0.1 Introduction to Programming Paradigms

0.1.1 In how many different ways can we reverse a sequence of integers?

We start the lecture by looking at three different Java programs which reverse a sequence of integers.
The *C-programmer* style:

```
public class Rev {
        public static void show (Integer[] v){
                for (Integer i:v)
                        System.out.println(i);
        }
        public static void main (String[] args) {
                Integer[] v = new Integer[] {1,2,3,4,5,6,7,8,9};
                int i=0;
                while (i < v.length/2){
                        int t = v[i];
                        v[i] = v[v.length-1-i];
                        v[v.length-1-i] = t;
                        i++;
                }
                show(v);
        }
}
```

The first approach stores the sequence as an array, and performs reversal in-place, by swapping the first half of the array with the second. The code is **compact** and fairly **efficient** in terms of computational complexity.
The *functional* style:

```
class List {
        Integer val;
        List next;
        public List(Integer val, List next){
                this.val = val;
                this.next = next;
        }
        @Override
        public String toString(){
                if (next == null)
```

```
                        return val.toString();
                return val.toString()+" "+next.toString();
        }
}
public class FuncRev {

        private static List rev(List x, List y){
                if (x == null)
                        return y;
                return rev(x.next,new List(x.val,y));
        }
        public static List reverse (List l){
                return rev(l,null);
        }
        public static void main (String[] args) {
                List v = new List(1, new List(2, new List(3, new List(4, new List
                System.out.println(reverse(v));
        }
}
```

Unlike the former approach, here the code focuses more on **input-output**: reversal takes a list (instead of an array), and produces another list. The strategy relies on an auxiliary function `rev` which uses an accumulator to reverse the list. Both `rev` and the display function are recursive. However, `rev` is **tail-recursive** hence quite efficient.
The *object-oriented* style:

```
import java.util.Iterator;

class RVList<T> implements Iterable<T> {
        private T[] array;

        public RVList(T[] array){
                this.array = array;
        }

        @Override
        public Iterator<T> iterator() {
                return new Iterator<T>(){
                        private int crtIndex = array.length - 1;
```

```
                        @Override
                        public boolean hasNext(){
                                return crtIndex >= 0;
                        }

                        @Override
                        public T next(){
                                return array[crtIndex--];
                        }

                        @Override
                        public void remove () {}
                };
        }
}

public class OORev {

        public static void main (String[] args) {
                String[] s = new String[]{"1", "2", "3", "4", "5", "6"};

                Iterator<String> r = (new RVList<String>(s)).iterator();
                while (r.hasNext()){
                        System.out.println(r.next());
                }
        }
}
```

The final alternative relies on the Java concept of **iterator** of collections. Informally, a collection of elements has the property that its elements can be *enumerated* or *iterated*. Any such collection is an `Iterable` (extends the Java interface Iterable). Here, `RVList` is such an iterable collection, which takes its elements at construction, from an array. `RVList` is **generic**, hence its elements can be of any type `T`.
The fundamental trait of an `Iterable` object is that it implements the method `iterator`, which returns just that. An **iterator** is an object which allows enumerating all elements of the collection at hand, via the methods `hasNext` and `next`. Sometimes (as in the case of a `Set`) the order is unimportant. Here, the order matters - elements are explored in their reverse order. Technically, the iterator object is an **anonymous class**.
This alternative focuses on **list traversal**, and on the **generality** of the approach.

Basically, any Java collection (for which order makes sense) can be transformed to an array (via the `toArray` method) and subsequently - an `RVList` which can be traversed in reverse order via its iterator.

0.1.2 Why so many reversals?

The reason for choosing reversal as a running example is that the task is algorithmically trivial: take the sequence of elements of the collection at hand, be it array of list (or anything else), in their **reverse** order.
However, there are **many different ways** for implementing reversal, and each makes perfect sense in some setting. Moreover, some of these ways are: **subtle**, **conceptually challenging**, and require **higher skill/knowledge** in operating the programming language at hand.
Our point, and one of the major objectives of this lecture, is to emphasise that, apart from developing fast and correct algorithms, a task of equal challenge and importance is **writing down the code**, in the **suitable programming language**.
We have clear metrics for choosing algorithms. Do we also have metrics for **code writing**? For instance:

- legibility
- compactness (number of code lines)
- ease of use (can other programmers use the code easily)
- extensibility (can other programmers add modifications to the code easily)
- good documentation

is a plausible list. While some of these criteria overlap and are difficult to assess objectively, programmers more often than not agree that some programs are **well-written while others are poor** (w.r.t. some criteria).

0.1.3 How many other ways?

There are many possible variations to our three examples, but a few elements do stand out:

- the functional style for **representing a list** in the second example - very akin to Abstract Datatypes
- the functional style for reversal - relying on recursion, in the same example

- the Object Oriented style for **traversing a list** in the third example - which although is tightly linked to Java Collections, can be migrated to any other object-oriented language.

These **elements of style** are frequently called **design patterns** - generic ways of writing code, which can be deployed for different implementations.
Some elements of style may have some common ground, or rely on certain traits of the programming language. These latter are called programming **paradigms**. For instance, writing programs as recursive functions as well as representing data as ADTs (recall that constructors *are* functions) are both styles of the **functional** paradigm.
In this lecture, we shall go over the following paradigms:

- imperative
- object oriented
- **functional**
- **logical** (or logic programming)
- **associative** (or rule-based)

0.2 Other questions

- Is there a **right** programming language? How to people choose programming languages?
- Who invented paradigms? (A history between Lisp vs C)
- Relationship between paradigms and programs (one-to-one, one-to-many, many-to-many)
- How **extensible** can programs be?