# Exercises
# Collection Classes

---

## 12.1 Shuffle List

Write a `Shuffle` operation that disorders the elements of a collection in a random fashion. A shuffle operation is useful in many context. There is no `Shuffle` operation in **`System.Collections.Generic.List<T>`**. In the similar Java libraries there is a shuffle method.

In which class do you want to place the `Shuffle` operation? You may consider to make use of extension methods.

You can decide on programming either a mutating or a non-mutating variant of the operation. Be sure to understand the difference between these two options.

Test the operation on **`List<Card>`**. The class `Card` (representing a playing card) is one of the classes we have seen earlier in the course.

Solution

---

## 12.2 Course and Project classes

In the earlier exercise about courses and projects (found in the lecture about abstract classes and interfaces) we refined the program about `BooleanCourse`, `GradedCourse`, and `Project` to use a common superclass called `Course`. Revise your solution (or the model solution) such that the courses in the class `Project` are represented as a variable of type `List<Course>` instead of by use of four variables of type `Course`.

Reimplement and simplify the method `Passed` in class `Project`. Take advantage of the new representation of the courses in a project, such that the "3 out of 4 rule" (see the original exercise) is implemented in a more natural way.

Solution

---

## 12.3 Switching from Dictionary to SortedDictionary

The program on this slide instantiates a **`Dictionary<Person,BankAccount>`**. As recommended earlier in this lecture, we should work with the dictionary via a variable of the interface type **`IDictionary<K,V>`**.

You are now asked to replace **`Dictionary<Person,BankAccount>`** with **`SortedDictionary<Person,BankAccount>`** in the above mentioned program.

This causes a minor problem. Identify the problem, and fix it.

Can you tell the difference between the output of the program on this slide and the output of your revised program?

You can access the `BankAccount` and `Person` classes in the web version of the material.

Solution

---

## 12.4 Explicit use of iterator - instead of using foreach

In this program we will make direct use of an iterator (an enumerator) instead of traversing with use of foreach.

In the animal collection program, which we have seen earlier in this lecture, we traverse the animal collections several times with use of foreach. Replace each use of foreach with an application of an iterator.

Solution

---

## 12.5 Using multiple interators

In this exercise we will see how to make good use of two interators of a single collection. Our starting point will be the type `Interval`, and the iterator which we have programmed for this type earlier in this teaching material.

For a given interval `I`, generate a list of all possible pairs (e,f) where both e and f come from `I`. As an example, the interval `new Interval(1,2)` should give rise to the list (1, 1), (1, 2), (2, 1), and (2, 2). For the purpose of generating the list of all such pairs, request two iterators from the interval, and traverse the interval appropriately in two nested while loops.

Like in the previous exercise, it is suggested that you use the operations in `IEnumerator`. Such a solution gives the best understanding of the use of multiple iterators. It is, however, also possible to solve this exercise by nested foreach loops.

Solution

---

## 12.6 The iterator behind a yield

Reprogram the iterator in class `GivenCollection` without using the **yield return** statement in the `GetEnumerator` method.

---

## 12.7 Infinite Collections of Integers

*In several respects, this exercise previews ideas from LINQ.*

Program a class `IntegersFrom` which represent an inifinite sequence of integers (of type `long`) from a given starting point. The class must implement the interface `IEnumerable<long>`.

First, program a version without use of **yield return**, and next program a version which uses **yield return**. Compare the two versions. (It is surprisingly tricky to program the version which uses native iterators (enumerators) in C#, without using **yield return**. You may chose to study my implementation (in the solution) instead of spending time on programming the class yourself.)

As an alternative to the class `IntegersFrom`, make an extension method `AdInifinitum` (which means 'to inifinity') in the type `long` , which enumerates an infinite sequence of integers from a given starting point:

```
public static IEnumerable AdInfinitum(this long from){...}
```

Make an additional extension method in the interface `IEnumerable<long>` which filters an infinite sequence of integers with use of a given predicate

```
public static IEnumerable<long> Filter(this IEnumerable<long> source,
                                       Func<long, bool> pred){...}
```

Use `Filter` to obtain all even integers, and all primes.

Finally, program an extension method in `IEnumerable<long>` that adds two infinite sequences of integers together (pair-by-pair):

```
public static IEnumerable<long> Add(this IEnumerable<long> source,
                                    IEnumerable<long> other){...}
```

Add all *natural numbers*, `1L.AdInfinitum()`, to itself, and get all even numbers again.

Solution

---

Generated: Monday February 7, 2011, 12:21:46