Projektowanie i Konstrukcja Systemów Rozproszonych

Wykład 1
Podstawy środowiska i współbieżność w .NET

Co to jest .NET?

- Rozwiązanie kompleksowe
  - języki programowania – C#, VB.NET, J#, MC++, ...
  - środowisko uruchomieniowe
  - zbiór bibliotek
  - narzędzia programistyczne
- wieloprocesorowe (i wielosystemowe)
- objęte standardami ECMA* i ISO

*ECMA – przykładowa komisja standaryzująca MS OOXML

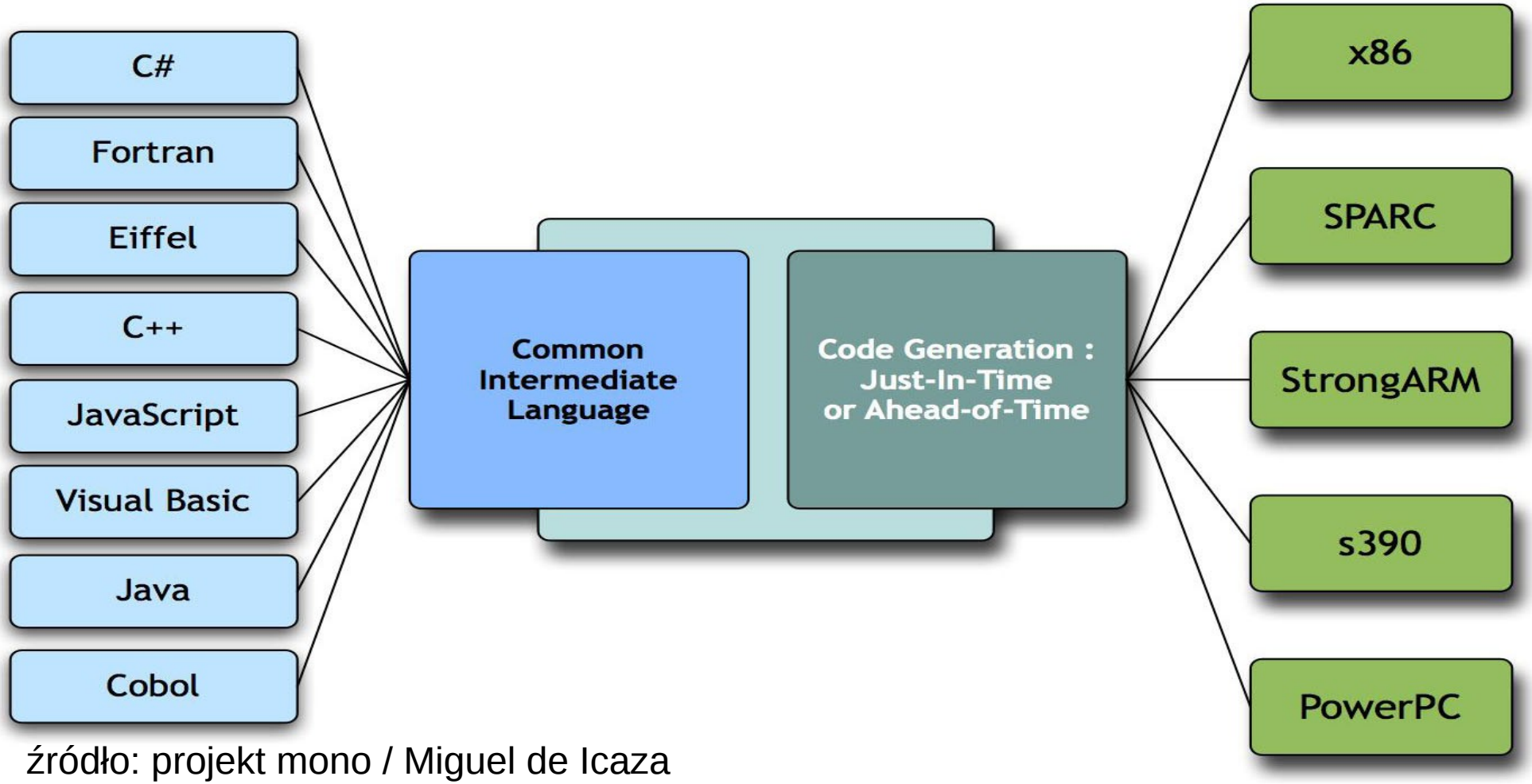
Przewodniczący: Mr. J. Paoli (Microsoft)

Mrs. I. Valet-Harper (Microsoft)

Vice-przewodniczący: Mr. A. Farquhar (British Library)

Sekretarz: Dr. Istvan Sebestyen (Ecma)

Co to jest .NET? (2)



źródło: projekt mono / Miguel de Icaza

Historia

- dlaczego .NET?
- lipiec 2000 – Microsoft, Hewlett-Packard i Intel opracowują standardy CLI i języka C#
- grudzień 2001 – standaryzacja przez ECMA jako ECMA-334 (C#) oraz ECMA-335 (CLI)
- kwiecień 2003 – standaryzacja przez ISO jako ISO/IEC 23270 (C#) i ISO/IEC 23271 (CLI)

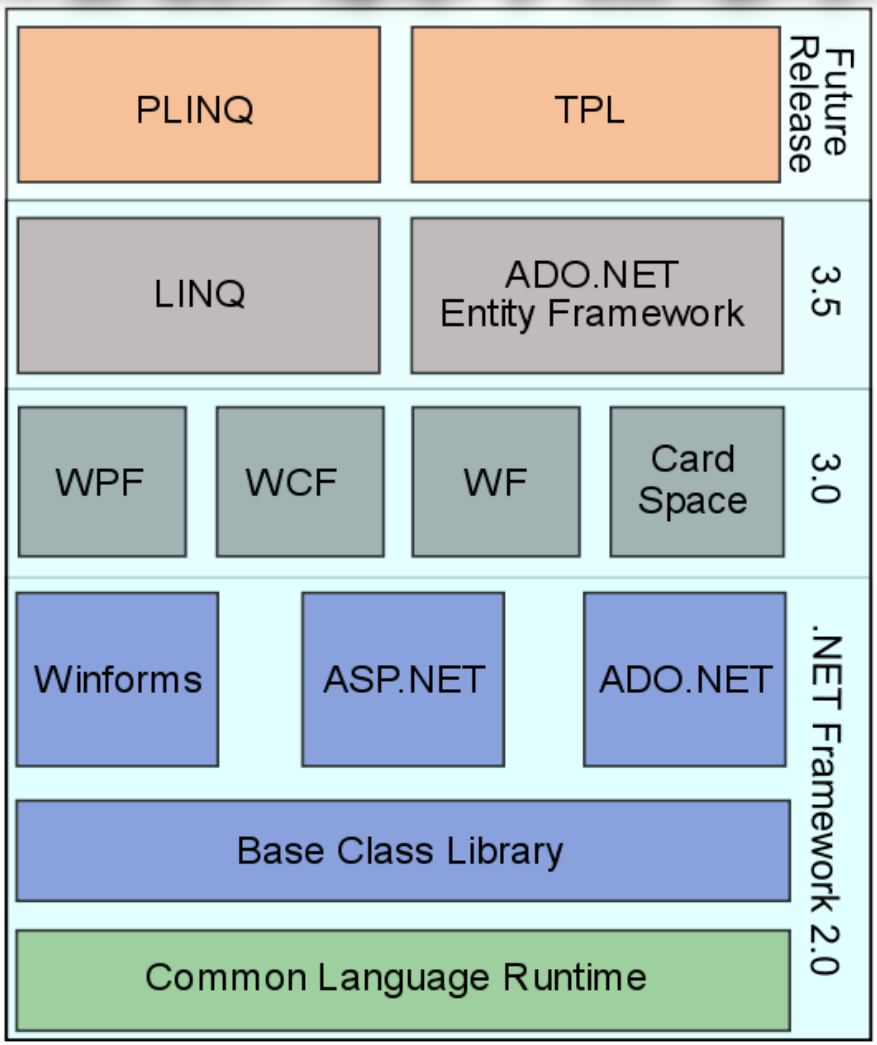
Historia (2)

- Wersje framework
  - MS .NET Framework 1.0 (C# 1, CLR/BCL v1.0) – 01.2002,
    - MS .NET Framework 1.1 (C# 1, CLR/BCL v1.1) i Compact Framework 1.0 – 04.2003
    - MS .NET Framework 2.0 (C# 2, CLR/BCL v2.0) i Micro Framework –11.2005,
    - MS .NET Framework 3.0 (C# 2, CLR/BCL v2.0) – 11.2006
      - Windows Presentation Foundation (WPF, Avalone),
      - Windows Communication Foundation (WCF, Indigo),
      - Windows Workflow Foundation (WF),
      - Windows CardSpace (WCS, InfoCard)

Historia (3) – „Dział marketingu”

- Wersje framework
  - MS .NET Framework 3.5 – 11.2007
    - .NET Framework 2.0 SP1 – C# 2, CLR/BCL v2.0 SP1
    - .NET Framework 3.0 SP1 – C# 2, CLR/BCL v2.0 SP1 + WCF, WPF, WCS, WF
    - .NET Framework 3.5 – C# 3, .NET 3.0 SP1, LINQ
  - MS .NET Framework 3.5 SP1 – 08.2008
    - .NET Framework 2.0 SP2 – C# 2, CLR/BCL v2.0 SP2
    - .NET Framework 3.0 SP2 – C# 2, CLR/BCL v2.0 SP2 + WCF, WPF, WCS, WF
    - .NET 3.5 SP1 – C# 3, .NET 3.0 SP2, LINQ, EF, MVC
    - .NET 3.5 SP1 + GDR
  - MS .NET Framework 4.0 – C# 4, 04.2010 (CLR 4.0)
  - MS .NET Framework 4.5 – C# 5, 08.2012 (CLR 4.5); 4.5.1 VS2013

Architektura



The .NET Framework Stack

Historia (4)

- Inne framework'i
  - mono (http://www.mono-project.com)
    - 6.2001 – ogłoszenie projektu przez Miguela de Icaza
    - 5.2004 – mono 1.0 (MS .NET Framework 1.1)
    - 10.2006 – mono 1.2 (MS .NET 1.1 z MWF, .NET 2.0)
    - 09.2008 – mono 2.0 (C# 3.0, MWF 2.0)
    - 03.2009 – mono 2.4 (Mono.SIMD)
    - 12.2009 – mono 2.6 (moonlight, soft debugger, profil 4.0)
    - 10.2010 – mono 2.8 (sgen, profil 4.0, C# 4.0, WCF, ASP MVC 2)
    - 02.2011 – mono 2.10 (sgen, nowy kompilator i profiler)
    - 10.2012 – mono 3.0 (c# 5, async)
    - 07.2013 – mono 3.2

Specyfika środowiska

- kod zarządzany
  - zarządzanie pamięcią i Garbage Collector
- budowa aplikacji – assembly = kod pośredni + manifest -> dll, exe,
- kompilacja Just-in-Time
- cache bibliotek GAC, konfiguracja bibliotek
- reflection API

Języki

- w MS .NET dostępne: C#, VB.NET, MC++
- w innych projektach: Java, Nemerle, Boo, ...
- DLR (Dynamic Language Runtime) – Python, JavaScript (EcmaScript 3.0), Visual Basic, Ruby, F#
- zalecany C#
  - stworzony pod platformę
  - naturalna składnia
  - bardzo podobny do języka Java
  - aktualnie wersja 5.0 języka (z Framework 4.5)

Programowanie w .NET (1)

- Typy danych
  - proste i referencyjne
  - struktury i klasy
- Definicja klasy

```
public class Demo1 {
    public Demo1 () {}
    public ~Demo1 () {}
    public void Add1 (int x, int y) { y += x}
    public void Add2 (int x, ref int y) {y += x}
    public void Add3 (int x, out int y) {y = x}
}
```

Programowanie w .NET (2)

- Parametry
  - domyślnie – przez (kopiowaną) referencję,
  - *out* – przekazanie wyniku na zewnątrz,
  - *ref* – wymuszenie przekazania przez referencję,
- Poziomy dostępu
  - *public*
  - *protected*
  - *private*
  - *internal*
  - *protected internal*

Programowanie w .NET (3)

- Składowe klasy
  - pola, metody, konstruktory/destruktory
  - właściwości

```
public int Sthing {
    get { return m_sth;}
    set { m_sth = value;}
}
```

  - delegacje
  - zdarzenia
  - atrybuty

Programowanie w .NET (4)

- Składowe klasy
  - statyczne

```
public class X {public static int m_x = 1;}
```

  - stałe

```
public class Y {public const int PI = 3.1;}
```

  - składowe „sealed”

```
public sealed class Y {int PI = 3.1;}
```

    - klasa – nie można z niej dziedziczyć,
    - metoda – nie może być przeciążona,
    - każda struktura jest *sealed*

Programowanie w .NET (5)

- Wyliczenia

```
public enum ProjLevel : byte {
    Lite = 0x01,
    Normal = 0x02
}
```

- Rzutowanie
  - `as` – rzutuj i zwróć referencję nowego typu

```
Cat c = animal as Cat;  //can be null
```

  - `is` – sprawdź, czy rzutowanie możliwe *(bool)*

```
if (animal is Cat) {...
```

Programowanie w .NET (6)

- Delegacje
  - odpowiedniki wskaźników na metodę

```
public delegate string FancyStringDel  (int x);
public string ConvOne (int val) {...}
public void Method () {
   FancyStringDel del = new FancyDelString
  (ConvOne);
   del (2);
}
```

Programowanie w .NET (7)

- Zdarzenia – komunikacja wiele-wiele między klasami

```
public delegate void CalcDel (int x);
public event CalcDel CalculationDone;
CalculationDone (3);
...
CalculationDone += new CalcDel (ShowRes);
CalculationDone -= new CalcDel (ShowRes);
```

Programowanie w .NET (8)

- Dziedziczenie – tylko publiczne

```
class Truck : Vehicle {
    void GoTo (string s) {...}
    void GoTo (Point p) {…}}
```

- Pojedyncze dziedziczenie
- Przeciążanie metod i operatorów

```
public static Point operator+ (Point p, int dx,
  int dy) {...}
```

Programowanie w .NET (9)

- Przesłanianie i ukrywanie metod

```
public class Car {
    public virtual double GetMaxSpeed (...);
    public bool Stopped (...);
}
```

- Przesłanianie i ukrywanie metod (cd)

```
public class SportCar : IComparable, Car {
    public override double GetMaxSpeed (...);
    public new string Stopped (...);
}
```

Programowanie w .NET (10)

- Klasa bazowa – składowa `base`
- Klasa abstrakcyjna – modyfikator `abstract`
- *System.ApplicationException*
  - `try / catch / finally / throw`

Programowanie w .NET (11)

- Atrybuty

```
[AttributeUsage(AttributeTargets.All)]
public class AuthInfo : System.Attribute {
    public AuthInfo(string name) {...}}
...
[AuthInfo(”Bono”)]
public class AnyClass {...}
...
MemberInfo mi = typeof(AnyClass);
object [] atts = mi.
  GetCustomAttributes(typeof(AuthInfo),true);
```

Programowanie w .NET – C# 2.0

- Metody anonimowe

```
button1.Click += delegate(object s, EventArgs e) {
    MessageBox.Show("You clicked the button”);
};
```

- Iteratory

```
public class Colors {
  public string R, G, B;
  public IEnumerator<string> GetEnumerator() {
     yield return R; yield return G; yield return B;
  }}
foreach (String s in Colors) {...}
```

Programowanie w .NET – C# 2.0 (2)

- klasy częściowe

```
public partial class PartialClass {...}
```

- operator `??`
- generics
  - tworzenie klas, metod, struktur z nieznanym, ale ustalanym i sprawdzanym w czasie kompilacji typem
  - idea bazująca na templates z C++
- Nullable types: `int? x = null`

Programowanie w .NET – C# 2.0 (3)

- generics

```
public class Dictionary<K,V>
  {
     public void Add(K key, V value)
     {...
       if (((IComparable)key).CompareTo(x) < 0) {...}
     }
  }
// błąd, typ K nie zawiera CompareTo (object O)
public class Dictionary<K,V> where K : IComparable
```

Zmiany w .NET 3.0

- brak zmian w języku i podstawowych bibliotekach
- dodanie zestawy bibliotek *WinFX*
  - *Windows Communication Foundation*
  - *Windows Presentation Foundation*
  - *Windows Workflow Foundation*
  - *Windows CardSpace*

C# 3.0, (.NET Framework 3.5)

- zmienne typowane implicite

```
var orders = new Dictionary<int,Order>();
```

- metody rozszerzające *(extension methods)*

```
public static T[] Slice<T>(this T[] source, int index, int count){
   if (index < 0 || count < 0 || source.Length – index < count)
       throw new ArgumentException();
   T[] result = new T[count];
   Array.Copy(source, index, result, 0, count);
   return result;

}

...

int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int[] a = digits.Slice(4, 3);
```

C# 3.0 (2)

- wyrażenia lambda *(lambda expressions)* – rozszerzenie metod anonimowych

```
x => x + 1                  // Impl. typed, expr. body
x => { return x + 1; }      // Impl. typed, statement body
(int x) => x + 1            // Expl. typed, expr. body
(int x) => { return x + 1; } // Expl. typed, statement body
(x, y) => x * y                // Multiple parameters
() => Console.WriteLine()      // No parameters
delegate R Func<A,R>(A arg);
delegate void Action<T> (T obj);
Func<int,int> f1 = x => x + 1;          // Ok
Func<int,double> f2 = x => x + 1;       // Ok
Func<double,int> f3 = x => x + 1;       // Error
```

C# 3.0 (3)

- wyrażenia lambda – przykład

```
class ItemList<T>: List<T> {
   public int Sum(Func<T,int> selector) {
      int sum = 0;
      foreach (T item in this) sum += selector(item);
      return sum;
   }
}
...
void ComputeSums() {
   ItemList<Detail> orderDetails = GetOrderDetails(...);
   int totalUnits = orderDetails.Sum(d => d.UnitCount);
...
```

C# 3.0 (4)

- object initializers

```
public class Rectangle
   {
       Point p1 = new Point();
       Point p2 = new Point();
       public Point P1 { get { return p1; } }
       public Point P2 { get { return p2; } }
   }

var r = new Rectangle {
       P1 = { X = 0, Y = 1 },
       P2 = { X = 2, Y = 3 }
   };
```

- typy anonimowe

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2; // OK
```

C# 3.0 (5)

- LINQ: .NET Language-Integrated Query – integracja z SQL i XML

- ```
  from c in customers
  where c.City == "London"
  select c
  // tłumaczone na: customers.Where(c => c.City == "London")
  ```

- ```
  from c in customers
  where c.City == "London"
  from o in c.Orders
  where o.OrderDate.Year == 2005
  select new { c.Name, o.OrderID, o.Total }
  //tłumaczone na:
  customers.Where(c => c.City == "London").
  SelectMany(c =>
      c.Orders.
      Where(o => o.OrderDate.Year == 2005).
      Select(o => new { c.Name, o.OrderID, o.Total })
  )
  ```

C# 4.0 (1)

- dynamic language runtime (DLR)
  - wsparcie języków dynamicznych i skryptowych
  - *IronPython*, *IronRuby*
  - `dynamic` – sprawdzanie typów w czasie uruchomienia
- wprowadzono krotki (*tuples*) – klasa `Tuple`

```
Tuple<string, Nullable<int>>[] scores =
                    { new Tuple<string, Nullable<int>>("Jack", 78),
                      new Tuple<string, Nullable<int>>("Abbey", 92),...
```

- wzorzec *`IObserver(T)`* i *`IObservable(T)`*

C# 4.0 (2)

- kowariancja i kontrawariancja dla typów ogólnych
  - kowariancja

```
class Base {...}
class Derived : Base {
    public static void Main() {
         IEnumerable<Base> newbase = new List<Derived>();
    }
}
```

- parametry nazwane i opcjonalne

```
public void ExampleMethod(int required, string optionalstr = "default string",
            int optionalint = 10) {...}
anExample.ExampleMethod(3, optionalint: 4);
```

C# 4.0 (3)

- kowariancja i kontrawariancja dla typów ogólnych
  - kontrawariancja

```
void DoSomethingToAFrog(Action<Frog> action, Frog frog)
{
    action(frog);
}
...
Action<Animal> feed = animal = >animal.Feed();
DoSomethingToAFrog(feed, new Frog());
...
```

C# 4.0 (4)

- Task Parallel Library i PLINQ
  - Task

```
Action<object> action = (object obj) => {
            Console.WriteLine("Task={0}, obj={1}, Thread={2}", Task.CurrentId,
obj.ToString(), Thread.CurrentThread.ManagedThreadId); };
Task t3 = new Task(action, "gamma");
t3.RunSynchronously();
t3.Wait();
```

  - PLINQ

```
var source = Enumerable.Range(1, 10000);
var evenNums = from num in source.AsParallel()
               where Compute(num) > 0
               select num;
```

C# 5.0 (1)

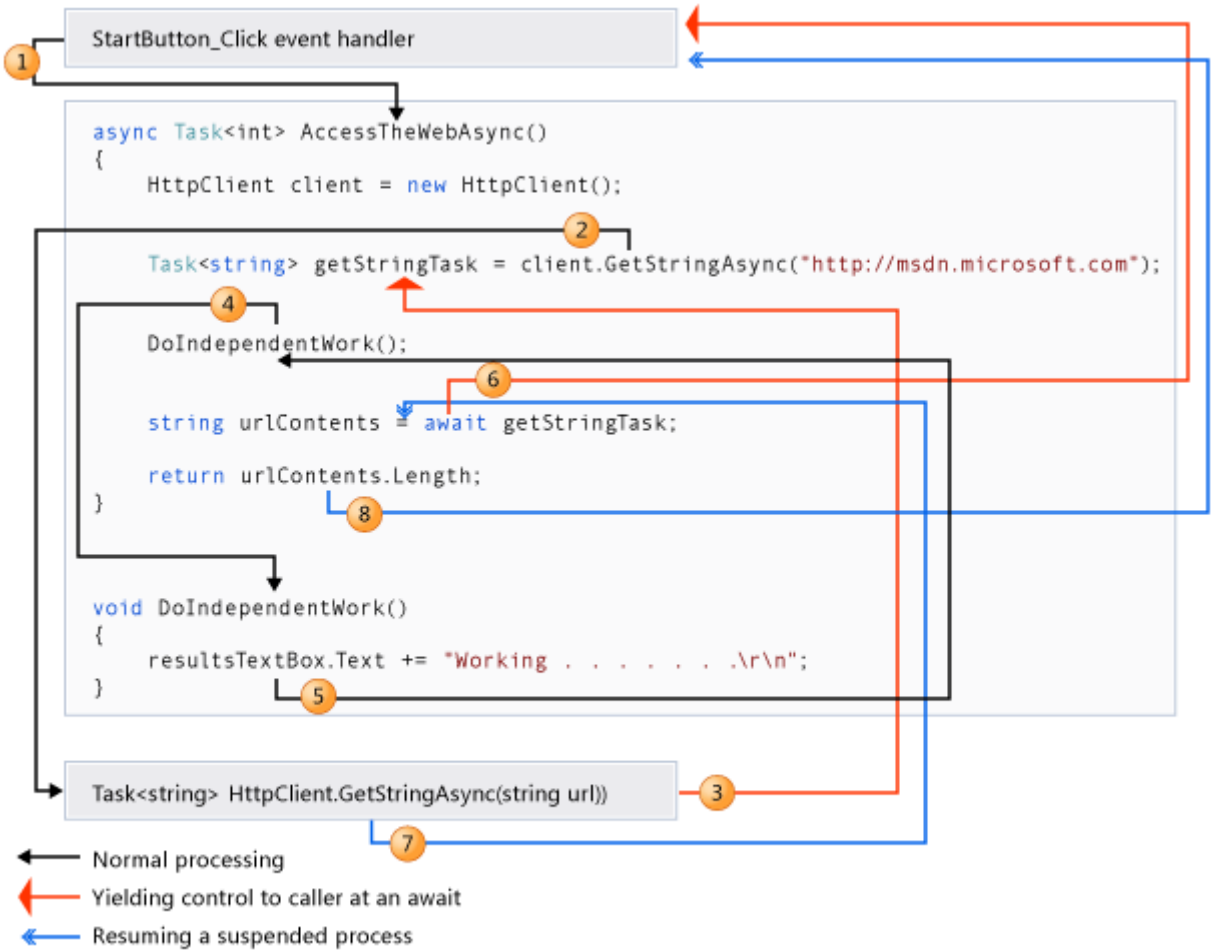
- Metody async (za msdn.microsoft.com)

```
async Task<int> AccessTheWebAsync()
{
    Task<byte[]> getURLTask = client.GetURLAsync("http://msdn.microsoft.com");
    DoIndependentWork();
    string urlContents = await getURLTask;
    return urlContents.Length;
}

private async Task<byte[]> GetURLAsync(string url)
{
    var content = new MemoryStream();
    var webReq = (HttpWebRequest)WebRequest.Create(url);
    using (WebResponse response = await webReq.GetResponseAsync())
    {
        using (Stream responseStream = response.GetResponseStream())
        {
            await responseStream.CopyToAsync(content);
        }
    }
    return content.ToArray();
}
```

C# 5.0 (2)

- Metody async (za msdn.microsoft.com)



C# 5.0 (3)

- Wiele mniejszych zmian dla programowania równoległego, np. TPL dataflow

```
var items = new BufferBlock<T>(
    new DataflowBlockOptions { BoundedCapacity = 100 });
Task.Run(async delegate
{
    while(true)
    {
        T item = Produce();
        await items.SendAsync(item);
    }
};

Task.Run(async delegate
{
    while(true)
    {
        T item = await items.ReceiveAsync();
        Process(item);
    }
});
```

Wątki i synchronizacja

- instrukcja *lock*, metody *synchronized*, kolekcje
- przestrzeń nazw *System.Threading*
  - *Thread*
  - *ThreadPool*
  - *Auto/ManualResetEvent*
  - *Interlocked*
  - *Mutex*
  - *Monitor*
  - *Semaphore*
  - *ReaderWriterLock*
  - *BackgroundWorker*

Thread

- tworzy i kontroluje nowy wątek
- można zgubić referencję bez zabicia wątku
- delegacje *ThreadStart, ParameterizedThreadStart*
- *Thread.CurrentThread*
- *Join (), Sleep (), VolitileRead (), VolitileWrite ()*
- *Interrupt (), Abort()* i wyjątki
- *ThreadState, IsBackground*

```
public static void ThreadProc() {...}
Thread t = new Thread(new ThreadStart(ThreadProc));
t.Start ();
```

ThreadPool

- często wątki przez dużą cześć czasu są w stanie *Wait*
- zbiór wątków, z których każdy monitoruje kilka operacji w stanie oczekiwania lub wykonuje krótkie zadania
- kiedy operacja staje się gotowa, 1 z wolnych wątków wykonuje funkcję zwrotną
- wszystkie wątki są *background*

```
ThreadPool.QueueUserWorkItem(new
  WaitCallback(ThreadProc), object);

ThreadPool.RegisterWaitForSingleObject (

    WaitHandle, WaitOrTimerCallback, state, timeout,

    executeOnlyOnce);
```

Auto/ManualResetEvent

- Zdarzenie do komunikacji między wątkami
- Stan zdarzenia – *zamknięte/otwarte,* stan jest pamiętany
- Wywołanie *WaitOne ()* – oczekiwanie na zdarzenie
- *AutoResetEvent* – wywołanie *Set ()* wypuszcza jeden wątek i wraca do *zamknięte* albo zostawia w *otwarte*
- *ManualResetEvent* – wywołanie *Set ()* wypuszcza wszystkie wątki i zostawia *otwarte* aż do *Reset ()*
- *WaitHandle.WaitAll (WaitHandle [])*
- *WaitHandle.WaitAny (WaitHandle [])*

ThreadPool – przykład

```
public void CallbackFunc (object o){
    StateInfo si = (StateInfo) o;
    Do Work (); }
public class StateInfo {...}
public void MainMethod () {
    ManualResetEvent evnt = new ManualResetEvent (false);
    WaitOrTimerCallback wotc = new WaitOrTimerCallback (CallbackFunc);
    StateInfo si = new StateInfo (...);
    int timeout = 100; // milliseconds
    bool onetime_exec = true;
    ThreadPool. RegisterWaitForSingleObject (evnt, wotc, si, timeout,
       onetime_exec);
    ...
    evnt.Set();
}
```

Mutex

- standardowy mechanizm zamka
- *WaitOne (), ReleaseMutex()*
- sprawdza tożsamość wątków
- *AbandonedMutexException*
- *ApplicationException*, jeśli obiekt nie był właścicielem wątku

```
Mutex m = new Mutex (true, ”nukem”);
m.WaitOne ();
LaunchNukes ();
m.ReleaseMutex ();
```

Monitor

- statyczna klasa
- wykonywany na typie **referencyjnym**
- *Enter, TryEnter, Exit* – wejście/wyjście z monitora
- *Wait* – opuszczenie sekcji krytycznej i zaśnięcie na monitorze
- *Pulse, PulseAll* – sygnały

```
private Object x;
Monitor.Enter(x);
try {}
finally { Monitor.Exit(x); }
```

ReaderWriterLock(Slim)

- Implementacja blokady dla problemu czytelników i pisarzy
- *Acquire/ReleaseReaderLock*
- *Acquire/ReleaseWriterLock*
- *LockCookie lc = UpgradeToWriterLock*
- *DowngradeFromWriterLock (ref LockCookie)*
- Wszystkie metody z jawnym timeout'em
- Szybsza wersja *Slim* – m.in. uproszczenie rekursji zamków

BackgroundWorker

- brak marshallingu do wątka GUI *(Form.Invoke)*
- wsparcie w VS Designer
- uruchomienie zadania
  `DoWork += bw_DoWork; bw.RunWorkerAsync ("Message to worker");`
- raportowanie postępu
  - zdarzenia `RunWorkerCompleted` i `ProgressChanged`
  - właściwość `WorkerReportsProgress = true`
  - wołanie `ReportProgress` z metody w `DoWork`
- anulowanie
  - właściwość `WorkerSupportsCancelation = true`
  - właściwość `CancellationPending` sprawdzana w metodzie z `DoWork`
  - ustawienie w zdarzeniu właściwości `Cancel = true i return`
  - wołanie `CancelAsync` żądaniem anulowania

Inne podejścia

- *Interlocked – Add (int, int), Increment (), Exchange ()*
- sekcje `lock (Object) {...}`
  - nie na *this, typeof (this), "string"*
- metody synchronizowane

```
using System.Runtime.CompilerServices;
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public void AsynchroMetod ()
```

- operacje na GUI

```
Form.Invoke (Delegate method, pars Object[] args)
```

Wątki i synchronizacja – Tasks

- przestrzeń nazw *System.Threading.Tasks*
  - *Parallel.For* i *Parallel.ForEach*
  - *Task* i *Task<T>*
  - *TaskFactory*
  - *Barrier*
  - *SpinLock*
- przestrzeń nazw *System.Collections.Concurrent*
  - *ConcurrentBag<T>*
  - *ConcurrentDictionary<TKey, Tvalue>*
  - *ConcurrentQueue<T>*
  - *ConcurrentStack<T>*

Task i *Task<T>*

- „lekki” wątek, wiele zadań może być uruchomionych w ramach 1 wątku (podobnie do *ThreadPool)*
- zarządzaniem zadaniami zajmuje się implementacja *TaskScheduler*
- opcje *PreferFairness*, *LongRunning* i *AttachedToParent*
- możliwość tworzenia zadań zagnieżdżonych i kontynuacji

```
var taskA = new Task(() => Console.WriteLine("Hello from taskA."));
taskA.Start();
var taskB = Task.Factory.StartNew(() => Console.WriteLine("Hello from
   taskB."));
Task<double> taskRetVal = Task<double>.Factory.StartNew(() =>
   DoComputation1());
Console.WriteLine (taskRetVal.Result);
```

Task i *Task<T>* (2)

- continuations

```
Task<string> reportData2 = Task.Factory.StartNew(() => GetFileData())
                           .ContinueWith((x) => Analyze(x.Result))
                           .ContinueWith((y) => Summarize(y.Result));
```

- zadania potomne

```
var parent = Task.Factory.StartNew(() => {
    Console.WriteLine("Parent task beginning.");
    var child = Task.Factory.StartNew(() => {
        Thread.SpinWait(5000000); Console.WriteLine("Child completed.");
    }, TaskCreationOptions.AttachedToParent);
});
parent.Wait();
Console.WriteLine("Parent task completed.");
```

Task i *Task<T>* (3)

- obsługa zadań
  - *void Wait ()* – rzuca wyjątki:
    - *ObjectDisposedException* – po *Dispose ()*
    - *AggregateException* – anulowanie lub wew. wyjątek
  - *static void WaitAll (params Task[] tasks)*
  - *static void WaitAny (params Task[] tasks)*
  - *static Task<Task> WhenAny(IEnumerable<Task> tasks)*
  - *static Task<Task> WhenAll(IEnumerable<Task> tasks)*
    - pozwala czekać na Task:
      await Task.WhenAll(tasks);

Task i *Task<T>* (4)

- wyjątki

```
var task1 = Task.Factory.StartNew(() =>
{
    throw new MyCustomException("I'm bad, but not too bad!");
});
Try {
    task1.Wait();
}
catch (AggregateException ae) {
    foreach (var e in ae.InnerExceptions)
    {
        if (e is MyCustomException)
```

Task i *Task<T>* (5)

- anulowanie i wyjątki

```
private static Int32 Sum(CancellationToken ct, Int32 n) {
    for (;;) {
        ct.ThrowIfCancellationRequested();
        …
    }
    return sum;
}
public static void Main() {
    CancellationTokenSource cts = new CancellationTokenSource();
    Task<int32> t = new Task<int32>(() => Sum(cts.Token, 1000), cts.Token);
    t.Start();
    cts.Cancel();
    try {
        // po anulowaniu Result rzuci AggregateException
        Console.WriteLine("The sum is: " + t.Result);
    } catch (AggregateException ae) {
        ae.Handle(e => e is OperationCanceledException);
        Console.WriteLine("Sum was canceled");
    }
}
}
```

async i Task/Task<T>

- przykłady *async*

```
public async Task<string> GetStringAsync(string a){
     await Task.Delay (3000);
     return "async: " + a;
}
public async Task<string> GetStringNotAsync(string a){
     Thread.Sleep (3000);
     return "notasync: " + a;
}
Console.WriteLine("Starting async");
Task<string> at = GetStringAsync("aaa");
Console.WriteLine("Called async, it is working!");
string s = await at;
Console.WriteLine(s); // czekamy
Console.WriteLine("Starting notasync");
Task<string> nat = GetStringNotAsync("nnn"); // czekamy
Console.WriteLine("Called notasync, working? not...");
string ns = await nat;
Console.WriteLine(ns);
Console.WriteLine(await GetStringAsync("ccc")); // czekamy
```

async i Task/Task<T> (2)

- porównanie

```
Task<int> TaskDivideAsync(int nominator, int denominator)
{
    Task<int> task = new Task<int>(() =>
    {
        Thread.Sleep(10000);
        return (nominator / denominator);
    });
    task.Start();
    return (task);
}

async static Task<int> TaskDivideAsync(int nominator, int denominator)
{
    await TaskEx.Run(() => Thread.Sleep(10000));
    return (nominator / denominator);
}
```