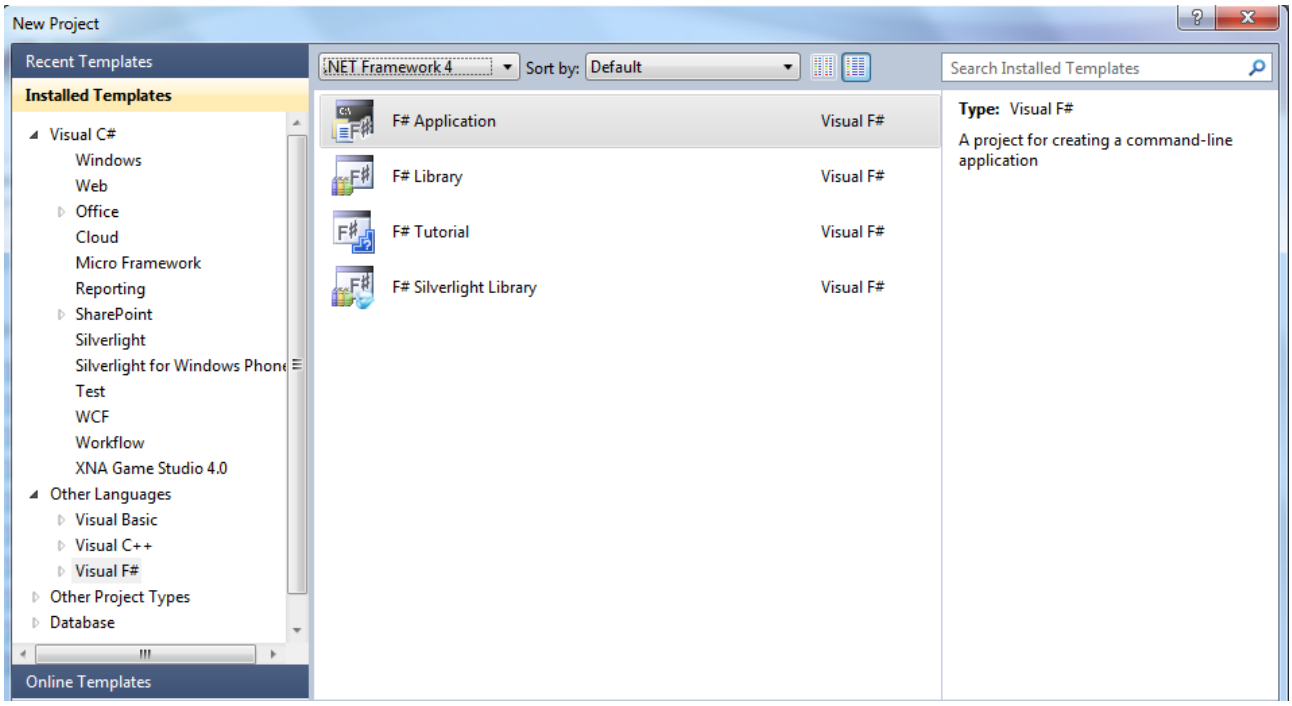
Short introduction to F#



F# and C#

- Typical math related work programmed in F#
  - Used in C# applications
- F# -> CIL

F# - Function Programming

- First intro example

```
open System

let sayHello() = printfn "Hello World!"
let f x = x*x

let main()=
  sayHello()
  printfn "resultatet er %i" (f 22)


main()
```

- Notice:
  - open System
  - The function sayHello()
  - f x function
  - Definitionen of main
  - Declaration of main

- second intro example

```
open System

let sayHello() = printfn "Hello World!"
let f x = x*x

let midterVærdi a b =
      let midt = (b-a)/ 2
      midt + a

let main()=
  sayHello()
  printfn "resultatet er %i" (f 22)
  printfn "MidterVærdi( 5 11) er %i" (midterVærdi 5 11)
  printfn "MidterVærdi( 11 5) er %i" (midterVærdi 11 5)

main()
```

Keywords in F# (source: http://msdn.microsoft.com/en-us/library/dd233249(VS.100).aspx)

The following table shows all F# keywords in alphabetical order, together with brief descriptions and links to relevant topics that contain more information.

Keyword	Link	Description
abstract	Members (F#) Abstract Classes (F#)	Indicates a method that either has no implementation in the type in which it is declared or that is virtual and has a default implementation.

and	let Bindings (F#) Members (F#) Constraints (F#)	Used in mutually recursive bindings, in property declarations, and with multiple constraints on generic parameters.
as	Classes (F#) Patterns (F#)	Used to give the current class object an object name. Also used to give a name to a whole pattern within a pattern match.
assert	Assertions (F#)	Used to verify code during debugging.
base	Classes (F#) Inheritance (F#)	Used as the name of the base class object.
begin	Verbose Syntax (F#)	In verbose syntax, indicates the start of a code block.
class	Classes (F#)	In verbose syntax, indicates the start of a class definition.
default	Members (F#)	Indicates an implementation of an abstract method; used together with an abstract method declaration to create a virtual method.
delegate	Delegates (F#)	Used to declare a delegate.
do	do Bindings (F#) Loops: for...to Expression (F#) Loops: for...in Expression (F#) Loops: while...do Expression (F#)	Used in looping constructs or to execute imperative code.
done	Verbose Syntax (F#)	In verbose syntax, indicates the end of a block of code in a looping expression.
downcast	Casting and Conversions (F#)	Used to convert to a type that is lower in the inheritance chain.
downto	Loops: for...to Expression (F#)	In a for expression, used when counting in reverse.
elif	Conditional Expressions: if... then...else (F#)	Used in conditional branching. A short form of else if.
else	Conditional Expressions: if... then...else (F#)	Used in conditional branching.
end	Structures (F#) Discriminated Unions (F#) Records (F#) Type Extensions (F#)	In type definitions and type extensions, indicates the end of a section of member definitions. In verbose syntax, used to specify the end of a code block that starts with the begin keyword.

	Verbose Syntax (F#)	
exception	Exception Handling (F#) Exception Types (F#)	Used to declare an exception type.
extern	External Functions (F#)	Indicates that a declared program element is defined in another binary or assembly.
false	Primitive Types (F#)	Used as a Boolean literal.
finally	Exceptions: The try...finally Expression (F#)	Used together with try to introduce a block of code that executes regardless of whether an exception occurs.
for	Loops: for...to Expression (F#) Loops: for...in Expression (F#)	Used in looping constructs.
fun	Lambda Expressions: The fun Keyword (F#)	Used in lambda expressions, also known as anonymous functions.
function	Match Expressions (F#) Lambda Expressions: The fun Keyword (F#)	Used as a shorter alternative to the fun keyword and a match expression in a lambda expression that has pattern matching on a single argument.
global	Namespaces (F#)	Used to reference the top-level .NET namespace.
if	Conditional Expressions: if... then...else (F#)	Used in conditional branching constructs.
in	Loops: for...in Expression (F#) Verbose Syntax (F#)	Used for sequence expressions and, in verbose syntax, to separate expressions from bindings.
inherit	Inheritance (F#)	Used to specify a base class or base interface.
inline	Functions (F#) Inline Functions (F#)	Used to indicate a function that should be integrated directly into the caller's code.
interface	Interfaces (F#)	Used to declare and implement interfaces.
internal	Access Control (F#)	Used to specify that a member is visible inside an assembly but not outside it.
lazy	Lazy Computations (F#)	Used to specify a computation that is to be performed only when a result is needed.
let	let Bindings (F#)	Used to associate, or bind, a name to a value or function.
match	Match Expressions (F#)	Used to branch by comparing a value to a pattern.
member	Members (F#)	Used to declare a property or method in an object type.
module	Modules (F#)	Used to associate a name with a group of related types, values, and functions, to logically separate it from other code.
mutable	let Bindings (F#)	Used to declare a variable, that is, a value that can be

		changed.
namespace	Namespaces (F#)	Used to associate a name with a group of related types and modules, to logically separate it from other code.
new	Constructors (F#) Constraints (F#)	Used to declare, define, or invoke a constructor that creates or that can create an object. Also used in generic parameter constraints to indicate that a type must have a certain constructor.
not	Symbol and Operator Reference (F#) Constraints (F#)	Not actually a keyword. However, not struct in combination is used as a generic parameter constraint.
null	Null Values (F#) Constraints (F#)	Indicates the absence of an object. Also used in generic parameter constraints.
of	Discriminated Unions (F#) Delegates (F#) Exception Types (F#)	Used in discriminated unions to indicate the type of categories of values, and in delegate and exception declarations.
open	Import Declarations: The open Keyword (F#)	Used to make the contents of a namespace or module available without qualification.
or	Symbol and Operator Reference (F#) Constraints (F#)	Used with Boolean conditions as a Boolean or operator. Equivalent to ||. Also used in member constraints.
override	Members (F#)	Used to implement a version of an abstract or virtual method that differs from the base version.
private	Access Control (F#)	Restricts access to a member to code in the same type or module.
public	Access Control (F#)	Allows access to a member from outside the type.
rec	Functions (F#)	Used to indicate that a function is recursive.
return	Asynchronous Workflows (F#) Computation Expressions (F#)	Used to indicate a value to provide as the result of a computation expression.
static	Members (F#)	Used to indicate a method or property that can be called without an instance of a type, or a value member that is shared among all instances of a type.
struct	Structures (F#) Constraints (F#)	Used to declare a structure type. Also used in generic parameter constraints. Used for OCaml compatibility in module definitions.

then	Conditional Expressions: if... then...else (F#) Constructors (F#)	Used in conditional expressions. Also used to perform side effects after object construction.
to	Loops: for...to Expression (F#)	Used in for loops to indicate a range.
true	Primitive Types (F#)	Used as a Boolean literal.
try	Exceptions: The try...with Expression (F#) Exceptions: The try...finally Expression (F#)	Used to introduce a block of code that might generate an exception. Used together with with or finally.
type	F# Types Classes (F#) Records (F#) Structures (F#) Enumerations (F#) Discriminated Unions (F#) Type Abbreviations (F#) Units of Measure (F#)	Used to declare a class, record, structure, discriminated union, enumeration type, unit of measure, or type abbreviation.
upcast	Casting and Conversions (F#)	Used to convert to a type that is higher in the inheritance chain.
use	Resource Management: The use Keyword (F#)	Used instead of let for values that require Dispose to be called to free resources.
val	Explicit Fields: The val Keyword (F#) Signatures (F#) Members (F#)	Used in a signature to indicate a value, or in a type to declare a member, in limited situations.
void	Primitive Types (F#)	Indicates the .NET void type. Used when interoperating with other .NET languages.
when	Constraints (F#)	Used for Boolean conditions (when guards) on pattern matches and to introduce a constraint clause for a generic type parameter.
while	Loops: while...do Expression (F#)	Introduces a looping construct.

with	Match Expressions (F#) Object Expressions (F#) Type Extensions (F#) Exceptions: The try...with Expression (F#)	Used together with the match keyword in pattern matching expressions. Also used in object expressions, record copying expressions, and type extensions to introduce member definitions, and to introduce exception handlers.
yield	Sequences (F#)	Used in a sequence expression to produce a value for a sequence.

Types in F# (source: http://msdn.microsoft.com/en-us/library/dd233193.aspx)

Type	Description	Suffix or prefix	Examples
sbyte	signed 8-bit integer	y	86y
byte	unsigned 8-bit natural number	uy	86uy
int16	signed 16-bit integer	s	86s
uint16	unsigned 16-bit natural number	us	86us
nativeint	native pointer as an integer value	n	0x00002D3Fn
unativeint	native pointer as an unsigned natural number	un	0x00002D3Fun
int64	signed 64-bit integer	L	86L
uint64	unsigned 64-bit natural number	UL	86UL
single, float32	32-bit floating point number	F or f	4.14F or 4.14f
		lf	0x00000000lf
float; double	64-bit floating point number	none	4.14 or 2.3E+32 or 2.3e+32
		LF	0x0000000000000000LF
bigint	integer not limited to 64-bit representation	I	99999999999999999999999999I
decimal	fractional number represented as a fixed point or rational number	M or m	0.7833M or 0.7833m
Char	Unicode character	none	'a'
String	Unicode string	none	"text\n" or @"c:\filename"
byte	ASCII character	B	'a'B
byte[]	ASCII string	B	"text"B
String or byte[]	verbatim string	@ prefix	@"\\server\share" (Unicode) @"\\server\share"B (ASCII)

Recursion

- Example

```
open System

let sayHello() = printfn "Hello World!"
let f x = x*x

let midterVærdi a b =
      let midt = (b-a)/ 2
      midt + a

let rec Fibonacci x =
      match x with
      | 1 -> 1
      | 2 -> 1
      | x -> Fibonacci(x-1) + Fibonacci(x-2)



let main()=
  sayHello()
  printfn "resultatet er %i" (f 22)
  printfn "MidterVærdi( 5 11) er %i" (midterVærdi 5 11)
  printfn "MidterVærdi( 11 5) er %i" (midterVærdi 11 5)

  printfn "Fibonacci(6) er %i" (Fibonacci 6)
main()
```

- Fibonacci sequences:
  - Add the two nearest numbers in the row
  - 1,1,2,3,5,8,13....
  - Notice : 1 and 2 are base conditions
  - Notice: The recursive call goes towards base conditions
  - match compares and stop automatically
  - Notice also the keyword **rec**

Anonymous functions (Lambda functions)

- Example

```
open System

let sayHello() = printfn "Hello World!"
let f x = x*x

let midterVærdi a b =
      let midt = (b-a)/ 2
      midt + a

let rec Fibonacci x =
      match x with
      | 1 -> 1
      | 2 -> 1
      | x -> Fibonacci(x-1) + Fibonacci(x-2)

let y = (fun z t -> z+t) 44 48

let main()=
  sayHello()
  printfn "resultatet er %i" (f 22)
  printfn "MidterVærdi( 5 11) er %i" (midterVærdi 5 11)
  printfn "MidterVærdi( 11 5) er %i" (midterVærdi 11 5)

  printfn "Fibonacci(6) er %i" (Fibonacci 6)

  printfn "y er %i" (y)
main()
```

- fun is a more compact keyword than function

Lists

- Several ways to concatenate elements
- lists can also be concatenated

```
open System


let sayHello() = printfn "Hello World!"
let f x = x*x

let midterVærdi a b =
      let midt = (b-a)/ 2
      midt + a

let rec Fibonacci x =
      match x with
      | 1 -> 1
      | 2 -> 1
      | x -> Fibonacci(x-1) + Fibonacci(x-2)

let y = (fun z t -> z+t) 44 48

let scopeConcatList = "1"::"2"::[]
let nemList = ["3";"4"]
let concatList = ["5";"6"]@ ["7";"8"]

let UdskrivListe l =
     Console.WriteLine(l.ToString())

let rec UdskrivListe2 lst =  // benytter rekursion
    match lst with
    | [] -> ()
    | h :: t ->
        printf "%s\n" h
        UdskrivListe2 t

let main()=
  sayHello()
  printfn "resultatet er %i" (f 22)
  printfn "MidterVærdi( 5 11) er %i" (midterVærdi 5 11)
  printfn "MidterVærdi( 11 5) er %i" (midterVærdi 11 5)

  printfn "Fibonacci(6) er %i" (Fibonacci 6)

  printfn "y er %i" (y)

  UdskrivListe scopeConcatList
  UdskrivListe2 nemList
  UdskrivListe concatList

main()
```

- box exists (of course !)
  - let oList =[box 1; box 2.0, box ”string”]

control flow

- Example

```
open System


let sayHello() = printfn "Hello World!"
let f x = x*x

let midterVærdi a b =
      let midt = (b-a)/ 2
      midt + a

let rec Fibonacci x =
      match x with
      | 1 -> 1
      | 2 -> 1
      | x -> Fibonacci(x-1) + Fibonacci(x-2)

let y = (fun z t -> z+t) 44 48

let scopeConcatList = "1"::"2"::[]
let nemList = ["3";"4"]
let concatList = ["5";"6"]@ ["7";"8"]

let UdskrivListe l =
     Console.WriteLine(l.ToString())

let rec UdskrivListe2 lst =  // benytter rekursion
    match lst with
    | [] -> ()
    | h :: t ->
        printf "%s\n" h
        UdskrivListe2 t


let valg p=
  if p > 10 then
    "A"
  else
    "B"
```

```
let main()=
  sayHello()
  printfn "resultatet er %i" (f 22)
  printfn "MidterVærdi( 5 11) er %i" (midterVærdi 5 11)
  printfn "MidterVærdi( 11 5) er %i" (midterVærdi 11 5)

  printfn "Fibonacci(6) er %i" (Fibonacci 6)

  printfn "y er %i" (y)

  UdskrivListe scopeConcatList
  UdskrivListe2 nemList
  UdskrivListe concatList

  printfn "if %s" (valg 9)

main()
```