CSE 1030

Yves Lespérance

Lecture Notes

Week 4 — Mixing Static and Non-Static Features

Recommended Readings:
Savitch Ch. 4 & 5 and Van Breugel & Roumani Ch. 3

Static Features in Non-Utility Classes

In addition to the usual instance attributes, constructors and instance methods, a non-utility class may also define some static features:

- class/static attributes/variables/constants — these are attributes of the whole class, not individual instances;
- class/static methods — these are operations on the class itself, not its instances.

Counting Instances

Suppose we wanted to keep track of how many credit cards (not necessarily different) have been created.

We can do this by adding a counter attribute to the `CreditCard` class, and incrementing it in the constructor(s). This counter should be a *static variable* because it is an attribute of the whole class, not its instances.

To declare it, we would add

```
private static int count = 0;
```

to the class definition. Note that we *initialize* the attribute as it is declared.

We would also change the constructor to increment the counter, e.g.

```
public CreditCard(int no, String aName, double aLimit)
 { assert 0 < no && no <= 999999 && aLimit > 0; // precon
   this.number = String.format("%06d", no) + "-";
   int digitSum = 0;
   while(no > 0)
   {  digitSum = digitSum + no % 10;
      no = no / 10;
   }
   this.number = number + (MOD - digitSum % MOD);
   this.name = aName;
   this.limit = aLimit;
   this.balance = 0;
   CreditCard.count++;  // new!
}
```

The other constructor works by delegating to the one above, so it need not be changed:

```
public CreditCard(int no, String aName)
{
   this(no, aName, DEFAULT_LIMIT);
}
```


We would also add a class/static method to retrieve the value of the counter:

```
public static int getCount()
{
   return CreditCard.count;
}
```

Then, we could use these as follows:

```
System.out.println("CreditCard count is "
                   + CreditCard.getCount());
CreditCard c1 = new CreditCard(703,"John");
System.out.println("CreditCard count is "
                   + CreditCard.getCount());
CreditCard c2 = new CreditCard(502,"Mary");
System.out.println("CreditCard count is "
                   + CreditCard.getCount());
```


Note that this only keeps track of how many times the class's constructors have been called. It does not count the number of distinct credit cards in existence or ensure that different credit card objects have different numbers. If we wanted to do this, we would have to maintain in a class attribute the set of existing credit cards and ensure that new credit card objects are unique with retpect to their number.

Stamping s Serial No. on Objects

We can easily stamp the value of the class's counter on each new object, to give it a serial number.

To do this we add an instance attribute for it

```
private int serialNo;
```

to the class definition.

We would also change the constructor to save the counter in the new object's serial number attribute, e.g.

```
public CreditCard(int no, String aName, double aLimit)
 { assert 0 < no && no <= 999999 && aLimit > 0; // precon
   this.number = String.format("%06d", no) + "-";
   ...
   this.number = number + (MOD - digitSum % MOD);
   this.name = aName;
   this.limit = aLimit;
   this.balance = 0;
   CreditCard.count++;
   this.serialNo = count;  // new!
}
```


We also define an accessor to retrieve the value of serial number attribute.

```
public int getSerialNo()
{
   return this.serialNo;
}
```

You use this as follows:

```
CreditCard c1 = new CreditCard(703,"John");
System.out.println("c1's serial no is "
                   + c1.getSerialNo());
```


Class Constants

A class may also define some *constants* for its users. For e.g., the class `CreditCard` defines the constant `DEFAULT_LIMIT`, to store the default limit amount. To declare `DEFAULT_LIMIT` for e.g., we write

```
public static final double DEFAULT_LIMIT = 1000.0;
```

in the `CreditCard` class definition.

These are *constant attributes of the class*, not of its instances. You must refer to them using the class name, e.g.

```
System.out.println(CreditCard.DEFAULT_LIMIT);
```


Class Methods to Check Argument Legality

Suppose we want to add a method `isLegal(int no)` to the `CreditCard` class to allow users to check whether the passed number would be a legal credit card number (before calling the constructor):

```
public static boolean isLegal(int no)
{
   return (0 < no && no <= MAX_NO);
}
```

This cannot be an instance method because there is no instance yet. We can make it a *class method*. It would belong to the class, not to one of its instances. The method must be called *on the class*, e.g.

```
if (CreditCard.isLegal(703)) ...
```

Maintaining a Singleton

Ensuring at most one instance of a class.

May be approriate to save ressources, e.g. database connection.

Prevent client from using constructors by making them private.

Provide static method `getInstance()` to clients to obtain the unique instance.

Store the instance in a static attribute.

Can create the instance when attribute is initialized.

Can call constructors from inside the class, even if they are private.

Can also create the instance only when `getInstance()` is first called.

Enforcing One instance per State

Amounts to maintaining singleton for each state.

Prevent client from using constructors by making them private.

Provide static method `getInstance(instance attribute values)` to clients to obtain an instance with these attribute values.

Store instances in a collection and retrieve them from it as necessary.


```
private static Map<String, Rectangle> instances =
   new TreeMap<String, Rectangle>();

public static Rectangle getInstance(int width, int height)
{
   String key = width + "-" + height;
   Rectangle instance = Rectangle.instances.get(key);
   if (instance == null)
   {
      instance = new Rectangle(width, height);
      Rectangle.instances.put(key, instance);
   }
   return instance;
}
```


Note that the key of a map should be made from immutable attributes of the object.

To make an attribute immutable, ensure that there is no mutator for it and that no method changes its value. (The class should also be made `final` to ensure no one can define a subclass that has mutators.)

An object is immutable if all its attributes (i.e. its whole state) are immutable, e.g. `String`, `Integer`, etc. To change an immutable attribute we have to create a new one.

Loop Invariants

A *loop invariant* is a property/boolean expression that holds at the beginning of every iteration of a loop.

Can be used to verify loop algorithm/code is correct.

If loop invariant is true before the first iteration,

and if whenever it is true at the beginning of an iteration, it must be true at the end,

then it will be true at the beginning and end of every iteration.


```
public static int pow(int a, int b)
{
  int pow = ?
  for (int i = 0; i < ??; i++)
  {
     pow = pow * a;
  }
  return pow;
}
```

If we make `?` be `1`, then have loop invariant $pow = a^i$.

If loop exits, the exit condition must be true.

Exit condition together with the loop invariant may be sufficient to show that the loop achieves its postcondition.

In the example, if we make `??` be `b`, then loop exit condition $i = b$ together with loop invariant $pow = a^i$ implies $pow = a^b$, which is postcondition of method.

Actually loop exit condition is really $i \geq b$, but it can also be shown that i is never greater than b.

To guarantee correctness, must also show that the loop condition must eventually become false.

Class Invariants

A *class invariant* is a property of the state of any class instance that must always hold.

E.g. `this.width >= 0 && this.height >= 0` for `Rectangle` class.

E.g. `this.balance <= this.limit` for `CreditCard` class.

Must be true after each constructor invocation.

If true before a public method is invoked it must be true after.

During the method invocation, it need not always be true. E.g. ensuring that tax is correctly calculated while adding an iten to an order.