CMSC 202

Exceptions II

Methods may fail for multiple reasons

```
public void withdraw(double amount){
  if(amount < 0)
    throw new Exception("Amount Negative");
  if(amount > balance)
    throw new IllegalArgumentException("Overdraft of" + amount);
  balance -= amount;
}
```

- Withdraw can fail in multiple ways. Each is its own exceptional event and should be handled differently.
  - Overdraft's usually incur a penalty against your account until you have move money around accordingly.

Multiple `catch` Blocks

- A `try` block can call a method that potentially throws any number of exception values, and they can be of differing types
  - In any one execution of a `try` block, at most one exception can be thrown (since a throw statement ends the execution of the `try` block)
  - However, different types of exception values can be thrown on different executions of the `try` block

Multiple **`catch`** Blocks

- Each **`catch`** block can only catch values of the exception class type given in the **`catch`** block heading
- Different types of exceptions can be caught by placing more than one **`catch`** block after a **`try`** block
  - Any number of **`catch`** blocks can be included, but they must be placed in the correct order

Multiple catch Blocks

```
public void main(String[] args){
   BankAccount account = new BankAccount(100.00);
   Scanner input = new Scanner(System.in);

   System.out.print("Enter deposit amount: ");
   int amt = input.nextInt();
   try{
            account.withdraw(amt);

   }
   catch (IllegalArgumentException e){
            // code that "handles" the overdraft exception
   }
   catch (Exception e){
            // code that "handles" the negative exception
   }
}
```

Catch the More Specific Exception First

- When catching multiple exceptions, the order of the `catch` blocks is important
  - When an exception is thrown in a `try` block, the `catch` blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed

Catch the More Specific Exception First

```
public void main(String[] args){
   BankAccount account = new BankAccount(100.00);
   Scanner input = new Scanner(System.in);

   System.out.print("Enter withdraw amount: ");
   int amt = input.nextInt();
   try{
               account.withdraw(amt);

   }
   catch (Exception e){ // problems
      // code that "handles" the negative exception
   }
   catch (IllegalArgumentException e){
      // code that "handles" the overdraft exception
   }
}
```

Catch the More Specific Exception First

- Because a **`DepositNegativeException`** and **`DepositTooSmallException`** are types of **`Exception`**, all exceptions will be caught by the first **`catch`** block before ever reaching the second or third block
  - The catch blocks for **`DepositNegativeException`** and **`DepositTooSmallException`** will never be used!
- For the correct ordering, simply put the catch block for Exception last.

Defining Exception Classes

- A `throw` statement can throw an exception object of any exception class
- Instead of using a predefined class, exception classes can be programmer-defined
  - These can be tailored to carry the precise kinds of information needed in the `catch` block
  - A different type of exception can be defined to identify each different exceptional situation

Defining Exception Classes

- Every exception class to be defined must be a derived class of some already defined exception class
  - It can be a derived class of any exception class in the standard Java libraries, or of any programmer defined exception class
- Constructors are the most important members to define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only

Exception Object Characteristics

- The two most important things about an exception object are its type (exception class) and the message it carries
  - The message is sent along with the exception object as an instance variable
  - This message can be recovered with the accessor method **`getMessage`**, so that the catch block can use the message

Programmer-Defined Exception Class Guidelines

- Exception classes may be programmer-defined, but every such class must be a derived class of an already existing exception class
- The class **`Exception`** can be used as the base class, unless another exception class would be more suitable
- At least two constructors should be defined, sometimes more
- The exception class should allow for the fact that the method **`getMessage`** is inherited

Preserve `getMessage`

- For all predefined exception classes, `getMessage` returns the string that is passed to its constructor as an argument
  - Or it will return a default string if no argument is used with the constructor
- This behavior must be preserved in all programmer-defined exception class
  - A constructor must be included having a string parameter whose body begins with a call to `super`
  - The call to `super` must use the parameter as its argument
  - A no-argument constructor must also be included whose body begins with a call to `super`
  - This call to `super` must use a default string as its argument

A Programmer-Defined Exception Class

Display 9.3 A Programmer-Defined Exception Class

```java
public class DivisionByZeroException extends Exception
{
    public DivisionByZeroException()
    {
        super("Division by Zero!");
    }

    public DivisionByZeroException(String message)
    {
        super(message);
    }
}
```

You can do more in an exception constructor, but this form is common.

super is an invocation of the constructor for the base class Exception.

Tip: An Exception Class Can Carry a Message of Any Type: double Message

- An exception class constructor can be defined that takes an argument of another type
  - It would stores its value in an instance variable
  - It would need to define accessor methods for this instance variable
- A programmer defined exception class may include any information that might be helpful to the recipient

An Exception Class with an **`double`** Message

```
public class InvalidAmountException extends Exception {

    private double amount;

    public InvalidAmountException(double amount){
        super("Invalid Amount Exception: $" + amount);

        this.amount = amount;
    }

    public double getAmount(){
        return amount;
    }
}
```

Declaring Exceptions in a `throws` Clause

- If a method can throw an exception but does not catch it, it must provide a warning
  - This warning is called a ***`throws`*** *clause*
  - The process of including an exception class in a throws clause is called *declaring the exception*

```
throws AnException   //throws clause
```

```
public int deposit( int amt ) throws DepositNegativeException,
                DepositTooSmallException
{
   if (amt < 0 )
         throw new DepositNegativeException( );
   if (amt < minDeposit)
         throw new DepositTooSmallException( );
   balance += deposit;
}
```

Checked and Unchecked Exceptions

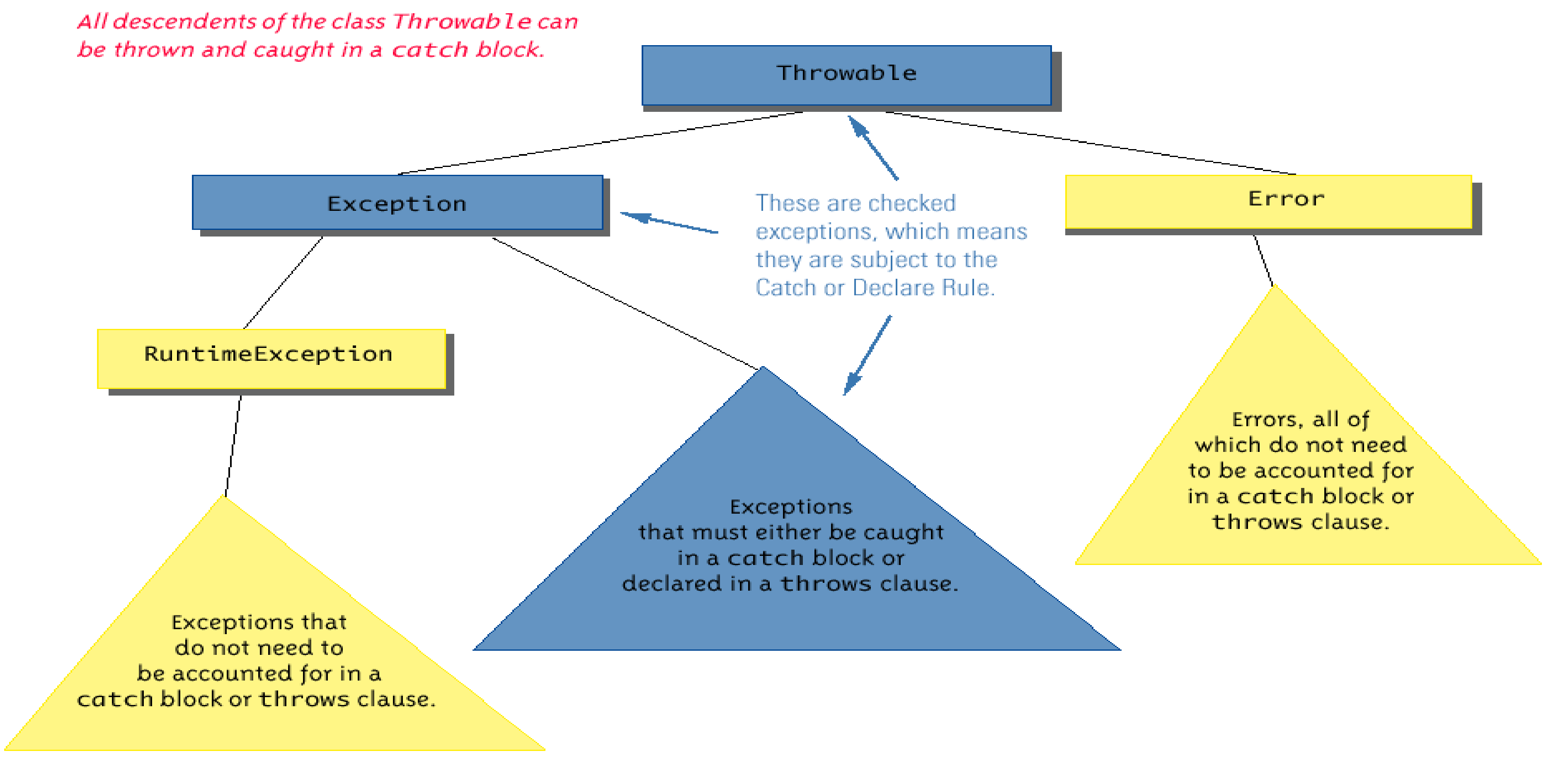
- Exceptions that are subject to the catch or declare rule are called ***checked*** exceptions
  - The compiler checks to see if they are accounted for with either a catch block or a throws clause
  - The classes **`Throwable`**, **`Exception`**, and all descendants of the class **`Exception`** are checked exceptions
- All other exceptions are ***unchecked*** exceptions
- The class **`Error`** and all its descendant classes are called *error classes*
  - Error classes are *not* subject to the Catch or Declare Rule

Exceptions to the Catch or Declare Rule

- Checked exceptions must follow the Catch or Declare Rule
  - Programs in which these exceptions can be thrown will not compile until they are handled properly
- Unchecked exceptions are exempt from the Catch or Declare Rule
  - Programs in which these exceptions are thrown simply need to be corrected, as they result from some sort of error

Hierarchy of Throwable Objects

Display 9.10 Hierarchy of Throwable Objects


All descendents of the class Throwable can be thrown and caught in a catch block.
Throwable
Exception
Error
These are checked exceptions, which means they are subject to the Catch or Declare Rule.
RuntimeException
Exceptions that must either be caught in a catch block or declared in a throws clause.
Errors, all of which do not need to be accounted for in a catch block or throws clause.
Exceptions that do not need to be accounted for in a catch block or throws clause.

Runtime Exceptions

- Runtime exceptions are
  - Unchecked
  - Frequently thrown by Java automatically when there is a bug in your program
    - Referencing a null pointer
    - Array index out of bounds
  - May also be thrown and/or propagated by your program for run-time issues

Constructors and Exceptions

Up until now we've had no way to recover if a bad parameter was passed to a constructor. We usually just exited the program with System.exit(). A better way is to throw an exception

```
public BankAccount(double balance) throws InvalidAmountException{
    if(balance < 0)
        throw new InvalidAmountException(balance);
    this.balance = balance;
}
```

- This code will not compile until you satisfy the Catch or Declare Rule.
- We can throw exceptions in constructors when class invariants are not maintained during construction.

trying constructors

```
public static void main(String[] args){

   BankAccount account;
   try{
      Scanner input = new Scanner(System.in);
      // get amount to deposit
      System.out.println("Enter a Starting Balance:  ");
      double amt = input.nextDouble();
      account = new BankAccount(amt);
   }
   catch(InvalidAmountException e){
      System.err.println(e.getMessage());
   }
   catch(Exception e){
      System.err.println(e.getMessage());
   }
}
```

- Here we try to construct a BankAccount Object. Anytime we create an account with a amt < 0 and InvalidAmountException is thrown and delt with appropriately.

The `finally` Block

- The `finally` block contains code to be executed whether or not an exception is thrown in a `try` block
  - If it is used, a `finally` block is placed after a `try` block and its following `catch` blocks

```
try
{  . . .  }
catch( ExceptionClass1 e )
{  . . .  }
  . . .
catch( ExceptionClassN e )
{  . . .  }
finally
{
  CodeToBeExecutedInAllCases
}
```

The `finally` Block

- If the `try-catch-finally` blocks are inside a method definition, there are three possibilities when the code is run:
  1. The `try` block runs to the end, no exception is thrown, and the finally block is executed
  2. An exception is thrown in the `try` block, caught in one of the `catch` blocks, and the `finally` block is executed
  3. An exception is thrown in the `try` block, there is no matching `catch` block in the method, the `finally` block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

When to use a finally block

- The finally block should contain code that you always want to run whether or not an exception occurred.
- Generally the finally block contains code to release resources other than memory
  - Close files
  - Close internet connection
  - Clear the screen