CS 213 : Software Methodology

Spring 2017

Sesh Venugopal

Lecture 3: Jan 24

Inheritance: Private Fields/Static Members
Design Aspects of Static Members

Inheritance - Private Fields

```
public class Point {
   private int x,y;
   ...
}


public class ColoredPoint extends Point {
   // x and y inherited but HIDDEN
   ...
   public int getX() { // override inherited getX()
      return x;
   }
}
```

COMPILE?

WILL NOT COMPILE
because x is hidden

Inheritance - Private Fields

```
public class Point {
   private int x,y;
   ...
}
```

```
public class ColoredPoint extends Point {
   // x and y inherited but HIDDEN
   ...  // getX() is NOT overridden
}
```

```
public class PointApp {
   public static void
   main(String[] args) {
      ColoredPoint cp = new ColoredPoint(4,5,"blue");

      System.out.println(cp.x); // ?
```

WILL NOT COMPILE, x is hidden

```
      System.out.println(cp.getX()); // ? 4
```

Inherited `getX()` method is able to access the `x` field

```
}
```

Inheritance - Static Members

```
public class Supercl {
   static int x;
   public static void m() {
       System.out.println(
         "in class Supercl");
       }
   }
}
```

```
public class Subcl
extends Supercl { }
```

```
public class StaticTest {
   public static void main(String[] args) {
      Supercl supercl = new Supercl();
      System.out.println(supercl.x);  // ?  0
      supercl.m(); // ?  "in class Supercl"
      Subcl subcl = new Subcl();
      System.out.println(subcl.x); // ?  0 – inherited from Supercl
      subcl.m(); // ? "in class Supercl"  – inherited from Supercl
   }
}
```

Inheritance - Static Fields

```
public class Supercl {
   static int x;
   public static void m() {
      System.out.println("in class Supercl");
   }
}
```

```
public class Subcl
extends Supercl {
   int x=3;
}
```

```
public class StaticTest {
   public static void main(String[] args) {
      Subcl subcl  = new Subcl();
      System.out.println(subcl.x); // ? 3 – instance field x
      Supercl supercl  = new Subcl();
      System.out.println(supercl.x); // ? 0 – inherited static field x  ! ! !
   }
}
```

static type

dynamic type

INHERITED STATIC FIELDS ARE STATICALLY BOUND (TO REFERENCE TYPE), NOT DYNAMICALLY BOUND (TO INSTANCE TYPE)

Static Method Call Binding

```
public class Sorter {

   public static void
   sort(String[] names) {
      System.out.println(
        "simple sort";
      }
   }
}
```

```
public class IllustratedSorter
extends Sorter {

   // override
   public static void
   sort(String[] names)
      System.out.println(
        "illustrated sort";
      }
   }
}
```

```
Sorter p = new IllustratedSorter();

p.sort(); // ? "simple sort"
```

static type

dynamic type

`sort()` is statically bound to `p`, meaning since `Sorter` is the static type of `p`, the `sort()` method in `Sorter` is called

Why Static?
Design Aspects

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

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


   public static int getCount() {
      return count;
   }


}
```

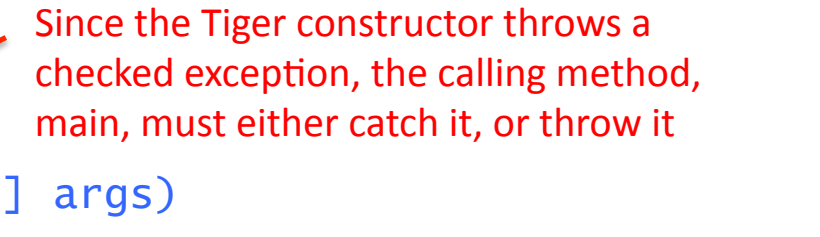
Overrides method inherited from Object class, called by garbage collector when object goes out of scope

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:




```
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
        ...
    }
}
```

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