Module 16

- Review of Previous Week
- What are Exceptions?
- Why do we need them?
- How can we use them?
- Errors vs Exceptions
- Additional resources

Review of Previous Week

- HTTP and HTTP Methods.
- Request and response.
- Request handlers.
- Long running process.
- Lambdas.
- JSON. Why do we need it?
- What is Spark?
- What is Gradle?

What are Exceptions?

An exception is any event that occurs while a program is running that stops the intended flow of the application.

Examples

- The program tries to read a file that has been deleted.
- The program tries to parse text to JSON but it receives a different format.
- The program asks a user to provide input of one type, but the user gives an unexpected format.

The code where an error occurs is not the code that knows how to properly deal with the error.

Take the example of reading a file, the code involved with reading a file is often written to be very reusable.

It knows very little about the reasons *why* a file is being read - it just knows how to accept a file location and perhaps file type.

The code where an error occurs is not the code that knows how to properly deal with the error.

Adding the info about what kind of message to give to a user, or how to recover would make the reading file code very complicated and specific.

Instead the reading file code can let the rest of the application know that an exception has occurred and we can keep the reading file code small and handle the exception elsewhere.

Why do we need them?

As programmers, we only have control over our applications while we are writing them.

Once they are running in the wild unexpected things occur: users provide invalid input, files are deleted or corrupted, networks go down!

If we don't handle these unexpected events our programs will crash and in some extreme cases our system memory or hardware can be damaged.

Exceptions provide a way for us to prevent serious damage and guard against the most error prone components of our applications.

How can we use them?

Exceptions sound pretty rough! Why would we want to use them?

1. Some parts of our applications are more error prone than others, we can put rules around these to force us to deal with the fact that these are dangerous spots.
2. We can cause a specific type of exception to occur in order to prevent more serious problems from occurring.
3. We can create code that deals with specific types of problems and recovers from them.
4. We can create code that deals with general, unforeseen problems.

Custom `Exception`s

```
class InvalidUserGradeException extends Exception {
  private String givenGrade;

  public InvalidUserGradeException(String givenGrade) {
    super("Invalid User Grade");
    this.givenGrade = givenGrade;
  }

  public InvalidUserGradeException(String msg, String givenGrad
    super(msg);
    this.givenGrade = givenGrade;
  }

  public String getGivenGrade() {
    return givenGrade;
  }
}
```

The `throw` keyword

```
if (grade < 0 || grade > 100) {
  throw new InvalidUserGradeException();
}
```

The `throws` keyword

```
class Student {
  // code...

  public void setGrade(int grade) throws InvalidUserGradeExcept
    // code...
  }

  // code...
}
```

`throws` with `throw`

```
// code/Sample.java
public class Sample {
  public static void main(String[] args)
      throws IllegalArgumentException, InvalidUserGradeExceptic
    if (args.length == 0) {
      throw new IllegalArgumentException("Expecting a grade arg
    } else {
      int grade = Integer.parseInt(args[0]);

      if (grade < 0 || grade > 100) {
        throw new InvalidUserGradeException(Integer.toString(gr
      } else {
        System.out.println("The user's grade is now " + grade);
      }
    }
  }
}
```

The `try` / `catch` / `finally` keywords

```
try {
  // Code that may throw an exception
} catch (Exception ex) {
  // Handle the error and do something else
} finally {
  // Code to run after success or failure
}
```

`try`

```
try {
  // code
}
// catch and finally blocks...
```

`catch`

```
try {
  // code...
} catch (ExceptionType ex) {
  // code...
} catch (ExceptionType ex) {
  // code...
} catch (ExceptionType ex) {
  // code...
}
```

Which `catch` block will run with this code?

```
try {
  throw new ArithmeticException();
} catch (NullPointerException ex) {
  System.out.println("Got NullPointerException");
} catch (ArrayIndexOutOfBoundsException ex) {
  System.out.println("Got ArrayIndexOutOfBoundsException");
} catch (ArithmeticException ex) {
  System.out.println("Got ArithmeticException");
} catch (Exception ex) {
  System.out.println("Got any Exception");
}
```

finally

```
// See code/Finally.java
try {
  System.out.println("The `try` code block is running.");
  throw new Exception();
} catch (Exception ex) {
  System.out.println("The `catch` code block is running.");
  return;
} finally {
  System.out.println("The `finally` code block is running.");
}
```

`catch` and/or `finally`

You need at least one `catch` or one `finally` block. A `try` block with no `catch/finally` is not allowed. And order matters, `try` must be first, `catch` next, and `finally` always goes last.

Will this compile and/or run?

```
try {
  System.out.println("Running code");
}
```

Will this compile and/or run?

```
try {
  alwaysThrowsException();
} finally {
  System.out.println("In finally block.");
} catch (NullPointerException ex) {
  System.out.println("An exception was thrown.");
}
```

Will this compile and/or run?

```
finally {
  System.out.println("In finally block.");
} catch (NullPointerException ex) {
  System.out.println("An exception was thrown.");
} try {
  alwaysThrowsException();
}
```

Will this compile and/or run?

```
try {
  alwaysThrowsException();
} catch {
  System.out.println("An exception was thrown.");
}
```

Will this compile and/or run?

```
try {
  alwaysThrowsException();
} catch (NullPointerException ex) {
  System.out.println("An exception was thrown.");
}
```

Will this compile and/or run?

```
try {
  alwaysThrowsException();
} catch (NullPointerException ex) {
  System.out.println("An exception was thrown.");
} finally {
  System.out.println("In finally block.");
}
```

Information found in `Exception`s

- `public String getMessage()`: *Returns the detail message string...* [4]
- `public StackTraceElement[] getStackTrace()`: *Provides programmatic access to the stack trace information printed by printStackTrace()*. [5]
- `public void printStackTrace()`: *Prints this throwable and its backtrace to the standard error stream*. [6]

Example stack trace: `StacktraceExample.java`

```java
class MyException extends Exception {}

public class StacktraceExample {
  public static void main(String[] a) throws MyException {
    one();
  }

  public static void one() throws MyException {
    two();
  }

  public static void two() throws MyException {
    three();
  }

  public static void three() throws MyException {
    throw new MyException();
  }
}
```

Example stack trace: output

```
$ javac StacktraceExample.java
$ java StacktraceExample
Exception in thread "main" MyException
        at Program.three(StacktraceExample.java:17)
        at Program.two(StacktraceExample.java:13)
        at Program.one(StacktraceExample.java:9)
        at Program.main(StacktraceExample.java:5)
```

Errors vs Exceptions

Errors and Exceptions both inherit from the `java.lang.Throwable` Object, but they have one clear distinction.

Errors represent catastrophic events that cannot be recovered from: for example running out of memory or other system resources. The application can't fix the environment it is running in, or install additional memory.

Exceptions represent problems that can be recovered from: for example trying to read a file that has been deleted. The application can simply tell the user that the file was unable to be found and move on.

Common exception classes

- `NullPointerException` : *Thrown when an application attempts to use null in a case where an object is required.* [1]
- `ArrayIndexOutOfBoundsException` : *Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.* [2]
- `ArithmeticException` : *Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.* [3]

Additional resources

- *Lesson 16: Exceptions* by Michael Fudge
  https://youtu.be/R86ObiKhMNc

Reference list

1. https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html
2. https://docs.oracle.com/javase/7/docs/api/java/lang/ArrayIndexOutOfBoundsException.html
3. https://docs.oracle.com/javase/7/docs/api/java/lang/ArithmeticException.html
4. https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html#getMessage()
5. https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html#getStackTrace()
6. https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html#printStackTrace()