CS 213 : Software Methodology

Spring 2019

Sesh Venugopal

Lecture 4: Jan 31

Design Aspects of Static Members - 2

Why Static?
Design Aspects

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 bound on number of tigers

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

Every time a new Tiger instance is attempted to be created, count has to be checked, and if ok, then count has to be incremented

And every time a Tiger instance goes out of scope (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 > 2000) {
          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

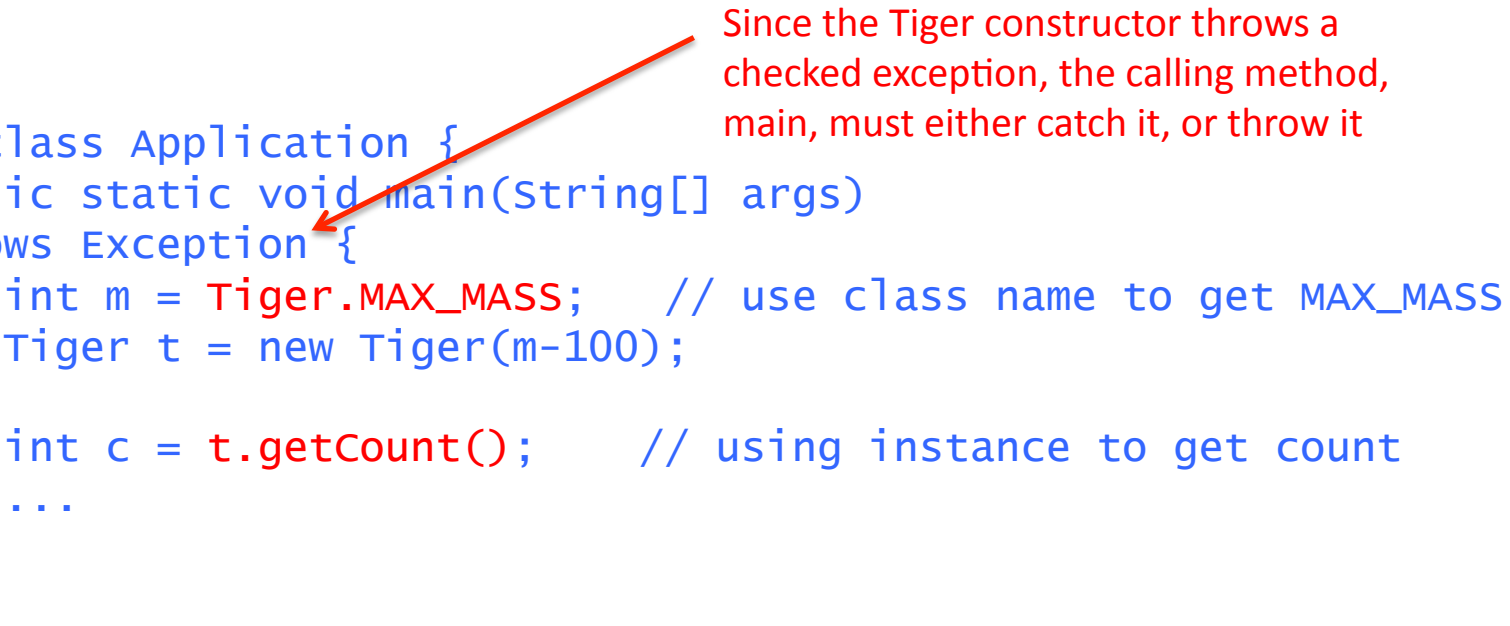
```
public class Tiger {

   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, 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:



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

Static/Non-Static Mix: Another Example

- Parsing a string into an integer, e.g. “123” -> 123 – where to provide this functionality?

  OPTIONS:

  - Have a `String` instance method, say, `parseAsInteger` that returns an `int`, e.g.

    ```
    int i = “123”.parseAsInteger();
    ```

    Bad design: An instance method should be applicable to ALL instances. But not all strings are parsable as integers

  - Have a `String` static method, say, `parseAsInteger` that returns an `int`, e.g.

    ```
    int i = String.parseAsInteger(“123”);
    ```

  - Have an `Integer` static method, say, `parseInt` that returns an `int`, e.g.

    ```
    int i = Integer.parseInt(“123”);
    ```

- Of the second and third choices, which one is better? Why? `Integer.parseInt` is better

  Think of converting strings to doubles, floats also –
  having all these types of conversions in `String` would require `String` to know about formats of other types, which is NOT its business.
  Best to localize custom functionality in the corresponding target (converted type) classes.