Poznan University of Technology

Object Oriented Programming

Jakub Piotr Hamerliński, M.Eng.

Object Oriented Programming

Agenda

- Exceptions (Java) and mistakes in exception handling
- Solution to Utility class in OOP
- Task

Exception handling in Java

Object Oriented Programming

Exceptions in Java

Java has much more complex exceptions handling than C++.
In Java, apart from classic **try**, **catch**, and **throw**, we have **throws** keyword.
The **throws** keyword is used to declare which exceptions can be thrown from a method, while the **throw** keyword is used to explicitly throw an exception within a method or block of code.

```
1   @Override
2   public void read() throws IOException {
3     File file = new File(this.filePath);
4     FileInputStream stream = new FileInputStream(file);
5     int i;
6     while ((i = stream.read()) != -1) {
7       Character character = new Character(i);
8       System.out.print(character.value());
9     }
10    stream.close();
11  }
```

Object Oriented Programming

Types of exceptions in Java

Checked exceptions are caught at compile time whereas a runtime or unchecked exceptions are, as it states, at runtime.
A **checked exceptions must be handled** either by re-throwing or with a try catch block, **a runtime are not required to be handled**.

Object Oriented Programming

Approach to exceptions in Java - Good rules

There is long lasting debate whether which type is better or which is wrong.
I believe that:

- We should always use checked exceptions,
- We should neither throw nor use unchecked exceptions.
- We should always declare one exception type in the throws block.
- **We should never catch without re-throwing**.

All methods in all Java interfaces should be declared either as **safe i.e. throws nothing** or **unsafe i.e. throws Exception**. In that case everything becomes logical and clear. If you want to stay safe, take responsibility for failure handling. Otherwise, be unsafe and let your users worry about safety.

Object Oriented Programming

Approach to exceptions in Java - Exception re-throwing

Re-throwing is approach where after we catch exception, we throw it further:

```
if (!file.exists()) {
  throw new IllegalArgumentException(
    String.format(
      "User profile file %s doesn't exist",
      file.getAbsolutePath()
    )
  );
}
```

```
try {
  Files.delete(file);
} catch (IOException ex) {
  throw new IllegalArgumentException(
    String.format(
      "Can't delete user profile data file %s",
      file.getAbsolutePath()
    ),
    ex
  );
}
```

Solution to Utility class in OOP

Object Oriented Programming
Solution to Utility class in OOP

A utility class (aka helper class) is a "structure" that has only static methods and encapsulates no state. StringUtils, IOUtils, FileUtils from Apache Commons; Iterables and Iterators from Guava, and Files from JDK7 are perfect examples of utility classes.
Here, we want to follow the DRY principle and avoid duplication. Therefore, we place common code blocks into utility classes and reuse them when necessary:

```
1  // This is a terrible design, don't reuse
2  public class NumberUtils {
3    public static int max(int a, int b) {
4      return a > b ? a : b;
5    }
6  }
```

Seems neat but it is a terrible practice. Why? Here's why!

Object Oriented Programming

Solution to Utility class in OOP

Utility classes are not proper objects; therefore, they don't fit into object-oriented world. They were inherited from procedural programming, mostly because we were used to a functional decomposition paradigm back then.

Object Oriented Programming

Solution to Utility class in OOP

In an object-oriented paradigm, we should instantiate and compose objects, thus letting them manage data when and how they desire. Instead of calling supplementary static functions, we should create objects that are capable of exposing the behavior we are seeking.

Object Oriented Programming

Solution to Utility class in OOP - example

```
// This is a terrible design, don't reuse
void transform(File in, File out) {
  Collection<String> src = FileUtils.readLines(in, "UTF-8");
  Collection<String> dest = new ArrayList<>(src.size());
  for (String line : src) {
    dest.add(line.trim());
  }
  FileUtils.writeLines(out, dest, "UTF-8");
}
```

```
void transform(File in, File out) {
  Collection<String> src = new Trimmed(
    new FileLines(new UnicodeFile(in))
  );
  Collection<String> dest = new FileLines(
    new UnicodeFile(out)
  );
  dest.addAll(src);
}
```

Object Oriented Programming

Tasks

1)Refactor NumberUtils class in Java or C++, so it will be **truly object oriented** (no static methods). Show usage of the refactored class.

```java
public class NumberUtils {
  public static int max(int a, int b) { return a > b ? a : b;}
  public static int min(int a, int b) { return a < b ? a : b;}
  public static float avg(int a, int b) { return (a + b) / 2;}
}
```

```cpp
class NumberUtils {
 public:
  static int max(int a, int b) { return a > b ? a : b;}
  static int min(int a, int b) { return a < b ? a : b;}
  static float avg(int a, int b) { return (a + b) / 2;}
};
```

Object Oriented Programming
Tasks

2)Refactor Logarithm class in Java or C++, so it will be **error-proof** (try-catch-throw). Show usage of the refactored class.

```
1   public class Logarithm implements Number {
2     private double base, argument;
3     @Override
4     public double doubleValue() { return Math.log(this.argument) / Math.log(this.base);}
5     public Logarithm(double inputBase, double inputArgument) {
6       this.base = inputBase;
7       this.argument = inputArgument;
8     }
9   }
```

Object Oriented Programming
Tasks

2)Refactor Logarithm class in Java or C++, so it will be **error-proof** (try-catch-throw). Show usage of the refactored class.

```
#include <cmath>
class Logarithm : public Number {
 private:
  double base, argument;
 public:
  double doubleValue() { return log(this->argument) / log(this->base);}
  Logarithm(double inputBase, double inputArgument) {
    this->base = inputBase;
    this->argument = inputArgument;
  }
};
```