CS 213 : Software Methodology

Spring 2023

Sesh Venugopal

Jan 30

Design Aspects of Static Members

Why Static?
Design Aspects

Static for Non-Object Oriented Programming

Static for Non Object-Oriented Programming

Suppose you want to write a program that just echoes whatever is typed in:

```
public class Echo {
    public static void main(String[] args)
    throws IOException {
        BufferedReader br = new BufferedReader(
                                new InputStreamReader(System.in));
        System.out.print("> ");
        String line = br.readLine();
        System.out.println(line);
    }
}
```

This program works without having to create any `Echo` objects – the Virtual Machine executes the main method directly on the `Echo` class (not via an `Echo` object) because the main method is declared static

Calling the main method directly on the class makes the design NOT object-oriented: Object orientation implies that there is an object or an instance of which a field is accessed, or on which a method is executed

Static Methods for “Functions”

Static Methods for "Functions"

An extreme use of static methods is in the `java.lang.Math` class in which every single method is static – why?

```
public class Math {
   public static float abs(float a) {...}
   ...
   public static int max(int a, int b) {...}
   ...
   public static double sqrt(double a) {...}
   ...
}
```

The reason is that every method implements a self-sufficient mathematical function with inputs and result: once the function returns, there is nothing to be kept around (as in a field of an object) for later recall/use.

In other words there is no state to be maintained

The `Math` methods can be called directly on the class, for example:

```
double sqroot = Math.sqrt(35);
```

In fact, you CANNOT create an instance of the `Math` class - "instantiation" is not allowed

Static Fields for Constants

Static Fields for Constants

`Math` is a "utility" class because *all* methods are static or "utility" methods – the class is just an umbrella under which a whole lot of math functions are gathered together

Aside from the utility methods, the `Math` class also has two static fields to store the values for the constants `E` (natural log base e) and `PI` (for the constant pi)

```
public class Math {
    ...
    public static final double E ...
    public static final double PI ...
    ...
}
```

final means it can be never be assigned to afterward, so initialization MUST be done at declaration time
(hence, a constant)

~~`Math.PI = Math.PI * 2;`~~

Since the constants are static, they can be accessed via class names (without objects):

```
double area = Math.PI * radius * radius;
```

Static Fields for Sharing Among Instances

Static Fields for Sharing Among Instances

Consider a class for which only a limited number of instances are allowed.

For instance, some kind of ecological simulation that populates a forest with tigers – want to put a limit on number of tigers

Need to keep track of current count, IN THE TIGER CLASS

Whenever an attempt is made to create a new Tiger instance, count has to be checked, and if under limit, then count has to be incremented

And whenever a Tiger instance is no longer in play (say a Tiger dies or is transported to another location), the count of tigers has to be decremented

Tiger – Static field count

```
public class Tiger {
    public static final int MAX_COUNT=10;
    public static final int MAX_MASS=2000;
    private static int count=0;
    public Tiger(int mass)
    throws Exception {
        if (count == MAX_COUNT) {
            throw new Exception("Max count exceeded");
        }
        if (mass < 0 || mass > MAX_MASS) {
            throw new IllegalArgumentException("Unacceptable mass");
        }
        count++;
    }
    ...
}
```

Class property, shared by instances

This is a "checked" exception, so the constructor must declare a throws

"Unchecked/runtime" exception, no throws declaration needed (but it is a subclass of Exception, so is covered by the throws Exception declaration)

Tiger – Static count field shared by instances

```
public class Tiger {

    public static final int MAX_COUNT=10;
    public static final int MAX_MASS=2000;

    private static int count=0;

    public Tiger(int mass)
    throws Exception {
       ...
       count++;

    }


    public static int getCount() {
       return count;
    }

}
```

A client would want to know how many Tiger instances are around BEFORE creating (or not) another instance

Since `count` is private and static, it has to be accessed via a method that is a property of the class, not of an instance, i.e. the method is `static`.

Static: Access

- Static fields and methods are accessed via the class name, or if they are mixed in with instance fields and methods, they *may* be accessed via an instance of the class:

Since the Tiger constructor throws a checked exception, the calling method, `main`, must either catch it, or throw it

```
public class Application {
    public static void main(String[] args)
    throws Exception {
        int m = Tiger.MAX_MASS;   // use class name to get MAX_MASS
        Tiger t = new Tiger(m-100);

        int c = t.getCount();    // using instance to get count
        ...
    }
}
```

You may use an instance to acess a static field or method, but it is not good practice

Static: Access

- The part of the application you are working on may not be the only one creating `Tiger` instances. So, even for the first instance you want to create, you need to know count before you decide whether you can create another instance or not.

```
int currCount = Tiger.getCount();  // use class name

if (currCount < Tiger.MAX_COUNT) {
   Tiger t= new Tiger(...);
    ...
} else {
   . . . // do whatever
}
```

Always use class name to get at static members of a class, even in situations where you can use an instance, so that your code adheres to the design implication of static