CS 213 : Software Methodology

Fall 2020

Sesh Venugopal

Design Aspects of Static Members

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 stand-alone functions

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

```
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 Non-Constant 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 > MAX_MASS) {
          throw new IllegalArgumentException("Unacceptable mass");
       }
       count++
    }
    ...
}
```

Class property, shared by instances

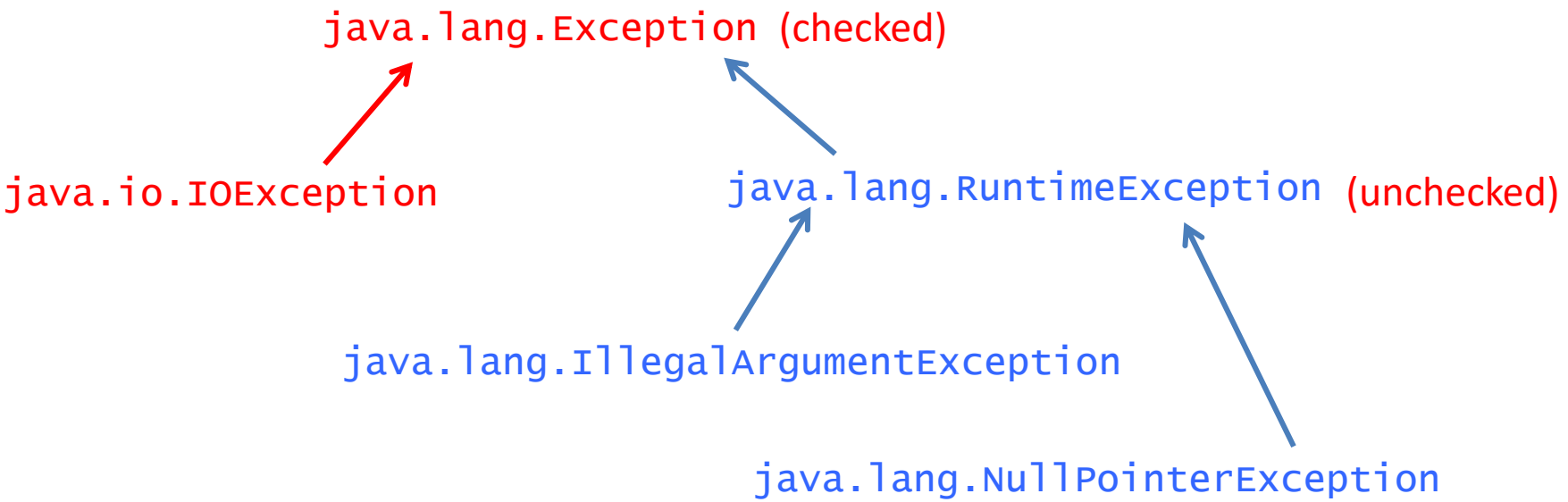
Checked and Unchecked Exceptions

```
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++
    }
    ...
}
```

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

"Unchecked/runtime" exception, no throws declaration needed in header

Checked and Unchecked Exceptions



Checked or Unchecked Exception: How to Decide Which to Use?

Use checked exception if error is not fatal – an application can find a way to recover and move on

Compiler insists on throws clause to make the application aware of the exception. The application can then be forced to either try/catch the exception (if it has a way to recover), or pass the buck by throwing it in turn – either way the exception can't be ignored

Use unchecked exception if error is fatal – an application will expect not to recover from the error

Since a throws clause is not required, an application wouldn't know the exception is thrown - in general, the source code might not be available when using libraries. So the application wouldn't know to trap the exception with a try/catch. So the exception would travel all the way back to the VM, which would stop the program (crash)

Checked and Unchecked Exceptions

```
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++
    }
    ...
}
```

Unchecked exception, no throws declaration needed (*but it is a subclass of Exception two levels down, so is covered by the throws Exception declaration – an IllegalArgumentException is also a plain old Exception*)

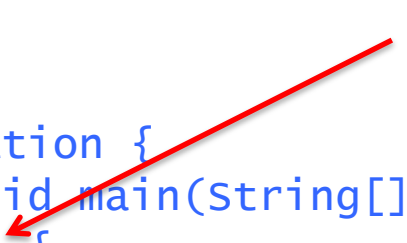
Tiger – Static count field shared by instances

```
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;
    }

}
```

Since count is static, the method should preferably be static.

Static: Access

- Static fields and methods are typically accessed via the class name, but if the class has instances, then the static members *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

Static/Non-Static Mix: A Design 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.