how to use the abstract interface; it just creates the objects that implement it

Factories help to limit change

- If a change request relates to the creation of an object, the change will likely occur in a factory

    - all client code will remain unaffected

- If a change request does not relate to the creation of objects, the change will likely occur in the use of an object or the features it provides

    - your factories can be ignored as you work to implement the change

Abstract Factory and Factory Method

- We've already seen several factory patterns

    - Factory Method: Pizza and Pizza Store example

        - Have client code use an abstract method that returns a needed instance of an interface

        - Have a subclass implementation determine the concrete implementation that is returned

    - Abstract Factory: Pizza Ingredients Example

        - Pattern that creates groups of related objects

Singleton Pattern

- Used to ensure that only one instance of a particular class ever gets created and that there is just one (global) way to gain access to that instance

- Let's derive this pattern by starting with a class that has no restrictions on who can create it

Deriving Singleton (I)

- public class Ball {

  - private String color;

  - public Ball(String color) { this.color = color; }

  - public void bounce() { System.out.println("boing!"); }

- }

- Ball b1 = new Ball("red"); Ball b2 = new Ball("green");

- b1.bounce(); b2.bounce();

Problem: Universal Instantiation

- As long as a client object "knows about" the name of the class Ball, it can create instances of Ball

  - Ball b1 = new Ball("orange");

- This is because the constructor is public. We can stop unauthorized creation of Ball instances by making the constructor private

Deriving Singleton (II)

- public class Ball {

  - private String color;

  - **private** Ball(String color) { this.color = color; }

  - public void bounce() { System.out.println("boing!"); }

- }

- // next line now impossible by any method outside of Ball

- Ball b2 = new Ball("red");

Problem: No Point of Access!

- Now that the constructor is private, no class can gain access to instances of Ball

  - But our requirements were that there would be at least one way to get access to an instance of Ball

- We need a method to return an instance of Ball

  - But since there is no way to get access to an instance of Ball, the method can NOT be an **instance** method

    - This means it needs to be a **class** method, aka **static**

Deriving Singleton (III)

- public class Ball {

  - private String color;

  - **private** Ball(String color) { this.color = color; }

  - public void bounce() { System.out.println("boing!"); }

  - **public static Ball getInstance(String color) {**

    - **return new Ball(color);**

  - **}**

- }

Problem: Back to Universal Instantiation

- We are back to the problem where any client can create an instance of Ball; instead of saying this:

  - Ball b1 = new Ball("blue");

- they just say

  - Ball b1 = Ball.getInstance("blue");

- Need to ensure only one instance is ever created

  - Need a static variable to store that instance

    - No instance variables are available in static methods

Deriving Singleton (IV)

- public class Ball {

    - **private static Ball ball;**

    - private String color;

    - **private** Ball(String color) { this.color = color; }

    - public void bounce() { System.out.println("boing!"); }

    - **public static Ball getInstance(String color) {**

        - **return ball;**

    - **}**

- }

Problem: No instance!

- Now the getInstance method returns null each time it is called

  - Need to check instance variable to see if it is null

    - If so, create an instance

  - Otherwise return the single instance

Deriving Singleton (V)

- public class Ball {

    - **private static Ball ball;**

    - **private** Ball(String color) { this.color = color; }

    - **public static Ball getInstance(String color) {**

        - **if (ball == null) { ball = new Ball(color); }**

        - **return ball;**

    - **}**

- }

Problem: First Parameter Wins

- The previous code shows the Singleton pattern

  - private constructor

  - private static instance variable to store instance

  - public static method to gain access to instance

    - creates object if needed; returns it

- But this code ignores the fact that a parameter is being passed in; so if a "red" ball is created all subsequent requests for a "green" ball are ignored

Solution: Use a Map

- The solution to the final problem is to change the private static instance variable to a Map

    - private Map<String, Ball> toybox = new HashMap…

- Then check if the map contains an instance for a given value of the parameter

    - this ensures that only one ball of a given color is ever created

    - this is an acceptable variation of the Singleton pattern

- **DEMO**

Singleton Pattern: Structure

Singleton
static my_instance : Singleton
static getInstance() : Singleton private Singleton()

Singleton involves only a single class (not typically called Singleton). That class is a full-fledged class with other attributes and methods (not shown)

The class has a static variable that points at a single instance of the class.

The class has a private constructor (to prevent other code from instantiating the class) and a static method that provides access to the single instance

World's Smallest Java-based Singleton Class

```java
 1  public class Singleton {
 2
 3        private static Singleton uniqueInstance;
 4
 5        private Singleton() {}
 6
 7        public static Singleton getInstance() {
 8                if (uniqueInstance == null) {
 9                        uniqueInstance = new Singleton();
10                }
11                return uniqueInstance;
12        }
13  }
14
```

Meets Requirements: static var, static method, private constructor

Thread Safe?

- The Java code just shown is not thread safe

    - This means that it is possible for two threads to attempt to create the singleton for the first time simultaneously

    - If both threads check to see if the static variable is empty at the same time, they will both proceed to creating an instance and you will end up with two instances of the singleton object (not good!)

        - Example Next Slide

Program to Test Thread Safety

```java
public class Creator implements Runnable {

    private int id;

    public Creator(int id) {
        this.id = id;
    }

    public void run() {
        try {
            Thread.sleep(200L);
        } catch (Exception e) {
        }
        Singleton s = Singleton.getInstance();
        System.out.println("s" + id + " = " + s);
    }

    public static void main(String[] args) {
        Thread[] creators = new Thread[10];
        for (int i = 0; i < 10; i++) {
            creators[i] = new Thread(new Creator(i));
        }
        for (int i = 0; i < 10; i++) {
            creators[i].start();
        }
    }

}
```

Creates a "runnable" object that can be assigned to a thread.

When its run, its sleeps for a short time, gets an instance of the Singleton, and prints out its object id.

The main routine, creates ten runnable objects, assigns them to ten threads and starts each of the threads

Output for Non Thread-Safe Singleton Code

- s9 = Singleton@45d068
- s8 = Singleton@45d068
- s3 = Singleton@45d068
- s6 = Singleton@45d068
- s1 = Singleton@45d068
- s0 = Singleton@ab50cd
- s5 = Singleton@45d068
- s4 = Singleton@45d068
- s7 = Singleton@45d068
- s2 = Singleton@45d068

Whoops!

Thread 0 created an instance of the Singleton class at memory location ab50cd at the same time that another thread (we don't know which one) created an additional instance of Singleton at memory location 45d068!

How to Fix?

```java
1  public class Singleton {
2
3          private static Singleton uniqueInstance;
4
5          private Singleton() {}
6
7          public static synchronized Singleton getInstance() {
8                  if (uniqueInstance == null) {
9                          uniqueInstance = new Singleton();
10                 }
11                 return uniqueInstance;
12         }
13
14 }
15
```

In Java, the easiest fix is to add the **synchronized** keyword to the getInstance() method.

Object Pool

- A variant of the Singleton Pattern is known as the Object Pool pattern

    - This allows some instances (x) of an object to be created up to some maximum number (m)

        - where $1 < x \leq m$

    - In this case, each instance provides a reusable service (typically tied to system resources such as network connections, printers, etc.) and clients don't care which instance they receive

Object Pool Structure Diagram
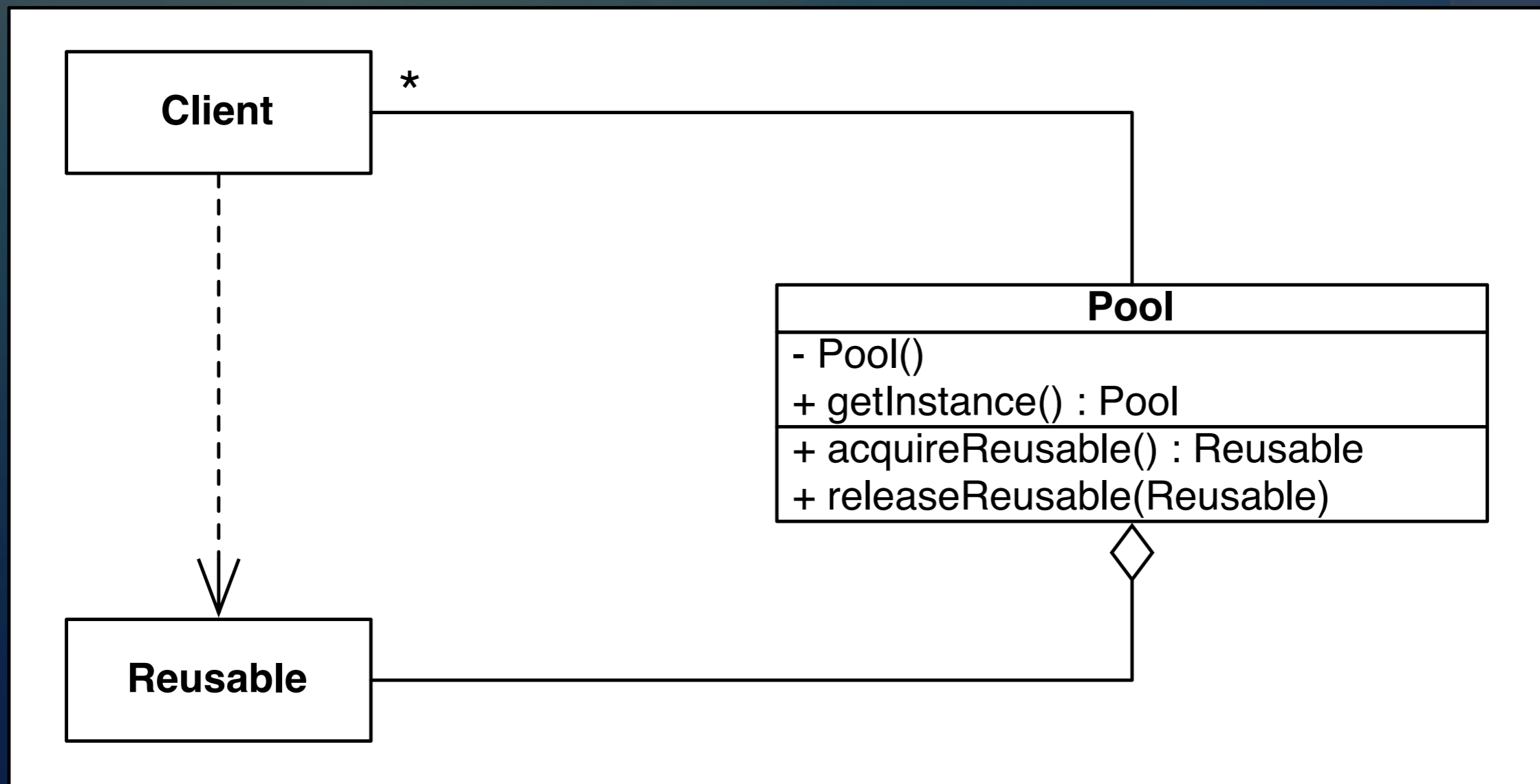
Common Use: Thread Pool

- One place in which the Object Pool pattern is used frequently is in multithreaded applications

    - where each thread is a "computation resource" that can be used to execute one or more tasks associated with the system

    - when a task needs to be done, a thread is pulled out of the pool and assigned the task; it executes the task and is then released back to the pool to (eventually) work on other tasks

Examples (1)

- First, a "homegrown" example

  - ThreadPool implemented as a blocking queue of Thread subclasses called Processors

  - Goal is to calculate the number of primes between 1 and 20M (20,000,000).

  - Producer creates tasks to calculate primes between a subset of numbers, say 1 to 250,000; 250,001 to …

  - Processor calculates in separate thread

  - Consumer joins with Processors and merges the results

Examples (2)

- An example that makes use of a thread pool provided by the Java Concurrency API (<u>example taken from this excellent book</u>)

  - The class that implements the object pool pattern is known as an ExecutorService

    - You pass it instances of a class called Callable

    - It returns instances of a class called Future

      - You hold onto the Future object while Callable executes in the background

        - then retrieve Callable's result from the Future object

Wrapping Up

- Looked at

  - the use of Factories in OO Design

  - the Singleton and Object Pool Design Patterns

    - saw example of thread-safe singletons

    - saw use of thread pools in java.util.concurrent

Coming Up Next

- Lecture 25: More Patterns

- Lecture 26: Textbook Wrap Up

- Homework 6 Due Friday