# Effective Java – Exercises

**NB!** All Exercises should be carried out from workspace Examples

# X1 – Object class methods

Project: **Time**

Package: **affix.java.effective.time**

Complete the code stubs for class **Time** according to JavaDoc specifications.

Override Object methods equals(), hashCode(), toString(), compareTo(Time that)

Run predefined tests found in separate files and as a suite stored in directory **test**.

Correct errors until all tests pass producing a green bar in JUnit output.

# X2 – Immutable class

Project: **Time**

Package: **affix.java.effective.time**

Refactor class **Time** in order to make it immutable, i.e. prohibit any methods that can change object values, make class non-inheritable and use keyword final for attributes.

Rerun predefined tests found in separate files and as a suite stored in directory **test**.

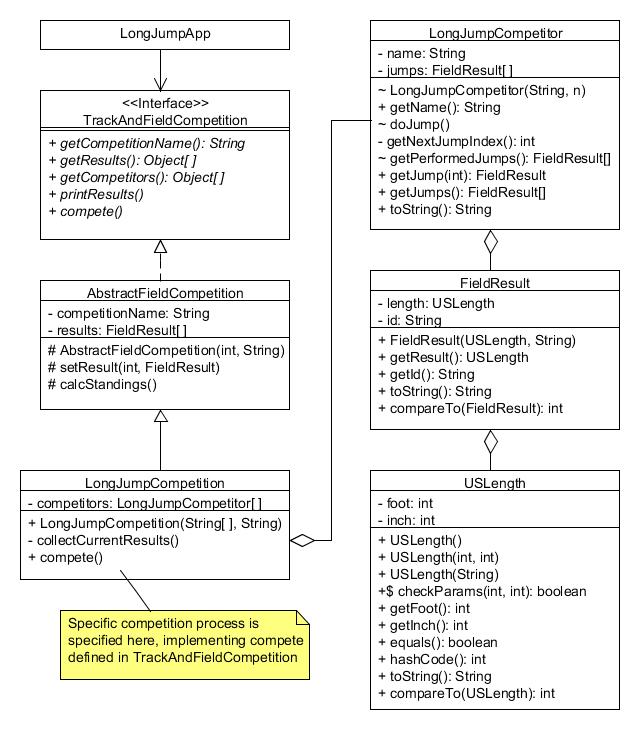
# X3a – Software Design: Classes and Interfaces

Project: **TrackAndField**

Packages: **affix.java.effective.field, affix.java.effective.trackandfield**

Study UML **LongJump** below + implementation code. Run **LongJumpApp**

TrackAndField - LongJump



# X3b – Software Design: Classes and Interfaces

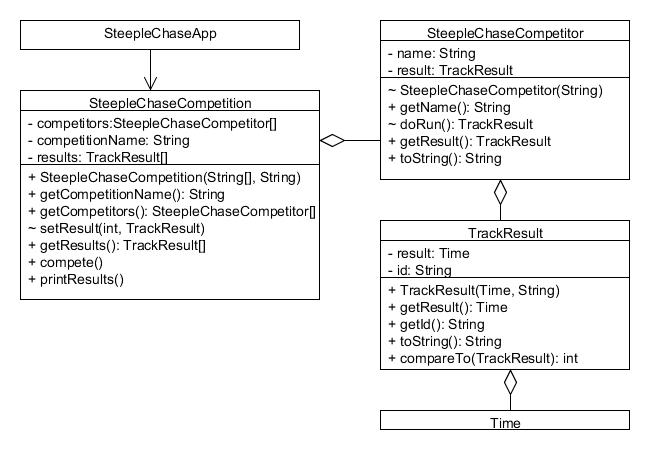
Project: **TrackAndField**

Packages: **affix.java.effective.track, affix.java.effective.trackandfield**

SteepleChase is a parallel application to LongJump

Study UML **SteepleChase** below. Run **SteepleChaseApp**.

SteepleChase

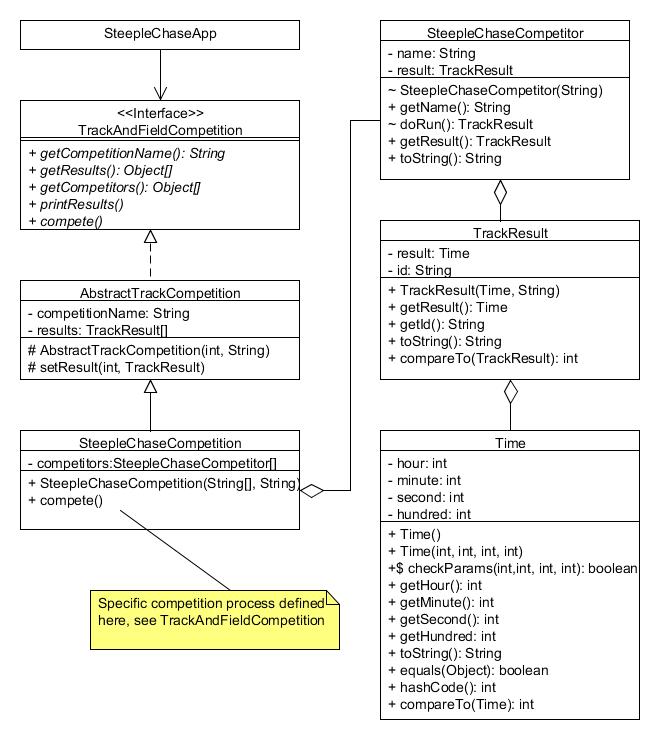


Refactor existing code in order to comply with the structure presented in UML below.

**SteepleChaseCompetition** should be split up into two separate classes. The abstract class **AbstractTrackCompetition,** which implements **TrackAndFieldCompetition,** should define all general attributes and methods for any competition of Track type.

Run **SteepleChaseApp** making sure that the application is behaving as expected.

TrackAndField - SteepleChase



$ static, # protected, ~ package access

**X3c – Software Design: Classes and Interfaces**

Project: **TrackAndField**

Packages: **affix.java.effective.trackandfield**

Add default method **getWinningResult()** in interface TrackAndFieldCompetition, which should return the first result from the array that holds the results from a competition. Refactor main in LongJumpApp and SteepleChaseApp calling this method instead of local calculation of winning result.

**X3x – Software Design: Classes and Interfaces**

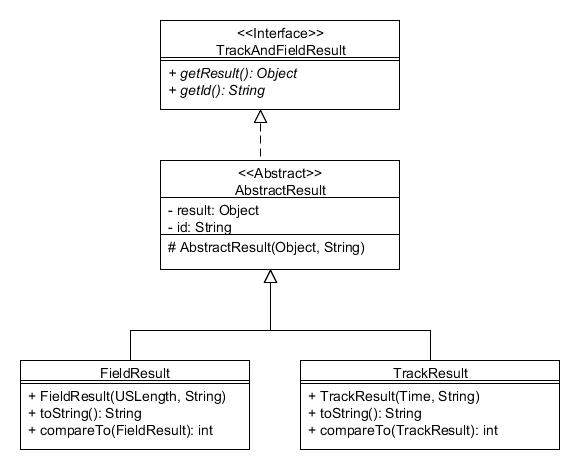
Project: **TrackAndField**

Packages: **affix.java.effective.field, affix.java.effective.track, affix.java.effective.trackandfield**

Study UML **TrackAndField - AbstractResult**

Refactor code in project TrackAndField introducing interface **TrackAndFieldResult,** partially implemented by a common abstract class **AbstractResult** that is inherited by concreate classes **FieldResult/TrackResult** used by application branches **LongJump**  and **SteepleChase** respectively**.**

TrackAndField - Abstract Result



**X4 – Formatting**

Project: **Time**

Package: **affix.java.effective.time**

Refactor method **toString()** in class Time providing a formatted String instead of using a StringBuilder.

Run JUnit tests, make sure that all still pass

**X5 – Date and Time**

Project: **DateAndTime**

Package: **affix.java8.dateandtime**

Add code to application **BirthdayWaitTest**, which should calculate the number of days from current day a person must wait until his/her next birthday.

***Tip:*** Use methods available in class LocalDate for handling dates effectively.

**X6a – Collections Framework**

Project: **LogInSystem**

Package: **affix.java.effective.collections**

A LogIn system is defined having one SysAdmin, implemented as a singleton, and a number of Users connected to the system by user name and password.

Study implementation code in parallel to running LogInGUIApp.

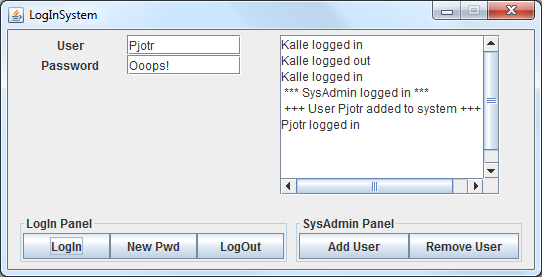
Run JUnit tests, make sure that all tests pass.

Refactor the application in order to use the Collections Framework effectively.

Class **User** should hold password history in a number of unique Strings that should cover the last 6 used passwords. A password cannot be null or an empty String.

NB! The current password should be stored “first” in this collection

Rerun JUnit tests, make sure that all tests pass.



**X6b – Collections Framework**

Project: **LogInSystem**

Package: **affix.java.effective.collections**

Class **LogInSystem** should hold any number of unique User objects in a key-value associated collection, where key should be user id as defined in User objects, and value should hold a complete User object.

Rerun JUnit tests, make sure that all tests pass.

**X7a – Enums**

Project: **LogInSystem**

Package: **affix.java.effective.collections**

Refactor class **User** in application LogInSystem.Exchange int constants defining various priorities for an Enum holding 3 levels of priority; LOW\_PRIORITY, MEDIUM\_PRIORITY and HIGH\_PRIORITY. Create a separate class **Priority** in a separate filewhere the enum is defined.

**X7x – Enums**

Project: **LogInSystem**

Package: **affix.java.effective.collections**

Refactor class **User** in application LogInSystemusing enum values LOGGED\_IN and LOGGED\_OUT instead of a boolean for attribute loggedIn. Add a third value BLOCKED that will be triggered when entering wrong password 3 times.

Refactor JUnit tests in order to support these changes in the LogInSystem.

**MoneyService**

**X8a – Exceptions**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

Study UML below for application **MoneyService**.

This is an application based on interface MoneyService which is implemented by main class **ExpressExchange**. The application is configured by reading data from a textfile which will set up start conditions.

Testrun the incomplete application from **ExpressExchangeApp** in order to get acquainted to the services supported.

Add missing code in class ExpressExchange according to the interface definitions for

methods buyMoney(…) and sellMoney(…)

Make sure that you support Exception best practices. Methods that deal with incorrect arguments should throw exceptions. Special circumstances, i.e. missing amount of a required Currency, should not lead to an Exception being thrown. This situation should be revealed using a state checking method.

***Tip:*** Add code to state checking method canAfford(Order), which should be called from buyMoney() and sellMoney() before a transaction is completed.

Run JUnit tests, make sure that all tests pass.

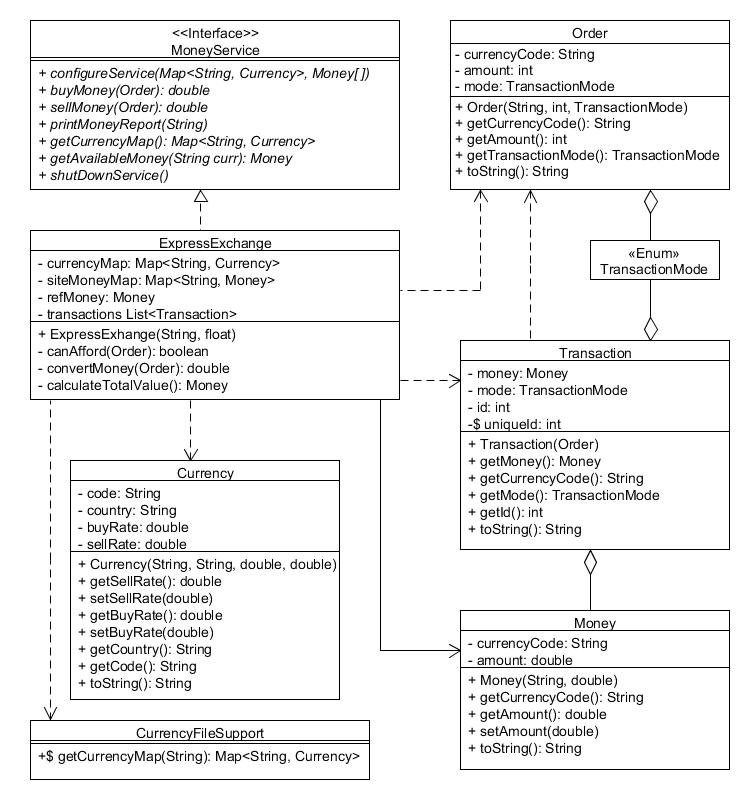
**X8b – Exceptions**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

Create **Transaction** objects at the end of methods buyMoney() and sellMoney(). These objects should be added to the prepared List< Transaction> transactions, tagged with **TransactionMode** (BUY or SELL), and later printed to console from method printMoneyReport().

**MoneyService**

****

**X9a – Serialization**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

Add serialization support for classes in MoneyService application. All transactions performed during a session should be stored in a serialized file, ***Transactions.ser***.

The serialization process should be triggered by button Exit in ExpressExchangeGUI, calling interface method shutDownService() in ExpressExchange. This method should call the prepared static method storeTransactions() in class MoneyServiceUtils

**X9b – Serialization**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

Complete code in **TransactionCollector** for reading of a serialized file recreating a List<Transaction>. Print out the contents of this List to console.

Implement sorting of the deserialized objects by defining a Comparator that use String currencyCode as first comparison criterion, and TransactionMode as second criteria.

***Tip:*** Use static methods defined in interface Comparator

**X9x – Serialization**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

Add attribute timeStamp of type LocalTime to class Transaction, which will be set by the constructor when creating a Transaction object. This attribute should also be part of the toString() representation of a Transaction object.

Create some new Transaction objects, store in a new serialized file when triggering exit of application. Make sure that the TransactionCollector can read and present the contents of the serialized file as before.

# X10 – Lambda Expressions

Project: **LambdasInAction**

Package: **affix.java8.effective.functionals**

Inspect class **TempConverter**, run the tests in main.

Add two new functions based on lambda expressions that convert a temperature in Celsius to Fahrenheit using conversion factor 9/5 and offset +32.

Produce test data and output confirming that your functions work as expected.

**X11 – Lambda Expressions**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

In methods buyMoney(…) and sellMoney(…) in class ExpressExchange you find a call to local method convertMoney(). Replace these two calls by defining two lambda expressions that will perform conversion correctly.

***Tip:*** Study types for arguments and return value for convertMoney() and define a BiFunction<T, U, R> for each transaction type (BUY/SELL). Input types T and U should be Currency and Integer respectively, R should be Double.

Run JUnit tests, make sure that all tests pass.

**X12a – Streams**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

Complete method calculateTransactionsValueForCurrency in **TransactionCollector,** which filters out all Transaction objects of a specific Currency (parameter to method), converts these objects to reference Money, sorts them in value order and finally prints them to console.

***Tip:*** Complete the stub code in valueTransformFunction when converting values. Call this function when applying the transformation part of the stream.

**X 12b – Streams**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

Add code to method sumUpTransactionValue () in **TransactionCollector** using a pipe of chained methods supported by the Streams API.

**X 12x – Streams**

Project: **MoneyService**

Package: **affix.java.effective.moneyservice**

Produce a Map<String, Transaction> based on all available Transaction objects in **TransactionCollector** using class Collectors method groupingBy(). Print out the contents of the map to console key by key.

The output should look something like the following:

Contents of transactionMap ordered by currency code

**CHF** [Transaction [id=21, mode=BUY, money=Money [Currency: CHF amount: 100,00 ]]]

**EUR** [Transaction [id=2, mode=BUY, money=Money [Currency: EUR amount: 125,00 ]], Transaction [id=10, mode=BUY, money=Money [Currency: EUR amount: 300,00 ]], Transaction [id=12, mode=BUY, money=Money [Currency: EUR amount: 300,00 ]], Transaction [id=13, mode=BUY, money=Money [Currency: EUR amount: 100,00 ]], Transaction [id=15, mode=BUY, money=Money [Currency: EUR amount: 50,00 ]], Transaction [id=3, mode=SELL, money=Money [Currency: EUR amount: 50,00 ]], Transaction [id=6, mode=SELL, money=Money [Currency: EUR amount: 200,00 ]], Transaction [id=11, mode=SELL, money=Money [Currency: EUR amount: 100,00 ]], Transaction [id=16, mode=SELL, money=Money [Currency: EUR amount: 250,00 ]], Transaction [id=17, mode=SELL, money=Money [Currency: EUR amount: 100,00 ]]]

**GBP** [Transaction [id=4, mode=BUY, money=Money [Currency: GBP amount: 700,00 ]], Transaction [id=20, mode=SELL, money=Money [Currency: GBP amount: 250,00 ]]]

…