Exceptions

What are Exceptions?

- They represent an exceptional event - an error that occurs during runtime
- They allow controlled management of errors
- They are objects: classes form an extensible exception hierarchy
- They may be caught and handled, or propagated
- When an error occurs within a method it creates an object and hands it off to the runtime system
- The exception object contains information about:
    - the error
    - error type
    - the state of the program when the error occured
- After a method throws an exception
    - The runtime system attempts to find something to handle it
    - It checks the methods that had been called to get to the method where the error occured
    - The list of methods is known as the call stack or stack trace
    - When an appropriate method has been found with an exception handler the runtime system passes the exception to it

Catching Exceptions

- Try-catch blocks inside methods, that could throw an exception
- The try block contains the code which may throw an exception
- The catch block handles the thrown exception
- One try block can have multiple catch blocks
- See example

Declaring Exceptions

- Call exception-throwing methods, see example

Exception Types

- Two kinds: Checked and unchecked
- Checked exceptions:
    - in the java.lang.Exception class
    - Checked at compile-time
    - If a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.
- Unchecked exceptions:
    - in the java.lang.RunTimeException class
    - Not checked at compile-time
    - Most of the times these exceptions occur due to the bad data provided by the user during the user-program interaction.
- Errors are in the java.lang.Error class
- Rule of thumb: You should never throw or catch errors, which are reserved for indicating trouble in JVM.

The finally block

- The last catch block associated with a try block may be followed by a finally block
- You can use a try block without associated catch, if it has a finally block
- Often used for locks in concurrent packages
- The finally block's code is guaranteed to execute in nearly all circumstances
    - whenever the execution leaves the try-construct
- See example

The try-with-resources Statement

- When resources like streams, connections, etc. are used they have to be closed using a finally block.
- try-with resources, also referred as automatic resource management was introduced in Java 7 as a new exception handling mechanism
- It automatically closes the resources used within the try-catch block
- To use this statement the required resource should be declared within the parenthesis
- See example
- To use a class with try-with-resources statement it should implement AutoCloseable interface
- More than one class can be declared in try-with-resources statement

Throwing Exceptions

- User-written methods can throw exceptions.
- Intance must be constructed of the chosen exception type and the throw keyword should be used
- Checked exception types thrown from a method must be represented in the method's "throws" list.
- See example

User-defined Exceptions

- Users can create their own exceptions in Java with the following rules:
    - All exceptions must be a child of java.lang.Throwable
- Checked exceptions must extend the Exception class
- Unchecked exceptions must extend the RunTimeException class
- See example

Bad practice with Exceptions

- **Catching Throwable and Error:** Throwable is the superclass of all errors and exceptions in Java. Error is the superclass of all errors which are not meant to be caught by applications
- **Throwable.printStackTrace(…) should never be called:** Difficult to analyze logs for debugging
- **Generic exceptions Error, RuntimeException, Throwable and Exception should never be thrown:** prevents classes from catching the intended exceptions. Thus, a caller cannot examine the exception to determine why it was thrown and consequently cannot attempt recovery.
- **Exception handlers should preserve the original exception:** If not, then exception is lost and debugging will be a pain.
- **System.out or System.err should not be used to log exceptions:** one might simply loose the important error messages. Use a logging framework.
- **Exception swallowing a'ka empty catch block**
- See examples

Benefits of Exception Handling

- Separating error-handling code from "regular" code
    - Provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program
- Propagating errors up the call stack:
    - Java's exception handling mechanism works in such a way that error reports are propagated up the call stack.
    - This is because whenever an exception occurs, Java's runtime environment checks the call stack backwards to identify methods that can catch the exception.
    - When a program includes several calls between methods, propagation of exceptions up the call stack ensures that exceptions are caught by the right methods.
- Meaningful error reporting
- Identifying error types:
    - Java provides several super classes and sub classes that group exceptions based on their type
    - The super classes provide functionality to handle exceptions of a general type
    - Sub classes provide functionality to handle specific exception types.
    - A method can catch and handle a specific exception type by using a sub class object.
    - In case a method needs to handle multiple exceptions that are of the same group, you can specify an object of a super class in the method's catch statement.