Lab 01: Basic Java Syntax

The objective of this lab is to test your understanding of the Java syntax, declarations, data types, assignments, and flow control structures.
Create a command-line calculator with the following usage:

```
Calculator <operator> <operand> <operand> ...
```

where <operator> is one of +, -, *, or / and <operand> is a number.
For example:

```
$ java Calculator + 1 1 2 $ java Calculator + 1 2 3 6 $ java Calculator
- 8 5 1 2 $ java Calculator "*" 8 5 2 80 $ java Calculator / 100 5 2 10
```

To do this lab:

1. Create a new Java project (in Eclipse)
2. Create a new Java class (call it Calculator)
3. Create a main(...) method in this class
4. Parse the operator and each operand into usable types
5. Perform the operation specified by the operator for each operand
   Build this program iteratively, starting with the addition (+) operation
6. Print the result
7. Handle errors: invalid usage, divide by zero, etc.
8. Test your code by running it from the command line (hint, from your project's bin directory)
   When running the calculator from the command line, be sure to double-quote the * operator (i.e. "*") as the operating system interprets an unquoted * as a listing of files in the current directory.

The game:

- Get the + operation to work
- Get the - operation to work
- Get the * operation to work
- Get the / operation to work
- Handle at least two invalid usage error conditions (e.g. invalid operator and not enough operands)
- Handle divide by zero error condition
- Handle condition where operands are not valid numbers (hint use operandAsString.matches(String regex))
- Write your code efficiently (no code repetition)

- Bonus: Change the command line to allow multiple operators (e.g. 1 * 2 + 3)

Lab 02 : Basic OOP in Java

The objective of this lab is to test your understanding of OOP in Java, how classes differ from objects and how static data differs from instance data.

1. Model a checking account by defining a CheckingAccount class in a new project:
    1. Each CheckingAccount object should contain:
        - A number field, represented by an int
        - A balance field, represented by a double
    2. The number field should be auto-assigned on creation to a unique value for each account
       Auto-increment it for each new account
    3. Add a mechanism for checking accounts to print themselves to the STDOUT
2. Write a separate application class (e.g. BankDemo) that creates two (or more) CheckingAccount objects.
    1. Set the balance field on each account to values of your choosing
    2. Add and subtract balance from each object
    3. For each object, print the account number and the final balance

The game:

- Model the basic CheckingAccount state
- Auto-assign the number field to a unique value
- Make it possible to print checking accounts
- Create two or more distinct instances of the checking account type

Lab 03: Intermediate OOP

The objective of this lab is for you to test your understanding of encapsulation of objects as well as practice object composition. This lab expands on lab 02.

1. Revise your CheckingAccount class.
    1. Implement getters for the number and balance fields
    2. Implement two constructors:
        1. One should accept an initial value for the balance field
        2. The other should be a no-argument constructor that initializes the balance to 0.0.
    3. Implement a credit(double) method that increases the account balance by the specified amount.
    4. Implement a debit(double) method that decreases the account balance by the specified amount.
    5. Implement a transferTo(CheckingAccount, double) method that debits the specified amount from the current object and credits the same amount to the CheckingAccountobject provided as an argument.

        Consider the return values of your credit(), debit(), and transferTo() methods. Should they all have a void return type? Can you think of useful values they could return?
    6. Assign appropriate access modifiers to all fields, methods, and constructors so that they are properly *encapsulated*.
2. Expand the demo class (e.g. BankDemo) to create at least two CheckingAccount objects and exercises all of the methods you've implemented.
3. Each checking account should be associated with a customer.
    1. Define a Customer class:
        1. Each Customer should have firstName, lastName, and ssn fields, all stored as String values.
        2. Implement appropriate getter, setter, and constructor methods for your Customer class.
    2. Modify your CheckingAccount class:
        1. Add an unmodifiable field of type Customer
        2. Implement the appropriate accessor (i.e. getter method)
        3. Modify the constructors to require a Customer object
    3. Update your application demo class to test the features you added

1. Create one or more Customer objects
2. Create one or more CheckingAccount objects, making sure to assign a Customer object to each CheckingAccount
3. Test the new capabilities of your classes

The game:

- Added the required setters/getters to CheckingAccount
- Added the required constructors to CheckingAccount
- Implemented CheckingAccount.credit(double) method
- Implemented CheckingAccount.debit(double) method
- Implemented CheckingAccount.transferTo(double) method
- Used appropriate return values from credit(), debit(), and transferTo() methods on CheckingAccount
- Encapsulated CheckingAccount
- Tested that encapsulated CheckingAccount works
- Created fully encapsulated Customer class with all the required fields, methods, and constructors
- Updated CheckingAccount to link to Customer
- Tested that CheckingAccount with Customer works

Lab 04: Inheritance

The objective of this lab is to test your understanding of inheritance in Java.

- The bank wants to manage deposit accounts in addition to a checking account
    - They want a SavingsAccount class that has the same features as a CheckingAccount, plus the ability to
        - Set the interest rate across **all** savings accounts - e.g. invoking setInterestRate(0.02) would set a new 2% annual interest rate
        - Credit interest for each account on monthly basis - e.g., bank would invoke creditInterest() twelve times per year
        - For simplicity, assume that the monthly interest rate is just the annual rate divided by 12.
    - They want to be able to manage mixed collections of checking and accounts - e.g. easily print balances for all accounts in a collection
    - They are thinking of adding new product types in the future, like CDs and money market accounts
    - For any given account in a collection, they want to be able to determine the account type
- How should you design the inheritance hierarchy to support this?
    - Should you use CheckingAccount as the base class for the hierarchy? Or should you create a new base class that CheckingAccount and the other account types inherit from?
    - How might you support identifying the type of account?
- To do this lab:
    1. Implement the new class hierarchy with the requested features and test it out with your application
    2. Set the shared interest rate for savings accounts
    3. Build an array of mixed account types (minimum of two each)
    4. Loop through the array 12 times and credit just the savings accounts
    5. Loop through the array and print all accounts
    6. Verify that the interest is calculated correctly

The game:

- Model accounts bank wants to have correctly

- Provide ability for the bank to set the shared interest rate on savings accounts
- Provide ability for the bank to credit interest to savings accounts
- Create a demo program to test out your new accounts
- Write your code efficiently (no code repetition)

Lab 05: Overriding java.lang.Object Methods

The purpose of this lab is to test your understanding of java.lang.Object and its role in Java.

1. Override the toString() method for the Customer and your bank account classes and return a reasonable representation of object information for each class
2. Get rid of the print() method on the bank account classes
3. Override equals(Object) and hashCode() methods for both customers and bank accounts
    - Consider which properties are significant and only use those in your implementations of the two methods

- Update your demo program to test customers and accounts for equality and print them to STDOUT using their toString() methods

The game:

- Get Customer.toString() to return useful string representation for customers
- Get BankAccount.toString() to return useful string representation for bank accounts
- Get CheckingAccount.toString() to return useful string representation for checking accounts
- Get SavingsAccount.toString() to return useful string representation for savings accounts
- Implement Customer.equals(Object) correctly (as per the contract)
- Implement Customer.hashCode() correctly (as per the contract)
- Implement BankAccount.equals(Object) correctly (as per the contract)
- Implement BankAccount.hashCode() correctly (as per the contract)
- Prove that your new methods work correctly in a demo program

Lab 06 : Packaging

The purpose of this lab is to test your understanding of packaging in Java.

1. Separate all of your bank-related code into distinct logical packages, including sub-packages if necessary
    - For example, the demo class could be in a different package from the bank classes
    - Use a unique package prefix for your organization
2. Import classes from other packages as needed
3. Hide classes/methods/fields that should not be visible from the outside of your packages
4. Create a JAR file that packages all of your code and run your demo class by adding the JAR file to your CLASSPATH
5. Create an *executable* jar file a that packages all of your code and run it

The game:

- Refactor your bank-related code into at least two packages and use import statement(s) as needed
- Protect the name-space of your packages
- Use protected keyword at least once (appropriately)
- Create a JAR file and run the demo code from command-line by adding the JAR file to your CLASSPATH
- Create an executable JAR file and run the demo code from command-line

Lab 07: JavaDoc

The purpose of this lab is to test your understanding of JavaDoc.

1. Document your bank classes using javadoc comments
    1. Document all classes (consider using @author and @version tags)
    2. Document all public and protected fields, constructors, and methods (use @param and @return tags)
2. Generate the JavaDoc HTML API
    1. Only include public and protected classes, fields, constructors, and methods
    2. Link to Java's library JavaDoc
3. Verify that it matches your source files

The game:

- Document Customer class completely
- Document BankAccount class completely
- Document CheckingAccount class completely
- Document SavingsAccount class completely
- Generate JavaDoc HTML API

Lab 08: Exceptions

The objective of this lab is to test your understanding of Java exceptions.

Currently, our bank accounts return false on debit(double), credit(double), and transferTo(BankAccount, double) operations but these return values can be ignored by the users of our APIs.

1. Update the bank account class(es) to throw exceptions on invalid usage, rather than return false value
    - Throw IllegalArgumentException on negative amount
    - Throw NullPointerException when the bank account we are transferring funds to is null (should we leave this up to the JVM to do automatically?)
    - Update bank account constructors to require correct usage
    - Create a new NonSufficientFundsException exception and throw it when the requested debit (or transfer) operation exceeds the balance
        - Define NonSufficientFundsException as a subclass of Exception
        - Provide the state that will hold the balance that was available at the time this exception was created as well as the funds that were requested
        - Implement a constructor to initialize the state of this exception
        - Provide getters for the available balance and the requested funds
2. Update Customer to require correct usage
3. Notice that you now have compilation errors in your banking test application because of the checked exceptions thrown by debit(double) and transferTo(BankAccount, double).
    - Modify your banking application to use try-catch structures around your debit() and/or transferTo(BankAccount, double) calls.
    - If a NonSufficientFundsException condition occurs, report on the console that the debit was not allowed.
    - Test for negative-amount conditions
    - Test transferTo(BankAccount other, double amount) with other==null

The game:

- In bank account class(es), update credit(double), debit(double), and transferTo(BankAccount, double) to throw an IllegalArgumentException on negative balance.

- In bank account class(es), update transferTo(BankAccount, double) to throw a NullPointerException when the bank account is null.
- In bank account class(es), update constructors to enforce correct initialization.
- Update Customer class to enforce correct initialization and usage.
- Create NonSufficientFundsException class and throw it from debit(double) and transferTo(BankAccount, double) in the bank account class(s) when the requested amount exceeds the available balance.
- Update the demo program to test your exception conditions.

Lab 09: Java shell

The objective of this lab is to test your understanding of Java files and I/O.

Write a command-line shell in Java. This shell will implement some the of basic commands found in any Linux shell.

1. Start the shell by running the Java program and supplying a directory.
   - e.g. java lab09.Shell *rootDirectory*
2. The current directory serves as the prompt. e.g. */homeDirectory>*
3. Consider your program structure:
   a. Use a an interface to define the template for all commands.
   b. Have a REPL-based program that reads the command, evaluates it, processes it and loops until "quit" is entered.
   c. Command requires varying amount of parameters.
   d. Unknown commands should not break the program
4. Implement as many commands as time permits:
   - cd *directory*
   - pwd
   - ls
   - cp *source destination*
   - rm *file*
   - cat *file*

Lab 10: Remote Java shell

The objective of this lab is to test your understanding of Java net.

Building on the lab 09, make your java shell accessible remotely.

- Write a server that will accept incoming requests via a server socket.
- Attach those new requests to the local shell.
- Send all command to the local shell.
- Return the response back to the client.
- Server will start with port number to use as a daemon port.
- Use telnet to connect to the server.

As a bonus, write a remote shell client that acts as a remote client in lieu of telnet.

Lab 11: Collections

The objective of this lab is to test your understanding of rudimentary Collections

Part 1:
Define this array:

-
  ```
  private static final String[][] NUMBER_ARRAY =
  {
                              {"2", "Two"},
                              {"7", "Seven"},
                              {"1", "One"},
                              {"4", "Four"},
                              {"8", "Eight"},
                              {"6", "Six"},
                              {"0", "Zero"},
                              {"9", "Nine"},
                              {"3", "Three"},
                              {"5", "Five"}
  };
  ```

- Store the number string (e.g. :"one") inside a list using the corresponding number (e.g. 1) as the index.
- Store each as a pair inside a generic Pair object.
- Print the value of the array

Part 2:

- Using the same array above, store the array as a map
- Use the number string as a key to the map and the number itself as a value
- Lookup each number string and print its value.

Lab 12: Advanced Collections

The objective of this lab is to test your understanding of advanced uses of Collections.

Implement a movie database using this interface:

```
public interface MovieDatabase {
	/**
	 * Adds a movie to the database with the given categories and year released.
	 *
	 * @param name The movie name
	 * @param categories The type of movie
	 * @param yearReleased The year released.
	 */
	void add(String name, Set<String> categories, Integer yearReleased);

	/**
	 * Overloaded method to add a movie with only one category.
	 *
	 * @param name The movie name
	 * @param category The type of movie
	 * @param yearReleased The year released.
	 */
	void add(String name, String category, Integer yearReleased);

	/**
	 * Find the movie by its name.
	 *
	 * @param name The name of the movie to find.
	 * @return The found movie.
	 */
	Movie find(String name);

	/**
	 * Returns the list of movies with that category.
	 *
	 * @param category The category to find.
	 * @return The list of movies corresponding to the category.
	 */
	List<String> getByCategory(String category);

	/**
	 * Deletes the movie with the given name.
	 *
	 * @param name The name of the movie to delete.
	 * @return True if the movie was found - false otherwise.
	 */
	boolean delete(String name);
}
```

The movie database will be stored in the database.
Define a holder class (Movie.java) that will contain the name, date released and categories.

Lab 13: Threading with Runnable

The objective of this lab is to test your understanding of threading using Runnable.

Write a program that will find the number of prime numbers in a given range in parallel:

- Start X amount of threads using runnable
- Each thread will loop through a range and return the number of prime numbers found in the range
- Fire off all threads at the same time
- Once all threads have completed, query the runnable to return the number of prime numbers found.

To save time, use this method to determine whether or not a number is prime:

```java
private boolean isPrime(int primeCandidate) {
   boolean isPrimeFound = true;

   if (primeCandidate > 2) {
      for (int testValue = 2;
         testValue <= Math.sqrt(primeCandidate); ++testValue) {
         if (primeCandidate % testValue == 0) {
            isPrimeFound = false;
            break;
         }
       }
    }
    else if (primeCandidate != 2) {
       isPrimeFound = false;
    }

    return isPrimeFound;
}
```

Lab 14: Threading with Callable

The objective of this lab is to test your understanding of threading using Callable.

Reimplement the same program using Callable and Futures.

Lab 15: Enums

The objective of this lab is to test your understanding of Java enums.

This RGB color chart the level of brightness in the RGB model for red, green and blue respectively. It is expressed as a percentage of brightness (from 0 to 255):

```
WHITE  (1.00f, 1.00f, 1.00f),
SILVER (0.75f, 0.75f, 0.75f),
GRAY   (0.50f, 0.50f, 0.50f),
BLACK  (0.00f, 0.00f, 0.00f),
RED    (1.00f, 0.00f, 0.00f),
MAROON (0.50f, 0.00f, 0.00f),
YELLOW (1.00f, 1.00f, 0.00f),
OLIVE  (0.50f, 0.50f, 0.00f),
LIME   (0.00f, 1.00f, 0.00f),
GREEN  (0.00f, 0.50f, 0.00f),
AQUA   (0.00f, 1.00f, 1.00f),
TEAL   (0.00f, 0.50f, 0.50f),
BLUE   (0.00f, 0.00f, 1.00f),
NAVY   (0.00f, 0.00f, 0.50f),
FUSHIA (1.00f, 0.00f, 1.00f),
PURPLE (0.50f, 0.00f, 0.50f);
```

Write a program that adds the brightness of any two colors and returns the set of other colors with matching brightness:

- Adding the three RGB colors yields the brightness of the color.
- Use enums to express each color.
- Implement a method in the enum that returns the set of colors matching in brightness.
- Test every 2-color combination and print the result (hint: don't do this manually!).

Lab 16: Annotations

The objective of this lab is to test your understanding of annotation in both their usage and creation.

Define a new method-level annotation (Transaction) that will be used in a framework that controls database transactions. This annotation should have two fields:

- Transaction enabled: true of false (default is false)
- Commit type: XA or one-phased (default is one-phase)

Define three classes with one do-nothing method declares this annotation:

- EmployeeDataAccessor: requires a transaction with XA commit
- OrderDataAccessor: requires no transaction
- ManagerDataAccessor: requires a transction with one-phase commit

Test your annotation by instantiating these classes and printing their annotation configuration.

Lab 17: JDBC

The objective of this lab is to test your understanding of JDBC.

Implement a database-backed implementation of the MovieDatabase interface from lab 12. Use this modified interface adapted for the database:

```
public interface MovieDatabase {

        /**
         * Adds a movie to the database with the given categories and year released.
         *
         * @param name The movie name
         * @param categories The type of movie
         * @param dateReleased The year released.
         * @return The newly added row represented as an object.
         * @throws StorageException In case of error interacting with database.
         */
        Movie add(String name, Set<String> categories, Date dateReleased)
               throws StorageException;

        /**
         * Overloaded method to add a movie with only one category.
         *
         * @param name The movie name
         * @param category The type of movie
         * @param yearReleased The year released.
         * @return The newly added row represented as an object.
         * @throws StorageException In case of error interacting with database.
         */
        public Movie add(String name, String category, Date dateReleased)
               throws StorageException;

        /**
         * Find the movie by its name.
         *
         * @param name The name of the movie to find.
         * @return The found movie or null if not found.
         * @throws StorageException In case of error interacting with database.
         */
        Movie find(String name) throws StorageException;

        /**
         * Returns the list of movies with that category.
         *
         * @param category The category to find.
         * @return The set of movies corresponding to the category or an empty set \
         * if none found.
         * @throws StorageException In case of error interacting with database.
         */
        Set<String> getByCategory(String category) throws StorageException;

         /**
          * Updates the movie name or date released as per the given moie object.
          *
          * @param movie The updates to make (only considers movie name or date
          * released).
          * @throws StorageException In case of error interacting with database.
          */
```

```
 void update(Movie movie) throws StorageException;

/**
 * Deletes the movie with the given name.
 *
 * @param name The name of the movie to delete.
 * @throws StorageException In case of error with storage device.
 */
void delete(String name) throws StorageException;

/**
 * Clears entire movie database
 * @throws StorageException In case of error interacting with database.
 */
void clear() throws StorageException;
```

Create the database schema from lab17.db using these command:

- mysql -u root -p
- CREATE USER ***'your_username'***@'localhost' IDENTIFIED BY ***'your_password'***;
- CREATE DATABASE IF NOT EXISTS movie;
- GRANT ALL ON movie.* TO ***your_username***@localhost IDENTIFIED BY ***'your_password'***;
- exit
- mysql -u ***your_username*** -p movie < ***your_root***/src/lab17/db/init-schema.sql
- To view database
  - mysql -u ***your_username*** -p
  - use movie;
  - show tables;
  - select * from category;
  - select * from movie;

Obtain your connection object using this:

- DriverManager.getConnection("jdbc:mysql://localhost:3306/movie?useSSL=false", "***your_username***", "***your_password***");