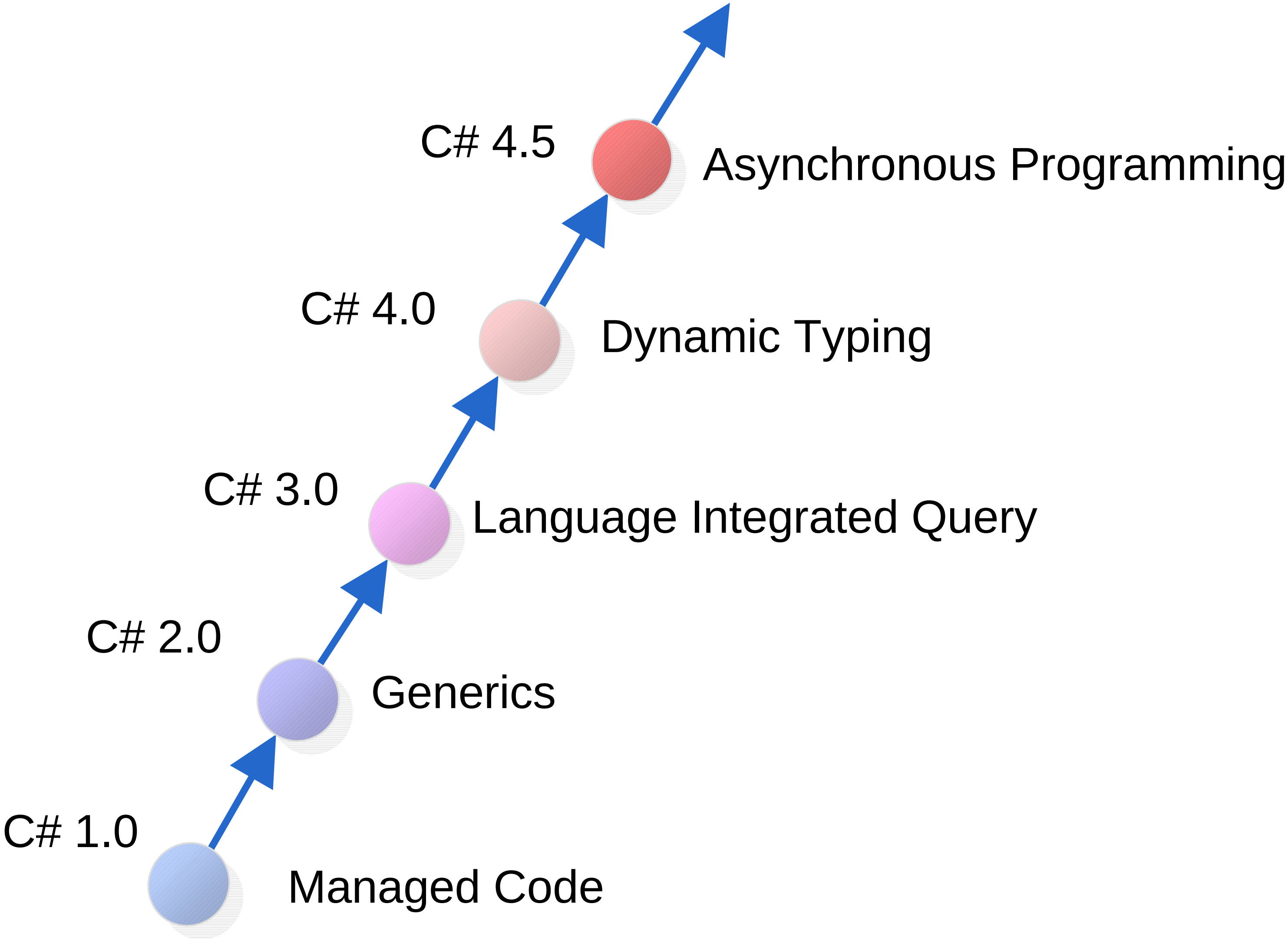
New Features since 2.0

- Automatic Properties
- Object and Collection Initializers
- Anonymous Types
- Partial Methods
- Extension Methods
- Lambda Expressions
- LINQ
- Dynamic Typing
- Optional and Named Parameters
- Safe Co- and Contra-Variance for Generic Types
- await and async

C# Evolution


C# 4.5
Asynchronous Programming
C# 4.0
Dynamic Typing
C# 3.0
Language Integrated Query
C# 2.0
Generics
C# 1.0
Managed Code

Automatic Properties

Automatic Properties

The following pattern is very common

```
private string name;
public string Name {
  get { return name; }
  set { name = value; }
}
```

Instead of that, one can simply write

```
public string Name { get; set; }
```

must be `get;` and `set;`

compiler generates the private field and the get/set accessors

`set` can be declared `private`

```
public string Name { get; private set;
}
```

- can only be set by the declaring class
- other classes can only read it
  (kind of read-only)

Object and Collection Initializers

Object Initializers

- **For creating and initializing objects in a single expression**

If you have a class (or a struct) with properties or fields like this

```
class Student {
  public string Name;
  public int Id;
  public string Field { get; set; }
  public Student() {}
  public Student(string name) { Name = name; }
}
```

you can create and initialize an object as follows:

```
Student s1 = new Student("John") {Id = 2009001, Field = "Computing" };
Student s2 = new Student {Name = "Ann", Id = 2009002, Field = "Mathematics" };
```

empty brackets can be omitted

Collection Initializers

- **For creating and initializing collections in a single expression**

Values can be specified after creation

```
var intList = new List<int> { 1, 2, 3, 4, 5 };
```

```
var personList = new List<Student> {
  new Student("John") {Field = "Computing" },
  new Student("Ann") {Field = "Mathematics" }
};
```

collection initializers and object initializers can be combined

```
var phoneBook = new Dictionary<string, int>
{
  { "John Doe", 4711 },
  { "Alice Miller", 3456 },
  { "Lucy Sky", 7256 }
};
```

initialization of a two-dimensional collection

Compiler translates this into

```
List<int> intList = new List<int>();
intList.Add(1); intList.Add(2); intList.Add(3);
intList.Add(4); intList.Add(5);
```

all collections support the *Add* method

Anonymous Types

Anonymous Types

- **For creating tuples of an anonymous (i.e. nameless) type**

```
var obj = new { Name = "John", Id = 100 };
```

```
class ??? {
  public string Name { get;  private set; }
  public int Id { get;private set; }
}
```

creates an object of a new type with the properties *Name* and *Id*

the type of the right-hand side value
type inference!

```
??? obj = new ???();
obj.Name = "John";
obj.Id = 100;
```

- **Even simpler, if the values are composed from <u>existing names</u>**

```
class Student {
  public string Name;
  public int Id { get; set; }
}
...
Student s = new Student();
string city = "London";
```

- properties of a existing object
- fields of an existing object
- local variables

```
var obj = new {s.Name, s.Id, city};
```

anonymous type with properties *Name*, *Id* and *city*

… Anonymous Types -- Details

```
var obj = new { Id = x, student.Name };
```

- Generated properties (*Id*, *Name*) are read only!
- Generated properties can be named explicitly (*Id* = *x*) or implicitly (*student.Name*). Explicit and implicit naming can be mixed (although uncommon).
- Anonymous types are compatible with *Object*
- Compiler generates a *ToString()* method for every anonymous type

```
Console.WriteLine(obj);
```

⇨ `{ Id = 1234, Name = "John Doe" }`

Type Inference -- var

```
var x = ...;
```

- *var* can only be used for **local variable** declarations (not for parameters and fields)
- variable **must be initialized in the declaration**
- the type of the variable is **inferred** from the initialization expression

Typical usage

```
var obj = new { Width = 100, Height = 50 };
```

```
??? obj = ...
```

```
var dict = new Dictionary<string, int>();
```

```
Dictionary<string, int> dict =
  new Dictionary<string, int>();
```

In principle, the following is also possible

```
var x = 3;
var s = "John";
```

```
int x = 3;
string s = "John";
```

but this is not recommended!

Partial Methods

Partial Methods

- **For providing user-defined hooks in automatically generated code**

Example

```
public partial class Accelerator {
  public void Accelerate() {
    BeforeAccelerate();
    ... do accelerate actions ...
    AfterAccelerate();
  }
  partial void BeforeAccelerate();
  partial void AfterAccelerate();
}
```

- must be partial methods in a partial class or struct
- must not have private, public, ...
- must be void
- must not have out parameters

Compiler does not generate calls

... unless some other part of this class supplies the bodies

```
public partial class Accelerator {
  partial void BeforeAccelerate() { ... }
  partial void AfterAccelerate() { ... }
}
```

Another Example

- Enabling/disabling trace output

```
partial class Stack {
  int[] data = new int[100];
  int len = 0;
  public void Push(int x) {
    Print("-- Push " + x + ", len = " + (len + 1));
    data[len++] = x;
  }
  public int Pop() {
    Print("-- Pop " + data[len-1] + ", len = " + (len - 1));
    return data[--len];
  }
  partial void Print(string s);
}
```

```
Stack s = new Stack();
s.Push(3);
int x = s.Pop();
```

no trace output so far

- Now we compile also the second part of *Stack*

```
partial class Stack {
  partial void Print(string s) {
    Console.WriteLine(s);
  }
}
```

Output

```
-- Push 3, len = 1
-- Pop 3, len = 0
```

Extension Methods

Extension Methods

- **Allow programmers to add functionality to an existing class**

Existing class Fraction

```
class Fraction {
  public int z, n;
  public Fraction
      (int z,int n) {...}
  ...
}
```

Assume that we want to extend it with an *Inverse* and an *Add* method

Extension methods for class Fraction

```
static class FractionUtils {
  public static Fraction Inverse (this Fraction f) {
    return new Fraction(f.n, f.z);
  }
  public static void Add (this Fraction f, int x) {
    f.z += x * f.n;
  }
}
```

Usage

```
Fraction f = new Fraction(1, 2);

f = f.Inverse();
// f = FractionUtils.Inverse(f);

f.Add(2);
// FractionUtils.Add(f, 2);
```

- must be declared in a **static class**
- must be **static methods**
- **first parameter** must be declared with ***this*** and must denote the class, to which the method should be added

- Can be called like instance methods of *Fraction*
- However, can only access public members of *Fraction*

Predeclared Extension Methods

System.Linq.Enumerable has predeclared extension methods for IEnumerable<T>

```
delegate TRes Func<T1, TRes>(T1 a);
```

```
namespace System.Linq {

  public static class Enumerable {

    public static IEnumerable<T> Where<T> (this IEnumerable<T> source, Func<T, bool> f) {
      ... returns all values x from source, for which f(x) == true ...
    }
    ...
  }
}
```

Usage

makes *Where* visible

```
Enumerable.Where(list, i => i > 0)
```

```
using System.Linq;
...
List<int> list = ... list of integer values ...;
IEnumerable<int> result = list.Where(i => i > 0);

foreach(int i in list.Where(i => i > 0) {Console.WriteLine(“”+i));}
```

Compiler does *type inference*

list is declared as *List<int>*
==> *T = int*
==> *i* is of type *int*

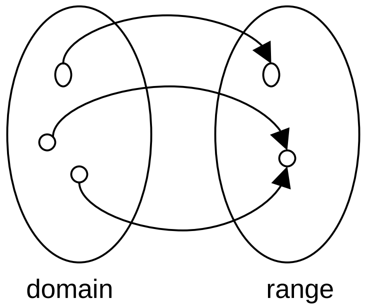
Can be applied to all collections and arrays!

```
string[] a = {"Bob", "Ann", "Sue", "Bart"};
IEnumerable<string> result =
  a.Where(s => s.StartsWith("B"));
```

Lambda Expressions

λ Calculus

- Functions are fundamental in computer science and mathematics
- in mathematics
  - values in the domain are transformed to values in the range
  - f x -> y



- in computer science
  - input is transformed to some output
- Examples
  - I (x) = x -> x
  - Sqr(x) = x -> x^2

- If you don't bother to name the function you simply call them λ
  - $\lambda x.x^2$
- λ calculus is important part of the theoretical computer science (Church 1940)
  - e.g. higher order functions = functions as arguments (e.g. differentiation)

Functional Programming

- Imperative programming (also object oriented) is based on states (of the programs and objects) and mutable data (value of variables)

- Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions
  - "pure" functional programming has no states and no mutable data
- History
  - Early functional programming languages (1960)
    - *LISP, APL, ML*
- Revival
    - *mixed languages, functional extensions to non-functional languages*
    - *Scala, clojure, C# 3.0*
- Purely functional programs have no shared state thus simplify concurrent programming

C# Lambda Expressions

= Short form for delegate values

```
delegate int Function(int x);  int Square(int x) { return x * x; }
                               int Inc(int x) { return x + 1; }
```

C# 1.0

```
Function f;
f = new Function(Square);      ... f(3) ... // 9
f = new Function(Inc);         ... f(3) ... // 4
```

C# 2.0

```
f = delegate (int x) { return x * x; }     ... f(3) ...     // 9
f = delegate (int x) { return x + 1; }     ... f(3) ...     // 4
```

C# 3.0

```
f = x => x * x;                ... f(3) ... // 9
f = x => x + 1;                ... f(3) ... // 4
```

Example for Lambda Expressions

Applying a function to a sequence of integers

```
delegate int Function (int x);
```

```
int[] Apply (Function f, int[] data) {
  int[] result = new int[data.Length];
  for (int i = 0; i < data.Length; i++) {
    result[i] = f(data[i]);
  }
  return result;
}
```

```
int[] values = Apply ( i => i * i , new int[] {1, 2, 3, 4});
```

```
=> 1, 4, 9, 16
```

Lambda Expressions -- Details

General form *Parameters* "=>" (*Expr* | *Block*)

Lambdas can have 0, 1 or more parameters

```
()        => ...
x         => ...
(x, y)    => ...
(x, y, z) => ...
```

```
// no parameters
// 1 parameter
// 2 parameters
// 3 parameters
```

Parameters can have types as well as `ref/out` modifiers

```
(int x) => ...
(string s, int x) => ...
(ref int x ) => ...
(int x, out int y) => ...
```

```
// must be in brackets although
just 1 parameter
```

Parameter types are usually not specified;
They are inferred from the declaration of the delegate to which they are assigned

```
delegate bool Func(int x, int y);
```

```
Func f =  (x, y) => x > y;
```

must be *int* must be *bool*

... Lambda Expressions -- Details

```
Parameters  "=>"  (Expr | Block)
```

Right-hand side is usually a result expression

```
x    =>  x * x                      // returns x * x
(x, y)   =>  x + y                  // returns x + y
```

Right-hand side can be a block returning a result

```
n  =>{int sum = 0;
      for (int i = 1; i <= n; i++) sum += i;
      return sum;
     }
```

Right-hand side does not return a result if the corresponding delegate is a void method

```
delegate void Proc(int x);
```

```
Proc p =   x => { Console.WriteLine(x); };
```

Right-hand side can access outer local variables (-> closures)

```
int sum = 0;
Proc p =   x => { sum += x; };
```

… Lambda Expressions -- Generic Delegate

- Delegate Type

```
delegate int Func ();
delegate double Func (double p);
```

- Generic Delegate Type
  - are also supported since C# 2.0

```
public delegate void Del<T>(T item);

public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

… Lambda Expressions -- Examples

Namespace *System.Linq* defines several generic delegate types

```
delegate TRes Func<TRes> ();
delegate TRes Func<T1, TRes> (T1 a);
delegate TRes Func<T1, T2, TRes> (T1 a, T2 b);
delegate TRes Func<T1, T2, T3, TRes> (T1 a, T2 b, T3 c);
delegate TRes Func<T1, T2, T3, T4, TRes> (T1 a, T2 b, T3 c, T4 d);
```

```
delegate void Action ();
delegate void Action<T1> (T1 a);
delegate void Action<T1, T2> (T1 a, T2 b);
delegate void Action<T1, T2, T3> (T1 a, T2 b, T3 c);
delegate void Action<T1, T2, T3, T4> (T1 a, T2 b, T3 c, T4 d);
```

Examples

	Call	*Result*
`Func<int, int>  f1 = x => 2 * x + 1;`	`f1(3);`	`7`
`Func<int, int, bool> f2 = (x, y) => x > y;`	`f2(5, 3);`	`true`
`Func<string, int, string> f3 = (s, i) => s.Substring(i);`	`f3("Hello", 2);`	`"llo"`
`Func<int[]> f4 = () => new int[] { 1, 2, 3, 4, 5 };`	`f4();`	`{1, 2, 3, 4,5}`
`Action a1 = () => { Console.WriteLine("Hello"); };`	`a1();`	`Hello`
`Action<int, int> a2 = (x, y) => { Console.WriteLine(x + y); };`	`a2(1, 2);`	`3`

LINQ

LINQ - Language Integrated Query

SQL-like queries in C#

■ LINQ to Objects	Queries on arrays and collections (*IEnumerable<T>*)
■ LINQ to SQL	Queries on databases (generating SQL)
■ LINQ to XML	Queries that generate XML

Everything is fully type checked!

Namespaces: *System.Linq*, *System.Xml.Linq*, *System.Data.Linq*

Conceptual novelties of LINQ

- Brings programming and databases closer together
- Integrates functional programming concepts into C# (lambda expressions)
- Promotes declarative programming style (anonymous types, object initializers)
- Introduces type inference

LINQ Queries to Objects (Example)

SQL-like queries on arbitrary collections (`IEnumerable<T>`)

Sample collection

```
string[] cities = {"London", "New York", "Paris", "Berlin", "Berikon"};
```

Query

```
IEnumerable<string> result =
  from c in cities
  select c;
```

```
IEnumerable<string> result =
  from c in cities
  where c.StartsWith("B")
  orderby c
  select c.ToUpper();
```

Result

```
foreach (string s in result) Console.WriteLine(s);
```

```
London
New York
Paris
Berlin
Berikon
```

```
Berikon
Berlin
```

LINQ queries are translated into ***lambda expressions*** and ***extension methods***

Query Expressions

Translation of Query Expressions

Example: Assume that we have the following declarations

```
class Student {
  public int Id { get; set; }
  public string Name { get; set; }
  public string Subject { get; set; }
}
```

```
List<Student> students = ...;
```

Translation

```
var result =
  from s in students
  where s.Subject == "Computing"
  orderby s.Name
  select new {s.Id, s.Name};
```

```
var result =
  students
  .Where( s => s.Subject == "Computing")
  .OrderBy( s => s.Name )
  .Select( s => new {s.Id, s.Name} );
```

lambda expressions

extension methods of IEnumerable<T>

anonymous type

```
foreach (var s in result)
  Console.WriteLine(s.Id + " " + s.Name);
```

A Closer Look at the Query Syntax

range variable

data source (supporting IEnumerable<T>)

```
var result = from s in students
         where s.Subject == "Computing"
         orderby s.Name
         select new { s.Id, s.Name };
```

clauses

projection

Note: The result is not a sequence of values but a "cursor" that is advanced when necessary (e.g. in a foreach loop or in other queries)

result is *IEnumerable<T'>* where *T'* is the type of the projection

7 kinds of query clauses

from	defines a range variable and a data source
where	filters elements of the data source
orderby	sorts elements of the data source
select	projects range variable(s) to elements of the result sequence
group	groups data source elements (converts sequence of elements into sequence of groups)
join	joins elements of multiple data sources
let	defines auxiliary variables

LINK Query Syntax

```
QueryExpr =
  "from" [Type] variable "in" SrcExpr
  QueryBody.

QueryBody =
  { "from" [Type] variable "in" SrcExpr
  | "where" BoolExpr
  | "orderby" Expr ["descending"] { "," Expr ["descending"] }
  | "join" [Type] variable "in" SrcExpr "on" Expr "equals" Expr ["into" variable]
  | "let" variable "=" Expr
  }
  ( "select" ProjectionExpr ["into" variable QueryBody]
  | "group" ProjectionExpr "by" Expr ["into" variable QueryBody]
  ).
```

SrcExpr	a data source implementing *IEnumerable<T>*
BoolExpr	a C# expression of type *bool*
Expr	a C# expression
ProjectionExpr	a C# expression defining the result elements

expressions on
the range variable(s)

Note: Query has to start with a *from*
Query has to end with a *select* or *group*

Range Variables

- Introduced in *from* and *join* clauses (also in *into* phrases)

```
from s in students
join m in marks on s.Id equals m.Id
group s by s.Subject into g
```

- Iterate over elements of the data source

- If the data source is of type *IEnumerable<T>* the range variable is of type *T* (the type can also be explicitly specified)

  students is of type *List<Student>*
  s is of type *Student*

- Range variables are read only!

- Scoping:
  - their names must be distinct from the names of outer local variables
  - their scope ends at the end of the query expression or at the next *into* phrase

Grouping

- Transforms input elements into **key/value pairs**
- Collects values with the same key into a group

```
var result =
  from s in students
  group s.Name by s.Subject;
```

value *key*

List<Student>	key/value pairs	*result* IEnumerable<IGrouping>

`Name="John", Id=2009001, Subject="Computing"`
`Name="Ann", Id=2009002, Subject="Mathematics"`
`Name="Sue", Id=2009003, Subject="Computing"`
`Name="Bob", Id=2009004, Subject="Mathematics"`
`...`
	`("Computing", "John")`
	`("Mathematics", "Ann")`
	`("Computing", "Sue")`
	`("Mathematics", "Bob")`
	`...`
		`"Computing": ("John", "Sue")`
		`"Mathematics": ("Ann", "Bob")`
		`...`

IGrouping<TKey, TElement>

- property *Key*
- group is of type *IEnumerable<TElement>*

```
foreach (var group in result) {
  Console.WriteLine(group.Key);
  foreach (var name in group) Console.WriteLine("  " + name);
}
```

```
Computing
   John
   Sue
Mathematics
   Ann
   Bob
...
```

Grouping into another Range Variable

- Necessary when you want to process the groups further

```
var result =
  from s in students
  group s by s.Subject into g
  select new { Field = g.Key, N = g.Count() };
```

s is not visible here any more but *g* is visible

```
foreach (var x in result) {
  Console.WriteLine(x.Field + " occurs " +
                              x.N + " times");
}
```

```
Computing occurs 2
times
Mathematics occurs 2
times
...
```

```
foreach (var x in result) Console.WriteLine(x);
```

```
{ Field = Computing, N = 2 }
{ Field = Mathematics, N = 2 }
...
```

calls *x.ToString()* of anonymous type

`group s ... into g` converts a sequence of students into a sequence of groups

Joins

- Combines records from multiple data sources if their keys match

```
class Student {
  public int      Id { get; set; }
  public string Name { get; set; }
  public string Subject { get; set; }
}
```

```
class Marking {
  public int      StudId { get; set; }
  public string   Course { get; set; }
  public int      Mark { get; set; }
}
```

```
var students = new List<Student> {...};
```

```
var marks = new List<Marking> {...};
```

Id	*Name*	*Subject*
2008001	"John Doe"	"Computing"
2008002	"Linda Miller"	"Chemistry"
2009001	"Ann Foster"	"Mathematics"
2009002	"Sam Dough"	"Computing"
...	...	...

StudId	*Course*	*Mark*
2008001	"Programming"	3
2008001	"Databases"	2
2008001	"Computer Graphics"	1
2008002	"Organic Chemistry"	1
...	...	...

- Join (explicit)

must be "equals" and not "=="

```
var result =
  from s in students
  join m in marks on s.Id equals m.StudId
  select s.Name + ", " + m.Course + ", " + m.Mark;
```

```
John Doe, Programming, 3
John Doe, Databases, 2
John Doe, Computer Graphics, 1
Linda Miller, Organic Chemistry, 1
...
```

Joins (implicit)

- Alternative way to specify the Join

```
var result =
  from s in students
  from m in marks
  where s.Id == m.StudId
  select s.Name + ", " + m.Course + ", " + m.Mark;
```

```
John Doe, Programming, 3
John Doe, Databases, 2
John Doe, Computer Graphics, 1
Linda Miller, Organic Chemistry, 1
...
```

- Result is the same but the query is less efficient
- builds the cross product (combines every student with every mark)
- filters out those results that match the where clause

Group Joins

- Makes matching records from the second data source a subgroup

```
var result =
  from s in students
  join m in marks on s.Id equals m.StudId into list
  select new { Name = s.Name, Marks = list };
```

does not become invisible by *into*

Name	Marks
John Doe	2008001, Programming, 3 2008001, Databases, 2 2008001, Computer Graphics, 1
Linda Miller	2008002, Organic Chemistry, 1 2008002, Mathematics, 2
...	

- Processing the result

```
foreach (var group in result) {
  Console.WriteLine(group.Name);
  foreach (var m in group) {
    Console.WriteLine("  " + m.Course + ", " + m.Mark);
  }
}
```

```
John Doe
  Programming, 3
  Databases, 2
  Computer Graphics, 1
Linda Miller
  Organic Chemistry, 1
  Mathematics, 2
...
```

let Clauses

- **Introduce auxiliary variables that can be used like range variables**

```
var result =
  from s in students
  where s.Subject == "Computing"
  let year = s.Id / 1000
  where year == 2009
  select s.Name;
```

Id	*Name*	*Subject*
2008001	"John Doe"	"Computing"
2008002	"Linda Miller"	"Chemistry"
2009001	"Ann Foster"	"Mathematics"
2009002	"Sam Dough"	"Computing"
...	...	...

```
foreach (string s in
result) {
  Console.WriteLine(s);
}
```

Result

```
Sam Dough
```

Further Extension Methods

- In class *System.Linq.Enumerable*

Can be applied to all *IEnumerable*<*T*>: query results, collections, arrays, ...

Assume: *e* is of type *IEnumerable*<*T*>

`e.Any(i => i < 0)`	true, if any element of *e* is < 0
`e.All(i => i > 0)`	true, if all elements of *e* are > 0
`e.Take(3)`	takes the first 3 elements of *e*
`e.Skip(2)`	drops the first 2 elements of *e*
`e.TakeWhile(i => i < 1000)`	takes elements from *e* as long as predicate is true
`e.SkipWhile(i => i < 100)`	drops elements from *e* as long as predicate is true
`e.Distinct()`	yields *e* without duplicates
`e.Concat(e2)`	appends *e2* to *e*
`e.Reverse()`	yields *e* in reverse order
`e.ToList()`	converts an *IEnumerable*<*T*> into a *List*<*T*>
`e.ToArray()`	converts an *IEnumerable*<*T*> into a *T*[]
`e.OfType<string>()`	yields all elements of *e* that are of type string

...

LINQ to XML

XElement and XAttribute

Creating simple elements

(Namespace *System.Xml.Linq*)

```
XElement e = new XElement("name", "John");
Console.WriteLine(e);
```

```
<name>John</name>
```

Creating nested elements

```
XElement e = new XElement("student",
  new XElement("name", "John"),
  new XElement("subject", "Computing"));
Console.WriteLine(e);
```

```
<student>
  <name>John</name>
  <subject>Computing</subject>
</student>
```

Creating elements with attributes

```
XElement e = new XElement("student",
  new XAttribute("id", 2009001),
  new XElement("name", "John"),
  new XElement("subject", "Computing"));
Console.WriteLine(e);
```

```
<student id=2009001>
  <name>John</name>
  <subject>Computing</subject>
</student>
```

Reading an XML file

```
XElement e = XElement.Load(new XmlTextReader("input.xml"));
```

Generating XML with LINQ

```
using System.Linq;
using System.Xml.Linq;
...
XElement xmlData =
  new XElement("students",
    from s in students
    where s.Subject == "Computing"
    select new XElement("student",
      new XAttribute("id", s.Id),
      new XElement("name", s.Name)
    )
  );
Console.WriteLine(xmlData);
```

IEnumerable<XElement>

Output

```
<students>
  <student id="2008001">
    <name>John Doe</name>
  </student>
  <student id="2009002">
    <name>Sam Dough</name>
  </student>
  ...
</students>
```

Processing XML with LINQ

School of Engineering

```
using System.Linq;
using System.Xml.Linq;
...
XElement xmlData = ...;
```

xmlData

```
<students>
  <student id="2008001">
    <name>John Doe</name>
  </student>
  <student id="2009002">
    <name>Sam Dough</name>
  </student>
  ...
</students>
```

```
IEnumerable<Student> result =
  from e in xmlData.Elements("student")
  select new Student {
    Name = e.Element("name").Value,
    Id = Convert.ToInt32(e.Attribute("id").Value),
    Subject = "Computing"
  };
```

xmlData.Elements("student")	returns all subelements of *xmlData* that have the tag name "student" as an *IEnumerable<XElement>*
e.Element("name")	returns the first subelement of *e* that has the tag "name"

LINQ to DataSets

LINQ Queries to DataSets

- New Versions .NET 4.0 of DataSets support LINQ
- e.g. SQL type of queries also for DataSet

```
DataSet ds = new DataSet();
FillOrders(ds); // this method fills the DataSet from a database

DataTable orders = ds.Tables["SalesOrderHeader"];

var query = from o in orders.ToQueryable()
            where o.Field<bool>("OnlineOrderFlag") == true
            select new { SalesOrderID = o.Field<int>("SalesOrderID"),
                         OrderDate = o.Field<DateTime>("OrderDate") };

foreach(var order in query) {
    Console.WriteLine("{0}\t{1:d}", order.SalesOrderID, order.OrderDate);
}
```

http://msdn.microsoft.com/en-us/library/aa697427%28VS.80%29.aspx

New Features in C# 4.0

released end of 2009
VS 2010

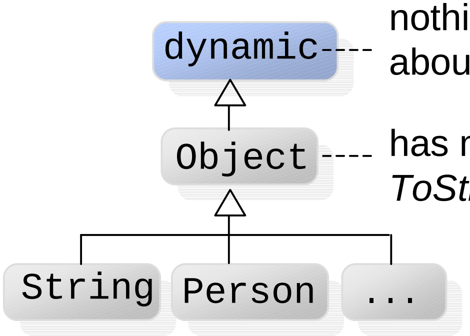
- Dynamic Typing
- Optional and Named Parameters
- Safe Co- and Contra-Variance for Generic Types
- ...

Dynamic Typing

Type dynamic

`dynamic d;` *d* can hold a value of any type

Can be considered to be a base type of *Object*



nothing is known about this type

has methods like *ToString*, *GetType*, ...

anything can be assigned to *d*

```
d = 5;
d = 'x';
d = true;
d = "Hello";
d = new Person();
...
```

possibly boxing

implicit conversion back

```
int i     = d;
char c    = d;
bool b    = d;
string s  = d;
Person p  = d;
...
```

with run-time check

For objects whose type is statically unknown

- objects of dynamic languages (Python, Ruby, ...)
- COM objects
- HTML DOM objects
- objects retrieved via reflection

$\Bigg\}$ simplifies interoperation with dynamic languages

Difference to `var v = ...;`

Compiler knows the type of *v* but not the type of *d*

Difference to `Object o;`

Object is a normal class which is known to the compiler

Operations on dynamic Variables

- **Have to be checked at run time** (defers type checking from compile to run time)

Code	Operation	Checks to be performed at run time
`dynamic d;`		
`d.Foo(3);`	*method call*	does the run-time type of *d* have a method *Foo*? does this method have an *int* parameter?
`d.f = ...;`	*field access*	does the run-time type of *d* have a field *f*? does the type of *f* match the assigned expression?
`d.P = d.P + 1;`	*property access*	does the run-time type of *d* have a property *P*? does the type of *P* match its use?
`d[5] = d[3];`	*indexer access*	does the run-time type of *d* have an indexer? does the type of this indexer match its use?
`d = d + 1;`	*operator access*	does the run-time type of *d* support the operator +? does the result type of this operator match its use?
`d(1, 2);`	*delegate call*	is the run-time type of *d* a delegate? does this delegate have two *int* parameters?

- The result of any dynamic operation is again dynamic
- A dynamic operation is about 5-10 times slower than a statically checked operation!

Run-time Lookup

- **How is d.Foo(3) invoked at run time?**

```
Type t = d.GetType()
if (t is a .NET type) {
  //--- use reflection to call this method
  MethodInfo m = t.GetMethod("Foo", new Type[] {typeof(int)});
  if (m == null) throw new Exception(...);
  m.Invoke(d, new Object[] {3});
}
```

For plain .NET objects

```
else if (t is a COM type) {
  //--- use COM's IDispatch mechanism to call the method
  ... pass (d, t, "Foo", 3) to COM and do the IDispatch ...
  ... throw an exception if the call is not possible ...
}
```

For COM objects (e.g. Excel, Word, ...)

```
else if (t implements IDynamicObject) {
  //--- let IDynamicObject do the call (for dynamic
languages)
  ... pass (d, t, "Foo", 3) to IDynamicObject ...
  ... throw an exception if the call is not possible ...
}
```

For objects of dynamic languages

Interfacing to other object models is usually done by implementing *IDynamicObject*

dynamic Overload Resolution

```
void Foo(string s) {...}
void Foo(int i) {...}
```

```
dynamic val = "abc";
Foo(val);
```

will invoke *Foo(string)*

```
dynamic val = 3;
Foo(val);
```

will invoke *Foo(int)*

Overload resolution is done at run time if one of the parameters is *dynamic*

http://www.developerfusion.com/article/9789/c-40-goes-dynamic-a-step-too-far/

Optional and Named Parameters

Optional Parameters

- **Declared with default values in the parameter list**

required — *optional*

```
void Sort<T>(T[] array, int from = 0, int to = -1, bool ascending = true,
                                                    bool ignoreCase = false) {
  if (to == -1) to = array.Length - 1;
  ...
}
```

would like to use *array.Length - 1*
but *Length* is not a compile-time constant

- Optional parameters must be declared after the required parameters
- Default values must be evaluable at compile time (constant expressions)

Usage

```
int[] a = {3, 5, 2, 6, 8, 4};
Sort(a, 0, a.Length - 1, true, false);
Sort(a);
Sort(a, 0, -1);
Sort(a, 0, -1, true);
```

parameters listed explicitly

from == 0, to == -1, ascending == true, ignoreCase = false

ascending == true, ignoreCase = false

ignoreCase = false

Optional parameters cannot be omitted from the middle

```
Sort(a, , , true);
```

Optional Parameters and Named Parameters

Parameters can be identified *by name* instead of *by position*

```
void Foo (int a, int b, int c, int d) {...}
```

can be called as

```
Foo(1, 2, 3, 4);
Foo(1, 2, c:3, d:4);
Foo(1, d:4, c:3, b:2);
```

identified *by name*

identified *by position*

- positional parameters must preceed named parameters
- named parameters can occur in any order

Useful for long lists of optional parameters

```
void Sort<T>(T[] array, from = 0, to = -1, ascending = true, ignoreCase = false) { … }
```

```
Sort(a, ascending: true);
Sort(a, ignoreCase: true, from: 3);
```

Optional Parameters and Overriding

- **Overridden methods can have parameters with different default values**

```
class A {
  public virtual void M (int x = 1, int y = 2) { Console.WriteLine(x + ", " + y); }
}
```

```
class B: A {
  public override void M (int x, int y = 3) { Console.WriteLine(x + ", " + y); }
}
```

no optional parameter any more

- **Call**

```
A a = new B();
a.M();
```

```
=> a.M(1, 2);
```

calls *B.M* but output is

```
1, 2
```

optional Parameters are passed by the caller according to the <u>static type</u> of *a*

```
B b = new B();
b.M(5);
```

```
=> b.M(5, 3);
```

calls *B.M*, output is

```
5, 3
```

5 must be specified because *x* is not optional

Safe Co- and Contra-Variance for Generic Types

Covariance

- Co-variance: the types are leveled up according the inheritance hierarchy, e.g. in overwritten methods

```
Class A {
   void foo(A a) {…}
}
```

```
Class B : A  {
   void foo(B b) {…}
}
```

- Can be achieved for interfaces with generics

```
Interface A<E> {
   void foo(E e) {…}
}
```

```
Class B : A<B>  {
   void foo(E e) {…}
}
```

e.g. Comparable, compareTo

Situation up to C# 3.0

List<T1> is incompatible with *List<T2>*

Why?

```
List<String> stringList = new List<String> { "John", "Ann", "Bob" };
List<Object> objList = stringList;  // not allowed -- but assume it were
```

```
objList[0] = 100;  // ok for the compiler
String s = stringList[0]; // would retrieve an int as a string
```

Problem

objList[i] can be assigned a value (of any type)
=> *stringList* is not necessarily a list of strings any more

Solution

objList = *stringList*; can be allowed if *objList* is never modified
i.e., if values are only retrieved from *objList* but never added or modified

Safe Co-Variant Generic Types

- If a type parameter is only used in "output positions" it can be marked with *out*

```
interface Sequence<out T>
{
  int Lengh { get; }
  T this[int i] { get; }
}
```

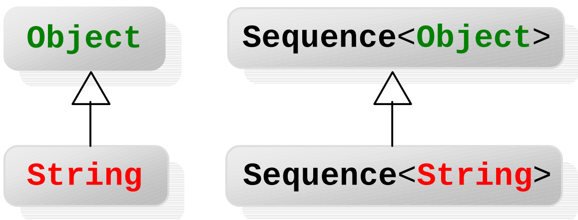
T is used in output position: only get but not set

This allows

```
Sequence<String> strings = ...;
Sequence<Object> objects;
objects = strings;
```

objects[i] will yield *Object*s which happen to be *String*s
=> safe, because a *objects* cannot be modified

Co-variance



if *String* is assignable to *Object*
then *Sequence<String>* is assignable to *Sequence<Object>*

Safe Contra-Variant Generic Types

- If a type parameter is only used in "input positions" it can be marked with *in*

```
interface IComparer<in T> {
  int Compare(T x, T y);
}
```

T is used in input position: only set but not get

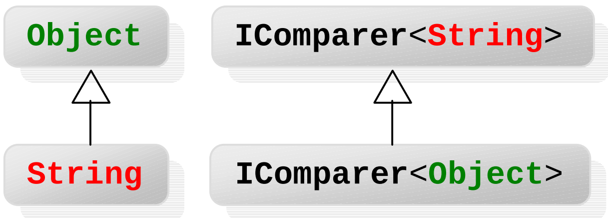
This allows

```
IComparer<Object> objComparer = ...;
IComparer<String> stringComparer;
stringComparer = objComparer;
int x = stringComparer.Compare("John", "Sue");
```

"John" "Sue"

will call
objComparer.Compare(Object x, Object y)
=> safe because Strings are Objects

Contra-variance



if *String* is assignable to *Object*
then *IComparable<Object>*
is assignable to *IComparable<String>*

Restrictions

- **Co/Contra-variance can only be used for interfaces and delegate types**

  Not for classes, because classes can have fields that can be read and written

- **interface I <out T> { ... }**
  - *T* can only be used as a **return type** (not as an *out* or *ref* parameter)
  - Types that replace *T* must be **reference types** (not value types)
    Sequence<int> cannot be assigned to *Sequence<Object>*

- **interface I<in T> { ... }**
  - Types that replace *T* must be **reference types** (not value types)
    IComparer<int> cannot be assigned to *IComparer<short>*

Safe Co/Contra-Variance for Delegates

```
delegate TResult Func<in TArg, out TResult> (TArg val);
```

```
String HashCodeAsString(Object obj) {
  return obj.GetHashCode().ToString();
}
```

```
Func<Object, String> f1 = HashCodeAsString;
String s = f1(new Person());
```

The following works as well

```
Func<String, Object> f2 = HashCodeAsString;
Object o = f2("Hello");
```

- "Hello" is passed to *obj* 4
  TArg is contra-variant: *Func<String, ...>* $\Leftarrow$ *Func<Object, ...>*
- The hash code as a *String* is returned as an *Object* 4
  TResult is co-variant: *Func<..., Object>* $\Leftarrow$ *Func<..., String>*

Co-variant Arrays vs. Co-variant Generics

Object arrays

```
Object[] objArr;
String[] strArr = ... ;
```

```
objArr = strArr;
```

```
objArr[i] = val;
```

run-time check whether the run-time type of *val* is assignable to the run-time type of the elements of *objArr*

Generics

```
interface Sequence<T> { ... }
Sequence<Object> objects;
Sequence<String> strings = ... ;
```

```
objects = strings;
```

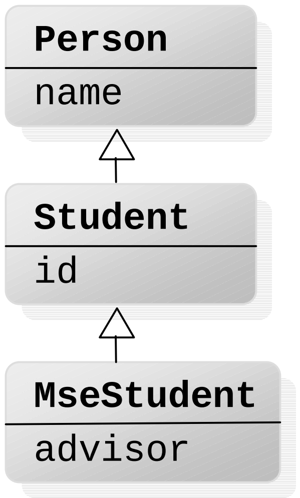
But

```
interface Sequence<out T> { ... }
Sequence<Object> objects;
Sequence<String> strings = ... ;
```

```
objects = strings;
```

no run-time checks necessary because *objects* cannot be modified

Another Summarizing Example




```
interface IEnumerable<out T> {  // in
System.Collections.Generic
  IEnumerator<T> GetEnumerator();
}
interface IPrintable<in T> {
  void Print(T val);
}
```

```
class PersonPrinter: IPrintable<Person> {...}     //
prints name
class StudentPrinter: IPrintable<Student> {...}   //
prints name, id
...
```

```
void Process (IEnumerable<Student> students, IPrintable<Student> printer) { ... }
```

⇧ ⇧

can be called with `IEnumerable<MseStudent>` `IPrintable<Person>` prints the names of all MSE students

co-variance *contra-variance*

safe because IEnumerable has <out T> IPrintable has <in T>

async & await

async & await

- To simplify the writing of asynchronous methods

```
async Task<int> fooAsync(){
   // e.g. call other async methods
   return 42;
}
```

```
async Task bar()
    Task fooTask = fooAsync();
    DoIndependentWork();
    int i = await fooTask ;
    DoDependentWork(i);
}
```

- async: the method signature of an asynchronous includes an async modifier.
  - The name of an async method, by convention, ends with an "Async" suffix.
- Task: The return Type is Task<T> or Task (if void method)
- await: wait until async called method returns

Applications

Application area	Supporting APIs that contain async methods
Web access	HttpClient , SyndicationClient
Working with files	StorageFile, StreamWriter, StreamReader, XmlReader
Working with images	MediaCapture, BitmapEncoder, BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations

async & await example

async modifier

return type is Task or Task<T>

The method name ends in "Async."

```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    Task<string> getStrTask=client.GetStringAsync("http://msdn.microsoft.com");

    DoIndependentWork();

    string urlContents = await getStrTask;

    return urlContents.Length;
}
```

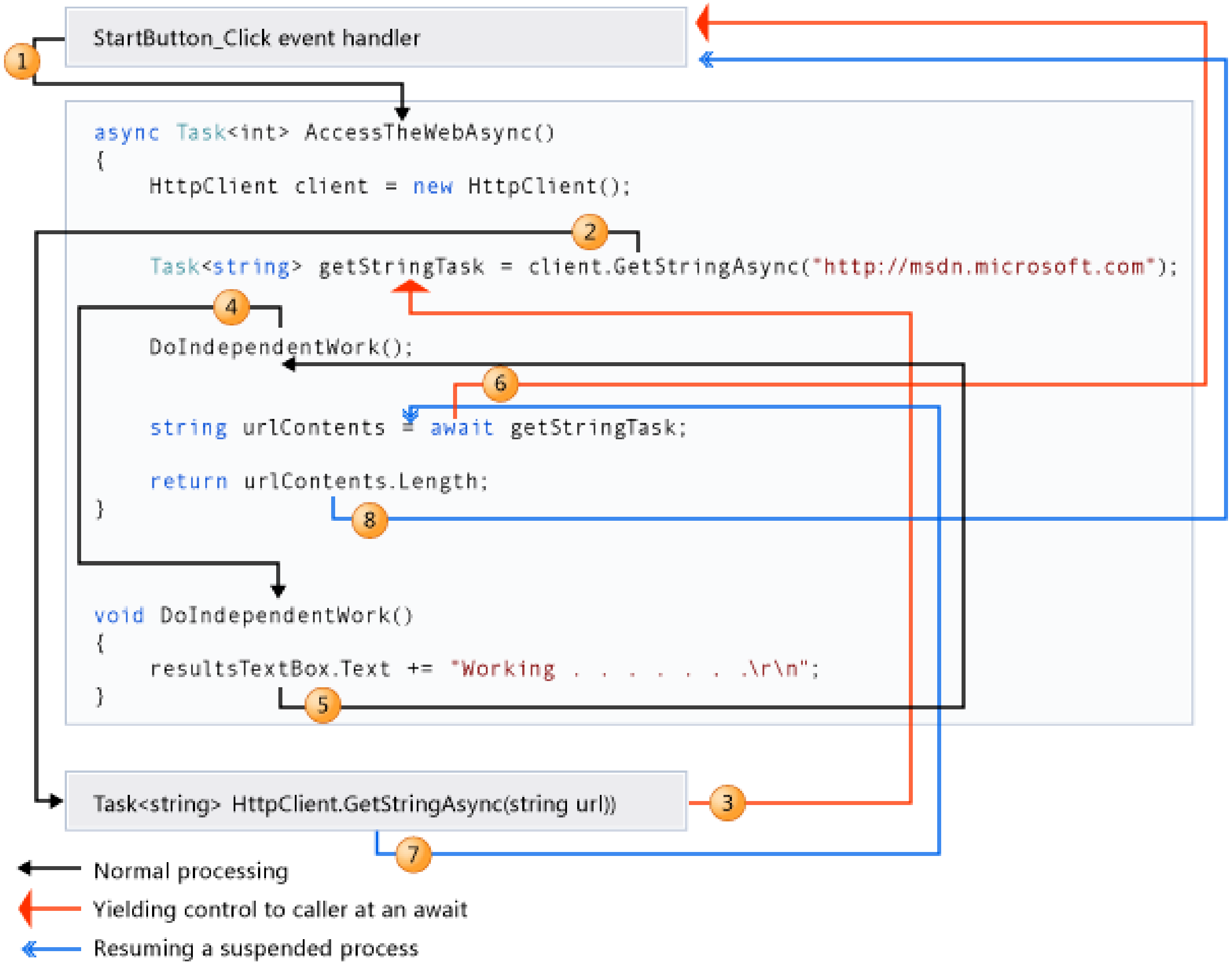
call an async method internally

do some independent work while waiting

wait until httpClient returns content

Task<int> because the return statement returns an integer.

async & await explained



… async & await explained

1. An event handler calls and awaits the AccessTheWebAsync async method.
2. AccessTheWebAsync creates an HttpClient instance and calls the GetStringAsync asynchronous method to download the contents of a website as a string.
3. Something happens in GetStringAsync that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, GetStringAsync yields control to its caller, AccessTheWebAsync.

4 GetStringAsync returns a Task<TResult> where TResult is a string, and AccessTheWebAsync assigns the task to the getStringTask variable. The task represents the ongoing process for the call to GetStringAsync, with a commitment to produce an actual string value when the work is complete.

5. Because getStringTask hasn't been awaited yet, AccessTheWebAsync can continue with other work that doesn't depend on the final result from GetStringAsync. That work is represented by a call to the synchronous method DoIndependentWork.
6. DoIndependentWork is a synchronous method that does its work and returns to its caller.
7. GetStringAsync completes and produces a string result. The string result isn't returned by the call to GetStringAsync in the way that you might expect.

Fragen ?