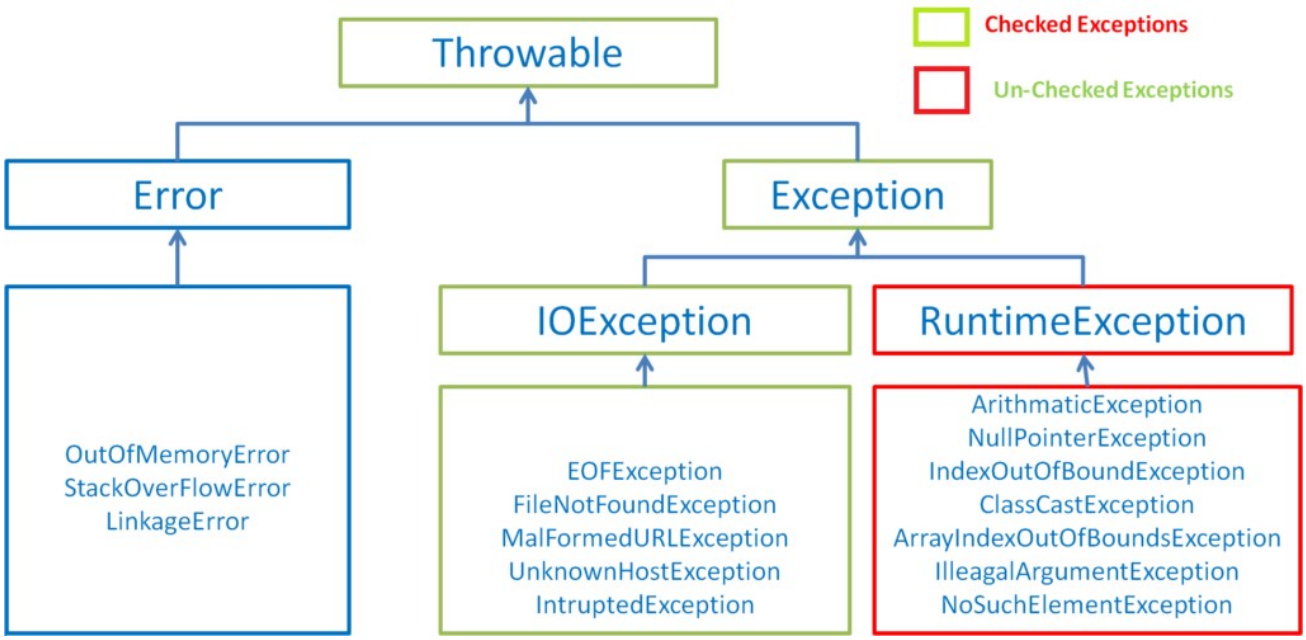
CSCI165 Computer Science II
Lab Assignment

Objectives:

- Exception handling with custom exception classes
- Develop an understanding of checked and unchecked exceptions in Java
- Understanding the “catch or throw” paradigm
- Implementing try, catch, throw and throws

The main goal of this lab is to implement custom exception handling. With that in mind, you will be refactoring some classes from a previous lab to implement exception handling. Ultimately we want to ***separate the code that creates errors from the code that responds to errors***. We do this to allow programs flexibility in how they wish to respond to an exceptional circumstance. Think of exception handling as a ***message passing*** model that allows the ability to communicate information about the state of the application while it is running. Our classes should simply signal that exceptional circumstances have occurred and allow the ***calling environment*** to specify how to respond, unless there is a single clear and obvious fix for a problem.

You will be defining some custom exception classes and experimenting with this model.



We do not worry about things on the **Error** branch of this hierarchy as these are generally out of the programmer’s control and would not be something we would have our application respond to. Developers focus on the **Exception** branch and decide whether we want our exceptions to be **checked or unchecked.** In this context “checked” means the compiler will force any checked exceptions to be dealt with, or the code will not compile.

An **unchecked** Exception is something that is generally the result of bad programming and should be fixed with properly formulated logic, instead of exception handling.

TASK ONE – Create custom Exception classes: You will need access to the following classes from the last Account lab in order to complete this task.

- Customer
- Account
- SavingsAccount
- CheckingAccount

Some of the policies and exceptions that are mentioned in the following instructions ***will need extra data*** in order for the classes to properly track and enforce. I have explicitly mentioned what you need to add in some cases, but not all. Make common sense decisions and add extra variables and methods if you find them necessary.

!!Include comments describing your design choices!!

In separate files each named appropriately, define the following Custom Exceptions. Include overloaded constructors that allow for data to be passed when throwing. At minimum there should be an overloaded constructor that accepts a String to handle contextual messages. Add as many as you feel you need and include comments. The goal here is to be able to communicate as much information on the program state as possible.

These exceptions should all be ***checked exceptions*** (be sure to develop an understanding of the differences between a checked and unchecked exception).

- **IDNotWellFormedException:** Thrown from the ***Customer*** class when the following policy is violated:
    - The Customer ID must start with a letter, followed by 3 digits
    - Include descriptive message, including the offending ID.
    - You decide the appropriate methods/constructors to use here.

- **InvalidAccountNumberException**: Thrown from the ***Account*** class when the following policy is violated:
    - Account number must be 5 digits. Java's strict type checking will handle enforcement of numeric types, but a valid account number must be 5 digits.

- **InvalidBalanceException:** Thrown from Account subclasses when the following policies are violated
    - **CheckingAccount:** Balance *would exceed* maximum allowed balance during an attempted deposit (add a ***public static final*** field to the CheckingAccount class to represent the maximum balance for all checking accounts). When throwing the exception include a descriptive message, including the amount that would exceed the balance.

      Include the ***throws clause*** at the super class ***deposit( )*** level, which means that it will

also have to be at the subclass level.

- **Note:** An overridden method cannot throw a broader exception class than its ancestor

- **SavingsAccount:** Balance *would fall below* minimum balance during a withdrawal (add a ***static final*** field to the SavingsAccount class to represent the minimum balance). Prevent the withdrawal from happening. When throwing the exception include descriptive message, including the amount that would be below the balance.

  Include the throws clause at the super class ***withdraw( )*** level, which means that it will also have to be at the subclass level.

- **OverdraftException:** Thrown when the following policies are violated

  - **CheckingAccount:** withdrawal would exceed over draft limits

  - **SavingsAccount:**
    - number of withdrawals exceeds maximum number of withdrawals. (add a ***public static final*** field to the SavingsAccount class to represent the maximum number of withdrawals allowed). Add any other variables you deem necessary to implement this policy.
    - withdrawal dollar amount > withdrawal limit dollar amount

  - Include the ***throws*** clause at the Account level's ***withdraw()*** method. This abstraction dictates that it must also be at the subclass level.

  - **Note:** A method can throw more than one exception

TASK TWO: Implement the policies described above by adding checks in the appropriate methods where these issues could occur. **No redundancies!** Instead of explicitly handling the issues at the method level source of the problem, throw the correct type of exception.

TASK THREE: Create a Driver app that demonstrates that your classes can detect and throw the correct types of exceptions. The Driver app is where you will implement the **try/catch** logic to provide an explicit response to the issue. For your proof of concept here you just need to demonstrate that you have correctly modeled the **throw/throws** and **try/catch** paradigm, The responses can be as simple as displaying the contextual messages and any associated data.

Rubric on the next page

Requirement	Points
Correctness: Classes meet all requirements and functions as described in the instructions. Driver contains all proof of concept requirements	10
Exception Classes: Exceptions are of the “checked” variety and defined as described in the instructions	10
Class Updates: Classes are refactored to handle the new policies introduced in the lab and comments on design choices are included in the code	8
Javadoc: methods are documented using appropriate Javadoc style comments	2