CSC207 Week 6

Larry Zhang

Logistics

- Midterm: next Friday (October 27), 5:10 PM - 7:00 PM
- 110 minutes
- No aid
- Room: TBA (Note it’s not the lecture room)
- Past tests posted on the course website: http://axiom.utm.utoronto.ca/~207/17f/tests.shtml
- Arnold will send emails with more info.

Design Patterns

/r/LifeProTips

LPT: If your fire alarm goes off, call your pets and give them a treat. Eventually they will come when the alarm goes off, saving you from wasting time looking for pets during an evacuation.

LPT If somebody comes to your door selling a home security system and asks if you have one, always say yes.

LPT: Download Wifi analyzer to determine what channel your router should broadcast its Wifi Signal. Improved my wifi speeds by 22x

LPT: When trying to solve a computer error code by doing a google search, include the word "solved" in your search.

LPT: If you want a PS4/Xbox One on the cheap, check Craigslist/eBay around the time that semester report cards come out.

Design Patterns

Design patterns are like “LifeProTips” for software design.

- Effective use of the OO Paradigm.
- Using the patterns results in flexible, malleable, maintainable code.
- It usually allows plug and play additions to existing code.
- It gives developers a vocabulary to talk about recurring class organizations.

Design Patterns for Today

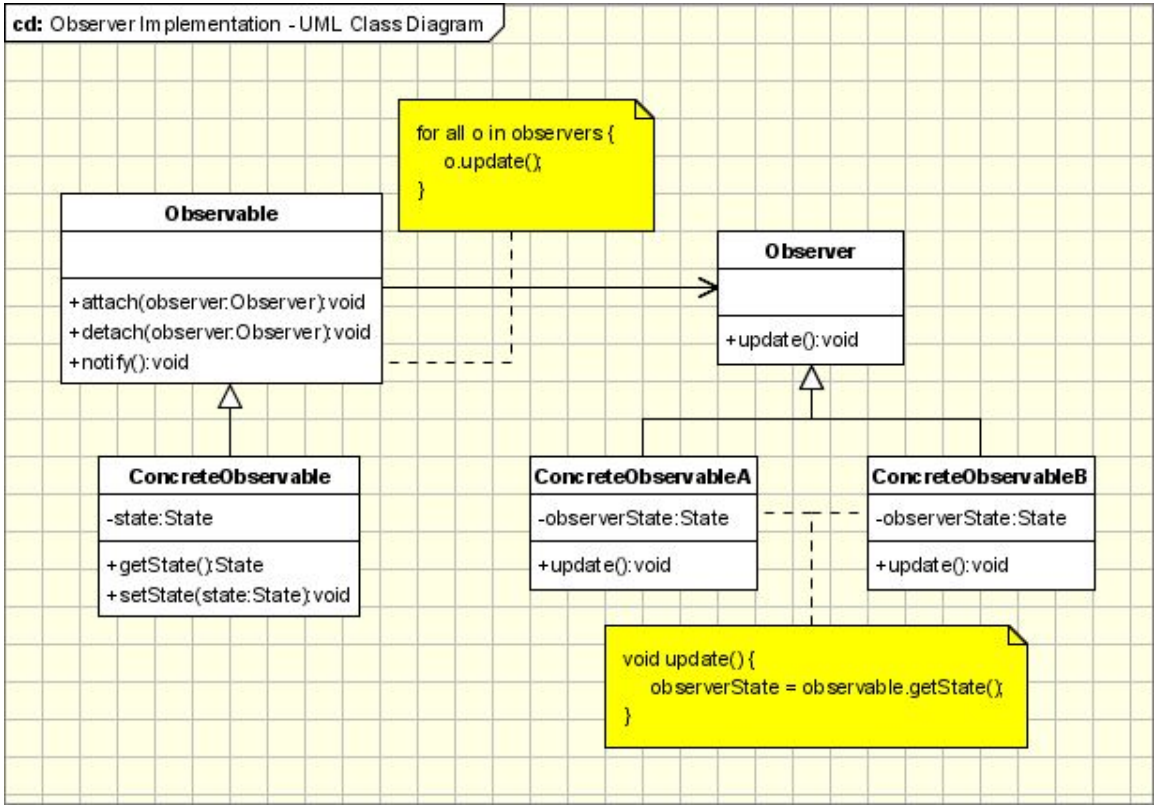
- Observer
- Singleton
- Iterator

The **Observer** Pattern

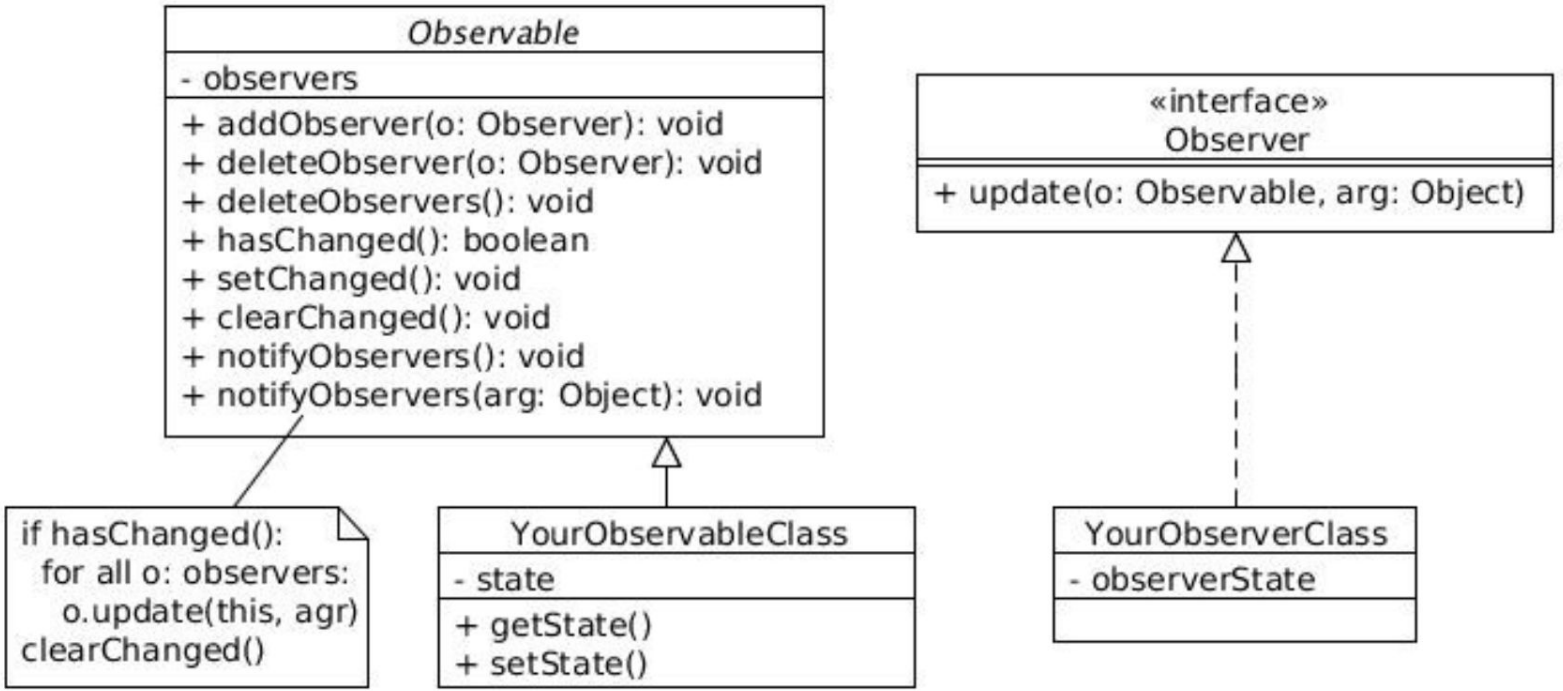
Goals

- When certain objects need to be informed about the changes occurred in other objects.
- One object changes state, all its dependents are notified and updated automatically.
- Programmer only need to worry about how the observer changes if the observables changes, and connecting the observers and observables.
- Real-life example: setting up Pre-Authorized Bill Payment

General Implementation



Java Implementation



DEMO #1

Implement Our Own Observer/Observable

`observer/Observable.java`

`observer/Observer.java`

DEMO #2

Use Our Observer/Observable

observer/LightBulb.java
observer/LightBulbObserver.java
observer/LightBulbObservable.java
observer/LightBulbMain.java

Advanced Issues in Observer Pattern

Push and **Pull** communication methods

- **Push** model: the Observable send all information about the change to the Observers (using an Object), and it's up to Observer to decide what information to use.
- You may end up pushing a lot of not-needed information to the Observers

```
// in Observable
def notify(Object X):
    for each observer ob:
       ob.update(X)
```

```
// in Observer
def update(Object X):
    use X to do stuff
    or not use it at all
```

Advanced Issues in Observer Pattern

Push and **Pull** communication methods

- **Pull** model: the Observable just send a short message **“a change happened”** to the Observers, then the Observer is responsible for **pulling** the **data they need** from the Observable
- Less waste of information passed, but more steps of communication.
- Also need to worry about concurrency, b/c multiple access on Observable.

```
// in Observable
// no need to passed an object
def notify():
    for each observer ob:
       ob.update()
```

```
// in Observer
def update():
    // only pull what I need
    data = this.observable.pullSomeData()
    do_stuff(data)
```

Push and Pull: real-life example

- **Push**: Arnold sends a **long** email to students in the class
  - “Assignment 2 is out. Here is the handout. Here is the starter code. Here is the Q&A. Here is my office hour. Here is a picture of my dog … (many other things) ... OK. You have all the information you ever need. You know what to do.”
  - Each student reads the email, use the information, and do the right thing.
- **Pull**: Arnold sends a **short** email to students in the class.
  - “Something changed”
  - Each student needs to go ask Arnold to get the information they need, then do their thing.

The **Singleton** Pattern

Singleton Pattern

Application needs one, and only one, instance of an object.

Also, provides a global point of access to that instance.

We want to have the following (sort of):

```
// create a new object of Singleton
Singleton s1 = new Singleton();

// don't create a new one, use the one created before
// so s1 and s2 are referring to the same object
Singleton s2 = new Singleton();
```

Attempt #1 to implement Singleton

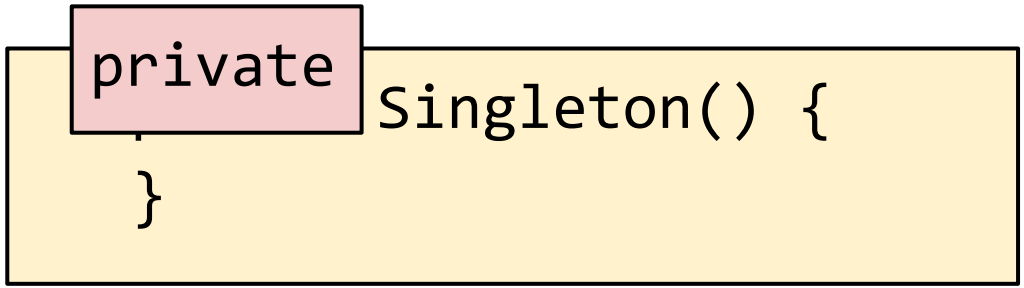
- Have a **static counter** counting the number of instances created so far
- And check the counter in the **constructor**, then do the right thing.

```
public class Singleton {
    static int counter = 0;
    public Singleton() {
        if (counter == 0) {
            // create new object …
            counter++;
        }
        else {
            // don’t create new object ...
        }
    }
}
```

Doesn’t work: whenever you invoke the constructor method, you are unavoidably creating a new object!

We can’t let other people use the constructor!

How do we **forbid** the use of a constructor by other classes?



Valid implementation #1

```
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

When is the object create?

- Right away **when the program starts**

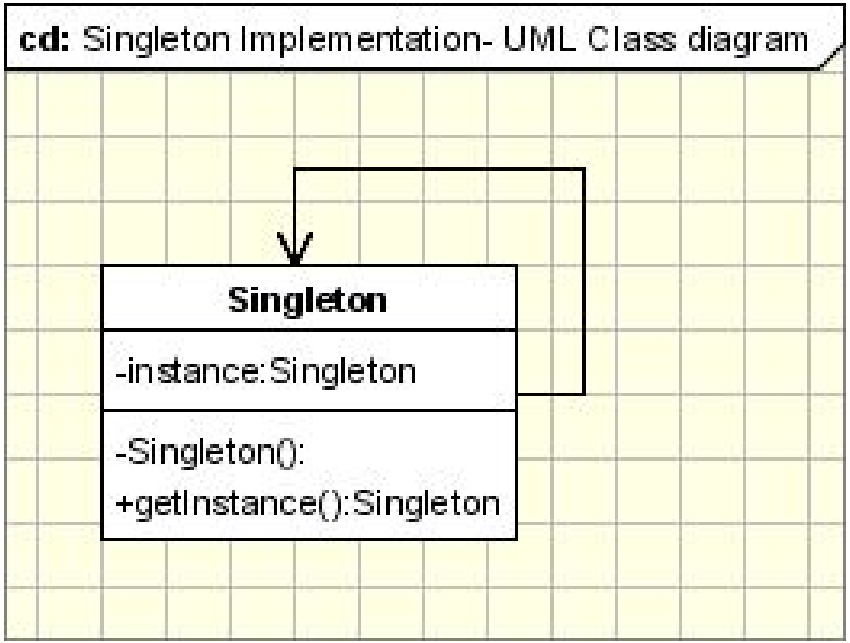
Valid implementation #2

```
public final class Singleton {
    private static volatile Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

When is the object created?

- When the **first time you use** the Singleton class
- This is called **Lazy Initialization**

UML for Singleton



DEMO #3

Singleton

`singleton/Singleton.java`
`singleton/UseSingleton.java`

The **Iterator** Pattern

The Iterator Pattern

Collections (group of objects) are one of most often used data types in software development.

- Java built-in collection: Array, ArrayList, HashMap, etc.
- User may also define their own collection classes, e.g., SongCollection, GitLog, FacebookFriends, etc.

The **goal** of the Iterator pattern is to:

- have a **unified** mechanism to **traverse** any collection
- **hide** the internal implementation of the collection, i.e., how the elements are really stored.

The two ways of traversing that we like

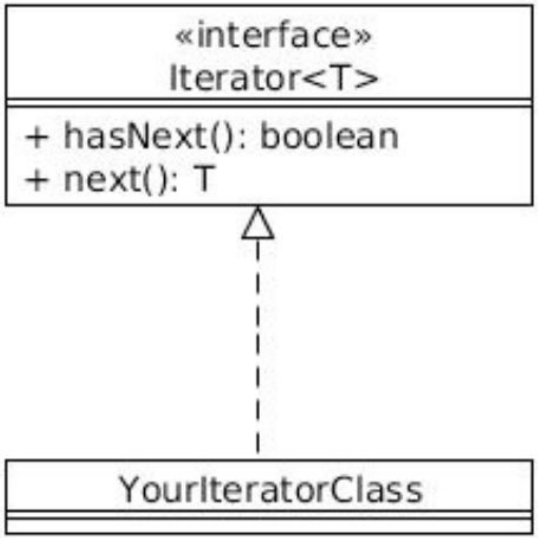
```
Collection c;
Iterator it = c.iterator();

while (it.hasNext()) {
    print(it.next());
}
```

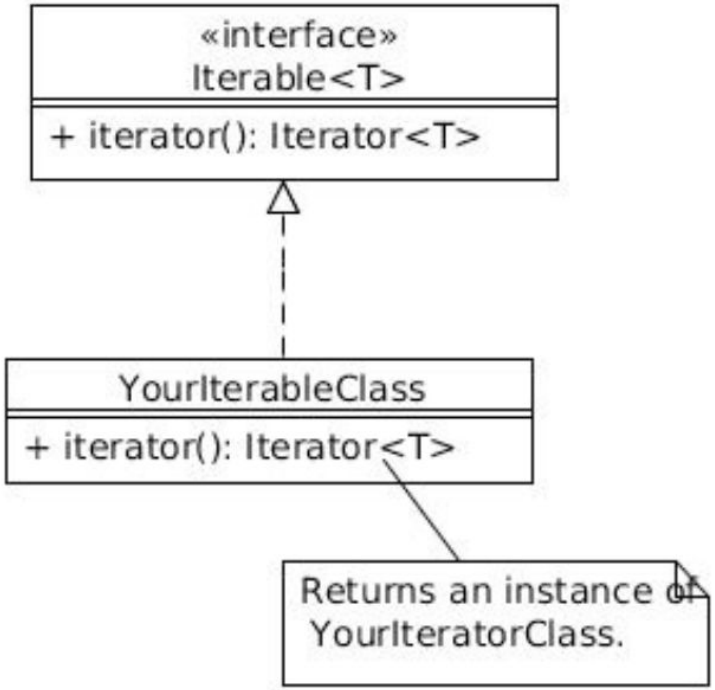
```
Collection c;

for (Object o: c) {
    print(o);
}
```

iterator() returns an iterator that points to the beginning of the collection
hasNext() returns if the iterator is not at the end of the collection.
next() returns the current item and moves the iterator one step forward.

How to implement in Java?

- The Collection class implements the **Iterable<> interface**
  - need to implement an **iterator()** method, which returns an iterator
- The Collection's iterator class implements the **Iterator<> interface**, which involves implementing:
  - **constructor**: create a new iterator pointing at the beginning of the collection
  - **hasNext**(): return False iff the iterator is at the end of the collection
  - **next**(): return the current item, move iterator one step forward.


«interface»
Iterator<T>
+ hasNext(): boolean
+ next(): T
YourIteratorClass


«interface»
Iterable<T>
+ iterator(): Iterator<T>
YourIterableClass
+ iterator(): Iterator<T>
Returns an instance of
YourIteratorClass.

DEMO #4

Simple Iterator Example

`iterator/SimpleIteratorDemo.java`

DEMO #5

User-Defined Collection and Iterator

`iterator/Song.java`
`iterator/SongCollection.java`
`iterator/SongCollectionIterator.java`
`iterator/SongCollectionmain.java`

```
# Python:
for i in range(1, 10, 2):
    print(i)
```

DEMO #6

Range and RangeIterator

iterator/Range.java
iterator/RangeIterator.java
iterator/RangeMain.java

DEMO #7

An “Infinite” Collection: AllIntegers

`iterator/AllIntegers.java`
`iterator/AllIntegersIterator.java`
`iterator/AllIntegersMain.java`

EXERCISE: Change the code to get:

- all even natural number
- all odd natural numbers
  - all multiples of 42
  - all prime numbers
- all real numbers (is it possible?)

References

See the “Readings” section on the course website.

In general, http://www.oodesign.com/ is good read.