



Ще разгледаме:

- Генетични типове;
- Lambda expressions в C# и C++;
- Паралелни конструктори в C++;
- Асинхронни изчисления в стандарта на C++.



Generics (генетични типове)



- Целта е сходна на целите на ООП - algorithm reusing . Механизмът е въведен в CLR на .NET
- Реализациите да се отнасят за обекти от различен тип;
- Всичко може да е 'генетичен тип': 'генетичен референтен тип', 'генетичен стойностен тип', 'генетичен интерфейс' и 'генетичен делегат'. Разбира се и 'генетичен метод'.
- Нека създадем генетичен списък: `List<T>` (произнася се : List of Tee):

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
    public List();
    public void Add(T item);
    public void Sort( IComparer<T> comparer);
    public T[] ToArray(0);
    ....
    public Int32 Count {get;}
    ...
}
```

Обикновено се именуват с T
или Time (напр TKey)

- Нека използваме списъка:

```
private static void SomeMethod() {
    List<DateTime> dtList = new List<DateTime>();
```

```
dtList.Add(DateTime.Now);    //OK! Няма boxing
dtList.Add("2/2/2011");     //грешка при компилация
```



Предимства на генетичните (пораждащи) класове:

1. Разработчикът **не е нужно задължително да се притежава сорса** на генетичния алгоритъм (за разлика от C++ templates или Java generics) за да прекомпилира;

(При шаблоните, компилаторът генерира separate source-code functions (именована: **specializations**) при всяко отделно повикване на ф-ия шаблон или инстанция на шаблонизиран клас.)

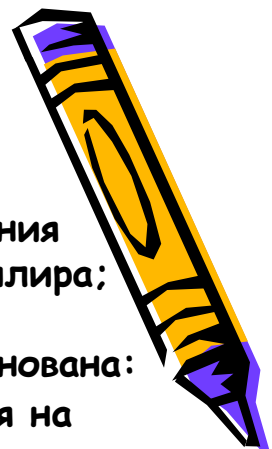
2. **Type safety**

3. **ясен код**: рядко се налагат type casts;

4. **Подобрена производителност**: преди генетиците, същото се постигаше с използване на Object типа. Това налага непрекъснато пакетиране (boxing), което изисква памет и ресурс, форсира често включване на с-мата за garbage collection. При генетичните алгоритми няма пакетиране. Това подобрява десетки пъти производителността.



При генетични реализации CLR средата генерира 'native code' за всеки метод, първият път когато методът се повика с указан тип данни. Това разбира се, увеличава размера на кода, но не намалява производителността





Microsoft препоръчва ползване на генетични класове от Framework Class Library (FCL) вместо не-генетичните им еквиваленти.

Съществува имплементация (на Wintellect) Power Collection library, която прехвърля класовете от старата Standard Template Library към CLR среда. Тя е free.

За да се поддържат генетични имплементации, към .NET се добавиха:

1. Нови IL инструкции, четящи конкретния тип на аргумента;
2. Метаданното описание се обогатява с описание на типа на параметрите;
3. Променя се синтаксисът на C#, Visual Basic и т.н.
4. Променят се компилаторите;
5. Променя се JIT компилаторът, така че да генерира 'native code' за всяко повикване с конкретен тип на аргумент.

..



Open & Closed тип

Тип с генетични параметри се нарича 'open type' тъй като не допуска CLR да конструира инстанции директно (така беше и при интерфейсите)

Когато кодът се обърне към генетичен тип, се подават реални параметри. Тогава типът се нарича вече 'closed type' и за него се прави инстанция.

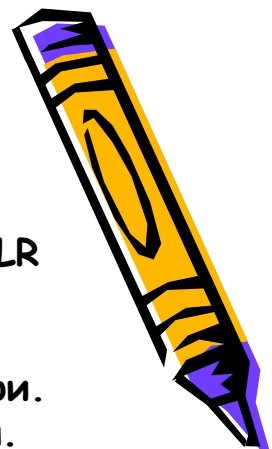
Генетични типове и наследяемост

Това си е нормален тип и наследяемост е напълно допустима. Пример:

```
internal sealed class Node<T> {  
    public T m_data;  
    public Node<T> m_next;  
  
    public Node(T data) : this(data,nul) { }  
    public Node(T data, Node<T> next)  
        { m_data = data; m_next = next; }  
    ....  
}
```

Използваме в произведен тип, наследил и ползващ горния Node<T>(..):

```
private static void SomeDataLinkedList() {  
    Node<Char> head = new Node<Char>('C');  
    head = new Node<Char>('B', head);  
    head = new Node<Char>('A', head);  
}
```



Преименоване на генетичен тип

С цел удобство, е честа практика:

ако имаме: `List<DateTime> dtl = new List<DateTime>();`

предedefинираме: `internal sealed class DateTimeList : List<DateTime> {}`

И тогава можем да създадем списък от генетичен тип по традиционния начин:
`DateTimeList dtl = new DateTimeList();`

Обработка на генетични типове: code explosion

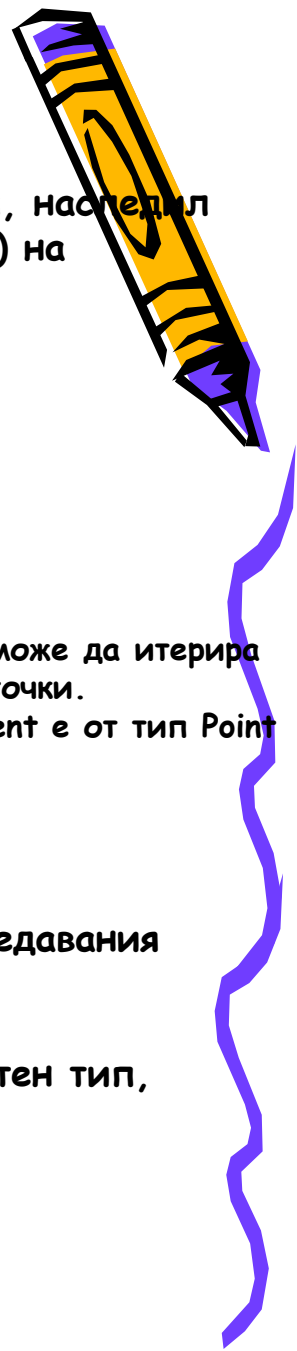
- При повикване на метод от генетичен тип, JIT компилаторът прави заместването и създава 'native code' за точно този метод с точно тези подадени параметри.
- CLR генерира native code за всяка комбинация **метод/тип**. Това води до 'code explosion'.
- Ако впоследствие, метод се повика със същия тип аргумент, не се генерира повторен код.
- Еднократно се генерира и код в случаите, когато аргументите са от референтен тип. Напр:

`List<String>`

`List<Stream>`

макар и аргументите всъщност да сочат съвсем различни неща.





Генетични интерфейси

Без поддръжка на генетични интерфейси, всеки път когато ще създаваме value тип, наследил не-генетичен интерфейс, трябва да се случи вътрешно пакетиране (преобразуване) на аргументите (boxing). Това е загуба на ресурс и бързодействие. Избягва се чрез Генетични интерфейси:

Ето един стандартен генетичен интерфейс:

```
public interface IEnumerator<T> : IDisposable, IEnumerator {  
    T Current { get; }  
}
```

Ето клас, който имплементира горния интерфейс над тип Point:

```
internal sealed class Triangle : IEnumerator<Point> {  
    private Point[] m_vertices;  
    public Point Current { get { ... } ... }  
}
```

Triangle обектът може да итерира през масива от точки. Пропъртието Current е от тип Point

Генетични делегати

CLR поддържа и генетични делегати за да ползва предимствата на генетичните предавания (type-safe, без непрекъснато пакетиране например към Object).

Тъй като делегат е всъщност дефиниран клас с няколко метода (това ще бъде пояснено в курса - Програмни Среди), когато се дефинира делегатен тип, компилаторът поражда съответните методи с реалния тип параметри





Още някои особености около генетиците:

-Имаме: **генетични методи** (всичко в тях е нормално)

-В **C#** имаме **properties, events, operator methods, constructors and finalizers**

които не могат да имат типови параметри (не могат да са генетични).

Такива , обаче, могат да се дефинират вътре в генетичен тип и тогава кодът им може да ползва типовите параметри на обхващания генетичен тип директно

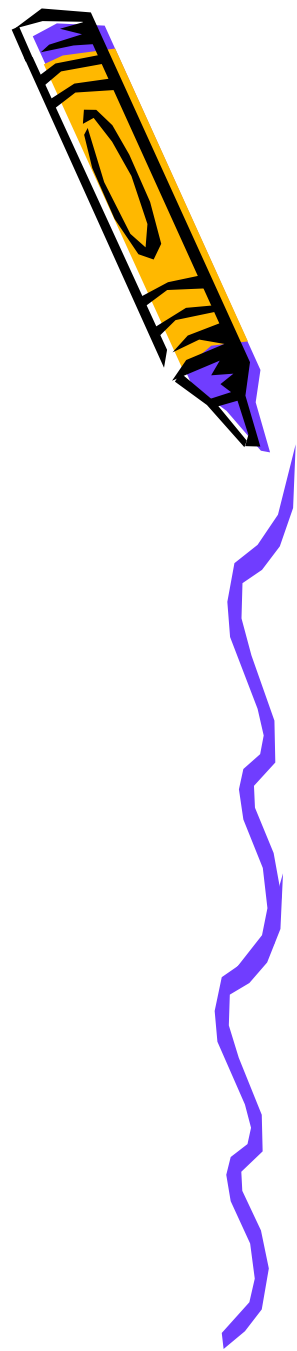
-**Ограничители** (в генетични типове) - терминът е: **constraints**

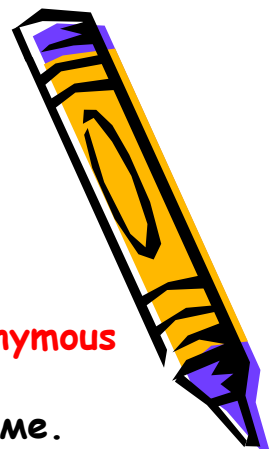
чрез тях може да се ограничи броя на типовете, които могат да са заместители в аргументите на генетичен тип:

```
public static T Min<T>(T o1, T o2) where T : IComparable
{
    if(o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```



Lambda Expressions





Lambda Expressions (синтаксис от C++ и C#)

Много от модерните езици поддържат концепцията за безименните функции (**anonymous function**).

1. безименната функция е обикновена функция, имаща тяло, но не и име.
2. **lambda expression** неявно дефинира клас и създава т.нар. **function object**, имащ типа на този клас.
3. може да считате, lambda expression за безименна функция **запазваща състояние** при достъп до променливи в обсега ѝ.

В софтуерната теория има **функции-обекти** и **функции през указатели**:

function pointers и **function objects** имат всяка своите предимства и недостатъци:

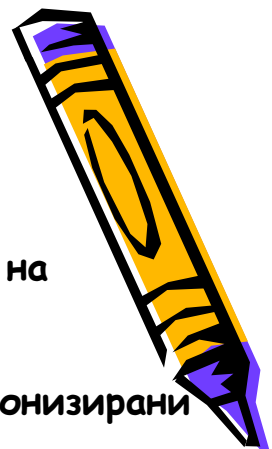
- **function pointers** налагат минимални синтактични добавки, но пък не позволяват запазване на състояние;
- **function objects** запазват състояние, но налагат синтактични усложнения, свързани с дефинирането на нов клас.

Lambda expressions е програмистка техника, комбинираща предимствата на function pointers и на function objects и същевременно - избягваща недостатъците им. Lambda expressions са по-гъвкави, запазват състоянието си - точно както и function objects, като при това техния компактен синтаксис премахва нуждата от явно дефиниране на клас - нещо, което се изисква от function objects



Малко теория за функциите - обекти:

function object, или още известна като **functor**, е всеки тип, поддържащ оператор **operator()** - наричан от своя страна **"call operator"**.



Function objects при употреба имат 2 предимства спрямо стандартните повиквания на функции.

1. **function object** запазва състоянието си.
2. **function object** е тип и следователно може да се ползва като параметър в шаблонизирани (template) повиквания.

За да създадем **function object**, първо създаваме типа и след това имплементираме **operator()**:

```
class Functor
{
public:
    int operator()(int a, int b)
    {
        return a < b;
    }
};

int main()
{
    Functor f;
    int a = 5;
    int b = 7;
    int ans = f(a, b);
}
```



Последният ред на **main()** показва как се вика **function object**. Повикването изглежда като повикване на обикновена функция, но всъщност е повикване на **operator()** на типа - **Functor**

Т.е.:

lambda expression са анонимни функции, с добро приложение при създаване на делегати, обработчици на събития и някои специални типове (напр. expression tree).

Lambda expressions са добри и при създаване на LINQ заявки.

При създаване на lambda expression, вие следва да специфицирате (C#) input parameters (ако има такива) от лявата страна на lambda оператора \Rightarrow , и да опишете израза или блока оператори от другата страна.

Например, lambda expression $x \Rightarrow x * x$

указва , че параметър x е входен и връщаната стойност ще бъде за x на квадрат. Този израз може да се присвои на делегатен тип, например така:

```
delegate int del(int i);  
static void Main(string[] args)  
{ del myDelegate = x => x * x;  
  int j = myDelegate(5);           //j = 25  
}
```



Lambdas, както споменахме , могат да се ползват и като аргумент към стандартна LINQ заявка, минаваща през метод
(напр. Where<TSource> на изброим тип):

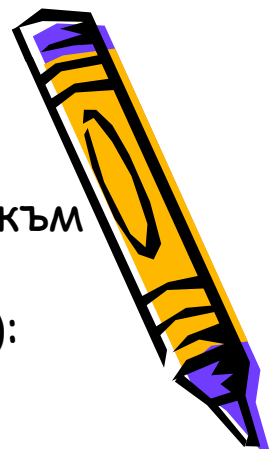
```
List<string> fruits =  
    new List<string> { "apple", "passionfruit", "banana", "mango",  
                      "orange", "blueberry", "grape", "strawberry"  
    };
```

```
IEnumerable<string> query = fruits.Where(fruit => fruit.Length < 6);
```

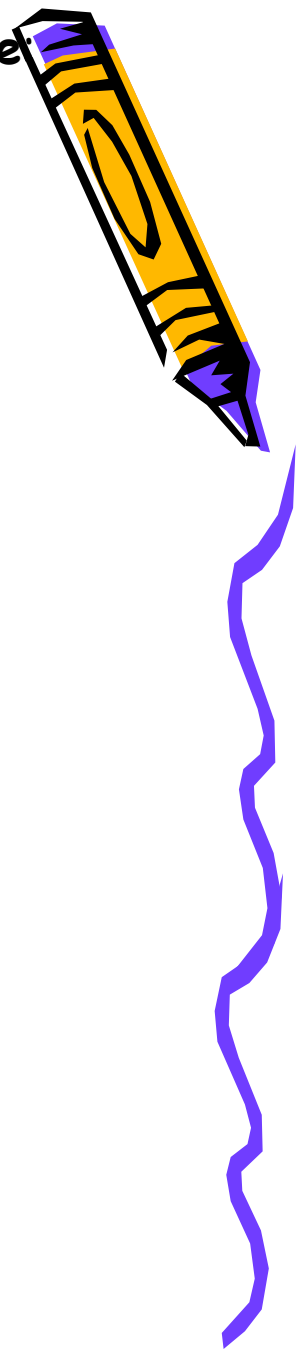
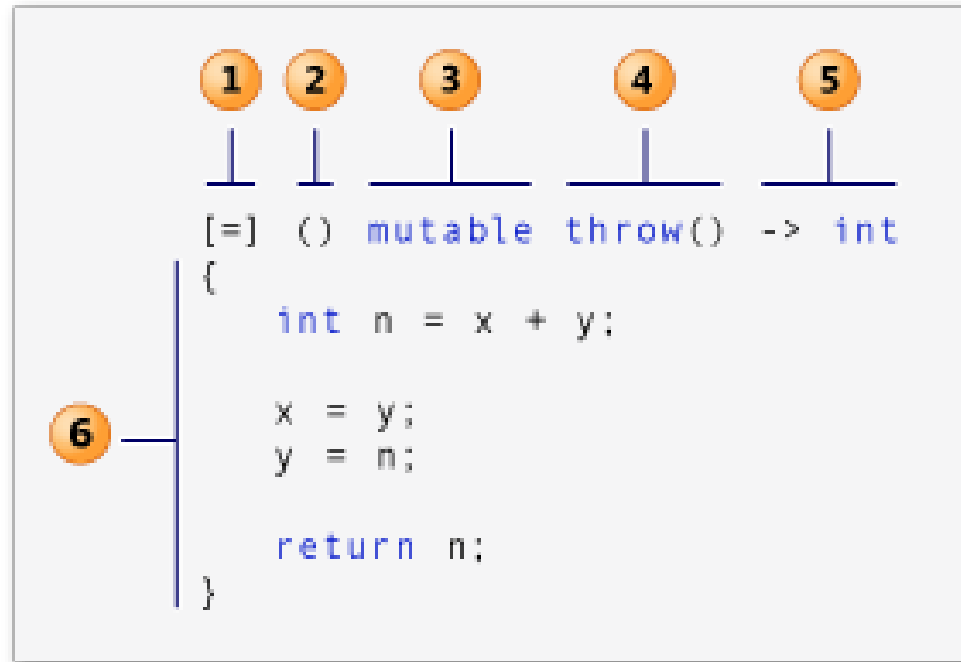
```
foreach (string fruit in query)  
    { Console.WriteLine(fruit);}
```

This code produces the following output:

```
apple  
mango  
grape
```



Сега да се върнем към формалното описание за lambda изразите
Синтаксис на Lambda Expression (C++)



- 1. **lambda-introducer** (наричан още **capture clause**)
- 2. **lambda-parameter-declaration-list** (наричан **parameter list**)
- 3. **mutable-specification**
- 4. **exception-specification**
- 5. **lambda-return-type-clause** (**return type**)
- 6. **compound-statement** (**lambda body**)



Capture Clause

lambda expression могат да достъпват променливи по стойност или по референция:

променливите предхождани от ampersand (&) се достъпват по референция, а променливите , нямащи префикс & - по стойност.

Ако началния оператор - capture clause [] - е празен, това индикира, че в тялото на lambda expression не се изпълнява достъп до променливи.

Ако укажете default capture mode - например чрез

& или = в първи елемент на capture clause, то:

& означава , че всички променливи в тялото се ползват по reference;

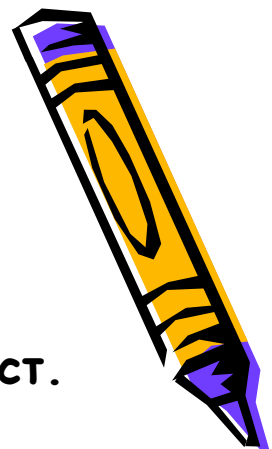
= указва, че в тялото на lambda expression, достъпът до променливите е по стойност.

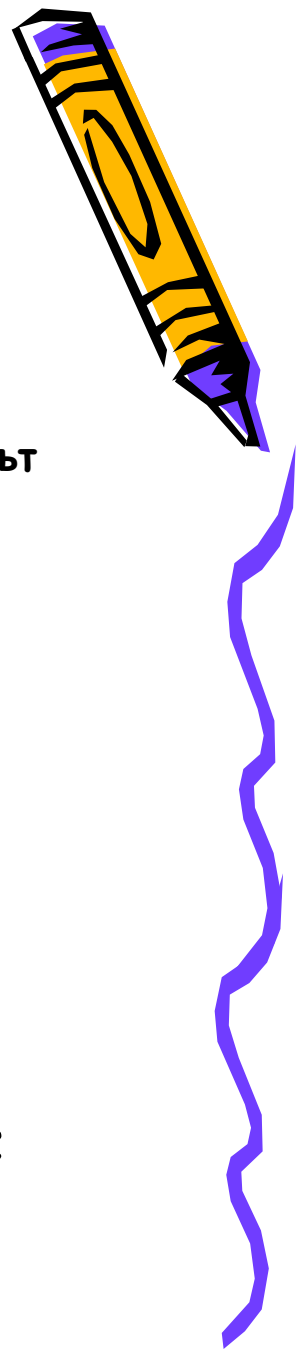
Например, ако в тялото на lambda expression външната променлива `total` се ползва по reference, докато външната променлива `factor` - по стойност, това е указано чрез следните (еквивалентни) capture clauses:

[&total, factor]

[&, factor]

[=, &total]





Списък параметри

Списъкът параметри на `lambda expression` изглежда така, както и списъкът параметри на функция, със следните отлики:

- Списъкът параметри няма аргументи по подразбиране.
- Списъкът параметри не може да е с променлив брой.
- Списъкът параметри не съдържа безименни параметри.

списъкът параметри на `lambda expression` е опционен.

Пример :

```
int z = [=] { return x + y; }();
```



Спецификация „Mutable“

служебната дума „mutable“ разрешава в тялото на `lambda expression` да се извършват промени над променливите, ползвани по стойност



Специфицирайки : **throw()** **указвате, че**

`lambda expression` не може да генерира exceptions

Частта - **return** на `lambda expression` наподобява описанието на `return` типа на всеки метод или функция. Синтактично, секцията „`return type`“ следва списъка параметри и се предхожда от **->**

```
int main()
{ int m = 0, n = 0;
  [&, n] (int a) { m = ++n + a; }(4);           // подаденият параметър е един, int и е = 4
  cout << m << endl << n << endl;
}
```

Примерът извежда следния резултат на конзолата:

5
0

Тъй като променливата **n** е ползвана по стойност, нейната начална стойност 0 остава непроменена след повикването на `lambda expression`



Синтактични разлики при lambda Expressions в C#



И тук lambda expression е анонимна функция, която може да съдържа в себе си изрази и оператори.

Тя служи **за създаване и на делегати (delegate).**

lambda operator тук се пише:

=> и се произнася **"goes to"**.

В **лявата част** на lambda operator са специфицирани input параметрите (ако ги има) и в **дясната част** са изразите или блока оператори.

lambda expression:

x => x * x

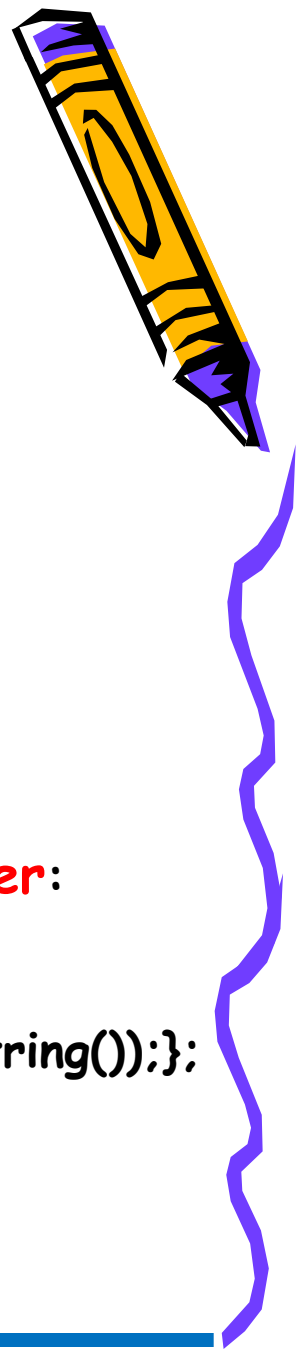
се чете: "x goes to x пъти по x."

изразът може да се присвои на делегатен тип така:



Спомнете си примера:

```
delegate int del(int i);  
static void Main(string[] args)  
{ del myDelegate = x => x * x;  
  int j = myDelegate(5); //j = 25 }
```



друго използване на lambda expression:

// ползване на lambda expression за дефиниране на **event handler**:

```
this.Click +=  
(s, e) => { MessageBox.Show(((MouseEventArgs)e).Location.ToString());};
```



Lambda оператор (body)

lambda операторът прилича на expression lambda, с отликата , че е заграден в {}, т.е. позволява блокиране:

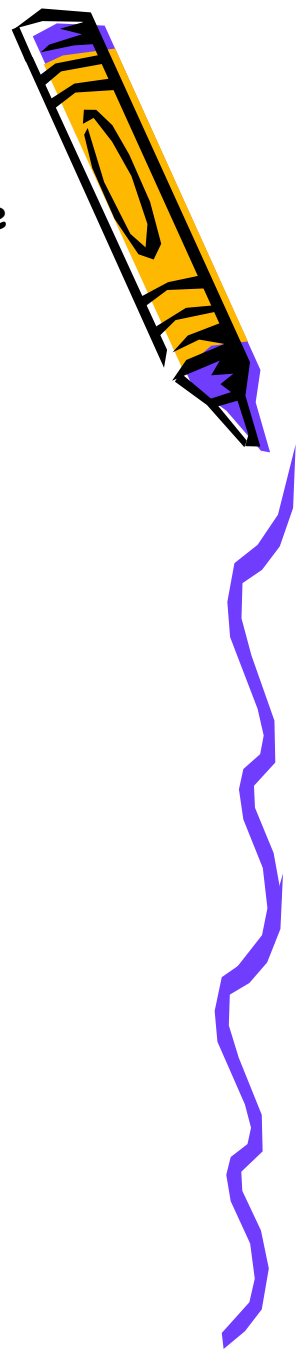
(input parameters) => {statement;}

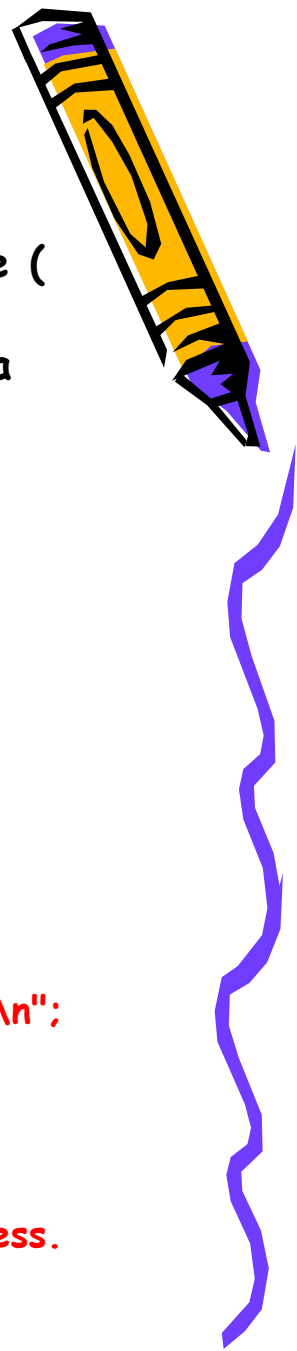
Или примерът:

```
delegate void TestDelegate(string s);
```

```
...  
TestDelegate myDel = n => { string s = n + " " + "World";  
                           Console.WriteLine(s);  
                           };
```

```
myDel("Hello");
```





Lambda в асинхронни изчисления

lambda expressions и операторите лесно се ползват в асинхронни блокове (където има async и await – ще ги разгледаме след малко).

Следва пример с Windows Forms event handler, който вика async метод, а handler седи в очакване на резултат през await (без ламбда..):

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        // ExampleMethodAsync returns a Task.
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\r\n";
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

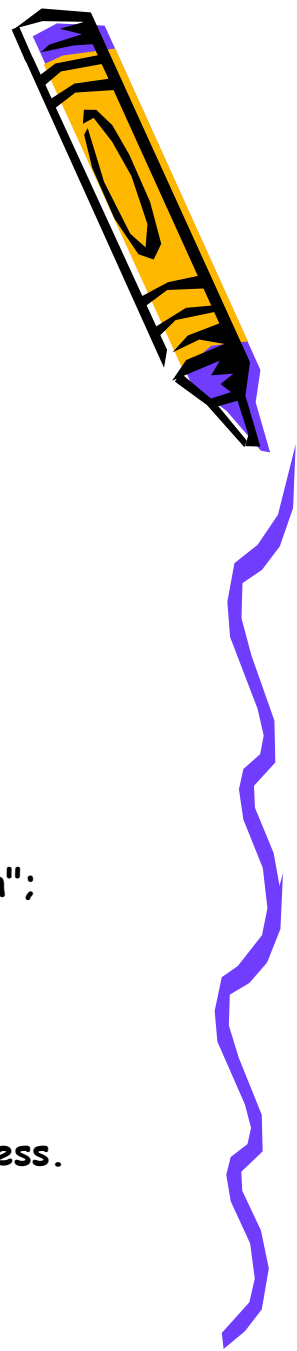


Сега същият **event handler** е викан през **async lambda**.

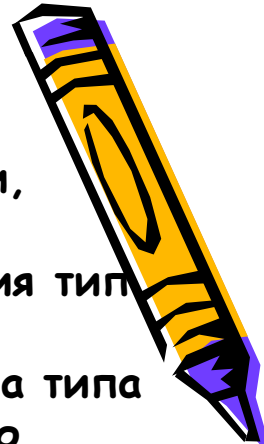
За да стане това, добавяме **async** модификатора преди списъка с **lambda** параметри:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            // ExampleMethodAsync returns a Task.
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\r\n";
        };
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```



Решаване на проблема за типа при lambda израз



Често при описание на lambdas, не указвате тип на входните параметри, защото компилаторът може да определи типа динамично: през операторите в тялото, типа на параметрите от описанието на делегатния тип или други източници.

- Например, за query operators, input параметъра има тип, съответен на типа на източника. Т.е. ако заявката се отнася за `IEnumerable<Customer>`, то типа на input променливата следва да е наследник на Customer object. Това означава, че имате достъп до методите и properties на Customer:

`customers.Where(c => c.City == "London");`

Правилата за типа на lambdas параметрите, ползвани в делегатен тип са:

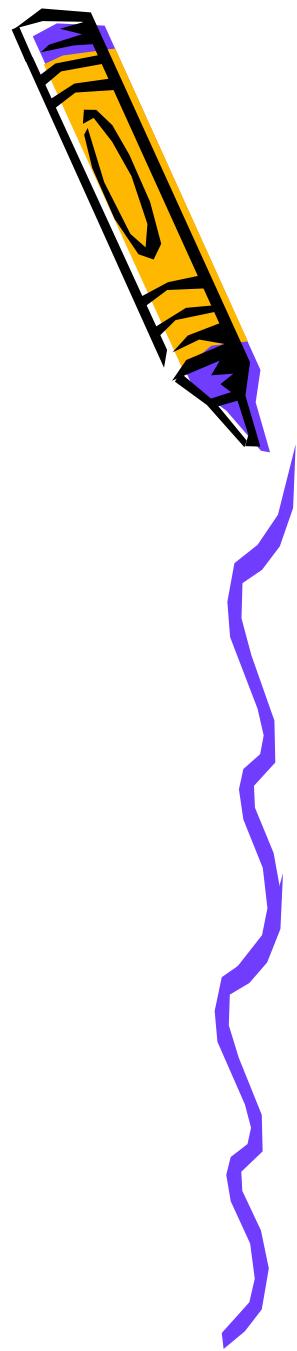
- lambda следва да съдържа същия брой параметри като делегатния тип.
- Всеки input parameter в lambda да е implicitly convertible към съответстващия делегатен параметър.
- Връщаната стойност от lambda (ако има такава) следва да е implicitly convertible към делегатния възвратен тип.

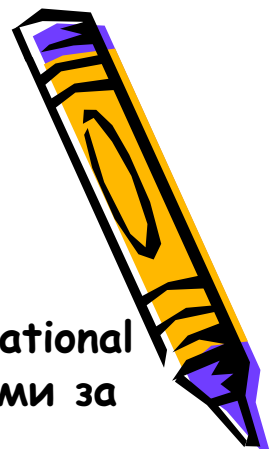
Изобщо, lambda expressions нямат тип и такъв не е заложен в общата система за типовете. Все пак, ако се говори за „тип“ на lambda expression,

става дума за делегатния тип или за типа на израза, към който lambda expression ще се преобразува.



Стандартизирани във Visual C++
средства за паралелизация
(Concurrency Features)





От версия **C++11**, в описанието на стандарта на C++ (приет от International Organization for Standardization (ISO) през 2011) се въвеждат формализми за паралелизация - concurrency.

Паралелизация на приложения се извършва и до този момент от разработчиците на C++ приложения, но чрез third-party library или директно чрез APIs на операционната среда.

Ще представим последователно:

Асинхронни задачи: това са части на общия алгоритъм, свързани помежду си чрез данните, които консумират или продуцират.

Нишки (threads): изпълними блокове от кода, които се администрират като една единица от runtime средата. Те са свързани с tasks в смисъл, че tasks се изпълняват посредством една или множество нишки (threads).

Thread internals: променливи в рамките на thread, генерирани от нишки exceptions обекти и т.н.



1. Асинхронни задачи

Ето чисто последователен код:

```
int a, b, c;
int calculateA()      { return a+a*b; }
int calculateB()      { return a*(a+a*(a+1)); }
int calculateC()      { return b*(b+1)-b; }
int main(int argc, char *argv[])
{  getUserData(); // initializes a and b
  c = calculateA() * (calculateB() + calculateC());
  showResult();
}
```

Функцията `main` кара потребителя да въведе някакви данни и ги подава за обработка на 3 функции: `calculateA`, `calculateB` and `calculateC`. Изчислените резултати се ползват за да се формира крайния изход.

Нека предположим, че трите изчисляващи функции въвеждат закъснение между 1 и 3 секунди всяка. При положение, че те се изпълняват строго последователно, това би въвело общо закъснение до 9 секунди.

Тези функции са независими една от друга - биха могли да се запуснат и в паралел.
Говорим за **async** финкции:



```
int main(int argc, char *argv[])
```

```
{    getUserData();  
    future<int> f1 = async(calculateB), f2 = async(calculateC);  
    c = calculateA() * (f1.get() + f2.get());  
    showResult();  
}
```

Въвеждаме 2 нови концепции: **async** и **future**. И двете са дефинирани в `std namespace`.

Декларацията **future** се отнася към функция, lambda или function object (functor) и декларира място завръщане на резултат в бъдеще.

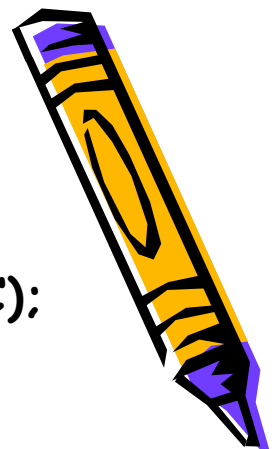
Може да се счита, че `future` описва мястото на евентуален бъдещ резултат.

Кой е този резултат?

- изработеният от асинхронно повикваната функция.

В някакъв момент, ще ни е нужен резултатът от трите, паралелно извикани функции. С повикване на метода **get()** към всеки `future`, изчисленията се блокират, до получаване на стойността.

Общото закъснение в този случай е не по-голямо от 3 секунди.



2. Нишки (Threads)

Асинхронният модел, представен по-горе, върши добра работа. Но в някои сценарии, сигурно бихте искали да имате по-голям контрол и възможности за намеса.

C++ след версия 11 стандартизира класовете - **thread**, декларирани в `<thread>` header в namespace - `std`.

Threads (нишки) предоставят по-добри методи за синхронизация и координиране, включително - предоставяне на изчислителен ресурс към друга нишка и изчакване за определено време или докато завърши дадена обработка.

в следващия пример,
имаме **lambda** функция, на която се подава `int` аргумент. Тя отпечатва на конзола кратните на аргумента, които са `< 100,000`:





```
auto multiple_finder = [](int n) {  
    for (int i = 0; i < 100000; i++)  
        if (i%n==0)  
            cout << i << " is a multiple of " << n << endl;  
};
```

```
int main(int argc, char *argv[])  
{  
    thread th(multiple_finder, 23456);  
    multiple_finder(34567);  
    th.join();  
}
```

синхронизира с
текущата нишка

По принцип, на нишката подаваме: lambda; или функция или functor.

В main сме стартирали функцията в 2 нишки, с подадени различни параметри. Да погледнем към резултата от работата ѝ (последователностите на активиране сигурно ще варират при различни пускания):

```
0 is a multiple of 23456  
0 is a multiple of 34567  
23456 is a multiple of 23456  
34567 is a multiple of 34567  
46912 is a multiple of 23456  
69134 is a multiple of 34567  
70368 is a multiple of 23456  
93824 is a multiple of 23456
```



Можем да преработим примера от предната секция (този с `asynchronous` задачи) и сега да работи с нишки. За целта въвеждаме нова концепция: **promise**.

Декларацията **promise** може да се разглежда като декларираща обект, в който резултатът ще се постави,но когато стане наличен (затова говорим за `sink` обект).

Това е като временно място за съхранение.

Използването на този резултат, ще стане в асоциирания с **promise** - **future**.




Кодът по-долу, запуска 3 нишки (вместо асинхронните tasks преди) с използване на promises и във всяка от тях се изчислява междинен резултат (calculateA(), calculateB() или calculateC()).

```
typedef int (*calculate)(void);
```

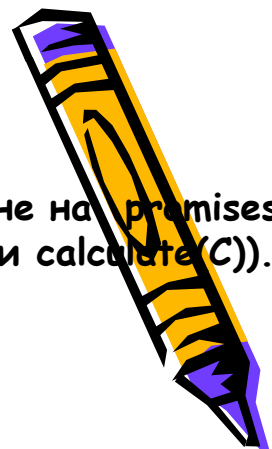
```
void func2promise(calculate f, promise<int> &p)  
{ p.set_value(f()); }
```

```
main(int argc, char *argv[])  
{  
    getUserData();  
    promise<int> p1, p2;  
    future<int> f1 = p1.get_future(), f2 = p2.get_future();  
    thread t1(&func2promise, calculateB, std::ref(p1)),  
            t2(&func2promise, calculateC, std::ref(p2));  
    c = (calculateA() + f1.get()) * f2.get();
```

```
    t1.join(); t2.join();  
    showResult();  
}
```



Име на асоциирана с
нишката функция + 2-та
параметър на
повикването ѝ





3. Променливи и Exceptions в рамките на нишка

В C++ можете да дефинирате **глобални променливи**, чиито обseg в ограничен в рамките на цялото приложение (включващо и нишките му). Но що се отнася до нишките, имате възможност да **дефинирате тези global променливи и така, че всяка нишка да пази тяхно свое копие**.

Тази концепция е за т.нар. **thread local storage** и се декларира така:

```
thread_local int subtotal = 0;
```

При такава декларация, в обсега на функцията (която ще поражда нишки), видимостта на променливата е свита до тялото на функцията, като обаче, всяка нишка поддържа свое

„static“ копие на променливата.

За момента, декларацията: ***thread_local*** не е налична за Visual C++. Следва да се дефинира спецификация по следния начин:

```
#define thread_local __declspec(thread)
```





Синхронизиране при паралелно изпълнение

Прекрасно би било, ако приложенията можеха да се разцепят на 100% независими, асинхронни задачи (asynchronous tasks). На практика, това едва ли е достижимо, поне що се отнася за данни, които всички обработват в паралел.

Ето защо се налага въвеждане на C++ примитиви, за избягване на състезания (**race conditions**) между тях.

Atomic types:

атомарните типове са подобни на примитивните типове, но позволяват thread-safe модификации.

Mutexes и заключвания:

конструктори, позволяващи оформяне на thread-safe критични секции.

условни променливи:

декларации, позволяващи блокиране на нишка, докато се удовлетвори даден критерий.





Атомарни типове

В header файла - `<atomic>` са въведени няколко примитивни типа:

`atomic_char`, `atomic_int` и др.

— дефинирани да избягват взаимни блокировки. Следователно, тези типове са еквивалентни на синонимните си, без `atomic_` prefix, но с тази разлика, че

дефинициите им на оператори за присвояване (`==`, `++`, `--`, `+=`, `*=` и др.) са защитени от състезания (race conditions).





В долния пример, имаме 2 паралелни нишки (едната е main) , работещи с различни елементи на общ вектор, както и с общи променливи:

```
atomic_uint total_iterations;
vector<unsigned> v;
int main(int argc, char *argv[])
{ total_iterations = 0; v.scramble_vector(1000);
  thread th(find_element, 0);
  find_element(100);
  th.join();
  cout << total_iterations << " total iterations." << endl;
}
```

Открива
първо
срещане на
ел в обсега,
отговарящ на
условието

void find_element(unsigned element)

```
{ unsigned iterations = 0;
  find_if(begin(v), end(v), [=, &iterations](const unsigned i) -> bool {
    ++iterations;
    return (i==element);
  });
```

```
total_iterations+= iterations;
cout << "Thread #" << this_thread::get_id() << ": found after " <<
iterations << " iterations." << endl;
}
```



Mut(ual) Ex(clusion) и заключвания



В header файла - <mutex> са дефинирани набор от заключващи класове, предназначени за **critical секции**.

Така че, можете да дефинирате **mutex**, оформящ критична секция, обхващаща набор от функции или методи. Само една нишка може да получи достъп до **mutex** и да го заключи.

Нишка, опитваща да заключи **mutex**, остава блокирана докато **mutex** стане достъпен или пропада. В средата на тези 2 крайности, е алтернативата - класът **timed_mutex**, който отнесен към **mutex** позволява блокировката му за определен интервал, преди окончателно пропадане.

Обектът - заключен **mutex** следва явно **да се отключи (unlock)**, така че други да могат да го ползват (да го заключат -lock). Пропуск в тази последователност води до непредвидимо поведение на приложението— което понякога трудно се открива, подобно на изтичането на памет. Пропускане да се освободи заключен **mutex** може да блокира други и цялото приложение да не функционира нормално.

За щастие, C++ locking класовете, които работят с **mutex** имат код в **destructor, който освобождава mutex ако е бил заключен.**



показаният код дефинира **critical секция с помощта на mutex mx:**

mutex mx;

void funcA();

void funcB();

int main()

{ thread th(funcA); funcB();

th.join();

}

Горният mutex се използва за да гарантира, че двете функции (funcA и funcB) работят паралелно , без да влизат едновременно в критичната секция.

Функцията funcA искаме да изчака, ако е нужно, за да влезе в критичната секция. За да постигнем това, използваме най-простия заключващ механизъм - **lock_guard**:

void funcA()

{ for (int i = 0; i<3; ++i)

{ this_thread::sleep_for(chrono::seconds(1));

cout << this_thread::get_id() << ": locking with wait... " << endl;

lock_guard<mutex> lg(mx);

... // Do something in the critical region. Print "lock secured"

cout << this_thread::get_id() << ": releasing lock." << endl;

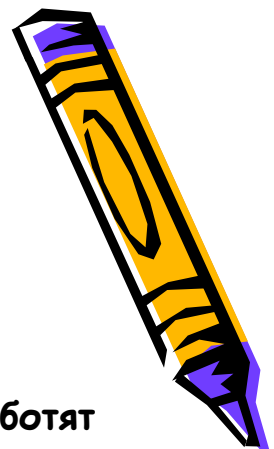
}

}

По начина по който е дефинирана, funcA ще достъпва critical region 3 пъти.

Функция funcB от своя страна, ще опита достъп, но ако mutex е вече заключен, тя ще изчака 1 секунда и отново ще опита достъп до критичната секция.

Описаният механизъм се нарича **unique_lock** , изпълняващ зададена политика („policy“):
try_to_lock_t



void funcB()

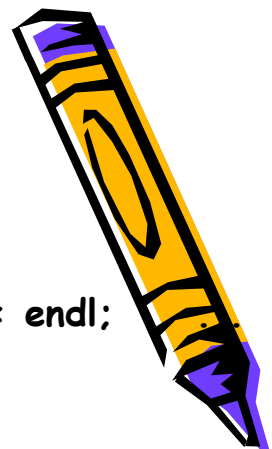
```
{ int successful_attempts = 0;
  for (int i = 0; i<5; ++i)
  { unique_lock<mutex> ul(mx, try_to_lock_t());
    if (ul)
    { ++successful_attempts;
      cout << this_thread::get_id() << ": lock attempt successful." << endl;
      // Do something in the critical region
      cout << this_thread::get_id() << ": releasing lock." << endl;
    } else {
      cout << this_thread::get_id() <<
        ": lock attempt unsuccessful. Hibernating..." << endl;
      this_thread::sleep_for(chrono::seconds(1));
    }
  }
  cout << this_thread::get_id() << ": " << successful_attempts
    << " successful attempts." << endl;
}
```

По начина по който е дефинирана, funcB ще се пробва до 5 пъти да влезе в critical region.

Показани са резултати от изпълнението.

funcB ще прави 5 опита и ще получи достъп до critical секцията двукратно.

funcB: lock attempt successful.
funcA: locking with wait ...
funcB: releasing lock.
funcA: lock secured ...
funcB: lock attempt unsuccessful.
Hibernating ...
funcA: releasing lock.
funcB: lock attempt successful.
funcA: locking with wait ...
funcB: releasing lock.
funcA: lock secured ...
funcB: lock attempt unsuccessful.
Hibernating ...
funcB: lock attempt unsuccessful.
Hibernating ...
funcA: releasing lock.
funcB: 2 successful attempts.
funcA: locking with wait ...
funcA: lock secured ...
funcA: releasing lock.



Условни променливи

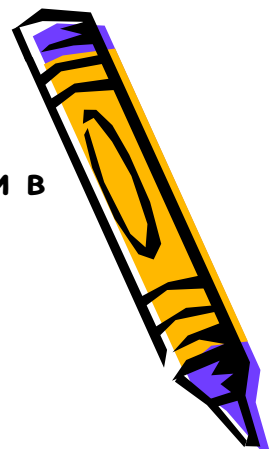
В header файла - `<condition_variable>` са дефинирани и някои примитиви, нужни в случаите, когато координацията между нишките минава през **events**.

Функцията - `producer` вкарва елемент в опашка:

```
mutex mq;  
condition_variable cv;  
queue<int> q;  
void producer()  
{ for (int i = 0; i < 3; ++i) {    ... // Produce element  
    cout << "Producer: element " << i << " queued." << endl;  
    mq.lock(); q.push(i); mq.unlock();  
    cv.notify_all();  
}  
}
```

Стандартната реализация на опашка не е thread-safe, така че вие следва да се подсигурите, че друг не я ползва в момента (например, ползвател не извлича елемент) по време на queuing.

Ако функцията - `consumer` опита да извлече елемент от опашката, а такъв няма. то тя просто чака известно време за изпълнено условие (condition variable) преди да опита повторно, като след 2 последователни неуспешни опита приключва:



void consumer()

```
{ unique_lock<mutex> l(mq);
```

```
  int failed_attempts = 0;
```

```
  while (true) {
```

```
    mq.lock();
```

```
    if (q.size())
```

```
    { int elem = q.front();
```

```
      q.pop();
```

```
      mq.unlock();
```

```
      failed_attempts = 0;
```

```
      cout << "Consumer: fetching " << elem << " from queue." << endl;
```

```
      ... // Consume element
```

```
    } else {
```

```
      mq.unlock();
```

```
      if (++failed_attempts>1)
```

```
      {
```

```
      cout << "Consumer: two consecutive failed attempts -> Exiting." << endl;
```

```
      break;
```

```
    } else {
```

```
      cout << "Consumer: queue not ready -> going to sleep." << endl;
```

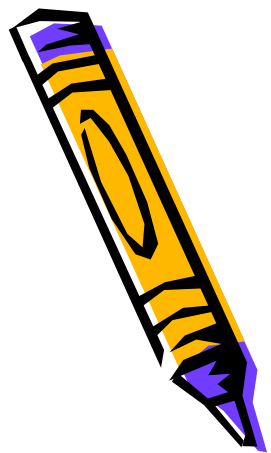
```
      cv.wait_for(l, chrono::seconds(5));
```

```
    }
```

```
  }
```

```
}
```

```
}
```

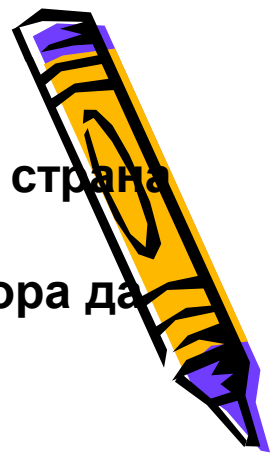


Функцията-консуматор ще бъде активирана чрез **notify_all** , от страна на producer винаги, когато има нов елемент.

По този начин, активната страна (producer) не оставя консуматора да е латентен за целия интервал, ако има наличен елемент.

Резултати на изхода:

Consumer: queue not ready -> going to sleep.
Producer: element 0 queued.
Consumer: fetching 0 from queue.
Consumer: queue not ready -> going to sleep.
Producer: element 1 queued.
Consumer: fetching 1 from queue.
Consumer: queue not ready -> going to sleep.
Producer: element 2 queued.
Producer: element 3 queued.
Consumer: fetching 2 from queue.
Producer: element 4 queued.
Consumer: fetching 3 from queue.
Consumer: fetching 4 from queue.
Consumer: queue not ready -> going to sleep.
Consumer: two consecutive failed attempts -> Exiting



4. Нови подобрения в асинхронното управление в .NET 4.5



Подобрения в асинхронния Debugging механизъм

Два проблема са съществени при дебъгване на асинхронен код:

“Как да дебъгваме асинхронен метод?”

и

“Какво е моментното състояние на задачите (tasks) в приложението?”

Visual Studio 2013 въвежда подобрения в т.нар. **“Call Stack”** и в **„Tasks windows”** с цел решаване горните проблеми по елегантен начин. Тези подобрения се поддържат за desktop, Web и мобилни - Windows Store apps след Windows 8.1 и са на разположение на C++ и JavaScript програмистите.

Фигура 1 демонстрира пример с асинхронен код.

Фигури 2 и 3 показват различията в call stack за Visual Studio 2012 и Visual Studio 2013 (за същия код).

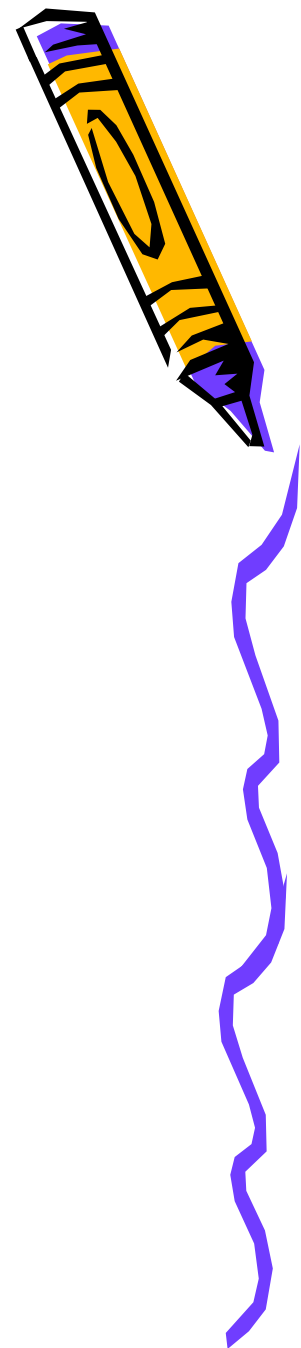


Фигура 1: пример с асинхронен код

```
private async void ShowSampleImg_Click(object sender, RoutedEventArgs e)
{
    string imgUrl = "http://example.com/sample.jpg";
    BitmapImage bitmap = new BitmapImage();
    bitmap.BeginInit();
    bitmap.StreamSource = await GetSampleImgMemStream(imgUrl);
    bitmap.EndInit();
    sampleImg.Source = bitmap;
}
```

```
private async Task<MemoryStream> GetSampleImgMemStream(string srcUri)
{
    Stream stream = await GetSampleImage(srcUri);
    var memStream = new MemoryStream();
    await stream.CopyToAsync(memStream);
    memStream.Position = 0;
    return memStream;
}
```

```
private async Task<Stream> GetSampleImage(string srcUri)
{
    HttpClient client = new HttpClient();
    Stream stream = await client.GetStreamAsync(srcUri);
    return stream;
}
```



Фигура 2: Visual Studio 2012 Call Stack Window



Call Stack	
Name	Language
SampleApp.exe!SampleApp.MainWindow.GetSampleImage(string srcUri) Line 50	C#
[Resuming Async Method]	
[External Code]	

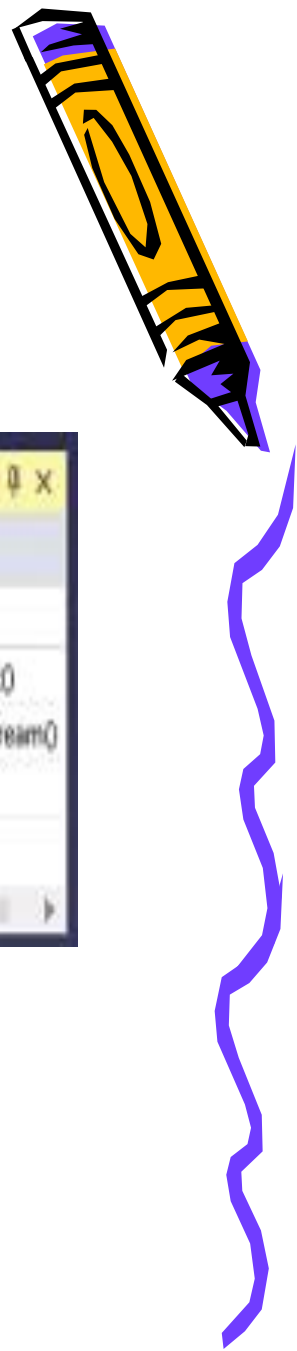
Call Stack	
Name	Language
SampleApp.exe!SampleApp.MainWindow.GetSampleImage(string srcUri) Line 50	C#
[Resuming Async Method]	
[External Code]	
[Async Call]	
SampleApp.exe!SampleApp.MainWindow.GetSampleImageStream(string srcUri) Line 40	C#
[Async Call]	
SampleApp.exe!SampleApp.MainWindow.ShowSampleImage_Click(object sender, System.Windows.RoutedEventArgs e) Line 34	C#



Фигура 3: Visual Studio 2013 Call Stack Window



Фигура 4: Visual Studio 2013 Tasks Window



Tasks						
	ID	Status	Start Time...	Duration (...)	Task	Location
▼	41	⌚ Awaiting	17.449	3.620	SampleApp.MainWindow.ShowSampleImg_Click()	SampleApp.MainWindow.ShowSampleImg_Click()
▼	40	⌚ Awaiting	17.447	3.622	SampleApp.MainWindow.GetSampleImgMemStream()	SampleApp.MainWindow.GetSampleImgMemStream()
▼	38	▶ Active	17.444	3.625	Async: <GetSampleImage>d_a	SampleApp.MainWindow.GetSampleImage



Подобряване на производителността при паралелизация

Въведено е ново състояние - "suspended" при което се освобождават ползваните критични ресурси за достъп от други (CPU, памет и т.н.), като е подсигурана възможност за бърз повторен достъп при възстановяване.

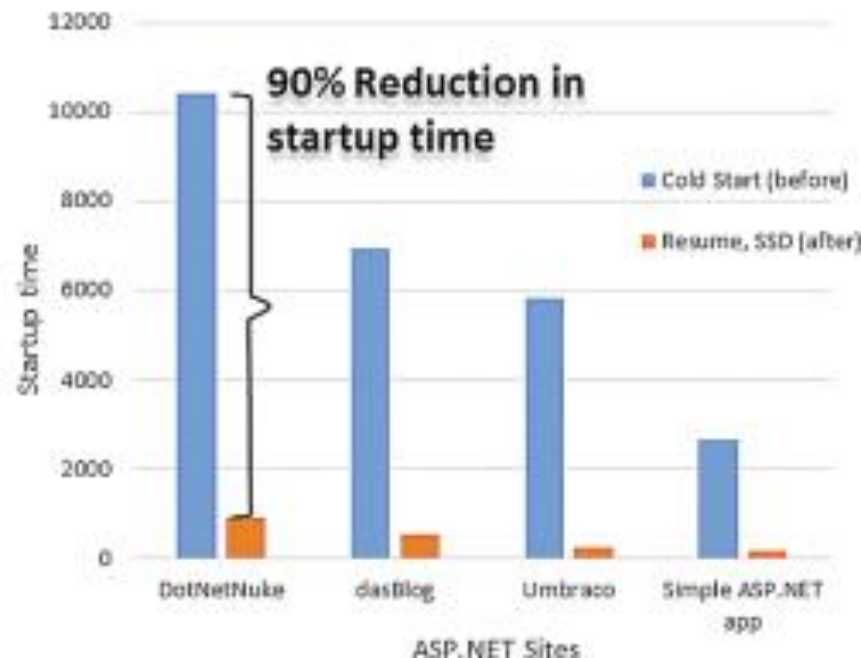
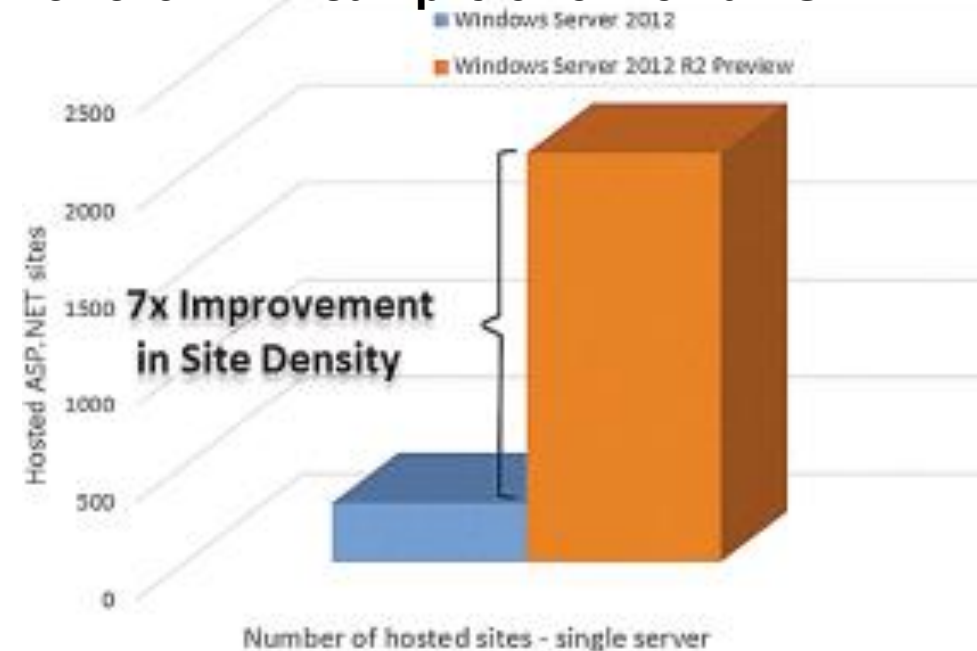
Фигурата показва преходите между състояния. Сайт стартира в inactive състояние. Той се зарежда в паметта и преминава в активно състояние в момента на заявка към началната му страница. След като в бил известно време - idle, сайтът преминава в състояние - suspended, като остава в т.нар. application pool . При повторна заявка, той се прехвърля ускорено в активно състояние. Цикълът продължава докато се достигне в terminated или неактивно състояние след по-значим период на idle.



Горните преходи се управляват от **Windows Server** и не изискват добавен код. Единствено при конфигурация на **IIS** за **application pool** следва да се укаже състояние - "Suspend".



Пример : производителност след описаните преходи и нови състояния за приложение на ASP.NET



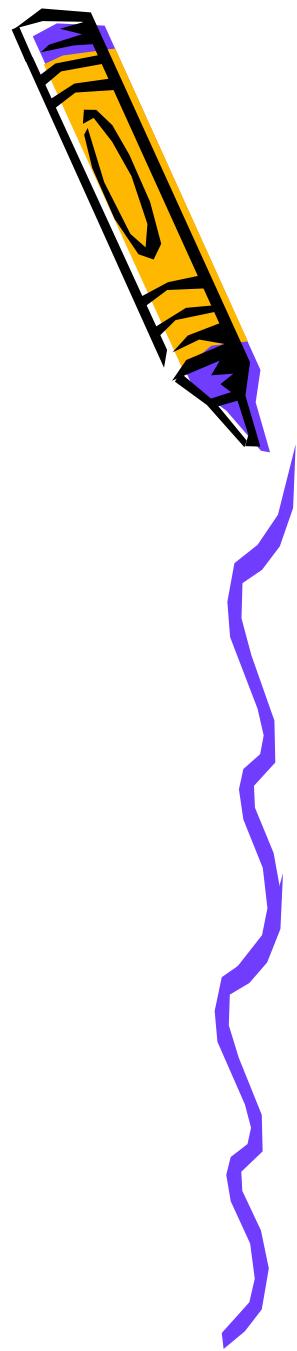
Подобрения в процеса на многоядрена JIT компилация:

Компилация за много ядра вече е възможна в JIT компилатора на ASP.NET. Измерванията на производителност показват, както се вижда, до 40% ускорение при първо запускане на многоядрения код (активиране на JIT компилацията). Самата JIT компилация също се извършва едновременно в наличните ядра, паралелно на изпълнението.

Също така в паралелен режим, динамично се зареждат нужните за ASP.NET асемблита.



T.hap. Asynchronous Pattern



The Task-based Asynchronous Pattern

Асинхронно програмиране с **Async** и **Await** (C# и Visual Basic)



Т.нар **Task-based Asynchronous Pattern (TAP)** е новата парадигма за асинхронно програмиране в .NET Framework. Основни елементи за опериране в нея са типовете: **Task** и **Task<TResult>**.

Това са и елементите, предмет на асинхронни операции.

Целта на **асинхронното програмиране** е: избягване на блокировки и задръствания, както и подобрене във времето на реакция на приложението.

Visual Studio след версия 2012 въвежда подхода с **асинхронното програмиране**.

Цялата работа е оставена на компилатора, като се постига ефектът на асинхронното програмиране, с познания като за писане на синхронен код.





Предимствата на асинхронния код са особено при дейности с чести, потенциални блокировки – каквито са операциите в web. Ако подобни действия се блокират в синхронния процес – цялото приложение замръзва. При асинхронна работа, приложението продължава с други фрагменти, независещи от web ресурса, до момента в който блокиращата задача приключи.

Таблицата по-долу, показва типични области, където асинхронното програмиране носи облаги. Показани са и API на .NET Framework 4.5 и на Windows Runtime, съдържащи методи с поддръжка на асинхронни процеси.

Приложна област

Web достъп

Работа с файлове

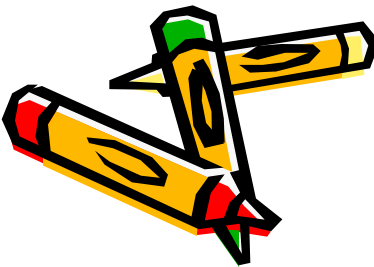
Работа с изображения

Поддръжка на async методи

HttpClient

StorageFile, StreamWriter,
StreamReader, XmlReader

MediaCapture, BitmapEncoder,
BitmapDecoder





Служебните думи : Async и Await на Visual Basic, както и async и await в C# са в основата на асинхронното програмиране.

Ползвайки ги, имате достъп до ресурсите на .NET Framework или на Windows Runtime за създаване на асинхронен метод.



Декларации Async и Await



```
public async Task DoSomethingAsync()  
{  
    // For this example, we're just going to (asynchronously) wait 100ms.  
    await Task.Delay(100);  
}
```

Служебната дума **"async"** разрешава обработката на **"await"** в същия метод (виж по-горе). Това е и всичко, което *async прави!*

Тя не структурира метода например в thread pool като нишка, нито пък извършва други действия или добавки в кода.

Отнесена към метод или лямбда-израз, предизвиква изпълнението им (разбира се, след повикване) **в рамките на същата нишка**, в която е викащия метод



Служебната дума **async единствено разрешава await**. До достигане на **"await"**, „async“ метод си се изпълнява точно както и всеки друг метод. Т.е. – синхронно.

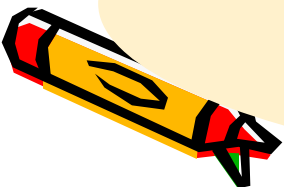
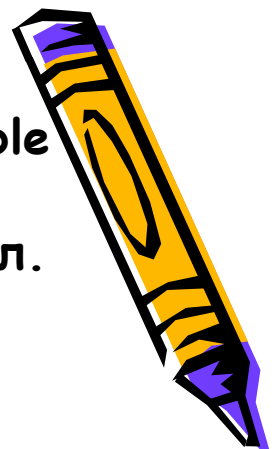
Едва при достигане на **"await"**, нещата стават асинхронни. **Await** е като унарен оператор: взема един аргумент - **awaitable** ("**awaitable**" е самата операция, изпълнявана по този начин). **Await** проверява този „**awaitable**" блок дали вече е привършил. **Ако **awaitable** е вече приключил - методът си продължава синхронното изпълнение.**

Ако **"await" проверката установи, че блокът - **awaitable** не е приключил, той се стартира в асинхронен режим.**
Когато **awaitable** приключи, то ще е изпълнен асинхронния блок.

След **await** блока, остатъкът на **async** метод ще продължи да се изпълнява в своя си **"context"**.

Запомнете: **Добре е да си мислите за **"await"** като за :
"асинхронен wait".**

Т.е. методът, описан като **async временно спира, докато **awaitable** блокът приключи (т.е. той чака), Но самата нишка не е блокирана (т.е. тя си е асинхронна) и може да извършва други задачи.**



Частта „Awaitable“

Споменахме, че “await” поема 1 аргумент – т.нар “awaitable”. Това е асинхронно изпълнявания блок.

Допустими за .NET са 2 **awaitable** типа:


Task<T> и **Task**

Важно за awaitable блока е:

Типът е awaitable, не метода, който го изработва и връща!

С други думи, **await** се отнася до резултата, върнат от async метод (например Task). **Await се отнася до Task!**

... защото методът връща *Task*, а не защото той е синхронен или асинхронен.

По същия начин можете да чакате с await резултата на не-асинхронен метод, връщащ Task: 



```
public async Task NewStuffAsync()  
{
```

```
    // Use await and have fun with the new stuff.
```

```
    await ...  
}
```

```
public Task MyOldTaskParallelLibraryCode()  
{
```

```
    // не сме в async метод - не можем да използваме в него await
```

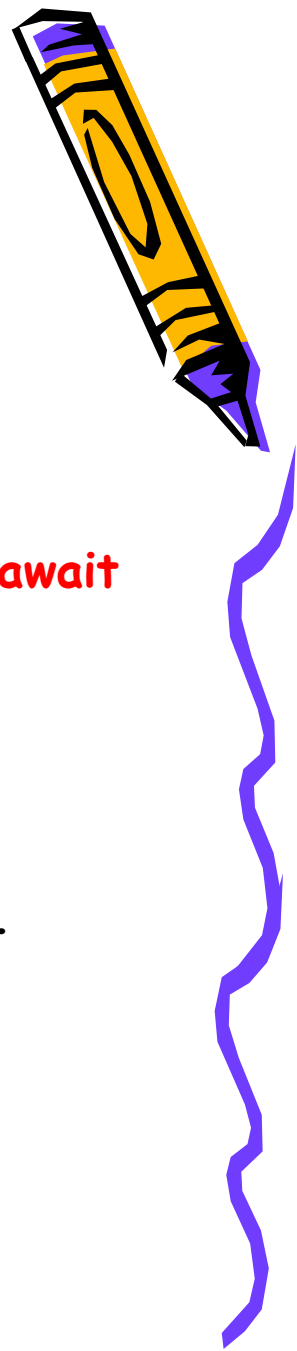
```
    ...  
}
```

```
public async Task ComposeAsync()  
{
```

```
    // We can await Tasks, regardless of where they come from.
```

```
    await NewStuffAsync();
```

```
    await MyOldTaskParallelLibraryCode();  
}
```



Return типове

Async методи връщат `Task<T>`, `Task` или `void`.

На практика, почти винаги връщат `Task<T>` или `Task`.

защо `Task<T>` или `Task`?

Защото те могат да са `awaitable`, докато `void` не. В един `async` метод връщащ `Task<T>` или `Task`, има какво да подадете на `await`.

Във `void` метод, няма какво да подадете на `await`.

`void` за върнат резултат се ползва само при дефиниране на `async event handlers`.

Всъщност, `async` метод, връщащ `Task` или `void` не връща нищо.

Async метод, връщащ `Task<T>` връща стойност от тип `T`:

```
public async Task<int> CalculateAnswer()
{
    await Task.Delay(100); // (Probably should be longer...)
    // връща типа "int", а не "Task<int>"
    return 42;
}
```



Има известно разминаване: декларираме връщана стойност от един тип, а реално връщаме стойност от друг тип:

```
public async Task<int> GetValue()
{
    await TaskEx.Delay(100);
    return 13;    // връщаният тип е "int", не "Task<int>"
}
```

Тогава, защо не напишем направо:

```
public async int GetValue()
{
    await TaskEx.Delay(100);
    return 13;    // връщаният тип е "int"
}
```

Но няма ли да има затруднения от страна на викащия?

1. Как ще се опише сигнатурата му- нали винаги ще е различна?

И как ще се структурира средата, чакаща най-различна стойност?

2. Нали задачите са предмета на превключване.

Затова, Async методи, връщащи стойност (<Tresult>), следва да са с имат описан тип - **Task<TResult>**.



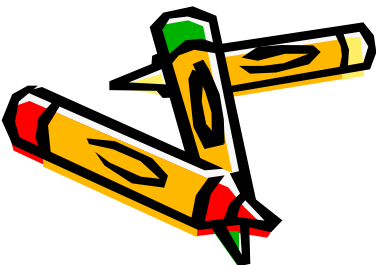


Да изясним разликата между **async void** и **async Task**.

async Task методът описва нормална асинхронна операция, която не връща стойност. Той може да се влага в други описани с `async/ await` методи.

async void методът служи за descriptor от "високо ниво" на асинхронна операция. Така например, той служи за създаване на event handlers.

Async void методите имат и друго приложение: в приложения с ASP.NET например, те служат и за информироване на web server, че страницата не е приключила, до достигането на return



Context

както споменахме вече, когато опишете с `await` `awaitable` от вграден тип, то `awaitable` ще превключи обхващащия "context" и ще продължи да го използва после.

Какво включва този "context"?

Простият отговор е:

- Ако асинхронността е в UI нишка: то това е нейния -UI context.
- Ако асинхронността е в обработката на ASP.NET заявка - става дума за контекста на ASP.NET заявката.
- В останалите случаи - става дума за thread pool context.

По-професионален отговор:

- Ако `SynchronizationContext.Current` не е null, то това е текущия `SynchronizationContext`.
- В противен случай, става дума за `TaskScheduler` контекст (`TaskScheduler.Default` е thread pool context).





// пример с WinForms

```
private async void DownloadFileButton_Click(object sender, EventArgs e)
{
    // тъй като чакаме asynchronously, UI нишката не е блокирана от file download
    await DownloadFileAsync(fileNameTextBox.Text);

    // тъй като възстановяваме в UI context, имаме достъп до UI elements.
    resultTextBox.Text = "File downloaded!";
}
```

// пример с ASP.NET

```
protected async void MyButton_Click(object sender, EventArgs e)
{
    // asynchronously чакаме. ASP.NET нишката не е блокирана от file download.
    // това позволява нишка да обработва и други заявки, докато чака
    await DownloadFileAsync(...);

    // възстановяваме в контекста на ASP.NET. Т.е имаме достъп до текущата заявка
    // може в асинхронната част и да сме били в друга нишка, но вече сме отново
    // в същия контекст - на ASP.NET заявката си

    Response.Write("File downloaded!");
}
```





Avoiding Context

В някои случаи не е добра идея да се синхронизираме все към викация (например "main") context.

Повечето async методи се проектират така че да позволяват превключване в други: техният await включва други операции, а всяка може да представлява от своя страна asynchronous operation (влагане).

Ако прецените, можете да забраните на страната - awaiter да подава своя текущ контекст. Това става с :

ConfigureAwait(false);



```
private async Task DownloadFileAsync(string fileName)
{
    // правим HttpClient за download на съдържание на файл
    var fileContents = await DownloadFileContentsAsync(fileName).ConfigureAwait(false);

    // с ConfigureAwait(false), вече не сме в началния си context.
    // работим в подразбиращия се - този на thread pool.

    // записваме file contents на диск
    await WriteToDiskAsync(fileName, fileContents).ConfigureAwait(false);

    // повторно ConfigureAwait(false) не е задължително - но е „добра практика“
}
```

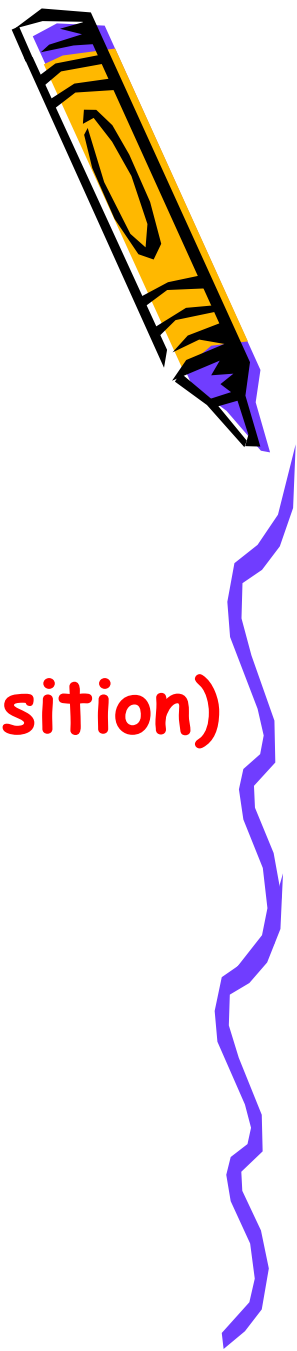
// нека обхванем в WinForms

```
private async void DownloadFileButton_Click(object sender, EventArgs e)
{
    // с asynchronously wait, нишката - UI не е блокирана за времето на file download
    await DownloadFileAsync(fileNameTextBox.Text);
    // възстановяваме се в UI context и имаме достъп до UI elements
    resultTextBox.Text = "File downloaded!";
}
```



DownloadFileButton_Click стартира в UI context и вика DownloadFileAsync. DownloadFileAsync също стартира в UI context, но го напуска с ConfigureAwait(false). За остатъка от DownloadFileAsync се работи в thread pool context. Все пак, когато DownloadFileAsync приключи и DownloadFileButton_Click() се възстанови - това е в UI context.





Структуры с асинхронност (async Composition)



Допустимо е стартиране на много асинхронни операции и `await` докато една / всички да привършат.

Това става така:

```
public async Task DoOperationsConcurrentlyAsync()
{ Task[] tasks = new Task[3];
  tasks[0] = DoOperation0Async();
  tasks[1] = DoOperation1Async();
  tasks[2] = DoOperation2Async();
  // и трите tasks работят.
  // пускаме await на всички - чакащ всички да завършат
  await Task.WhenAll(tasks);
}
```

```
public async Task<int> GetFirstToRespondAsync()
{
  // стартираме 2 web services. Чакаме първата от тях да отговори с резултат
  Task<int>[] tasks = new[] { WebService1Async(), WebService2Async() };

  // Await - структуриран да се чака до първи отговор
  Task<int> firstTask = await Task.WhenAny(tasks);

  // връщаме резултата
  return await firstTask;
}
```

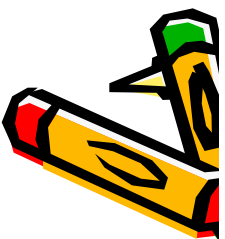
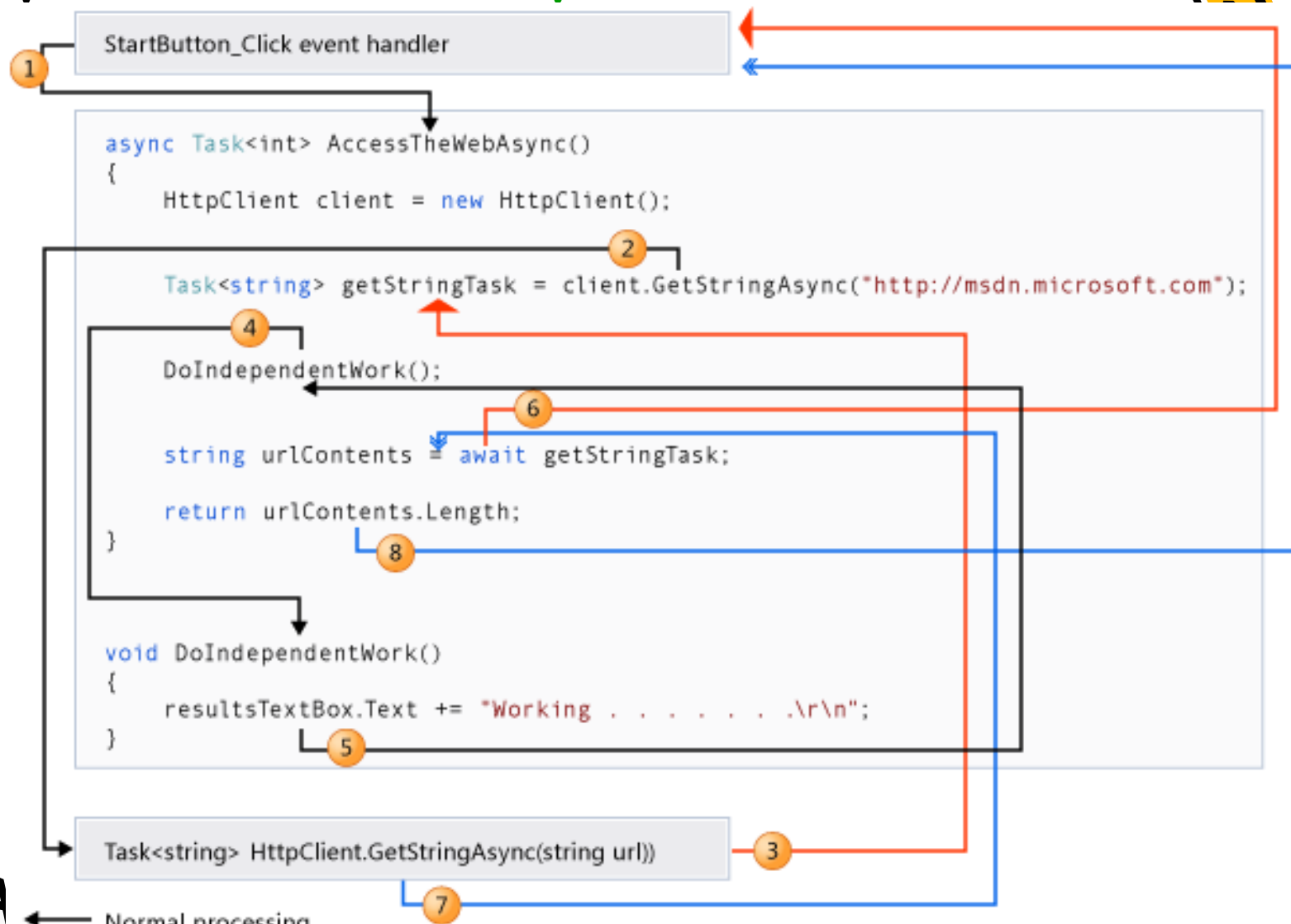




Съответствия на конструктори с блокировки и такива с await техники:

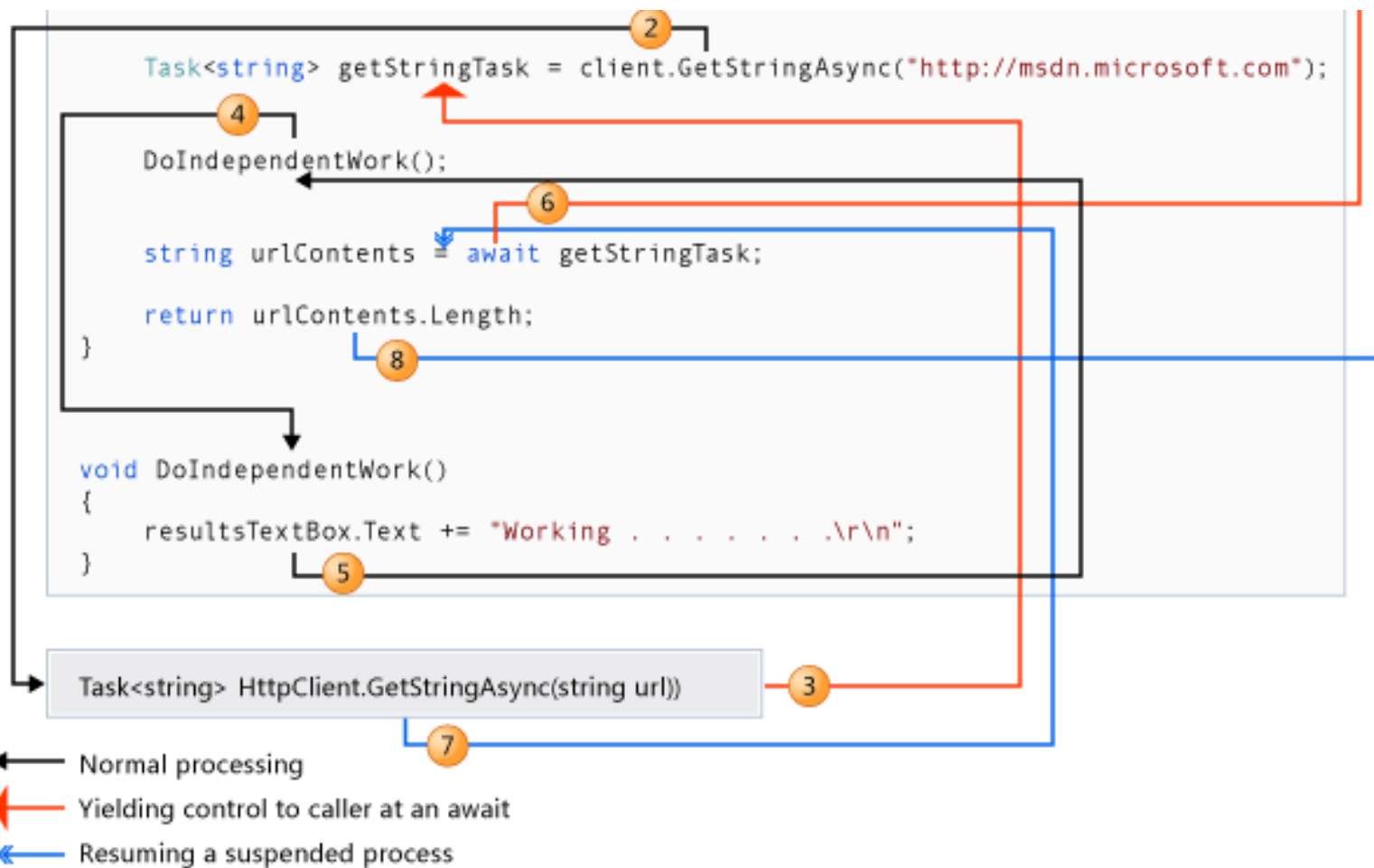
Стари	Нови	Описание
<code>task.Wait</code>	<code>await task</code>	Wait/await задача да привърши
<code>task.Result</code>	<code>await task</code>	Get на резултата от привършилия task
<code>Task.WaitAny</code>	<code>await Task.WhenAny</code>	Wait/await на поне 1 от набора tasks, който е привършил
<code>Task.WaitAll</code>	<code>await Task.WhenAll</code>	Wait/await на всички tasks да привършат
<code>Thread.Sleep</code>	<code>await Task.Delay</code>	Wait/await за времеви период
<code>Task constructor</code>	<code>Task.Run</code> or <code>TaskFactory.StartNew</code>	Стартира task

Пример: използване на **async** метод



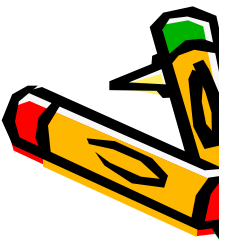
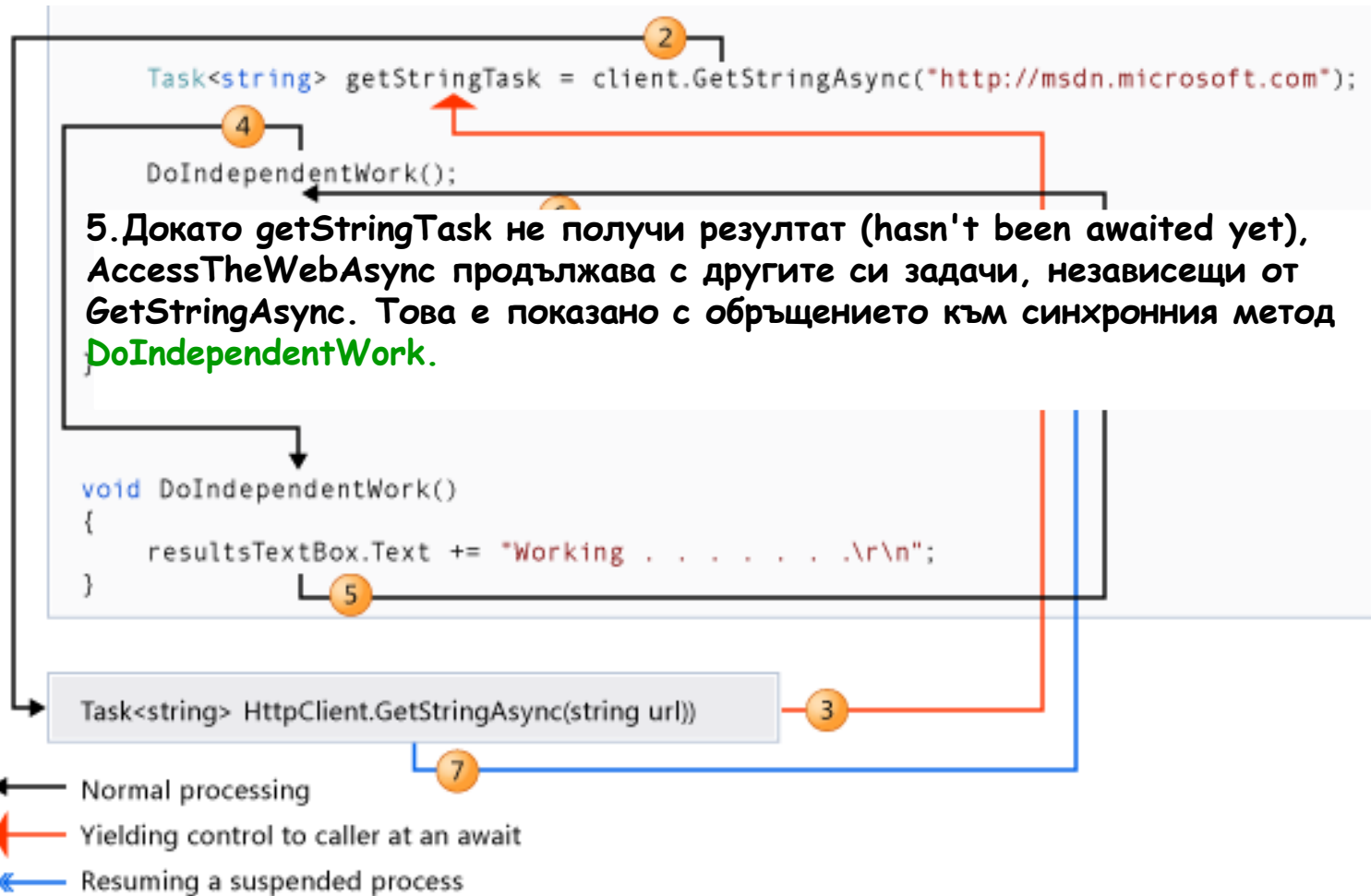


3. Нещо се случва в **GetStringAsync** и той минава в **suspended**. Може да чака нещо от **website** или друго блокирало го действие. За да не се блокират ресурсите, **GetStringAsync()** връща управлението на викащата страна - **AccessTheWebAsync()**



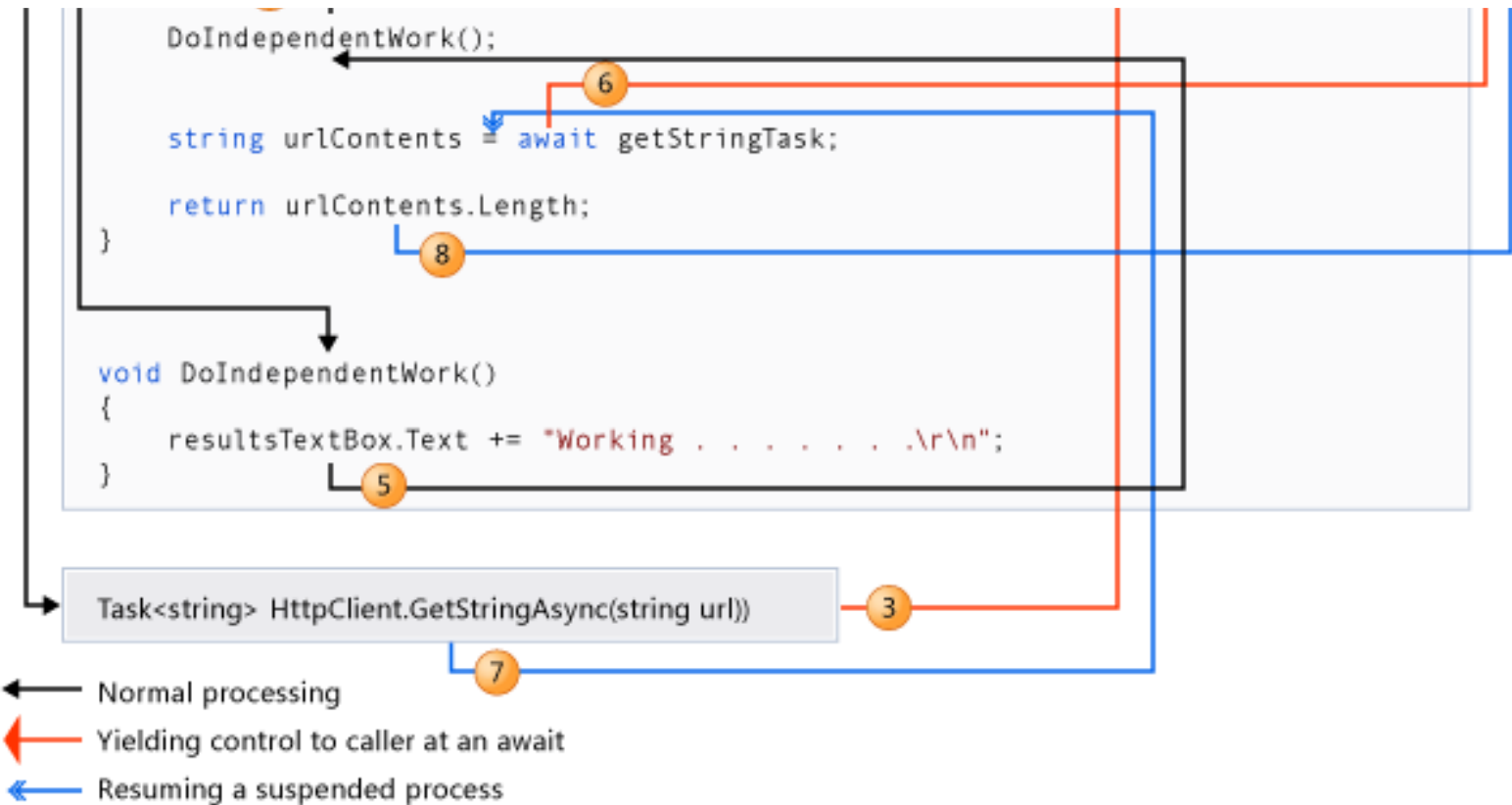


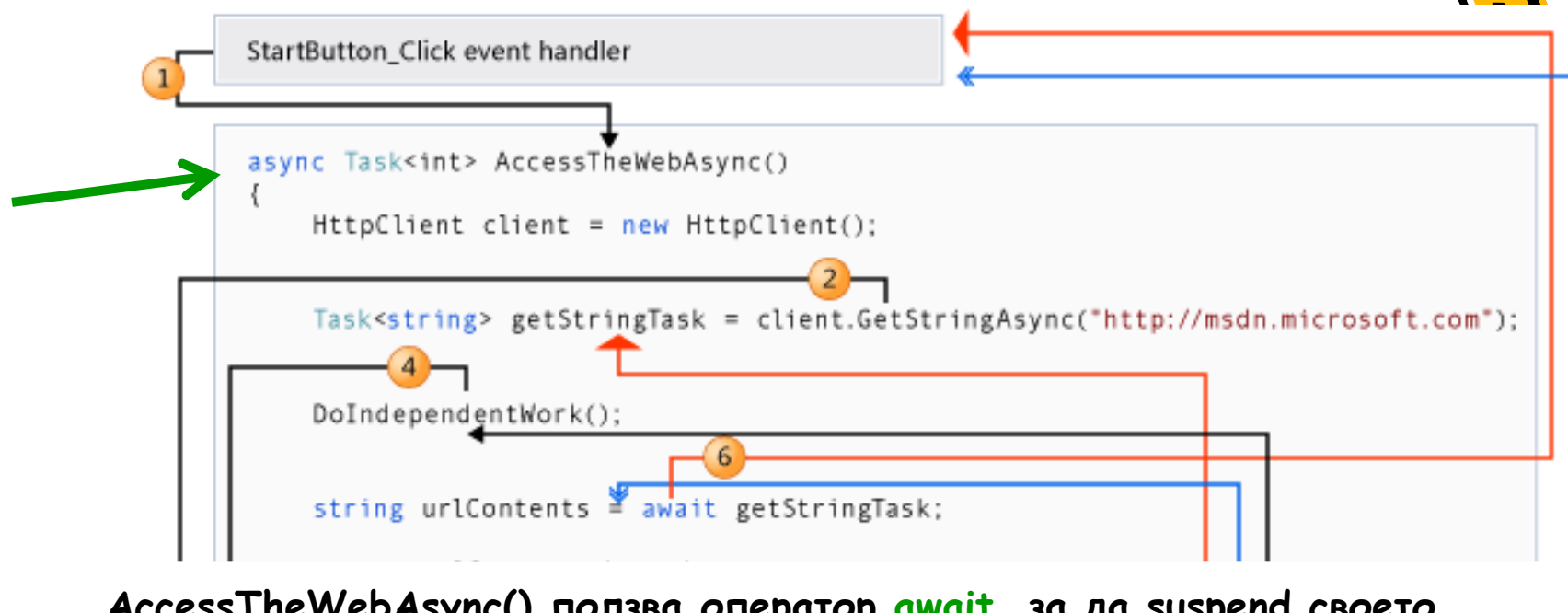
4. `GetStringAsync` връща `Task<TResult>` като `TResult` е низ и `AccessTheWebAsync` присвоява този task на променливата `getStringTask`.





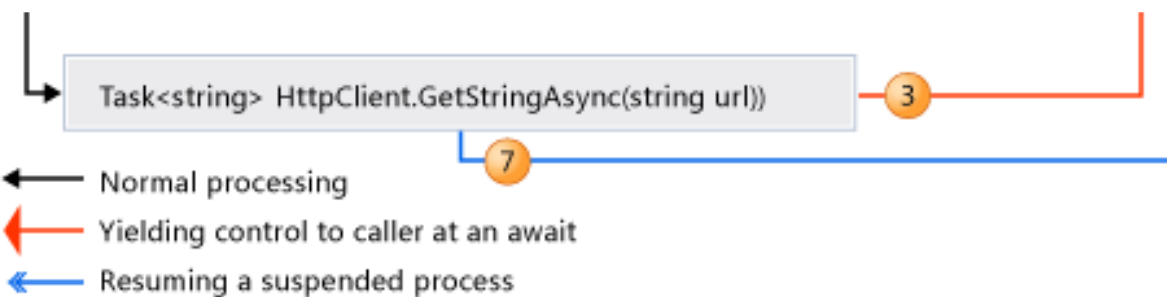
6. `DoIndependentWork` е синхронен метод, връщащ резултат към викация го. `AccessTheWebAsync()` приключи с дейностите, които са възможни без да е получен резултата в `getStringTask`. `AccessTheWebAsync()` ще обработва тази стойност, но не и преди тя да е получена.





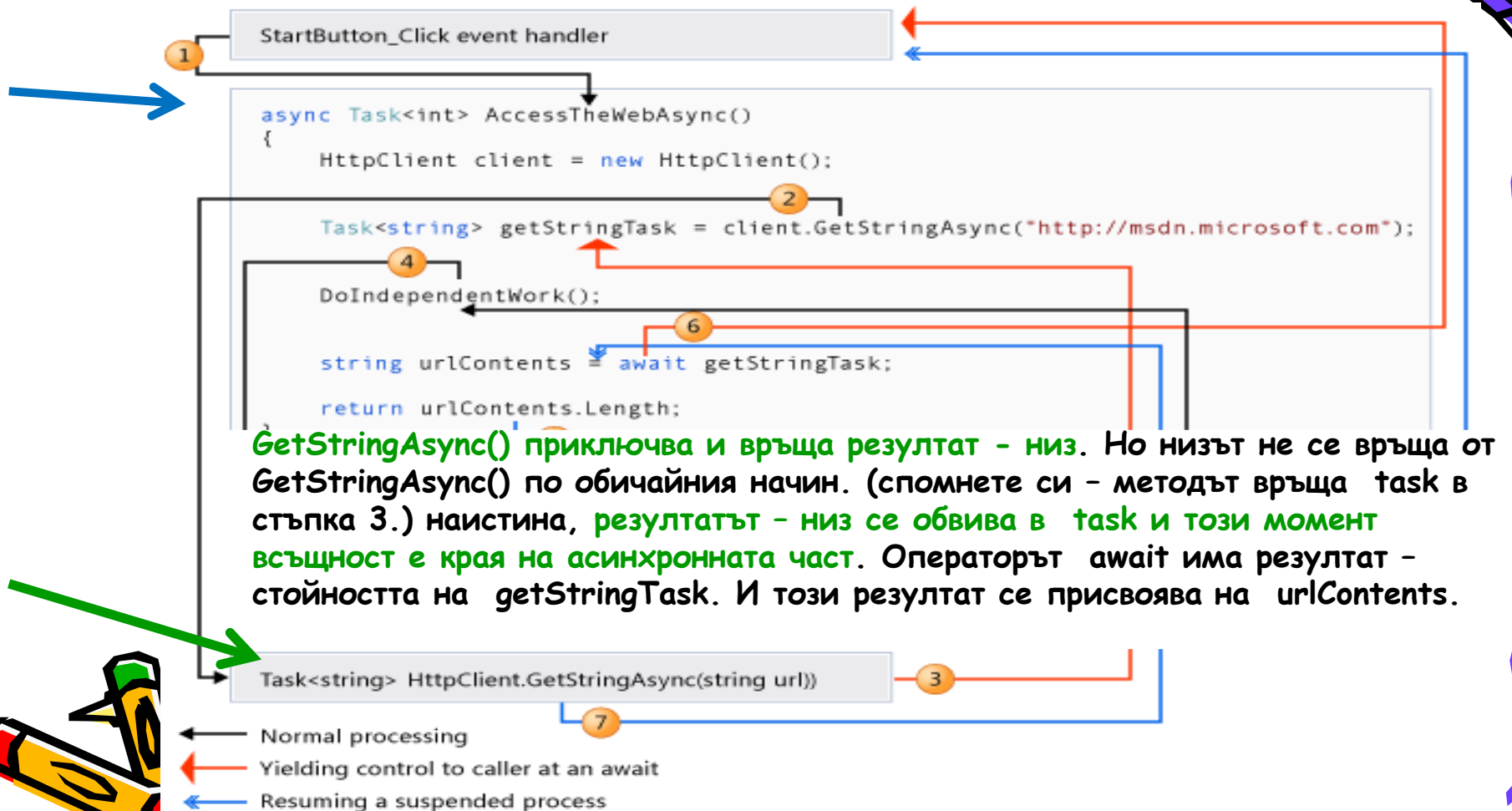
AccessTheWebAsync() ползва оператор **await** за да suspend своето изпълнение, като връща изчисленията към викащия го.

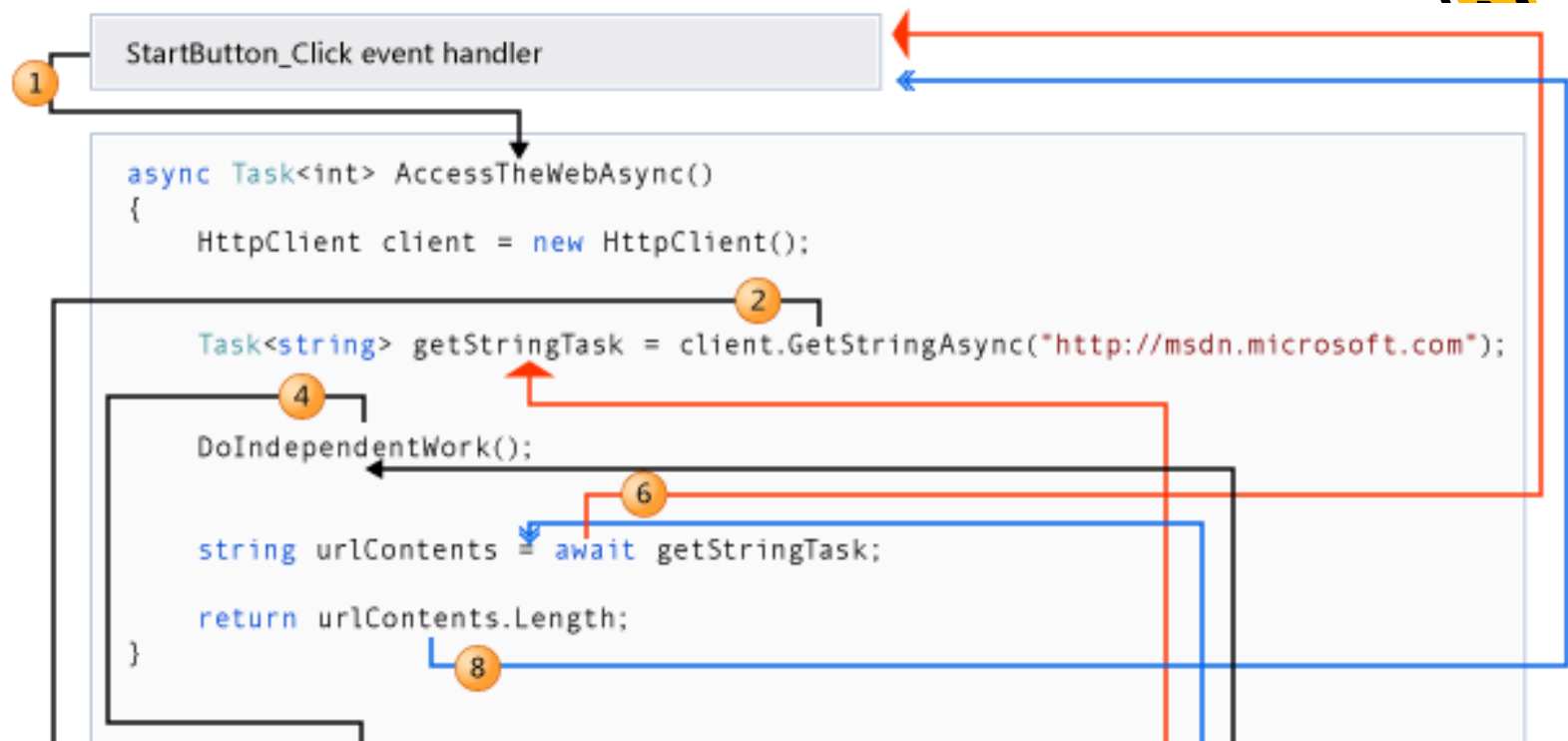
AccessTheWebAsync() връща **Task(Of Integer)** или **Task<int>** към викащия. Този task всъщност е „promise“ за създаване на integer резултат, съдържащ дължината на downloaded низа.



Във викащата ф-ия (в примера - event handler), работата продължава. Там може да се изпълняват други задачи, независещи от резултата върнат от `AccessTheWebAsync()` и докато той се изчаква (`awaiting`).

Event handler изчаква `AccessTheWebAsync()` а `AccessTheWebAsync()` изчаква за `GetStringAsync()`.





Щом `AccessTheWebAsync()` има резултат - низа, методът може да изчисли вече дължината му. Тогава и работата на целия `AccessTheWebAsync()` е приключила и зависалия event handler може да продължи. Event handler приключва, връща и принтва стойността на променливата - дължина на низа.

- ← Normal processing
- ← Yielding control to caller at an await
- ← Resuming a suspended process



Ако сега започвате с асинхронното програмиране, осмислете различията между синхронно и асинхронно поведение.

Синхронен метод завършва, когато изпълни операторите си и върне резултат,

докато

Асинхронен (async) метод формира задача (task) за връщаната си стойност в момента, когато минава в състояние - *suspended* (стъпки 3 и 6 в примера).

Когато *async* метод завърши, task се маркира като приключил и резултатът (ако има такъв), се обвива в task.

