Singleton / Multiton / Observer

Advanced Java Programming 2012-13

OO Design Patterns and Principles

Lecture Outline

- Singleton pattern
- Multiton pattern
- Observer pattern

The Singleton Pattern

The Singleton Pattern

- Some applications require that one and only one instance of a class is ever created
- Examples
  - Configuration classes e.g. games preferences
  - Global counters, keeping track of values being used in multiple classes of your application e.g. shopping cart
  - Hardware drivers e.g. printer
- How do you prevent a class from being instanced more than once?

The Singleton Pattern

- The *Singleton* pattern was developed to solve this common problem
  - It ensures that a class only has one instance and provides a global point of access to it
- The keystone in the pattern is a *private* constructor
  - ??????
- How does that work?
  - I’d have to have an instance of the class to call the constructor, but I can’t have an instance because no other class can instantiate it

The Singleton Pattern

```
public MyClass
{

   private MyClass ()
   {

   }

}
```

A public class

With a private constructor!

The following line of code in a test class is illegal

MyClass mc = new MyClass();

The Singleton Pattern

- Let me re-phrase that. The keystone of the singleton pattern is a private constructor and a <u>static method</u>
- Let's recall the characteristics of a static method
  - A static method can be called without creating an instance of the class e.g. System.out.println()
  - A static method is called by prefixing it with a class name e.g. Math.max(x,y);
  - Static methods cannot access any instance variables i.e. non-static variables

The Singleton Pattern

```
public MyClass
{

   private MyClass () {…}

   public static MyClass getInstance()
   {
      return new MyClass();
   }

}
```

A public class

With a private constructor!

And a static method that creates an instance of the class and returns it. So, in a test class we write:

```
MyClass mc = MyClass.getInstance();
```

PROBLEM SOLVED?

The Singleton Pattern

- Does this solve the problem?
  - No
- The static method can be called multiple times, resulting in multiple instances of *MyClass* floating around
- How do we get around this?
  - We need to do something in the *getInstance*() method to complete the class
- When called, the method should check if a *MyClass* object already exists
  - If it does, return that object
  - If it does not, call the constructor and return the object it creates

The Singleton Pattern

```
public MyClass
{
   private static MyClass unique;
   private MyClass () {…}

   public static MyClass getInstance()
   {
      if (unique == null)
      {
         unique = new MyClass();
      }
      return unique
   }
}
```

A static variable, initial value null

A static method which checks whether a *MyClass* object has ever been created. If NO, create a new one and return it. If YES, return a reference to the old one.

The Singleton Pattern

```
{
    // In a test class, check it works

    MyClass mc = MyClass.getInstance();
    System.out.println("Memory address:" + mc);

    MyClass mc2 = MyClass.getInstance();
    System.out.println("Memory address:" + mc2);

}
```

```
Output:
Memory address: MyClass@e86f41
Memory address: MyClass@e86f41
```

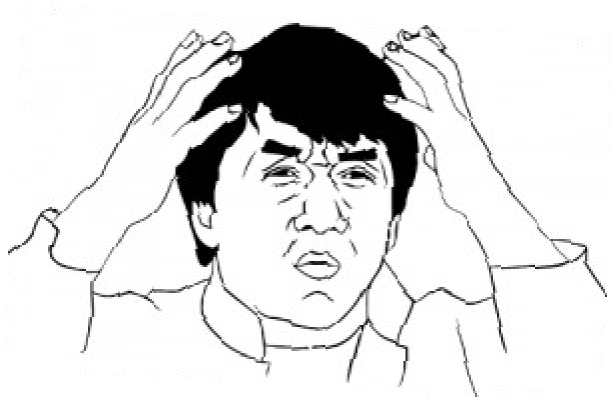
The Singleton Pattern

- Advantages of the singleton pattern
  - Lazy initialization (useful if initiation is a heavy process e.g. creating a DB connection)
- Disadvantages of singleton pattern
  - Makes it difficult to apply unit testing as you have introduced global state
    - Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. It is difficult to test small chunks of code when they incorporate global variables
    - The Java equivalent of global variables – static variables
  - Classloaders - If the Singleton class is loaded by 2 different class loaders we'll have 2 different classes, one for each class loader

The Multiton Pattern

The Multiton Pattern

- The *Multiton* pattern expands on the *Singleton* concept to manage a map of named instances as key-value pairs
- Rather than have a single instance per application the *Multiton* pattern instead ensures a single instance per key
- Say what?



The Multiton Pattern

- Last year you learnt about a few different collections for variables
  - Arrays
  - Vectors
- The Java Collections framework contains a <u>LOT</u> more
  - Sets
  - Trees
  - Queues
  - Stacks
  - Maps

The Multiton Pattern

- A map is a data collection which aligns a *key* with a *value*
- An everyday example of a map is a telephone book, which aligns names (Mark Truran) with numbers (2267)
- In Java, there are several different implementations of the map structure
  - *HashMap* is a good, all purpose map type
  - *HashMap* is type safe – you must pre-declare the type of variables you intend to store in it
  - You add (key,value) pairs to a *HashMap* using the *put*() method
  - You retrieve values from the *HashMap* by calling the *get*() method and passing a key

The Multiton Pattern

```
// Declare the HashMap
HashMap<String, int> telephoneBook;
telephoneBook = new HashMap<String, int>();

// Add a (k,v) pair
telephoneBook.put(“Mark Truran”, 2267);

// retrieve a value
int number = telehoneBook.get(“Mark Truran”);
```

The Multiton Pattern

- Now imagine a situation where you want to limit the instantiation of a class to one object per named group
- Assume we have a *Settings* class, which holds the default settings for computer users
  - There are three types of user – Technicians, Students and Academics
  - We only want one instance of this class for each group
  - We can do this by tweaking the Singleton to use a map instead of a single instance variable

The Multiton Pattern

```
public Settings {

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

    private MyClass () {…}

    public static Settings getInstance(String key)
    {
        Settings instance = instances.get(key);
        if (instance == null) {
            instance = new Settings();
            instances.put(key, instance);
        }
        return instance;
    }
}
```

Declare the map

If the key is not in the map, create a new (k,v) pair and return the key.

If the key is in the map, then get the matching value and return it

PROBLEM SOLVED?

The Multiton Pattern

- This implementation does not solve the problem because we can still get an arbitrary number of instances of the class, just by passing arbitrary strings
  - `Settings.getInstance(“Technician”); // OK`
  - `Settings.getInstance(“Student”); // OK`
  - `Settings.getInstance(“Academic”); // OK`
  - `Settings.getInstance(“Hobbit”); // BAD`
- We can solve this easily with an enum

```
public enum User {
ACADEMIC, TECHNICIAN, STUDENT }
```

The Multiton Pattern

```
public Settings {

    private static HashMap<User, Settings> instances =
    new HashMap<User, Settings>();

    private MyClass () {…}

    public static Settings getInstance(User u)
    {
        Settings instance = instances.get(u);
        if (instance == null) {
            instance = new Settings();
            instances.put(u, instance);
        }
        return instance;
    }
}
```

Declare the map with the enum type as the key

If the key is not in the map, create a new (k,v) pair and return the key.

If the key is in the map, then get the matching value and return it

The Multiton Pattern

- The effect of using an enum type instead of a *String*
  - `Settings.getInstance(“User.Technician”); // OK`
  - `Settings.getInstance(“User.Student”); // OK`
  - `Settings.getInstance(“User.Academic”); // OK`
  - `Settings.getInstance(“User.Hobbit”); // COMPILE ERR`

The Observer Pattern

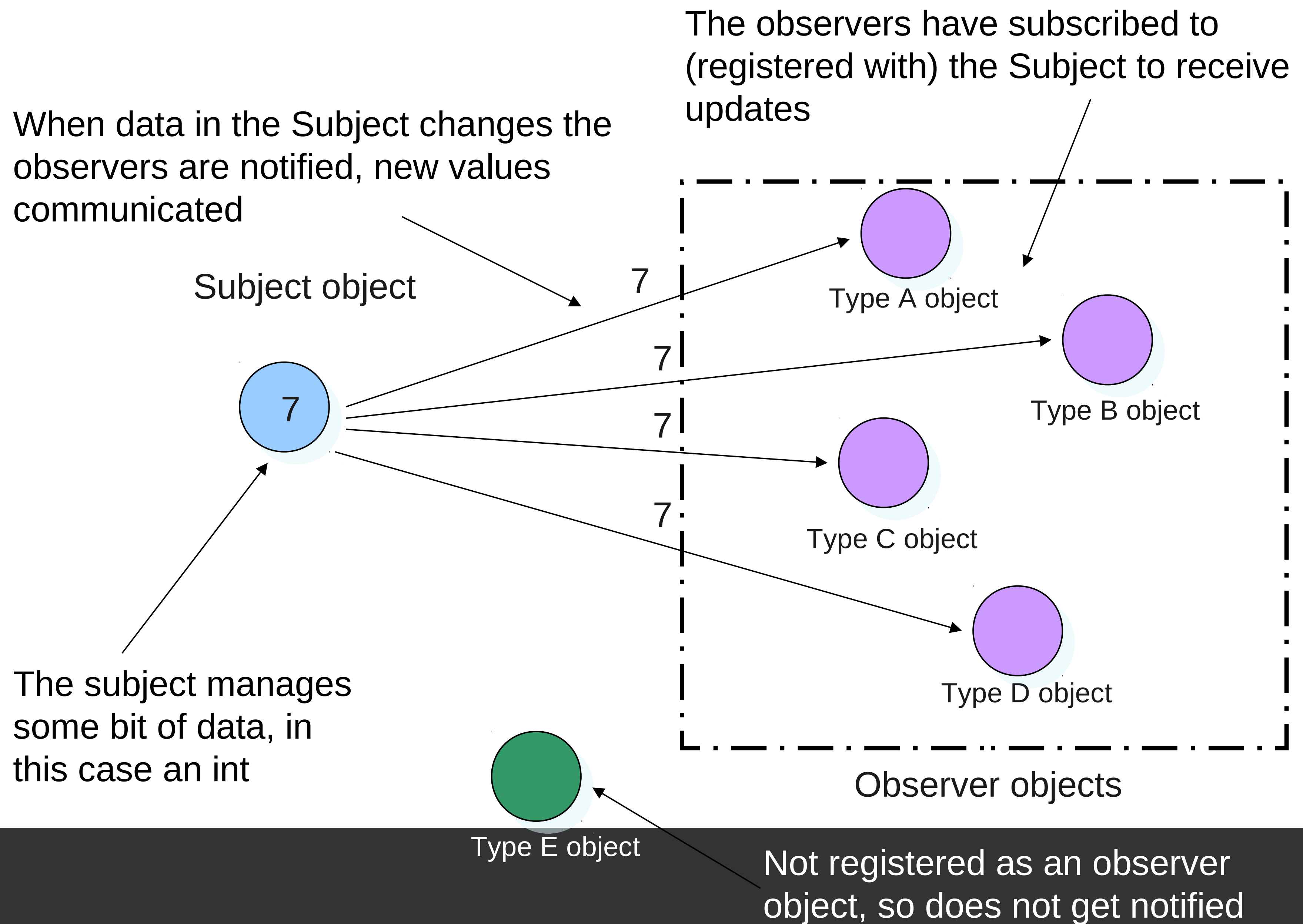
The Observer pattern

- You all know how newspaper subscriptions work
  - A newspaper goes into business and starts publishing newspapers
  - If you like the newspaper you subscribe – each new edition of the newspaper is delivered to you
  - As long as you are a subscriber you get new newspapers
  - You unsubscribe when you don’t want to receive that paper any longer
  - The deliveries stop – you have opted out
- Important point
  - The newspaper will maintain a list of subscribers
  - It could not work otherwise

The *Observer* pattern

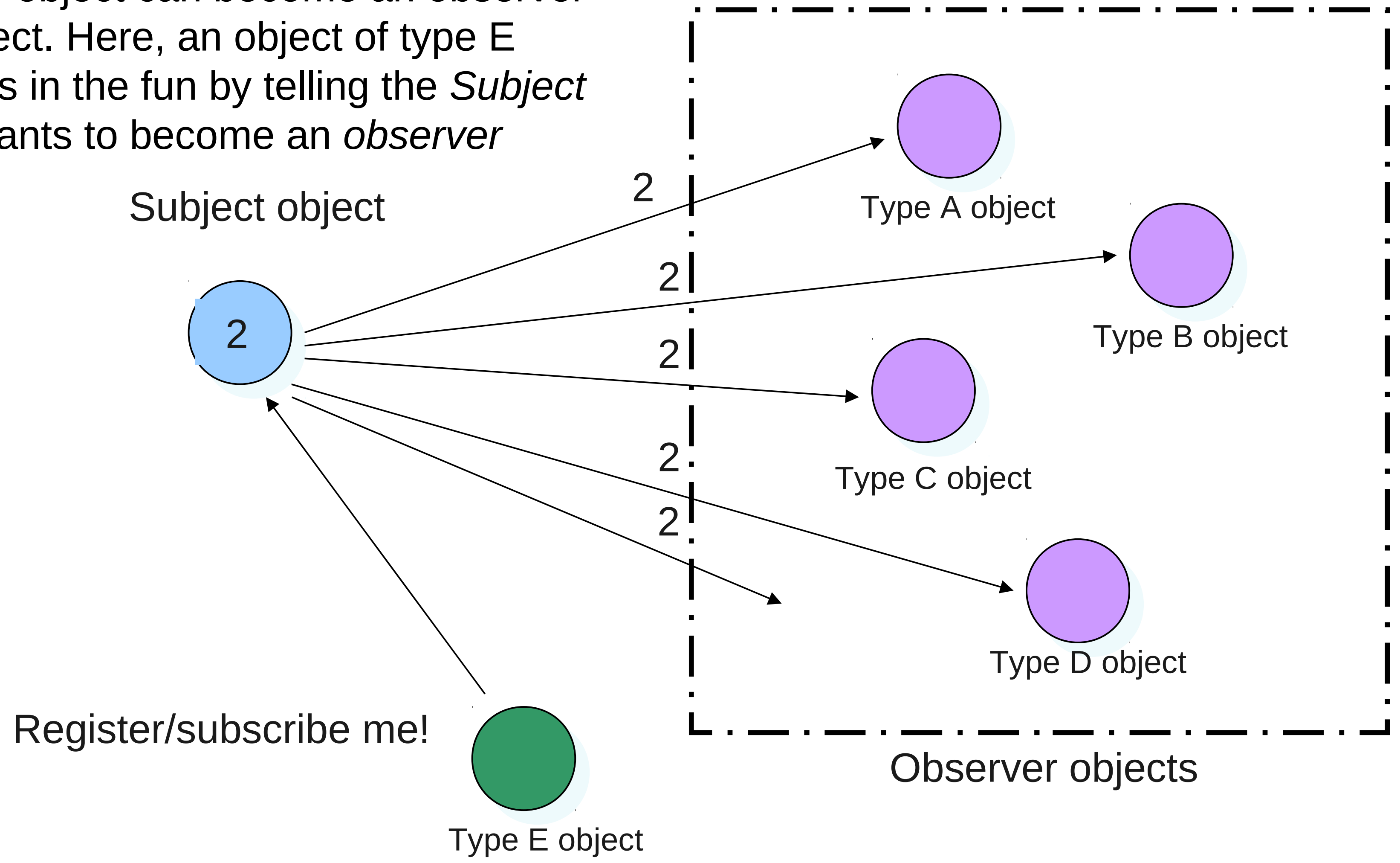
- If you understand the publish-subscribe model used by newspapers, you can understand the *Observer* design pattern
- The Observer pattern is exactly the same as the publish-subscribe model except
  - We replace the term 'publisher' with the term **SUBJECT**
  - We replace the term 'subscriber' with the term **OBSERVER**
- The observer pattern is a behavioural pattern
  - We use it to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Examples of this relationship in software engineering
  - You have a spreadsheet full of data. When the data is changed, you want a set of related diagrams to be updated

The *Observer* pattern



The *Observer* pattern

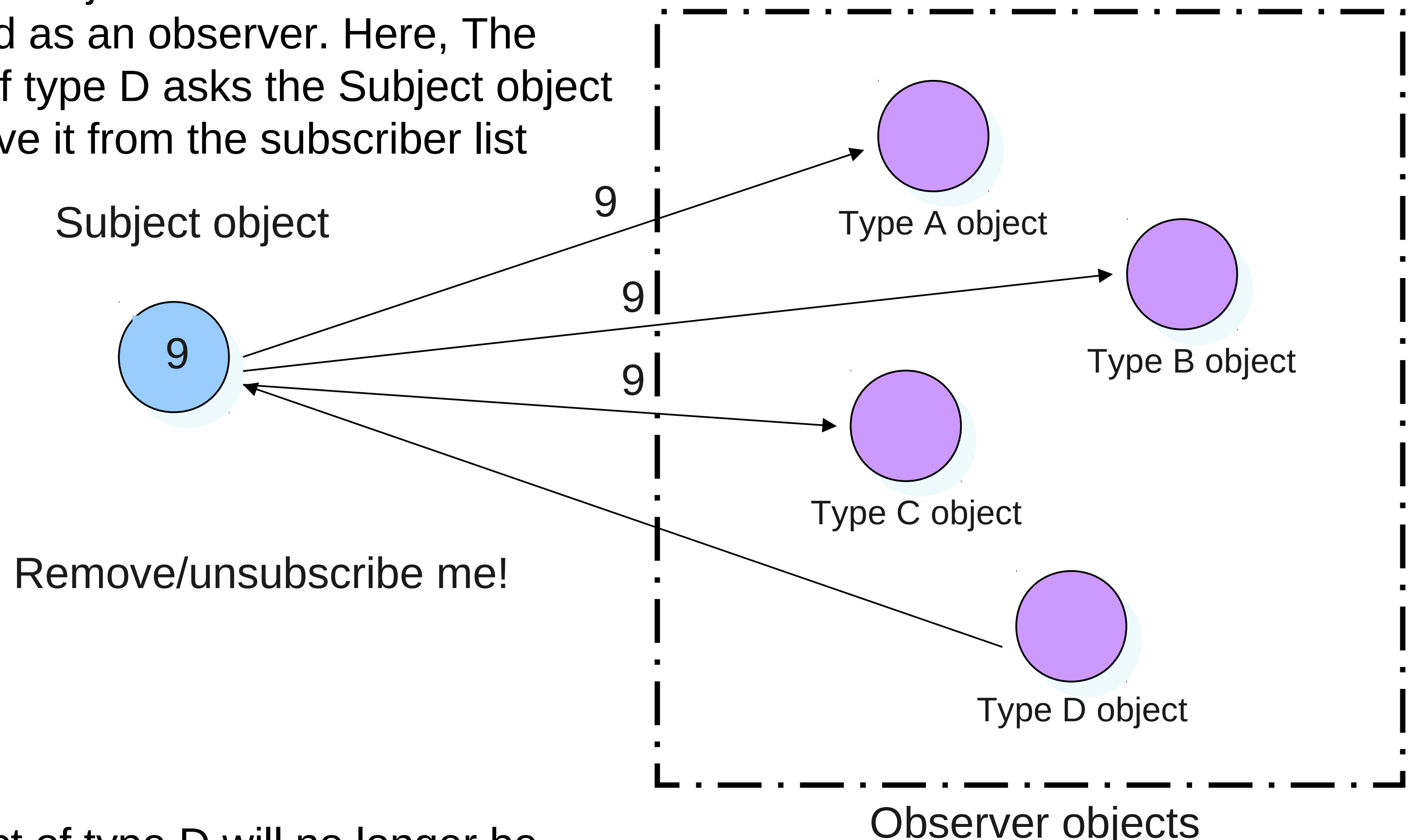
Any object can become an observer object. Here, an object of type E joins in the fun by telling the *Subject* it wants to become an *observer*



The object of type E will now be notified of any changes in state

The *Observer* pattern

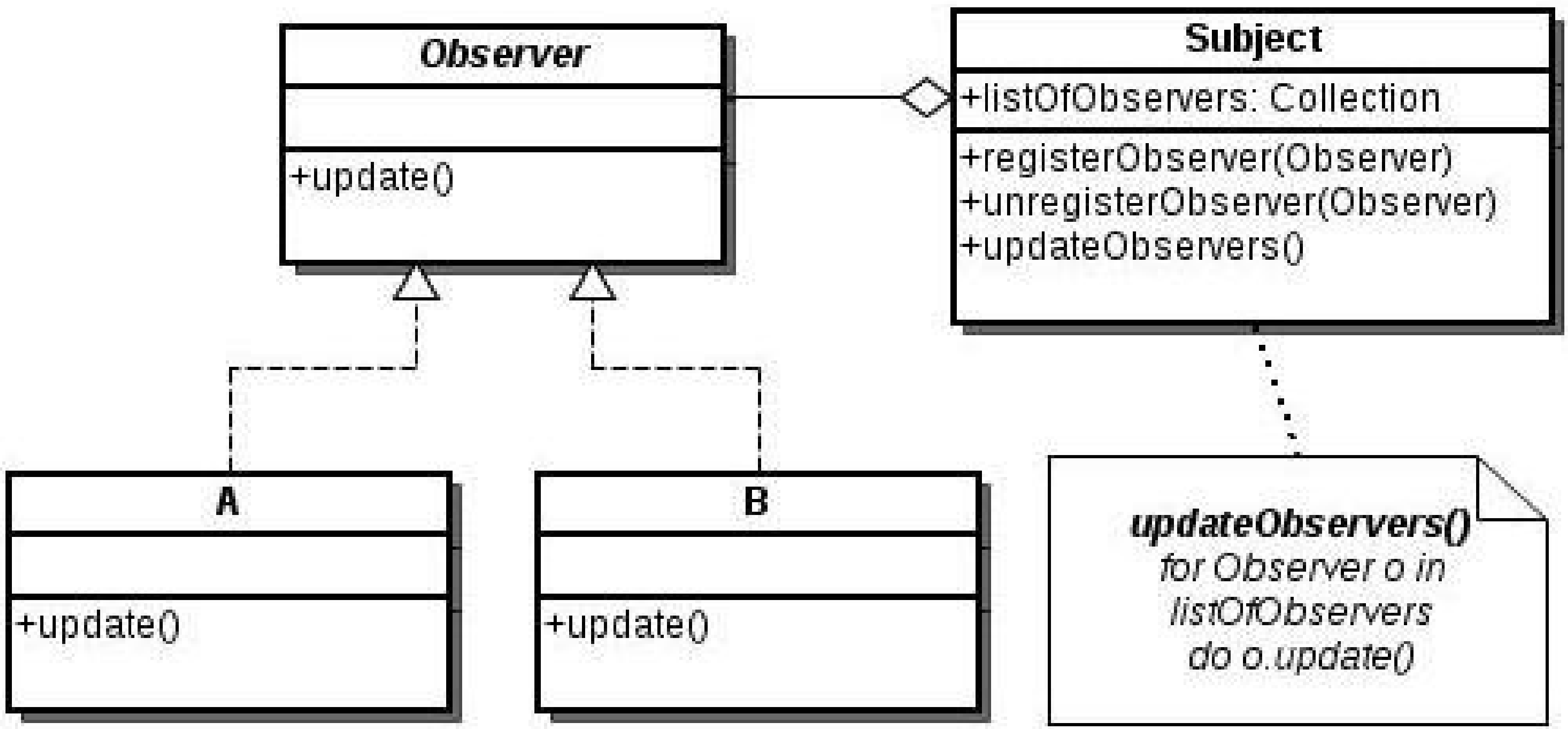
Observer objects can ask to be removed as an observer. Here, The object of type D asks the Subject object to remove it from the subscriber list



The object of type D will no longer be notified of any changes in state

The *Observer* pattern

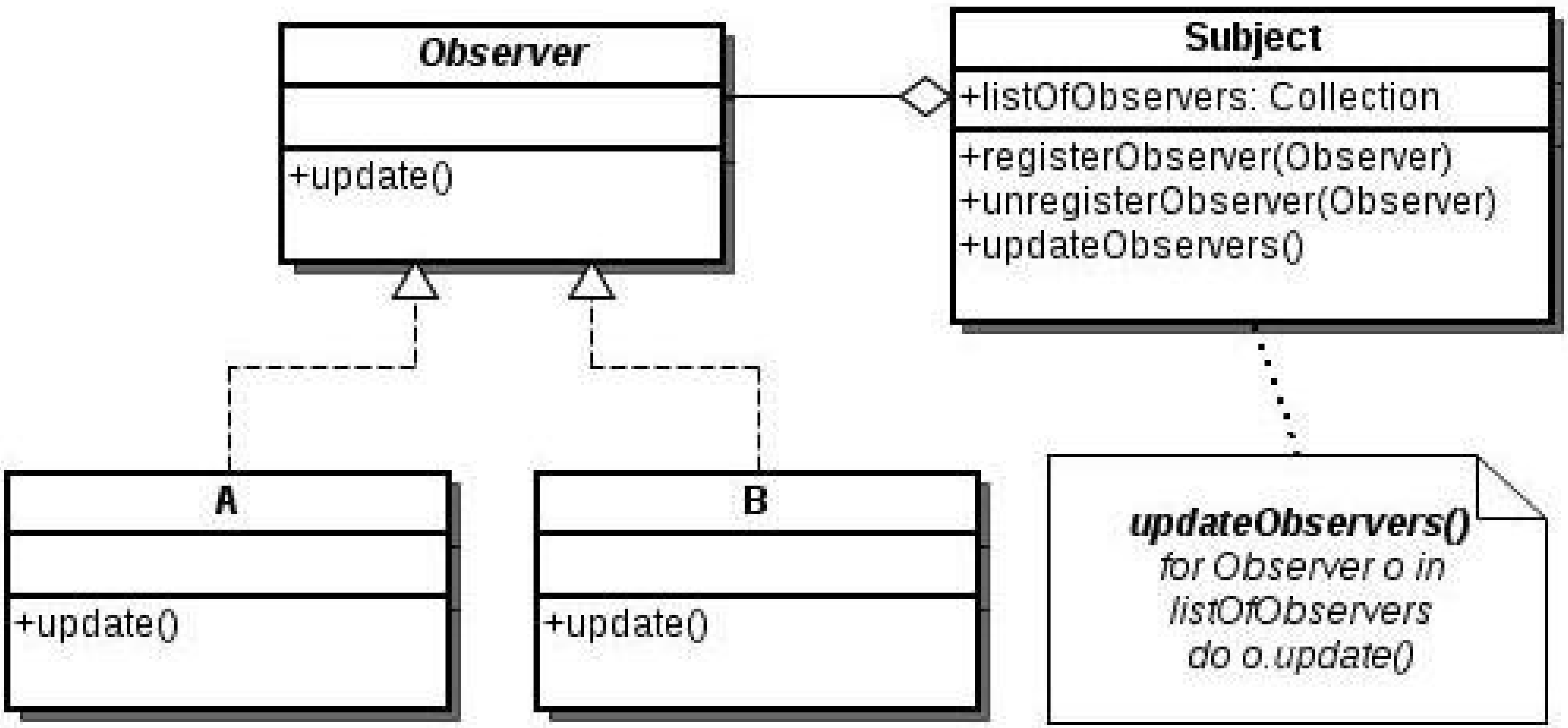
The UML diagram



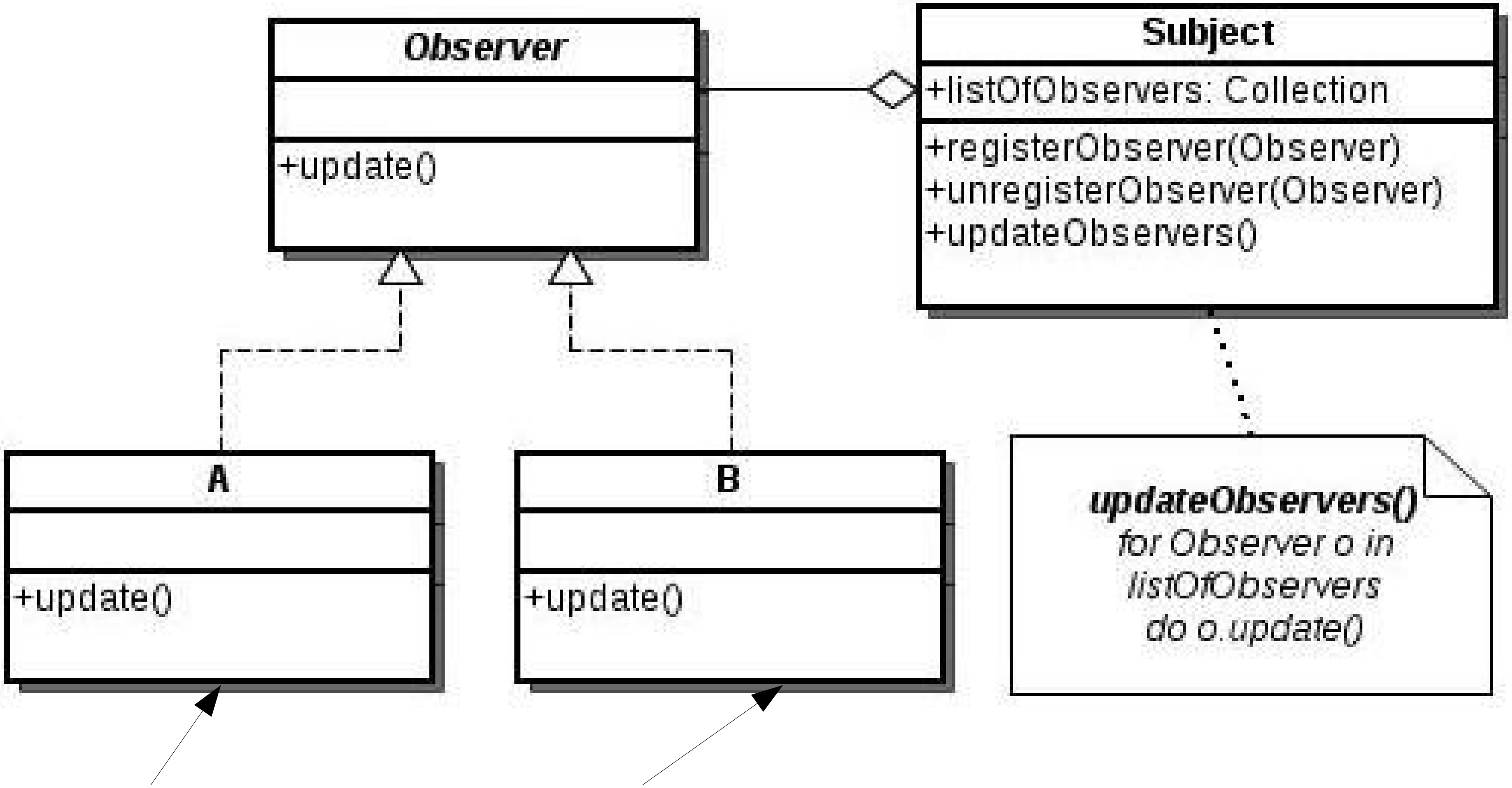
Let's break this diagram down...

The *Observer* pattern

This is the <u>subject</u>. The other objects are watching an instance of this class. We will come back to the methods later, but note that the Subject has a *list* of known observers as an instance variable



The *Observer* pattern



These are the observers. The observers are objects that want to be notified when there is some change in the subject. Note that each object is an instance of a class that implements the *Observer* interface. This mean that they MUST implement a method called *update*()

The *Observer* pattern

```
interface Observer
{
   public void update();
}

class A implements Observer
{
   public void update() {...}
}
class B implements Observer
{
   public void update() {...}
}
```

Interface

Concrete implementations of the *Observer* interface

The *Observer* pattern

- The *Subject* class has two specific tasks
- The first task is to maintain a list of known observers
- The list of observers is usually implemented using a suitable Java Collection class e.g. Vector, ArrayList etc.
- There are two methods for manipulating the list
  - *registerObserver(Observer o)*
    - Called when you want to add another *Observer* to the list
    - Parameter – a reference to the object to want to add
  - *unregisterObserver(Observer o)*
    - Called when you want to remove an *Observer* from the list
    - Parameter – a reference to the object to want to remove

The *Observer* pattern

A list of all interested objects.

```
class Subject
{
   ArrayList<Observer> al = new ArrayList<Observer>();

   public void registerObserver(Observer o)
   {
      al.add(o);
   }

   public void unregisterObserver(Observer o)
   {
      al.remove(o);
   }
}
```

The *Observer* pattern

- These are the steps we would take to test our list
  - Instance the *Subject* class
  - Instance one of the classes that implements *Object*
  - Register the *Observer* with the *Subject*
  - Unregister the *Observer* with the *Subject*

```
Subject s = new Subject();
A a = new A();
B b = new B();
s.registerObserver(a);
s.registerObserver(b);
s.unregisterObserver(a);
```

That is all there is to list management. Now, you can register any object with the subject. That is the first step.

The *Observer* pattern

- The second major task of the *Subject* class is to push updates to all *Observers* when its state changes
- In this example, we will use a single int variable as the 'state' that is being watched,

```
class Subject
{
    ArrayList<Observer> al = new ArrayList<Observer>();

    int state = 0;     // This is the 'thing' that
                        // the other objects are watching
    …
}
```

The *Observer* pattern

- Now imagine that something happens so that the value stored in this 'watched' variable changes
  - For example, a method called changeState() is called, which changes the value stored in the variable from 0 to 5
  - When that occurs, we need to notify all the observers
  - The first step involves calling the *updateObservers*() method

```
class Subject
{
    public void changeState()
    {
        state = 5;
        updateObservers();
    } ...
}
```

The *Observer* pattern

- When *updateObservers*() is called we need to iterate through the list of registered observers, calling the update() method of each object
  - We know they have this method because of the interface
  - When we call the update() method, we need to pass some information – in this case the new value for the int called 'state'
  - Note the polymorphic use of the interface type *Observer*

```
public void updateObservers()
{
    for(Observer o : al)
    {
        o.update(state);
    }
}
```

The *Observer* pattern

- In the classes implementing *Observer*, a local copy of the *state* variable is subsequently updated

```
class A implements Observer
{
    int state = 0;

    public void update(int i)
    {
        state = i;
        System.out.println("Updated A”
        System.out.println(“New value is " + state);
    }
}
```

The *Observer* pattern

- **Why is the Observer pattern useful?**
- It provides an object design where subjects and observers are *loosely coupled*
  - Two objects are loosely coupled when they can interact but have very little information about each other
  - The only thing a subject knows about an observer is that it implements a certain interface
  - It does not need to know the concrete class of the observer, what it does, or anything else about it
  - We can add new observers (and new <u>types</u> of observers) whenever we want
- Loosely coupled designs allow us to build flexible OO systems that handle change because they minimize the interdependency between objects
  - The more dependency, the more maintenance

Summary

- We use the *Singleton* pattern when we want to limit instantiation of a class to just one object
- The *Multiton* pattern is an extension of *Singleton* that allows us to manage a map of named instances
- We use the *Observer* pattern when we want to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Pre-reading**
  - Template pattern in HFDP and DPFD
  - Chain of responsibility pattern in HFDP and DPFD