Plan

- automatic properties
- implicit typing
- simplified initialization
- anonymous types
- lambda expressions
- extension methods
- linq (to xml)

automatic properties

- the pattern

```
private int cislo;
public int Cislo
{
 get { return cislo; }
 set { cislo = value; }
}
```

- can be replaced by simpler

```
public int Cislo; { get; set; }
```

automatic properties

- disatvantages of the former
  - code is longer and less readable
  - temptation to use a public field
  - the code with same purpose is in two places
- properties of the latter
  - both getter and setter can have different access
  - the generated backing field has no default value!

implicit typing

- if you use the `var` keyword the compiler will deduce the type of a variable
- the variable is still strongly typed (it is no VARIANT)
- the conditions
  - the variable is local
  - it is a single variable declared and initialized
  - the value is not null, method group or anonymous function
  - the type is known in compile time
  - any expression can be used

implicit typing

- for
  - less to write
  - less to read
  - you do not have to write the same thing twice like
    ```
    Dictionary<string, KeyValuePair<int, string>> dict
    =
     new Dictionary<string, KeyValuePair<int,
    string>>();
    ```
- against
  - code is less readable (the case of constants)
  - "it doesn't feel right"

implicit typing

- recomendations
- don't do it against others in your team
- when in doubt allways type explicitly
- using and foreach
- the code is more declarative

simplified initialization

- arrays can be initialized using the block syntax

```
int [] intar =
new int []  1, 2, 3;
```

- the idea is to use the same pattern to initialize other types
- the public properties are used as named elements
- it is possible to initialize embedded objects

simplified initialization of collections

- exactly the same syntax as for arrays
- the type must implement IEnumerable
- the type must have an Add method(s)
- a suitable overload of an Add method is called
- use it for
  - constant collections
  - unit test initialization
  - encapsulated parameters (both objects and collections)

implictly typed arrays

- the compiler can derive even the type of an array being initialized
- `new int [] { 1, 2, 3 }`
- is replaced by
- `new [] { 1, 2, 3 }`
- all the objects must be convertible to a single common object so
- `new[] {new StringWriter(), new MemoryStream() }` won't compile

anonymous types

- object initializer can be used as an expression (i.e. in place of a variable or constant)
- `new { Name = "Tom"; Age = 4; }`
- the class is being generated with
  - constructor with all the initialization values
  - public read-only properties
  - private read-only backing fields
  - overrides for Equals, GetHashCode and ToString
- projection initializers

lambda expressions

- a very handy way of representing delegates
- `Func<T>` delegate type
- any delegate syntax can be used to initialize `Func<T>`
- `(T x, S y) => b(x, y)`
- `(x, y) => b(x,y)`
- higher order functions can be created
- parentheses can be omitted for only one parameter

lambda expressions

- lambda expressions can be compiled form expression trees
- expression tree is a tree of objects representing a piece of code - an expression
- abstract class Expression has two important members
  - Type - indicates a type of the resulting expression
  - NodeType - indicates the kind of expression as indicated by the ExpressionType enum (e.g. Add, Multiply, Invoke)
- expressions are created using static methods of the Expression class

lambda expressions

- expression trees can be turned into delegates using the Compile method
- the generic Expression class is the type of the statically typed expressions
- lambda expressions can be converted to compiled expression trees - not all of them obviously
- the main purpose of the expression trees is to abstract the execution model from the desired logic
- we can define an expression in c#, then convert the tree to a native language of a particular platform (e.g. SQL)
- we still get some compiler checks (not the case when we use strings)

extension methods

- a way to extend the behavior of a group of objects
- they are methods :
  - of a non nested, non generic static class
  - at least one parameter
  - first parameter has only one modifies this
- you call them just like any other method of the corresponding type
- the best overload is called (of the most specific type)
- you can call e.m. even on a null reference (IsEmptyOrNull on string)

extension methods

- put extension methods in their own namespace
- it should allways be applicable to all instances of the type being extended
- decide whether it is applicable to the null reference and act accordingly
- put methods extending a particular type in one static class

extension methods

- the main reason for extension methods to exist is that they can be chained
- `x.Where(p1).Where(p2).Reverse()`
- instead of
- `s.Reverse(s.Where(s.Where(x, p1), p2))`

extension methods

- interesting extensions in the `Enumerable` static class
- they have usual static counterparts
- good for testing the behavior of some linq operators

linq to objects

- all this exists because of linq
- linq is a "syntactic sugar" for making the queries to various data sources
- more declarative
- unified
- checked by the compiler
- supported by IntelliSense

linq to objects

- Language INtegrated Query
- a feature consisting of language features and library classes
- all three of the following lines use linq
- `var names = Enumerable.Select(people, p => p.Name);`
- `var names = people.Select(p => p.Name);`
- `var names = from p in people select p.Name;`

linq - fundamental concepts

- sequences
  - you have access just to the current element
  - you do not know how many are yet to come
  - `IEnumerable<T>` is a sequence not a collection
- deferred execution - streaming vs. buffering
- query operators

linq - fundamental concepts

- the query expression just transforms a sequence in other sequences
- it is translated in the early phase of the compilation into a normal C# code
- it consists of
  - context words
  - range variables
  - expressions

linq - basic expressions

- `from x in y select x`
- `x` is a *range variable*
- `y` is a *source*
  - its scope is the expression
  - used for passing the data along the expression
  - it's basically just one element of the sequence
  - they are just variables for translation of expressions to lambda expressions

translation

- `from x in y select b(x)`
- translates into
- `y.Select(x => b(x))`
- the translation is textual - no interfaces needed, no particular types (not evem Enumerable)
- any variables in the lambda expressions are captured!

translation

- all the types are usually inferred (when using a generic collection as a source)
- if not use `from string x in list ...` which translates to
- `list.Cast<string()> ...`
- `OfType<T>` method can be used to filter out inappropriate values

projection

- the Select method singnature is
- `IEnumerable<TR> Select(Func<T, TR> selector)`
- the select is erased when trivial using other operators but
- `from x in y select x` is not the same as `y`!
- the projection expression uses the anonymous types extensively

Filtering

- the `where` keyword
- parameter is a predicate
- more filters are evaluated from left to right and all of them must be true
- group together the filters that are logically close

ordering

- `order by x, y [descending]`
- any number of expressions
- translates into `OrderBy` and `ThenBy[Descending]` methods
- return `IOrderedEnumerable<T>`
- last order by "wins"
- a buffering operation

let

- results of some operation can be saved using `let` keyword
- introduces a new range variable
- compiler uses so called transparent identifiers
- use when you need to precompute a value before executing the query

inner joins

- {LE} `join RRV in RS on LKS equals RKS`
- left sequence is streamed, right is buffered
- the ordering is $(l_1, r_{11}), (l_1, r_{12}), \ldots, (l_2, r_{21}), \ldots$
- in each of the key selectors only the corresponding range variable is in scope

group joins

- `into` keyword
- to each element of the left sequence a sequence of corresponding elements is selected
- the result is $(l_1, (r_{11}, r_{12}, \dots)), (l_2, (r_{21}, r_{22}, \dots), \dots$
- can be used to implement SQL left outer joins — the corresponding sequence can be `null`
- the resulting sequence is in bijective correspondece with the left one

cross joins

- just use two `from`
- the second from's source can be defined in terms of the first's range variable
- the result is $(l_1, (r_{11}, r_{12}, \dots)), (l_2, (r_{21}, r_{22}, \dots), \dots$
- can be used to implement SQL left outer joins — the corresponding sequence can be `null`
- the resulting sequence is in bijective correspondece with the left one

grouping

- `group by b(x)`, `x` is the range variable
- the result is $((b(x_1), (x_{11}, x_{12}, \dots)), (b(x_2), (x_{21}, x_{22}, \dots)), \dots)$
- every elements is the value of the grouping key plus the sequence of the corresponding elements
- the sequence of keys is streamed
- the resulting sequence is in bijective correspondece with the left one

query continuation

- `fst-query into ident snd-query(ident)`
- is the same as
- `from x in (fst-query) snd-query`
- into can be inserted after any select or group by

creating own linq providers

- `IQueryable` interface
- inherits from `IEnumerable`
- there is a `Queryable` class
- it uses expression trees (`Enumerable` uses delegates)

creating own linq providers

- `IQueryable` interface
- contains three properties — `Expression`, `Provider`, `ElementType`
- the constructor creates an initial expression
- to create a new query
  1. ask the existing query for its expression
  2. modify it to a new expression
  3. ask the existing query for its provider
  4. call `CreateQuery` on the provider using the new expression
- to set things in motion — call `GetEnumerator` on the query or `Execute` on the provider