CS342: Software Design

Nov. 7, 2017
Today’s topic

Singleton pattern

Iterator pattern
JUnit test case execution order

There’s no specific order which @Test method will be executed

```java
public class JunitAnnotation {
    @BeforeClass
    public static void beforeClass() {
        System.out.println("in before class");
    }

    @AfterClass
    public static void afterClass() {
        System.out.println("in after class");
    }

    @Before
    public void before() {
        System.out.println("in before");
    }

    @After
    public void after() {
        System.out.println("in after");
    }

    @Test
    public void test() {
        System.out.println("in test");
    }

    @Test
    public void testAgain() {
        System.out.println("in testAgain");
    }

    @Ignore
    public void ignoreTest() {
        System.out.println("in ignore test");
    }
}
```

Therefore, “in test” and “in testAgain” can show up either order
Singleton pattern: Create one-of-a-kind objects

There is only one instance of the class

- Ensures a class has only one instance, and provides a global point of access to it.
- Create a class and letting it manage a single instance of itself, preventing any other class from creating a new instance on its own.
- Whenever you need an instance, just query the class and it will hand you back the single instance.
- Usage: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards.
A class with a private constructor

```java
public MyClass {
    private MyClass() {}
    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

```java
public class Singleton {
    private static Singleton uniqueInstance;
    // other useful instance variables here
    private Singleton() {}
    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    // other useful methods here
}
```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
    public ChocolateBoiler() {
        empty = true;
        boiled = false;
    }
    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }
    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }
    public boolean isEmpty() {
        return empty;
    }
    public boolean isBoiled() {
        return boiled;
    }
}
However, there is only one boiler...

If multiple instances of boiler class are created, we will run into a big problem.

Singleton is needed!

```java
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    // rest of ChocolateBoiler code...
}
```
singleton chocolate boiler

```java
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
}
```

// rest of ChocolateBoiler code...
There is only one instance of the class

- The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.
- Create a class and letting it manage a single instance of itself, preventing any other class from creating a new instance on its own.
- Whenever you need an instance, just query the class and it will hand you back the single instance.
- Usage: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards.
The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

The uniqueInstance class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.
Singleton benefits and risk

There is only one instance of the class

- Better resource usage: you don’t have to create a new object when you need it and then destroy it after you are done
- Performance gain
- Commonly used in SOA: instantiate a service
- One of a kind means clients may fight to use it!
- Thread-safe?
Chocolate boiler and three threads

public static ChocolateBoiler getInstance() {
    if (uniqueInstance == null) {
        uniqueInstance = new ChocolateBoiler();
    }
    return uniqueInstance;
}
How to make singleton thread-safe?

1. Use synchronized modifier -- expensive, decrease performance 100x. Use it only when ‘getInstance()’ isn’t critical.

2. Eagerly create instance.

```java
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

3. Double check locking: First check to see if an instance is created, and if not, THEN use synchronizes (see next page).
“volatile” ensures multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.
Iterator pattern

Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Allows traversal of the elements of an aggregate without exposing the underlying implementation
- Places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be
Case study: two restaurants merging

“Diner” provides lunch, “Pancake house” provides breakfast

● For the waitress to work, we need to put the menus together
● Menu item: name, description, price
Menu item: two restaurant has the same class

class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                     String description,
                     boolean vegetarian,
                     double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
public class PancakeHouseMenu implements Menu {
    ArrayList<String> menuItems;
    public PancakeHouseMenu() {
        menuItems = new ArrayList<String>();
        addItem("K&B's Pancake Breakfast",
                "Pancakes with scrambled eggs, and toast",
                true, 2.99);
        addItem("Regular Pancake Breakfast",
                "Pancakes with fried eggs, sausage",
                false, 2.99);
        addItem("Blueberry Pancakes",
                "Pancakes made with fresh blueberries",
                true, 3.49);
        addItem("Waffles",
                "With choice of blueberries or strawberries",
                true, 3.59);
    }
    public void addItem(String name, String description, boolean vegetarian, double price) {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }
    public ArrayList<String> getMenuItems() {
        return menuItems;
    }
    // other menu methods here
}
public class DinerMenu implements Menu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;
    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];
        addItem("Vegetarian BLT", "Bacon with lettuce & tomato on whole wheat", true, 2.99);
        addItem("BLT", "Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day", "Soup of the day, with a side of potato salad", false, 3.29);
        addItem("Hotdog", "A hot dog, with onions, topped with cheese", false, 3.05);
    }
    public void addItem(String name, String description,
            boolean vegetarian, double price) {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full! Can’t add item to menu");
        } else {
            menuItems[numberOfItems] = menuItem;
            numberOfItems = numberOfItems + 1;
        }
    }
    public MenuItem[] getMenuItems() {
        return menuItems;
    }
}
Waitress class

```java
public class Waitress {
    public void printMenu () {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();
        DinerMenu dinerMenu = new DinerMenu();
        MenuItem[] lunchItems = dinerMenu.getMenuItems();

        for (int i = 0; i < breakfastItems.size(); i++) {
            MenuItem menuItem = (MenuItem)breakfastItems.get(i);
            System.out.print(menuItem.getName() + " ");
            System.out.println(menuItem.getPrice() + " ");
            System.out.println(menuItem.getDescription());
        }
        for (int i = 0; i < lunchItems.length; i++) {
            MenuItem menuItem = lunchItems[i];
            System.out.print(menuItem.getName() + " ");
            System.out.println(menuItem.getPrice() + " ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

It’s annoying to go through both menus to do something.

- Search menu by name
- Look for item under $5?
- What if we have a third, fourth, … menu?
- Look for vegetarian items?
The iteration behavior varies here

1. To iterate through the breakfast items we use the size() and get() methods on the ArrayList:

   ```java
   for (int i = 0; i < breakfastItems.size(); i++) {
       MenuItem menuItem = (MenuItem)breakfastItems.get(i);
   }
   ```

2. And to iterate through the lunch items we use the Array length field and the array subscript notation on the MenuItem Array.

   ```java
   for (int i = 0; i < lunchItems.length; i++) {
       MenuItem menuItem = lunchItems[i];
   }
   ```
Encapsulate it:

5. Now what if we create an object, let’s call it an Iterator, that encapsulates the way we iterate through a collection of objects? Let’s try this on the ArrayList.

```java
Iterator iterator = breakfastMenu.createIterator();
while (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem)iterator.next();
}
```

We ask the breakfastMenu for an iterator of its MenuItems.

4. Let’s try that on the Array too:

```java
Iterator iterator = lunchMenu.createIterator();
while (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem)iterator.next();
}
```

And while there are more items left...
Iterator interface and pattern

The `hasNext()` method tells us if there are more elements in the aggregate to iterate through.

The `next()` method returns the next object in the aggregate.

DinerMenuIterator is an implementation of Iterator that knows how to iterate over an array of MenuItems.
Add iterator to menu

```java
public interface Iterator {
    boolean hasNext();
    Object next();
}

public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;
    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }
    public Object next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }
    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```
Reworked diner menu with iterator

```java
public class DinerMenu implements Menu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // public MenuItem[] getMenuItems() {
    //     return menuItems;
    // }
    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }
    // other menu methods here
}
```
Reworked print menu method

```java
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;
    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }
    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n-----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }
    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem) iterator.next();
            System.out.print(menuItem.getName() + "", "");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
    // other methods here
}
```

Something still doesn’t look right...

- Reference to concrete class
- If we add another menu, waitress class needs to change again...
- We will get to it later
Test the code

```java
class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);
        waitress.printMenu();
    }
}
```

---

% java DinerMenuTestDrive

MENU

**BREAKFAST**
K&B’s Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

**LUNCH**
Vegetarian BLT, 2.99 -- (Fakin’) Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
Use java’s Iterator interface

```java
public Iterator createIterator() {
    return menuItems.iterator();
}
```

Instead of creating our own iterator now, we just call the iterator() method on the menuItems ArrayList.

Pancake menu is easy because it uses ArrayList. Just remove PancakeHouseMenulterator, import java.util.Iterator, change createIterator()
Use java’s Iterator interface

```java
import java.util.Iterator;
public class DinerMenuIterator implements Iterator {
    MenuItem[] list;
    int position = 0;
    public DinerMenuIterator(MenuItem[] list) {
        this.list = list;
    }
    public Object next() {
        // implementation here
    }
    public boolean hasNext() {
        // implementation here
    }
    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
            (“You can’t remove an item until you’ve done at least one next()”);
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```
public interface Menu {
    public Iterator createIterator();
}

import java.util.Iterator;
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }
    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }
    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + " ",
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
Single responsibility principle

A class should have only one reason to change

- When design class, assign it with one responsibility and one only
- A class should only do one thing and do it well

Cohesion: a measure of how closely a class or a module supports a single purpose or responsibility

- How closely methods/functions of a class are related
- High cohesion class == adhering to single responsibility principle

Single responsibility principle: intra-class structure; Least knowledge : inter-class

- High cohesion, low coupling == good design