# 1. Introduction

In this session, we will have the opportunity to review data type inheritance using the Java language, and, at the same time, work with some simple ADTs, their implementation, and testing. Different issues regarding inheritance will be addressed, requiring you to put in practice your knowledge about this topic. The goal is not to study this topic from scratch; we assume that you have a basic knowledge about it, at least at the level of what is covered in the CS1 course. The activities include exercises in the following topics:

* Java interfaces
* Inheritance and data type hierarchies
* Polymorphism
* Code reusability

The topics covered here are to be extensively used throughout the course, and they are of extreme importance in modern software development based on the object-oriented methodology. Most of the discussion here can be found in the textbook (See Ch. 2), although some modifications have been introduced for the purpose of the exercises and activities in agenda.

We will consider linear ordered structures as follows. These are collections of 1 or more elements in which the order of elements is important. For any given instance, if not empty, each one of its elements occupies a particular position that is identified from other positions by an *index value*. Such collections can be finite or infinite. If finite, index values range from 1 to n or from 0 to n-1, where n is the length of the particular instance[[1]](#footnote-0). In the case of an infinite structure, the range of index values covers all positive integers and perhaps 0. Two instances of such type of object are equal, if and only if, they are of the same type, they have equal lengths, and elements in positions with the same index value in both instances are equal. Some examples of this type of structure to be used in this activity are: *tuples* and *mathematical progressions*. In the case of mathematical progressions, we will work with two particular types of progressions: *arithmetic progression* and *geometric progression*. A brief definitions for these structures follows.

* *Tuple* – is an ordered structure consisting of a finite number of one or more components. Each component occupies a fixed position in the structure. For example: an ordered pair is a tuple of length 2. It has two components: first component and second component. A triple is a tuple of length 3. In general, a tuple of length n (for n>0) has n components. On any instance, each component in a tuple contains a particular value. Tuples are usually represented as: (c1, c2, …., cn), where n is its length. Components are distinguished from each other by an index value that is unique, in this case, in the range from 1 to n. For a tuple of length n, we use index 1 to refer to the first component, index 2 to refer to the second component, and so on, up to index n to refer to its n-th component. A tuple has at least the following operations: to determine its length, to access a particular component value, and to modify the current value in a particular component. For instance, the *ordered pair* is a particular type of tuple; that is, a tuple of length 2.

Examples:

(2, 3) – Component values are 2 and 3

(3, 4, 3, 3) – Component values are 3, 4, 3, and 3

(10) – Only one component

* *Progression* – is a mathematical construct that defines a sequence of one or more real numbers in a given order. The sequence may be finite or infinite. Its definition usually consists of a fixed number of initial values in the sequence, plus some mathematical rule that states how to determine new elements in the sequence based on previous elements. We say that a closed form solution for a particular progression can be derived if it is possible to have an expression that exactly computes any term of the progression as a function of n, the index value for the particular term, and which does not depend on any other term.

Examples:

* 1. *Fibonacci progression*: first two elements are defined as 1. Starting with the third element, the value corresponding to that particular element is determined by adding the previous two. The first 10 values in this type of sequence are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … We can derive a closed form solution for this progression, but we will not use it here.
  2. *Arithmetic progression*: The first value is given. Starting with the second element, the value corresponding to that particular element is determined by adding a fixed amount (common difference) to the previous element. For example, the first 10 values in an arithmetic sequence whose first value is 5 and the difference between two consecutive elements is 3 are: 5, 8, 11, 14, 17, 20, 23, 26, 29, 32. The following is a *closed-form* solution for this. Let *t(n*) be the *n*-th term in the progression. Then we can show *t(n) = (first value) + (common difference)\*(n-1).*
  3. *Geometric progression:* The first value is given. Starting with the second element, the value corresponding to that particular element is determined as the product of a fixed value (common factor) and the previous element. For example, the first 10 values in a geometric sequence whose first value is 3 and the quotient between two consecutive elements is 2 are: 3, 6, 12, 24, 48, 96, 192, 384, 768, and 1536. The following is a *closed-form* solution for this progression. Let *t(n*) be the *n*-th term in the progression. Then we can show *t(n) = (first value) \* (common factor)(n-1)*, where *xr* means *x* to the power *r*.

In this lab, we will first work with tuples, arithmetic sequences, and geometric sequences, using them to have a chance to experiment with inheritance and other related issues using Java. As an exercise, you will implement a class corresponding to the Fibonacci Sequence. For the purpose of this activity, we are assuming that the first element or term in these structures has index 1 (not 0), that the second element has index 2, and so on.

# **2. Interfaces, Inheritance, and Polymorphism**

The hierarchical organization of data types has proved to be a valuable alternative to manage the complexity of using several different data types in computer systems and achieve code reusability in a quite intuitive manner. In addition, they are useful in writing generic code. The Java programming language provides different constructs to support this.

A Java *interface* is a specification of a data type. It declares what operations such data type must have. Interfaces need to be implemented by classes as a way to make real the particular data type they specify. An *abstract class* is also a specification of a data type, but different from an interface in that it can include partial implementation of methods that such data type must have. They need to be eventually extended by other non-abstract classes for the specified data type.

Besides supporting hierarchical structuring of data types, the Java language supports *polymorphism*. This refers to operations that are automatically adapted to execute different algorithms based on the type of object it is applied to during the execution of the program. For instance, if I1 is an interface for which a method f() is specified, and x is a reference to an object of type I1 (the object that x points to has a data type corresponding to a class that implements I1 or some other interface that extends I1 or is a subclass of another class that implements I1), then the statement x.f() executes the particular method f() corresponding to the implementation of the class data type of the object that x is currently pointing to. As you should know, the particular instance of f() that will be executed is the one found in that particular class, or, if not there, the first one found when going upward in the hierarchy of classes, in the path towards the class Object, which is at the top of the hierarchy.

When designing data types, one needs to decide which should be specified as interfaces, which need to be partially implemented as abstract classes, and which need to be implemented as classes. In addition, it is required to determine which classes need to be implemented as subclasses of other classes. Usually, such hierarchy is not the first thing one comes up with; the need for certain data types is discovered first. The hierarchy may come after careful analysis of the different data types and discovery of the hierarchical relationship that may be exploited. When doing this, new data types integrating common properties of already discovered data types may be introduced. Those become super classes of the ones being integrated, or may be specified as interfaces that the others have to implement. If that integration is achieved by a class, then it is usually implemented as an abstract class whenever the particular data type it represents, by itself, does not represent a fully functional object unless extended somehow (by a subclass).

# 3. Hierarchical Organization of Some Ordered Structures

Consider the three types of ordered structures that were described. We can distinguish different operations that may be applied to some of these data types.

* **int length()** – number of elements in the structure. This operation only applies to ordered structures that are tuples, or that are of finite length.
* **double firstValue()**- returns the first value in the ordered structure. This operation applies only to ordered structures that are progressions; it can be applied at any time. Besides returning the first value in the progression, the object instance is left in a state such that if the **nextValue()** operation is executed afterwards, then the value to return will be that of the second term in the progression.
* **double nextValue()**- this operation is meant to be used when iterating over a progression. It is not valid if **firstValue()** has not been executed on the same structure. If valid, the first time it is executed it returns the second element. In general, if it is valid to execute (after the last **firstValue()** has been executed), then the n-th execution of this operation returns the (n+1)-th value in the structure (the one corresponding to the (n+1)-th term).

For example: if x is a reference to a Fibonacci sequence, then the following shows the values returned in the execution of each of the statements given:

**x.firstValue()** returns 1

**x.nextValue()** returns 1

**x.nextValue()** returns 2

**x.nextValue()** returns 3

**x.nextValue()** returns 5

**x.firstValue()** returns 1

**x.nextValue()** returns 1

**x.nextValue()** returns 2

If x is a reference to an Arithmetic sequence, whose first value is 3 and common difference is 4, then the following shows the values returned in the execution of each of the statements given:

**x.firstValue()** returns 3

**x.nextValue()** returns 7

**x.nextValue()** returns 11

**x.nextValue()** returns 15

**x.nextValue()** returns 19

**x.firstValue()** returns 3

**x.nextValue()** returns 7

**x.nextValue()** returns 11

* **double getTerm(int index)** – returns the term value at the particular position (corresponding to **index**) in the current instance of the structure. This operation can be applied to any ordered structure. The value of **index** has to be valid; otherwise, some exception is thrown. This happens in general if the value of index is less than 1 or, in the case of the **Tuple**, if such value exceeds the length of the structure. When applied to a progression, the correct execution of this operation does not alter its “current” value.
* **void printAllTerms(int n)** – outputs (prints) the first n terms of the particular ordered structure. It can be applied to any ordered structure, but the value of n must correspond to a valid index, just as in the method **getTerm(…)** previously described.
* **void setTerm(int index, double value)** – changes the current value of the particular term (corresponding to **index**) to value. The value of **index** has to be valid. This method makes sense only for tuples. The reason is that to change any term in a progression (as are described here) is not a valid operation since it may leave the particular sequence in an inconsistent state. For example, for an arithmetic progression the n-th term depends on what the first value and common factor values are. The terms cannot change unless those two values are changed.

Based on the previous discussion, we can discover a strategy to implement the hierarchy of ordered structures data types that has been described. Bear in mind that the goal is to facilitate the management of the different data types and to achieve code reusability as much as possible. It is also important to recall that in general there are not unique ways to design solutions to computational problems, hence any strategy followed here may be modified and perhaps replaced with a better one. (That is precisely why analysis is important and why there is a need of Computer Engineers or Computer Scientists)

We can identify the following data types : **OrderedNumberStructure**, **Tuple**, **Progression**, **Geometric**, **Arithmetic**, and **Fibonacci**. The hierarchy is established by the following (in the context of this activity – this might be considered different in other contexts):

* tuples are ordered structures that are not progressions
* geometric sequences, arithmetic sequences, and Fibonacci sequences are different types of progressions.

From the operations that were described, we have:

* Any type of *ordered structure* must support the two operations: **getTerm**, and **printAllTerms**.
* An *ordered structure* that is a **Tuple** must support, in addition, the following operations: **length** and **setTerm**.
* An ordered structure that is a **Progression** must support in addition the following operations: **firstValue** and **nextValue**.
* The ordered structures that are **Geometric**, **Arithmetic**, and **Fibonacci** only support (at least in the current context) those operations that are supported by **Progression** (and hence those inherited from **OrderedNumberStructure**), but what makes them different is in the way each of those operations is implemented – they must be consistent with their respective descriptions.

The following figure shows a hierarchical organization of the data types that have been discussed, and which is consistent with the descriptions given[[2]](#footnote-1). The class **Fibonacci** is excluded for the moment. Remember that any operation that has the same name as the class it belongs to is a *constructor* for that type of object. Notice that **OrderedNumberStructure** is designed as a *Java* *interface*, while **Progression** is an *abstract class*. The others are *Java* *classes*.



Figure 1. Hierarchical relationship

For each one of these classes, we include its instance fields, as well as the methods that it must implement. Under this design strategy, the **Progression** is defined as an *abstract class*, but only one method is abstract: **nextValue**. The reason is that its implementation needs to know the details of the particular sequence. The other methods, **getTerm**, **printAllTerms**, and **firstValue**, can be implemented since they can be implemented by properly invoking other methods or from the actual instance fields of the class. In addition, a progression object has a property, say “*current value*”, which represents what the *current value* is in the particular object. Such value is initially its first value, but changes any time the **nextValue** method is executed on the object.

For example, one possible implementation for that class is as follows:

**public abstract class Progression implements OrderedNumberStructure {**

**private double first; // the first value**

**protected double current;**

**// current is the current value of the object – it changes**

**// to “the value of the next term” whenever method**

**// “nextValue” is applied to the object.**

**public Progression(double first) {**

**this.first = first;**

**current = first;**

**}**

**public double firstValue() {**

**current = first;**

**return current;**

**}**

**public void printAllTerms(int n) throws**

**InvalidParameterException**

**{**

**if (n <= 0)**

**throw new InvalidParameterException(**

**"printAllTerms: Invalid argument value = " + n);**

**System.*out*.println("Index --- Term Value");**

**for (int i=1; i<=n; i++) {**

**System.*out*.println((i) + "---" + this.getTerm(i));**

**}**

**}**

**public double getTerm(int n) throws IndexOutOfBoundsException {**

**if (n <= 0)**

**throw new IndexOutOfBoundsException(**

**"printAllTerms: Invalid argument value = " + n);**

**double value = this.firstValue();**

**for (int i=1; i<n; i++)**

**value = this.nextValue();**

**return value;**

**}**

**public abstract double nextValue();**

**}**

Remember that we cannot directly instantiate objects of a type defined by an abstract class. For instance, we cannot use: **new Progression(…)**. Such classes need to be extended. In this particular case, we need to implement extension subclasses for each of the particular types of progressions that were described. Each of those subclasses will have the necessary implementations of the abstract methods.

# 4. Activities for this Lab

We now present some exercises based on the previous discussion. You already have access to Java code corresponding to some partial implementations of the structures just described. Each exercise asks to work upon such code in order to comply with its particular specifications.

**Exercise 1.** Import the code (partial project) received as a new project in Eclipse. Run class **ProgressionTester1**. The output to produce is shown in the file named “**output1**”, located inside package **output**. (Answer “proceed” to the error message displayed by Eclipse when you try to initiate the execution)

**Exercise 2.** Implement a second tester: **ProgressionTester2** – the tester should be able to declare a variable of type **Progression** and output the sum of the first **n** terms for any value of **n** that is *greater than zero*.

**public class ProgressionTester2 {**

**public static void main(String[] args) {**

**Progression p = new Arithmetic(3, 2);**

**// outputs the sum of first 5 terms in p**

**printSumOfTerms(p, 5);**

**p = new Geometric(3, 2);**

**printSumOfTerms(p, 5);**

**}**

**/\*\* Prints the sum of the first terms in a**

**progression.**

**@param p the progression**

**@param n the number of terms to consider**

**\*\*/**

**private static void printSumOfTerms(**

**Progression p, int n)**

**{**

**// pre: n is valid**

… add code to compute, and assign to **sum,** the sum of the first **n**

terms in **p**

**System.*out*.println("Sum of first " + n**

**+ " terms in " + p + " is: " + sum);**

**}**

**}**

Once you complete your code, the output (assuming no other changes besides adding the missing code to compute the sum) is the following:

**Sum of first 5 terms in orderedStructures.Arithmetic@b4199 is: 35.0**

**Sum of first 5 terms in orderedStructures.Geometric@c29ab2 is: 93.0**

Note: your output may have a different hexadecimal value immediately after the @ symbol.

**Exercise 3.** Modify the implementation in order to comply with the following. We would like the output produced by class **ProgressionTester2** (with no changes to the current version of that class) to be as follows:

**Sum of first 5 terms in Arith(3, 2) is: 35.0**

**Sum of first 5 terms in Geom(3, 2) is: 93.0**

What you need to do is just add the appropriate toString method to each of the classes: **Arithmetic** and **Geometric**. No other changes are allowed.

**Exercise 4.** Introduce a new class, **Fibonacci**, to implement the **Fibonacci** progression. No modification is allowed in class Progression – it must work as it is. Part of the class is as follows, you just need to complete the method **nextValue**, without changing what is already given.

**public class Fibonacci extends Progression {**

**private double prev;**

**public Fibonacci() {**

**this(1);**

**prev = 0;**

**}**

**private Fibonacci(double first) {**

**super(first);**

**}**

**@Override**

**public double nextValue() {**

**// add the necessary code here**

…

**return current;**

**}**

**public double firstValue() {**

**double value = super.firstValue();**

**prev = 0;**

**return value;**

**}**

**}**

Test your solution by removing the proper comment lines inside **ProgressionTester1**. The output is shown in file named **output2**.

**Exercise 5.** You should notice that in this type of hierarchy of data types, the higher a particular method is implemented (the closest to the data type on top), the more code reusability can be achieved. However, this usually comes with more inefficient execution for some particular subtypes since the implementation has to be more generic. This is true because the lower (in the hierarchy) the implementation is done, the more knowledge one has about the particular implementation of the data type represented by the class where it takes place. That knowledge can be usually exploited to produce more efficient solutions. For instance, take the case of the method **getTerm**. In the hierarchy given, it has been implemented only once, inside class **Progression**. Notice that there is no implementation for this method below this point in that hierarchy. What we can do is to leave it as it is, and, for those subtypes in which we can exploit particular characteristics to provide more efficient solutions, do so by overriding that method. For example, for the cases of the **Geometric** and the **Arithmetic** data types, we can exploit the knowledge of the closed-form solutions (shown elsewhere in this document) for each term in the progression.

Modify classes **Geometric** and **Arithmetic** by adding the proper implementation of method **getTerm** on each. No modification to other parts of the code is allowed. In both cases, a constant time solution can be written for these particular operation.

After completing these modifications, if you run the tests already in place, you should get the same results (if not, something is wrong with your implementation) and, even though it is hard to notice for the given tests, the execution time should now be faster.

**Exercise 6.** Modify your implementation to comply with the following: As described in the previous discussion, the **nextValue()** operation is invalid if the **firstValue()** method has not been previously executed, at least once, since the creation of the particular progression instance. Add the necessary code to make it throw exception **IllegalStateException** whenever that invalid state is detected. Test.

1. For certain types of ordered structures, it is common to consider the index of the first element as 0; while for others, the first index is 1. [↑](#footnote-ref-0)
2. Symbols +, -, and # correspond to access types *public*, *private*, and *protected*, respectively. [↑](#footnote-ref-1)