The F# Programming Language

Mónika Mészáros
E-mail: bonnie@inf.elte.hu

Department of Programming Languages and Compilers
ELTE

May 22, 2009

Outline of the presentation

1. Introduction to F#
2. Characteristics of F#
3. Standard Developer Tools
4. Language Basics
5. Lists
6. Higher-Order Functions
7. Tuples
8. Records

Outline of the presentation

1. Introduction to F#
2. Characteristics of F#
3. Standard Developer Tools
4. Language Basics
5. Lists
6. Higher-Order Functions
7. Tuples
8. Records

Outline of the presentation

1. Introduction to F#
2. Characteristics of F#
3. Standard Developer Tools
4. Language Basics
5. Lists
6. Higher-Order Functions
7. Tuples
8. Records

Outline of the presentation

1. Introduction to F#
2. Characteristics of F#
3. Standard Developer Tools
4. Language Basics
5. Lists
6. Higher-Order Functions
7. Tuples
8. Records

Outline of the presentation

1. Introduction to F#
2. Characteristics of F#
3. Standard Developer Tools
4. Language Basics
5. Lists
6. Higher-Order Functions
7. Tuples
8. Records

Outline of the presentation

1. Introduction to F#
2. Characteristics of F#
3. Standard Developer Tools
4. Language Basics
5. Lists
6. Higher-Order Functions
7. Tuples
8. Records

Outline of the presentation

1. Introduction to F#
2. Characteristics of F#
3. Standard Developer Tools
4. Language Basics
5. Lists
6. Higher-Order Functions
7. Tuples
8. Records

Outline of the presentation

1. Introduction to F#
2. Characteristics of F#
3. Standard Developer Tools
4. Language Basics
5. Lists
6. Higher-Order Functions
7. Tuples
8. Records

Introduction to F#

- **F#** is a
  - Functional
  - Imperative and
  - Object-oriented

  programming language.
- F# is "`OCaml` for `.NET`"

Introduction to F#

- **F#** is a
  - Functional
  - Imperative and
  - Object-oriented

  programming language.
- F# is "`OCaml` for `.NET`"

Notable features

- Strongly typed
- Type inference
- Performance profile like that of C#
- Easy access to entire range of powerful .NET libraries
- Speed of native code execution on the concurrent, portable, and distributed .NET Framework
- Option of a top-rate Visual Studio integration
- Cross-compiling core shared with the OCaml language

Notable features

- Strongly typed
- Type inference
- Performance profile like that of C#
- Easy access to entire range of powerful .NET libraries
- Speed of native code execution on the concurrent, portable, and distributed .NET Framework
- Option of a top-rate Visual Studio integration
- Cross-compiling core shared with the OCaml language

Notable features

- Strongly typed
- Type inference
- Performance profile like that of C#
- Easy access to entire range of powerful .NET libraries
- Speed of native code execution on the concurrent, portable, and distributed .NET Framework
- Option of a top-rate Visual Studio integration
- Cross-compiling core shared with the OCaml language

Notable features

- Strongly typed
- Type inference
- Performance profile like that of `C#`
- Easy access to entire range of powerful `.NET` libraries
- Speed of native code execution on the concurrent, portable, and distributed `.NET Framework`
- Option of a top-rate `Visual Studio` integration
- Cross-compiling core shared with the `OCaml` language

Notable features

- Strongly typed
- Type inference
- Performance profile like that of `C#`
- Easy access to entire range of powerful `.NET` libraries
- Speed of native code execution on the concurrent, portable, and distributed `.NET Framework`
- Option of a top-rate `Visual Studio` integration
- Cross-compiling core shared with the `OCaml` language

Notable features

- Strongly typed
- Type inference
- Performance profile like that of C#
- Easy access to entire range of powerful .NET libraries
- Speed of native code execution on the concurrent, portable, and distributed .NET Framework
- Option of a top-rate Visual Studio integration
- Cross-compiling core shared with the OCaml language

Notable features

- Strongly typed
- Type inference
- Performance profile like that of C#
- Easy access to entire range of powerful .NET libraries
- Speed of native code execution on the concurrent, portable, and distributed .NET Framework
- Option of a top-rate Visual Studio integration
- Cross-compiling core shared with the OCaml language

Other features

- An F# program consists of type, class and function definitions and expression**s**
- Computation means evaluation of **all** the expressions one by one
- F# uses **strict** evaluation
- F# is **not** pure (programs may contain side-effects)
- Off-side rule only in "**lightweight**" syntax, which can be turned on by `#light` ("hash-light") compiler directive (it is recommended to keep `#light` on)

Other features

- An F# program consists of type, class and function definitions and expression**s**
- Computation means evaluation of **all** the expressions one by one
- F# uses **strict** evaluation
- F# is **not** pure (programs may contain side-effects)
- Off-side rule only in "**lightweight**" syntax, which can be turned on by `#light` ("hash-light") compiler directive (it is recommended to keep `#light` on)

Other features

- An F# program consists of type, class and function definitions and expression**s**
- Computation means evaluation of **all** the expressions one by one
- F# uses **strict** evaluation
- F# is **not** pure (programs may contain side-effects)
- Off-side rule only in "**lightweight**" syntax, which can be turned on by `#light` ("hash-light") compiler directive (it is recommended to keep `#light` on)

Other features

- An F# program consists of type, class and function definitions and expression**s**
- Computation means evaluation of **all** the expressions one by one
- F# uses **strict** evaluation
- F# is **not** pure (programs may contain side-effects)
- Off-side rule only in "**lightweight**" syntax, which can be turned on by `#light` ("hash-light") compiler directive (it is recommended to keep `#light` on)

Other features

- An F# program consists of type, class and function definitions and expression**s**
- Computation means evaluation of **all** the expressions one by one
- F# uses **strict** evaluation
- F# is **not** pure (programs may contain side-effects)
- Off-side rule only in "**lightweight**" syntax, which can be turned on by `#light` ("hash-light") compiler directive (it is recommended to keep `#light` on)

Standard Developer Tools

Basically, standard developer environments for F# are as follows:

- F# Interactive
- Microsoft Visual Studio integration via an Add-In

Standard Developer Tools

Basically, standard developer environments for F# are as follows:

- F# Interactive
- Microsoft Visual Studio integration via an Add-In

F# Interactive: Read-Eval-Print

In spirit of LISP or Haskell functional programming languages, F# also offers an opportunity for interactive software development.
This is called **F# Interactive** or **FSI** for short.

- Console application
- Every feature is available
- Ideal for brainstorming
- Structure and behavior of programs can be analyzed
- Expressions must be terminated with "; ;"
- Runs over Mono

F# Interactive: Read-Eval-Print

In spirit of LISP or Haskell functional programming languages, F# also offers an opportunity for interactive software development.
This is called **F# Interactive** or **FSI** for short.

- Console application
- Every feature is available
- Ideal for brainstorming
- Structure and behavior of programs can be analyzed
- Expressions must be terminated with ";;"
- Runs over Mono

F# Interactive: Read-Eval-Print

In spirit of LISP or Haskell functional programming languages, F# also offers an opportunity for interactive software development.
This is called **F# Interactive** or **FSI** for short.

- Console application
- Every feature is available
- Ideal for brainstorming
- Structure and behavior of programs can be analyzed
- Expressions must be terminated with "; ;"
- Runs over Mono

F# Interactive: Read-Eval-Print

In spirit of LISP or Haskell functional programming languages, F# also offers an opportunity for interactive software development.
This is called **F# Interactive** or **FSI** for short.

- Console application
- Every feature is available
- Ideal for brainstorming
- Structure and behavior of programs can be analyzed
- Expressions must be terminated with ";;"
- Runs over Mono

F# Interactive: Read-Eval-Print

In spirit of LISP or Haskell functional programming languages, F# also offers an opportunity for interactive software development.
This is called **F# Interactive** or **FSI** for short.

- Console application
- Every feature is available
- Ideal for brainstorming
- Structure and behavior of programs can be analyzed
- Expressions must be terminated with ";;"
- Runs over Mono

F# Interactive: Read-Eval-Print

In spirit of LISP or Haskell functional programming languages, F# also offers an opportunity for interactive software development.
This is called **F# Interactive** or **FSI** for short.

- Console application
- Every feature is available
- Ideal for brainstorming
- Structure and behavior of programs can be analyzed
- Expressions must be terminated with "`;;`"
- Runs over Mono

F# Interactive: Read-Eval-Print

In spirit of LISP or Haskell functional programming languages, F# also offers an opportunity for interactive software development.
This is called **F# Interactive** or **FSI** for short.

- Console application
- Every feature is available
- Ideal for brainstorming
- Structure and behavior of programs can be analyzed
- Expressions must be terminated with "`;;`"
- Runs over Mono

Some Easy Expressions

- Launch the F# Interactive
- Try the following expressions:

```
> let square x = x * x;;
> square 4;;
> let numbers = [1 .. 10];;
> let squares = List.map square numbers;;
> squares;;
> List.map square numbers;;
```

Some Easy Expressions

- Launch the F# Interactive
- Try the following expressions:

```
> let square x = x * x;;
> square 4;;
> let numbers = [1 ..  10];;
> let squares = List.map square numbers;;
> squares;;
> List.map square numbers;;
```

Some Easy Expressions

- Launch the F# Interactive
- Try the following expressions:

```
> let square x = x * x;;
> square 4;;
> let numbers = [1 .. 10];;
> let squares = List.map square numbers;;
> squares;;
> List.map square numbers;;
```

Some Easy Expressions

- Launch the F# Interactive
- Try the following expressions:

```
> let square x = x * x;;
> square 4;;
> let numbers = [1 ..  10];;
> let squares = List.map square numbers;;
> squares;;
> List.map square numbers;;
```

Some Easy Expressions

- Launch the F# Interactive
- Try the following expressions:

```
> let square x = x * x;;
> square 4;;
> let numbers = [1 ..  10];;
> let squares = List.map square numbers;;
> squares;;
> List.map square numbers;;
```

Some Easy Expressions

- Launch the F# Interactive
- Try the following expressions:

```
> let square x = x * x;;
> square 4;;
> let numbers = [1 ..  10];;
> let squares = List.map square numbers;;
> squares;;
> List.map square numbers;;
```

Some Easy Expressions

- Launch the F# Interactive
- Try the following expressions:

```
> let square x = x * x;;
> square 4;;
> let numbers = [1 ..  10];;
> let squares = List.map square numbers;;
> squares;;
> List.map square numbers;;
```

Visual Development: Microsoft Visual F#

Recent F# distributions (1.9.6.2 CTP, September 2008) include an Add-In for the **Microsoft Visual Studio 2008** Development Environment.

It offers the following additional advantages:

- Syntax highlighting
- Showing derived types in tooltips
- Support for debugging
- Every other service of the Visual Studio Ecosystem is available
- Microsoft Visual Studio 2010 includes complete support

Visual Development: Microsoft Visual F#

Recent F# distributions (1.9.6.2 CTP, September 2008) include an Add-In for the **Microsoft Visual Studio 2008** Development Environment.
It offers the following additional advantages:

- Syntax highlighting
- Showing derived types in tooltips
- Support for debugging
- Every other service of the Visual Studio Ecosystem is available
- Microsoft Visual Studio 2010 includes complete support

Visual Development: Microsoft Visual F#

Recent F# distributions (1.9.6.2 CTP, September 2008) include an Add-In for the **Microsoft Visual Studio 2008** Development Environment.

It offers the following additional advantages:

- Syntax highlighting
- Showing derived types in tooltips
- Support for debugging
- Every other service of the Visual Studio Ecosystem is available
- Microsoft Visual Studio 2010 includes complete support

Visual Development: Microsoft Visual F#

Recent F# distributions (1.9.6.2 CTP, September 2008) include an Add-In for the **Microsoft Visual Studio 2008** Development Environment.
It offers the following additional advantages:

- Syntax highlighting
- Showing derived types in tooltips
- Support for debugging
- Every other service of the Visual Studio Ecosystem is available
- Microsoft Visual Studio 2010 includes complete support

Visual Development: Microsoft Visual F#

Recent F# distributions (1.9.6.2 CTP, September 2008) include an Add-In for the **Microsoft Visual Studio 2008** Development Environment.
It offers the following additional advantages:

- Syntax highlighting
- Showing derived types in tooltips
- Support for debugging
- Every other service of the Visual Studio Ecosystem is available
- Microsoft Visual Studio 2010 includes complete support

Visual Development: Microsoft Visual F#

Recent F# distributions (1.9.6.2 CTP, September 2008) include an Add-In for the **Microsoft Visual Studio 2008** Development Environment.

It offers the following additional advantages:

- Syntax highlighting
- Showing derived types in tooltips
- Support for debugging
- Every other service of the Visual Studio Ecosystem is available
- Microsoft Visual Studio 2010 includes complete support

Visual Development: Microsoft Visual F#

Recent F# distributions (1.9.6.2 CTP, September 2008) include an Add-In for the **Microsoft Visual Studio 2008** Development Environment.
It offers the following additional advantages:

- Syntax highlighting
- Showing derived types in tooltips
- Support for debugging
- Every other service of the Visual Studio Ecosystem is available
- Microsoft Visual Studio 2010 includes complete support

A Very Simple Visual F# Project

- Create a new Project, use the **F# Application** template.
- Insert the following into the empty code editor:

```
#light
let rec factorial n =
  match n with
    | 0            -> 1
    | n when n > 0 -> n * (factorial (n - 1))

printfn "5! = %A" (factorial 5)
System.Console.ReadKey () |> ignore
```

- Press **F5** to build and run the program.
- **Note**: recursive functions denoted by "`let rec`"

A Very Simple Visual F# Project

- Create a new Project, use the **F# Application** template.
- Insert the following into the empty code editor:

```
#light
let rec factorial n =
  match n with
    | 0            -> 1
    | n when n > 0 -> n * (factorial (n - 1))

printfn "5! = %A" (factorial 5)
System.Console.ReadKey () |> ignore
```

- Press **F5** to build and run the program.
- **Note**: recursive functions denoted by "`let rec`"

A Very Simple Visual F# Project

- Create a new Project, use the **F# Application** template.
- Insert the following into the empty code editor:

```
#light
let rec factorial n =
  match n with
    | 0            -> 1
    | n when n > 0 -> n * (factorial (n - 1))

printfn "5! = %A" (factorial 5)
System.Console.ReadKey () |> ignore
```

- Press **F5** to build and run the program.
- **Note**: recursive functions denoted by "`let rec`"

A Very Simple Visual F# Project

- Create a new Project, use the **F# Application** template.
- Insert the following into the empty code editor:

```
#light
let rec factorial n =
  match n with
    | 0            -> 1
    | n when n > 0 -> n * (factorial (n - 1))

printfn "5! = %A" (factorial 5)
System.Console.ReadKey () |> ignore
```

- Press **F5** to build and run the program.
- **Note**: recursive functions denoted by "**let rec**"

Type Inference

- The F# compiler figures the type information out for the programmer.
- In case of aritmetic operators, F# defaults to `int`, a signed 32-bit integer.

```
> let square x = x * x;;
val square : int -> int
```

- It is possible to add "type annotations" for function parameters and return values.

```
> let concat (x: string) y = x + y;;
val concat : string -> string -> string
```

Type Inference

- The F# compiler figures the type information out for the programmer.
- In case of aritmetic operators, F# defaults to `int`, a signed 32-bit integer.

```
> let square x = x * x;;
val square : int -> int
```

- It is possible to add "type annotations" for function parameters and return values.

```
> let concat (x: string) y = x + y;;
val concat : string -> string -> string
```

Type Inference

- The F# compiler figures the type information out for the programmer.
- In case of aritmetic operators, F# defaults to `int`, a signed 32-bit integer.

```
> let square x = x * x;;
val square : int -> int
```

- It is possible to add "type annotations" for function parameters and return values.

```
> let concat (x: string) y = x + y;;
val concat : string -> string -> string
```

Type Inference

- The F# compiler figures the type information out for the programmer.
- In case of aritmetic operators, F# defaults to `int`, a signed 32-bit integer.

```
> let square x = x * x;;
val square : int -> int
```

- It is possible to add "type annotations" for function parameters and return values.

```
> let concat (x: string) y = x + y;;
val concat : string -> string -> string
```

Pattern Matching

- Wildcard "_" matches anything.
- Arbitrary expression can be executed to determine if the pattern is matched.
- Dynamic type tests are possible too.
  Syntax:

```
match <expression> with
  | <pattern1> -> <expression1>
  | <pattern2> -> <expression2>
  ...
```

- Pattern Guards (when <logical expression> - between <pattern> and "->")

Pattern Matching

- Wildcard "_" matches anything.
- Arbitrary expression can be executed to determine if the pattern is matched.
- Dynamic type tests are possible too.

  Syntax:

```
match <expression> with
  | <pattern1> -> <expression1>
  | <pattern2> -> <expression2>
  ...
```

- Pattern Guards (when <logical expression> - between <pattern> and "->")

Pattern Matching

- Wildcard "_" matches anything.
- Arbitrary expression can be executed to determine if the pattern is matched.
- Dynamic type tests are possible too.

  Syntax:

```
match <expression> with
  | <pattern1> -> <expression1>
  | <pattern2> -> <expression2>
  ...
```

- Pattern Guards (`when <logical expression>` - between <pattern> and "`->`")

Pattern Matching

- Wildcard "_" matches anything.
- Arbitrary expression can be executed to determine if the pattern is matched.
- Dynamic type tests are possible too.
  Syntax:

```
match <expression> with
  | <pattern1> -> <expression1>
  | <pattern2> -> <expression2>
  ...
```

- Pattern Guards (when <logical expression> - between <pattern> and "->")

Pattern Matching

- Wildcard "_" matches anything.
- Arbitrary expression can be executed to determine if the pattern is matched.
- Dynamic type tests are possible too.
  Syntax:

```
match <expression> with
  | <pattern1> -> <expression1>
  | <pattern2> -> <expression2>
  ...
```

- Pattern Guards (when <logical expression>- between <pattern> and "->")

Interoperability with .NET

- F# is built on top of .NET, any .NET library can be called:

```
System.Console.ReadKey ()
```

- .NET namespaces can be opened and their types are brought into scope:

```
open System

Console.ReadKey ()
```

Interoperability with .NET

- F# is built on top of .NET, any .NET library can be called:

```
System.Console.ReadKey ()
```

- .NET namespaces can be opened and their types are brought into scope:

```
open System

Console.ReadKey ()
```

Interoperability with .NET

- F# is built on top of .NET, any .NET library can be called:

```
System.Console.ReadKey ()
```

- .NET namespaces can be opened and their types are brought into scope:

```
open System

Console.ReadKey ()
```

Exercises

- 1. Write a function which determines whether the argument is odd or not
  Hint: modulo function: %, logical values: `true`, `false`
  Signature: `odd : int -> bool`
- 2. Write a function which computes x^y
  Rules: $n^0 = 1$, $n^m = n * n^{m-1}$
  Signature: `power : int -> int -> int`
- Test the functions!

`http://people.inf.elte.hu/bonnie/cefp/fsharp.pdf`

Exercises

- 1. Write a function which determines whether the argument is odd or not
  Hint: modulo function: %, logical values: `true`, `false`
  Signature: `odd : int -> bool`
- 2. Write a function which computes x^y
  Rules: $n^0 = 1$, $n^m = n * n^{m-1}$
  Signature: `power : int -> int -> int`
- Test the functions!

`http://people.inf.elte.hu/bonnie/cefp/fsharp.pdf`

Exercises

- 1. Write a function which determines whether the argument is odd or not
  Hint: modulo function: %, logical values: `true`, `false`
  Signature: `odd :  int -> bool`
- 2. Write a function which computes x^y
  Rules: $n^0 = 1$, $n^m = n * n^{m-1}$
  Signature: `power :  int -> int -> int`
- Test the functions!

Exercises

- 1. Write a function which determines whether the argument is odd or not
  Hint: modulo function: %, logical values: `true`, `false`
  Signature: `odd : int -> bool`
- 2. Write a function which computes x^y
  Rules: $n^0 = 1$, $n^m = n * n^{m-1}$
  Signature: `power : int -> int -> int`
- Test the functions!

`http://people.inf.elte.hu/bonnie/cefp/fsharp.pdf`

Solutions

```
let odd n =
  match (n%2) with
    | 1 -> true
    | _ -> false
```

```
let rec power n m =
  match m with
    | 0 -> 1
    | m -> n * (power n (m-1))
```

Solutions

```
let odd n =
  match (n%2) with
    | 1 -> true
    | _ -> false
```

```
let rec power n m =
  match m with
    | 0 -> 1
    | m -> n * (power n (m-1))
```

Lists

Quick syntax introduction for using lists

- Define a list:

```
let letters = ['e'; 'i'; 'o'; 'u']
```

- Attach item to front (cons):

```
let cons = 'a' :: letters
```

- Concat two lists:

```
let more_letters = letters @ ['y'; 'z']
```

Lists

Quick syntax introduction for using lists

- Define a list:

```
let letters = [’e’; ’i’; ’o’; ’u’]
```

- Attach item to front (cons):

```
let cons = ’a’ :: letters
```

- Concat two lists:

```
let more_letters = letters @ [’y’; ’z’]
```

Lists

Quick syntax introduction for using lists

- Define a list:

```
let letters = [’e’; ’i’; ’o’; ’u’]
```

- Attach item to front (cons):

```
let cons = ’a’ :: letters
```

- Concat two lists:

```
let more_letters = letters @ [’y’; ’z’]
```

Lists

Quick syntax introduction for using lists

- Define a list:

```
let letters = [’e’; ’i’; ’o’; ’u’]
```

- Attach item to front (`cons`):

```
let cons = ’a’ :: letters
```

- Concat two lists:

```
let more_letters = letters @ [’y’; ’z’]
```

Pattern Matching for Lists

Patterns on lists:

- [] - empty list
- x::xs - list with at least 1 element
- [x] - list with only one element
- etc.

Exercise

- 3. Find the maximum of the list
  Signature: `maximum : 'a list -> 'a`
  Hint: use the `max : 'a -> 'a -> 'a` function!

Solution

```
let rec maximum l =
  match l with
    | [x]   -> x
    | x::xs -> max x (maximum xs)
```

Higher-Order Functions

- There can be also anonymous functions ("**lambda expressions**") defined, like:

```
(fun x -> x % 2 = 0)
```

- **Higher order functions**
  example
  `List.map : ('a -> 'b) -> 'a list -> 'b list`
- Putting them together:

```
> List.map (fun x -> x % 2 = 0) [1 .. 5];;
val it : bool list
= [false; true; false; true; false]
```

Higher-Order Functions

- There can be also anonymous functions ("**lambda expressions**") defined, like:

```
(fun x -> x % 2 = 0)
```

- **Higher order functions**
  example
  `List.map : ('a -> 'b) -> 'a list -> 'b list`
- Putting them together:

```
> List.map (fun x -> x % 2 = 0) [1 .. 5];;
val it : bool list
= [false; true; false; true; false]
```

Higher-Order Functions

- There can be also anonymous functions ("**lambda expressions**") defined, like:

```
(fun x -> x % 2 = 0)
```

- **Higher order functions**
  example
  `List.map : ('a -> 'b) -> 'a list -> 'b list`
- Putting them together:

```
> List.map (fun x -> x % 2 = 0) [1 .. 5];;
val it : bool list
= [false; true; false; true; false]
```

Higher-Order Functions

```
exists :  ('a -> bool) -> 'a list -> bool
```

```
let rec exists p l =
  match l with
    | []               -> false
    | x :: xs when p x -> true
    | x :: xs          -> exists p xs
```

Exercises

- 4. filter: selecting elements satisfying a property
  Signature:
  `filter : ('a -> bool) -> 'a list -> 'a list`
- 5. map: function applied elementwise (length is preserved)
  Signature:
  `map :  ('a -> 'b) -> 'a list -> 'b list`
  Apply a function (first parameter) to all element in the list (second parameter)

Exercises

- 4. filter: selecting elements satisfying a property
  Signature:
  `filter : ('a -> bool) -> 'a list -> 'a list`
- 5. map: function applied elementwise (length is preserved)
  Signature:
  `map :  ('a -> 'b) -> 'a list -> 'b list`
  Apply a function (first parameter) to all element in the list (second parameter)

Solutions

```
let rec filter p l =
  match l with
    | []               -> []
    | x :: xs when p x -> x :: (filter p xs)
    | x :: xs          -> filter p xs
```

```
let rec map f l =
  match l with
    | []      -> []
    | x :: xs -> (f x) :: (map f xs)
```

Solutions

```
let rec filter p l =
  match l with
    | []                -> []
    | x :: xs when p x -> x :: (filter p xs)
    | x :: xs           -> filter p xs
```

```
let rec map f l =
  match l with
    | []       -> []
    | x :: xs -> (f x) :: (map f xs)
```

- A tuple is an ordered collection of values treated like an atomic unit.
- Allows to keep things organized by grouping related values together without introducing a new type.
- Functions can even take tuples as arguments.
- Sometimes tuples are used for communication with .NET libraries.

Using Tuples

- Definition of a tuple:

```
> let tuple = (1, false, "text");;
val tuple : int * bool * string
```

- Function accepting a tuple:

```
> let printBlogInfo (title, url)
  = printfn "%s blog is at ’%s’"
      owner title url;;
val printBlogInfo : string * string -> unit
```

Using Tuples

- Definition of a tuple:

```
> let tuple = (1, false, "text");;
val tuple : int * bool * string
```

- Function accepting a tuple:

```
> let printBlogInfo (title, url)
  = printfn "%s blog is at ’%s’"
      owner title url;;
val printBlogInfo : string * string -> unit
```

- Records are for declaring a type with public properties.
- Through type inference, the compiler will figure out the type of the record by setting its values.
- Records can be "cloned".

Basic Record Usage

- Definition of a record type:

```
type Person =
  { Name: string
  ; DateOfBirth: System.DateTime }
```

- Construction of record values by record labels:

```
> { Name = "Bill"
  ; DateOfBirth
    = new System.DateTime(1962,09,02) };;
val it : Person
 = { Name="Bill"; DateOfBirth = 09/02/1962 }
```

Basic Record Usage

- Definition of a record type:

```
type Person =
  { Name: string
  ; DateOfBirth: System.DateTime }
```

- Construction of record values by record labels:

```
> { Name = "Bill"
  ; DateOfBirth
    = new System.DateTime(1962,09,02) };;
val it : Person
 = { Name="Bill"; DateOfBirth = 09/02/1962 }
```

Cloning Records

There is a convenient syntax to clone all the values in the record, creating a new value, with some values replaced.

```
type Point3D = { X: float; Y: float; Z: float }
let p1 = { X = 3.0; Y = 4.0; Z = 5.0 }
```

```
> let p2 = { p1 with Y = 0.0; Z = 0.0 };;
val p2 : Point3D
```

The definition of p2 is identical to this:

```
let p2 = { X = p1.X; Y = 0.0; Z = 0.0 }
```

This expression from does not mutate the values of a record.

Dynamic Type Test via Patterns

```
let getType (x : obj) =
  match x with
    | :? string      -> "x is a string"
    | :? int         -> "x is an int"
    | :? Exception   -> "x is an exception"
```