

CSE201: Monsoon 2020
Advanced Programming

Lecture 22: Introduction to Design Patterns

Vivek Kumar

Computer Science and Engineering

IIT Delhi

vivekk@iiitd.ac.in

Today's Lecture

- One remaining topic in multithreading
 - Deadlocks
- Introduction to design patterns
 - Iterator
 - Singleton
 - Flyweight
 - (Acknowledgement: CSE331, University of Washington)



DEADLOCK

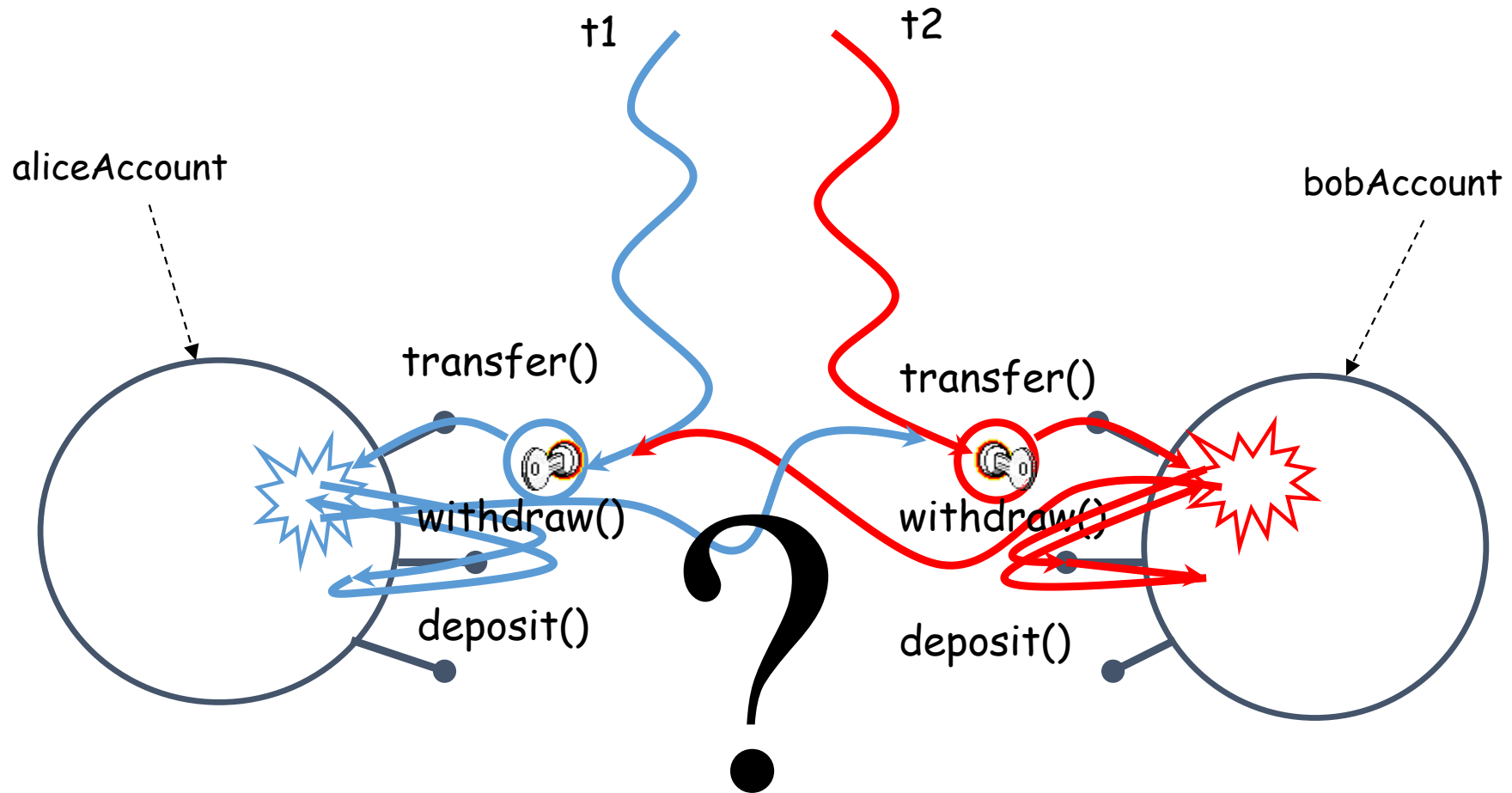
Let's Code a **D**eadlock

```
public class BankAccount {  
    private volatile float balance;  
    public synchronized void deposit(float amount) {  
        balance += amount;  
    }  
    public synchronized void withdraw(float amount) {  
        balance -= amount;  
    }  
    public synchronized void transfer(float amount,  
                                       BankAccount target) {  
        withdraw(amount);  
        target.deposit(amount);  
    }  
}
```

```
public class MoneyTransfer implements Runnable {  
    private BankAccount source, target;  
    private float amount;  
    public MoneyTransfer(BankAccount from,  
                        BankAccount to, float amount) {  
        this.source = from;  
        this.target = to;  
        this.amount = amount;  
    }  
    public void run() {  
        source.transfer(amount, target);  
    }  
}
```

```
BankAccount aliceAccount = new BankAccount();  
BankAccount bobAccount = new BankAccount();  
...  
// At one place  
Runnable transaction1 = new MoneyTransfer(aliceAccount, bobAccount, 1200);  
Thread t1 = new Thread(transaction1);  
t1.start();  
  
// At another place  
Runnable transaction2 = new MoneyTransfer(bobAccount, aliceAccount, 700);  
Thread t2 = new Thread(transaction2);  
t2.start();
```

Let's Analyze Our Bank Transaction



Deadlock Avoidance

- **Deadlock occurs when multiple threads need the same set of locks but obtain them in different order**
- Not so easy to avoid deadlocks
- It's an active research area

Let's try simple remedies to fix
our Bank Transaction program

Deadlock Avoidance

- Lock ordering

- Ensure that all locks are taken in same order by any thread

- Lock timeout

- Put a timeout on lock attempts
 - Not possible with monitor locks
 - You will need `java.util.concurrent.ReentrantLock`

Now Let's Resolve the Deadlock

```
public class BankAccount {  
    private volatile float balance;  
    final int account_id;  
  
    public BankAccount(int i) { account_id = i; }  
  
    public synchronized void deposit(float amount) {  
        balance += amount;  
    }  
  
    public synchronized void withdraw(float amount) {  
        balance -= amount;  
    }  
  
    public synchronized void transfer(float amount,  
                                      BankAccount target) {  
        withdraw(amount);  
        target.deposit(amount);  
    }  
}
```

```
public class MoneyTransfer implements Runnable {  
    private BankAccount source, target;  
    private float amount;  
    public MoneyTransfer(BankAccount from,  
                        BankAccount to, float amount) {  
        this.source = from;  
        this.target = to;  
        this.amount = amount;  
    }  
    public void run() {  
        Object obj1 = null, obj2 = null;  
        if(source.account_id > target.account_id) {  
            obj1=target; obj2=source;  
        }  
        else { obj1=source; obj2=target; }  
        synchronized(obj1) { synchronized(obj2) {  
            source.transfer(amount, target);  
        } }  
    }  
}
```

```
BankAccount aliceAccount = new BankAccount(1); // account_id = 1;  
BankAccount bobAccount = new BankAccount(2);  // account_id = 2;  
...  
// At one place  
Runnable transaction1 = new MoneyTransfer(aliceAccount, bobAccount, 1200);  
Thread t1 = new Thread(transaction1);  
t1.start();  
  
// At another place  
Runnable transaction2 = new MoneyTransfer(bobAccount, aliceAccount, 700);  
Thread t2 = new Thread(transaction2);  
t2.start();
```

- We are using lock ordering technique here to resolve the deadlock
- Lock on BankAccount objects are taken in run() method as per the ascending order value of the account_id
 - Recall monitor locks are reentrant



Let's change gears...

Design Patterns

What is Design Pattern

- It is a solution for a repeatable problem in the software design
- This is not a complete design for a software system that can be directly transformed into code
- It is a description or **template** for how to solve the problem that can be used in many different situations

Why Study Patterns

- Reuse tried, proven solutions
 - Provides a head start
 - Avoids gotchas later (unanticipated things)
 - No need to reinvent the wheel
- Establish common terminology
 - Design patterns provide a common point of reference
 - Easier to say, “We could use Strategy here.”
- Provide a higher level prospective
 - Frees us from dealing with the details too early

“GoF” (Gang of Four) patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
 - Factory Method Abstract Factory Singleton
 - Builder Prototype
- **Structural Patterns** *(how objects/classes can be combined)*
 - Adapter Bridge Composite
 - Decorator Facade Flyweight
 - Proxy
- **Behavioral Patterns** *(communication between objects)*
 - Command Interpreter Iterator
 - Mediator Observer State
 - Strategy Chain of Responsibility Visitor
 - Template Method

In 1990 a group called the Gang of Four or "GoF" (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns in the book "Design Patterns: Elements of Reusable Object-Oriented Software"

Pattern: Iterator

objects that traverse collections

Pattern: Iterator

- Recurring Problem

- How can you loop over all objects in any collection. You don't want to change client code when the collection changes. Want the same methods

- Solution

1. Provide a standard *iterator* object supplied by all data structures
2. The implementation performs traversals, does bookkeeping
3. The implementation has knowledge about the representation
4. Results are communicated to clients via a standard interface

- Consequences

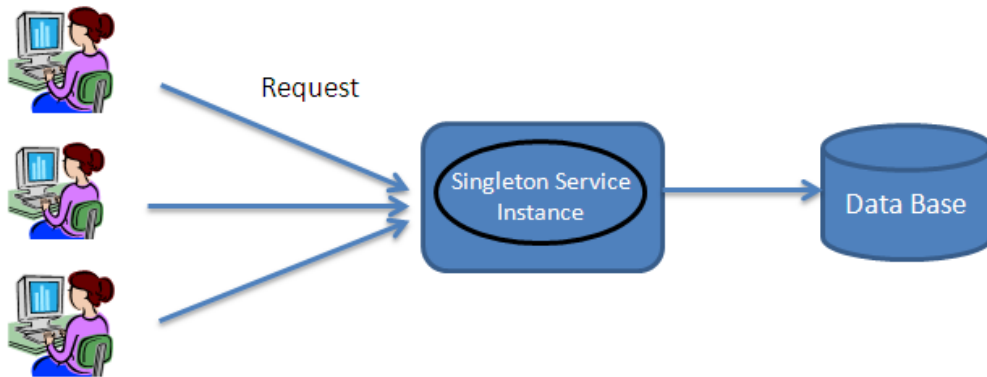
- Can change collection class details without changing code to traverse the collection

Pattern: Singleton

A class that has only a single instance

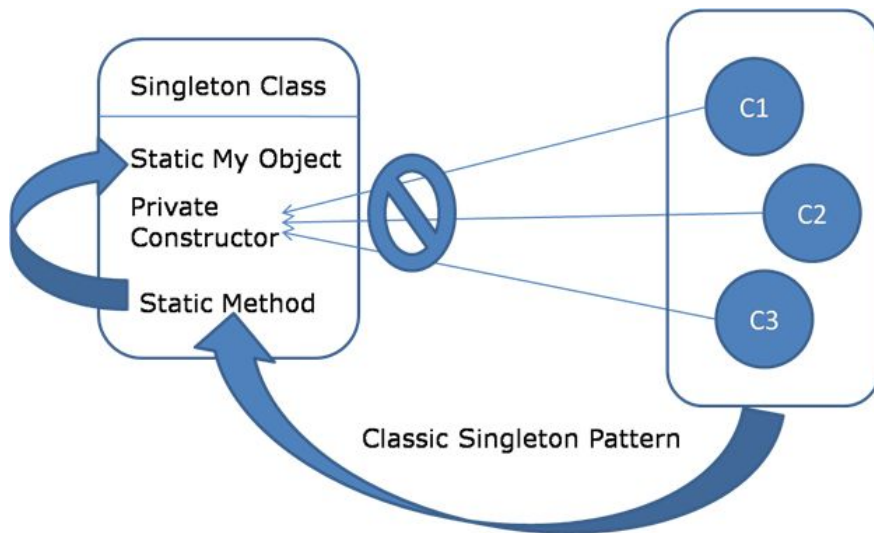


Pattern: Singleton



- Recurring problem
 - Sometimes we only ever need one instance of a particular class
 - It should be illegal to have another instance of the same class
- Solution
 - Singleton pattern – ensuring that a class has at most one instance
 - Providing global access to that instance

Implementing Singleton



1. Make constructor private so that no client is able to call it from outside
2. Declare a single private static instance of the class
3. Write a getInstance() method (or similar) that allows access to the single instance
 - Ensure thread safety in case multiple threads can access this method

Singleton Example

```
public class RandomGenerator {  
    private static RandomGenerator gen = null;  
    public static RandomGenerator getInstance()  
    {  
        if (gen == null) {  
            gen = new RandomGenerator();  
        }  
        return gen;  
    }  
    private RandomGenerator() {}  
    ...  
}
```

- Creates a new random generator
- Clients will not use the constructor directly but will instead call getInstance to obtain a RandomGenerator object that is shared by all classes in the application
- Lazy initialization
 - Can wait until client asks for the instance to create it
 - How to ensure thread safety?

Singleton Comparator

```
public class LengthComparator
    implements Comparator<String> {
    private static LengthComparator comp = null;
    public static LengthComparator getInstance()
    {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }
    private LengthComparator() {}
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

- Comparators make great singletons because they have no state
- Saves memory by not allowing the creation of more than one object

Pattern: Flyweight

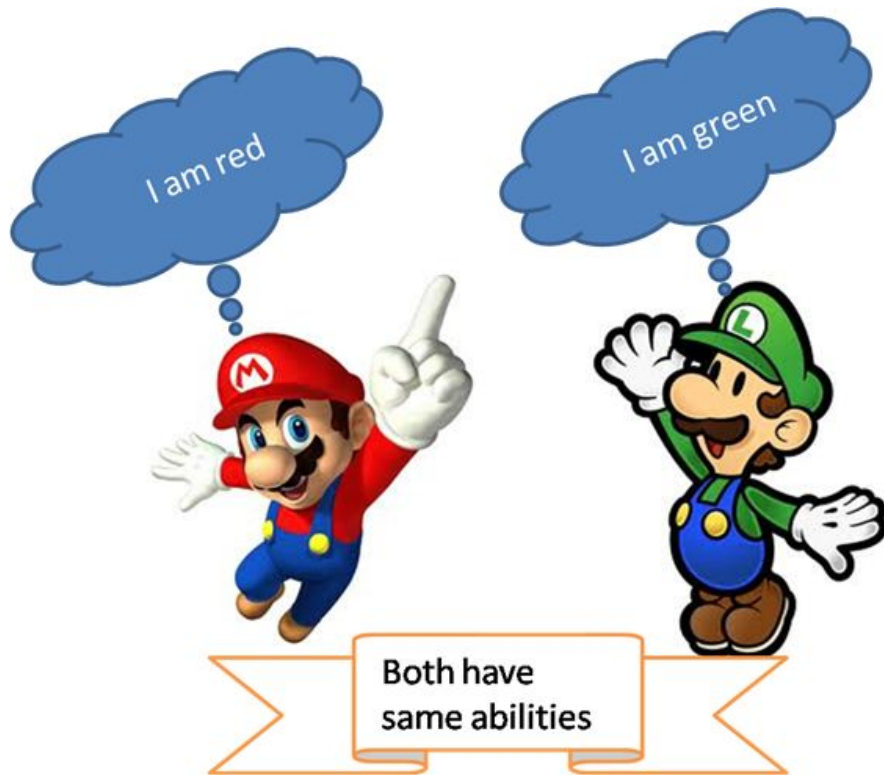
*a class that has only one instance for
each unique state*

Pattern: Flyweight

- Problem

- **Redundant objects** can bog down the system
 - Many objects have the same state
- Example: File objects that represent the same file on disk
 - `new File("chatlog.txt")`
 - `new File("chatlog.txt")`
 - `new File("chatlog.txt")`
 - ...
 - `new File("notes.txt")`
- Example: Date objects that represent the same date of the year
 - `new Date(4, 18)`
 - `new Date(4, 18)`

Pattern: Flyweight



- An assurance that no more than one instance of a class will have identical state
 - Achieved by caching identical instances of objects.
 - Similar to singleton, but one instance for each unique object state
 - Useful when there are many instances, but many are equivalent

Implementing a Flyweight (1/2)

```
public class Flyweightd {  
    private static Map<KeyType, Flyweightd> instances  
        = new HashMap<KeyType, Flyweightd>();  
  
    private Flyweightd(...) { ... }  
  
    public static Flyweightd getInstance(KeyType key) {  
        if (!instances.contains(key)) {  
            instances.put(key, new Flyweightd(key));  
        }  
        return instances.get(key);  
    }  
}
```

Implementing a Flyweight (2/2)

```
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; }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```



```
public class Point {

    private static Map<String, Point> instances =
        new HashMap<String, Point>();

    public static Point getInstance(int x, int y)
    {
        String key = x + ", " + y;
        if (!instances.containsKey(key)) {
            instances.put(key, new Point(x, y));
        }
        return instances.get(key);
    }

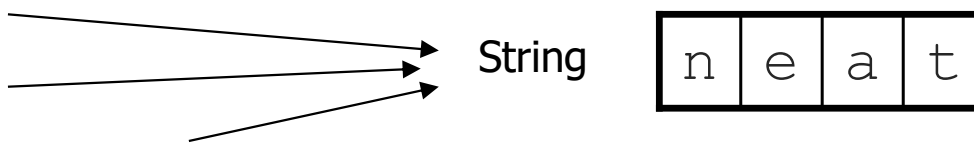
    private final int x, y; // immutable
    private Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

Flyweighting in String by JVM

- The possible combinations for Strings is close to infinite, hence JVM maintains a cache for strings, called the **string constant pool**
 - It is empty at startup and is filled constantly during the lifecycle of the JVM
- Java String objects are automatically flyweighted by the JVM **whenever** possible
 - If you declare two string variables that point to the same literal.
 - If you concatenate two string literals to match another literal

```
String a = "neat";  
String b = "neat";  
String c = "n" +  
"eat";
```



String

n	e	a	t
---	---	---	---

Next Lecture

- More design patterns