Programming 2

Course material

Moodle-course:

- “2021 IT1.2 Programming”

Yellow Book:

- *references on Moodle*

Assignments:

- weekly (6x), individual, mandatory
- hand-in on Moodle (CodeGrade), deadline following week
- <u>all</u> AutoTests <u>and</u> manual check must pass (10/10)

Program term 1.2

01 (wk-46)	enumerations / structures / classes
02 (wk-47)	2-dim arrays / flow control
03 (wk-48)	lists / dictionaries
04 (wk-49)	file I/O / error handling
05 (wk-50)	program structure
06 (wk-51)	program structure
07 (wk-52)	Christmas holiday
08 (wk-53)	Christmas holiday

---

09 (wk-01)	practice exam
10 (wk-02)	*exams*
11 (wk-03)	*retake exams*
12 (wk-04)	*retake exams*

Rules concerning methods

- This term (Programming 2) we will be using a lot of methods, since the programs will get larger and larger, and we want to keep them readable…

- Rules concerning methods:
  1) give each method a meaningful name
  2) one method, one task! *(don't mix multiple tasks in one method)*
  3) each method has a maximum of **30 code-lines**

static methods…

First of all… get rid of 'static' methods!

```
static void Main(string[] args)
{
    Console.Write("Enter a year: ");
    int year = int.Parse(Console.ReadLine());

    if (IsLeapYear(year))
        Console.WriteLine($"{year} is a leap year.");
    else
        Console.WriteLine($"{year} is not a leap year.");

    Console.ReadKey();
}

static bool IsLeapYear(int year)
{
    if (year < 0) return false;

    bool deelDoor400 = ((year % 400) == 0);
    bool deelDoor100 = ((year % 100) == 0);
    bool deelDoor4 = ((year % 4) == 0);

    return (deelDoor400 || (deelDoor4 && !deelDoor100));
}
```

We don't want to use 'static' anymore!

From now on, the main will be like…

```
class Program
{
    static void Main(string[] args)
    {
        Program myProgram = new Program();
        myProgram.Start();
    }

    void Start()
    {
        int year = 1900;
        if (IsLeapYear(year))
            Console.WriteLine($"{year} is a leap year.");
        else
            Console.WriteLine($"{year} is not a leap year.");
    }

    bool IsLeapYear(int year)
    {
        if (year < 0) return false;

        bool deelDoor400 = ((year % 400) == 0);
        bool deelDoor100 = ((year % 100) == 0);
        bool deelDoor4 = ((year % 4) == 0);

        return (deelDoor400 || (deelDoor4 && !deelDoor100));
    }
}
```

The Main-method creates a (Program) object (via new) and will only call method Start.

All other methods (here 'Start' and 'IsLeapYear') don't have to be static anymore!

Enumerations

Enumerations

Take a look at the following method:

```
public bool IsWeekend(int dayOfWeek)
{
  return ((dayOfWeek == 0) || (dayOfWeek == 6));
}
```

Some calls:

```
// impossible... but the compiler will not complain!
bool result = IsWeekend(100);

// valid input, but which day do we mean?
result = IsWeekend(1);
```

Enumerations

- An enumeration is a set of (related) <u>named</u> constants, e.g.:
  - enum Gender: Male, Female
  - enum StudentType: FulltimeStudent, ParttimeStudent

- Enumerations are "strongly typed constants"

  (to convert an enumeration value we need to explicitely use a conversion; this prevent us from making programming errors!)

Enumerations

```
public enum DayOfWeek
{
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}
```

```
class Program
{
  static void Main(string[] args)
  {
    bool result1 = IsWeekend(DayOfWeek.Monday); // valid input
    bool result2 = IsWeekend(1);                // invalid input
  }

  public static bool IsWeekend(DayOfWeek day)
  {
    return ((day == DayOfWeek.Saturday) || (day == DayOfWeek.Sunday));
  }
}
```

(more readable AND compiler will complain when not used correctly)

Enumerations

All options of an enumeration have an integer value, starting at 0.

This can be changed:

```
public enum DayOfWeek
{
  Sunday, Monday, Tuesday, Wednesday = 8, Thursday, Friday, Saturday
}
```

Now Sunday still has the value 0, but Wedsnesday=8, Thursday=9, etc.

```
public enum DayOfWeek
{
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}
```

```
static void Main(string[] args)
{
  // loop through all days
  for (DayOfWeek d = DayOfWeek.Sunday; d<= DayOfWeek.Saturday; d++)
  {
    Console.WriteLine(d);
  }

  // read day number (e.g. "1")
  string s = Console.ReadLine();
  DayOfWeek n = (DayOfWeek)int.Parse(s);

  switch (n)
  {
    case DayOfWeek.Friday:
      Console.WriteLine("Almost weekend!");
      break;
    case DayOfWeek.Monday:
      Console.WriteLine("Back to work again...");
      break;
  }
}
```

(program output)
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
1 *(input of user)*
Back to work again...

Enumerations – more examples

LightState { On, Off }

TrafficLightState { Red, Orange, Green, Error }

MachineState { Operational, Maintenance, Broken, … }

PoliticalParty { Democratic, Republican, Independent }

Orientation { Portrait, Landscape }

…

Enumerations – read/display methods

- Create a method for <u>reading</u> an enumeration *(including check for valid input)*

```
DayOfWeek day = ReadDayOfWeek("enter a day: ");
```

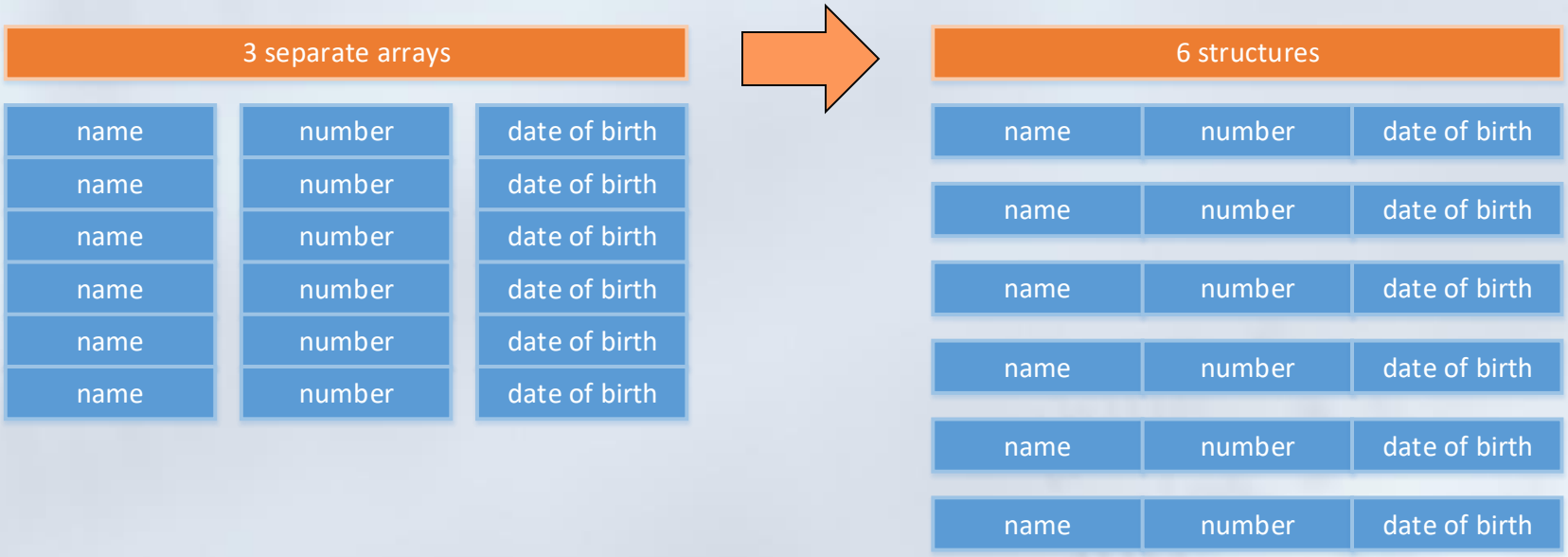
- Also create a method for <u>displaying</u> an enumeration *(screen)*

```
DisplayDayOfWeek(day);
```

Structures

Structures

- Group elements that belong to each other…



3 arrays

1 array with structs

So far we've used…

```
Console.Write("Enter number of students: ");
int nrOfStudents = Int32.Parse(Console.ReadLine());

// create arrays
string[] students = new string[nrOfStudents];
float[] grades = new float[nrOfStudents];

// read names
for (int i = 0; i < nrOfStudents; i++)
{
  Console.Write("Enter name of student {0}: ", i + 1);
  students[i] = Console.ReadLine();
}

// read grades
for (int i = 0; i < nrOfStudents; i++)
{
  Console.Write("Enter grade of {0}: ", students[i]);
  grades[i] = float.Parse(Console.ReadLine());
}

// print students and their grades
PrintStudents(students, grades);
```

The names and grades of the students are stored in separate arrays. We can not store this in 1 single array.

What if we also need to store (for every student) a number, date of birth and city? arrays, arrays, arrays…

If we want to print the students via a method, we need to pass all arrays.

Struct (each in a separate file)

- With a 'struct' we can group all information of a student (name, grade, …)

```
struct Student
{
  public string Name;
  public string Number;
  public DateTime DateOfBirth;
  public float[] Grades;
  public string City;
}
```

```
static void Main(string[] args)
{
  Student student1;

  student1.Name = "Peter Moore";
  student1.Number = "13092";
  student1.DateOfBirth = new DateTime(1971, 6, 24);
  student1.Grades = new float[10];
  student1.City = "Haarlem";

  Console.WriteLine("Name: {0}", student1.Name);
}
```

Revisited…

```
Console.Write("Enter number of students: ");
int nrOfStudents = Int32.Parse(Console.ReadLine());

// create array of students
Student[] students = new Student[nrOfStudents];

// read names
for (int i = 0; i < nrOfStudents; i++)
{
  Console.Write("Enter name of student {0}: ", i + 1);
  students[i].Name = Console.ReadLine();
}

// read grades
for (int i = 0; i < nrOfStudents; i++)
{
  Console.Write("Enter grade of {0}: ", students[i].Name);
  students[i].Grade = float.Parse(Console.ReadLine());
}

// print students (names, grades, ...)
PrintStudents(students);
```

We only need one array now, containing items of (a new created) datatype 'Student'.

If we want to print the students via a method, we need to pass only one array, even if we add more fields to the struct Student.

Structs – read/display methods

- Create a method for <u>reading</u> a struct
  (including check for valid input)

```
Student user = ReadStudent("Enter your data: ");
```

- Also create a method for <u>displaying</u> a struct
  (screen)

```
DisplayStudent(user);
```

Classes

struct vs class

- Everything that can be done with a struct (*grouping multiple variables*) can also be done with a class …

```
struct Student
{
    public string Name;
    public int Age;
}
```

```
class Student
{
    public string Name;
    public int Age;
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Program myProgram = new Program();
        myProgram.Start();
    }

    void Start()
    {
        Student student;
        student = new Student();
        student.Name = "Peter";
        student.Age = 19;

        Console.WriteLine("name: {0}", student.Name);
        Console.WriteLine("age: {0}", student.Age);
    }
}
```

If Student is a class, then we need to use new !

struct vs class

- … but normally a class is used (especially in object-oriented applications)

- Importance difference: a struct is a 'value type' and a class is a 'reference type'

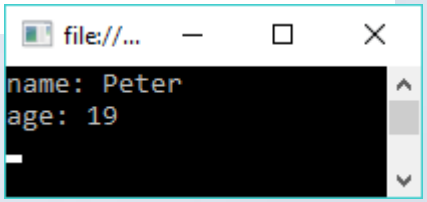
  → a value type (variable) is stored on the stack, a reference type (variable) is stored on the heap

  → if a value type is copied, all its fields are copied; if a reference type is copied, only the reference (pointer) is copied!

struct vs class

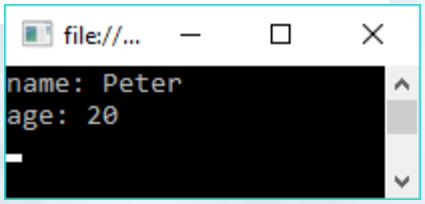
```
class Program
{
    static void Main(string[] args)
    {
        Program myProgram = new Program();
        myProgram.Start();
    }

    void Start()
    {
        Student student1;
        student1 = new Student();
        student1.Name = "Peter";
        student1.Age = 19;

        Student student2;
        student2 = student1;    // copy student 1
        student2.Age = 20;

        Console.WriteLine("name: {0}", student1.Name);
        Console.WriteLine("age: {0}", student1.Age);
    }
}
```

```
struct Student
{
    public string Name;
    public int Age;
}
```



student1 is <u>not modified</u>!

```
class Student
{
    public string Name;
    public int Age;
}
```



student1 is <u>modified</u>!

Homework

- Read paragraphs ‘Yellow Book’
  (references can be found on Moodle)

- Assignments week 1
  (can be found on Moodle)