# Exceptions/Debugging/ Annotations/RegEx in Java

# Exceptions Overview

- A prescription for handling errors in the Java language
- Not meant as flow control for normal execution; these are exceptions after all.
- There are two types of exceptions in Java
  - “Checked” exceptions
  - “Unchecked” exceptions

# How not to handle exceptions when coding (in any language)

- Returning a “special” value 不要使用!
  - Returning `null` or `-1`
  - Can be confusing - type does not document the exception
  - Can be restrictive - what if the value you need to return is actually `-1`?
- Instead use exceptions (most languages have a notion of exceptions) but only if the case is indeed exceptional (not simply for flow-control)
- If an exception does not make sense, return a type (or wrapper type) detailing the issue

object

# Exception Example

```
1  public class ExceptionsExample {
2
3      public static void main(String[] args) {
4          try {
5              ExceptionsExample example = new ExceptionsExample(null);
6          } catch (NullPointerException npe) {
7              System.out.printf("Cannot pass a null value to ExceptionsExample%n");
8          }
9      }
10
11     private final String value;
12
13     public ExceptionsExample(String value) {
14         if (value == null) {
15             throw new NullPointerException();
16         }
17         this.value = value;
18     }
19
20 }
```

# Checked Exceptions

- Must be caught or rethrown
- Catching or re-throwing is statically “checked” by the compiler
- In general, only use if the program can recover from this failure. Does this exception have a graceful route to recovery.
  - Otherwise, code becomes bloated with exception handling that isn’t aiding in recovery but simply appeasing the compiler.

# Checked Exception Example

```
1  public class CheckedExceptionExample {
2
3      public void printJson(byte[] json) {
4          try {
5              String parsed = parseJson(json);
6              System.out.printf("%s%n", parsed);
7          } catch (IOException ioe) {
8              System.out.printf("Could not parse JSON - %s%n", ioe.getMessage());
9          }
10     }
11
12     public String parseJson(byte[] json) throws IOException {
13         StringBuilder buffer = new StringBuilder();
14         for (byte data : json) {
15             if (data > 0x20) {
16                 throw new IOException();
17             }
18             buffer.append((char) data);
19         }
20         return buffer.toString();
21     }
22
23 }
```

declare for checked exception

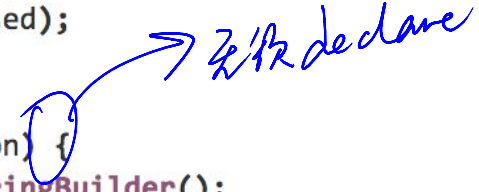
must handle or propagate

# Unchecked Exceptions

- Need not be caught, rethrown or declared
- In general, only use when the program cannot be expected to recover from the failure.
- Callers can choose to catch these exceptions (and recover) they're just not required by the compiler.

# Unchecked Exceptions Example

```
1  public class UncheckedExceptionExample {
2
3      public void printJson(byte[] json) {
4          String parsed = parseJson(json);
5          System.out.printf("%s%n", parsed);
6      }
7
8      public String parseJson(byte[] json) {
9          StringBuilder buffer = new StringBuilder();
10         for (byte data : json) {
11             if (data > 0x20) {
12                 throw new IllegalArgumentException();
13             }
14             buffer.append((char) data);
15         }
16         return buffer.toString();
17     }
18
19 }
```



# Throwing Exceptions

- Done by using keyword `throw` (i.e., `throw new NullPointerException()`)
- Must declare checked exceptions as being 'thrown'
- Do not have to but can declare unchecked exceptions
  - Not often done. If you do this it does not mean the caller must catch the exception
- Cannot override a method and throw a more generic exception
  - But can override and throw a more specific exception
- If overriding a method without a declared checked exception, cannot add one

# Catching Exceptions

- Use the try/catch mechanism provided by Java
- If you do not catch the type thrown it is handled by the caller
  - If the exception is checked you must either catch it or declare your method as throwing it
  - If the exception is unchecked and not handled up the call-stack, then the JVM stops because of the “uncaught” exception
- If no exception is thrown within the `try` block then the catch block is not executed. Additionally, if the type of the `catch` block does not match the thrown exception then the `catch` block is not executed.
- Multiple catch blocks are handled in order of declaration
- Cannot catch checked exceptions which are not thrown.

# Exceptions are Throwables are Objects

- “Exceptions” in Java are actually a subtype of `Throwable`
- There are two main types that extend `Throwable`
  - `Exception`
    - Includes checked and unchecked exceptions.
    - Unchecked exceptions should extend from `RuntimeException`
  - `Error`
    - An exception indicating a system failure (i.e., no more memory).
    - In general, should not extend this class and very rarely throw it yourself

# Hierarchy When Catching

- More generic exception in hierarchy can handle more specific.
  - I.e., parent types can catch all of their children
  - Because of this, order matters. What does the following print?

```
1  public void readFile(String fileName) {
2      try {
3          if (true) {
4              throw new FileNotFoundException();
5          }
6      } catch (IOException ioe) {
7          System.out.printf("IOException thrown and caught");
8      } catch (FileNotFoundException fnfe) {
9          System.out.printf("FileNotFoundException thrown and caught");
10     }
11 }
```

# Multiple Exception Types in Catch

- Java 1.7 added ability to handle disparate exception types in the same catch statement.

```
1  public class MultipleTypeInCatch {
2
3      public void multipleCatch() {
4
5          try {
6              // do something
7          } catch (IllegalArgumentException | NullPointerException e) {
8              System.out.printf("Something bad happened %s%n", e.getMessage());
9          }
10
11     }
12
13 }
```

# Custom Exceptions

- You can extend `Throwable` / `Exception` / `RuntimeException` / etc to create your own exception type.

```
1  public class ParsingException extends Exception {
2
3      public ParsingException(Throwable cause) {
4          super(cause);
5      }
6
7  }
```

- Not often necessary. Prefer existing exceptions (NullPointerException, IllegalArgumentException, IOException, etc)

# Rethrowing / Chaining Pattern

- To simplify your method signature (i.e., API) only throw one checked exception. Catch others and rethrow as one type.
  - Not always necessary, use discretion.

```
1  public class RethrowChainingExample {
2
3      public void read(String filePath) {
4          try {
5              FileInputStream stream = new FileInputStream(filePath);
6              int read;
7              while ((read = stream.read()) != -1) {
8                  if (read > 0xF) {
9                      throw new ParsingException();
10                 }
11                 System.out.printf("%d", read);
12             }
13         } catch (IOException fnfe) {
14             throw new ParsingException(fnfe);
15         }
16
17     }
18
19 }
```

# Finally Clause

- Used in conjunction with the try block, to allow code to be executed no matter whether an exception is thrown or not

```
1  public class FinallyClause {
2
3      public void finallyClause(String path) {
4          InputStream stream = null;
5          try {
6              stream = new FileInputStream(path);
7          } catch (IOException ioe) {
8              throw new RuntimeException(ioe);
9          } finally {
10             if (stream != null) {
11                 try {
12                     stream.close();
13                 } catch (IOException ioe) {
14                     // ignore; trying to close anyway
15                 }
16             }
17         }
18
19
20 }
```

# Finally Clause Gotcha

```
1  public class FinallyGotcha {
2
3      public static void main(String[] args) {
4          FinallyGotcha gotcha = new FinallyGotcha();
5          int result = gotcha.finallyGotcha();
6          System.out.printf("%d%n", result);
7      }
8
9      public int finallyGotcha() {
10         try {
11             if (true) {
12                 throw new RuntimeException();
13             }
14             return 1;
15         } catch (RuntimeException re) {
16             return 2;
17         } finally {
18             return 3;
19         }
20     }
21
22 }
```

不会从这里return直到执行到finally

# Try With Resources (Java 1.7)

- Introduced in Java 1.7 - handy way to close resources.
  - Avoids the boilerplate code seen two slides previously
  - Can create your own using AutoCloseable interface

```
1  public class FinallyClause {
2
3      public void finallyClause(String path) {
4          try (InputStream stream = new FileInputStream(path)) {
5              // TODO
6          } catch (IOException ioe) {
7              throw new RuntimeException(ioe);
8          }
9      }
10
11 }
```

# Exceptions and Logging

- **Never** print the exception stack-trace using `e.printStackTrace();`
  - As this often simply squashes the exception handling.
- Instead prefer a Logging implementation
  - Java provides a common API - **java.util.logger**
  - **log4j** is a popular library but outdated
  - Latest/best is **slf4j**

```
1  public class ExceptionLogging {
2
3      private final static Logger LOG = Logger.getLogger(ExceptionLogging.class.getSimpleName());
4
5      public InputStream open(String file) {
6          try {
7              return new FileInputStream(file);
8          } catch (IOException ioe) {
9              LOG.log(Level.SEVERE, ioe.getMessage(), ioe);
10             return null;
11         }
12     }
```

# Debugging!

- Do early and often.
- Do not litter log statements into your program, instead use the debugger (jdb or an IDE).
- Always compile your code with **-Xlint:all**
- Do not debug code by adding a **main** method for testing. Test your code using unit-tests instead. Debugging by **main** methods pollutes your code and adds unnecessary dependencies

# Annotations

- Form of meta programming
  - I.e., annotates your code with metadata which can be used.
- Adds metadata to your code which can be used by programmers for clarity as well as compilers/tooling to do static code analysis
  - We've already seen this with **@Override**

```
21      @Override public int hashCode() {
22          return variable != null ? variable.hashCode() : 0;
23      }
```

# Creating Your Own

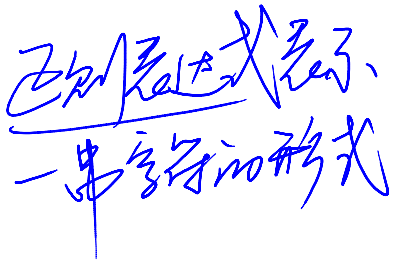
```
1  @Target(ElementType.METHOD)
2  @Retention(RetentionPolicy.RUNTIME)
3  public @interface Authorized {
4
5      String headerKey() default "Bearer";
6
7  }
```

annotation实现

annotation名字

# Regular Expressions

- Encourage you to read the Java tutorial
  - http://docs.oracle.com/javase/tutorial/essential/regex/
- Similar to *Perl* in syntax
- Two main classes
  - `java.util.regex.Pattern`
  - `java.util.regex.Matcher`



```java
public class RegexExample {

    private static final Pattern REG_EX = Pattern.compile("\\d\\d");

    public static void main(String[] args) {
        RegexExample example = new RegexExample();
        example.match("Foo 01 Bar");

    }

    private void match(String input) {
        Matcher matcher = REG_EX.matcher(input);
        while (matcher.find()) {
            String match = matcher.group();
            System.out.printf("%s%n", match);
        }
    }

}
```

# Read Chapter 12

All sections except 2.9

# Homework 6

https://github.com/NYU-CS9053/Spring-2016/homework/week6