# Comp 311 HW06
# Assignment 6:
# Due 11:59pm Monday, October 23, 2023
# Preliminaries

If you have no prior experience programming Java, read Chapter 1 of my monograph entitled "The Elements of Object-Oriented Program Design" (the only file in the wiki file-list https://wiki.rice.edu/confluence/display/FPSCALA/Readings). You should also download the **`drjava.jar`** file as explained below. These notes are also recommended for anyone who wants a refresher on the Java design patterns relevant to functional programming in Java.

This homework can be done using the Functional Language level of the pedagogic programming environment DrJava, which auto-generates all constructors and accessors as well as the redefinitions ("overrides") of the **`equals`** and **`toString`** methods for concrete classes on the assumption that each such class constitutes a free algebraic type. DrJava supports essentially the same interface as DrRacket. The most recent build of DrJava (a Java 8 jar file called **`drjava.jar`**) can be downloaded from https://www.cs.rice.edu/~javaplt/drjavarice/.

As an alternative, you can use a conventional Java IDE like IntelliJ. By the way, Mac OS X is hostile to unlicensed Java apps like DrJava so if your personal computer is a Mac and you already know how to write programs in Java, you should probably use conventional Java. In your Java code, you cannot use any mutation (modification of the value of a field or local variable) or Java library code (other than classes in **`java.lang.*`**. If you use conventional Java, make sure that your code runs in Java 8 and that your tests work with Junit 4. In the absence of the code augmentation performed by the Functional Language level in DrJava, you will have to define the constructors, the accessors, and the **`equals`** and **`toString`** methods for each concrete class. Your redefinition of **`equals`** should implement structural equality (the behavior of **`equals?`** in Racket). Your redefinition of **`toString`** should return a **`String`** identical to the program text that constructs the object (except that the keyword **`new`** is elided). The **`toString`** method that is automatically generated by DrJava (in the

Functional Language Level) does precisely this. Java runtime diagnostics often dump **`toString`** representations of the objects involved in aborting errors, but this messages can be inscrutable because the default implementation of **`toString`** inherited from Object is cryptic. (Try applying the **`toString`** method to an array!)

Note that DrJava only works when it is run with a Java 8 SDK (available from Amazon Corretto or Oracle).

# Composite Design Pattern for List

The following is an object-oriented formulation of lists of integers.

- **`IntList`** is an abstract list of **`int`**.
- **`EmptyIntList`** is an **`IntList`**
- **`ConsIntList(first, rest)`**, where **`first`** is an **`int`** and **`rest`** is an **`IntList`**.

The above can be implemented in functional Java (as supported by the DrJava functional language level as follows.

```
/** Abstract list structure
  * IntList := EmptyIntList + ConsIntList(int, IntList) */
abstract class IntList { }

/** Concrete empty list structure containing nothing. */
class EmptyIntList extends IntList { }

/** Concrete non-empty list structure containing an int, called first, and an
  * IntList called rest. */
class ConsIntList extends IntList {
    int first;
    IntList rest;
}
```

The above implementation is an example of what is called the **Composite Design Pattern**. The composite design pattern is a special case of the union pattern where one or more of the variants for the union type **`T`** contains fields of root type **`T`**. In this pattern, the union is called a composite. Here the union type is **`IntList`** and the variant **`ConsIntList`** is said to be a composite because it includes a field of type **`IntList`**.

The composite pattern also prescribes a coding pattern for the methods that process the composite type. Typically, the method code for each variant class derived from the abstract class that is the "parent" class for the variants. When a variant is called to perform an operation, the code in the variant traverses its fields of root type and calls on them to perform the same operation. It allows a client to treat an instance of type **`T`** and its embedded instances uniformly using polymorphism.

This coding pattern is called the *interpreter design pattern*: it interprets the abstract behavior of a class (as specified in the contract of the abstract method) in each of its concrete subclasses. The composite pattern refers to the structure of the composite type hierarchy, while the interpreter pattern refers to how the behavior of the variants of the type are defined uniformly via object-oriented polymorphism.

# Interpreter Design Pattern for List

The interpreter design pattern applied to the above composite list structure prescribes a coding pattern for list operations that is analogous to Racket function template. It entails declaring an abstract method for each list operation in the abstract list class, **`IntList`**, and defining corresponding concrete methods in the concrete list subclasses: the empty list class, **`EmptyIntList`**, and the non-empty list class, **`ConsIntList`**. The concrete method for **`EmptyIntList`** corresponds to the base case in the Racket function template while the concrete method in **`ConstIntList`** corresponds to the recursive case by calling the same method on its **`rest`**.

The following is the coding template for the interpreter design pattern for **`IntList`** and its subclasses.

```
abstract class IntList {
    abstract returnType methodName(parameter_list);
}

class EmptyIntList extends IntList {
    returnType methodName(parameter_list) {
        // base case code
    }
}
```

```
class ConsIntList extends IntList {
    int first;
    IntList rest;
    returnType methodName(parameter_list) {
        // ... first ...
        // ... rest.methodName(parameter_list) ...
    }
}
```

# Problems

In your assignment repository, the stub file **`IntList.dj`** (named with file extension **`.dj`** for compatibility with Functional Java in Drjava) contains essentially the code given above. If you use an IDE like IntelliJ, you should rename **`IntList.dj`** as **`IntList.java`** and fill in the (trivial) definitions of the auto-generated constructors, accessors, and the **`equals`** and **`toString`** methods in all concrete classes. For each problem below simply augment the three classes provided in the stub file (or their renamed equivalents if you are using conventional Java). In addition, edit the accompanying JUnit test file (following the requirements of Junit 4) named **`IntListTest.dj`** (or **`IntListTest.java`** in conventional Java) to create unit tests for each problem. DrJava requires the names of JUnit test files to end with the letters **`Test`**, which is probably a good idea anyway. Our grading script which uses DrJava will handle either name. Drjava uses the file extension to determine if a file is an ordinary Java file or a Functional Java file. Place the tests for each problem in a test method with a name matching the method being tested. For example, the name for the test method for **`contains`** should be named **`testContains`** or something similar. (The exact method names do not matter as long as the begin with the prefix **`test`**, since we will run your **`IntListTest`** class using a Junit 4 runner which uses reflection to discover all methods beginning with the prefix **`test`**.)

Apply the interpreter design pattern to **`IntList`** and its subclasses provided in the **`IntList.dj`** file to write all of the following methods as augmentations (additional code) of the **`IntList`** class. In addition, as stated above, edit the provided JUnit test class, **`IntListTest`** to create tests for all of your non-trivial (everything but successors) methods in the **`IntList`** class. (Note: if you use conventional Java you have to write a few more tests because your **`IntList`** class contains more methods.)

- (10 pts.) **`boolean contains(int key)`** returns **`true`** if **`key`** is in the list, **`false`** otherwise.
- (10 pts.) **`int length()`** computes the length of the list.
- (10 pts.) **`int sum()`** computes the sum of the elements in the list.
- (10 pts.) **`double average()`** computes the average of the elements in the list; returns **`0`** if the list is empty.
  **Hint**: you can *cast* an **`int`** to **`double`** by using the prefix operator **`(double)`**.
- (10 pts.) **`IntList notGreaterThan(int bound)`** returns a list of elements in this list that are less or equal to **`bound`** .
- (10 pts.) **`IntList remove(int key)`** returns a list of all elements in this list that are not equal to **`key`** .
- (10 pts.) **`IntList subst(int oldN, int newN)`** returns a list of all elements in this list with **`oldN`** replaced by **`newN`** .
- (30 pts.) **`IntList merge(IntList other)`** merges **`this`** list with the input list **`other`**, assuming that **`this`** list and **`other`** are sorted in ascending order. Note that the lists need not have the same length.

  **Hint**: add a method **`mergeHelp(ConsIntList other)`** that does all of the work if one list is non-empty (a **`ConsIntList`**). Only **`mergeHelp`** is recursive. Use dynamic dispatch on the list that may be empty. Recall that **`a.merge(b)`** is equivalent to **`b.merge(a)`** . This problem is the Java analog of the **`merge-help`** function that you wrote in Assigment 2.

You may find it helpful to write Template Instantiations for all of the methods that you define as an intermediate step in developing your code **BUT DO NOT submit** these Template Instantiations (or corresponding Templates) as part of your code documentation. The structure of your program implicitly provides this information. Confine the documentation of your Java code to writing contracts using **`javadoc`** notation, opening the behavioral contract (preceding the corresponding definition) with **`/**`** and closing it with **`*/`**. For inherited methods do not repeat contracts given in superclasses.

This assignment is intentionally very easy (a reprise of HW1 and HW2 in Functional Java instead of Racket) so you can become familiar with writing functional code in Java and writing unit tests for the defined methods.

# Hints

1. You can simplify your coding if you add some “convenience” fields and methods to the abstract class **`IntList`** such as:

   ```
   static final EmptyIntList EMPTY = new EmptyIntList();
   ConsIntList cons(int n) { return new ConsIntList(n, this); }
   ```

   which enables you to write

   ```
   EMPTY.cons(2)
   ```

   Instead of

   ```
   new ConsIntList(2, new EmptyIntList())
   ```

   The stub file already includes the two members shown above in **`IntList`**.

2. Avoid using the **`public`** attribute in general (except for methods in interfaces which *must* be public) and particularly for classes because Java has some funny rules about the names of files containing public classes (and how many public classes can be placed in a single file). So we will not declare our classes as **`public`**.