CS 213 : Software Methodology

Spring 2016

Lecture 4: Jan 28
Static Members of Class – Why/When
Inner Classes

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 it non 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"

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 mathematical function (i.e. a process with inputs and outputs), and 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

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

Apart 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 ...
   ...
}
```

Again, these constants can be directly accessed (without objects):

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

`E` and `PI` are constants because their values cannot be changed (`final`)

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

Static Fields for Sharing Among Instances

Another use of a static field is to record a data value as the property of the class, to be shared among all instances, but is not a constant (can be changed)

A classic example of this is a counter that keeps track of how many objects of a class are in existence:

```
public class Tiger {
   ...
   private static int count=0;

   public Tiger(...) {
      ...
      count++;
   }
   protected void finalize() throws Throwable {
      ...
      count--;
   }
   ...
}
```

Static (Class) Fields and Methods Mixed with Non-Static (Instance) Fields and Methods

```
public class Tiger {

   public static final MAX_MASS=2000;

   private static int count=0;

   public Tiger(int mass) {
      ...
      count++;
   }

   protected void finalize()
   throws Throwable {
      ...
      count--;
   }

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

A client needs 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:

```
public class Application {
    public static void main(String[] args) {
        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
        ...
    }
}
```

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 < maxCount) {
   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?

  CHOICES:

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

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

    Bad design: Parsing an int is not an inherent/characteristic property of a String – not all strings can be parsed 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?

  Static method in `Integer` is better. (Think of converting strings to doubles, floats also – having all these types of conversions in `String` would over-extend the `String` class, better to distribute/localize in the classes corresponding to the converted type.)

Inner Classes

```
public class LinkedList<T> {

   public static class Node<E> {  // inner class
      E data;
      Node<E> next;
      public Node(E data,
                  Node<E> next) {...}
   }

   Node<T> front;
   int size;
   ...
   public void addFront(T item) {
       front = new Node<T>(item, front);
       size++;
   }
   ...
}
```

Since nodes are the building blocks of linked lists, a `Node` class can be defined inside a linked list to emphasize this (will get to the static thing in a bit….)

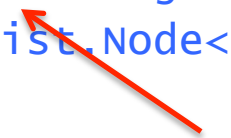
Inside the `LinkedList` class, references to the `Node` type are no different than if `Node` had been defined outside `LinkedList`

Inner Classes

```
public class LinkedList<T> {

   public static class Node<E> {  // inner class
      E data;
      Node<E> next;
      public Node(E data,
                  Node<E> next) {...}
   }
...
}

// in some application code outside of LinkedList class
LinkedList.Node<Integer> temp =
   new LinkedList.Node<Integer>(10, null);
```



Reference to `Node` needs to be qualified with `LinkedList` prefix

Non Static Inner Class

```
public class LinkedList<T> {

   public static class Node<E> {  // inner class
      E data;
      Node<E> next;
      public Node(E data,
                  Node<E> next) {...}
   }
...
}

// in some application code outside of LinkedList class
LinkedList<Integer>.Node<Integer> temp =
   new LinkedList<Integer>().new Node<Integer>(10, null);
```

Can only create a `Node` instance off of a `LinkedList` instance