CS 619 Introduction to OO Design and Development

GoF Patterns (Part 4)

Fall 2014

Restricting Object Creation

- problem: sometimes we will really only ever need one instance of a particular class
  - examples: keyboard reader, bank data collection, printer spooler
  - we'd like to make it illegal to have more than one, just for safety's sake
- why we care:
  - creating lots of objects can take a lot of time
  - extra objects take up memory
  - it is a pain to deal with different objects floating around if they are essentially the same

Singleton example 1

- consider a singleton class RandomGenerator that generates random numbers

```
public class RandomGenerator {
  private static RandomGenerator gen = new
  RandomGenerator();

  public static RandomGenerator getInstance() {
    return gen;
  }

  private RandomGenerator() {}

  public double nextNumber() {
    return Math.random();
  }
}
```

Singleton example 2

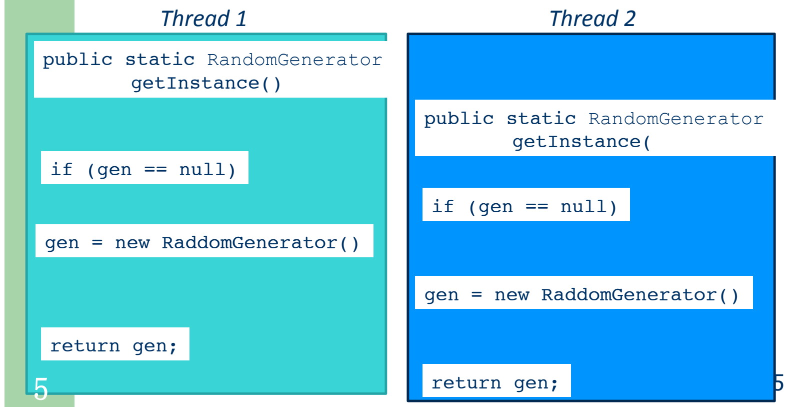
- variation: don't create the instance until needed

```
// Generates random numbers.
public class RandomGenerator {
  private static RandomGenerator gen = null;

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

- What could go wrong with this version?

Thread Example

A dual processor machine, with two threads calling the *getInstance()* method for RandomGenerator



Thread 1

```
public static RandomGenerator
        getInstance()

if (gen == null)

gen = new RaddomGenerator()

return gen;
```

Thread 2

```
public static RandomGenerator
        getInstance(

if (gen == null)

gen = new RaddomGenerator()

return gen;
```

5

Singleton example 3

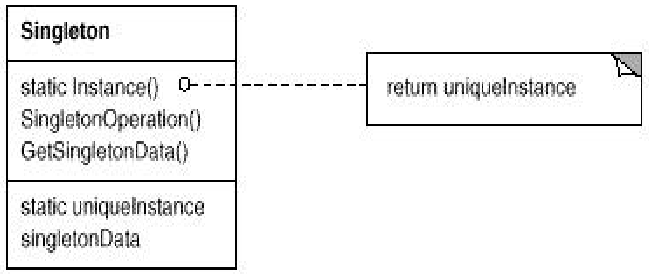
- variation: solve concurrency issue by locking

```
// Generates random numbers.
public class RandomGenerator {
  private static RandomGenerator gen = null;

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

Singleton example 4 (Double-checked locking)

```
public class RandomGenerator {
  private volatile static RandomGenerator gen = null;

  public static RandomGenerator getInstance() {
    if (gen == null) {
      synchronized (RandomGenerator.class) {
         if (gen == null) // must check again
           gen = new RandomGenerator();
      }
    }
    return gen;
  }
}
```

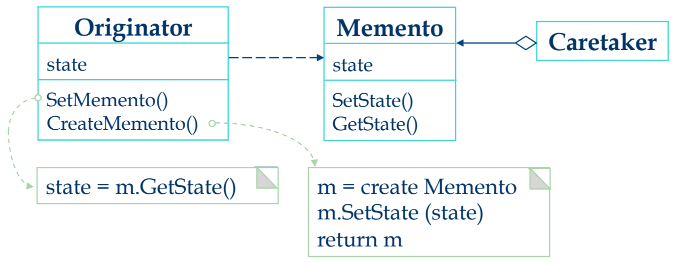
Summary



- The singleton pattern ensures that there is just one instance of a class.
- The singleton pattern provides a global access point.
- The pattern is implemented by using a private constructor and a static method combined with a static variable.
- Possible problems with multithreading.

Pattern: Memento

a memory snapshot of an object's state

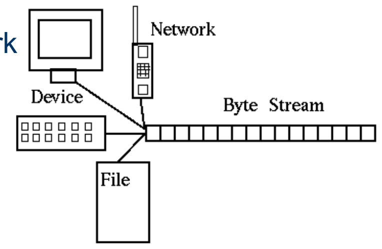


GoF Memento pattern

- problem: sometimes we want to hold onto a version of an important object's state at a particular moment
- **memento**: a saved "snapshot" of the state of an object or objects for possible later use; useful for:
  - writing an Undo / Redo operation
  - ensuring consistent state in a network
  - persistency; save / load state between executions of program
  - we'll examine Memento in the context of saving an object to disk using streams

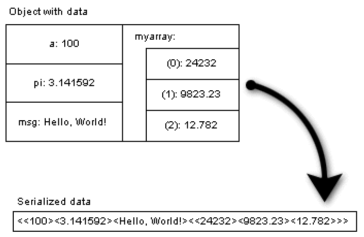
I/O streams, briefly

- **stream**: an abstraction of a source or target of data
- bytes "flow" to (output) and from (input) streams
- can represent many data sources:
  - files on hard disk
  - another computer on network
  - web page
  - input device (keyboard, mouse, etc.)



Serialization

- **serialization**: reading / writing objects and their exact state using I/O streams
  - allows objects themselves to be written to files, across network, to internet, etc.
  - lets you save your objects to disk and restore later
  - avoids converting object's state into arbitrary text format



Classes used for serialization

in `java.io` package:

- ObjectOutputStream class represents a connection to which an object can be written / sent (saved)

```
public class ObjectOutputStream
    public ObjectOutputStream(OutputStream out)
    public void writeObject(Object o)
         throws IOException
```

- ObjectInputStream class represents a connection from which an object can be read / received (loaded)

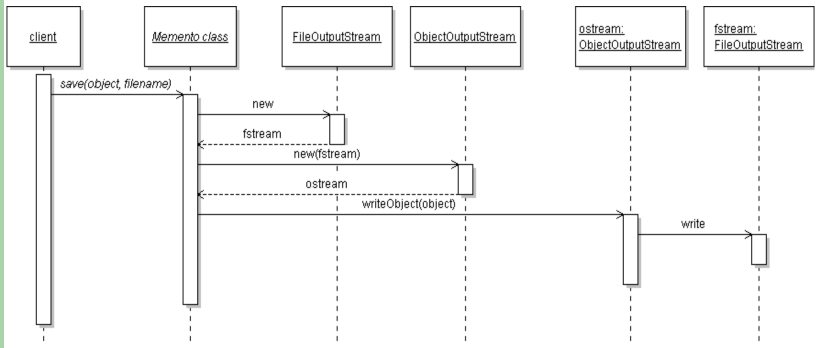
```
public class ObjectInputStream
    public ObjectInputStream(InputStream in)
    public Object readObject() throws Exception
```

Serialization example

- recommendation: use a Memento class that has save/load code

```
// write the object named someObject to file "file.dat"
try {
  OutputStream os = new FileOutputStream("file.dat");
  ObjectOutputStream oos = new ObjectOutputStream(os);
  oos.writeObject(someObject);
  os.close();
} catch (IOException e) { ... }

// load the object named someObject from file "file.dat"
try {
  InputStream is = new FileInputStream("file.dat");
  ObjectInputStream ois = new ObjectInputStream(is);
  ArrayList someList = (ArrayList)ois.readObject();
  is.close();
} catch (Exception e) { ... }
```

Memento sequence diagram



Making your classes serializable

- must implement the (methodless) `java.io.Serializable` interface for your class to be compatible with object input/output streams

```
public class BankAccount implements Serializable
{
   ...
```

- ensure that all instance variables inside your class are either serializable or declared `transient`
  - transient fields won't be saved when object is serialized

Back to singleton...

- Let's make our (singleton) game model serializable

- What can happen if a singleton is saved and loaded?
  - Is it possible to have more than one instance of the singleton's type in our system?
  - Does this violate the Singleton pattern? Will it break our code? If so, how can we fix it?

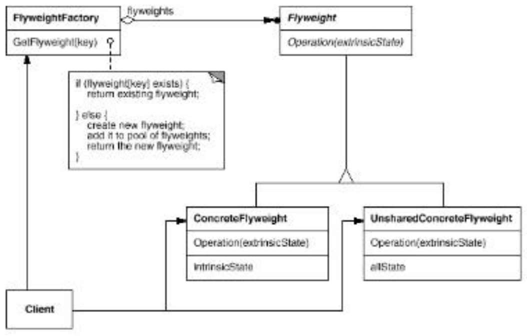
Singleton example 5

- Make our singleton serializable.
- has strict checks to make sure that we have not saved a stale reference to the singleton object

```
// Generates random numbers.
public class RandomGenerator {
  private static RandomGenerator gen = null;
  ...

  public double nextNumber() {
    // put code like this in methods that use/modify it
    if (this != gen)
      throw new IllegalStateException("not singleton");

    return Math.random();
  }
}
```

GoF Pattern: Flyweight

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


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

```
if (flyweight[key] exists) {
    return existing flyweight;
} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
```

Problem of redundant objects

- problem: existence of redundant objects can bog down system
  - many objects have same state
  - intrinsic vs. extrinsic state

  - example: File objects that represent the same file on disk
    - new File("mobydick.txt")
    - new File("mobydick.txt")
    - new File("mobydick.txt")

      ...
    - new File("notes.txt")
    - new File("notes.txt")

GoF Flyweight pattern

- **flyweight**: an assurance that no more than one instance of a class will have identical state
  - achieved by caching identical instances of objects to reduce object construction
  - similar to singleton, but has many instances, one for each unique-state object
  - useful for cases when there are many instances of a type but many are the same
  - can be used in conjunction with Factory pattern to create a very efficient object-builder
  - examples in Java: String, Image / Toolkit, Formatter

Flyweight and Java Strings

- Flyweighted strings
  - Java Strings are flyweighted by the compiler wherever possible
  - can be flyweighted at runtime with the `intern` method

```
public class StringTest {
  public static void main(String[] args) {
    String fly  = "fly", weight  = "weight";
    String fly2 = "fly", weight2 = "weight";

    System.out.println(fly == fly2);                  // true
    System.out.println(weight == weight2);            // true

    String distinctString = fly + weight;
    System.out.println(distinctString == "flyweight");// false

    String flyweight = (fly + weight).intern();
    System.out.println(flyweight == "flyweight");      // true
  }
}
```

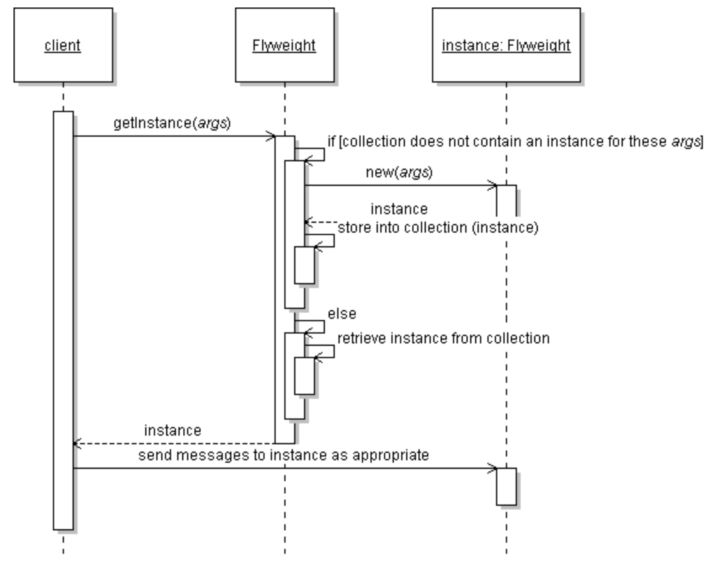
Implementing a Flyweight

- flyweighting works best on *immutable* objects
  - **immutable**: cannot be changed once constructed

class pseudo-code sketch:

```
public class Flyweighted {
    static map/table of instances
    private constructor
    static method to get an instance
        if we have created this type of instance before, get it from
        map and return it
        otherwise, make the new instance, store and return it
}
```

Flyweight sequence diagram


client
Flyweight
instance: Flyweight
getInstance(args)
if [collection does not contain an instance for these args]
new(args)
instance
store into collection (instance)
else
retrieve instance from collection
instance
send messages to instance as appropriate

Implementing a Flyweight

```
public class Flyweighted {
  Map or table of instances

  private Flyweighted() {}

  public static Flyweighted getInstance(Object key) {
    if (!myInstances.contains(key)) {
      Flyweighted fw = new Flyweighted(key);
      myInstances.put(key, fw);
      return fw;
    } else
      return (Flyweighted)myInstances.get(key);
  }
}
```

Class before flyweighting

- A class to be flyweighted

```
public class Point {
  private int x, y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() { return this.x; }
  public int getY() { return this.y; }

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

Class after flyweighting

- A class that has been flyweighted!

```
public class Point {
  private static Map instances = new HashMap();

  public static Point getInstance(int x, int y) {
    String key = x + ", " + y;
    if (instances.containsKey(key))// re-use existing pt
      return (Point)instances.get(key);

    Point p = new Point(x, y);
    instances.put(key, p);
    return p;
  }

  private final int x, y;  // immutable

  private Point(int x, int y) {
    ...
```

Summary

- **OO Basics**
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
- **OO Principles**
  - **Encapsulate what varies**
  - **Favor composition over inheritance**
  - **Program to interfaces not to implementations** (Depend on abstracts. Do not depend on concrete classes)
  - **Classes should be open for extension but closed for modification** (Strive for loosely coupled designs between objects that interact.)