Java Fast Track Labs

Introduction

The following labs can be downloaded from:

https://github.com/ThoughtFlow/FastTrackToJava

Each lab is modeled as a Java package with *init* and *fin* as sub-packages. The *init* package contains the initial code from which to start and the *fin* package contains the final solution. You may simply modify the code directly in *init* or copy the code in another package then modify it. Resist the urge to look at the answers in the *fin* package!

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 (Intellij)
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.

Game plan:

- 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

Game plan:

- 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

Game plan:

- 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

Game plan:

- 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

Game plan:

- 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: 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

Game plan:

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

Lab 07: 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

Game plan:

- 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 08: 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);

Starting with the lab08.init, implement getEqualBrightness() and getTotalBrightnessInRange() in RgbColor:

- getEqualBrightness: returns the set of RgbColors that have an equal amount of brightness as the given color.
- getTotalBrightnessInRange: returns the total red/green/blue brightness in the range
- Test your program with TestRgbColor (provided).

Lab 09: 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)

Use the three provided classes and add the Transaction annotation like this:

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

Write an annotation analyzer that will print the annotation details of each class. Use AnnotationAnalyzer as a starting point.

Lab 10: 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"}
      };
  ```
- Create a generic class Pair that can store any two objects (T, U). Make it immutable and implement the getter methods as well as the equals, hashCode and toString (hint: use Intellij's code generation).
- Store the first column of NUMBER_ARRAY as T and the second as U.
- Store the array in a collection type that supports order and permits duplicates, then list the elements.

Part 2:

- Store the NUMBER_ARRAY in a map where the key is the first column of the array and value is the pair object composed of the first and second column.
- Iterator over the map using the key set and print the elements.

Lab 11: 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 MovieDb {
        /**
         * Adds a movie to the database with the given categories, name and year
         * released.
         *
         * @param categories The set of categories for the new movie.
         * @param name The name of the movie.
         * @param yearReleased The year of release
         */
        void add(Set<Category> categories, String name, Integer yearReleased);

        /**
         * Adds a movie to the database with the given category, name and year
         * released.
         *
         * @param category The category for the new movie
         * @param name The name of the movie.
         * @param yearReleased The year of release
         */
        void add(Category category, String name, Integer yearReleased);

        /**
         * Searches for the given movie title and returns as a Movie record.
         *
         * @param name The name of the movie to search.
         * @return The found movie or null if not found.
         */
        Movie findByName(String name);

        /**
         * Searches by category and returns the list of movies for the given category.
         *
         * @param category The category name to search.
         * @return The list of movies matching the category or an empty list.
         */
        List<String> findByCategory(Category category);

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

Use a map whose key is the category and whose value is the list of movies for that category.

Lab 12: The lambda form

Objective: test your understanding of how to implement lambdas.

Use these four pre-created interfaces in lab12.init:

1. Interface1.java
   public void printSquareOfA(int a);
2. Interface2.java

   public int getSquareOfA(int a);
3. Interface3.java

   public int getAxB(int a, int b);
4. Interface4.java

   public void consume(int a);
5. Interface5.java

   public double getPi();
6. Interface6.java

   public Double create(double x);

Implement these lambdas:

1. Implement a lambda squares the parameter value and prints the result conforming to Interface1.
2. Implement a lambda that will return the square of the parameter value using interface 2.
3. Implement a lambda that will return the multiplication of the two parameter values using Interface3.
4. Implement a lambda that will return the value of pi as a double using Interface5.
5. Implement a lambda that returns the square of the parameter value itself conforming to Interface2. Provide a default method that pretty prints a message around getSquareOfA().
6. Implement a static method in Interface3 that multiplies the two numbers. Call the method and print the result.
7. Implement a static method reference for interface4 and use caculateAndConsume with Interface2 to square the value of the parameter and print the results.
8. Implement a lambda that uses the static method Double.valueOf() to implement interface6.

Lab 13: Standard functional interfaces

Objective: test your understanding of how to use the standard functional interfaces.

1. Replace functional interfaces 1 through 5 in lab13.init with their standard functional interface equivalent. Test it with LambdaTest.
2. Use Functions to create a series of functions that:
    a. Double, square, cube then negate a number using andThen
    b. Double, square, cube then negate a number using compose
    c. Use FunctionalComposition in lab13.init as a starting point.

Lab 14: Using functionalized collections

Objective: test your understanding of how the newly functionalized collections library in Java 8.

Using this interface:

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

        /**
         * Adds a movie to the database with the given category, name and year
         * released.
         *
         * @param category The category for the new movie
         * @param name The name of the movie.
         * @param yearReleased The year of release
         */
        void add(Category category, String name, Integer yearReleased);

        /**
         * Searches for the given movie title and returns as a Movie record.
         *
         * @param name The name of the movie to search.
         * @return The found movie or null if not found.
         */
        Movie findByName(String name);

        /**
         * Searches by category and returns the list of movies for the given category.
         *
         * @param category The category name to search.
         * @return The list of movies matching the category or an empty list.
         */
        List<String> findByCategory(Category category);

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

- Convert each method in FunctionalMovieDb using the functionalized collection methods of sets, lists and maps.
- Fill in the methods where indicated by “implement this”.
- Refer to ImperativeMovieDb for an implementation using imperative method.
- Test your implementation with TestMovieDb.

Lab 15: Using the Java networking libraries

Objective: test your understanding of the networking libraries in Java.

1. Write a chat server that can send messages over TCP/IP using NIO.
    - The server listens over a port waiting for new connections to be established.
    - When new connections are established, the server requests that the client enter a name to identify itself and a room in which to enter.
    - Messages are then exchanged with all other users in that room. Users in other rooms only see their own room messages.
    - A user exits the room by typing “bye”. Room will remain open for other users. Users can re-enter the room later.
    - Use telnet as a chat client.
2. Write a simple chat client that can use the chat server:
    - The client prompts the user to enter their name and room to enter.
    - The client can then send the data to the chat server and begins chatting with any other use in the room.
    - The chat client is single threaded meaning it will either wait for the client user to enter a message or for others to respond. It switches from one mode to the other when the client user has typed a message or when a network message has been received. Data gets buffered when it isn’t ready to be read or written.
    - Client ends when the user types “bye”.

Lab 16: Read/Write locks with conditions

Objective: test your understanding of Java's Read/Write locks

Refactor the movie database from lab06 to make it thread-safe:

- Use the class ThreadSafeMovieDb as a starting point.
- Use read/write locks to access the database class attribute in a thread-safe way.
- Look for markers in the code for help in locating areas to protect.
- Use the pre-made TestMovieDb to test your implementation.

Lab 17: Using the executor service with callables and futures

Objective: test your understanding of the executor service, callables and futures.

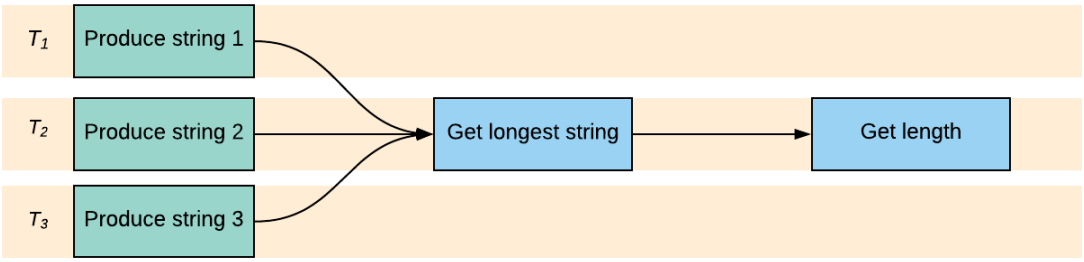
Write an application that counts the number of prime numbers in ranges using the ExecutorService:

- Start with the single-threaded implementation ThreadedPrimeNumberFinder. Re-implement the method countPrimes.
- Choose the appropriate ExecutorService implementation and mind the pool size.
- Use submit/invoke, call and future, then shutdown the pool.
- Each range is 1000 elements.
- Each range is calculated by different threads using the executor service.
- Print the number of primes found for all ranges.
- Use the method Util.isPrime (lab.util package) to determine if a number is prime.
- There are 78,498 prime numbers between 1 and 1,000,000.

Lab 18: Using promises to find prime numbers

Objective: test your understanding of promises.

1. Implement a chained set of promises that:
    - Call produceString1, produceString2 and produceString3 asynchronously. Each return a string.
    - Get the longest string of the three.
    - Return the length of that string as an integer.
    - Use PromiseCombiner as a starting point and implement weavePromise().
    - Should return: 10.



2. Re-implement the prime number finder from lab 8 using promises:
    - Start with PromisesPrimeNumberFinder and implement the method createPromise();
3. Add exception handling to the promise:
    - Add exception handling in the promise to handle exceptions. This handler should simply return 0 and continue with the next range.
    - Print an error message but continue anyway.
    - Test with a negative range.
    - Start with PromisesPrimeNumberFinder and implement the method createPromiseWithException();

Lab 19: Using streams

Objective: test your understanding and practice thinking in streams.

Use streams to implement these algorithms:

1. Iterate through numbers from 0 to 100:
   - Print out all the even numbers.
   - Then, modify your algorithm to add only odd numbers 0, 100.
   - Then, modify your algorithm to add only odd numbers 0, 100 but remove prime numbers.
   - Then, modify your algorithm to find the smallest int whose factorial is >= 1,000,000
2. Given a list of strings, print each string that is a palindrome:
   - Then, modify your algorithm to return the original word (unstripped).
3. Implement the Fizz Buzz algorithm:
   - Iterate from 1 to 100.
   - Print "Fizz" for every number divisible by 3 and "Buzz" for every number divisible by 5.

Lab 20: Packaging

Objective: test your understanding of Java packaging concepts.

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. (For the purposes of this lab, use lab20.init.com.mycorp)
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 from the jar file.
5. Create an *executable* jar file that packages all of your code and run it.

The plan:

- 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.