EECS 2030 Winter 2020: Lab 5

Lassonde School of Engineering, York University

INTRODUCTION TO LAB-5

Students in the same lab section are allowed to work in groups of 2 or 3 students. This lab will be graded for style and correctness. This lab is due on **20th Mar 2020, 11pm**.

STYLE RULES FOR CODING

The style rules are not overly restrictive in EECS2030.
1) Your programs should use the normal Java conventions (class names begin with an uppercase letter, variable names begin with a lowercase letter, public static final constants should be in all caps, etc.).
2) In general, use short but descriptive variable names. There are exceptions to this rule; for example, traditional loop variables are often called i, j, k, etc. Avoid very long names; they are hard to read, take up too much screen space, and are easy to mistype.
3) Use a consistent indentation size. Beware of the TAB vs SPACE problem: Tabs have no fixed size; one editor might interpret a tab to be 4 spaces and another might use 8 spaces. If you mix tabs and spaces, you will have indenting errors when your code is viewed in different editors.
4) Use a consistent and aligned brace style.
5) Insert a space around operators (except the period ".")
6) Avoid using "magic numbers". A magic number is a number that appears in a program in place of a named constant.
7) A good IDE (integrated development environment) such as eclipse will correct many style errors for you. In eclipse, you can select the code that you want to format, right click to bring up a context menu, and choose Source → Format to automatically format code.

Learning Outcomes

By completing this lab, you are expected to have a better understanding of inheritance, polymorphism and dynamic binding, and abstract class. You are also expected to understand how Sets are implemented in Java Collection framework.

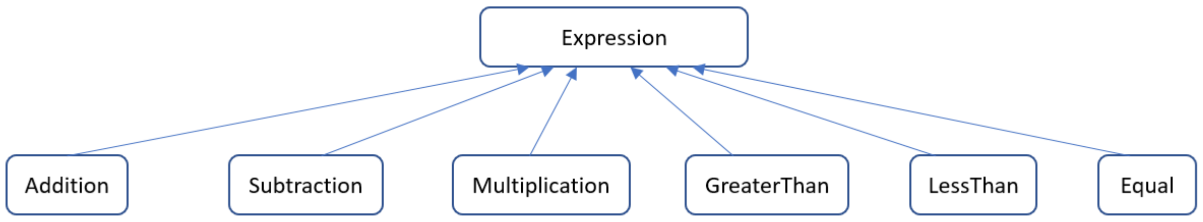
LAB EXERCISE 1. Inheritance, wrapper classes.

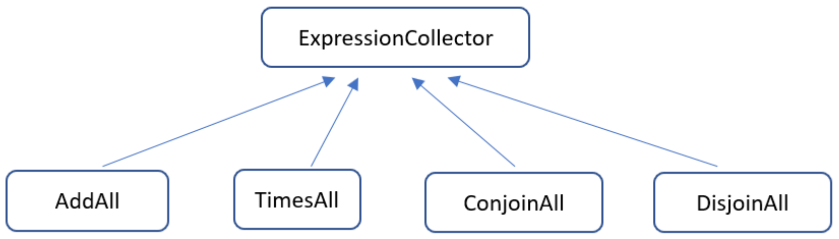
There are six kinds of (binary) expressions: **addition**, **subtraction**, **multiplications**,

equality, **greater-than**, and **less-than**. The first three kinds are arithmetic expressions which evaluate to integers. The last three kinds are relational expressions which evaluate to Booleans (i.e., true or false). These six expression classes both inherit a more general class **Expression**, which maintains some basic attributes and methods that are common to all the six expressions (e.g., attributes to store operands and value, method to get value). There are four kinds of expression collectors (each of which collecting a list of expressions):

- An **add-all** collector adds up the evaluation results of its collected expressions. When there are no expressions collected, the default evaluation result is 0.
- A **times-all** collector multiplies the evaluation results of its collected expressions. When there are no expressions collected, the default evaluation result is 1.
- A **conjoin-all** collector takes the conjunction (logical AND) of the evaluation results of its collected expressions. When there are no expressions collected, the default evaluation result is *true*.
- A **disjoin-all** collector takes the disjunction (logical OR) of the evaluation results of its collected expressions. When there are no expressions collected, the default evaluation result is *false*.

These four classes both inherit a more general class **ExpressionCollector**, which maintains basic attributes and that are common to all the four collectors (e.g., the list to maintain the expression collectors, method to get result value).





You can assume that an expression collector only evaluates its collected expressions when it is type-correct: the collected expressions are either all arithmetic expressions (for **add-all** and **times-all** collectors) or all relational expressions (for **conjoin-all** and **disjoin-all** collections). We define that two expression collectors are equal if they are both type-correct and their evaluation results are the same.

The figure above shows the two inheritance hierarchies for expressions and expression collector which you are required to implement. They are also hinted in the JUnit tests given to you. Each arrow in the diagram corresponds to the **extends** keywords in Java.

Implementation:

- Implement the given inheritance hierarchies. Given a JUnit test class (which relies on certain API of methods) and its expected outputs, complete the base classes, and implement the hinted subclasses such that all tests pass.
  You may find wrapper classes useful for manipulating results of evaluations.
  A wrapper class example is provided to you, which may help you understand how wrapper classes work.

LAB EXERCISE 2. Inheritance, abstract classes, Sets in Collections

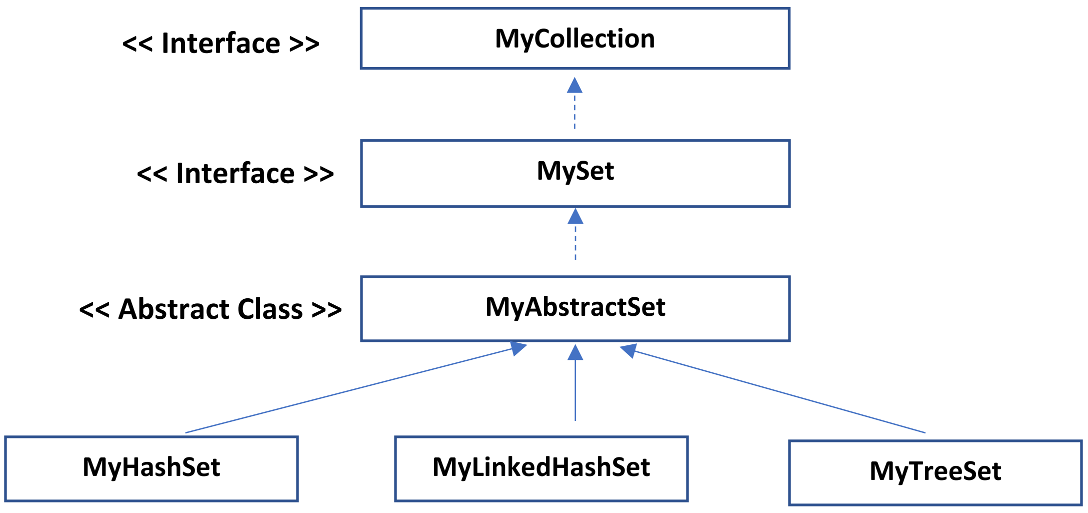
Recall that in Java collection framework, we have three concrete Set classes: **HashSet**, **LinkedHashSet**, and **TreeSet**. Among them, **HashSet** maintains no particular order of its elements, **LinkedHashSet** maintains the insertion order of its elements, and **TreeSet** maintains the natural order of its elements. We have three concrete Map classes: **HashMap**, **LinkledHashMap**, and **TreeMap**. Among them, **HashMap** maintains no particular order of its keys, **LinkedHashMap** maintains the insertion order of its keys, and **TreeMap** maintains the natural order of its keys.
So are there any relations between **HashSet** and **HashMap**, between **LinkedHashSet** and **LinkedHashMap**, and between **TreeSet** and **TreeMap**?

It turns out that we can use Maps to implement Sets. The idea is that we can use a Map to simulate a Set, where the elements of the Set are the key of the Map, and each of the keys is associated with a Dummy object as its value.

Can we say that Set "*is a*" Map with Dummy values, and thus use inheritance, letting Set extends Map? From Java's point of view of inheritance, when Set inherits (extends) Map, it implies that Set has all the functionalities of Map. However, Set does not support Map methods such as `KeySet()`, `EntrySet()`, `put(key, value)`. So this is not a good design choice.

A better approach, which Java adopts, is to use aggregation/composition. Specifically, a Set class "*has a*" Map as its attribute. Public operations on the Set, such as adding, removing, internally operate on the attribute Map, where elements of the Set are maintained as keys of the Map and values of the keys are references to an universal Dummy object.

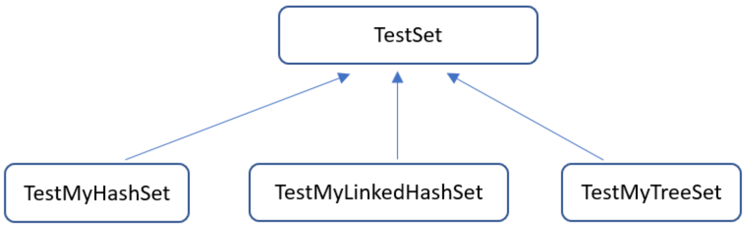
In package **myUtil**, there is an (generic) interface named **MyCollection**, which mimics the **Collection** interface in Java Collection framework, defining some basic operations for a collection, e.g., add or remove an element from a container. There is also a (generic) sub-interface **MySet** which mimics the **Set** interface, extending **MyCollection** interface but declaring no new methods (The real **Set** interface in Java repeats the methods in **Collection** and add some restrictions such as no duplicated elements. For simplicity, here **MySet** is an empty interface.) There is an abstract class **MyAbstractSet**, which mimics the **AbstractCollection** and **AbstractSet** class in Java Collection, implementing the **MySet** interface. This class defines some attributes and implements some method that can be implemented at that level (without knowing the real implementation). For example, `size()`, `isEmpty()`. Methods that cannot be implemented at this level (because they are dependent on the concrete implementation of the subclasses), such as `add()`, `remove()`, `clear()`, are declared to be abstract, which means the definition of the methods are postponed, to be given in each subclass. This abstract class are inherited (extended) by three concrete subclasses: **MyHashSet**, **MyLinkedHashSet**, and **MyTreeSet**, which mimic **HashSet**, **LinkedHashSet** and **TreeSet** respectively. The inheritance hierarchy is shown below.



Implementation:

- Complete classes **MyHashSet, MyLinkedHashSet** and **MyTreeSet,** which mimics **HashSet, LinkedHashSet** and **TreeSet** classes in Java Collection framework, respectively. These class extends the abstract class **MyAbstractSet** class, inheriting fields and methods defined in the abstract class, and are expected to implement the abstract (unimplemented) methods in the abstract class, including `add()`, `remove()`. Similar to the corresponding classes in Java, **MyHashSet** maintains a **HashMap** as its field, **MyLinkedHashSet** maintains a **LinkedHashMap** as its field, and **MyTreeSet** maintains a **TreeMap** as its field.

- Tests given to you also conform to an inheritance hierarchy (which mirrors the Set classes hierarchy in the figure above).



The abstract class **TestSet** defines common tests shared by both the three subclasses, such as tests for `add()`, `remove()`, `size()`. The only method that is abstract (unimplemented) is the `loadSet()` method (a.k.a. factory method): `protected abstract MySet loadSet();` All other test methods in **TestSet** calls this abstract method (which is to be implemented in each of the three subclasses). Both the three subclasses inherit the same set of test methods defined in base class **TreeSet**, sharing the test cases.
The subclasses also add some specific test cases to test the status of the internal maps, and for **MyLinkedHashSet** and **MyTreeSet**, also to test the order of their elements.
Definitions of `loadSet()` just need to return the corresponding subclass type, as shown in one of the subclasses. Complete the other two subclasses.
Run the three test subclasses separately to test the three concrete Set implementations. You can also run all the 3 test subclasses together using the provided test suite class named **AllTest.java**.

SUBMIT INSTRUCTIONS

FOR STUDENTS NOT WORKING IN A GROUP

Submit your solutions using the submit command. Refer to lab0 (manual) webpage for instructions. Remember that you first need to find your workspace directory, then you need to find your project directory. In your project directory, your files will be located under different package directories.

- Your **Addition** class should be in the directory **EEСS2030LAB5W20_stu/src/expressions.** Once you are in the directory, , submit the files individually, or, submit some or all files together. For example, by issuing
  submit 2030 lab5 Addition.java Subtraction.java Multiplication.java
  submit 2030 lab5 GreaterThan.java LessThan.java Equal.java
  submit 2030 lab5 AddAll.java TimesAll.java ConjoinAll.java DisjoinAll.java

- Your **MyHashSet.java** and others should be in the directory
  EECS2030LAB5W20_stu/src/myUtil
  Once you are in the directory, submit the files individually, or, submit some or all files together, for example, by issuing
  submit 2030 lab5   MyHashSet.java MyLinkedHashSet.java   MyTreeSet.java
  submit 2030 lab5 TestMyLinkedHashSet.java TestMyTreeSet.java

Alternatively, you can submit using the department's web submission system.

FOR STUDENTS WORKING IN A GROUP

If you are working in a group, one of the group members submit program files as shown above. This group member also needs to submit a text file containing EECS login names of your group members. Create a plain text file named **group.txt**. You can do this in eclipse using the menu File → New → File, or, use any other text editors. Type your login names into the file with each login name on its own line. For example, if the students with EECS login names rey, finn, and dameronp, worked in a group the contents of **group.txt** would be:
rey
finn
dameronp

Submit your group information using submit command. In the directory where the text file is saved, issue **submit 2030 lab5 group.txt**

SUBMIT INSTRUCTIONS FOR STUDENTS OUTSIDE PRISM LAB

It is possible to submit work from outside the Prism lab, but the process is not trivial; do not attempt to do so at the last minute if the process is new to you. The process for submitting from outside of the Prism lab involves the following steps:
1) transfer the files from your computer to the undergraduate EECS server red.eecs.yorku.ca
2) remotely log in to your EECS account
3) submit your newly transferred files in your remote login session using the instructions for submitting from within the lab
4) repeat Steps 1 and 3 as required

Windows users will likely need to install additional software first. Mac users have all of the required software as part of MacOS.

Alternatively, you can submit using the department’s web submission system.